Linköping Studies in Science and Technology

Dissertation No. 1290

Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems

by

Viacheslav Izosimov

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2009

ISBN 978-91-7393-482-4 ISSN 0345-7524
PRINTED IN LINKÖPING, SWEDEN
BY LINKÖPINGS UNIVERSITET

To Evgenia

Abstract

SAFETY-CRITICAL APPLICATIONS have to function correctly and deliver high level of quality-of-service even in the presence of faults. This thesis deals with techniques for tolerating effects of transient and intermittent faults. Re-execution, software replication, and rollback recovery with checkpointing are used to provide the required level of fault tolerance at the software level. Hardening is used to increase the reliability of hardware components. These techniques are considered in the context of distributed real-time systems with static and quasi-static scheduling.

Many safety-critical applications have also strict time and cost constrains, which means that not only faults have to be tolerated but also the constraints should be satisfied. Hence, efficient system design approaches with careful consideration of fault tolerance are required. This thesis proposes several design optimization strategies and scheduling techniques that take fault tolerance into account. The design optimization tasks addressed include, among others, process mapping, fault tolerance policy assignment, checkpoint distribution, and trading-off between hardware hardening and software re-execution. Particular optimization approaches are also proposed to consider debugability requirements of fault-tolerant applications. Finally, quality-of-service aspects have been addressed in the thesis for fault-tolerant embedded systems with soft and hard timing constraints.

The proposed scheduling and design optimization strategies have been thoroughly evaluated with extensive experiments. The experimental results show that considering fault tolerance during system-level design optimization is essential when designing cost-effective and high-quality fault-tolerant embedded systems.

Acknowledgements

I WOULD LIKE to thank my advisors Prof. Zebo Peng, Prof. Petru Eles, and Dr. Paul Pop for guiding me through years of graduate studies and for their valuable comments on this thesis. Despite having four often contradictory points of view, after long discussions, we could always find a common agreement.

Special thanks to Dr. Ilia Polian from the University of Freiburg for his good sense of humour and productive collaboration resulted in the hardware hardening part of this thesis.

Many thanks to the CUGS graduate school for supporting my research and providing excellent courses, and to the ARTES++ graduate school for supporting my travelling.

I would also like to express many thanks to my current and former colleagues at ESLAB and IDA for creating nice and friendly working environment. I will never forget our julbords and fikas.

I am also grateful to my family and friends who have supported me during work on this thesis. I would like to exceptionally thank my parents, Victor Izosimov and Galina Lvova, who have been encouraging me during many long years of my studies. Finally, I devote this thesis to my beloved wife, Yevgeniya Kyselova, for her love, patience, and constant support.

Linköping, November 2009

Viacheslav Izosimov

Contents

List of Figures

List of Tables

PART I
Preliminaries

Chapter 1
Introduction

THIS THESIS DEALS with the analysis and optimization of safety-critical real-time applications implemented on fault-tolerant distributed embedded systems. Such systems are responsible for critical control functions in aircraft, automobiles, robots, telecommunication and medical equipment. Therefore, they have to function correctly and meet timing constraints even in the presence of faults.

Faults in distributed embedded systems can be permanent, intermittent or transient. Permanent faults cause long-term malfunctioning of components, while transient and intermittent faults appear for a short time. The effects of transient and intermittent faults, even though they appear for a short time, can be devastating. They may corrupt data or lead to logic miscalculations, which can result in a fatal failure or dramatic quality-of-service deterioration. Transient and intermittent faults appear at a rate much higher than the rate of permanent faults and, thus, are very common in modern electronic systems.

Transient and intermittent faults can be addressed in *hardware* with hardening techniques, i.e., improving the hardware technology and architecture to reduce the fault rate, or in *soft-*

ware. We consider *hardware-based* hardening techniques and several *software-based* fault tolerance techniques, including re-execution, software replication, and rollback recovery with checkpointing.

Safety-critical real-time applications have to be implemented such that they satisfy strict timing requirements and tolerate faults without exceeding a given amount of resources. Moreover, not only timeliness, reliability and cost-related requirements have to be considered but also other issues such as debugability and testability have to be taken into account.

In this introductory chapter, we motivate the importance of considering transient and intermittent faults during the design optimization of embedded systems. We introduce the design optimization problems addressed and present the main contributions of our work. We also present an overview of the thesis with short descriptions of the chapters.

1.1 Motivation

In this section we discuss the main sources of transient and intermittent faults and how to consider such faults during design optimization.

1.1.1 TRANSIENT AND INTERMITTENT FAULTS

There are several reasons why the rate of transient and intermittent faults is increasing in modern electronic systems, including high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [Mah04, Con03, Har01].

The first type of faults, *transient faults*, cause components to malfunction for a short time, leading to corruption of memory or miscalculations in logic, and then disappear [Sto96, Kor07]. A good example of a transient fault is the fault caused by solar radiation or electromagnetic interference. The rate of transient

faults is often much higher compared to the rate of permanent faults. Transient-to-permanent fault ratios can vary between 2:1 and 50:1 [Sos94], and more recently 100:1 or higher [Kop04]. Automobiles, for example, are largely affected by transient faults [Cor04a, Han02] and proper fault tolerance techniques against transient faults are needed.

Another type of faults, which we consider, are *intermittent faults*. Although an intermittent fault manifests itself similar to a transient fault, i.e., appears for a short time and then disappears, this fault will *re-appear* at some later time [Sto96, Kor07]. For example, intermittent faults can be triggered by one improperly placed device affecting other components through a radio emission or via a power supply. One such component can also create several intermittent faults at the same time.

It is observed that already now more than 50% of automotive electronic components returned to the vendor have no physical defects, and the malfunctioning is the result of intermittent and transient faults produced by other components [Kim99].

Causes of transient and intermittent faults can vary a lot. There exist several possible causes of these faults, including:

- *(solar) radiation* (mostly *neutrons*) that can affect electronic systems not only on the Earth orbit and in space but also on the ground [Sri96, Nor96, Tan96, Ros05, Bau01, Vel07];
- *electromagnetic interference* by mobile phones, wireless communication equipment [Str06], power lines, and radars [Han02];
- *lightning storms* that can affect power supply, current lines, or directly electronic components [Hei05];
- *internal electromagnetic interference* [Wan03];
- *crosstalk* between two or more internal wires [Met98];
- *ion particles in the silicon* that are generated by radioactive elements naturally present in the silicon [May78];
- *temperature variations* [Wei04];
- *power supply fluctuations* due to influence of internal components [Jun04]; and

- *loose connectors* [Pet06], for example, between a network cable and the distributed components attached to it.

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

1.1.2 FAULT TOLERANCE AND DESIGN OPTIMIZATION

Safety-critical applications have strict time and cost constraints and have to deliver a high level of quality of service, which means that not only faults have be to tolerated but also the imposed constraints have to be satisfied.

Traditionally, hardware replication was used as a fault tolerance technique against transient faults. For example, in the MARS [Kop90, Kop89] approach each fault-tolerant component is composed of three computation units, two main units and one shadow unit. Once a transient fault is detected, the faulty component must restart while the system is operating with the non-faulty component. This architecture can tolerate one permanent fault and one transient fault at a time, or two transient faults. Another example is the XBW [Cla98] architecture, where hardware duplication is combined with double process execution. Four process replicas are run in total. Such an architecture can tolerate either two transient faults or one transient fault with one permanent fault. Interesting implementations can be also found in avionics. For example, an airborne architecture, which contains seven hardware replicas that can tolerate up to three transient faults, has been studied in [Als01] based on the flight control system of the JAS 39 Gripen aircraft. However, this solu-

tion is very costly and can be used only if the amount of resources is virtually unlimited. In other words, existing architectures are either too costly or are unable to tolerate multiple transient faults.

In order to reduce cost, other techniques are required such as software replication [Xie04, Che99], recovery with checkpointing [Jie96, Pun97, Bar08, Yin03, Yin06, Aya08, Kri93], and re-execution [Kan03a]. However, if applied in a straightforward manner to an existing design, software-based techniques against transient faults introduce significant time overheads, which can lead to unschedulable solutions. Time overhead can be reduced with hardening techniques [Gar06, Hay07, Moh03, Zha06, Zho06, Zho08], which reduce the transient fault rate and, hence, the number of faults propagated to the software level. On the other hand, using more reliable and/or faster components, or a larger number of resources, may not be affordable due to cost constraints. Therefore, efficient design optimization techniques are required, in order to meet time and cost constraints within the given resources, in the context of fault-tolerant systems.

Transient faults are also common for communication channels, even though we do not deal with them in this thesis. We assume that transient faults on the bus are addressed at the communication level, for example, with the use of efficient error correction codes [Pir06, Bal06, Ema07], through hardware replication of the bus [Kop03, Sil07], and/or acknowledgements/ retransmissions [Jon08]. Solutions such as a cyclic redundancy code (CRC) are implemented in communication protocols available on the market [Kop93, Kop03, Fle04].

1.2 Contributions

In our approach, an embedded application is represented as a set of soft and hard real-time processes [But99] communicating by sending messages. Hard processes represent time-constrained parts of the application, which must be always executed and meet deadlines. A soft process can complete after its deadline and its completion time is associated with a value function that characterizes its contribution to the quality-of-service of the application.

Hard and soft processes are *mapped* on computation nodes connected to a communication infrastructure. Processes and communication schedules are determined off-line by *quasi-static scheduling* that generates a tree of fault-tolerant schedules that maximize the quality-of-service value of the application and, at the same time, guarantees deadlines for hard processes. At run time, an online runtime scheduler with very low online overhead would select the appropriate schedule based on the occurrence of faults and the actual execution times of processes. Our design optimization considers the impact of communications on the overall system performance.

Reliability of computation nodes can be increased with hardening in order to reduce the number of transients faults propagating to the software level. To provide resiliency against transient faults propagated to the software, various fault tolerance techniques can be applied to the application processes, such as re-execution, replication, or recovery with checkpointing. Design optimization algorithms consider the various overheads introduced by the different techniques and determine which are to be applied for each process. In addition to performance, quality-of-service and cost-related requirements, debugability and testability of embedded systems are also taken into account during design optimization.

The main contributions of this thesis are the following:

- **Scheduling techniques with fault tolerance** [Izo05, Pop09, Izo06b, Izo10b]. In the thesis we propose two scheduling techniques for dealing with transient faults in the context of hard real-time systems. The first technique, shifting-based scheduling, is able to quickly produce efficient schedules of processes and messages, where the order of processes is preserved in the case of faults and communications on the bus are fixed. The second technique, conditional scheduling, creates more efficient schedules than the ones generated with the shifting-based scheduling, by overcoming restrictions of fixed communications on the bus and allowing changing of process order in the case of faults. This approach allows to take into account testability and debugability requirements of safety-critical applications.
- **Value-based scheduling techniques with fault tolerance** [Izo08a, Izo08b, Izo10a]. These scheduling techniques produce a tree of fault-tolerant schedules for embedded systems composed of soft and hard processes, such that the quality-of-service of the application is maximized and all hard deadlines are satisfied.
- **Mapping optimization strategies** [Izo05, Izo06a, Pop09, Izo10b], which produce an efficient mapping of processes and process replicas on the computation nodes.
- **Fault tolerance policy assignment strategies** [Izo05, Izo06c, Pop09] for assigning the appropriate combinations of fault tolerance techniques to processes, such that the faults are tolerated and the deadlines are satisfied within the imposed cost constraints.
- **An approach to optimization of checkpoint distribution in rollback recovery** [Izo06c, Pop09]. We propose an approach to calculate the optimal checkpoint distribution in the context of a single process and an optimization heuristic to determine an appropriate checkpoint distribution for real-time applications composed of many processes.

- **A design approach for trading-off between component hardening level and number of re-executions** [Izo09]. In this approach we combine component hardening with re-executions in software in order to provide a fault-tolerant system that satisfies cost and time constraints and, at the same time, meets the specified reliability goal. The design approach is based on the system failure probability (SFP) analysis that connects the global reliability of the system with the reliability levels of the hardened hardware components and the number of re-executions introduced into software.

1.3 Thesis Overview

Part II and Part IV of the thesis are devoted to various design optimization approaches for *hard real-time applications*, where hard real-time constraints have to be satisfied even in the presence of faults. In Part III, we extend our approach to systems composed of hard and soft real-time processes. In addition to hard real-time constraints being satisfied we also perform a value-based optimization of the overall quality-of-service.

The thesis structure is, thus, as follows:

Part I. Preliminaries:

- **Chapter 2** introduces basic concepts of fault tolerance in software and in hardware in the context of system-level design and optimization algorithms and presents the related work.
- **Chapter 3** presents our hardware architecture and application models with our quality-of-service and fault model. We introduce the notion of transparency and frozenness related to testability and debugability requirements of applications. This chapter also discusses software-level fault tolerance techniques in the context of static cyclic scheduling.

Part II. Hard Real-Time Systems:

- **Chapter 4** presents two scheduling techniques with fault tolerance requirements, including scheduling with transparency/performance trade-offs, in the context of hard real-time systems. These scheduling techniques are used by design optimization strategies presented in the later chapters to derive fault-tolerant schedules.
- **Chapter 5** discusses mapping and policy assignment optimization issues. We propose a mapping and fault tolerance policy assignment strategy that combines software replication with re-execution.
- **Chapter 6** introduces our checkpoint distribution strategies. We also present mapping and policy assignment optimization with checkpointing.

Part III. Mixed Soft and Hard Real-Time Systems:

- **Chapter 7** presents value-based scheduling techniques to produce a tree of fault-tolerant schedules for monoprocessor embedded systems composed of soft and hard processes. The level of quality-of-service must be maximized and hard deadlines must be satisfied even in the worst-case scenarios and in the presence of faults. We suggest an efficient tree-size optimization algorithm to reduce the number of necessary fault-tolerant schedules in the schedule tree.
- **Chapter 8** proposes value-based scheduling techniques for distributed embedded systems composed of soft and hard processes. We use a signalling mechanism to provide synchronization between computation nodes, which increases efficiency of the generated schedules.

Part IV. Embedded Systems with Hardened Components:

- **Chapter 9** proposes a system failure probability (SFP) analysis to determine if the reliability goal of the system is met under a given hardening and re-execution setup. The SFP

analysis is used by our design optimization strategy in Chapter 10.

- **Chapter 10** presents a design strategy to trade-off between hardening and software-level fault tolerance in the context of hard real-time systems. We propose a number of design optimization heuristics to minimize the system hardware cost while satisfying time constraints and reliability requirements.

Part V. Conclusion:

- **Chapter 11**, finally, presents our conclusions and possible directions of future work based on the material presented in this thesis.

Chapter 2
Background and Related Work

THIS CHAPTER presents background and related work in the area of system-level design, including a generic design flow for embedded systems. We also discuss software and hardware-level fault tolerance techniques. Finally, we present relevant research work on design optimization for fault-tolerant embedded systems and suggest a possible design flow with fault tolerance.

2.1 Design and Optimization

System-level design of embedded systems is typically composed of several steps, as illustrated in Figure 2.1. In the "System Specification" step, an abstract system model is developed. In our application model, functional blocks are represented as processes and communication data is encapsulated into messages. Time constraints are imposed in form of deadlines assigned to the whole application, to individual processes or to groups of dependent processes.

The hardware architecture is selected in the "Architecture Selection" step. The architecture for automotive applications that we consider in this thesis consists of a set of computation nodes connected to a bus. The computation nodes are heterogeneous and have different performance characteristics and reliability properties. They also have different costs, depending on their performance, reliability, power consumption and other parameters. Designers should choose an architecture with a good price-to-quality ratio within the imposed cost constraints.

In the "Mapping & Hardware/Software Partitioning" step, mapping of application processes on computation nodes has to be decided such that the performance of the system is maximized and given design constraints are satisfied [Pra94, Pop04b, Pop04c, Pop04a]. These can include memory constraints, power constraints, as well as security- and safety-related constraints. Some processes can be implemented in hardware using ASICs or FPGAs. The decision on whether to implement processes in hardware is taken during hardware/software partitioning of the application [Cho95, Ele97, Ern93, Bol97, Dav99, Axe96, Mir05].

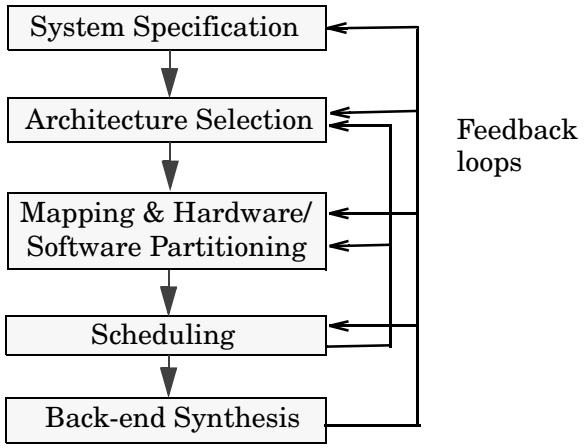


Figure 2.1: Generic Design Flow

After mapping and partitioning, the execution order and start times of processes are considered in the "Scheduling" step. Scheduling can be either static or dynamic. In the case of dynamic scheduling, start times are determined online based on priorities assigned to the processes [Liu73, Tin94, Aud95]. In static cyclic scheduling [Kop97, Jia00], start times of processes and sending times of messages are pre-defined off-line and stored in form of *schedule tables*. Researchers have developed several algorithms to efficiently produce static schedules off-line. Many of these algorithms are based on list scheduling heuristics [Cof72, Deo98, Jor97, Kwo96]. However, off-line static cyclic scheduling lacks flexibility and, unless extended with adaptive functionality, cannot handle overloads or efficiently provide fault recovery [Dim01, Kan03a]. In this thesis we overcome the limitations of static cyclic scheduling by employing *quasi-static scheduling techniques*, which will be used to design fault-tolerant systems and can provide the flexibility needed to efficiently handle soft real-time processes [Cor04b]. Quasi-static scheduling algorithms produce a tree of schedules, between which the scheduler switches at runtime based on the conditions (such as fault occurrences or process finishing times) calculated online, during the runtime of the application.

If, according to the resulted schedule, deadlines are not satisfied or the desired quality-of-service level is not achieved, then either mapping or partitioning should be changed (see the feedback line in Figure 2.1). If no acceptable solution in terms of quality, costs or deadlines can be found by optimizing process mapping and/or the schedule, then the hardware architecture needs to be modified and the optimization will be performed again.

After a desirable implementation has been found, the back-end system synthesis of a prototype will be performed for both hardware and software (shown as the last step in the design flow).

If the prototype does not meet requirements, then either the design or specification will have to be changed. However, redesign of the prototype has to be avoided as much as possible by efficient design optimization in the early design stages, in order to reduce design costs.

2.2 Fault Tolerance Techniques

In this section, we present first several error-detection techniques that can be applied against transient faults. Then, we discuss software-based fault tolerance techniques such as re-execution, rollback recovery with checkpointing, and software replication, and introduce hardening techniques.

2.2.1 ERROR DETECTION TECHNIQUES

In order to achieve fault tolerance, a first requirement is that transient faults have to be detected. Researchers have proposed several error-detection techniques against transient faults, including watchdogs, assertions, signatures, duplication, and memory protection codes.

Signatures. Signatures [Nah02a, Jie92, Mir95, Sci98, Nic04] are among the most powerful error detection techniques. In this technique, a set of logic operations can be assigned with pre-computed "check symbols" (or "checksum") that indicate whether a fault has happened during those logic operations. Signatures can be implemented either in hardware, as a parallel test unit, or in software. Both hardware and software signatures can be systematically applied without knowledge of implementation details.

Watchdogs. In the case of watchdogs [Ben03, Mah88, Mir95], program flow or transmitted data is periodically checked for the presence of faults. The simplest watchdog schema, *watchdog timer*, monitors the execution time of processes, whether it exceeds a certain limit [Mir95]. Another approach is to incorpo-

rate simplified signatures into a watchdog. For example, it is possible to calculate a general "checksum" that indicates correct behaviour of a computation node [Sos94]. Then, the watchdog will periodically test the computation node with that checksum. Watchdogs can be implemented either in hardware as a separate processor [Ben03, Mah88] or in software as a special test program.

Assertions. Assertions [Gol03, Hil00, Pet05] are an application-level error-detection technique, where logical test statements indicate erroneous program behaviour (for example, with an "*if*" statement: *if not* <assertion> *then* <error>). The logical statements can be either directly inserted into the program or can be implemented in an external test mechanism. In contrast to watchdogs, assertions are purely application-specific and require extensive knowledge of the application details. However, assertions are able to provide much higher error coverage than watchdogs.

Duplication. If the results produced by duplicated entities are different, then this indicates the presence of a fault. Examples of duplicated entities are duplicated instructions [Nah02b], functions [Gom06], procedure calls [Nah02c], and whole processes. Duplication is usually applied on top of other error detection techniques to increase error coverage.

Memory protection codes. Memory units, which store program code or data, can be protected with error detection and correction codes (EDACs) [Shi00, Pen95]. An EDAC code separately protects each memory block to avoid propagation of faults. A common schema is "single-error-correcting, double-error-detecting" (SEC-DED) [Pen95] that can correct one fault and detect two faults simultaneously in each protected memory block.

Other error-detection techniques. There are several other error-detections techniques, for example, transistor-level current monitoring [Tsi01] or the widely-used parity-bit check.

Error coverage of error-detection techniques has to be as high as possible. Therefore, several error-detection techniques are

often applied together. For example, hardware signatures can be combined with transistor-level current monitoring, memory protection codes and watchdogs. In addition, the application can contain assertions and duplicated procedure calls.

Error-detection techniques introduce an *error-detection overhead*, which is the time needed for detecting faults. The error-detection overhead can vary a lot with the error-detection technique used. In our work, unless other specified, we account the error-detection overhead in the worst-case execution time of processes.

2.2.2 RE-EXECUTION

In software, after a transient fault is detected, a fault tolerance mechanism has to be invoked to handle this fault. The simplest fault tolerance technique to recover from fault occurrences is re-execution [Kan03a]. With re-execution, a process is executed again if affected by faults.

The time needed for the detection of faults is accounted for by *error-detection overhead*. When a process is re-executed after a fault has been detected, the system restores all initial inputs of that process. The process re-execution operation requires some time for this, which is captured by the *recovery overhead*. In order to be restored, the initial inputs to a process have to be stored before the process is executed for first time. For the sake of simplicity, however, we will ignore this particular overhead, except for the discussion of rollback recovery with checkpointing in Section 2.2.3, Section 3.2, and Chapter 6.[1] The error detection

1. The overhead due to saving process inputs does not influence the design decisions during mapping and policy assignment optimization when re-execution is used. However, we will consider this overhead in rollback recovery with checkpointing as part of the checkpointing overhead during the discussion about checkpoint optimization in Chapter 6.

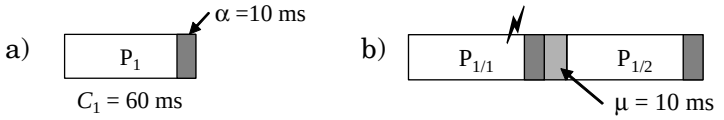


Figure 2.2: Re-execution

and recovery overheads will be denoted throughout this work with α and μ, respectively.

Figure 2.2 shows the re-execution of process P_1 in the presence of a single fault. As illustrated in Figure 2.2a, the process has the worst-case execution time of 60 ms, which includes the error-detection overhead α of 10 ms. In Figure 2.2b process P_1 experiences a fault and is re-executed. We will denote the j-th execution of process P_i as $P_{i/j}$. Accordingly, the first execution of process P_1 is denoted as $P_{1/1}$ and its re-execution $P_{1/2}$. The recovery overhead μ = 10 ms is depicted as a light grey rectangle in Figure 2.2.

2.2.3 ROLLBACK RECOVERY WITH CHECKPOINTING

The time needed for re-execution can be reduced with more complex fault tolerance techniques such as *rollback recovery with checkpointing* [Pun97, Bar08, Yin03, Yin06, Ora94, Aya08, Kri93]. The main principle of this technique is to restore the last non-faulty state of the failing process. The last non-faulty state, or *checkpoint*, has to be saved in advance in the static memory and will be restored if the process fails. The part of the process between two checkpoints or between a checkpoint and the end of the process is called an *execution segment*.[1]

1. Note that re-execution can be considered as rollback recovery with a single checkpoint, where this checkpoint is the initial process state and the execution segment is the whole process.

There are several approaches to distribute checkpoints. One approach is to insert checkpoints in the places where saving of process states is the fastest [Ziv97]. However, this approach is *application-specific* and requires knowledge of application details. Another approach is to *systematically* insert checkpoints, for example, at *equal* intervals [Yin06, Pun97, Kwa01].

An example of rollback recovery with checkpointing is presented in Figure 2.3. We consider processes P_1 with the worst-case execution time of 60 ms and error-detection overhead α of 10 ms, as depicted in Figure 2.3a. In Figure 2.3b, two checkpoints are inserted at equal intervals. The first checkpoint is the initial state of process P_1. The second checkpoint, placed in the middle of process execution, is for storing an intermediate process state. Thus, process P_1 is composed of two execution segments. We will name the k-th execution segment of process P_i as P_i^k. Accordingly, the first execution segment of process P_1 is P_1^1 and its second segment is P_1^2. Saving process states, including saving initial inputs, at checkpoints, takes a certain amount of time that is considered in the *checkpointing overhead* χ, depicted as a black rectangle.

In Figure 2.3c, a fault affects the second execution segment P_1^2 of process P_1. This faulty segment is executed again starting from the second checkpoint. Note that the error-detection overhead α is not considered in the last recovery in the context of rollback recovery with checkpointing because, in this example, we assume that a maximum of one fault can happen.

We will denote the j-th execution of k-th execution segment of process P_i as $P_{i/j}^k$. Accordingly, the first execution of execution

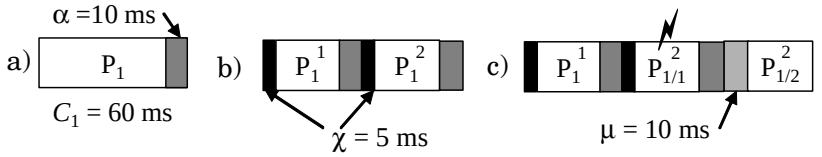


Figure 2.3: Rollback Recovery with Checkpointing

segment P_1^2 has the name $P_{1/1}^2$ and its second execution is named $P_{1/2}^2$. Note that we will not use the index j if we only have one execution of a segment or a process, as, for example, P_1's first execution segment P_1^1 in Figure 2.3c.

When recovering, similar to re-execution, we consider a recovery overhead μ, which includes the time needed to restore checkpoints. In Figure 2.3c, the recovery overhead μ, depicted with a light gray rectangle, is 10 ms for process P_1.

The fact that only a part of a process has to be restarted for tolerating faults, not the whole process, can considerably reduce the time overhead of rollback recovery with checkpointing compared to re-execution.

2.2.4 ACTIVE AND PASSIVE REPLICATION

The disadvantage of rollback recovery techniques, such as re-execution[1] and rollback recovery with checkpointing, is that they are unable to explore spare capacity of available computation nodes and, by this, to possibly reduce the schedule length. If the process experiences a fault, then it has to recover on the same computation node. In contrast to rollback recovery technique, *active and passive replication* techniques can utilize available spare capacity of other computation nodes. Moreover, active replication provides the possibility of *spatial redundancy*, e.g. the ability to execute process replicas in parallel on different computation nodes.

In the case of active replication [Xie04], all replicas of processes are executed independently of fault occurrences. In the case of passive replication, also known as *primary-backup* [Ahn97, Sze05], on the other hand, replicas are executed only if faults occur. In Figure 2.4 we illustrate primary-backup and active replication. We consider process P_1 with the worst-case execution time of 60 ms and error-detection overhead α of 10 ms,

1. Sometimes referred as rollback recovery with a single checkpoint.

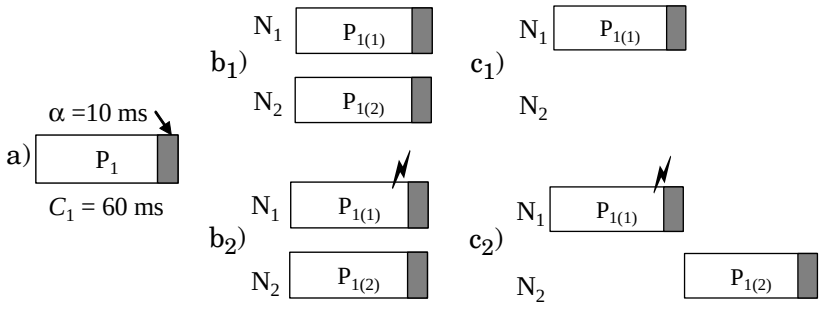


Figure 2.4: Active Replication (b) and Primary-Backup (c)

see Figure 2.4a. Process P_1 will be replicated on two computation nodes N_1 and N_2, which is enough to tolerate a single fault. We will name the j-th replica of process P_i as $P_{i(j)}$. Note that, for the sake of uniformity, we will consider the original process as *the first replica*. Hence, the replica of process P_1 is named $P_{1(2)}$ and process P_1 itself is named as $P_{1(1)}$.

In the case of active replication, illustrated in Figure 2.4b, replicas $P_{1(1)}$ and $P_{1(2)}$ are executed in parallel, which, in this case, improves system performance. However, active replication occupies more resources compared to primary-backup because $P_{1(1)}$ and $P_{1(2)}$ have to run even if there is no fault, as shown in Figure 2.4b$_1$. In the case of primary-backup, illustrated in Figure 2.4c, the "backup" replica $P_{1(2)}$ is activated only if a fault occurs in $P_{1(1)}$. However, if faults occur, primary-backup takes more time to complete compared to active replication as shown in Figure 2.4c$_2$, compared to Figure 2.4b$_2$. To improve performance, primary-backup can be enhanced with checkpointing as discussed, for example, in [Sze05] so that only a part of the replica is executed in the case of faults.

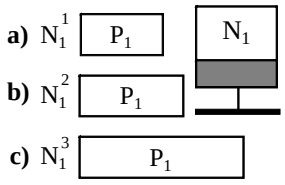
In our work, we are mostly interested in active replication. This type of replication provides the possibility of spatial redundancy, which is lacking in rollback recovery.

2.2.5 HARDENING

Transient faults can also be addressed with hardening techniques, i.e., improving the hardware architecture to reduce the transient fault rate. Researchers have proposed a variety of hardware hardening techniques. Zhang et al. [Zha06] have proposed an approach to hardening of flip-flops, resulting in a small area overhead and significant reduction in the transient fault rate. Mohanram and Touba [Moh03] have studied hardening of combinatorial circuits. Zhou et al. [Zho08] have proposed a "filtering technique" for hardening of combinatorial circuits. Zhou and Mohanram [Zho06] have studied the problem of gate resizing as a technique to reduce the transient fault rate. Garg et al. [Gar06] have connected diodes to the duplicated gates to implement an efficient and fast voting mechanism. Finally, a hardening approach to be applied in early design stages has been presented in [Hay07], which is based on the transient fault detection probability analysis.

Nevertheless, hardening comes with a significant overhead in terms of cost and speed [Pat08, Tro06]. The main factors which affect the cost are the increased silicon area, additional design effort, lower production quantities, excessive power consumption, and protection mechanisms against radiation, such as shields. Hardened circuits are also significantly slower than the regular ones. Manufacturers of hardened circuits are often forced to use technologies few generations back [Pat08, Tro06]. Hardening also enlarges the critical path of the circuit, because of a voting mechanism [Gar06] and increased silicon area.

To reduce the probability of faults, the designer can choose to use a hardened, i.e., a more reliable, version of the computation node. Such a hardened version will be called an h-version. Thus, each node N_j is available in several versions, with different hardening levels, denoted with h. We denote N_j^h the h-version of node N_j, and with C_j^h the cost associated with N_j^h. In Figure 2.5 we consider one process, P_1, and one computation node, N_1, with



N_1	$h = 1$		$h = 2$		$h = 3$	
	t	p	t	p	t	p
P_1	80	$4 \cdot 10^{-2}$	100	$4 \cdot 10^{-4}$	160	$4 \cdot 10^{-6}$
Cost	10		20		40	

Figure 2.5: Hardening

three h-versions, N_1^1 without hardening and N_1^2 and N_1^3 progressively more hardened. The execution times (t) and failure probabilities (p) for the process on different h-versions of node N_1 are shown in the table. The corresponding costs are also associated with these versions (given at the bottom of the table). For example, with the h-version N_1^2, the failure probability is reduced by two orders of magnitude, compared to the first version N_1^1. However, using N_1^2 will cost twice as much as the solution with less hardening. Moreover, with more hardening, due to performance degradation, the execution time on N_1^3 is twice as much as on the first version N_1^1.

2.3 Transparency

A common systematic approach for debugging embedded software is to insert observation points into software and hardware [Vra97, Tri05, Sav97] for observing the system behaviour under various circumstances. The observation points are usually inserted by an expert, or can be automatically injected based on statistical methods [Bou04]. 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.

Tolerating transient faults leads to many execution scenarios, which are dynamically adjusted in the case of fault occurrences.

The number of execution scenarios grows exponentially with the number of processes and the number of tolerated transient faults. In order to debug, test, or verify the system, all its execution scenarios have to be taken into account. Therefore, monitoring observation points for all these scenarios is often infeasible and debugging, verification and testing become very difficult.

The overall number of possible recovery scenarios can be considerably reduced by restricting the system behaviour, in particular, by introducing *transparency requirements* or, simply, *transparency*. A transparent recovery scheme has been proposed in [Kan03a], 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 debugability, where the occurrence of faults in a certain process does not affect the execution of other processes, which 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. To our knowledge, most of the design strategies proposed so far [Yin03, Yin06, Xie04, Pin08, Sri95, Mel04, Aya08, Bar08, Aid05] 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 [Kan03a].

2.4 Design Optimization with Fault Tolerance

Fault-tolerant embedded systems have to be optimized in order to meet time, quality-of-service, and cost constraints. Researchers have shown that schedulability of an application can be guaranteed for pre-emptive online scheduling under the presence of a single transient fault [Ber94, Bur96, Han03].

Liberato et al. [Lib00] 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. Ying Zhang and Chakrabarty [Yin03] have proposed a checkpointing optimization approach for online fixed-priority scheduling to tolerate k faults in periodic real-time tasks during a hyperperiod. The application is run on a monoprocessor system and only rollback recovery with checkpointing is considered as a fault tolerance technique.

Hardware/software co-synthesis with fault tolerance has been addressed in [Sri95] in the context of event-driven scheduling. Hardware and software architectures have been synthesized simultaneously, providing a specified level of fault tolerance and meeting the performance constraints, while minimizing the system costs. Safety-critical processes are re-executed in order to tolerate transient fault occurrences. This approach, in principle, also addresses the problem of tolerating multiple transient faults, but does not consider static cyclic scheduling. Design optimization is limited to only hardware/software co-design, where some of the software functionality is migrated to ASICs for improving performance. Both hard and soft real-time constraints are considered in this work. However, value-based scheduling optimization is not performed, assuming unchanged fixed priorities of process executions in all hardware/software co-design solutions.

Xie et al. [Xie04] have proposed a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Introducing redundant processes into a pre-

designed schedule has been used in [Con05] in order to improve error detection. Both approaches only consider one single fault.

Szentivanyi et al. [Sze05] have proposed a checkpoint optimization approach in the context of servers running high-availability applications, employing primary-backup replication strategy. The queuing theory has been used to mathematically model system behaviour in the context of requests arriving to the servers, and in order to optimize system availability in the presence of faults. However, as the authors in [Sze05] have re-called, their approach is not suitable for hard real-time applications, as the ones discussed in this thesis. Moreover, fault tolerance policy assignment, mapping and scheduling with fault tolerance are not addressed in [Sze05].

Ayav et al. [Aya08] 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. Shye et al. [Shy07] have developed a process-level redundancy approach against multiple transient faults with active replication on multi-core processors in general-purpose computing systems. Design optimization and scheduling are not addressed in [Shy07], assuming a given fault tolerance and execution setup.

Wattanapongsakorn and Levitan [Wat04] have optimized reliability for embedded systems, both for hardware and software, taking costs aspects into account. However, their technique is limited to only software or hardware *permanent* faults of a component, i.e., transient faults are not addressed. Traditional hardware replication and N-version programming are used as fault tolerance techniques. A simulated annealing-based algorithm has been developed to provide design optimization of *the reliability against permanent faults* with reduced hardware and software costs of the fault tolerance. However, neither mapping optimization nor scheduling have been addressed in [Wat04].

Aidemark et al. [Aid05] have developed a framework for node-level fault tolerance in distributed real-time systems. Systematic and application-specific error detection mechanisms with

recovery have been used to ensure fail-silent behaviour of computation nodes in the presence of transient faults. Although a fixed priority scheduling with reserved recovery slacks is assumed to be employed, design optimization and scheduling with fault tolerance are not addressed in [Aid05].

Power-related optimization issues of fault-tolerant embedded systems have been studied in [Yin06, Jia05, Zhu05, Mel04, Wei06, Pop07]. Ying Zhang et al. [Yin04, Yin06] have studied fault tolerance and dynamic power management in the context of message-passing distributed systems. The number of checkpoints has been optimized in order to improve power consumption and meet timing constraints of the system without, however, performing fault tolerance-aware optimization of mapping and scheduling. Fault tolerance has been applied on top of a pre-designed system, whose process mapping and scheduling ignore the fault tolerance issue. Jian-Jun Han and Qing-Hua Li [Jia05] have proposed a scheduling optimization algorithm for reducing power consumption in the context of *online* least-execution-time-first scheduling. Dakai Zhu et al. [Zhu05] have studied sequential and parallel recovery schemes on a set of distributed servers, which tolerate arbitrary faults affecting aperiodic tasks/requests. They use very different application model and neither consider hard deadlines nor optimize scheduling or mapping of processes. Melhem et al. [Mel04] have considered checkpointing for rollback recovery in the context of *online earliest-deadline-first* (EDF) scheduling on a *monoprocessor* embedded system. They have proposed two checkpointing policies for reducing power consumption, where the number of checkpoints can be analytically determined in the given context of EDF scheduling. Wei et al. [Wei06] have proposed an *online* scheduling algorithm for power consumption minimization in the context of hard real-time *monoprocessor* systems. Pop et al. [Pop07] have studied reliability and power consumption of distributed embedded systems. Mapping has been considered as given to the problem and only scheduling has been optimized. A

scheduling technique, based on our scheduling approach presented in Chapter 4 of this thesis, has been proposed to provide a schedulable solution, which satisfies the reliability against the given number of transient faults with the lowest-possible power consumption.

Kandasamy et al. [Kan03a] have proposed constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. The work has been later extended with fault-tolerant transmission of messages on a time-division multiple access bus [Kan03b]. Both papers consider only one fault per computation node, and only process re-execution is used.

Very few research work is devoted to general design optimization in the context of fault tolerance. For example, Pinello et al. [Pin04, Pin08] have proposed a simple heuristic for combining several static schedules in order to mask fault patterns. Passive replication has been used in [Alo01] to handle a single failure in multiprocessor systems so that timing constraints are satisfied. Multiple failures have been addressed with active replication in [Gir03] in order to guarantee a required level of fault tolerance and satisfy time constraints. None of these previous work, however, has considered optimal assignment of fault tolerance policies, nor has addressed multiple transient faults in the context of static cyclic scheduling.

Regarding hardware, a variety of hardening optimization techniques against transient faults have been developed, which optimize hardware cost and area overhead with respect to hardware reliability [Zha06, Moh03, Zho06, Zho08, Hay07, Gar06]. However, these techniques target optimization of hardware alone and do not consider embedded applications, which will be executed on this hardware and their fault recovery capabilities. Thus, hardening may either lead to unnecessary overdesign of hardware or to its insufficient reliability since hardware designers will use unrealistic assumptions about the software and the system as a whole. To overcome this limitation, hardware hard-

ening levels should be optimized in a more global system context, taking into account properties of the application and system requirements.

Regarding soft real-time systems, researchers have shown how faults can be tolerated with active replication while maximizing the quality level of the system [Mel00]. During runtime, the resource manager allocates available system resource for each arrived process such that the overall quality of the system is not compromised while degree of the fault tolerance is maintained. An *online* greedy resource allocation algorithm has been proposed, which incrementally chooses waiting process replicas and allocate them to the least loaded processors. In [Ayd00] faults are tolerated while maximizing the reward in the context of online scheduling and an imprecise computation model, where processes are composed of mandatory and optional parts. Monoprocessor architecture is considered and the fault tolerance is provided with *online* recovering of the task parts. In [Fux95] the trade-off between performance and fault tolerance, based on active replication, is considered in the context of online scheduling. This, however, incurs a large overhead during runtime which seriously affects the quality of the results. None of the above approaches considers value-based scheduling optimization in the context of static cyclic scheduling. In general, the considered value-based optimization is either very limited and based on costly active replication [Mel00, Fux95] or restricted to monoprocessor systems with online scheduling [Ayd00].

Hard and soft real-time systems have been traditionally scheduled using very different techniques [Kop97]. However, many applications have both components with hard and soft timing constraints [But99]. Therefore, researchers have recently proposed techniques for addressing mixed hard and soft real-time systems [But99, Dav93, Cor04b]. Particularly, Cortes et al. [Cor04b] have developed a design approach for multiprocessor embedded systems composed of soft and hard processes. None of

the above mentioned work on mixed soft and hard real-time systems, however, addresses fault tolerance aspects.

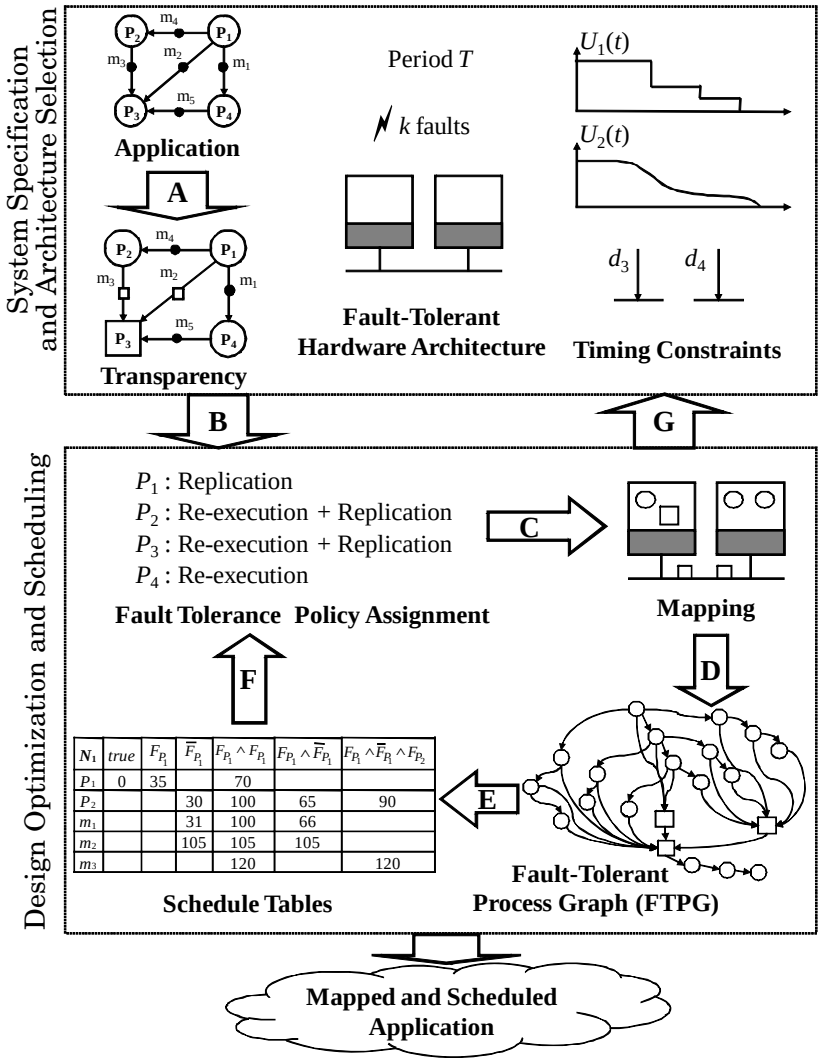
Hereafter we present the summary of limitations of previous research work, which we address in this thesis:

- design optimization of embedded systems with fault tolerance is usually restricted to a single aspect, as, for example, process mapping is not considered together with fault tolerance issues;
- fault tolerance policy assignment, e.g., deciding which fault tolerance technique or combination of techniques to apply to a certain process, is not considered;
- multiple faults are not addressed in the context of static cyclic scheduling;
- transparency, if at all addressed, is restricted to a whole computation node and is not flexible;
- fault tolerance aspects are not considered for mixed soft and hard real-time systems, i.e., the value-based optimization in the context of fault-tolerant mixed soft/hard embedded systems is not addressed;
- reliability of hardware is usually addressed alone, without considering software-level fault tolerance, which may lead to unnecessarily expensive solutions.

2.4.1 DESIGN FLOW WITH FAULT TOLERANCE TECHNIQUES

In Figure 2.6 we enhance the generic design flow presented in Figure 2.1, with the consideration of fault tolerance techniques. In the "System Specification and Architecture Selection" stage, designers specify, besides other functional and non-functional properties, timing constraints, for example, deadlines, and select a certain fault-tolerant architecture. They also set the maximum number k of transient faults in the application period T, which must be tolerated in software for the selected architecture. Designers can introduce transparency requirements in order to improve the debugability and testability on the selected archi-

tecture (step **A** in Figure 2.6). Based on the number k of transient faults and the transparency requirements, design



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

Figure 2.6: Design Flow with Fault Tolerance

optimization and scheduling are performed in the "Design Optimization and Scheduling" stage.[1]

In the "Fault Tolerance Policy Assignment" step in Figure 2.6, processes are assigned with fault tolerance techniques against transient faults. For example, some processes can be assigned with recovery (re-execution), some with active replication, and some with a combination of recovery and replication. In the context of rollback recovery, we also determine the adequate number of checkpoints. The policy assignment is passed over to the "Mapping" step (**C**), where a mapping algorithm optimizes the placement of application processes and replicas on the computation nodes. After that, the application is translated in step **D** into an intermediate representation, a "Fault-Tolerant Process Graph", which is used in the scheduling step. The fault-tolerant process graph (FTPG) representation captures the transparency requirements and all possible combinations of fault occurrences. Considering the mapping solution and the FTPG, a fault-tolerant schedule is synthesized (**E**) as a set of "Schedule Tables", which are captured in a schedule tree.

The generated schedules *have to* meet hard deadlines even in the presence of k faults in the context of limited amount of resources. If the application is unschedulable, the designer has to change the policy assignment and/or mapping (**F**). If a valid solution cannot be obtained after an extensive iterative mapping and policy assignment optimization, then the system specification and requirements, for example, transparency requirements or timing constraints, have to be adjusted or the fault-tolerant hardware architecture has to be modified (**G**).

---

1. Our design optimization and scheduling strategies, presented in Part II and Part III of the thesis, in general, follow this design flow with the maximum number k of transient faults provided by the designer. In Part IV, however, the number k of transient faults to be considered is calculated based on our system failure probability analysis.

Chapter 3
Preliminaries

IN THIS CHAPTER we introduce our application and quality-of-service (utility) models, hardware architecture, and our fault model. We also present our approach to process recovery in the context of static cyclic scheduling.

3.1 System Model

In this section we present details regarding our application models, including a quality-of-service model, and system architecture.

3.1.1 HARD REAL-TIME APPLICATIONS

In Part II and Part IV of this thesis, we will consider *hard real-time* applications. We model a hard real-time application $\mathcal{A}$ as a set of directed, acyclic graphs merged into a single hypergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node $P_i \in \mathcal{V}$ represents one process. An edge $e_{ij} \in \mathcal{E}$ from P_i to P_j indicates that the output of P_i is the input of P_j. Processes are non-preemptable and cannot be interrupted by other processes. Processes send their output values encapsu-

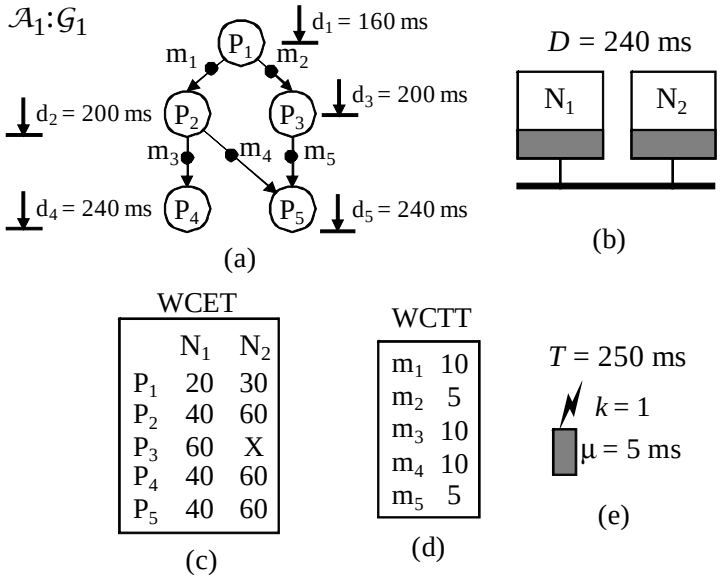


Figure 3.1: Hard Real-Time Application

lated in messages, when completed. *All required inputs* have to arrive before activation of the process. Figure 3.1a shows a simple application $\mathcal{A}_1$ represented as a graph $\mathcal{G}_1$ composed of five nodes (processes P_1 to P_5) connected with five edges (messages m_1 to m_5).

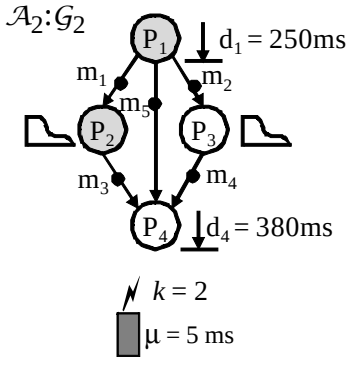
In a hard real-time application, violation of a deadline is not allowed. We capture time constraints with hard deadlines $d_i \in \mathcal{D}$, associated to processes in the application $\mathcal{A}$. In Figure 3.1a, all processes have to complete before their deadlines d_i, for example, process P_1 has to complete before d_1 = 160 ms and process P_5 before d_5 = 240 ms. In this thesis, we will often represent the hard deadlines in form of a global cumulative deadline D[1].

1. 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 $C_{dummy} = D - d_i$, which, however, is not allocated to any resource [Pop03].

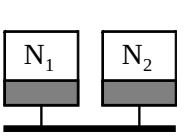
3.1.2 MIXED SOFT AND HARD REAL-TIME APPLICATIONS

In Part III of this thesis, we will consider *mixed soft and hard real-time* applications, and will extend our hard real-time approaches to deal also with *soft timing constraints*. Similar to the hard real-time application model, we model a mixed application $\mathcal{A}$ as a set of directed, acyclic graphs merged into a single hypergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node $P_i \in \mathcal{V}$ represents one process. An edge $e_{ij} \in \mathcal{E}$ from P_i to P_j indicates that the output of P_i is the input of P_j. The mixed application consists of hard and soft real-time processes. Hard processes are mandatory to execute and have to meet their hard deadlines. Soft processes, as opposed to hard ones, can complete after their deadlines. Completion time of a soft process is associated with a value (utility) function that characterizes its contribution to the quality-of-service of the application. Violation of a soft timing constraint is not as critical as violation of a hard deadline. However, it may lead to application quality deterioration, as will be discussed in Section 3.1.6. Moreover, a soft process may not start at all, e.g. may be *dropped*, to let a hard or a more important soft process execute instead.

In Figure 3.2 we have an application $\mathcal{A}_2$ consisting of the process graph $\mathcal{G}_2$ with four processes, P_1 to P_4. The hard part



	$\mathcal{M}(P_i)$	BCET	AET	WCET
P_1	N_1	30	50	70
P_2	N_1	50	60	100
P_3	N_2	50	75	120
P_4	N_2	30	60	80



$m_2 : m_3 : 10$ ms

$m_5 : 5$ ms

Figure 3.2: Mixed Soft and Hard Real-Time Application

consists of hard processes P_1 and P_4 and hard message m_5. Process P_1 has deadline d_1 = 250 ms and process P_4 has deadline d_4 = 380 ms. The soft part consists of soft process P_2 with soft messages m_1 and m_3, and soft process P_3 with soft messages m_2 and m_4, respectively. A soft process can complete after its deadline and the utility functions are associated to soft processes, as will be discussed in Section 3.1.6. The decision which soft process to execute and when should eventually increase the overall quality-of-service of the application without, however, violation of hard deadlines in the hard part.

3.1.3 BASIC SYSTEM ARCHITECTURE

The real-time application is assumed to run on a hardware architecture, which is composed of a set of computation nodes connected to a communication infrastructure. Each node consists of a memory subsystem, a communication controller, and a central processing unit (CPU). For example, an architecture composed of two computation nodes (N_1 and N_2) connected to a bus is shown in Figure 3.1b.

The application processes have to be *mapped* on the computation nodes. The mapping of an application process is determined by a function $\mathcal{M}$: $\mathcal{V} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. For a process $P_i \in \mathcal{V}$, its mapping $\mathcal{M}(P_i)$ is the node N_j to which P_i is assigned for execution. We consider that the mapping of the application is not fixed and has to be determined as a part of the design optimization.

For real-time applications, we know, first of all, a worst-case execution time (WCET), t_{ij}^{w}. Although finding the WCET of a process is not trivial, there exists an extensive portfolio of methods that can provide designers with safe worst-case execution time estimations [Erm05, Sun95, Hea02, Jon05, Gus05, Lin00, Col03, Her00, Wil08]. Figure 3.1c shows the worst-case execution times of processes of the application $\mathcal{A}_1$ depicted in Figure 3.1a. For example, process P_2 has the worst-case execu-

tion time of 40 ms if mapped on computation node N_1 and 60 ms if mapped on computation node N_2. By "X" we show mapping restrictions. For example, process P_3 cannot be mapped on computation node N_2.

Besides a worst-case execution time (WCET), ${t_{ij}}^w$, we also know for each process P_i, when mapped on node N_j, a best-case execution time (BCET), ${t_{ij}}^b$, and an expected (average) execution time (AET), ${t_{ij}}^e$, given by

$$t_{ij}^e = \int_{t_{ij}^b}^{t_{ij}^w} tE_{ij}(t)dt$$

for an arbitrary continuous execution time probability distribution $E_{ij}(t)$ of process P_i on computation node N_j. In Figure 3.2, with a mixed application $\mathcal{A}_2$, the execution times for processes P_1, P_2, P_3 and P_4 and transmission times of messages are shown in the table, as if processes P_1 and P_2 are mapped on computation node N_1, and P_3 and P_4 on N_2.

Processes mapped on different computation nodes communicate by messages sent over the bus. We consider that the worst-case sizes of messages are given, which can be implicitly translated into the worst-case transmission times on the bus. For example, Figure 3.1d shows the worst-case transmission times of messages implicitly derived from the message worst-case sizes, in the case the messages are transmitted over 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.

3.1.4 FAULT TOLERANCE REQUIREMENTS

In our system model, we consider that at most k transient faults may occur during a hyperperiod T of the application running on the given architecture. The application software has to tolerate these transient faults by fault tolerance techniques such as rollback recovery or active replication. For example, application

$\mathcal{A}_1$ has to tolerate $k = 1$ transient faults in the hyperperiod T = 250 ms, as depicted in Figure 3.1e; application $\mathcal{A}_2$ in Figure 3.2 has to tolerate $k = 2$ transient faults in its hyperperiod T = 400 ms.

Overheads related to fault tolerance are also part of our system model. When recovering from faults, we explicitly consider a recovery overhead μ, which includes the time needed to restore a process state and restart the process (or a part of the process). For example, recovery overhead μ is 5 ms for both applications $\mathcal{A}_1$ and $\mathcal{A}_2$, as depicted in Figure 3.1e and Figure 3.2, respectively.

Checkpointing overhead χ and error detection overhead α are also considered, as discussed in Section 2.2. However, since they do not influence design optimization and scheduling decisions we will not explicitly model them except the discussion on checkpoint optimization in Section 3.2 and Chapter 6.

3.1.5 ADAPTIVE STATIC CYCLIC SCHEDULING

In this thesis, we will consider a *static cyclic scheduling based* approach. With static cyclic scheduling both communications and processes are scheduled such that start times are determined off-line using scheduling heuristics. These start and sending times are stored in form of schedule tables on each computation node. Then, the runtime scheduler of a computation node will use the schedule table of that node in order to invoke processes and send messages on the bus.

In Figure 3.3b we depict a static schedule for the application and the hardware architecture presented in Figure 3.1. Processes P_1, P_3 and P_5 are mapped on computation node N_1 (grey circles), while processes P_2 and P_4 are mapped on N_2 (white circles). The schedule table of the computation node N_1 contains start times of processes P_1, P_3 and P_5, which are 0, 20, and 100 ms, respectively, plus sending time of message m_1, which is 20 ms. The schedule table of N_2 contains start times of P_2 and P_4,

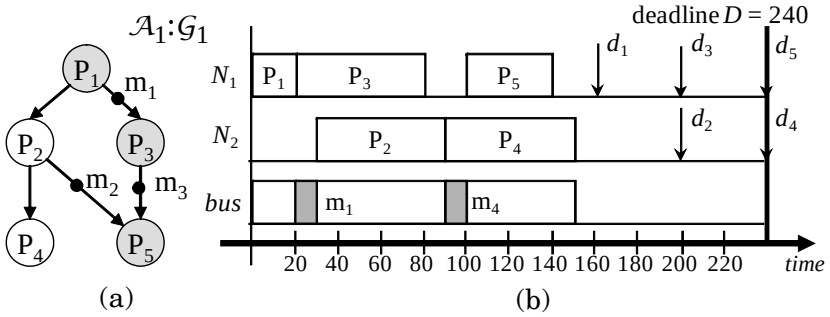


Figure 3.3: A Static Schedule

30 and 90 ms, plus sending time of message m_4, which is 90 ms. According to the static schedule, the application will complete at 150 ms, which satisfies the cumulative deadline D of 240 ms. All local deadlines of hard processes are also met.

Static cyclic scheduling, which we have used to generate the schedule in Figure 3.3b, is an attractive option for safety-critical applications. It provides predictability and deterministic behaviour. However, unless extended with adaptive functionality, it cannot efficiently handle faults and overload situations [Dim01, Kan03a] by adapting to particular conditions. For example, suppose that we have to handle the maximum $k = 1$ transient faults for application $\mathcal{A}_1$ in Figure 3.1, given the option that only one single static schedule can be used. In such a schedule, *each process* would need to be explicitly scheduled in *two* copies assuming the worst-case execution times of processes, which would lead to hard deadline violations and is not efficient.

In general, an application can have different execution scenarios [Ele00]. For example, some parts of the application might not be executed under certain conditions. In this case, several execution scenarios, corresponding to different conditions, have to be stored. At execution time, the runtime scheduler will choose the appropriate schedule that corresponds to the actual conditions. If the conditions change, the runtime scheduler will accordingly

switch to the appropriate precalculated schedule. Such a scheduling technique is called *quasi-static scheduling*.

Quasi-static mechanisms will be exploited in the following sections for capturing the behaviour of fault-tolerant applications. However, at first, to illustrate scheduling in the context of mixed soft and hard real-time systems, we will introduce our quality-of-service model, which is also a part of our system model.

3.1.6 QUALITY-OF-SERVICE MODEL

In our mixed soft and hard real-time application model, presented in Section 3.1.2, each soft process $P_i \in \mathcal{V}$ is assigned with a utility function $U_i(t)$, which is any non-increasing monotonic function of the completion time of a process. The *overall utility* of each execution scenario of the application is the sum of individual utilities produced by soft processes in this scenario.

In Figure 3.4 we depict utility function $U_2(t)$ for the soft process P_2 of the application $\mathcal{A}_2$ in Figure 3.2. According to the schedule in Figure 3.4a, P_2 completes at 110 ms and its utility would be 15. For a soft process P_i we have the option to "drop"

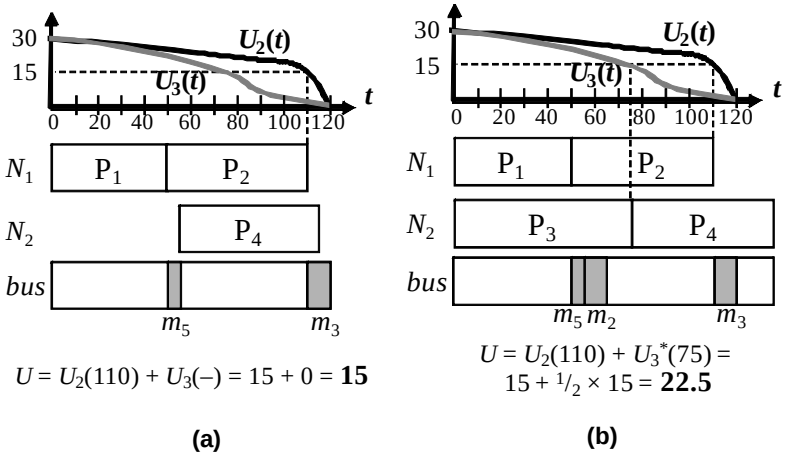


Figure 3.4: Utility Functions and Dropping

the process, and, thus, its utility will be 0, i.e., $U_i(-) = 0$. In Figure 3.4a we drop process P_3 of application $\mathcal{A}_2$. Thus, the overall utility of the application in this case will be $U = U_2(110) + U_3(-) = 15 + 0 = 15$. We may also drop soft messages of the application alone or together with the producer process. For example, in Figure 3.4a, messages m_2 is dropped. Dropping might be necessary in order to meet deadlines of hard processes, or to increase the overall system utility (e.g. by allowing other, potentially higher-value soft processes to complete).

If P_i, or its re-execution, is dropped but would produce an input for another process P_j, we assume that P_j will use an input value from a previous execution cycle, i.e., a "stale" value. Thus, output values produced by processes in one execution cycle can be reused by processes in the next cycles. For example, in Figure 3.4a, process P_2 will send message m_3 to update an input "stale" value for process P_4 in the next iteration. Reusing stale inputs, however, may lead to reduction in the utility value, i.e., utility of a process P_i would degrade to $U_i^*(t) = \sigma_i \times U_i(t)$, where σ_i represents the stale value coefficient. σ_i captures the degradation of utility that occurs due to dropping of processes, re-executions and messages, and is obtained according to an application-specific rule $\mathcal{R}$. In this thesis, we will consider that if a process P_i completes, but reuses stale inputs from one or more of its direct predecessors, the stale value coefficient is calculated as follows:

$$\sigma_i = \frac{1 + \sum_{P_j \in DP(P_i)} \sigma_j}{1 + \left|DP(P_i)\right|}$$

where $DP(P_i)$ is the set of P_i's direct predecessors.

If we apply the above rule to the execution scenario in Figure 3.4b, the overall utility is $U=U_2(110) + U_3^*(75) = 15 + {}^1/_2 \times 15 = 22.5$. The utility of process P_3 is degraded because it uses a "stale" value from process P_1, i.e., P_3 does not wait for input m_2

and its stale value coefficient $\sigma_3 = (1 + \sigma_1) / (1 + |DP(P_3)|) = (1 + 0) / (1 + 1) = {}^1/_2$.

3.2 Software-level Fault Tolerance Techniques

To illustrate adaptive scheduling with fault tolerance, in Figure 3.5a we show a simple application of two processes that has to tolerate the maximum number of two transient faults, i.e., $k = 2$.

As mentioned in Chapter 2, error detection itself introduces a certain time overhead, which is denoted with α_i for a process P_i. Usually, unless otherwise specified, we account for the error-detection overhead in the worst-case execution time of processes. In the case of re-execution or rollback recovery with checkpointing, a process restoration or recovery overhead μ_i has to be considered for a process P_i. The recovery overhead includes the time needed to restore the process state. Rollback recovery is also characterized by a checkpointing overhead χ_i, which is

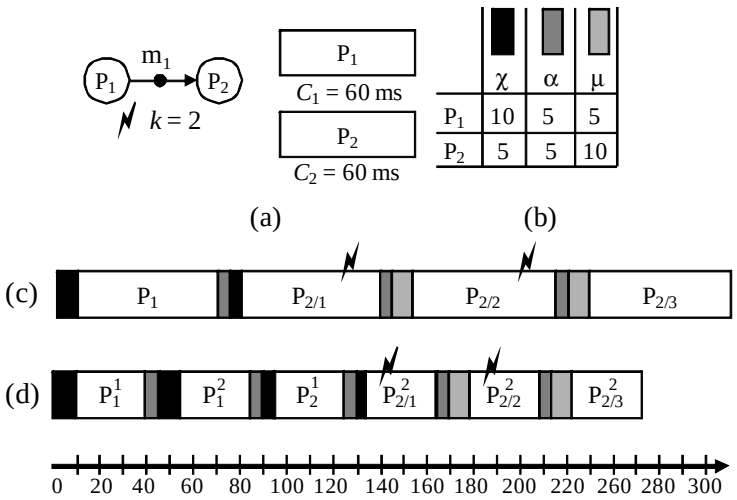


Figure 3.5: Fault Model and Fault Tolerance Techniques

related to the time needed to store initial and intermediate process states.

We consider that the worst-case time overheads related to the particular fault tolerance techniques are given. For example, Figure 3.5b shows recovery, detection and checkpointing overheads associated with the processes of the simple application depicted in Figure 3.5a. The worst-case fault scenarios of this application in the presence of two faults, if re-execution and rollback recovery with checkpointing are applied, are shown in Figure 3.5c and Figure 3.5d, respectively. As can be seen, the overheads related to the fault tolerance techniques have a significant impact on the overall system performance.

As discussed in Section 2.3, such fault tolerance techniques as re-execution and rollback recovery with checkpointing make debugging, testing, and verification potentially difficult. Transparency is one possible solution to this problem. Our approach to handling transparency is by introducing the notion of *frozenness* applied to a process or a message. A frozen process or a frozen message has to be scheduled at the same start time in all fault scenarios, independently of *external* fault occurrences[1].

Given an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ we will capture the transparency using a function $\mathcal{T}$: $\mathcal{W} \rightarrow \{Frozen, Regular\}$, where $\mathcal{W}$ is the set of all processes and messages. If $\mathcal{T}(w_i) = Frozen$, 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 [Kan03a] support a limited transparency setup, in which all the inter-processor messages are frozen, while all processes and all the intra-processor messages are regular.

1. External in respect to, i.e., outside, the frozen process or message.

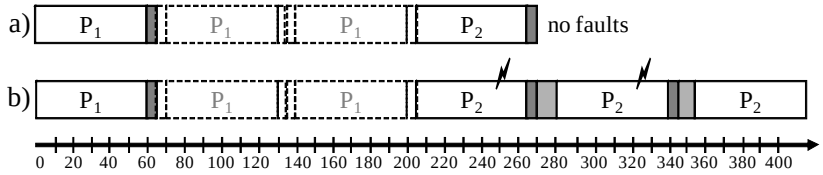


Figure 3.6: Transparency and Frozenness

Figure 3.6 shows the non-fault scenario and the worst-case fault scenario of the application depicted in Figure 3.5a, if re-execution is applied and process P_2 is frozen. Process P_2 is first scheduled at 205 ms in both execution scenarios independently of *external* fault occurrences, e.g., faults in process P_1. However, if fault occurrences are *internal*, i.e., within process P_2, process P_2 has to be re-executed as shown in Figure 3.6b.

3.2.1 RECOVERY IN THE CONTEXT OF STATIC CYCLIC SCHEDULING

In the context of static cyclic scheduling, each possible execution scenario has to be captured to provide the necessary adaptability to current conditions [Ele00].

Re-execution. In case of re-execution, faults lead to different execution scenarios that correspond to a set of *alternative schedules*. For example, considering the same application as in Figure 3.5a, with a maximum number of faults $k = 1$, re-execution will require three alternative schedules as depicted in Figure 3.7a. The fault scenario in which P_2 experiences a fault is shown with shaded circles. In the case of a fault in P_2, the run-time scheduler switches from the non-fault schedule S_0 to the schedule S_2 corresponding to a fault in process P_2.

Similarly, in the case of multiple faults, every fault occurrence will trigger a switching to the corresponding alternative schedule. Figure 3.7b represents a tree of constructed alternative schedules for the same application of two processes, if two tran-

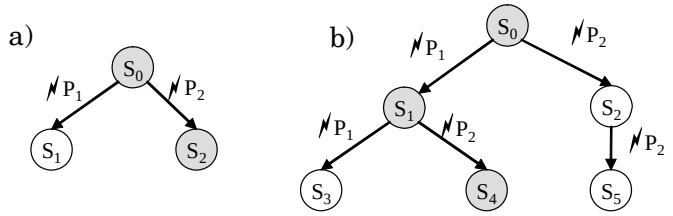


Figure 3.7: Alternative Schedules for Re-execution

sient faults can happen at maximum, i.e., $k = 2$. For example, as depicted with shaded circles, if process P_1 experiences a fault, the runtime scheduler switches from the non-fault schedule S_0 to schedule S_1. Then, if process P_2 experiences a fault, the runtime scheduler switches to schedule S_4.

Checkpointing. Figure 3.8 represents a tree of constructed alternative schedules for the same application in Figure 3.5a considering, this time, a rollback recovery schema with two checkpoints, as in Figure 3.5d. In the schedule, every execution segment P_i^j is considered as a "small process" that is recovered in the case of fault occurrences. Therefore, the number of alternative schedules is larger than it is in the case of pure re-execution. In Figure 3.8 we highlight the fault scenario presented in Figure 3.5d with shaded circles. The runtime scheduler switches between schedules S_0, S_4, and S_{14}.

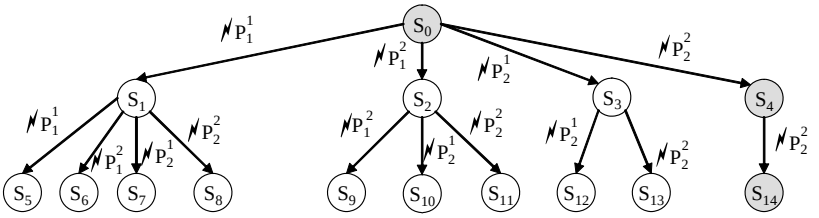


Figure 3.8: Alternative Schedules for Rollback Recovery with Checkpointing

Transparency. The size of the above tree of alternative schedules can be reduced by introducing transparency requirements. For example, in the case with frozen process P_2 for the application in Figure 3.5a, with the corresponding schedule depicted in Figure 3.6, it is sufficient to have a single schedule to capture the application behaviour in all possible fault scenarios even in case $k = 2$. This can be compared to 6 schedules, depicted in Figure 3.7b, for the same application but without transparency. However, the application with 6 schedules is more flexible, which will result in the much shorter worst schedule length than in the case depicted in Figure 3.6. We will investigate this relation between transparency requirements, the schedule tree size and the schedule length in the next Chapter 4, where we present our scheduling techniques with fault tolerance.

PART II
Hard Real-Time Systems

Chapter 4
Scheduling with Fault Tolerance Requirements

IN THIS CHAPTER we propose two scheduling techniques for fault-tolerant embedded systems in the context of hard real-time applications, namely *conditional scheduling* and *shifting-based scheduling*. Conditional scheduling produces shorter schedules than the shifting-based scheduling, and also allows to trade-off transparency for performance. Shifting-based scheduling, however, has the advantage of low memory requirements for storing alternative schedules and fast schedule generation time.

Both scheduling techniques are based on the *fault-tolerant process graph* (FTPG) representation.

Although the proposed scheduling algorithms are applicable for a variety of fault tolerance techniques, such as replication, re-execution, and rollback recovery with checkpointing, for the sake of simplicity, in this chapter we will discuss them in the context of only re-execution.

4.1 Performance/Transparency Trade-offs

As defined in Section 3.2, transparency refers to providing extended observability to the embedded application. The notion of transparency has been introduced with the property of frozenness applied to processes and messages, where a frozen process or a frozen message has to be scheduled independently of external fault occurrences.

Increased transparency makes a system easier to observe and debug. Moreover, since transparency reduces the number of execution scenarios, the amount of memory required to store alternative schedules, corresponding to these scenarios, is reduced. However, transparency increases the worst-case delays of processes, which can violate timing constraints of the application. These delays can be reduced by trading-off transparency for performance.

In the example in Figure 4.1a, we introduce transparency properties into the application $\mathcal{A}$. We let process P_3 and messages m_2 and m_3 be frozen, i.e., $\mathcal{T}(m_2)$ = *Frozen*, $\mathcal{T}(m_3)$ = *Frozen* and $\mathcal{T}(P_3)$ = *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. It is assumed that 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, with the sending processes and receiving processes mapped on different nodes, are scheduled on the bus. Four alternative execution scenarios are illustrated in Figure 4.1b-e.

The schedule in Figure 4.1b 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 execution of P_4, a second

fault occurs, the scheduler has to switch to yet another schedule illustrated in Figure 4.1c.

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 the case of faults in Figure 4.1c 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

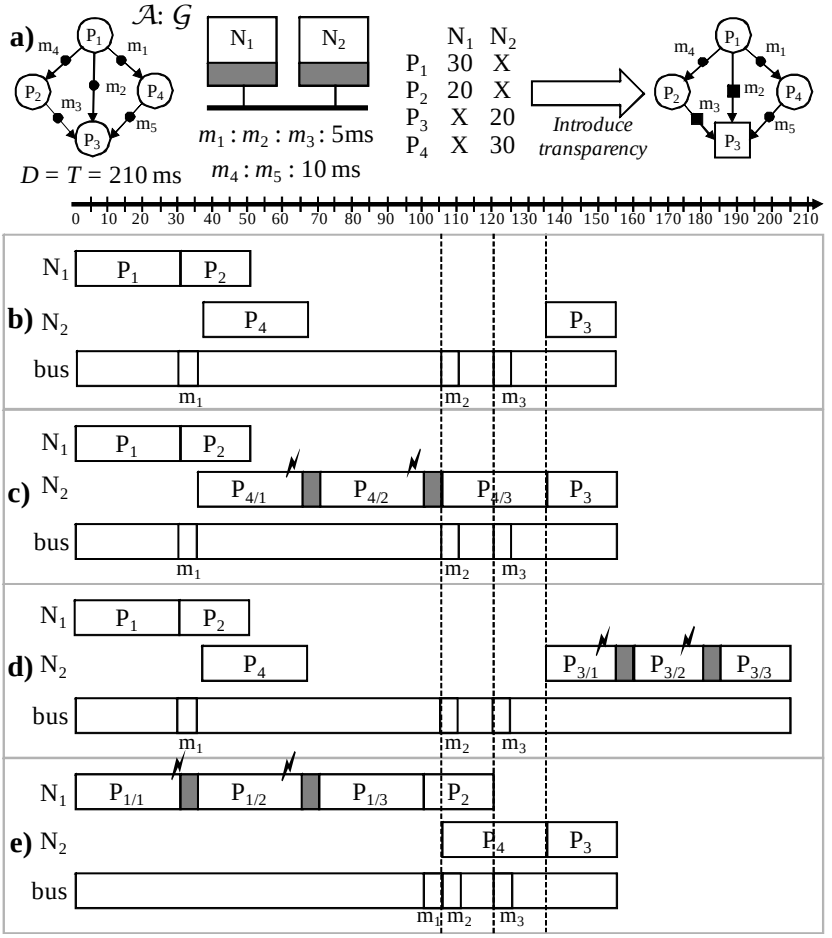


Figure 4.1: Application with Transparency

in all execution scenarios, which is illustrated with a dashed line crossing Figure 4.1. Even if no faults happen in process P_4, in the execution scenarios depicted in Figure 4.1d and Figure 4.1b, process P_3 will have to be delayed, which leads to a worst-case schedule as in Figure 4.1d. 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, can be sent at different times depending on fault occurrences in P_1, as illustrated, for example, in Figures 4.1b and 4.1e, respectively.

In Figure 4.2 we illustrate three alternatives, representing different transparency/performance setups for the application $\mathcal{A}$ in Figure 4.1. 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. 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-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 Figure 4.2a_2-c_2 to meet the deadline of 210 ms depicted with a thick vertical line.

In Figure 4.2a_1 and 4.2a_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 all 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 Figure 4.2a_1 and 4.2a_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 275 ms, which means that the deadline is missed.

The alternative in Figure 4.2b does not have any transparency restrictions. Figure 4.2b$_1$ shows the execution scenario if no fault occurs, while 4.2b$_2$ illustrates the worst-case scenario.

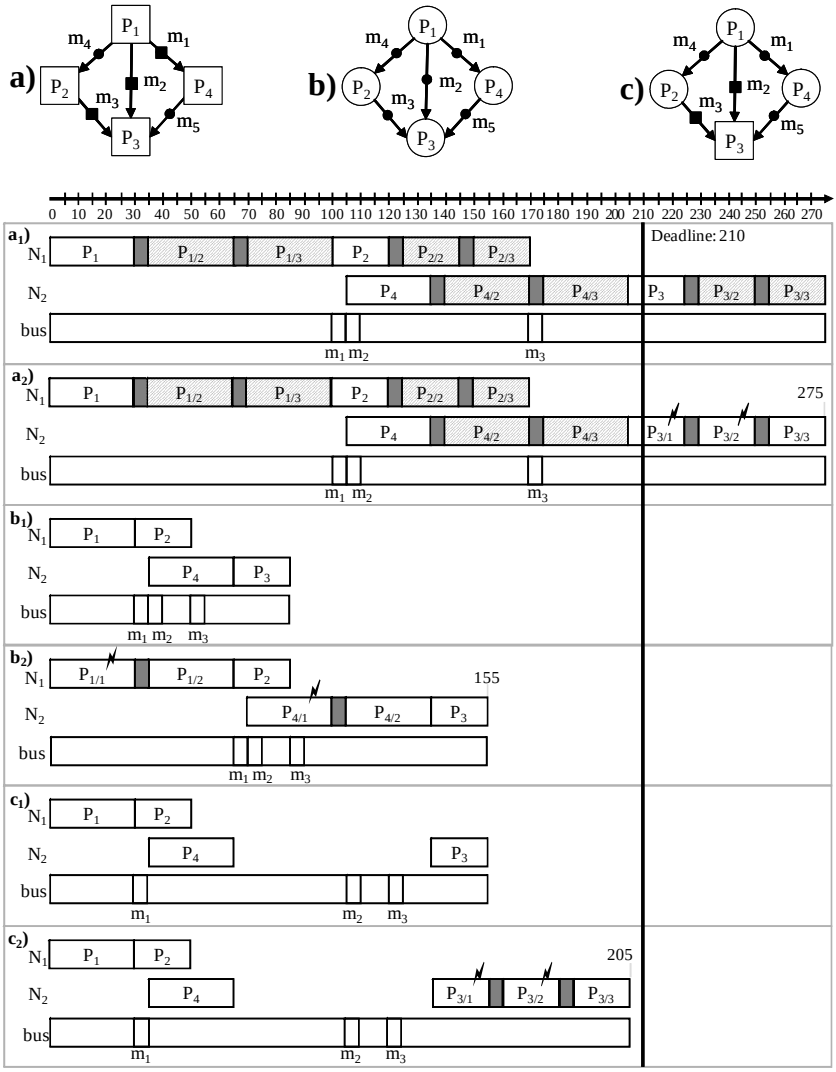


Figure 4.2: Trade-off between Transparency and Performance

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 that are customized to the actual fault scenarios and, thus, are more efficient. In Figure 4.2b$_2$, 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, which 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 Figure 4.2b$_2$, 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 Figure 4.2c, 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 Figure 4.2c$_2$, and the deadline is still met.

4.2 Fault-Tolerant Conditional Process Graph

The scheduling techniques presented in this section are 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 which is "true" if the fault happens and "false" if the fault does not happen.

In Figure 4.3a we have an application $\mathcal{A}$ modelled as a process graph $\mathcal{G}$, which 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 $\mathcal{A}$ to a fault-tolerant process graph (FTPG) G, represented in Figure 4.3b. 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 example, is P_1^1 to P_4^1 in Figure 4.3b, 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 Figure 4.3b captures all the fault scenarios that can happen during the execution of application $\mathcal{A}$ in

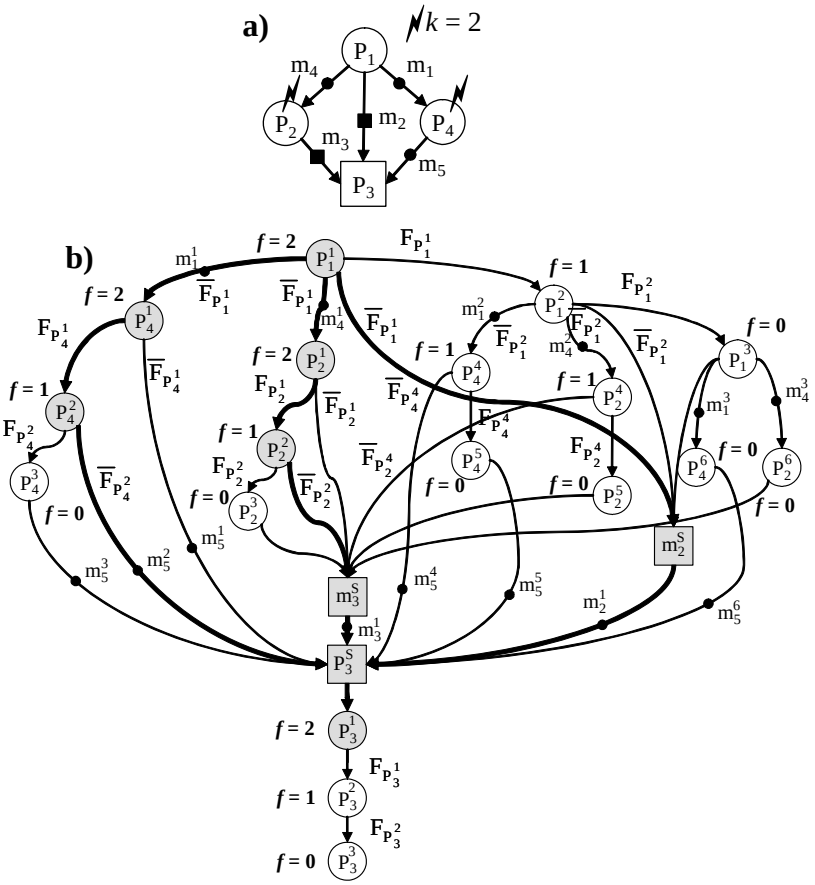


Figure 4.3: Fault-Tolerant Process Graph

Figure 4.3a. The subgraph marked with thicker edges and shaded nodes in Figure 4.3b captures the execution scenario when processes P_2 and P_4 experience one fault each, as illustrated in Figure 4.3a. 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 Figure 4.3b, before the copies corresponding to the possible executions of process P_3. The first execution P_3^1 of process P_3 has to be immediately scheduled after its synchronization node P_3^S. In Figure 4.3b, 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, can be activated after inputs coming on one of the alternative traces, corresponding to a particular fault scenario, have arrived. For example, a transmission on the edge $e_{12}^{1S^m}$, labelled $\overline{F}_{P_1^1}$, 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 where the value of its associated guard is true.

Definition. Formally, an FTPG corresponding to an application $\mathcal{A} = \mathcal{G}(\mathcal{V}, \mathcal{E})$ 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 P_i^m that will correspond to

the m^{th} copy of process $P_i \in \mathcal{V}$. 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 $\vartheta_i \in V_T$ 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 $P_i \in \mathcal{A}$ and with m_i^S the synchronization node corresponding to message $m_i \in \mathcal{A}$. 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; and
- $e_{ij}^{SS_m} \in E_S$ from P_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 traces starting from such a process, which correspond to complementary values of the condition, are disjoint[1]. Note that edges e_{ij}^{Sn}, $e_{ij}^{S_m n}$, e_{ij}^{SS}, $e_{ij}^{S_m S}$ and $e_{ij}^{SS_m}$ coming from a synchronization node cannot be conditional.

1. They can only meet in a synchronization node.

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

4.2.1 FTPG GENERATION

In Figure 4.4 we have outlined the BuildFTPG algorithm that traces processes in the application graph $\mathcal{G}$ with transparency requirements $\mathcal{T}$ in the presence of maximum k faults and 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 "fault" conditional edges with the "true" condition value (lines 3–5).

The root process and its copies 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 Figure 4.5a, we show the intermediate state resulted after this first step during the generation of the FTPG depicted in Figure 4.3b. 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 ready process list $\mathcal{L}_R$ (line 7). For generation of the FTPG, the order of processes in the ready process list $\mathcal{L}_R$ 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.

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 $\mathcal{L}_R = \emptyset$ **-- add successors of the root node to the process list**
7 **for** $\forall$Succ(P_i) $\in \mathcal{G}$ **do** $\mathcal{L}_R = \mathcal{L}_R \cup$ Succ(P_i)
8 **while** $\mathcal{L}_R \neq \emptyset$ **do -- trace all processes in the merged graph** $\mathcal{G}$
9 P_i = ExtractProcess($\mathcal{L}_R$)
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 synch. 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) $\notin \mathcal{L}_R$ **then** $\mathcal{L}_R = \mathcal{L}_R \cup$ Succ(P_i)
42 **end for**
43 **end while**
44 **return** G
end BuildFTPG

Figure 4.4: Generation of FTPG

For each process P_i, extracted from the ready process list $\mathcal{L}_R$, BuildFTPG prepares a set $\mathcal{VC}$ 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 $vc_n \in \mathcal{VC}$ 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 ready process list $\mathcal{L}_R$ and incorporate this process into the FTPG G. In the application graph $\mathcal{G}$, 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 $\{P_1^1, P_1^2, P_1^3\}$.

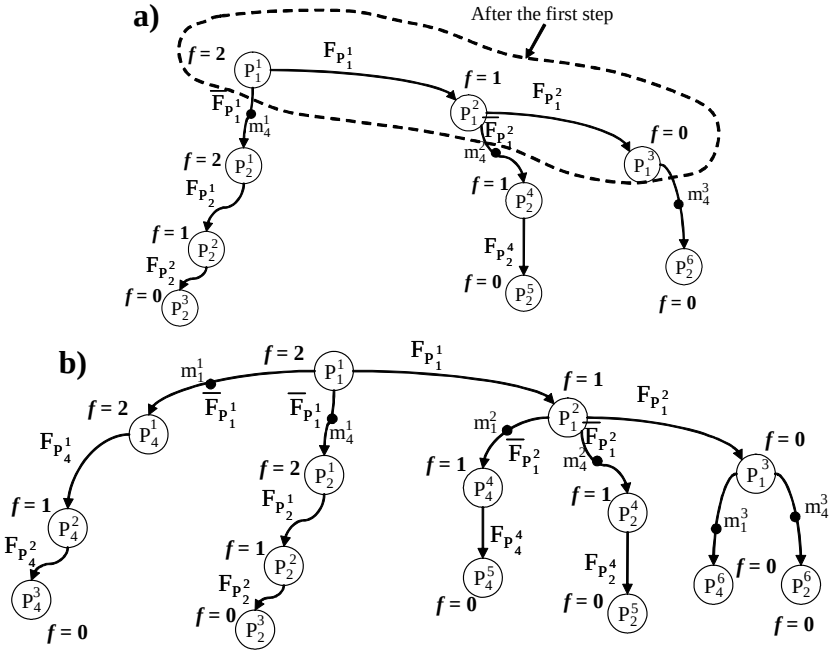


Figure 4.5: FTPG Generation Steps (1)

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 $F_{P_1^1}$.[1] 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 Figure 4.5a, the set of valid predecessors $\mathcal{VC}$ 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 $\mathcal{VC}$, 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.

For regular processes, such as P_2, the FTPG generation proceeds according to lines 28–38 in Figure 4.4. For each combination $vc_n \in \mathcal{VC}$, 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

1. An example of conflicting conditions 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 $\overline{F}_{P_1^2}$.

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 "fault" conditional edge $e_i^{h-1}{}_i^h$. 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 Figure 4.5a, 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 Figure 4.5b, 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 ready process list $\mathcal{L}_R$ (lines 40–42). For example, after P_2 and P_4 have been incorporated, process P_3 is placed into the ready process list $\mathcal{L}_R$. BuildFTPG continues until all processes and messages in the merged graph $\mathcal{G}$ are incorporated into the FTPG G, i.e., until the list $\mathcal{L}_R$ is empty (line 8).

After incorporating processes P_1, P_2 and P_4, the ready process list $\mathcal{L}_R$ 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 for

incorporating P_3 into the FTPG G will proceed according to lines 19-26 in Figure 4.4. In the application graph $\mathcal{G}$, process P_3 has three predecessors P_1, P_2, and P_4. Thus, its set of valid combinations $\mathcal{VC}$ of copies of the predecessor processes will be: $\{P_1^1, P_2^1, P_4^1\}$, $\{P_1^1, P_2^1, P_4^2\}$, $\{P_1^1, P_2^1, P_4^3\}$, $\{P_1^1, P_2^2, P_4^1\}$, $\{P_1^1, P_2^2, P_4^2\}$, $\{P_1^1, P_2^3, P_4^1\}$, $\{P_1^2, P_2^4, P_4^4\}$, $\{P_1^2, P_2^4, P_4^5\}$, $\{P_1^2, P_2^5, P_4^4\}$ 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 $\mathcal{VC}$ 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 Figure 4.6. m_2^S and m_3^S are connected to the copies of the predecessor processes with the following edges: $e_{12}^{1S_m}$, $e_{12}^{2S_m}$, and $e_{12}^{3S_m}$ (for m_2^S), and $e_{2\,2}^{1S_m}$, $e_{2\,2}^{2S_m}$, $e_{2\,2}^{3S_m}$, $e_{2\,2}^{4S_m}$, $e_{2\,2}^{5S_m}$, and $e_{2\,2}^{6S_m}$ (for m_3^S). The set of valid predecessors $\mathcal{VC}$ 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.

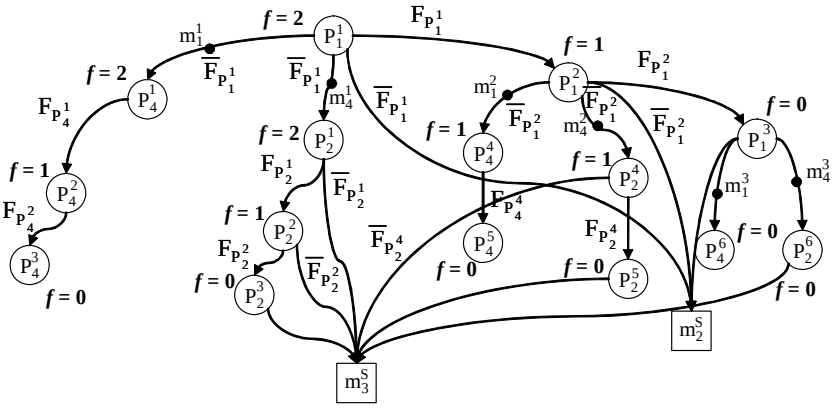


Figure 4.6: FTPG Generation Steps (2)

Since process P_3 is frozen, we first insert synchronization node P_3^S (line 19), as illustrated in Figure 4.3b, which is connected to the copies of the predecessor processes and the other synchronization nodes with the edges $e_{4\,3}^{1S}$, $e_{4\,3}^{2S}$, $e_{4\,3}^{3S}$, $e_{4\,3}^{4S}$, $e_{4\,3}^{5S}$, $e_{4\,3}^{6S}$, $e_{2\,3}^{S_m S}$ and $e_{3\,3}^{S_m S}$ (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 $e_{3\,3}^{S1}$. 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 "fault" conditional edges $e_{3\,3}^{1\,2}$ and $e_{3\,3}^{2\,3}$ (lines 24–26), which leads to the complete FTPG G depicted in Figure 4.3b. The algorithm will now terminate since the list $\mathcal{L}_R$ is empty.

4.3 Conditional Scheduling

Our conditional scheduling is based on the FTPG representation of the merged process graph $\mathcal{G}$, transformed as discussed in the previous section. In general, the problem that we will address with the FTPG-based conditional scheduling in this section can be formulated as follows: Given an application $\mathcal{A}$, mapped on an architecture consisting of a set of hardware nodes $\mathcal{N}$ interconnected via a broadcast bus B, and a set of transparency requirements on the application $\mathcal{T}(\mathcal{A})$, we are interested to determine the schedule table S such that the worst-case end-to-end delay δ_G, by which the application completes execution, is minimized, and the transparency requirements captured by $\mathcal{T}$ are satisfied. If the resulting delay is smaller than the deadline, the system is schedulable.

4.3.1 SCHEDULE TABLE

The output produced by the FTPG-based conditional scheduling algorithm is a schedule table that contains all the information needed for a distributed runtime scheduler to take decisions on

activation of processes and sending of messages. It is considered that, during execution, a 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 node. Figure 4.7 presents the schedules for nodes N_1 and N_2, which will be produced by the conditional scheduling algorithm in Figure 4.9 for the FTPG in Figure 4.3. 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 $\mathcal{A}$. 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 into a *signalling message*.[1]

1. In this work, we will use the same bus for broadcasting signalling messages as for the information messages. However, the presented approach can be also applied for architectures that contain a dedicated signalling bus.

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)				

Figure 4.7: Conditional Schedule Tables

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 [Ele00]. Only when the condition is known, i.e., has arrived with a signalling message, a decision can be taken that depends on this condition. In the schedule table, there is one row for each signalling message. For example, in Figure 4.7a, according to the schedule for node N_1, process P_1 is activated unconditionally at the time 0, given in the first column of the table. Activation of the rest of the processes, in a certain execution cycle, depends on the values of the conditions, i.e., the occurrence of faults during the execution of certain processes. For example, process P_2 has to be activated at $t = 30$ ms if $\overline{F}_{P_1^1}$ is true (no fault in P_1), and at $t = 100$ ms if $F_{P_1^1} \wedge F_{P_1^2}$ is true (faults in P_1 and its first re-execution), etc.

In Figure 4.8, we illustrate these signalling-based scheduling decisions. In Figure 4.8a, we show that, if process P_1 is affected by two faults, a corresponding "true" conditional value F_{P_1} is broadcasted twice over the bus with two signalling messages, at 30 and 65 ms, respectively, according to the schedule table in Figure 4.7a. Based on this information, regular message m_1 is sent after completion of re-execution $P_{1/3}$ at 100 ms. Regular process P_4, based on the broadcasted conditions, starts at 105

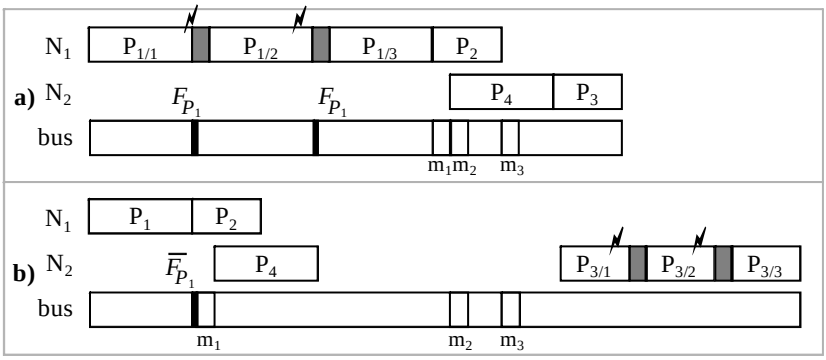


Figure 4.8: Signalling Messages

ms, after arrival of delayed message m_1. If process P_1 executes without faults, as depicted in Figure 4.8b, "false" conditional value $\overline{F}_{P_1}$ has to be broadcasted. In this case, according to the schedule tables in Figure 4.7, regular message m_1 is sent earlier, at 31 ms, and regular process P_4 starts at 36 ms. Note that, despite broadcasting of signalling messages, the start times of frozen messages m_2 and m_3, as well as of the frozen process P_3, are not affected.

In general, to produce a deterministic behaviour, which is globally consistent for any combination of conditions (faults), the schedule table, as the one depicted in Figure 4.7, has to fulfil the following requirements:

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

4.3.2 CONDITIONAL SCHEDULING ALGORITHM

According to our FTPG model, some processes can 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 [Ull75] 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 Figure 4.9. The FTScheduleSynthesis function produces the schedule table S, while taking as input the application graph $\mathcal{G}$ with the transparency requirements $\mathcal{T}$, the maximum number k of transient faults that have to be tolerated, the architecture consisting of computation nodes $\mathcal{N}$ and bus B, and the mapping $\mathcal{M}$.

Our synthesis approach employs a *list scheduling* based heuristic, FTPGScheduling, presented in Figure 4.11, for scheduling each alternative fault scenario. The heuristic not only derives

```
FTScheduleSynthesis(G, T, k, N, B, M)
1  S = ∅; G = BuildFTPG(G, T, k)
2  L_ϑ = GetSynchronizationNodes(G)
3  PCPPriorityFunction(G, L_ϑ)
4  if L_ϑ ≡ ∅ then
5    FTPGScheduling(G, M, ∅, S)
6  else
7    for each ϑ_i ∈ L_ϑ do
8      t_max = 0; K_ϑi = ∅
9      {t_max, K_ϑi} = FTPGScheduling(G, M, ϑ_i, S)
10     for each K_j ∈ K_ϑi do
11       Insert(S, ϑ_i, t_max, K_j)
12     end for
13   end for
14 end if
15 return S
end FTScheduleSynthesis
```

Figure 4.9: Fault-Tolerant Schedule Synthesis Strategy

dedicated schedules for each fault scenario but also enforces the requirements (1) to (3) presented in Section 4.3.1. It schedules synchronization nodes at the same start time in all of these alternative schedules.

In the first line of the FTScheduleSynthesis algorithm (Figure 4.9), we initialize the schedule table S and build the FTPG G as presented in Section 4.2.1.[1] If the FTPG does not contain any synchronization node ($\mathcal{L}_\vartheta \equiv \emptyset$), we perform the FTPG scheduling for the whole FTPG graph at once (lines 4–5).

If the FTPG contains at least one synchronization node $\vartheta_i \in \mathcal{L}_\vartheta$, where $\mathcal{L}_\vartheta$ is the list of synchronization nodes, the procedure is different (lines 7–13). A synchronization node ϑ_i must have the same start time t_i in the schedule S, regardless of the guard $K_j \in \mathcal{K}_{\vartheta_i}$, which captures the necessary activation conditions for ϑ_i under which it is scheduled. For example, the synchronization node m_2^S in Figure 4.10 has the same start time of 105 ms, in each corresponding column of the table in Figure 4.7.

In order to determine the start time t_i of a synchronization node $\vartheta_i \in \mathcal{L}_\vartheta$, we will have to investigate all the alternative fault-scenarios (modelled as different alternative traces through the FTPG) that lead to ϑ_i. Figure 4.10 depicts the three alternative traces that lead to m_2^S for the graph in Figure 4.3b. These traces are generated using the FTPGScheduling function (called in line 9, Figure 4.9), which records the maximum start time t_{max} of ϑ_i over the start times in all the alternative traces. In addition, FTPGScheduling also records the set of guards $\mathcal{K}_{\vartheta_i}$ under which ϑ_i has to be scheduled. The synchronization node ϑ_i is then inserted into the schedule table in the columns corresponding to the guards in the set $\mathcal{K}_{\vartheta_i}$ at the unique time t_{max} (line 11 in Figure 4.9). For example, m_2^S is inserted at time t_{max} = 105 ms

1. 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.

in the columns corresponding to $\mathcal{K}_{m_2^S} = \{\overline{F}_{P_1^1}, \; F_{P_1^1} \wedge F_{P_1^2}, \; F_{P_1^1} \wedge \overline{F}_{P_1^2}\}$.

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 *fault* branch and the *no fault* branch (lines 21 and 23, Figure 4.11). Thus, the alternative traces are not activated simultaneously and resource sharing is correctly achieved. Signalling messages, 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 $r_j \in \mathcal{N} \cup \{B\}$, which is either a computation node $N_j \in \mathcal{N}$ or the bus B, the *highest priority ready node* X_i is extracted from the head of the local priority list $\mathcal{L}_{Rj}$

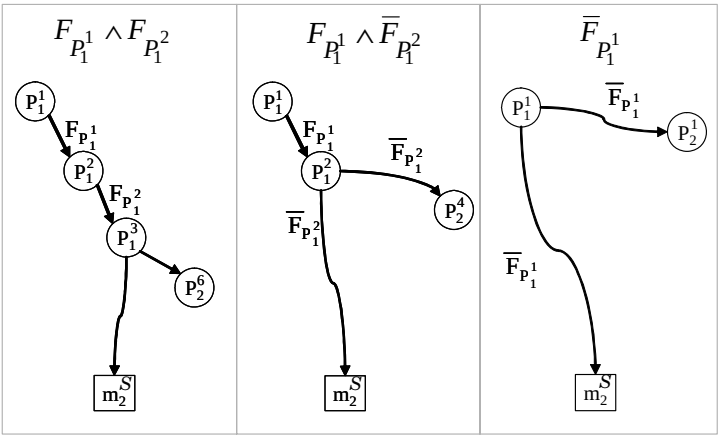


Figure 4.10: Alternative Traces Investigated by FTPGScheduling for the Synchronization Node m_2^S

(line 3). We use the *partial critical path* (PCP) priority function [Ele00] in order to assign priorities to the nodes (line 3 in FTScheduleSynthesis, Figure 4.9).

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 ϑ (line 8), the latest start time and the current guards are recorded (lines 10–11). If other

```
FTPGScheduling(G, M, ϑ, S)
1  while ∃X_i ∈ G | X_i ∉ S do
2  for each r_j ∈ N ∪ {B} do -- loop for each resource r_j
3     L_Rj = LocalReadyList(S, r_j, M) -- find unscheduled ready nodes on r_j
4     while L_Rj ≠ ∅ do
5        X_i = Head(L_Rj)
6        t = ResourceAvailable(r_j, X_i) -- the earliest time to accommodate X_i on r_j
7        K = KnownConditions(r_j, t) -- the conditions known to r_j at time t
8        if X_i ≡ ϑ then -- synchronization node currently under investigation
9           if t > t_max then
10             t_max = t -- the latest start time is recorded
11             K_ϑi = K_ϑi ∪ {K} -- the guard of synchronization node is recorded
12          end if
13          return {t_max, K_ϑi} -- exploration stops at the synchronization node ϑ
14       else if X_i ∈ V_T and X_i is unscheduled then -- other synch. nodes
15          continue -- are not scheduled at the moment
16       end if
17       Insert(S, X_i, t, K) -- the ready node X_i is placed in S under guard K
18       if X_i ∈ V_C then -- conditional process
19          Insert(S, SignallingMsg(X_i), t, K) -- broadcast conditional value
20          -- schedule the fault branch (recursive call for true branch)
21          FTPGScheduling(G, L_Rj ∪ GetReadyNodes(X_i, true))
22          -- schedule the non-fault branch (recursive call for false branch)
23          FTPGScheduling(G, L_Rj ∪ GetReadyNodes(X_i, false))
24       else
25          L_Rj = L_Rj ∪ GetReadyNodes(X_i)
26       end if
27    end while
28 end for
29 end while
end FTPGScheduling
```

Figure 4.11: Conditional Scheduling

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.[1] The time t is the time when the resource r_j is available (line 17). Guard K on the resource r_j is determined by the KnownConditions function (line 7).

Since we enforce the synchronization nodes to start at their latest time t_{max} to accommodate all the alternative traces, we might have to insert idle times on the resources. Thus, our ResourceAvailable function (line 6, Figure 4.11) will determine the start time $t \geq t_{asap}$ in the first continuous segment of time, which is 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 Figure 4.7a, m_2 is scheduled at 105 ms on the bus. We will later schedule m_1 at times 31, 100 and 66 ms on the bus (see Figure 4.7a).

4.4 Shifting-based Scheduling

Shifting-based scheduling is the second scheduling technique for synthesis of fault-tolerant schedules proposed in this thesis. This scheduling technique is an extension of the transparent recovery against single faults proposed in [Kan03a].

The problem that we address with shifting-based scheduling can be formulated as follows. Given an application $\mathcal{A}$, mapped on an architecture consisting of a set of hardware nodes $\mathcal{N}$ interconnected via a broadcast bus B, we are interested to determine the schedule table S with a *fixed execution order of processes* such that the worst-case end-to-end delay δ_G, by which the application completes execution, is minimized, and the transparency

1. Recall that synchronization nodes are inserted into the schedule table by the FTScheduleSynthesis function on line 11 in Figure 4.9.

requirements with *all messages on the bus frozen* are satisfied. If the resulting delay is smaller than the deadline, the system is schedulable.

In *shifting-based scheduling*, a fault occurring on one computation node is masked to the other computation nodes in the system but can impact processes on the same computation node. On a computation node N_i where a fault occurs, the scheduler has to switch to an alternative schedule that delays descendants of the faulty process running on the same computation node N_i. However, a fault happening on another computation node is not visible on N_i, even if the descendants of the faulty process are mapped on N_i.

Due to the imposed restrictions, the size of schedule tables for shifting-based scheduling is much smaller than the size of schedule tables produced with conditional scheduling. Moreover, shifting-based scheduling is significantly faster than the conditional scheduling algorithm presented in Section 4.3.2. However, first of all, shifting-based scheduling does not allow to trade-off transparency for performance since, by definition, *all messages on the bus, and only those, are frozen*. Secondly, because of the *fixed execution order of processes*, which does not change with fault occurrences, schedules generated with shifting-based scheduling are longer than those produced by conditional scheduling.

4.4.1 SHIFTING-BASED SCHEDULING ALGORITHM

The shifting-based scheduling algorithm is based on the FTPG representation, on top of which it introduces an additional ordering of processes, mapped on the same computation node, based on a certain priority function.

Let us illustrate the ordering with an example in Figure 4.12a showing an application $\mathcal{A}$ composed of five processes mapped on two computation nodes. Processes P_1, P_2 and P_4 are mapped on computation node N_1. Processes P_3 and P_5 are mapped on com-

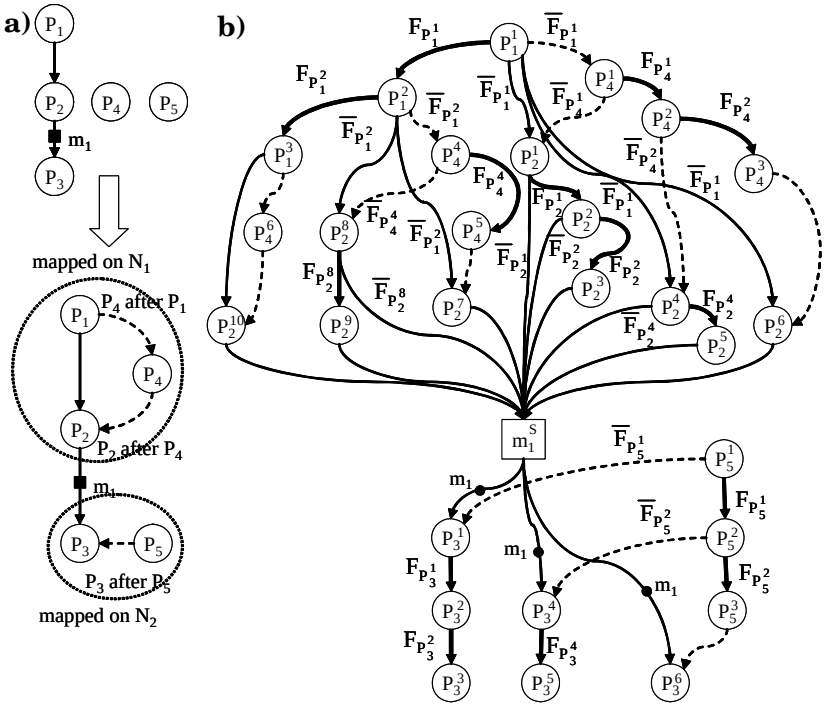


Figure 4.12: Ordered FTPG

putation node N_2. Message m_1 is frozen since it is transmitted through the bus. The order in Figure 4.12a (in the low part of the figure) is obtained using the partial critical path (PCP) priority function [Ele00] with the following ordering relations: (1) process P_4 cannot start before completion of process P_1; (2) P_2 cannot start before completion of P_4; and (3) P_3 cannot start before completion of P_5. The resulting FTPG with introduced ordering relations, when $k = 2$ transient faults can happen at maximum, is presented in Figure 4.12b. The introduced ordering relations order the execution of processes mapped on the same computation node in all execution scenarios.

For the shifting-based scheduling, the FTPG and the ordering relations are not explicitly generated. Instead, only a root sched-

ule is obtained off-line, which preserves the order of processes mapped on the same computation node in all execution scenarios. The root schedule consists of start times of processes in the non-faulty scenario and sending times of messages. In addition, it has to provide idle times for process recovering, called recovery slacks. The root schedule is later used by the runtime scheduler for extracting the execution scenario corresponding to a particular fault occurrence (which corresponds to a trace in the ordered FTPG). Such an approach significantly reduces the amount of memory required to store schedule tables.

Generation of the Root Schedule. The algorithm for generation of the root schedule is presented in Figure 4.13 and takes as input the application $\mathcal{A}$, the number k of transient faults that have to be tolerated, the architecture consisting of computation nodes $\mathcal{N}$ and bus B, the mapping $\mathcal{M}$, and produces the root schedule $\mathcal{RS}$.

Initial recovery slacks for all processes $P_i \in \mathcal{A}$ are calculated as $s_0(P_i) = k \times (C_i + \mu)$ (lines 2-4). (Later the recovery slacks of processes mapped on the same computation node will be merged to reduce timing overhead.)

The process graph $\mathcal{G}$ of application $\mathcal{A}$ is traversed starting from the root node (line 5). Process p is selected from the ready list $\mathcal{L}_R$ according to the partial critical path (PCP) priority function [Ele00] (line 7). The last scheduled process r on the computation node, on which p is mapped, is extracted from the root schedule S (line 9). Process p is scheduled and its start time is recorded in the root schedule (line 10). Then, its recovery slack $s(p)$ is adjusted such that it can accomodate recovering of processes scheduled before process p on the same computation node (lines 12-13). The adjustment is performed in two steps:

1. The idle time b between process p and the last scheduled process r is calculated (line 12).
2. The recovery slack $s(p)$ of process p is changed, if the recovery slack $s(r)$ of process r substracted with the idle time b is

```
RootScheduleGeneration(A, k, N, B, M)
1  RS = ∅
2  for ∀Pi ∈ A do -- obtaining initial recovery slacks
3    s(Pi) = k × (Ci + μ)
4  end for
5  LR = {RootNode(A)}
6  while LR ≠ ∅ do
7    p = SelectProcess(LR) -- select process from the ready list
8    -- the last scheduled process on the computation node, where p is mapped
9    r = CurrentProcess(RS{M(p)})
10   ScheduleProcess(p, RS{M(p)}) -- scheduling of process p
11   -- adjusting recovery slacks
12   b = start(p) - end(r) -- calculation of the idle time r and p
13   s(p) = max{s(p), s(r) - b} -- adjusting the recovery slack of process p
14   -- schedule messages sent by process p at the end of its recovery slack s
15   ScheduleOutgoingMessages(p, s(p), RS{M(p)})
16   Remove(p, LR) -- remove p from the ready list
17   -- add successors of p to the ready list
18   for ∀Succ(p) do
19     if Succ(p) ∉ LR then Add(Succ(p), LR)
20   end for
21 end while
22 return RS
end RootScheduleGeneration
```

Figure 4.13: Generation of Root Schedules

larger than the initial slack $s_0(p)$. Otherwise, the initial slack $s_0(p)$ is preserved (line 13).

If no process is scheduled before p, the initial slack $s_0(p)$ is preserved as $s(p)$. Outgoing messages sent by process p are scheduled at the end of the recovery slack $s(p)$ (line 15).

After the adjustment of the recovery slack, process p is removed from the ready list $\mathcal{L}_R$ (line 16) and its successors are added to the list (lines 18-20). After scheduling of all the processes in the application graph $\mathcal{A}$, the algorithm returns a root

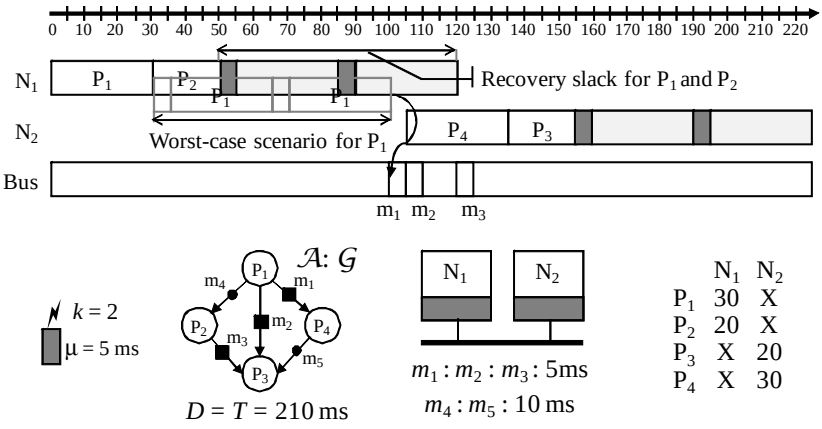


Figure 4.14: Example of a Root Schedule

schedule $\mathcal{RS}$ with start times of processes, sending times of messages, and recovery slacks (line 22).

In Figure 4.14 we present an example of a root schedule with recovery slacks. Application $\mathcal{A}$ is composed of four processes, where processes P_1 and P_2 are mapped on N_1 and processes P_3 and P_4 are mapped on N_2. Messages m_1, m_2 and m_3, to be transported on the bus, are frozen, according to the requirements of shifting-based scheduling. Processes P_1 and P_2 have start times 0 and 30 ms, respectively, and share a recovery slack of 70 ms, which is obtained as $max\{2 \times (30 + 5) - 0, 2 \times (20 + 5)\}$ (see the algorithm). Processes P_3 and P_4 have start times of 135 and 105 ms, respectively, and share a recovery slack of 70 ms. Messages m_1, m_2 and m_3 are sent at 100, 105, and 120 ms, respectively, at the end of the worst-case recovery intervals of the sender processes.

Extracting Execution Scenarios. In Figure 4.15, we show an example, where we extract one execution scenario from the root schedule of the application $\mathcal{A}$, depicted in Figure 4.14. In this execution scenario, process P_4 experiences two faults. P_4 starts at 105 ms according to the root schedule. Then, since a

fault has happened, P_4 has to be re-executed. The start time of P_4's re-execution, is obtained as 105 + 30 + 5 = 140 ms, where 30 is the worst-case execution time of P_4 and 5 is the recovery overhead μ. The re-execution P_4^1 experiences another fault and the start time of P_4's second re-execution P_4^2 is 140 + 30 + 5 = 175 ms. Process P_3 will be delayed because of the re-executions of process P_4. The current time *CRT*, at the moment when P_3 is activated, is 175 + 30 = 205 ms, which is more than 135 ms that is the schedule time of P_3 according to the root schedule. Therefore, process P_3 will be immediately executed at *CRT* = 205 ms. The application $\mathcal{A}$ will complete execution at 225 ms.

The runtime algorithm for extracting execution scenarios from the root schedule $\mathcal{RS}$ is presented in Figure 4.16. The runtime scheduler runs on each computation node $N_i \in \mathcal{N}$ and executes processes according to the order in the root schedule of node N_i until the last process in that root schedule is executed.

In the initialization phase, the current time *CRT* of the scheduler running on node N_i is set to 0 (line 1) and the first process p is extracted from the root schedule of node N_i (line 2). This process is executed according to its start time in the root schedule

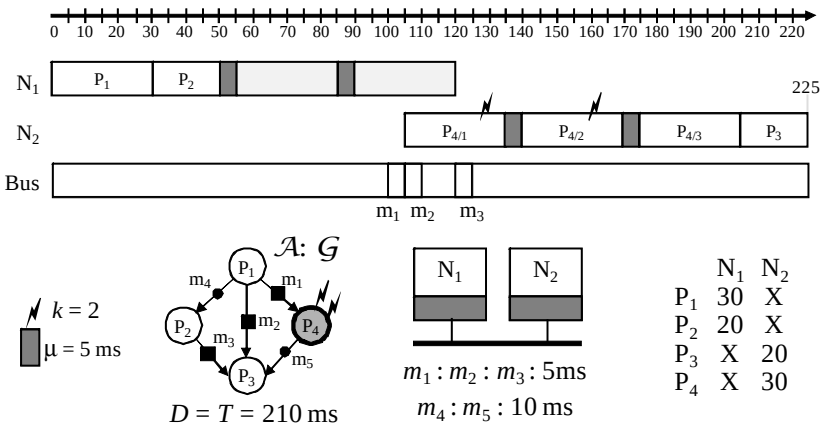


Figure 4.15: Example of an Execution Scenario

```
ExtractScenario(RS, N_i)
1  CRT = 0
2  p = GetFirstProcess(RS {N_i})
3  while p ≠ ∅ do
4    Execute(p, CRT)
5    while fault(p) do
6      Restore(p)
7      Execute(p, CRT + C_p)
8    end while
9    PlaceIntoCommBuffer(OutputMessages(p))
10   p = GetNextProcess(RS {N_i})
11 end while
end ExtractScenario
```

Figure 4.16: Extracting Execution Scenarios

(line 4). If p fails, then it is restored (line 6) and executed again with the time *shift* of its worst-case execution time C_p (line 7). It can be re-executed at most k times in the presence of k faults. When p is finally completed, its output messages are placed into the output buffer of the communication controller (line 9). The output message will be sent according to its sending times in the root schedule. After completion of process p, the next process is extracted from the root schedule of node N_i (line 10) and the algorithm continues with execution of this process.

Re-executions in the case of faults usually delay executions of the next processes in the root schedule. We have accommodated process delays into recovery slacks of the root schedule with the RootScheduleGeneration algorithm (Figure 4.13). Therefore, if process p is delayed due to re-executions of previous processes and cannot be executed at the start time pre-defined in the root schedule, it is immediately executed after been extracted, within its recovery slack (Execute function, line 7 in Figure 4.16).

4.5 Experimental Results

For the evaluation of our scheduling algorithms 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, which can happen during one execution cycle. The duration μ of the recovery 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 randomly assigned using both uniform and exponential distribution 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 Chapter 5, which we have restricted for evaluation purposes to mapping optimization with only re-execution. 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 Figure 4.9. 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 4.1 presents the average fault

Table 4.1: 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

tolerance overheads for each of the five transparency requirements. We can 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 processes with three faults, we have obtained an 86% overhead for 100% frozen messages, which is reduced to 58% for 50% frozen messages.

Table 4.2 presents the average memory[1] space per computation node (in kilobytes) required to store the schedule tables. Often, one process/message has the same start time under different conditions. Such entries into the table can be merged into a single table entry, headed by the union of the logical expressions. Thus, Table 4.2 reports the memory required after such a

Table 4.2: 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

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

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 18Kb to 4Kb. 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.

Our shifting-based scheduling (SBS), presented in Section 4.4, always preserves the same order of processes and messages in all execution scenarios and assumes that 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 Chapter 5, restricted to re-execution. We have considered that all inter-processor messages and only them are frozen in both cases. When comparing 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. However, shifting-based scheduling generates schedules for these applications in less than a quarter of a second and can produce root schedules for large graphs of 80, 100, and 120 processes with 4, 6, and 8 faults also in less than a quarter of a second. The schedule generation time does

Table 4.3: Memory Requirements (SBS), Kbytes

	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%	0.02			0.03			0.05			0.07		

not exceed 0.2 sec. even for 120 processes and 8 faults. Therefore, shifting-based scheduling can be effectively used inside design optimization, where transparency-related trade-offs are not considered.

The amount of memory needed to store root schedules is also very small as shown in Table 4.3. Moreover, due to the nature of the shifting-based scheduling algorithm, the amount of memory needed to store the root schedule does not change with the number of faults. Because of low memory requirements, shifting-based scheduling is suitable for synthesis of fault-tolerant schedules even for small microcontrollers.

4.5.1 CASE STUDY

Finally, we have considered a real-life example implementing a vehicle cruise controller (CC). The process graph that models the CC has 32 processes, and is presented in Appendix I.

The CC maintains a constant speed over 35 km/h and under 200km/h, offers an interface (buttons) to increase or decrease the reference speed, is able to resume its operation at the previous reference speed, and is suspended when the driver presses the brake pedal. The CC has been mapped on an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Braking System (ABS) and Transmission Control Module (TCM).

We have considered a deadline of 300 ms, $k = 2$ and $\mu = 2$ ms, and have obtained a fixed mapping of the CC on the computation nodes with the design optimization strategy from Chapter 5. SBS has produced an end-to-end delay of 384 ms, which is larger than the deadline. The CS approach reduces this delay to 346

ms, given that all inter-processor messages are frozen, which is also unschedulable. If we relax this transparency requirement and select 50% of the inter-processor messages as frozen, we can further reduce the delay to 274 ms, which will meet the deadline.

4.6 Conclusions

In this chapter, we have proposed two novel scheduling approaches for fault-tolerant embedded systems in the presence of multiple transient faults: conditional scheduling and shifting-based scheduling.

The main contribution of the first approach is the ability to handle performance versus transparency and memory size trade-offs. This scheduling approach generates the most efficient schedules.

The second scheduling approach handles only a fixed transparency setup, transparent recovery, where all messages on the bus have to be sent at fixed times, regardless of fault occurrences. Additionally all processes have to be executed in the same order in all execution scenarios. Even though this scheduling approach generates longer schedules, it is much faster than the conditional scheduling and requires less memory to store the generated schedule tables. These advantages make this scheduling technique suitable for microcontroller systems with strict memory constraints.

Chapter 5
Mapping and Fault Tolerance Policy Assignment

IN THIS CHAPTER we discuss mapping and fault tolerance policy assignment for hard real-time applications. For optimization of policy assignment we combine re-execution, which provides time redundancy, with replication, which provides spatial redundancy. The mapping and policy assignment optimization algorithms decide a process mapping and fault tolerance policy assignment such that the overheads due to fault tolerance are minimized. The application is scheduled using the shifting-based scheduling technique presented in Section 4.4.

5.1 Fault Tolerance Policy Assignment

In this thesis, by policy assignment we denote the decision on which fault tolerance techniques should be applied to a process. In this chapter, we will consider two techniques: re-execution and replication (see Figure 5.1).

The fault tolerance policy assignment is defined by three functions, $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$, as follows:

$\mathcal{P}$: $\mathcal{V} \rightarrow$ {*Replication*, *Re-execution*, *Replication & Re-execution*} determines whether a process is replicated, re-executed, or replicated and re-executed. When replication is used for a process P_i, we introduce several replicas into the application $\mathcal{A}$, and connect them to the predecessors and successors of P_i.

The function $\mathcal{Q}$: $\mathcal{V} \rightarrow$ N indicates the number of replicas for each process. For a certain process P_i, if $\mathcal{P}(P_i)$= *Replication*, then $\mathcal{Q}(P_i) = k$; if $\mathcal{P}(P_i)$ = *Re-execution*, then $\mathcal{Q}(P_i)$= 0; if $\mathcal{P}(P_i)$ = *Replication & Re-execution*, then $0 < \mathcal{Q}(P_i) < k$.

Let $\mathcal{V}_R$ be the set of replica processes introduced into the application. Replicas can be re-executed as well, if necessary. The function $\mathcal{R}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow$ N determines the number of re-executions for each process or replica. In Figure 5.1c, for example, we have $\mathcal{P}(P_1)$=*Replication & Re-execution*, $\mathcal{R}(P_{1(1)}) = 1$ and $\mathcal{R}(P_{1(2)}) = 0$.[1]

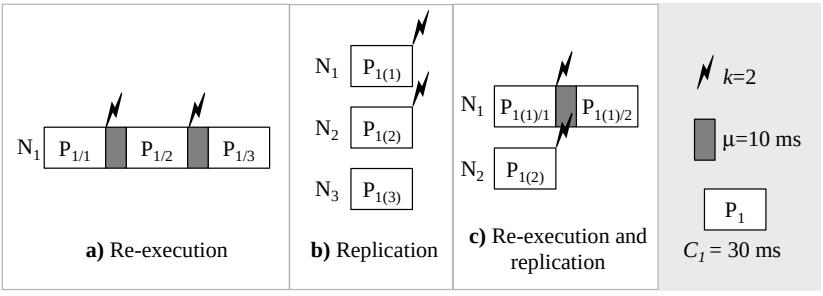


Figure 5.1: Policy Assignment: Re-execution + Replication

1. For the sake of uniformity, in the case of replication, we name the original process P_i as the first replica of process P_i, denoted with $P_{i(1)}$, see Section 2.2.4.

Each process $P_i \in \mathcal{V}$, besides its worst-case execution time C_i on each computation node, is characterized by a recovery overhead μ_i.

The mapping of a process is given by a function $\mathcal{M}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. The mapping $\mathcal{M}$ is not fixed and will have to be obtained during design optimization.

Thus, our problem formulation is as follows:

- As an input we have an application $\mathcal{A}$ given as a merged process graph (Section 3.1.1) and a system consisting of a set of nodes $\mathcal{N}$ connected to a bus B.
- The parameter k denotes the maximum number of transient faults that can appear in the system during one cycle of execution.

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

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

1. finding the fault tolerance policy assignment, given by $\mathcal{F} = <\mathcal{P}, \mathcal{Q}, \mathcal{R}>$, for the application $\mathcal{A}$;
2. deciding on a mapping $\mathcal{M}$ for each process P_i in the application $\mathcal{A}$ and for each replica in $\mathcal{V}_R$;
3. deriving the set S of schedule tables on each computation node.

The shifting-based scheduling presented in Section 4.4 with small modifications, which will be discussed in Section 5.2.2, is used to derive schedule tables for the application $\mathcal{A}$.

5.1.1 MOTIVATIONAL EXAMPLES

Let us, first, illustrate some of the issues related to policy assignment. In the example presented in Figure 5.2 we have the application $\mathcal{A}_1$ with three processes, P_1 to P_3, and an architecture with two nodes, N_1 and N_2. The worst-case execution times on each node are given in a table to the right of the architecture.

Note that N_1 is faster than N_2. We assume a single fault, thus $k = 1$. The recovery overhead μ is 10 ms. The application $\mathcal{A}_1$ has a deadline of 160 ms depicted with a thick vertical line. We have to decide which fault tolerance technique to use.

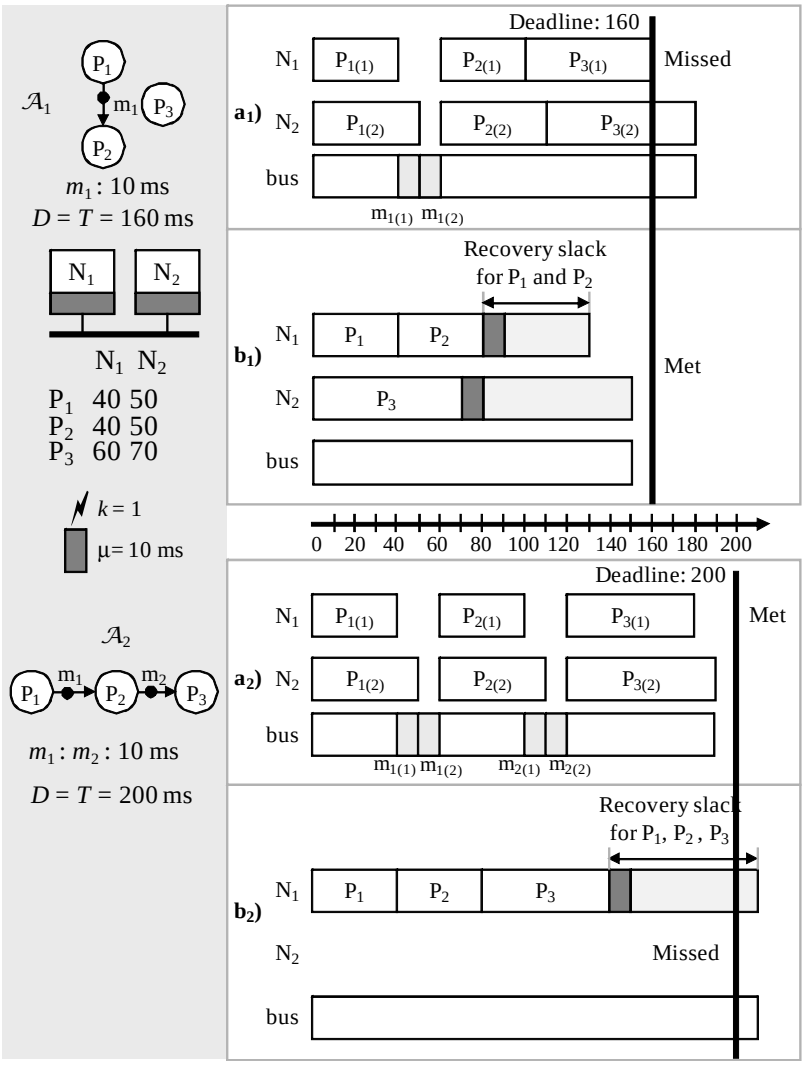


Figure 5.2: Comparison of Replication and Re-execution

In Figure 5.2 we depict the schedules for each node. Comparing the schedules in Figure $5.2a_1$ and Figure $5.2b_1$, we can observe that using only replication the deadline is missed (Figure $5.2a_1$). An additional delay is introduced with messages $m_{1(1)}$ and $m_{1(2)}$ sent from replicas $P_{1(1)}$ and $P_{1(2)}$ of process P_1, respectively, to replicas $P_{2(2)}$ and $P_{2(1)}$ of process P_2. In order to guarantee that time constraints are satisfied in the presence of faults, all re-executions of processes, which are accommodated in recovery slacks, have to finish before the deadline. Using only re-execution we are able to meet the deadline (Figure $5.2b_1$). However, if we consider a modified application $\mathcal{A}_2$ with process P_3 data dependent on P_2, the imposed deadline of 200 ms is missed in Figure $5.2b_2$ if only re-execution is used, and it is met when replication is used as in Figure $5.2a_2$.

This example shows that the particular technique to use has to be carefully adapted to the characteristics of the application and the available resources. Moreover, the best result is most likely to be obtained when both re-execution and replication are used together, some processes being re-executed, while others replicated.

Let us consider the example in Figure 5.3, where we have an application with four processes mapped on an architecture of two nodes. In Figure 5.3a all processes are re-executed, and the depicted schedule is optimal for re-execution, yet missing the deadline in the worst-case (process P_1 experiences a fault and is re-executed). However, combining re-execution with replication, as in Figure 5.3b where process P_1 is replicated, will meet the deadline even in the worst case (process P_2 is re-executed). In this case, P_2 will have to receive message $m_{1(1)}$ from replica $P_{1(1)}$ of process P_1, and process P_3 will have to receive message $m_{2(2)}$ from replica $P_{1(2)}$. Even though transmission of these messages will introduce a delay due to the inter-processor communication on the bus, this delay is compensated by the gain in performance because of replication of process P_1.

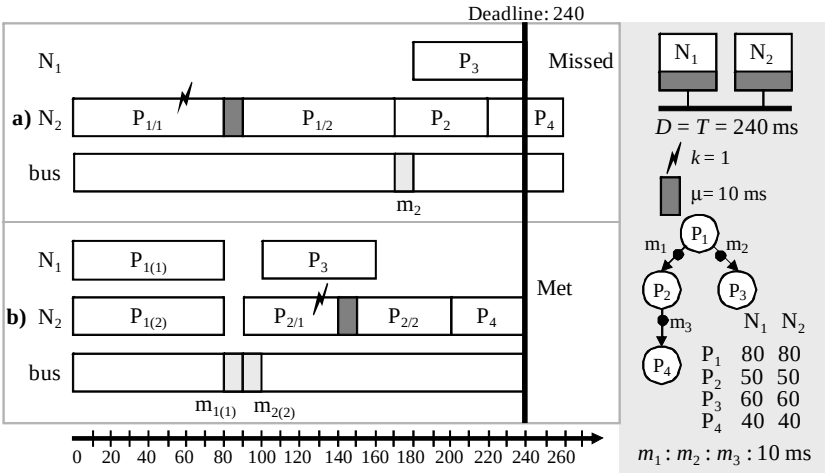


Figure 5.3: Combining Re-execution and Replication

5.2 Mapping with Fault Tolerance

In general, fault tolerance policy assignment cannot be done separately from process mapping. Consider the example in Figure 5.4. Let us suppose that we have applied a mapping algorithm without considering the fault tolerance aspects, and we have obtained the best possible mapping, depicted in Figure 5.4a, which has the shortest execution time. If we apply on top of this mapping a fault tolerance technique, for example, re-execution as in Figure 5.4b, we miss the deadline in the worst-case of re-execution of process P_3.

The actual fault tolerance policy, in this case re-execution, has to be considered during mapping of processes, and then the best mapping will be the one in Figure 5.4c, which clusters all processes on the same computation node. In this thesis, we will consider the assignment of fault tolerance policies at the same time with the mapping of processes to computation nodes in order to improve the quality of the final design.

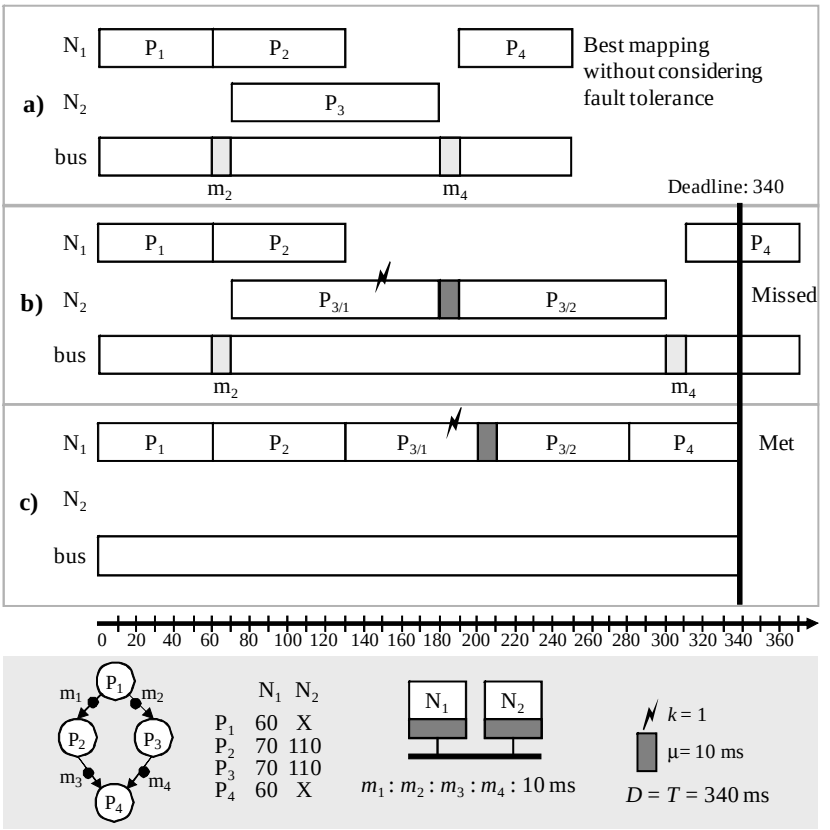


Figure 5.4: Mapping and Fault Tolerance

5.2.1 DESIGN OPTIMIZATION STRATEGY

The design problem formulated in the previous section is NP complete (both the scheduling and the mapping problems, considered separately, are already NP-complete [Gar03]). Our strategy is outlined in Figure 5.5 and has two steps:

1. In the first step (lines 1–2) we decide very quickly on an initial fault tolerance policy assignment $\mathcal{F}^0$ and mapping $\mathcal{M}^0$. The initial mapping and fault tolerance policy

```
MPAOptimizationStrategy(𝒜, 𝒩)
1  Step 1: ψ⁰ = InitialMPA(𝒜, 𝒩)
2          if S⁰ is schedulable then return ψ⁰ end if
3  Step 2: ψ = TabuSearchMPA(𝒜, 𝒩, ψ)
4          if S is schedulable then return ψ end if
5  return no_solution
end MPAOptimizationStrategy
```

Figure 5.5: Design Optimization Strategy for Fault Tolerance Policy Assignment

assignment algorithm (InitialMPA line 1 in Figure 5.5) assigns a re-execution policy to each process in the application $\mathcal{A}$ and produces a mapping that tries to balance the utilization among nodes. The application is then scheduled using the shifting-based scheduling algorithm presented in Section 4.4. If the application is schedulable the optimization strategy stops.

2. If the application is not schedulable, we use, in the second step, a tabu search-based algorithm TabuSearchMPA, presented in Section 5.2.3, that aims to improve the fault tolerance policy assignment and mapping obtained in the first step.

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

5.2.2 SCHEDULING AND REPLICATION

In Section 4.4, we presented the shifting-based scheduling algorithm with re-execution as the employed fault tolerance technique. For scheduling applications that combine re-execution and replication, this algorithm has to be slightly modified to capture properties of replica descendants, as illustrated in Figure 5.6. The notion of "ready process" will be different in the case of processes waiting inputs from replicas. In that case, a successor process P_s of replicated process P_i can be placed in the

root schedule at the earliest time moment t, at which at least one valid message $m_{i(j)}$ can arrive from a replica $P_{i(j)}$ of process P_i.[1] We also include in the set of valid messages $m_{i(j)}$ the output from replica $P_{i(j)}$ to successor P_s passed through the shared memory (if replica $P_{i(j)}$ and successor P_s are mapped on the same computation node).

Let us consider the example in Figure 5.6, where P_2 is replicated and we use a shifting-based scheduling without the above modification. In this case, P_3, the successor of P_2, is scheduled at the latest moment, when any of the messages to P_3 can arrive. Therefore, P_3 has to be placed in the schedule, as illustrated in Figure 5.6a, after message $m_{2(2)}$ from replica $P_{2(2)}$ has arrived. We should also introduce recovery slack for process P_3 for the case it experiences a fault.

However, the root schedule can be shortened by placing P_3 as in Figure 5.6b, immediately following replica $P_{2(1)}$ on N_1, if we use the updated notion of "ready process" for successors of repli-

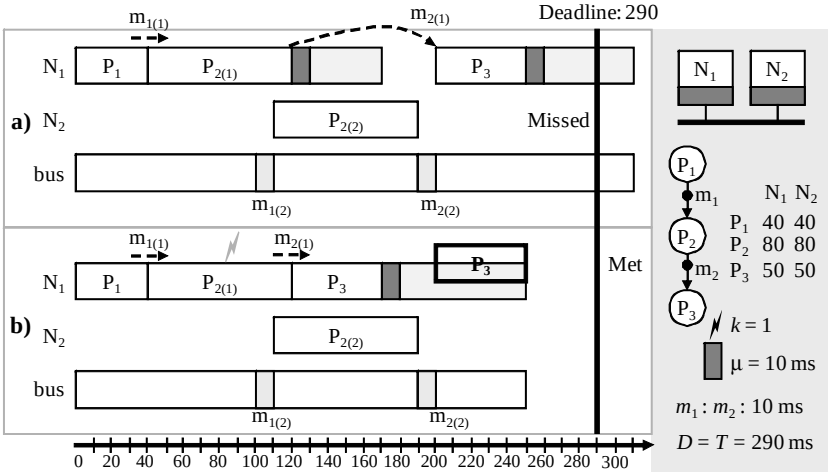


Figure 5.6: Scheduling Replica Descendants

1. We consider the original process P_i as a first replica, denoted with $P_{i(1)}$.

cated processes. In this root schedule, process P_3 will start immediately after replica $P_{2(1)}$ in the case that no fault has occurred in $P_{2(1)}$. If replica $P_{2(1)}$ fails, then, in the corresponding alternative schedule, process P_3 will be delayed until it receives message $m_{2(2)}$ from replica $P_{2(2)}$ on N_2 (shown with the thick-margin rectangle). In the root schedule, we should accommodate this delay into the recovery slack of process P_3 as shown in Figure 5.6b. Process P_3 can also experience faults. However, processes can experience at maximum k faults during one application run. In this example, P_2 and P_3 cannot be faulty at the same time because $k = 1$. Therefore, process P_3 will not need to be re-executed if it is delayed in order to receive message $m_{2(2)}$ (since, in this case, $P_{2(1)}$ has already failed). The only scenario, in which process P_3 can experience a fault, is the one where process P_3 is scheduled immediately after replica $P_{2(1)}$. In this case, however, re-execution of process P_3 is accommodated into the recovery slack. The same is true for the case if P_1 fails and, due to its re-execution, $P_{2(1)}$ and P_3 have to be delayed. As can be seen, the resulting root schedule depicted in Figure 5.6b is shorter than the one in Figure 5.6a and the application will meet its deadline.

5.2.3 OPTIMIZATION ALGORITHMS

For the optimization of the mapping and fault tolerance policy assignment we perform two steps, see Figure 5.5. The first step is a straightforward policy assignment with only re-execution, where mapping is obtained by a simple balancing of utilization of the computation nodes. If this step fails to produce a schedulable implementation, we use, in the next step, a tabu search-based optimization approach, TabuSearchMPA.

This approach investigates in each iteration all the processes on the critical path of the merged application graph $\mathcal{G}$, and use design transformations (moves) to change a design such that the critical path is reduced. Let us consider the example in

Figure 5.7, where we have an application of four processes that has to tolerate one fault, mapped on an architecture of two nodes. Let us assume that the current solution is the one depicted in Figure 5.7a. In order to generate neighbouring solutions, we perform design transformations that change the mapping of a process, and/or its fault tolerance policy. Thus, the neighbouring solutions considered by our heuristic, starting from Figure 5.7a, are those presented in Figure 5.7b–5.7e. Out of these, the solution in Figure 5.7c is the best in terms of schedule length.

One simple (*greedy*) optimization heuristic could have been to select in each iteration the best move found and apply it to modify the design. However, although a greedy approach can quickly find a reasonable solution, its disadvantage is that it can "get stuck" into a local optimum. Thus, to avoid this, we have implemented a *tabu search* algorithm [Ree93].

The tabu search algorithm, TabuSearchMPA, presented in Figure 5.8, takes as an input the merged application graph $\mathcal{G}$, the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}. The tabu search is based on a neighbourhood search technique, and thus in each iteration it generates the set of moves N^{now} that can be reached from the current solution x^{now} (line 7 in Figure 5.8). In our implementation, we only consider changing the mapping or fault tolerance policy assignment of the processes on the critical path, corresponding to the current solution, denoted with $\mathcal{CP}$ in Figure 5.8. For example, in Figure 5.7a, the critical path is formed by P_1, m_2 and P_3.

The key feature of a tabu search is that the neighbourhood solutions are modified based on a selective history of the states

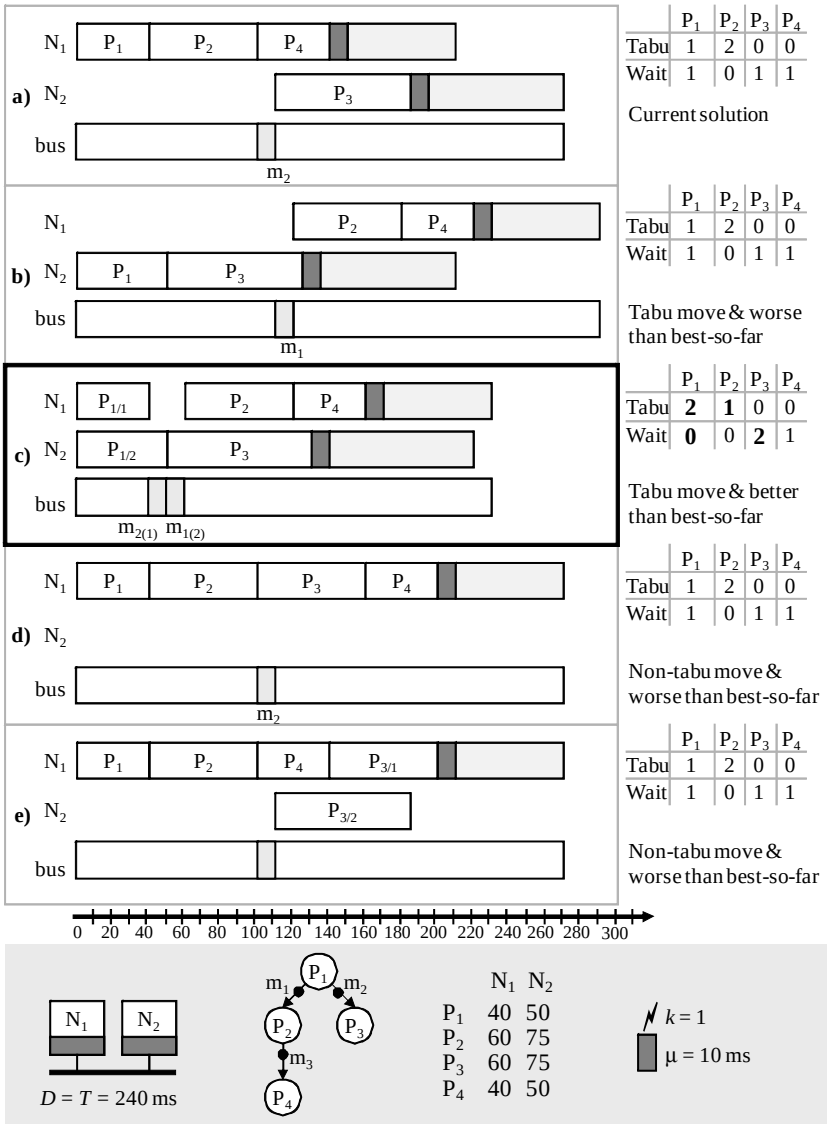


Figure 5.7: Moves and Tabu History

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

Figure 5.8: Tabu Search Algorithm for Optimization of Mapping and Fault Tolerance Policy Assignment

encountered during the search [Ree93]. The selective history is implemented in our case through the use of two tables, *Tabu* and *Wait*. Each process has an entry in these tables. If $Tabu(P_i)$ is non-zero, it means that the process is “tabu”, i.e., should not be selected for generating moves. Thus, a move will be removed

from the neighbourhood solutions if it is tabu (lines 9 and 10 of the algorithm). However, tabu moves are also accepted if they lead to solutions better than the best-so-far solution ($Cost(move(P_i)) < BestCost$, line 9). If $Wait(P_i)$ is greater than the number of processes in the graph, $|\mathcal{G}|$, the process has waited a long time and should be selected for *diversification* [Ree93], i.e., $move(P_i)$ can lead to the significantly different solution from those encountered previously during the search. In line 12 the search is diversified with moves that have waited a long time without being selected.

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

Figure 5.7 illustrates how the algorithm works. Let us consider that the current solution x^{now} is the one presented in Figure 5.7a, with the corresponding selective history presented to its right, and the best-so-far solution x^{best} being the one in Figure 5.3a. The generated solutions are presented in Figure 5.7b–5.7e. The solution (b) is removed from the set of considered solutions because it is tabu, and it is not better than x^{best}. Thus, solutions (c)–(e) are evaluated in the current iteration. Out of these, the solution in Figure 5.7c is selected, because although it is tabu, it is better than x^{best}. The table is updated as depicted to the right of Figure 5.7c in bold, and the iterations continue with solution (c) as the current solution.

5.3 Experimental Results

For the evaluation of our strategy for policy assignment and mapping we have used applications of 20, 40, 60, 80, and 100 processes (all unmapped and with no fault tolerance policy assigned) implemented on architectures consisting of 2, 3, 4, 5, and 6 nodes, respectively. We have varied the number of faults depending on the architecture size, considering 3, 4, 5, 6, and 7 faults for each architecture dimension, respectively. The recovery overhead μ has been set to 5 ms. Fifteen examples have randomly been generated for each application dimension, thus a total of 75 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 randomly assigned using both uniform and exponential distribution within the 10 to 100 ms, and 1 to 4 bytes ranges, respectively. The experiments have been performed on Sun Fire V250 computers.

We were first interested to evaluate the proposed optimization strategy in terms of overheads introduced due to fault tolerance. Hence, we have implemented each application, on its corresponding architecture, using the MPAOptimizationStrategy (MXR) strategy from Figure 5.5. In order to evaluate MXR, we have derived a reference non-fault-tolerant implementation, NFT, which ignores the fault tolerance issues. The NFT implementation has been produced as the result of an optimization similar to MXR but without any moves related to fault tolerance policy assignment. Compared to the NFT implementation thus obtained, we would like MXR to produce a fault-tolerant design with as little as possible overhead, using the same amount of hardware resources (nodes). For these experiments, we have derived the shortest schedule within an imposed time limit for optimization: 10 minutes for 20 processes, 20 for 40, 1 hour for

Table 5.1: Fault Tolerance Overheads with MXR (Compared to NFT) for Different Applications

Number of processes	*k*	% maximum	% average	% minimum
20	3	98	71	49
40	4	117	85	47
60	5	143	100	52
80	6	178	121	91
100	7	216	150	100

60, 2 hours and 20 minutes for 80 and 5 hours and 30 minutes for 100 processes.

The first results are presented in Table 5.1. Applications of 20, 40, 60, 80, and 100 processes are mapped on 2, 3, 4, 5, and 6 computation nodes, respectively. Accordingly, we change the number of faults from 3 to 7. In the three last columns, we present maximum, average and minimum time overheads introduced by MXR compared to NFT. Let δ_{MXR} and δ_{NFT} be the schedule lengths obtained using MXR and NFT. The overhead due to introduced fault tolerance is defined as $100 \times (\delta_{MXR} - \delta_{NFT}) / \delta_{NFT}$. We can see that the fault tolerance overheads grow with the application size. The MXR approach can offer fault tolerance within the constraints of the architecture at an average time overhead of approximately 100%. However, even for applications of 60 processes, there are cases where the overhead is as low as 52%.

We were also interested to evaluate our MXR approach in the case of different number of faults, while the application size and the number of computation nodes were fixed. We have considered applications with 60 processes mapped on four computation nodes, with the number k of faults being 2, 4, 6, 8, or 10. Table 5.2 shows that the time overheads due to fault tolerance increase with the number of tolerated faults. This is expected,

Table 5.2: Fault Tolerance Overheads due to MXR for Different Number of Faults in the Applications of 60 Processes Mapped on 4 Computation Nodes

k	% maximum	% average	% minimum
2	52	33	20
4	110	77	47
6	162	119	82
8	251	174	118
10	292	220	155

since we need more replicas and/or re-executions if there are more faults.

With a second set of experiments, we were interested to evaluate the quality of our MXR optimization approach. Thus, together with the MXR approach we have also evaluated two extreme approaches: MX that considers only re-execution, and MR which relies only on replication for tolerating faults. MX and MR use the same optimization approach as MRX, but, for fault tolerance, all processes are assigned only with re-execution or replication, respectively. In Figure 5.9 we present the average percentage deviations of the MX and MR from MXR in terms of overhead. We can see that by optimizing the combination of re-execution and replication, MXR performs much better compared to both MX and MR. On average, MXR is 77% and 17.6% better than MR and MX, respectively. This shows that considering re-execution at the same time with replication can lead to significant improvements.

In Figure 5.9 we have also presented a straightforward strategy SFX, which first derives a mapping without fault tolerance considerations (using MXR without fault tolerance moves) and then applies re-execution. This is a solution that can be obtained by a designer without the help of our fault tolerance optimization tools. We can see that the overheads thus obtained are very

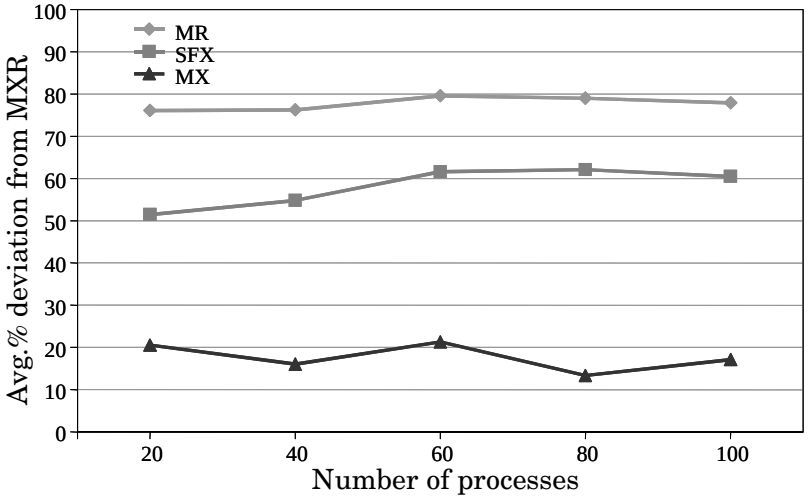


Figure 5.9: Comparing MXR with MX, MR and SFX

large compared to MXR, up to 58% on average. We can also notice that, despite the fact that both SFX and MX use only re-execution, MX is much better. This confirms that the optimization of the fault tolerance policy assignment has to be addressed at the same time with the mapping of functionality.

Finally, we have considered the real-life example implementing a vehicle cruise controller (CC), previously used to evaluate scheduling techniques in Chapter 4. We have considered the same deadline of 300 ms, the maximum number of faults $k = 2$, and a recovery overhead $\mu = 2$ ms.

In this setting, the MXR produced a schedulable fault-tolerant implementation with a worst-case system delay of 275 ms, and with an overhead compared to NFT of 65%. If only one single policy is used for fault tolerance, as in the case of MX and MR, the delay is 304 ms and 361 ms, respectively, and the deadline is missed.

5.4 Conclusions

In this chapter, we have proposed a strategy for fault tolerance policy assignment and mapping. With fault tolerance policy assignment, we can decide on which fault tolerance technique or which combination of techniques to assign to a certain process in the application. The fault tolerance technique can be re-execution, which provides time-redundancy, or active replication, which provides space-redundancy. The fault tolerance policy assignment has to be jointly optimized with process mapping. We have implemented a tabu search-based algorithm that assigns fault tolerance techniques to processes and decides on the mapping of processes, including replicas.

Chapter 6
Checkpointing-based Techniques

IN THIS CHAPTER we extend our previous techniques based on re-execution by introducing checkpoints. We, first, present our approach to optimize the number of checkpoints. Then, we extend the optimization strategy for fault tolerance policy assignment presented in Chapter 5 with checkpoint optimization.

6.1 Optimizing the Number of Checkpoints

Re-execution is a recovery technique with only one checkpoint, where a faulty process is restarted from the initial process state. In the general case of rollback recovery with checkpointing, however, a faulty process can be recovered from several checkpoints inserted into the process, which, potentially, will lead to smaller fault tolerance overheads. The number of checkpoints has a significant impact on the system performance and has to be optimized, as will be shown in this section.

6.1.1 LOCAL CHECKPOINTING OPTIMIZATION

First, we will illustrate issues of checkpoint optimization when processes are considered in isolation. In Figure 6.1 we have process P_1 with a worst-case execution time of C_1 = 50 ms. We consider a fault scenario with k = 2, the recovery overhead μ_1 equal to 15 ms, and checkpointing overhead χ_1 equal to 5 ms. The error-detection overhead α_1 is considered equal to 10 ms. Recovery, checkpointing and error-detection overheads are shown with light grey, black, and dark grey rectangles, respectively.

In the previous chapters, the error-detection overhead has been considered to be part of the worst-case execution time of processes. Throughout this chapter, however, we will explicitly consider the error-detection overhead since it directly influences the decision regarding the number of checkpoints introduced. In this chapter, we will consider equidistant checkpointing, which relies on using equal length time intervals between checkpoints, as has been discussed in Section 2.2.3.

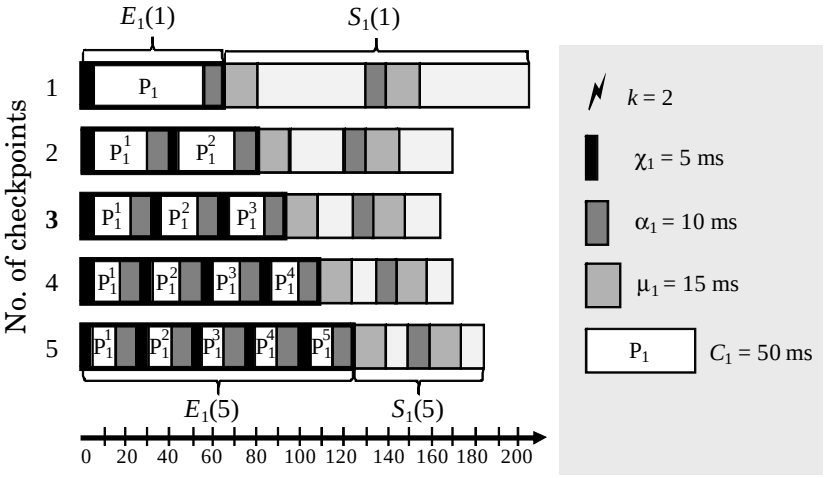


Figure 6.1: Locally Optimal Number of Checkpoints

In Figure 6.1 we depict the execution time needed for P_1 to tolerate two faults, considering from one to five checkpoints. Since P_1 has to tolerate two faults, the recovery slack S_1 has to be double the size of P_1 including the recovery overhead, as well as the error-detection overhead α_1 that has to be considered for the first re-execution of the process. Thus, for one checkpoint, the recovery slack S_1 of process P_1 is $(50 + 15) \times 2 + 10 = 140$ ms.

If two checkpoints are introduced, process P_1 will be split into two execution segments P_1^1 and P_1^2. In general, the execution segment is a part of the process execution between two checkpoints or a checkpoint and the end of the process. In the case of an error in process P_1, only the segments P_1^1 or P_1^2 have to be recovered, not the whole process, thus the recovery slack S_1 is reduced to $(50/2 + 15) \times 2 + 10 = 90$ ms.

By introducing more checkpoints, the recovery slack S_1 can be further reduced. However, there is a point over which the reduction in the recovery slack S_1 is offset by the increase in the overhead related to setting each checkpoint. We will name this overhead as a *constant checkpointing overhead* denoted as O_i for process P_i. In general, this overhead is the sum of checkpointing overhead χ_i and the error-detection overhead α_i. Because of the overhead associated with each checkpoint, the actual execution time E_1 of process P_1 is constantly increasing with the number of checkpoints (as shown with thick-margin rectangles around the process P_1 in Figure 6.1).

For process P_1 in Figure 6.1, going beyond three checkpoints will enlarge the total execution time $R_1 = S_1 + E_1$, when two faults occur.

In general, in the presence of k faults, the execution time R_i in the worst-case fault scenario of process P_i with n_i checkpoints can be obtained with the formula:

$$R_i(n_i) = E_i(n_i) + S_i(n_i) \tag{6.1}$$

$$\text{where } E_i(n_i) = C_i + n_i \times (\alpha_i + \chi_i)$$

$$\text{and } S_i(n_i) = \left(\frac{C_i}{n_i} + \mu_i\right) \times k + \alpha_i \times (k-1)$$

where $E_i(n_i)$ is the execution time of process P_i with n_i checkpoints in the case of no faults. $S_i(n_i)$ is the recovery slack of process P_i. C_i is the worst-case execution time of process P_i. $n_i \times (\alpha_i + \chi_i)$ is the overhead introduced with n_i checkpoints to the execution of process P_i. In the recovery slack $S_i(n_i)$, $C_i/n_i + \mu_i$ is the time needed to recover from a single fault, which has to be multiplied by k for recovering from k faults. The error-detection overhead α_i of process P_i has to be additionally considered in $k - 1$ recovered execution segments for detecting possible fault occurrences (except the last, k^{th}, recovery, where all k faults have already happened and been detected).

Let now n_i^0 be the optimal number of checkpoints for P_i, when P_i is considered in isolation. Punnekkat et al. [Pun97] derive a formula for n_i^0 in the context of preemptive scheduling and single fault assumption:

$$n_i^0 = \begin{cases} n_i^- = \left\lfloor \sqrt{\frac{C_i}{O_i}} \right\rfloor, \text{ if } C_i \le n_i^-(n_i^- + 1)O_i \\ n_i^+ = \left\lceil \sqrt{\frac{C_i}{O_i}} \right\rceil, \text{ if } C_i > n_i^-(n_i^- + 1)O_i \end{cases} \tag{6.2}$$

where O_i is a constant checkpointing overhead and C_i is the computation time of P_i (the worst-case execution time in our case).

We have extended formula (6.2) to consider k faults and detailed checkpointing overheads χ_i and α_i for process P_i, when process P_i is considered in isolation:

$$n_i^0 = \begin{cases} n_i^- = \left\lfloor \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rfloor, \text{ if } C_i \le n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \\ n_i^+ = \left\lceil \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rceil, \text{ if } C_i > n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \end{cases} \quad (6.3)$$

The proof of formula (6.3) can be found in Appendix II.

Formula (6.3) allows us to calculate the optimal number of checkpoints for a certain process *considered in isolation*. For example, in Figure 6.1, $n_1^0 = 3$:

$$n_1^- = \left\lfloor \sqrt{\frac{2 \times 50}{10 + 5}} \right\rfloor = 2 \rightarrow 2 \times (2 + 1)\frac{5 + 10}{2} = 45 < 50 \rightarrow n_1^0 = 3$$

6.1.2 GLOBAL CHECKPOINTING OPTIMIZATION

Calculating the number of checkpoints for each individual process will not produce a solution which is globally optimal for the whole application because processes share recovery slacks.

Let us consider the example in Figure 6.2, where we have two processes, P_1 and P_2 on a single computation node. We consider

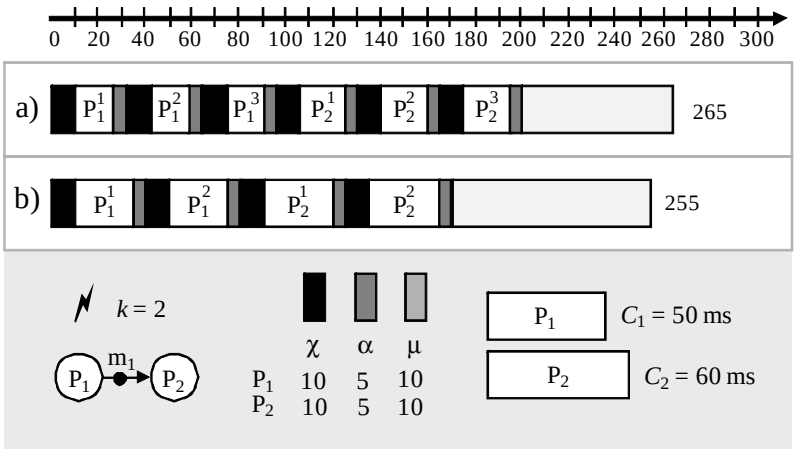


Figure 6.2: Globally Optimal Number of Checkpoints

two transient faults. The worst-case execution times and the fault tolerance overheads are depicted in the figure. In Figure 6.2a, processes P_1 and P_2 are assigned with the locally optimal number of checkpoints, $n_1^0 = 3$ and $n_2^0 = 3$, and share one recovery slack, depicted as a shaded rectangle. The size of the shared slack is equal to the individual recovery slack of process P_2 because its slack, which is $(60 / 3 + 10) \times 2 + 5 = 65$ ms, is larger than the slack of P_1, which is $(50 / 3 + 10) \times 2 + 5 = 58.3$ ms. The resulting schedule length is the sum of actual execution times of processes P_1 and P_2 and the size of their shared recovery slack of 65 ms:

$[50 + 3 \times (5 + 10)] + [60 + 3 \times (5 + 10)] + 65 = 265$ ms.

However, if we reduce the number of checkpoints to 2 for both processes, as shown in Figure 6.2b, the resulting schedule length is 255 ms, which is shorter than in the case of the locally optimal number of checkpoints. The shared recovery slack, in this case, is also equal to the individual recovery slack of process P_2 because its recovery slack, $(60 / 2 + 10) \times 2 + 5 = 85$ ms, is larger than P_1's recovery slack, $(50 / 3 + 10) \times 2 + 5 = 75$ ms. The resulting schedule length in Figure 6.2b is, hence, obtained as

$[50 + 2 \times (5 + 10)] + [60 + 2 \times (5 + 10)] + 85 = 255$ ms.

In general, slack sharing leads to a smaller number of checkpoints associated to processes, or, at a maximum, this number is the same as indicated by the local optima. This is the case because the shared recovery slack, obviously, cannot be larger than the sum of individual recovery slacks of the processes that share it. Therefore, the globally optimal number of checkpoints is always less or equal to the locally optimal number obtained with formula (6.3). Thus, formula (6.3) provides us with an upper bound on the number of checkpoints associated to individual processes. We will use this formula in order to bound the number of checkpoints explored with the optimization algorithm presented in Section 6.2.2.

6.2 Policy Assignment with Checkpointing

In this section we extend the fault tolerance policy assignment algorithm presented in Section 5.2.3 with checkpoint optimization. Here rollback recovery with checkpointing[1] will provide time redundancy, while the spatial redundancy is provided with replication, as shown in Figure 6.3. The combination of fault tolerance policies to be applied to each process is given by four functions:

- $\mathcal{P}$: $\mathcal{V} \rightarrow$ {*Replication*, *Checkpointing*, *Replication & Checkpointing*} determines whether a process is replicated, checkpointed, or replicated and checkpointed. When replication is used for a process P_i, we introduce several replicas into the application $\mathcal{A}$, and connect them to the predecessors and successors of P_i.
- The function $\mathcal{Q}$: $\mathcal{V} \rightarrow$ N indicates the number of replicas for each process. For a certain process P_i, if $\mathcal{P}(P_i)$ = *Replication*, then $\mathcal{Q}(P_i) = k$; if $\mathcal{P}(P_i)$ = *Checkpointing*, then $\mathcal{Q}(P_i) = 0$; if $\mathcal{P}(P_i)$ = *Replication & Checkpointing*, then $0 < \mathcal{Q}(P_i) < k$.
- Let $\mathcal{V}_R$ be the set of replica processes introduced into the application. Replicas can be checkpointed as well, if necessary. The function $\mathcal{R}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow$ N determines the number of

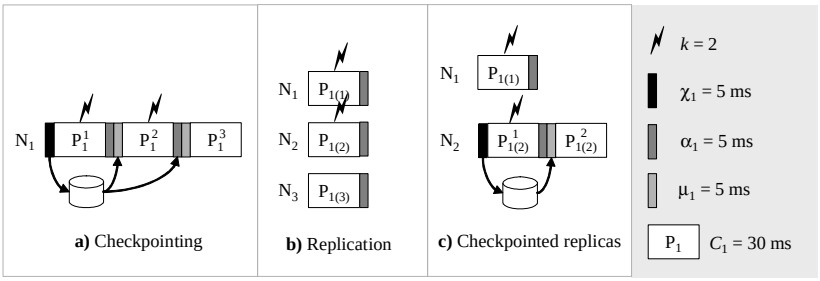


Figure 6.3: Policy Assignment: Checkpointing + Replication

1. From here and further on we will call the rollback recovery with checkpointing shortly *checkpointing*.

recoveries for each process or replica. In Figure 6.3c, for example, we have $\mathcal{P}(P_1)$ = *Replication & Checkpointing*, $\mathcal{R}(P_{1(1)}) = 0$ and $\mathcal{R}(P_{1(2)}) = 1$.

- The fourth function $\mathcal{X}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathbb{N}$ decides the number of checkpoints to be applied to processes in the application and the replicas in $\mathcal{V}_R$. We consider equidistant checkpointing, thus the checkpoints are equally distributed throughout the execution time of the process. If process $P_i \in \mathcal{V}$ or replica $P_{i(j)} \in \mathcal{V}_R$ is not checkpointed, then we have $\mathcal{X}(P_i) = 0$ or $\mathcal{X}(P_{i(j)}) = 0$, respectively, which is the case if the recovery is not used at all for this particular process or replica.

Each process $P_i \in \mathcal{V}$, besides its worst execution time C_i for each computation node, is characterized by an error detection overhead α_i, a recovery overhead μ_i, and checkpointing overhead χ_i.

The mapping of a process in the application is given by a function $\mathcal{M}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. The mapping $\mathcal{M}$ is not fixed and will have to be obtained during design optimization.

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

- As an input we have an application $\mathcal{A}$ given as a merged process graph (Section 3.1.1) and a system consisting of a set of nodes $\mathcal{N}$ connected to a bus B.
- The parameter k denotes the maximal number of transient faults that can appear in the system during one cycle of execution.

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

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

1. finding a fault tolerance policy assignment, given by $\mathcal{F} = <\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{X}>$, for each process P_i in the application $\mathcal{A}$; this also

includes the decision on the number of checkpoints $\mathcal{X}$ for each process P_i in the application $\mathcal{A}$ and each replica in $\mathcal{V}_R$;
2. deciding on a mapping $\mathcal{M}$ for each process P_i in the application $\mathcal{A}$;
3. deciding on a mapping $\mathcal{M}$ for each replica in $\mathcal{V}_R$;
4. deriving the set S of schedule tables on each computation node.

We will discuss policy assignment based on transparent recovery with replication, where all messages on the bus are set to be frozen, except those that are sent by replica processes. The shifting-based scheduling presented in Section 4.4 with small modifications, which have been discussed in Section 5.2.2, is used to derive schedule tables for the application $\mathcal{A}$. We calculate recovery slacks in the root schedule and introduce checkpointing overheads as discussed in Section 6.1.

6.2.1 OPTIMIZATION STRATEGY

The design problem formulated in the beginning of this section is NP-complete (both the scheduling and the mapping problems, considered separately, are already NP-complete [Gar03]). Therefore, our strategy is to utilize a heuristic and divide the problem into several, more manageable, subproblems. We will adapt our design optimization strategy from Chapter 5 (presented in Section 5.2.1), which combines re-execution and replication, to capture checkpointing optimization. Our optimization strategy with checkpointing optimization, based on the strategy in Section 5.2.1, is outlined in Figure 6.4. The strategy produces the configuration ψ leading to a schedulable fault-tolerant application and also has two steps:

1. In the first step (lines 1–2) we quickly decide on an initial fault tolerance policy assignment and an initial mapping. The initial mapping and fault tolerance policy assignment algorithm (InitialMPAChk line 1 in Figure 6.4) assigns a check-

```
MPAOptimizationStrategyChk(A, N)
1     Step 1:  ψ0 = InitialMPAChk(A, N)
2              if S0 is schedulable then return ψ0 end if
3     Step 2:  ψ = TabuSearchMPAChk(A, N, ψ0)
4              if S is schedulable then return ψ end if
5     return no_solution
end MPAOptimizationStrategyChk
```

Figure 6.4: Design Optimization Strategy for Fault Tolerance Policy Assignment with Checkpointing

pointing policy with a locally optimal number of checkpoints (using the equation (6.3)) to each process in the application $\mathcal{A}$ and produces a mapping that tries to balance the utilization among nodes and buses. The application is then scheduled using the shifting-based scheduling algorithm (see Section 4.4). If the application is schedulable the optimization strategy stops.

2. If the application is not schedulable, we use, in the second step, a tabu search-based algorithm, TabuSearchMPAChk (line 3), discussed in the next section.

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

6.2.2 OPTIMIZATION ALGORITHMS

For deciding the mapping and fault tolerance policy assignment with checkpointing we use a tabu search based heuristic approach, TabuSearchMPAChk, which is an adaptation of the TabuSearchMPA algorithm presented in Section 5.2.3. In addition to mapping and fault tolerance policy assignment, TabuSearchMPAChk will handle checkpoint distribution.

TabuSearchMPAChk uses design transformations (moves) to change a design such that the end-to-end delay of the root sched-

ule is reduced. In order to generate neighboring solutions, we perform the following types of transformations:

- changing the mapping of a process;
- changing the combination of fault tolerance policies for a process;
- changing the number of checkpoints used for a process.

The algorithm takes as an input the application graph $\mathcal{G}$, the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}. In each iteration of the tabu search algorithm it generates the set of moves N^{now} that can be performed from the current solution x^{now}. The cost function to be minimized by the tabu search is the end-to-end delay of the root schedule produced by the list scheduling algorithm. In order to reduce the huge design space, in our implementation, we only consider changing the mapping or fault tolerance policy of the processes on the critical path corresponding to the current solution.

Moreover, we also try to eliminate moves that change the number of checkpoints if it is clear that they do not lead to better results. Consider the example in Figure 6.5 where we have four processes, P_1 to P_4 mapped on two nodes, N_1 and N_2. The worst-case execution times of processes and their fault tolerance overheads are also given in the figure, and we have to tolerate at most two faults. The number of checkpoints calculated using the formula (6.3) are: $n_1^0 = 2$, $n_2^0 = 2$, $n_3^0 = 1$ and $n_4^0 = 3$, which are upper bounds on the number of checkpoints. Let us assume that our current solution is the one depicted in Figure 6.5a, where we have $\mathcal{X}(P_1) = 2$, $\mathcal{X}(P_2) = 1$, $\mathcal{X}(P_3) = 1$ and $\mathcal{X}(P_4) = 2$. Given a process P_i, with a current number of checkpoints $\mathcal{X}(P_i)$, our tabu search approach will generate moves with all possible checkpoints starting from 1 up to n_i^0. Thus, starting from the solution depicted in Figure 6.5a, we can have the following moves that modify the number of checkpoints: (1) decrease the number of checkpoints for P_1 to 1; (2) increase the number of checkpoints for P_2 to 2; (3) increase the number of checkpoints for P_4 to 3; (4)

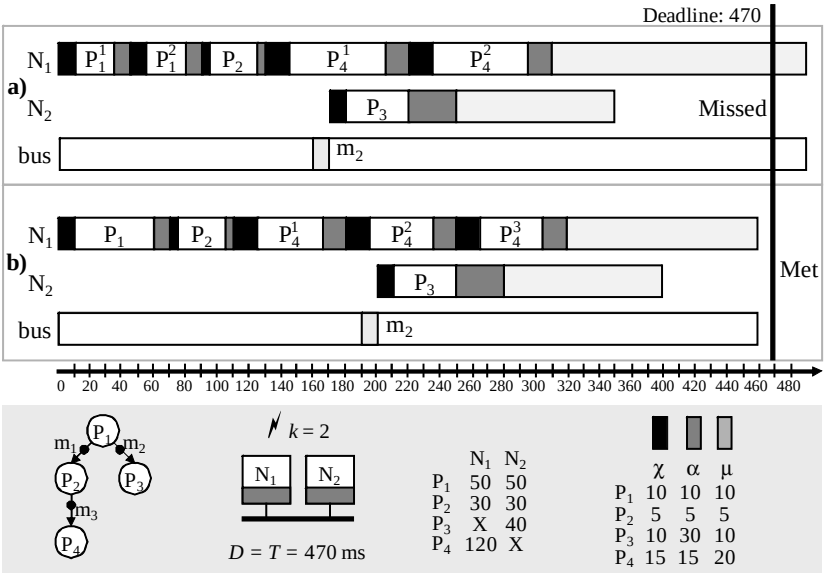


Figure 6.5: Restricting the Moves for Setting the Number of Checkpoints

decrease the number of checkpoints for P_4 to 1. Moves (1) and (3) will lead to the optimal number of checkpoints depicted in Figure 6.5b.

In order to reduce optimization time, our heuristic will not try moves (2) and (4), since they cannot lead to a shorter critical path, and, thus, a better root schedule. Regarding move (2), by increasing the number of checkpoints for P_2 we can reduce its recovery slack. However, P_2 shares its recovery slack with P_1 and segments of P_4, which have a larger execution time, and thus even if the necessary recovery slack for P_2 is reduced, it will not affect the size of the shared slack (and implicitly, of the root schedule) which is given by the largest process (or process segment) that shares the slack. Regarding move (4), we notice that by decreasing for P_4 the number of checkpoints to 1, we increase the recovery slack, which, in turn, increases the length of the root schedule.

The termination conditions and other aspects related to mapping and policy assignment of this tabu search algorithm follow the original algorithm TabuSearchMPA for mapping and policy assignment (without checkpoint optimization) presented in Section 5.2.3 of Chapter 5.

6.3 Experimental Results

For the evaluation of our design strategy with checkpointing we have used applications of 20, 40, 60, 80, and 100 processes (all unmapped and with no fault tolerance policy assigned) implemented on architectures consisting of 3, 4, 5, 6, and 7 nodes, respectively. We have varied the number of faults depending on the architecture size, considering 4, 5, 6, 7, and 8 faults for each architecture dimension, respectively. The recovery overhead μ has been set to 5 ms. We have also varied the fault tolerance overheads (checkpointing and error-detection) for each process, from 1% of its worst-case execution time up to 30%. Fifteen examples have been randomly generated for each application dimension, thus a total of 75 applications were used for experimental evaluation. The experiments have been performed on Sun Fire V250 computers.

We were interested to evaluate the quality of our optimization strategy given in Figure 6.4, with multiple checkpoints and replication (MCR). For evaluation of MCR, we have considered two other approaches: (1) MC that considers global checkpointing but without replication, and (2) MC0, similar to MC but where the number of checkpoints is fixed based on the formula (6.3), updated from [Pun97]. We have compared the quality of MCR to MC0 and MC. In Figures 6.6-6.8 we show the average percentage deviation of overheads obtained with MCR and MC from the baseline represented by MC0 (larger deviation means smaller overhead). From Figures 6.6-6.8 we can see that by optimizing the combination of checkpointing and replication MCR performs

much better compared to MC and MC0. This shows that considering checkpointing at the same time with replication can lead to significant improvements. Moreover, by considering the global optimization of the number of checkpoints, with MC, significant improvements can be gained over MC0 (which computes the optimal number of checkpoints for each process in isolation).

In Figure 6.6 we consider 4 computation nodes, 3 faults, and vary the application size from 40 to 100 processes. As the amount of available resources per application decreases, the improvement due to replication (part of MCR) will diminish, leading to a result comparable to MC.

In Figure 6.7, we were interested to evaluate our MCR approach in case the constant checkpointing overheads O (i.e., $\chi+\alpha$) associated to processes are varied. We have considered applications with 40 processes mapped on four computation

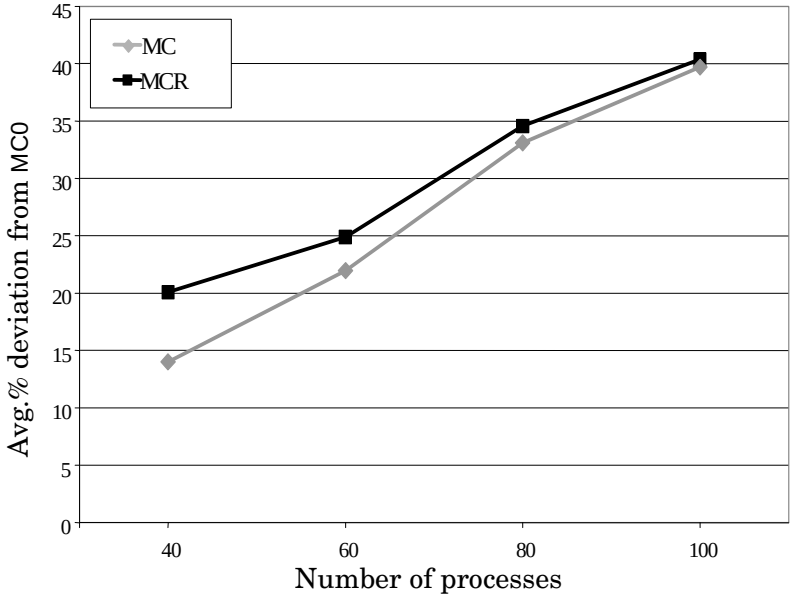


Figure 6.6: Deviation of MC and MCR from MC0 with Varying Application Size

nodes, and we have varied the constant checkpointing overhead from 2% of the worst-case execution time of a process up to 60%. We can see that, as the amount of checkpointing overheads increases, our optimization approaches are able to find increasingly better quality solutions compared to MC0.

We have also evaluated the MCR and MC approaches with increasing the maximum number of transient faults to be tolerated. We have considered applications with 40 processes mapped on 4 computation nodes, and varied k from 2 to 6, see Figure 6.8. As the number of faults increases, the improvement achieved over MC0 will stabilize to about 10% improvement (e.g., for $k = 10$, not shown in the figure, the improvement due to MC is 8%, while MCR improves with 10%).

Finally, we have considered the real-life example implementing a vehicle cruise controller (CC), used to evaluate scheduling

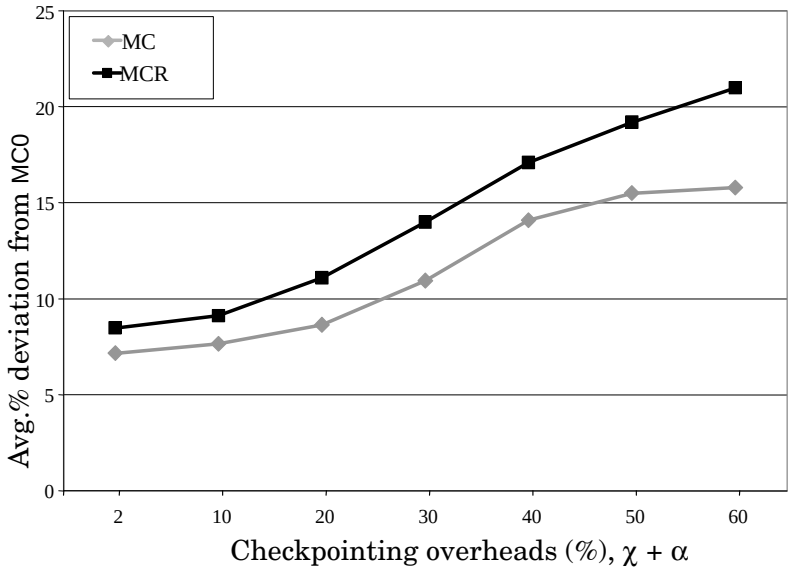


Figure 6.7: Deviation of MC and MCR from MC0 with Varying Checkpointing Overheads

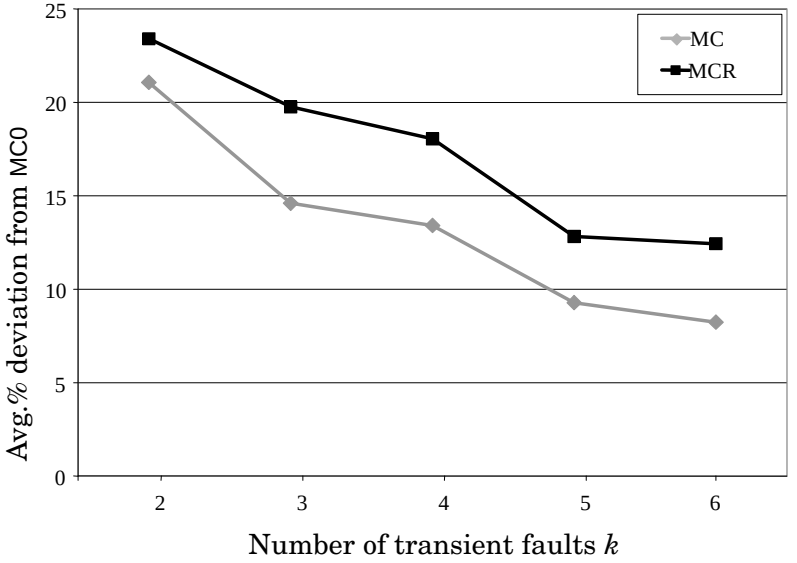


Figure 6.8: Deviation of MC and MCR from MC0 with Varying Number of Transient Faults

techniques in Chapter 4. We have considered a deadline of 300 ms, $k = 2$ faults and the constant checkpointing overheads are 10% of the worst-case execution time of the processes.

In this setting, the MCR has produced a schedulable fault-tolerant implementation with a worst-case system delay of 265 ms, and with an overhead, compared to NFT (which produces a non-fault-tolerant schedule of length 157 ms), of 69%. If we globally optimize the number of checkpoints (similarly to MCR but without considering the alternative of replication) using MC we obtain a schedulable implementation with a delay of 295 ms, compared to 319 ms produced by MC0, which is larger than the deadline. If replication only is used, in the case of MR, the delay is 369 ms, which, again, is greater than the deadline.

6.4 Conclusions

In this chapter we have addressed the problem of checkpoint optimization. First, we have discussed issues related to local optimization of the number of checkpoints. Second, we have shown that global optimization of checkpoint distribution significantly outperforms the local optimization strategy. We have extended the fault tolerance policy assignment and mapping optimization strategy presented in Chapter 5 with a global optimization of checkpoint distribution, and have shown its efficiency by experimental results.

PART III
Mixed Soft and Hard Real-Time Systems

Chapter 7
Value-based Scheduling for Monoprocessor Systems

IN THIS PART OF THE THESIS we will consider that safety critical applications are composed of soft and hard real-time processes. The hard processes in the application are critical and must always complete on time. A soft process can complete after its deadline and its completion time is associated with a value function that characterizes its contribution to the quality-of-service or *utility* of the application.

We are interested to guarantee the deadlines for the hard processes even in the presence of transient faults, while maximizing the overall utility. We will propose a quasi-static scheduling strategy, where a set of schedules is synthesized off-line and, at run time, the scheduler will select the appropriate schedule based on the occurrence of faults and the actual execution times of the processes.

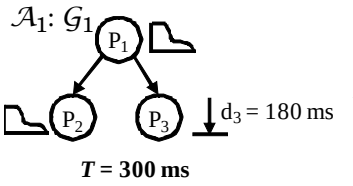
In this chapter we will propose an approach to the synthesis of fault-tolerant schedules for *monoprocessor* embedded systems with mixed soft and hard real-time constraints. We will employ process re-execution to recover from multiple transient faults.

Although in this chapter we focus on various aspects of value-based scheduling in the simplified context of monoprocessor embedded systems, we will show how our scheduling approach can be extended towards distributed embedded systems in Chapter 8.

7.1 Utility and Dropping

The processes of an application are either hard or soft, as discussed in Section 3.1.2. Hard processes are mandatory while the execution of soft processes is optional. In Figure 7.1 processes P_1 and P_2 are soft, while process P_3 is hard. Each soft process P_i is assigned with a utility function $U_i(t)$, which is any non-increasing monotonic function of the completion time of a process, as discussed in Section 3.1.6. Figure 7.2a illustrates a utility function $U_a(t)$ that is assigned to a soft process P_a. If P_a completes its execution at 60 ms, its utility would equal to 20, as illustrated in Figure 7.2a.

The overall utility of an application is the sum of individual utilities produced by the soft processes. The utility of the application depicted in Figure 7.2b, which is composed of two soft processes, P_b and P_c, is 25, in the case that P_b completes at 50 ms and P_c at 110 ms, giving utilities 15 and 10, respectively. Note that hard processes are not associated with utility functions but it has to be guaranteed that, under any circumstances, they are executed and meet their deadlines. For application $\mathcal{A}_1$,



	BCET	AET	WCET
P_1	20	50	80
P_2	30	50	70
P_3	40	60	80

Figure 7.1: Application Example with Soft and Hard Processes

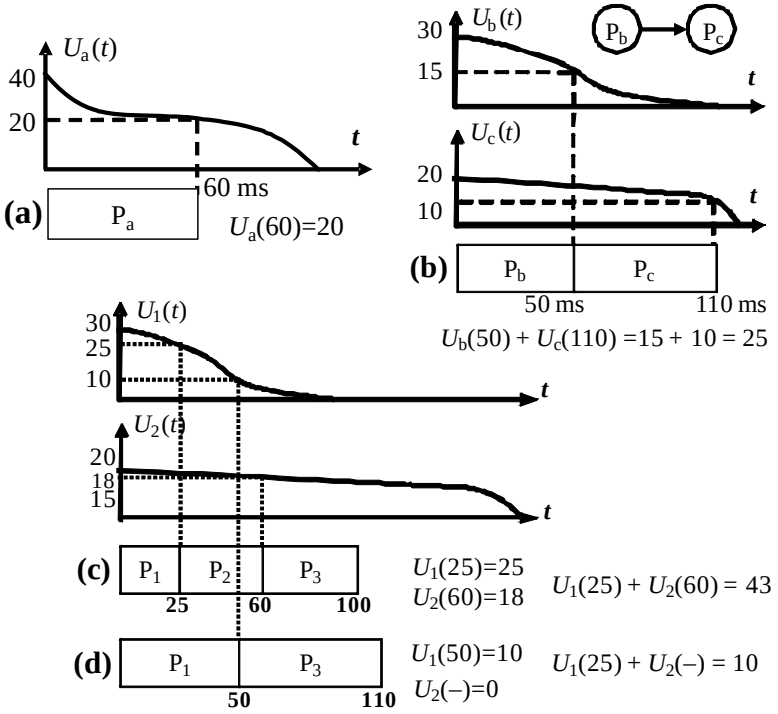


Figure 7.2: Utility Functions and Dropping

for example, the overall utility in the execution scenario in Figure 7.2c will be $U = U_1(25) + U_2(60) = 25 + 18 = 43$ (i.e., we do not account for hard process P_3 during the computation of the overall utility).

For a soft process P_i we have the option not to start it at all, and we say that we "drop" the process, and thus its utility will be 0, i.e., $U_i(–) = 0$. In the execution scenario in Figure 7.2d, for example, we drop process P_2 of application $\mathcal{A}_1$. Thus, process P_1 completes at 50 ms and process P_3 at 110 ms, which gives the total utility $U = U_1(50) + U_2(–) = 10 + 0 = 10$.

If P_i is dropped and is supposed to produce an input for another process P_j, we assume that P_j will use an input value from a previous execution cycle, i.e., a "stale" value, as dis-

cussed in Section 3.1.6. This can be the case in control applications, where a control loop executes periodically and will use values from previous runs if new ones are not available. To capture the degradation of quality that might be caused by using stale values, we update our utility model of a process P_i to $U_i^*(t) = \sigma_i \times U_i(t)$, as discussed in Section 3.1.6, where σ_i represents the stale value coefficient. σ_i captures the degradation of utility that occurs due to dropping of processes, and is obtained according to an application-specific rule $\mathcal{R}$. We will use the rule from Section 3.1.6[1], according to which, if P_i reuses stale inputs from one of its predecessors, the stale value coefficient is:

$$\sigma_i = \frac{1 + \sum_{P_j \in DP(P_i)} \sigma_j}{1 + |DP(P_i)|}$$

where $DP(P_i)$ is the set of P_i's direct predecessors. Note that we add "1" to the denominator and the dividend to account for P_i itself. The intuition behind this formula is that the impact of a stale value on P_i is in inverse proportion to the number of its inputs.

Let us illustrate the above rule on an additional example.[2] Suppose that soft process P_3 has two predecessors, P_1 and P_2. If P_1 is dropped while P_2 and P_3 are completed successfully, then, according to the above formula, $\sigma_3 = (1 + 0 + 1)/(1 + 2) = {}^2/_3$. Hence, $U_3^*(t) = {}^2/_3 \times U_3(t)$. The use of a stale value will propagate through the application. For example, if soft process P_4 is the only successor of P_3 and is completed, then $\sigma_4 = (1 + {}^2/_3) / (1 + 1) = {}^5/_6$. Hence, $U_4^*(t) = {}^5/_6 \times U_4(t)$.

---

1. However, our approach can be used with any other service degradation rule, different from the one presented here.
2. In Section 3.1.6 we have already provided a simplified example for computing utilities with service degradation rules.

Dropping might be necessary in order to meet deadlines of hard processes, or to increase the overall system utility (e.g. by allowing other, potentially higher-value, soft processes to complete).

7.2 Single Schedule vs. Schedule Tree

The goal of our scheduling strategy is to guarantee meeting the deadlines for hard processes, even in the case of faults, and to maximize the overall utility for soft processes. In addition, the utility of the no-fault scenario must not be compromised when building the fault-tolerant schedule, since the no-fault scenario is the most likely to happen.

In our scheduling strategy, we adapt the scheduling strategy for hard processes, which, as proposed in Chapter 4, uses a "recovery slack" in order to accommodate the time needed for re-executions in the case of faults. For each process P_i we assign a slack of length equal to $(t_i^w + \mu) \times f$, where f is the number of faults to tolerate, t_i^w is the worst-case execution time of process P_i, and μ is the recovery overhead. The slack is shared by several processes in order to reduce the time allocated for recovering from faults. We will refer to such a fault-tolerant schedule with recovery slacks as an f-schedule.

Let us illustrate how the value-based scheduling would work for application $\mathcal{A}_2$ in Figure 7.3 if only a single f-schedule is permitted. The application has to tolerate $k = 1$ faults and the recovery overhead μ is 10 ms for all processes. There are two possible orderings of processes: schedule S_1, "P_1, P_2, P_3" and schedule S_2, "P_1, P_3, P_2", for which the execution scenarios in the average non-fault case are shown in Figure 7.3b_1-b_2. With a recovery slack of 70 ms, P_1 would meet the deadline in both of them in the worst case and both schedules would complete before the period $T = 300$ ms. With our scheduling approach (with a single f-schedule) we have to decide off-line, which schedule to use. In the

average no-fault execution case that follows schedule S_1, process P_2 completes at 100 ms and process P_3 completes at 160 ms. The overall utility in the average case is $U = U_2(100) + U_3(160) = 20$

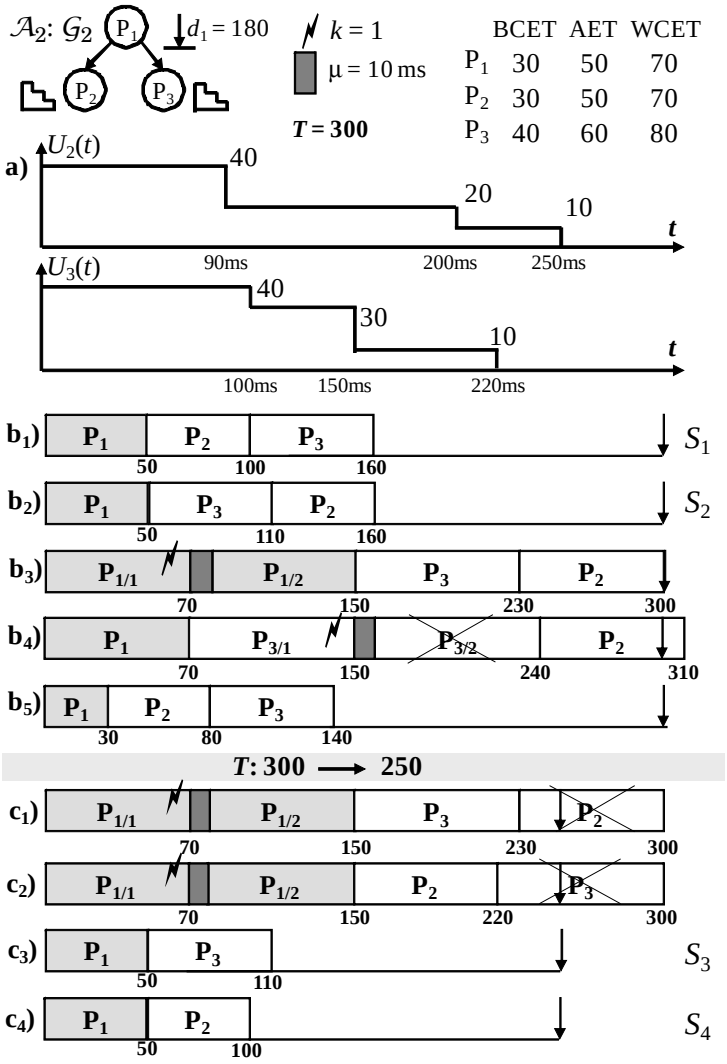


Figure 7.3: Scheduling Decisions for a Single Schedule

+ 10 = 30. In the average case[1] for S_2, process P_3 completes at 110 and P_2 completes at 160, which results in the overall utility $U = U_3(110) + U_2(160) = 40 + 20 = 60$. Thus, S_2 is better than S_1 on average and is, hence, preferred. However, if P_1 will finish sooner, as shown in Figure 7.3b$_5$, the ordering of S_1 is preferable, since it leads to a utility of $U = U_2(80) + U_3(140) = 40 + 30 = 70$, while the utility of S_2 would be only 60.

Hard processes have to be always executed and have to tolerate all k faults. Since soft processes can be dropped, this means that we do not have to re-execute them after a fault if their re-executions affect the deadlines of hard processes, lead to exceeding the period T, or if their re-executions reduce the overall utility. In Figure 7.3b$_4$, execution of process P_2 in the worst-case cannot complete within period T. Hence, process P_3 should not be re-executed. Moreover, in this example, dropping of $P_{3/2}$ is better for utility. If P_2 is executed instead of $P_{3/2}$, we get a utility of 10 even in the worst-case and may get utility of 20 if the execution of P_2 takes less time, while re-execution $P_{3/2}$ would lead to 0 utility.

In Figure 7.3c, we reduce the period T to 250 ms for illustrative purposes. In the worst case, if process P_1 is affected by a fault and all processes are executed with their worst-case execution times, as shown in Figure 7.3c$_1$, schedule S_2 will not complete within T. Neither will schedule S_1 do in Figure 7.3c$_2$. Since hard process P_1 has to be re-executed in the case of faults, the only option is to drop one of the soft processes, either P_2 or P_3. The resulting schedules S_3: "P_1, P_3" and S_4: "P_1, P_2" are depicted in Figure 7.3c$_3$ and Figure 7.3c$_4$, respectively. The average utility of S_3, $U = U_3(110) = 30$, is higher than the average utility of S_4, $U = U_2(100) = 20$. Hence, S_3 will be chosen.

The problem with a single f-schedule is that, although the faults are tolerated, the application cannot adapt well to a par-

1. We will refer to the average no-fault execution scenario case that follows a certain schedule S_i as the *average case* for this schedule.

ticular execution scenario, which happens online, and would always use the same pre-calculate single *f*-schedule. The overall utility cannot be, thus, improved even if this, potentially, is possible. To overcome the limitations of a single schedule, we will generate a *tree* of schedules. The main idea of the *schedule tree* is to generate off-line a set of schedules, each adapted to different situations that can happen online, such that the utility is maximized. These schedules will be available to a runtime scheduler, which will switch to the best one (the schedule that guarantees the hard deadlines and maximizes utility) depending on the occurrence of faults and on finishing times t_c of processes.

In the schedule tree, each node corresponds to a schedule, and each edge is the switching that will be performed if the condition on the edge becomes true during execution. The schedule tree in Figure 7.4 corresponds to the application $\mathcal{A}_2$ in Figure 7.3. We will use the utility functions depicted in Figure 7.3a. The schedule tree is constructed for the case $k = 1$ and contains 12 nodes. We group the nodes into 4 groups. Each schedule is denoted with S_i^j, where j stands for the group number. Group 1 corresponds to the no-fault scenario. Groups 2, 3 and 4 correspond to a set of schedules in the case of faults affecting processes P_1, P_2, and P_3, respectively. The schedules for the group 1 are presented in Figure 7.4b. The scheduler starts with the schedule S_1^1. If process P_1 completes after time 40, i.e., $t_c(P_1) > 40$, the scheduler switches to schedule S_2^1, which will produce a higher utility. If a fault happens in process P_1, the scheduler will switch to schedule S_1^2 that contains the re-execution $P_{1/2}$ of process P_1. Here we switch not because of utility maximization, but because of fault tolerance. The schedules for group 2 are depicted in Figure 7.4c. If the re-execution $P_{1/2}$ completes between 90 and 100 ms, i.e., $90 < t_c(P_{1/2}) < 100$, the scheduler switches from S_1^2 to S_2^2, which gives higher utility, and, if the re-execution completes after 100, i.e., $t_c(P_{1/2}) > 100$, it switches to S_3^2 in order to satisfy timing constraints. Schedule S_3^2 represents the situation illustrated in Figure 7.3c$_2$, where process P_3 had to be dropped. Otherwise,

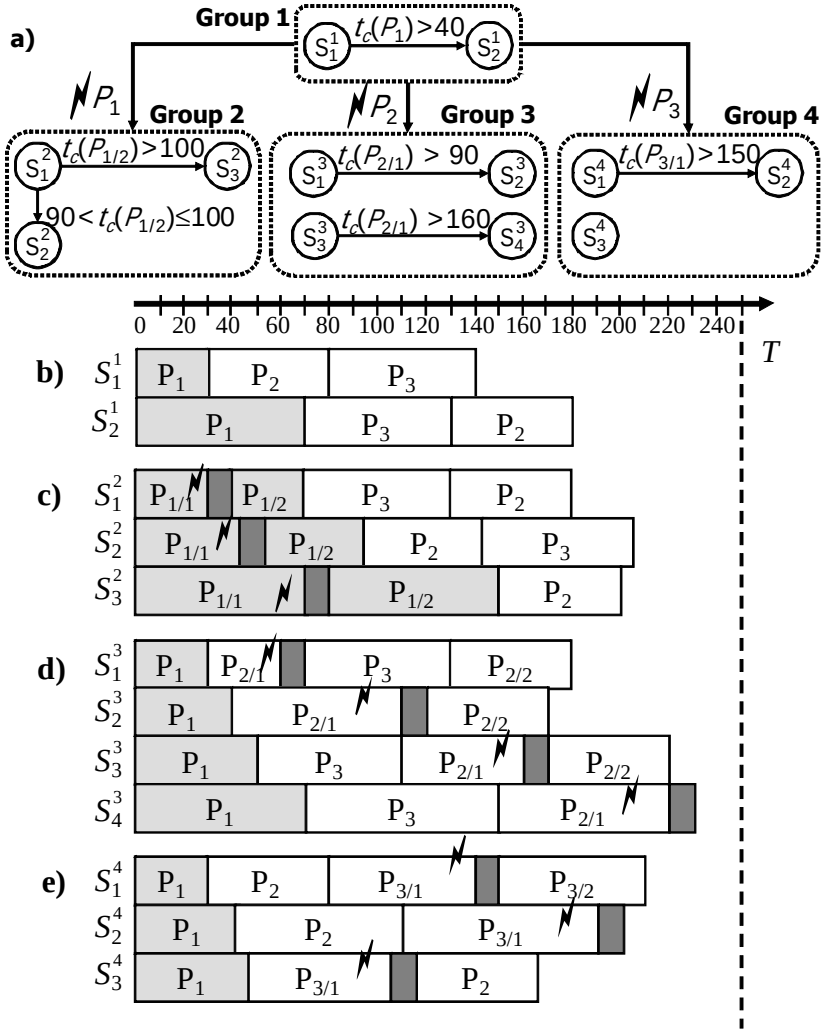


Figure 7.4: A Schedule Tree

execution of process P_3 will exceed the period T. Note that we choose to drop P_3, not P_2, because this gives a higher utility value.

The generation of a complete schedule tree with all the necessary schedules that captures different completion times of processes is practically infeasible for large applications. The number of fault scenarios grows exponentially with the number of faults and the number of processes, as has been discussed in Chapter 4. In addition, in each fault scenario, the processes may complete at different time moments. The combination of different completion times is also growing exponentially [Cor04b]. Thus, the main challenge of *schedule tree generation* for fault-tolerant systems with utility maximization is to generate an affordable number of schedules, which are able to produce a high utility.

7.3 Problem Formulation

In this section we present our problem formulation for value-based monoprocessor scheduling.

As an input to our problem we get an application $\mathcal{A}$, represented as a set of directed, acyclic graphs merged into a single hypergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, composed of soft real-time and hard real-time processes. Soft processes are assigned with utility functions $U_i(t)$ and hard processes with hard deadlines d_i, as presented in Section 3.1.6. Application $\mathcal{A}$ runs with a period T on a single computation node. The maximum number k of transient faults and the recovery overhead μ are given. We know the best and worst-case execution times for each process, as presented in Section 3.1.2. The execution time distributions for all processes are also given.

As an output, we have to obtain a schedule tree that maximizes the total utility U produced by the application and satisfies all hard deadlines in the worst case. The schedules in the tree are generated such that the utility is maximized in the case that processes execute with the expected execution times.

Schedules must be generated so that the utility is maximized with preference to the *more probable scenario.* The no-fault sce-

nario is the most likely to happen, and scenarios with less faults are more likely than those with more faults. This means that schedules for $f + 1$ faults should not compromise schedules for f faults.

7.4 Scheduling Strategy and Algorithms

Due to complexity, in our scheduling approach, which solves the scheduling problem presented in Section 7.3, we restrict the number of schedules that are part of the schedule tree. Our schedule tree generation strategy for fault tolerance is presented in Figure 7.5. We are interested in determining the best M schedules that will guarantee the hard deadlines (even in the case of faults) and maximize the overall utility. Thus, the function returns either a fault-tolerant schedule tree Φ of size M or fails to provide a schedulable solution.

We start by generating the f-schedule S_{root}, using the FTSG scheduling algorithm presented in Section 7.4.2[1]. The scheduling algorithm considers the situation where all the processes are

```
SchedulingStrategy(k, T, M, G): const R
1  S_root = FTSG(∅, k, T, G)
2  if S_root = ∅ then return unschedulable
3  else
4    set S_root as the root of fault-tolerant schedule tree Φ
5    Φ = FTTreeGeneration(S_root, k, T, M, G)
6    return Φ
7  end if
end SchedulingStrategy
```

Figure 7.5: General Scheduling Strategy

1. $\emptyset$ in the FTSG call in Figure 7.5 indicates that the root schedule S_{root} does not have a parent, i.e., $S_{parent} = \emptyset$.

executed with their worst-case execution times, while the utility is maximized for the case where processes are executed with their expected execution times (as was discussed in Figure 7.3). Thus, S_{root} contains the recovery slacks to tolerate k faults for hard processes and as many as possible faults for soft processes. The recovery slacks will be used by the runtime scheduler to re-execute processes online, without changing the order of process execution. Since this is the schedule assuming the worst-case execution times, many soft processes will be dropped to provide a schedulable solution.

If, according to S_{root}, the application is not schedulable, i.e., one or more hard processes miss their deadlines or if it exceeds period T, we conclude that we cannot find a schedulable solution and terminate. If the application is schedulable, we generate the schedule tree Φ starting from the schedule S_{root} by calling a schedule tree generation heuristic, presented in the next section. The tree generation heuristic will call FTSG to generate f-schedules in the schedule tree.

7.4.1 SCHEDULE TREE GENERATION

In general, a schedule tree generation heuristic should generate a tree that will adapt to different execution situations. However, tracing all execution scenarios is infeasible. Therefore, we have used the same principle as in [Cor04b] to reduce the number of schedules in the schedule tree Φ. We consider only the *expected* and the *worst-case* execution times of processes.

We begin the generation of the tree, as depicted in Figure 7.6, from the root schedule S_{root}, from which the scheduler will switch to the other schedules in the tree in order to improve the overall utility. We consider S_{root} as the current schedule ϕ_{max} ($\phi_{max} = S_{root}$, line 2). The tree generation algorithm is iterating while improving the total utility value with the new schedules ϕ_{new}, produced from the current schedule ϕ_{max} (lines 3-20). Alternatively, it stops if the maximum number M of f-schedules has

been reached. We evaluate each process P_i in ϕ_{max} as completed with *expected execution time*, and generate a corresponding schedule ϕ_{new} (lines 8-13). After ϕ_{new} is generated, we determine when it is profitable to change to it from ϕ_{max}, and at what completion times of process P_i (line 10), in the *interval partitioning step*. We heuristically compute *switching points* to ϕ_{new} by exploring possible completion times (from the earliest possible to the latest possible) of process P_i in ϕ_{max} with a given interval Δ. Then, we evaluate the new schedule tree containing also ϕ_{new} (line 11). This evaluation is done by simulation. During simulation, we run the actual runtime scheduler that executes a vari-

```
FTTreeGeneration(S_root, k, T, M, G): const R, Δ, dε
1  Φ = ∅
2  Φ' = S_root; φ_max = S_root
3  do
4    improvement = false
5    if φ_max ≠ ∅ and TreeSize(Φ) ≤ M then
6      improvement = true
7      Φ = Φ ∪ φ_max; Remove(φ_max, Φ')
8      for all P_i ∈ φ_max do
9        φ_new = FTSG(P_i, φ_max, k, T, G)
10       IntervalPartitioning(P_i, φ_max, φ_new, Δ)
11       U_new = Evaluate(Φ ∪ φ_new, dε)
12       if U_new > U_max then Φ' = Φ' ∪ φ_new
13     end for
14     U_max = 0; φ_max = ∅
15     for all φ_j ∈ Φ' do
16       U_j = Evaluate(Φ ∪ φ_j, dε)
17       if U_max < U_j then U_max = U_j; φ_max = φ_j
18     end for
19   end if
20 while improvement
21 return Φ
end FTTreeGeneration
```

Figure 7.6: Schedule Tree Generation

ety of execution scenarios, which are randomly generated based on the execution time distributions of application processes. We are able to run 10.000s execution scenarios in a matter of seconds. For each simulation run we obtain the utility produced by the application in this run, and also obtain the total utility U_{new} as an average over the utility values produced by the application in all the simulation runs so far. Simulation is performed until the value of the total utility U_{new} converges with a given error probability $d\varepsilon$. If ϕ_{new} improves the total utility of the schedule tree (as compared to the tree without ϕ_{new}), it is temporarily attached to the tree (added to the set Φ' of temporarily selected *f*-schedules, line 12). All new schedules, which are generated, are evaluated in this manner and the *f*-schedule ϕ_{max}, which gives the best improvement in terms of the total utility, is *selected* (lines 14-18) to be used in the next iteration (instead of the initial S_{root}). It is permanently added to the tree Φ and is removed from the temporal set Φ' in the next iteration (line 7).

7.4.2 GENERATION OF *F*-SCHEDULES

Our scheduling algorithm for generation of a single *f*-schedule is outlined in Figure 7.7. We construct a single *f*-schedule ϕ_{new}, which begins after switching from its parent schedule S_{parent} after completion of process P_i. We consider that P_i executes with its *expected execution time* for all execution scenarios in ϕ_{new}. Because of the reduced execution time of P_i, from the worst case to the expected, more soft processes can be potentially scheduled in ϕ_{new} than in S_{parent}, which will give an increased overall utility to the schedule ϕ_{new} compared to S_{parent}. (Note that in the case of generation of the root schedule S_{root}, if $S_{parent} = \varnothing$, all the processes are considered with their worst-case execution times.)

The algorithm iterates while there exist processes in the ready list $\mathcal{L}_R$ (lines 2-3). List $\mathcal{L}_R$ includes all "ready" soft and hard processes. By a "ready" hard process, we will understand an unscheduled hard process, whose all hard predecessors are

```
FTSG(P_i, S_parent, k, T, G): const R
1  φ_new = ∅
2  L_R = GetReadyNodes(G, S_parent)
3  while L_R ≠ ∅ do
4    A = LeadToSchedulableSolution(L_R, k, T, G)
5    Best = ProcessSelect(A, k, T, G)
6    Schedule(Best, φ_new)
7    AddSlack(Best, φ_new)
8    UpdateReadyNodes(Best, L_R)
9    Dropping(L_R, k, T, G)
10 end while
11 return φ_new
end FTSG
```

Figure 7.7: Single Schedule Generation

already scheduled. A "ready" soft process is simply an unscheduled soft process. The ready list $\mathcal{L}_R$ is prepared during initialization of the algorithm by collecting the "ready" processes, which have not been scheduled in S_{parent} (line 2).

At each iteration, the heuristic determines which processes, if chosen to be executed, lead to a schedulable solution (line 4), and copy them to the list A. Then the "best" process is selected from A (ProcessSelect, line 5). ProcessSelect selects either the "best" soft process that would potentially contribute to the greatest overall utility according to the MU priority function proposed in [Cor04b] or, if there are no soft processes in A, the hard process with the earliest deadline.

The MU priority function, which we adopt, computes for each soft process P_i the value, which constitutes of the utility $U_i^*(t)$ produced by P_i, scheduled as early as possible, and the sum of utility contributions of the other soft processes delayed because

of P_i. The utility contribution of the delayed soft process P_j is obtained as if process P_j would complete at time

$$t_j = \frac{t_j^E + t_j^L}{2}$$

where t_j^E and t_j^L are the completion times of process P_j in the case that P_j is scheduled after P_i as early as possible and as late as possible, respectively.

Note that we have modified the original MU priority function to capture dropping of soft processes by considering the *degraded utility* $U^*(t)$, obtained with a service degradation rule $\mathcal{R}$, instead of the original $U(t)$.

We schedule the selected process *Best* (line 6) and for each process we add recovery slack, as discussed in Section 7.2 (AddSlack, line 7). Recovery slacks of hard processes will accommodate all k faults. Recovery slacks of soft processes, however, must not reduce the utility value of the no-fault scenarios and will, thus, accommodate re-executions against as much faults as possible but, in general, not against all k faults.

Finally, the list $\mathcal{L}_R$ of ready nodes is updated with the ready successors of the scheduled process (line 8). Those soft processes, whose executions will not lead to any utility increase or would exceed application period T, are removed from the ready list $\mathcal{L}_R$, i.e., dropped (line 9).

The FTSG heuristic will return either $\phi_{new} = S_{root}$ or a fault-tolerant schedule ϕ_{new} that will be integrated into the schedule tree Φ by the FTTreeGeneration heuristic.

7.4.3 SWITCHING BETWEEN SCHEDULES

Online switching between schedules in the schedule tree is performed very fast. At each possible switching point, e.g. after completion of a process in the current schedule, the scheduler can have at most two possible alternatives, i.e., *to switch* or *not to switch* to the "new" schedule ϕ_{new}. We store a pointer to a

"new" schedule ϕ_{new} in an "old" schedule S_{parent}, when attaching the "new" schedule to the schedule tree (line 7, Figure 7.6), associated to a process P_i, whose finishing time triggers the potential switching to the "new" schedule. We also store the pre-computed switching intervals. Thus, the runtime scheduler will only check if the completion time of the process P_i matches the switching interval (i.e., an "if" statement has to be executed in the scheduler code) and, if so, will de-reference the pointer to the "new" schedule ϕ_{new}. Thus, no searching of the appropriate schedules is performed online and the online time overhead is practically neglectable.

7.5 Experimental Results

For our experimental evaluation, we have generated 450 applications with 10, 15, 20, 25, 30, 35, 40, 45, and 50 processes, respectively, where we have uniformly varied worst-case execution times of processes between 10 and 100 ms. We have generated best-case execution times randomly between 0 ms and the worst-case execution times. We consider that completion time of processes is uniformly distributed between the best-case execution time t_i^b and the worst-case execution time t_i^w, i.e., the expected execution time t_i^e is $(t_i^w - t_i^b) / 2$. The number k of tolerated faults has been set to 3 and the recovery overhead μ to 15 ms. The experiments have been run on a Pentium 4 2.8 GHz processor with 1Gb of memory.

In the first set of experiments we have evaluated the quality of single fault-tolerant schedules produced by our FTSG algorithm. We have compared it with a straightforward approach that works as follows: we obtain single non-fault-tolerant schedules that produce maximal value (e.g. as in [Cor04b]). Those schedules are then made fault-tolerant by adding recovery slacks to tolerate k faults in the hard processes. The soft proc-

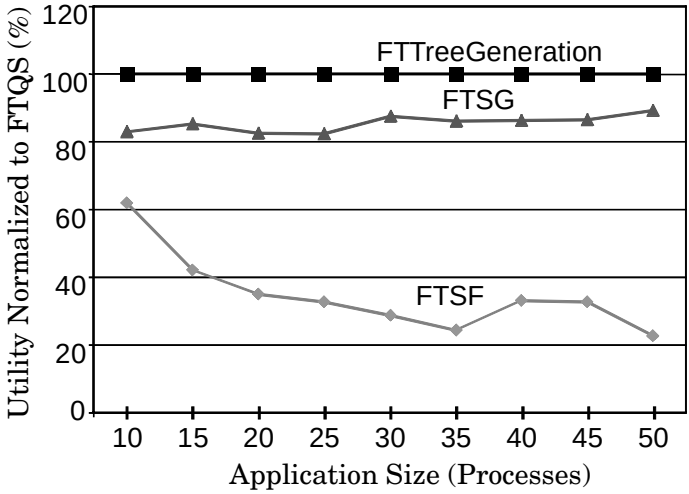


Figure 7.8: Comparison between FTTreeGeneration, FTSG and FTSF

esses with the lowest utility value are dropped until the application becomes schedulable. We call this straightforward algorithm FTSF. The experimental results given in Figure 7.8 show that FTSF is 20-70% worse in terms of utility compared to FTSG.

In a second set of experiments we were interested to determine the quality of our schedule tree generation approach for fault tolerance (FTTreeGeneration) in terms of overall utility for the no-fault scenario and for the fault scenarios. Figure 7.8 presents the normalized utility obtained by the three approaches, varying the size of applications. We have evaluated schedules generated by FTTreeGeneration, FTSG, and FTSF with extensive simulations. The overall utility for each case has been calculated as an average over all execution scenarios. Figure 7.8 shows the results for

the no-fault scenarios. We can see that FTTreeGeneration is 11-18% better than FTSG, where FTSG is the best of the single schedule alternatives.

7.6 Conclusions

In this chapter we have addressed fault-tolerant applications with soft and hard real-time constraints. The timing constraints have been captured using deadlines for hard processes and time/utility functions for soft processes.

We have proposed an approach to the synthesis of a tree of fault-tolerant schedules for mixed hard/soft applications run on monoprocessor embedded systems. Our value-based quasi-static scheduling approach guarantees the deadlines for the hard processes even in the case of faults, while maximizing the overall utility of the system.

Chapter 8
Value-based Scheduling for Distributed Systems

IN THIS CHAPTER we present an approach for scheduling of fault-tolerant applications composed of soft and hard real-time processes running on *distributed embedded systems*. We extend our monoprocessor quasi-static scheduling approach from Chapter 7. We propose a scheduling algorithm to generate a tree of fault-tolerant distributed schedules that maximize the application's quality value and guarantee hard deadlines even in the presence of faults. We also develop a signalling mechanism to transmit knowledge of the system state from one computation node to the others and we explicitly account for communications on the bus during generation of schedule tables.

8.1 Scheduling

Our problem formulation for the value-based scheduling in the distributed context is similar to the case of monoprocessor systems, presented in Section 7.3, with the difference that we consider a *distributed* bus-based architecture, where the mapping $\mathcal{M}$ of processes on computation nodes is given. We have to obtain a tree of fault-tolerant schedules on computation nodes that maximize the total utility U produced by the application and satisfies all hard deadlines in the worst case. In this section we propose a value-based quasi-static scheduling approach that solves this problem of utility maximization for fault-tolerant distributed systems.

In this chapter we adapt the scheduling strategy, which we have applied in Chapter 7 to generate fault-tolerant schedules for mixed soft and hard real-time monoprocessor systems. This strategy uses a "recovery slack" in the schedule to accommodate the time needed for re-executions in the case of faults. After each process P_i we will assign a slack equal to $(t_i^w + \mu) \times f$, where f is the number of faults to tolerate, t_i^w is the worst-case execution time of the process and μ is the re-execution overhead. The slack is shared between processes in order to reduce the time allocated for recovering from faults.

We extend our monoprocessor scheduling strategy from Chapter 7 to capture communications on the bus. In our scheduling approach for distributed hard real-time systems in Section 4.4, the sending and receiving times of messages on the bus are fixed and cannot be changed within one fault-tolerant schedule. Each message m_i, which is an output of process P_j, is always scheduled at fixed time on the bus after the worst-case completion time of P_j. In this chapter, we will refer to such a fault-tolerant multiprocessor schedule with recovery slacks and fixed communications as an f^N-schedule, where N is the number of computation nodes in the system.

8.1.1 SIGNALLING AND SCHEDULES

Our primary objective is to maximize the total utility value of the application. However, pure f^N schedules with fixed sending times of messages can lower the total utility due to imposed communication restrictions. Thus, we will propose a *signalling* mechanism to overcome this restriction, where a *signalling message* is *broadcasted* by computation node N_j to all other nodes to inform if a certain condition has been satisfied on that node. In our case, this *condition* is the completion of process P_i on node N_j at such time that makes execution of another pre-calculated schedule f_n^N more beneficial than the presently running f_p^N schedule. The condition can be used for scheduling if and only if it has been broadcasted to all computation nodes with the signalling message. Switching to the new schedule f_n^N is performed by all computation nodes in the system after receiving the broadcasted signalling message with the corresponding "true" condition value. In the proposed signalling approach, we will send one signalling message for each process, *after the worst-case execution time for hard processes* or *after the expected execution time for soft processes* in the schedule. The number of signalling messages has been limited to one per process, taking into account problem complexity and in order to provide a simple yet efficient solution. We will illustrate our signalling mechanism for soft and hard processes on two examples in Figure 8.1 and Figure 8.2, respectively.

In Figure 8.1, we consider application $\mathcal{A}_1$ with 5 processes, P_1 to P_5, where processes P_1 and P_5 are hard with deadlines of 170 and 400 ms, respectively, and processes P_2, P_3 and P_4 and all messages are soft. In Figure 8.1a, a single f^2 schedule S_0 is generated, as depicted above the Gantt chart that outlines a possible execution scenario that would follow S_0. Message sending times on the bus are fixed and message m_2 is, thus, always sent at 150 ms (denoted as m_2[150] in the schedule). m_2 has to wait

for the worst-case completion time of P_2.[1] The order of processes is preserved on computation nodes. However, the start times of

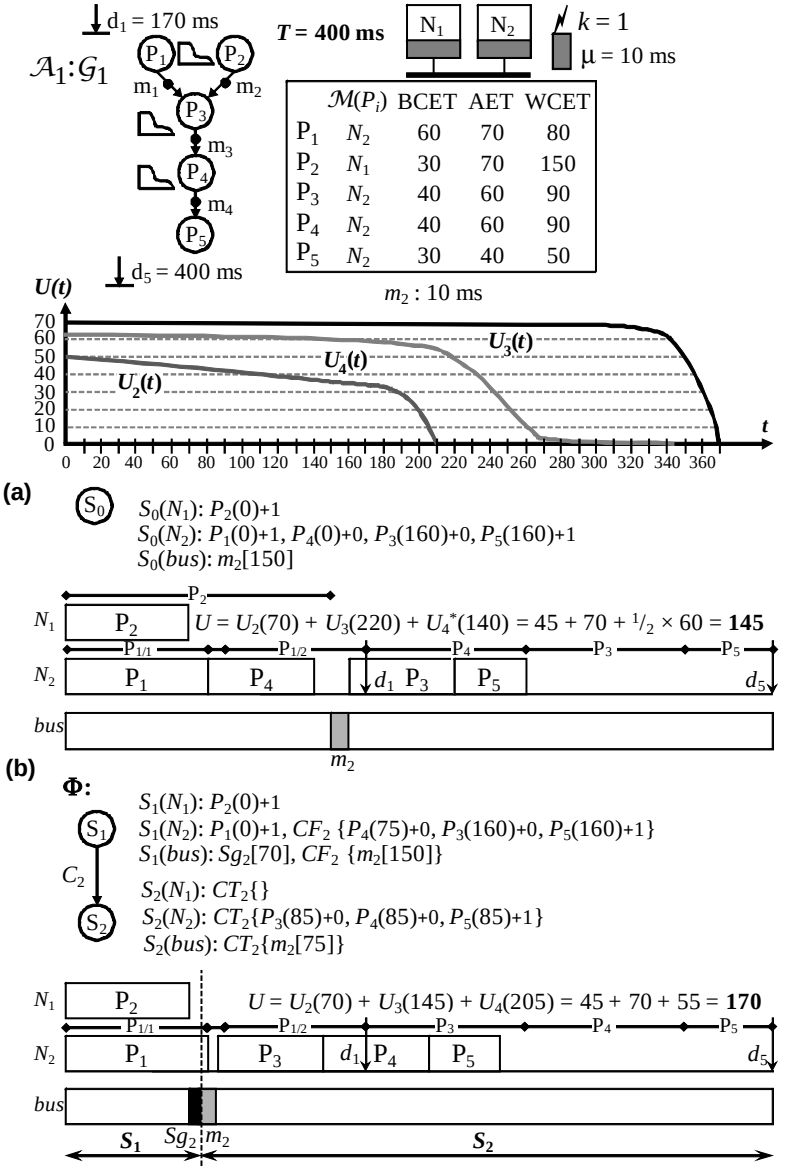


Figure 8.1: Signalling Example for a Soft Process

processes are varied, under the constraint that processes do not start earlier than their *earliest allowed start time* in the schedule. For example, process P_3 will wait until completion of process P_4 but will not start earlier than 160 ms, waiting for message m_2 (with the constraint denoted as $P_3(160)$ in the schedule). In schedule S_0 we also depict the number *rex* of *allowed re-executions* for processes with the "+*rex*" notation. For example, process P_2 is allowed to re-execute *rex* = 1 time to tolerate $k = 1$ fault, i.e., $P_2(0) + 1$, while processes P_3 and P_4 are not allowed to re-execute (*rex* = 0).

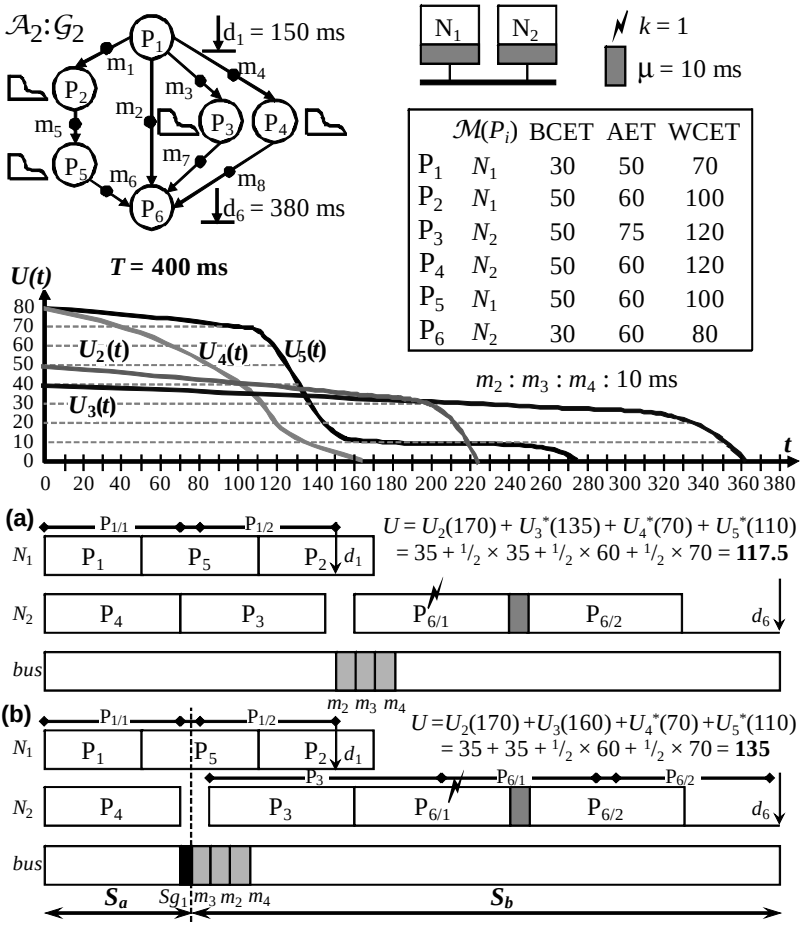
In the execution scenario in Figure 8.1a, which follows S_0, process P_2 completes at 70 ms with utility of 45. Process P_4 is executed directly after process P_1 with resulting utility of $^1/_2 \times 60 = 30$. Process P_3 is waiting for message m_2 and completes with utility of 70. The overall utility is, thus, 145.

In the schedule shown in Figure 8.1b, we employ the signalling mechanism. The signalling message Sg_2 is sent after the *expected execution time* of the *soft process* P_2, which can trigger the switch from the initial *root* schedule S_1 to the new schedule S_2. Schedule tree Φ, composed of schedules S_1 and S_2, is depicted above the Gantt chart in Figure 8.1b. Switching between the schedules is performed upon known "true" condition CT_2, which is broadcasted with the signalling message Sg_2. CT_2 informs that P_2 has completed at such time that switching to S_2 has become profitable. In the opposite case, when switching to S_2 is not profitable, due to, for example, that P_2 has not completed at that time, Sg_2 transports a "false" condition CF_2. Thus, two execution scenarios are possible in the tree Φ: under "true" condition CT_2 and under "false" condition CF_2. As illustrated in Figure 8.1b, in schedule S_1, in the CF_2 case, processes P_3, P_4 and P_5 and message m_2 are grouped under this condition as $CF_2\{P_4, P_3, P_5\}$ on node N_2 and as $CF_2\{m_2\}$ on the bus. They will be acti-

1. Note that if P_2 is re-executed, message m_2 will be, anyway, sent at 150 ms with a "stale" value.

vated in the schedule only when "false" condition CF_2 is received. In schedule S_2, to which the scheduler switches in the case of "true" condition CT_2, processes P_3, P_4 and P_5 and message m_2 are grouped under CT_2 that will activate them.

With the signalling mechanism, the overall utility produced by application $\mathcal{A}_1$ is improved. In the Gantt chart in Figure 8.1b, process P_2 completes at 70 ms with utility of 45. In this case



	$\mathcal{M}(P_i)$	BCET	AET	WCET
P_1	N_1	30	50	70
P_2	N_1	50	60	100
P_3	N_2	50	75	120
P_4	N_2	50	60	120
P_5	N_1	50	60	100
P_6	N_2	30	60	80

Figure 8.2: Signalling Example for a Hard Process

switching to the schedule S_2 is profitable and, hence, signalling message Sg_2 with the "true" condition CT_2 is broadcasted. The scheduler switches from S_1 to S_2 after arrival of Sg_2, as shown in the figure. According to the new order in schedule S_2, process P_4 is executed after its predecessor P_3, with the utilities of 55 and 70, respectively. The resulting utility of this execution scenario, thus, will be 170, instead of only 145 in Figure 8.1a.

As another example, in Figure 8.2 we depict execution scenarios for application $\mathcal{A}_2$, which is composed of 6 processes, P_1 to P_6, where processes P_2 to P_5 are soft with utility functions $U_2(t)$ to $U_5(t)$ and processes P_1 and P_6 are hard with deadlines 150 and 380 ms, respectively. Message m_2 is hard and all other messages are soft. A single f^2 schedule will produce the overall utility of 117.5 for the execution scenario depicted in Figure 8.2a. If, however, we send a signal Sg_1 after the *worst-case execution time* of the hard process P_1 and generate the corresponding schedule S_b, we can produce better overall utility of 135, by switching to the schedule S_b, as depicted in Figure 8.2b.

8.1.2 SCHEDULE TREE GENERATION

The tree of fault-tolerant schedules is constructed with the FTTreeGenerationDistributed heuristic, outlined in Figure 8.3. This heuristic is an almost exact copy of the FTTreeGeneration heuristic, which we have presented in Section 7.4.1 for monoprocessor systems. In the distributed context, however, the root schedule S_{root} and new schedules ϕ_{new} are generated with the FTSGDistributed heuristic, outlined in Figure 8.4, which takes care of communication and signalling. Note that, as an additional input, we also receive the mapping $\mathcal{M}$ of processes in the application graph $\mathcal{G}$.

Our f^N schedule generation heuristic, FTSGDistributed, is based on the monoprocessor FTSG heuristic, which we have proposed in Section 7.4.2, with a number of differences such as handling communication over the bus with signalling messages and

```
FTTreeGenerationDistributed(S_root, k, T, M, 𝓜, 𝓖): const 𝓡, Δ, dε
1  Φ = ∅
2  Φ' = S_root; ϕ_max = S_root
3  do
4    improvement = false
5    if ϕ_max ≠ ∅ and TreeSize(Φ) ≤ M then
6      improvement = true
7      Φ = Φ ∪ ϕ_max; Remove(ϕ_max, Φ')
8      for all P_i ∈ ϕ_max do
9        ϕ_new = FTSGDistributed(P_i, ϕ_max, k, T, 𝓜, 𝓖)
10       IntervalPartitioning(P_i, ϕ_max, ϕ_new, Δ)
11       U_new = Evaluate(Φ ∪ ϕ_new, dε)
12       if U_new > U_max then Φ' = Φ' ∪ ϕ_new
13     end for
14     U_max = 0; ϕ_max = ∅
15     for all ϕ_j ∈ Φ' do
16       U_j = Evaluate(Φ ∪ ϕ_j, dε)
17       if U_max < U_j then U_max = U_j; ϕ_max = ϕ_j
18     end for
19   end if
20 while improvement
21 return Φ
end FTTreeGenerationDistributed
```

Figure 8.3: Schedule Tree Generation in the Distributed Context

assigning *guards*. Processes and messages in FTSGDistributed are scheduled under *known conditions* or guards [Ele00]. The scheduler will switch from a parent schedule S_{parent} to the new schedule ϕ_{new} in the case it receives the signalling message Sg_i containing a *true* condition.

Current guard is initially set to *true* for the constructed schedule ϕ_{new} in the case of no parent schedule (and $\phi_{new} = S_{root}$ schedule is generated, for example, S_1 in Figure 8.1b). If there exists a parent schedule, the value of the current guard is initialized with the condition value upon which the schedule ϕ_{new} is acti-

vated by its parent schedule S_{parent} (line 1).[1] The current guard will be updated after scheduling of each signalling message Sg_j in ϕ_{new}.

During construction of ϕ_{new}, we consider that P_i executes with its *expected execution time* for all execution scenarios in ϕ_{new}. Because of the reduced execution time of P_i, from the worst case to the expected case, more soft processes can be potentially scheduled in ϕ_{new} than in S_{parent}, which will give an increased overall utility to the schedule ϕ_{new} compared to S_{parent}.

The algorithm in Figure 8.4, similar to the monoprocessor heuristic FTSG, iterates while there exist processes and messages in the ready list $\mathcal{L}_R$, which are selected to be scheduled

FTSGDistributed(P_i, S_{parent}, k, T, $\mathcal{M}$, $\mathcal{G}$): const $\mathcal{R}$

1 $\phi_{new} = \emptyset$; SetCurrentGuard(P_i, ϕ_{new})
2 $\mathcal{L}_R$ = GetReadyNodes($\mathcal{G}$)
3 **while** $\mathcal{L}_R \neq \emptyset$ **do**
4 A = LeadToSchedulableSolution($\mathcal{L}_R$, k, T, $\mathcal{M}$, $\mathcal{G}$)
5 **for each** resource $r_j \in \mathcal{N} \cup \{B\}$ **do** $Best_j$ = ProcessSelect(r_j,A,k,T,$\mathcal{M}$,$\mathcal{G}$)
6 $Best$ = SelectBestCRT(**all** $Best_j$)
7 K = ObtainGuards($Best$, ϕ_{new})
8 Schedule($Best$, K, ϕ_{new})
9 **if** $Best$ **is** a process **then**AddSlack($Best$, ϕ_{new}); AddSgMsg($Best$, $\mathcal{L}_R$)
10 **if** $Best$ **is** a signal **then** UpdateCurrentGuards($Best$, ϕ_{new})
11 UpdateReadyNodes($Best$, $\mathcal{L}_R$)
12 Dropping($\mathcal{L}_R$, k, T, $\mathcal{G}$)
13 **end while**
14 **return** ϕ_{new}
end FTSGDistributed

Figure 8.4: Single Schedule Generation in the Distributed Context

1. This condition value will be calculated online based on the actual finishing time of the process and will initiate switching to the schedule ϕ_{new}, as discussed in Section 8.1.3.

(lines 2-3). List $\mathcal{L}_R$ includes all "ready" soft and hard processes and messages. By a "ready" hard process or message, we will understand an unscheduled hard process or message, whose all hard predecessors are already scheduled. A "ready" soft process or message is simply an unscheduled soft process or message. At each iteration, the heuristic determines which processes and messages, if chosen to be executed, lead to a schedulable solution (line 4), and copy them to the list A. Then the "best" processes on computation nodes and the "best" message on the bus are selected from A (ProcessSelect, line 5) that would potentially contribute to the greatest overall utility. We select processes and messages according to the MU priority function proposed in [Cor04b] and modified by us to capture possible process dropping (see Section 7.4.2, where, during priority computation, we consider the degraded utility $U^*(t)$, obtained with a service degradation rule $\mathcal{R}$, instead of the original $U(t)$). Out of the "best" processes and messages on different resources, we select the one which can be scheduled the earliest (line 6).

We schedule the selected process or message *Best* under the current set of known conditions K (lines 7-8). For each process we add a recovery slack (AddSlack, line 9). Recovery slacks of hard processes will accommodate re-executions against all k faults. Recovery slacks of soft processes, however, must not reduce the utility value of the no fault scenarios and will, thus, accommodate re-executions against as much faults as possible but, in general, not against all k faults. For each process P_j we also add its signalling message Sg_j to the ready list $\mathcal{L}_R$ (AddSgMsg, line 9). When scheduling a signalling message (line 10), we change current guards of computation nodes at arrival time of the message.

Finally, the list $\mathcal{L}_R$ is updated with the ready successors of the scheduled process or message (line 11). Those soft processes, whose executions will not lead to any utility increase or would exceed application period T, are removed from the ready list $\mathcal{L}_R$, i.e., dropped (line 12).

The FTSGDistributed heuristic will return either $\phi_{new} = S_{root}$ or an f^N schedule ϕ_{new} that will be integrated into the schedule tree Φ by the FTTreeGenerationDistributed heuristic in Figure 8.3.

8.1.3 SWITCHING BETWEEN SCHEDULES

Switching between schedules in the schedule tree, at runtime, is also performed very fast but is different from the monoprocessor case. At each possible switching point, e.g. after completion of a process P_i, the scheduler can have at most two possible alternatives, i.e., *to signal* or *not to signal* the switching to the "new" schedule ϕ_{new}. As in the monoprocessor case, we store a pointer to the "new" schedule ϕ_{new} in its parent schedule S_{parent}, when attaching the "new" schedule ϕ_{new} to the schedule tree (line 7, Figure 8.3). However, the pointer is associated to *a signalling message* of the process P_i, whose finishing time triggers the potential switching to the "new" schedule ϕ_{new}. We also store the pre-computed switching intervals as attached to this process. Thus, the runtime scheduler checks if the completion time of the process P_i matches the switching interval and, if so, *encapsulates* the corresponding switching condition into the signalling message, which is broadcasted through the bus.

Upon arrival of the signalling message, the runtime scheduler on each computation node will de-reference the pointer, associated to this message, and will switch to the "new" schedule ϕ_{new}. Thus, no searching of the appropriate schedules is performed online in the distributed context and the online time overhead is reduced.

8.2 Experimental Results

For our experiments we have generated 1008 applications of 20, 30, 40, 50, 60, 70 and 80 processes (24 applications for each dimension) for 2, 3, 4, 5, 6 and 7 computation nodes. Execution

times of processes in each application have been varied from 10 to 100 ms, and message transmission times between 1 and 4 ms. Deadlines and utility functions have been assigned individually for each process in the application. For the main set of experiments we have considered that 50% of all processes in each application are soft and the other 50% are hard. We have set the maximum number k of transient faults to 3 and recovery overhead to 10% of process execution time. As the worst-case execution time of a process can be much longer than its expected execution time, we have also assigned a tail factor $TF_i = WCET_i / (AET_i \times 2)$ to each process P_i. Experiments have been run on a 2.83 GHz Intel Pentium Core2 Quad processor with 8 Gb memory.

At first, we were interested to evaluate our heuristics with the increased number of computation nodes and the number of processes. For this set of experiments we have set the tail factor to 5 for all soft processes in the applications. Table 8.1 shows an improvement of the schedule tree on top of a single f^N schedule in terms of total utility, considering an f^N schedule in the case of no faults as a 100% baseline. The average improvement is ranging from 3% to 22% in case of 20 processes on 6 computation nodes and 40 processes on 2 computation nodes, respectively. Note that in this table we depict the normalized utility values

Table 8.1: Normalized Utility ($U_n = U_{FTTree}/U_{f^N} \times 100\%$) and the Number of Schedules (n)

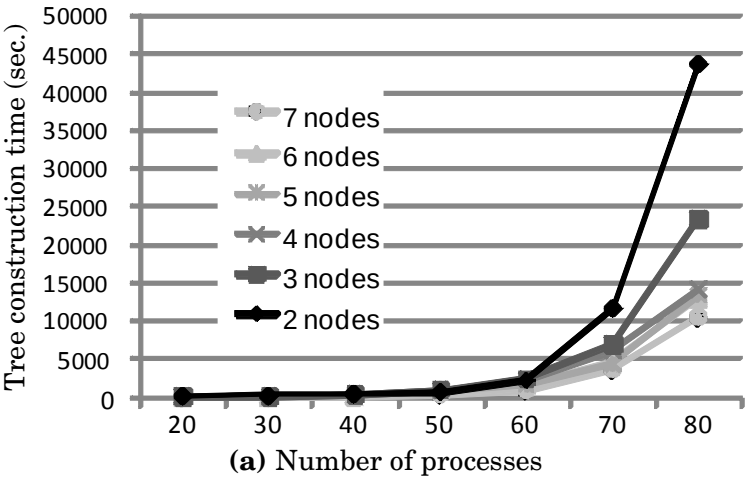
N	20 proc.		30 proc.		40 proc.		50 proc.		60 proc.		70 proc.		80 proc.	
	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n
2	117	4.3	119	5.3	122	5.6	117	5.8	116	5.4	114	6.5	112	4.9
3	111	3.8	113	4.6	119	6.9	119	6.0	118	5.8	115	7.4	114	7.5
4	109	2.7	112	4.5	113	5.4	118	7.0	115	7.0	115	7.3	113	6.8
5	106	2.5	112	2.6	113	5.6	112	5.8	116	8.0	113	5.8	115	7.0
6	103	2.2	109	4.2	110	4.5	112	6.1	115	7.3	113	6.7	113	6.3
7	103	1.8	106	3.0	109	4.8	112	5.4	110	6.0	110	5.8	112	7.0

for schedule trees that have been obtained in the case of *no faults*. During our experiments, we have observed that the utility values for faulty cases closely follow the no fault scenarios with about 1% decrease (on average) in the utility value for each fault, as illustrated, for example, in Figure 8.5b.

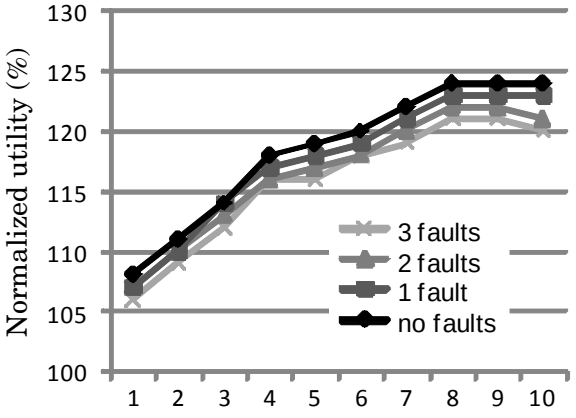
As the number of computation nodes increases, in general, the schedule tree improves less on top of a single f^N schedule, which could seem counterintuitive. However, this is because, with an increased number of computation nodes, less soft processes are allocated per node on average and the number of possible value-based intra-processor scheduling decisions by the tree generation algorithm is reduced. At the same time, the number of inter-processor scheduling decisions is supposed to increase. However, these decisions are less crucial from the point of view of obtaining utility due to the greater availability of computational resources, and a single f^N schedule with a fixed order of processes is sufficient to utilize them. Moreover, a limited number of signalling messages, e.g. one per process, restricts the number of possible inter-processor decisions by the tree generation algorithm. Hence, the total number of exploitable scheduling alternatives is reduced with more computation nodes. In case of 20 processes and 2 nodes, the average number of soft processes is 5 per node (considering 50% soft processes in the application, $20 \times 0.5 / 2 = 5$) and the utility improvement is 17%. In the case of 7 nodes, only an average of 1.4 soft processes are allocated per node and, hence, the utility improvement is only 3%. However, as the number of processes in the application is growing, the trend is softened. For 40 processes and 2 nodes the improvement is 22% with the average of 10 soft processes per node. For 7 nodes the improvement is 9% with 2.9 soft processes per node on average. For 80 processes, the improvement in case of 2 and 7 nodes is already the same, 12% for 20 and 5.7 soft processes per node, respectively.

The average size of the schedule tree is between 2.2 schedules for 20 processes on 6 nodes and 8.0 schedules for 60 processes on

5 nodes that correspond to 1.3Kb and 9.4Kb of memory, respectively. As the size of the application grows, the amount of memory per schedule will increase. Thus, a tree of 7 schedules for an 80-process application on 5 nodes would require 11.4Kb of



(a) Number of processes



(b) Tail factor, $TF = WCET / (AET \times 2)$

Figure 8.5: Experimental Results for Schedule Tree Generation in the Distributed Context

memory. During our experiments we could also observe that for many applications already 2-3 schedules give at least half of the utility improvement. For example, for 40 processes and 2 computation nodes, 3 schedules will give 21% improvement.

Off-line tree construction times with FTTreeGenerationDistributed are depicted in Figure 8.5a. Our heuristic produces a tree of schedules in a matter of minutes for 20, 30 and 40 processes, and below one hour for 50 and 60 processes. Tree construction time is around 4 hours for 5 nodes and 80 processes. Although the off-line tree construction time is high for large applications, the online overhead is still very small and is constant for all applications, in spite of the application size, as discussed in Section 8.1.3.

For a given total number of processes, the tree construction time is reduced with the number of computation nodes. This can be explained taking into account that the average number of soft processes per computation node is reduced, which leads to a smaller amount of scheduling alternatives to be explored by the tree generation algorithm. In case of 2 nodes, a greater number of soft processes for each computation node is allocated and, as the result, more valuable scheduling alternatives in total have to be considered than in the case of, for example, 7 nodes. Our tree generation algorithm, thus, has to spend much more time evaluating scheduling alternatives for less nodes than in the case of more nodes, as confirmed by the experimental results given in Figure 8.5a.

In our next set of experiments, we have varied the percentage of soft and hard processes in the applications. We have additionally generated 1080 applications of 20, 30, 40, 50 and 60 processes for 2, 3, 4, 5 and 6 computation nodes, respectively. The percentage of soft processes has been initially set to 10% and hard processes to 90%. We have gradually increased the percentage of soft processes up to 90% of soft processes and 10% of hard processes, with a step of 10% (and have generated 120 applications of different dimensions for each setup). The improvement

produced by FTTreeGenerationDistributed over all dimensions is between 14 and 16%. The average number of schedules has increased from 2 for 10% of soft processes to 6 for 90% of soft processes, with the increased execution time of the heuristic from 10 to 30 minutes.

We were also interested to evaluate our heuristic with different tail factors *TF*, which we have varied from 1 to 10 for applications with 40 processes on 2 nodes. As our results in Figure 8.5b show, improvement produced by FTTreeGeneration-Distributed is larger in the case of a greater tail factor. If for the tail factor 1, the improvement is 7.5%, it is around 20% for the tail factors of 5-6 and 24% for the tail factors above 7. This is due to the fact that, if the tail factor increases, more soft processes are dropped in a single f^N schedule while they are allowed to complete in the case of the schedule tree. The processes are more rarely executed with the execution times close to the worst case with the greater tail factors. Switching to another schedule in the schedule tree (after completion of several process) will allow many of these soft process to complete, without the risk of deadline violations, and contribute to the overall utility of the application. Note that the total utilities of fault scenarios, as depicted in Figure 8.5b, closely follow the no fault scenarios with only about 1% utility value decrease for each fault.

We have also run our experiments on a real-life example, the vehicle cruise controller (CC), previously used to evaluate scheduling algorithms in Chapter 4, which is composed of 32 processes. CC has been mapped on 3 computation units: Electronic Throttle Module (ETM), Anti-lock Braking System (ABS) and Transmission Control Module (TCM). We have considered that 16 processes, which closely involved with the actuators, are hard. The rest of processes have been assigned with utility functions. The tail factor is 5 with $k = 2$ transient faults, recovery overhead of 10% of process execution time and signalling message transmission time of 2 ms. In terms of total utility FTTreeGenerationDistributed could improve on top of a single f^3

schedule with 22%, with 4 schedules that would need 4.8Kb of memory.

8.3 Conclusions

In this chapter we have presented an approach to the scheduling of mixed soft and hard real-time *distributed embedded systems* with fault tolerance.

We have proposed a quasi-static scheduling approach that generates a tree of fault-tolerant schedules off-line that maximizes the total utility value of the application and satisfies deadlines in the distributed context. Depending on the current execution situation and fault occurrences, an online distributed scheduler chooses which schedule to execute on a particular computation node, based on the pre-computed switching conditions broadcasted with the signalling messages over the bus.

The obtained tree of fault-tolerant schedules can deliver an increased level of quality-of-service and guarantee timing constraints of safety-critical distributed applications under limited amount of resources.

PART IV
Embedded Systems with Hardened Components

Chapter 9
Hardware/Software Design for Fault Tolerance

IN THIS PART OF THE THESIS we combine hardware hardening with software-level fault tolerance in order to achieve the lowest-possible system hardware cost while satisfying hard deadlines and fulfilling the reliability requirements. We propose an optimization framework for fault tolerance, where we use re-execution to provide fault tolerance in software, while several alternative hardware platforms with various levels of hardening can be considered for implementation.

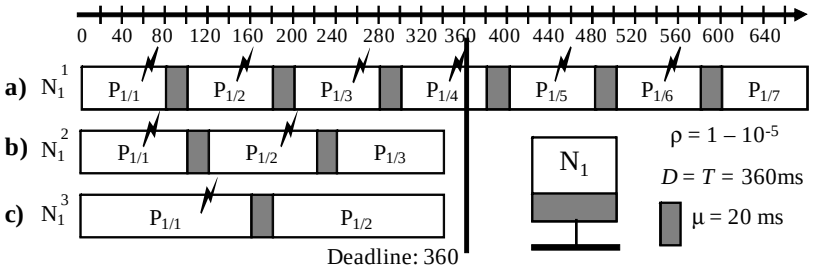
In this chapter we illustrate, on a simple example, the benefits of looking into software and hardware simultaneously to reduce the amount of software fault tolerance and minimize the cost of the hardware. We propose a system failure probability (SFP) analysis to ensure that the system architecture meets the reliability requirements. This analysis connects the levels of fault tolerance (maximum number of re-executions) in software to the levels of hardening in hardware (hardening levels). In the next chapter, we will propose a set of design optimization heuristics, based on the SFP analysis, in order to decide the hardening lev-

els of computation nodes, the mapping of processes on computation nodes, and the number of re-executions on each computation node.

9.1 Hardened Architecture and Motivational Example

To reduce the probability of transient faults, the designer can choose to use a *hardened*, i.e., a more reliable, version (h-version) of the computation node. Thus, each node N_j can be available in several versions N_j^h, with different hardening levels h, associated with the cost C_j^h, respectively. For each pair $\{P_i, N_j^h\}$, where process P_i is mapped to the h-version of node N_j, we know the worst-case execution time (WCET) t_{ijh} and the probability p_{ijh} of failure of a single execution of process P_i. WCETs are determined with worst-case analysis tools [Erm05, Sun95, Hea02, Jon05, Gus05, Lin00, Col03, Her00, Wil08] and the process failure probabilities can be obtained using fault injection tools [Aid01, Ste07].

In this part of our work, we will consider a reliability goal $\rho = 1 - \gamma$, with respect to transient faults, which has to be met by the safety-critical embedded system. γ is the maximum probability of a system failure due to transient faults on any computation node within a time unit, e.g. one hour of functionality. With sufficiently hardened nodes, the reliability goal can be achieved without any re-execution at software level, since the probability of the hardware failing due to transient faults is acceptably small. At the same time, as discussed in Section 2.2.5, hardware hardening comes with an increased cost and a significant overhead in terms of speed (*hardening performance degradation*). To reduce the cost or to increase the performance, designers can decide to reduce the hardening level. However, as the level of hardening decreases, the probability of faults being propagated to the software level will increase. Thus, in order to achieve the



$\mathcal{A}_1$:$\mathcal{G}_1$

N_1	h = 1		h = 2		h = 3	
	t	p	t	p	t	p
P_1	80	$4{\cdot}10^{-2}$	100	$4{\cdot}10^{-4}$	160	$4{\cdot}10^{-6}$
Cost	10		20		40	

Figure 9.1: Reduction of the Number of Re-executions with Hardening

reliability goal, a certain number of re-executions will have to be introduced at the software level, which, again, leads to the worst case performance degradation.

In the example, depicted in Figure 9.1, we show how hardening can improve schedulability if the fault rate is high. In Figure 9.1, we consider an application $\mathcal{A}_1$ composed of one process P_1, which runs on one computation node N_1, with three h-versions, N_1^1 without hardening and N_1^2 and N_1^3 progressively more hardened. The corresponding failure probabilities (denoted "p"), the WCET (denoted "t") and the costs are depicted in the table. The application runs with period T = 360 ms. We have to meet a deadline of 360 ms and the reliability goal of $1 - 10^{-5}$ within one hour. As will be shown in Section 9.2, the hardening levels are connected to the number of re-executions in software, to satisfy the reliability goal according to the SFP analysis. In this example, according to the SFP analysis, using

N_1^1, we have to introduce 6 re-executions to reach the reliability goal, as depicted in Figure 9.1a, which, in the worst case, will miss the deadline of 360 ms.[1] However, with the h-version N_1^2, the failure probability is reduced by two orders of magnitude, and only two re-executions are needed for satisfying the reliability goal ρ. This solution will meet the deadline as shown in Figure 9.1b. In case of the most hardened architecture depicted in Figure 9.1c, only one re-execution is needed. However, using N_1^3 will cost twice as much as the previous solution with less hardening. Moreover, due to hardening performance degradation, the solution with the maximal hardening will complete in the worst-case scenario exactly at the same time as the less hardened one. Thus, the architecture with N_1^2 should be chosen. The selected architecture will satisfy the reliability goal and meet deadlines with the lowest-possible hardware cost.

9.2 System Failure Probability (SFP) Analysis

In this section we present an analysis that determines the system failure probability, based on the number k of re-executions introduced in software and the process failure probability of the computation node with a given hardening level.

As an input, we get the mapping of process P_i on computation node N_j^h ($\mathcal{M}(P_i) = N_j^h$) and the process failure probability p_{ijh} of process P_i, executed on computation node N_j^h with hardening level h.

In the analysis, first, we calculate the probability $Pr(0; N_j^h)$ of no faults occurring (no faulty processes) during one iteration of the application on the h-version of node N_j, which is the proba-

1. Numerical computations for this example with the SFP analysis are presented in Appendix III. A computational example with the SFP analysis is also presented in Section 9.2.1 of this chapter.

bility that all processes mapped on N_j^h will be executed correctly:

$$Pr(0;N_j^h) = \prod_{\forall P_i | M(P_i) = N_j^h} (1 - p_{ijh}) \qquad (9.1)$$

We call the *f*-fault scenario the *combination with repetitions* of *f* on $\Pi(N_j^h)$, where $\Pi(N_j^h)$ is the number of processes mapped on the computation node N_j^h. Under a combination with repetitions of n on m, we understand the process of selecting n elements from a set of m elements, where each element can be selected more than once and the order of selection does not matter [Sta97].

For example, an application $\mathcal{A}$ is composed of processes P_1, P_2 and P_3, which are mapped on node N_1. k_1= 3 transient faults may occur, i.e., f = 3 in the worst case. Let us consider one possible fault scenario, depicted in Figure 9.2. Process P_1 fails and is re-executed, its re-execution fails but then it is re-executed again without faults. Process P_2 fails once and is re-executed without faults. Thus, in this fault scenario, from a set of processes P_1, P_2 and P_3, processes P_1 and P_2 are selected; moreover, process P_1 is selected twice, which corresponds to repetition of process P_1. Thus, the combination is $\{P_1, P_1, P_2\}$.

The probability of recovering from a particular combination of *f* faults consists of two probabilities, the probability that this combination of *f* faults has happened and that all the processes, mapped on N_j, will be eventually (re-)executed without faults. The latter probability is, in fact, the no fault probability $Pr(0;$

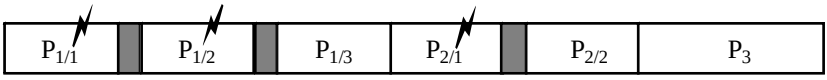


Figure 9.2: A Fault Scenario as a Combination with Repetitions

N_j^h). Thus, the probability of successful recovering from f faults in a particular fault scenario S^* is

$$Pr_{S^*}(f;N_j^h) = Pr(0;N_j^h) \cdot \prod_{s^* \in (S^*,m^*)} p_{s^*jh} \qquad (9.2)$$

where $|(S^*, m^*)| = f$, $(S^*, m^*) \subset (S, m)$, $S \subset \boldsymbol{N}$, $|S| = \Pi(N_j^h)$, $sup(m(a) \mid a \in S) = f$. The combination with repetitions is expressed here with a finite submultiset (S^*, m^*) of a multiset (S, m) [Sta97]. Informally, a multiset is simply a set with repetitions. Formally, in the present context, we define a multiset as a function $m: S \rightarrow \boldsymbol{N}$, on set S, which includes indices of all processes mapped on N_j^h, to the set $\boldsymbol{N}$ of (positive) natural numbers. For each process P_a with index a in S, the number of re-executions of process P_a in a particular fault scenario is the number $m(a)$, which is less or equal to f, the number of faults (expressed as a supremum of function $m(a)$). For example, in Figure 9.2, $m(1) = 2$ and $m(2) = 1$. The number of elements in S^* is f, e.g. the number of faulty processes. Thus, if a is repeated f times, $m(a) = f$, i.e., P_a fails f times, S^* will contain only repetitions of a and nothing else.

From (9.2), the probability that the system recovers from all possible f faults is a sum of probabilities of all f-fault recovery scenarios[1]:

$$Pr(f;N_j^h) = Pr(0;N_j^h) \cdot \sum_{(S^*,m^*) \subset (S,m)} \prod_{s^* \in (S^*,m^*)} p_{s^*jh} \qquad (9.3)$$

Suppose that we consider a situation with maximum k_j re-executions on the node N_j^h. The node fails if more than k_j faults are occurring. The failure probability of node N_j^h with k_j re-executions is

1. The combinations of faults in the re-executions are mutually exclusive.

$$Pr(f > k_j; N_j^h) = 1 - Pr(0; N_j^h) - \sum_{f=1}^{k_j} Pr(f; N_j^h) \qquad (9.4)$$

where we subtract from the initial failure probability with only hardware hardening, $1 - Pr(0; N_j^h)$, the probabilities of all the possible successful recovery scenarios provided with k_j re-executions.

Finally, the probability that the system composed of n computation nodes with k_j re-executions on each node N_j will not recover, in the case more than k_j faults have happened on any computation node N_j, can be obtained as follows:

$$Pr\left(\bigcup_{j=1}^{n} (f > k_j; N_j^h)\right) = 1 - \prod_{j=1}^{n} (1 - Pr(f > k_j; N_j^h)) \qquad (9.5)$$

According to the problem formulation, the system non-failure probability in the time unit τ (i.e., one hour) of functionality has to be equal or above the reliability goal $\rho = 1 - \gamma$, where γ is the maximum probability of a system failure due to transient faults within the time unit τ. Considering that the calculations above have been performed for one iteration of the application (i.e., within a period T), we obtain the following condition for our system to satisfy the reliability goal

$$\left(1 - Pr\left(\bigcup_{j=1}^{n} (f > k_j; N_j^h)\right)\right)^{\frac{\tau}{T}} \geq \rho \qquad (9.6)$$

9.2.1 COMPUTATION EXAMPLE

To illustrate how formulae (9.1)-(9.6) can be used in obtaining the number of re-execution to be introduced at software level, we will consider the architecture in Figure 9.3 for an application $\mathcal{A}_2$

(with all the necessary details provided in the figure). At first, we compute the probability of no faulty processes for both nodes N_1^2 and N_2^2:[1]

$Pr(0;N_1^2) = \lfloor(1 - 1.2\cdot10^{-5})\cdot(1 - 1.3\cdot10^{-5})\rfloor = 0.99997500015.$

$Pr(0;N_2^2) = \lfloor(1 - 1.2\cdot10^{-5})\cdot(1 - 1.3\cdot10^{-5})\rfloor = 0.99997500015.$

According to formulae (9.4) and (9.5),

$Pr(f > 0;\ N_1^2) = 1 - 0.99997500015 = 0.00002499985.$

$Pr(f > 0;\ N_2^2) = 1 - 0.99997500015 = 0.00002499985.$

$Pr((f > 0;\ N_1^2) \cup (f > 0;\ N_2^2)) = \lceil 1 - (1 - 0.00002499985) \cdot (1 - 0.00002499985)\rceil = 0.00004999908.$

The system period T is 360 ms, hence system reliability is $(1 - 0.00004999908)^{10000} = 0.60652865819$, which means that the system does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now consider $k_1 = 1$ and $k_2 = 1$:

$Pr(1;N_1^2) = \lfloor 0.99997500015\cdot(1.2\cdot10^{-5} + 1.3\cdot10^{-5})\rfloor = 0.00002499937.$

$Pr(1;N_2^2) = \lfloor 0.99997500015\cdot(1.2\cdot10^{-5} + 1.3\cdot10^{-5})\rfloor = 0.00002499937.$

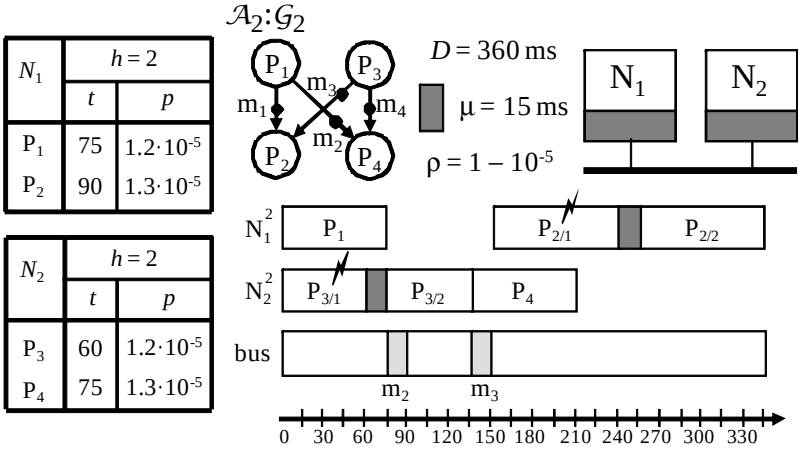


Figure 9.3: Computation Example with SFP Analysis

1. Symbols $\lceil$ and $\rceil$ indicate that numbers are *rounded up* with 10^{-11} accuracy; $\lfloor$ and $\rfloor$ indicate that numbers are *rounded down* with 10^{-11} accuracy. This is needed in order to keep the safeness of the probabilities calculated.

According to formulae (9.4) and (9.5),

$Pr(f>1;N_1^2) = (1 - 0.99997500015 - 0.00002499937) = 4.8 \cdot 10^{-10}$.

$Pr(f>1;N_2^2) = (1 - 0.99997500015 - 0.00002499937) = 4.8 \cdot 10^{-10}$.

$Pr((f > 1; N_1^2) \cup (f > 1; N_2^2)) = 9.6 \cdot 10^{-10}$.

Hence, the system reliability is $(1 - 9.6 \cdot 10^{-10})^{10000} = 0.99999040004$ and the system meets its reliability goal $\rho = 1 - 10^{-5}$. Thus, in order to meet its reliability requirements, one transient fault has to be tolerated on each computation node during one system period.

9.3 Conclusions

In this chapter we have illustrated the trade-off between increasing hardware reliability by hardening and tolerating transient faults at the software level. We have also proposed a system failure probability (SFP) analysis to evaluate if the system meets its reliability goal ρ with the given reliability of hardware components and the given number of re-executions in software. This analysis will play an important role in our optimization framework presented in the next chapter.

Chapter 10
Optimization with Hardware Hardening

IN THIS CHAPTER we propose an approach to design optimization of fault-tolerant hard real-time embedded systems, which combines hardware and software fault tolerance techniques. We consider the trade-off between hardware hardening and process re-execution to provide the required levels of fault tolerance against transient faults with the lowest-possible system hardware cost. We present design optimization heuristics to select the fault-tolerant architecture and decide process mapping such that the system cost is minimized, deadlines are satisfied, and the reliability requirements are fulfilled. Our heuristic will use the system failure probability (SFP) analysis, presented in Chapter 9, to evaluate the system reliability.

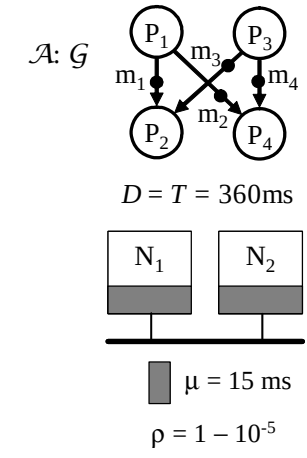
10.1 Motivational Example

In Figure 10.1 we consider several architecture selection alternatives in the distributed context, for the application $\mathcal{A}$, composed of four processes, which can be mapped on three h-versions of two nodes N_1 and N_2.

The cheapest two-node solution that meets the deadline and reliability goal is depicted in Figure 10.1a. The architecture consists of the h-versions N_1^2 and N_2^2 and costs 32 + 40 = 72 monetary units. Based on the SFP analysis, the reliability goal can be achieved with one re-execution on each computation node.[1] Let us now evaluate some possible monoprocessor architectures. With the architecture composed of only N_1^2 and the schedule presented in Figure 10.1b, according to the SFP analysis, the reliability goal is achieved with k_1 = 2 re-executions at software level.[2] As can be seen in Figure 10.1b, the application is unschedulable in this case. Similarly, the application is also unschedulable with the architecture composed of only N_2^2, with the schedule presented in Figure 10.1c. Figure 10.1d and Figure 10.1e depict the solutions obtained with the monoprocessor architecture composed of the most hardened versions of the nodes. In both cases, the reliability goal ρ is achieved without re-executions at software level ($k_j = 0$), however, as can be observed, the solution in Figure 10.1d is not schedulable, even though k_1 = 0 with the architecture consisting of N_1^3. This is because of the performance degradation due to the hardening. This degradation, however, is smaller in the case of N_2^3 and, thus, the solution in Figure 10.1e is schedulable. If we compare the two schedulable alternatives in Figure 10.1a and Figure 10.1e, we observe that the one consisting of less hardened nodes (Figure 10.1a) is more cost efficient than the monoprocessor alternative with the

1. See numerical computations for this architecture in Section 9.2.1, presented as a computation example with the SFP analysis.
2. Numerical computations for all architectures in this example are presented in Appendix III.

most hardened node (Figure 10.1e). In other words, considering a distributed system with two less hardened computation nodes



N_1	h = 1		h = 2		h = 3	
	t	p	t	p	t	p
P_1	60	$1.2 \cdot 10^{-3}$	75	$1.2 \cdot 10^{-5}$	90	$1.2 \cdot 10^{-10}$
P_2	75	$1.3 \cdot 10^{-3}$	90	$1.3 \cdot 10^{-5}$	105	$1.3 \cdot 10^{-10}$
P_3	60	$1.4 \cdot 10^{-3}$	75	$1.4 \cdot 10^{-5}$	90	$1.4 \cdot 10^{-10}$
P_4	75	$1.6 \cdot 10^{-3}$	90	$1.6 \cdot 10^{-5}$	105	$1.6 \cdot 10^{-10}$
Cost	16		32		64	

N_2	h = 1		h = 2		h = 3	
	t	p	t	p	t	p
P_1	50	$1 \cdot 10^{-3}$	60	$1 \cdot 10^{-5}$	75	$1 \cdot 10^{-10}$
P_2	65	$1.2 \cdot 10^{-3}$	75	$1.2 \cdot 10^{-5}$	90	$1.2 \cdot 10^{-10}$
P_3	50	$1.2 \cdot 10^{-3}$	60	$1.2 \cdot 10^{-5}$	75	$1.2 \cdot 10^{-10}$
P_4	65	$1.3 \cdot 10^{-3}$	75	$1.3 \cdot 10^{-5}$	90	$1.3 \cdot 10^{-10}$
Cost	20		40		80	

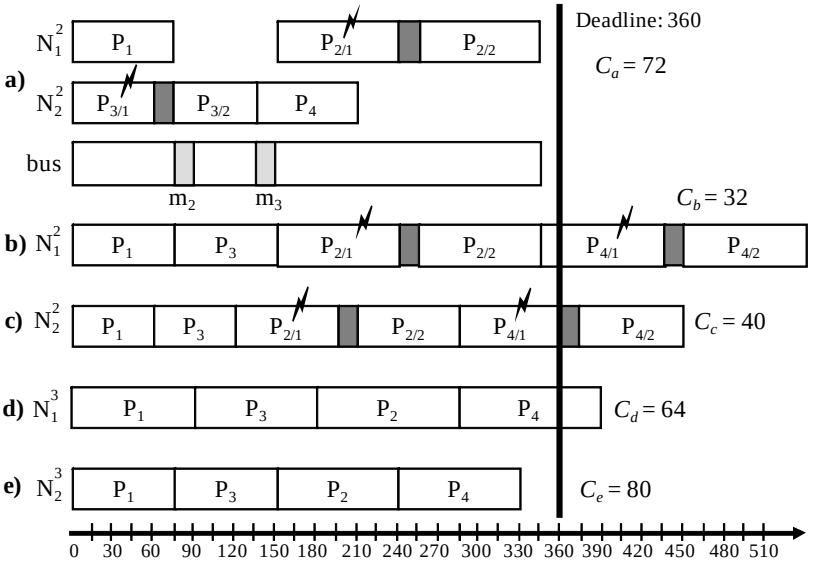


Figure 10.1: Selection of the Hardened Hardware Architecture

is less costly than considering a single node with extensive hardening.

As the example in Figure 10.1 illustrates, the decision on which architecture to select and how much hardening to use for each computation node in the architecture is crucial in providing cost-efficient design solutions with fault tolerance.

10.2 Problem Formulation

As an input to our design optimization problem we get an application $\mathcal{A}$, represented as a merged acyclic directed graph $\mathcal{G}$. Application $\mathcal{A}$ runs on a bus-based architecture, as discussed in Section 3.1.3, with hardened computation nodes as discussed in Section 9.1. The reliability goal ρ, the deadline, and the recovery overhead μ are given. Given is also a set of available computation nodes each with its available hardened h-versions and the corresponding costs. We know the worst-case execution times and the failure probabilities for each process on each h-version of computation node. The maximum transmission time of all messages, if sent over the bus, is given.

As an output, the following has to be produced: (1) a selection of the computation nodes and their hardening level; (2) a mapping of the processes to the nodes of the selected architecture; (3) the maximum number of re-executions on each computation node; and (4) a schedule of the processes and communications.

The selected architecture, the mapping and the schedule should be such that the total cost of the nodes is minimized, all deadlines are satisfied, and the reliability goal ρ is achieved. Achieving the reliability goal implies that hardening levels are selected and the number of re-executions are chosen on each node N_j such that the produced schedule, in the worst case, satisfies the deadlines.

10.3 Design Strategy and Algorithms

We illustrate our hardening optimization framework in Figure 10.2 as a sequence of iterative design optimization heuristics, represented as rotating circles in the figure. In the outer loop we explore available architectures, e.g., possible combinations of computation nodes. We evaluate the schedulability of each selected architecture and check if it meets the reliability goal ρ against transient faults. We also determine the cost of the architecture. The least costly and valid architecture will be eventually chosen as the final result of the overall design exploration.

From the outer *architecture optimization heuristic* for each chosen architecture, we call a *mapping heuristic*, which returns a valid mapping with schedule tables, and obtains the amount of hardware hardening and the number of re-executions. In the mapping heuristic, for each mapping move, we obtain the amount of hardening required for the mapping solution such that it is schedulable and meets the reliability goal ρ.

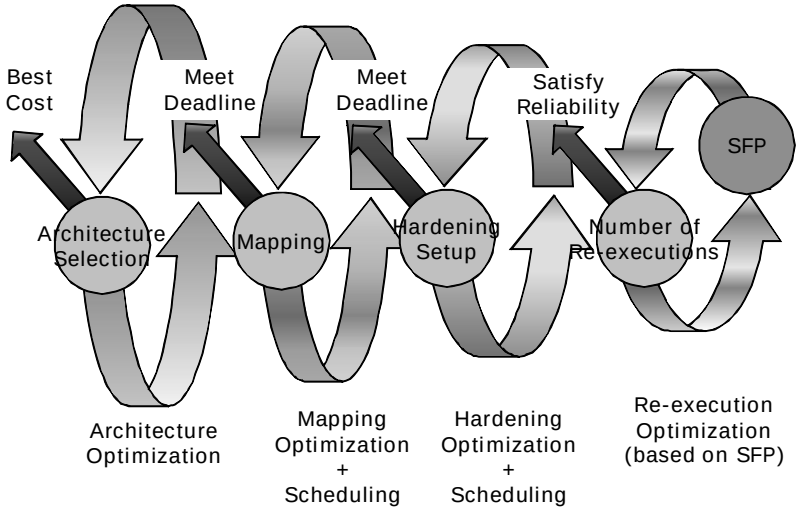


Figure 10.2: Optimization with Hardware Hardening

The amount of hardening, or hardening levels of computation nodes, are determined by the *hardening optimization heuristic*, which increases the reliability of computation nodes until the current mapping solution becomes schedulable.

Inside the hardening optimization heuristic, for each hardened and mapped solution, we obtain the necessary number of re-executions on each computation node with the *re-execution optimization heuristic*. This, innermost, heuristic assigns the number of re-executions to application processes until the reliability goal ρ is reached, which is evaluated with the *system failure probability* (SFP) analysis.

Our design strategy is outlined in Figure 10.3, which is the outermost architecture optimization heuristic in Figure 10.2. The design heuristic explores the set of architectures, and eventually selects that architecture that minimizes cost, while still meeting the schedulability and reliability requirements of the application. The heuristic starts with the monoprocessor architecture ($n = 1$), composed of only one (the fastest) node (lines 1-2). The mapping, selection of software and hardware fault tolerance (re-executions and hardening levels) and the schedule are obtained for this architecture (lines 5-9). If the application is *unschedulable*, the number of computation nodes is directly *increased*, and the fastest architecture with $n = n + 1$ nodes is chosen (line 15). If the application is schedulable on that architecture with n nodes, i.e., $SL \leq D$, the cost $HWCost$ of that architecture is stored as the best-so-far cost $HWCost_{best}$. The next fastest architecture with n nodes (in the case of no hardening) is then selected (line 18). If on that architecture the application is schedulable (after hardening is introduced) and the cost $HWCost < HWCost_{best}$, it is stored as the best-so-far. The procedure continues until the architecture with the maximum number of nodes is reached and evaluated.

If the cost of the next selected architecture with the minimum hardening levels is higher than (or equal to) the best-so-far cost $HWCost_{best}$, the architecture will be *ignored* (line 6).

The evaluation of an architecture is done at each iteration step with the MappingAlgorithm function, presented in Section 10.4. MappingAlgorithm receives as an input the selected architecture, produces the mapping, and returns the schedule corresponding to that mapping. The cost function used for optimization is also given as a parameter. We use two cost functions: (1) schedule length, which produces the shortest-possible schedule length SL for the selected architecture for the best-possible mapping (line 7), and (2) architecture hardware cost, in which the mapping algorithm takes an already schedulable application as an input

```
DesignStrategy(G, N, D, ρ)
1  n = 1
2  AR = SelectArch(N, n)
3  HWCost_best = MAX_COST
4  while n ≤ |N| do
5    SetMinHardening(AR)
6    if HWCost_best > GetCost(AR) then
7      SL = MappingAlgorithm(G,AR,D,ρ, ScheduleLengthOptimization)
8      if SL ≤ D then
9        HWCost = MappingAlgorithm(G, AR, D,ρ, CostOptimization)
10       if HWCost < HWCost_best then
11         HWCost_best = HWCost
12         AR_best = AR
13       end if
14     else
15       n = n + 1
16     end if
17   end if
18   AR = SelectNextArch(N, n)
19 end while
20 return AR_best
end DesignStrategy
```

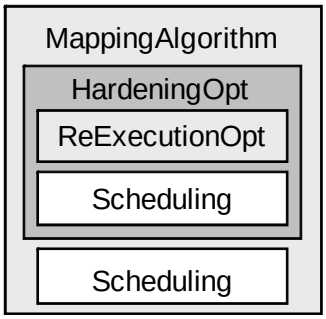


Figure 10.3: General Design Strategy with Hardening

and then optimizes the mapping to improve the cost of the application without impairing the schedulability (line 9). MappingAlgorithm tries a set of possible mappings, and for each mapping it optimizes the levels of fault tolerance in software and hardware, which are required to meet the reliability goal ρ. The levels of fault tolerance are optimized inside the mapping algorithm with the HardeningOpt heuristic presented in Section 10.5, which returns the levels of hardening and the number of re-executions in software. The relation between these functions is illustrated in Figure 10.3. The needed maximum number of re-executions in software is obtained with ReExecutionOpt heuristic, called inside HardeningOpt for each vector of hardening levels. Then the obtained alternative of fault tolerance levels is evaluated in terms of schedulability by the scheduling algorithm Scheduling, which is described in Section 10.6. After completion of HardeningOpt, Scheduling is called again to determine the schedule for each selected mapping alternative in MappingAlgorithm.

The basic idea behind our design strategy is that the change of the mapping immediately triggers the change of the hardening levels. To illustrate this, let us consider the application $\mathcal{A}$ in Figure 10.1 with the mapping corresponding to Figure 10.1a. Processes P_1 and P_2 are mapped on N_1, while processes P_3 and P_4 are mapped on N_2. Both nodes, N_1 and N_2, have the second hardening level (h = 2), N_1^2 and N_2^2. With this architecture, according to our SFP calculation, one re-execution is needed on each node in order to meet the reliability goal. As can be seen in Figure 10.1a, the deadlines are satisfied in this case. If, however, processes P_1 and P_2 are moved to node N_2, resulting in the mapping corresponding to Figure 10.1e, then using the third hardening level (h = 3) is the only option to guarantee the timing and reliability requirements, and this alternative will be chosen by our algorithm for the respective mapping. If, for a certain mapping, the application is not schedulable with any available hardening level, this mapping will be discarded by our algorithm.

10.4 Mapping Optimization

In our design strategy we use the MappingAlgorithm heuristic with two cost functions, schedule length and the architecture hardware cost, as presented in Figure 10.4.

For our mapping heuristic with hardened components, we have extended the TabuSearchMPA algorithm, proposed in Chapter 5, to consider the different hardening and re-execution levels. The tabu search algorithm takes as an input the application graph $\mathcal{G}$, the selected architecture $\mathcal{AR}$, deadlines $\mathcal{D}$, reliability goal ρ and a flag *Type*, which specifies if the mapping algorithm is used for schedule length optimization or for the architecture cost optimization, and produces a schedulable and fault-tolerant implementation x^{best}. In case of the schedule length optimization, an initial mapping is performed (InitialMapping, line 5) and the obtained initial schedule length for this mapping is set as the initial *Cost* (Scheduling, line 5). In the case of architecture hardware cost optimization, the initial *Cost* will be the given hardware cost (ObtainCost, line 7) and the initial mapping, best in terms of the schedule length but not in terms of cost, is already prepared (see our strategy in Figure 10.3). During architecture cost optimization, our mapping algorithm will search for the mapping with the lowest-possible architecture hardware cost.

The mapping heuristic, in both cases, investigates the processes on the critical path (line 12). Thus, at each iteration, processes on the critical part are selected for the re-mapping (GenerateMoves, line 12). Processes recently re-mapped are marked as "tabu" (by setting up the "tabu" counter) and are not touched unless leading to the solution better than the best so far (lines 14-15). Processes, which have been waiting for a long time to be re-mapped, are assigned with the waiting priorities and will be re-mapped first (line 17). The heuristic changes the mapping of a process if it leads to (1) a solution that is better than the best-so-far (including "tabu" processes), or (2) to a solution that is worse than the best-so-far but is better than the

other possible solutions (lines 20-25). The selected move is then applied (PerformMove, line 27), the current *Cost* is updated (lines 28-30) and the best-so-far *BestCost* is changed if necessary (line 32). At every iteration, the waiting counters are increased and

```
MappingAlgorithm(𝒢, 𝒜ℛ, 𝒟, ρ, Type {CostOptimization, ScheduleLengthOptimization})
 1  -- given a merged graph 𝒢 and an architecture 𝒜ℛ produces a mapping ℳ
 2  -- such that the solution is fault-tolerant, schedulable & meets reliability goal ρ
 3  -- optimizes either Type = "CostOptimization" or "ScheduleLengthOptimization"
 4  if Type ≡ ScheduleLengthOptimization then
        -- for the schedule length, do initial mapping
 5      x^best = x^now = InitialMapping(𝒜ℛ); BestCost = Scheduling(𝒢, 𝒜ℛ, x^best)
 6  else -- otherwise, get the previous mapping best in terms of the schedule length
 7    x^best = x^now = GetCurrentMapping(𝒜ℛ); BestCost = ObtainCost(𝒢, 𝒜ℛ, x^best)
 8  end if
 9  Tabu = ∅; Wait = ∅ -- The selective history is initially empty
10  while TerminationCondition not satisfied do
11    -- Determine the neighboring solutions considering the selective history
12    CP= CriticalPath(𝒢); N^now = GenerateMoves(CP) -- calls HardeningOpt inside
13    -- eliminate tabu moves if they are not better than the best-so-far
14    N^tabu = {move(P_i) | ∀ P_i ∈ CP ∧ Tabu(P_i)=0 ∧ Cost(move(P_i)) < BestCost}
15    N^non-tabu = N \ N^tabu
16    -- add diversification moves
17    N^waiting = {move(P_i) | ∀ P_i ∈ CP ∧ Wait(P_i) > |𝒢|}
18    N^now = N^non-tabu ∪ N^waiting
19    -- Select the move to be performed
20    x^now = SelectBest(N^now)
21    x^waiting = SelectBest(N^waiting); x^non-tabu = SelectBest(N^non-tabu)
22    if Cost(x^now) < BestCost then x = x^now -- select x^now if better than best-so-far
23      else if ∃ x^waiting then x = x^waiting -- otherwise diversify
24      else x = x^non-tabu -- if no better and no diversification, select best non-tabu
25    end if
26    -- Perform selected mapping move and determine levels of hardening
27    PerformMove(𝒜ℛ, x); HardeningOpt(𝒢, 𝒜ℛ, x, ρ, InitialHardening(x))
28    if Type ≡ ScheduleLengthOptimization then Cost = Scheduling(𝒢, 𝒜ℛ, x)
29    else Cost = ObtainCost(𝒢, 𝒜ℛ, x)
30    end if
31    -- Update the best-so-far solution and the selective history tables
32    If Cost < BestCost then x^best = x; BestCost = Cost end if
33    Update(Tabu); Update(Wait)
34  end while
35  return x^best -- return mapping ℳ
end MappingAlgorithm
```

Figure 10.4: Mapping Optimization with Hardening

the "tabu" counters are decreased (line 33). The heuristic stops after a certain number of steps without any improvement (*TerminationCondition*, line 10).

In order to evaluate a particular mapping in terms of cost and schedulability, for this mapping, before calling the scheduling heuristic, we have to obtain the hardening levels in hardware and the maximum number of re-executions in software. This is performed with the HardeningOpt function, presented in the next section. HardeningOpt is called inside the GenerateMoves function evaluating each available mapping move on the critical path (line 12) and before the "best" move is applied (line 27).

10.5 Hardening/Re-execution Optimization

Every time we evaluate a mapping move by the MappingAlgorithm, we run HardeningOpt and ReExecutionOpt to obtain hardening levels in hardware and the number of re-executions in software. The heuristic, outlined in Figure 10.5, takes as an input the architecture $\mathcal{AR}$ with the initial hardening levels H_0 and the given mapping $\mathcal{M}$. The obtained final hardening solution has to meet the reliability goal ρ.

At first, the hardening optimization heuristic reaches our first objective, *schedulability* of the application. The heuristic increases the schedulability by increasing the hardening levels in a greedy fashion, obtaining the number of re-executions for each vector of hardening (lines 1-9, with the IncreaseAllHardening function). The schedulability is evaluated with the Scheduling heuristic. We increase the hardening in this way to provide a larger number of alternatives to be explored in the next optimization step, where we preserve the schedulability and optimize cost. In the next step, once a schedulable solution is reached, we iteratively reduce hardening by one level for each node, again, at the same time obtaining the corresponding number of re-executions (lines 9-24). For example, in Figure 10.1a, we can reduce

from N_1^2 to N_1^1, and from N_2^2 to N_2^1. If the application is schedulable, as obtained with the Scheduling function (line 15), such a solution is accepted (line 16). Otherwise, it is not accepted, for example, in the case we reduce from N_1^2 to N_1^1. Among the schedulable hardened alternatives, we choose the one with the lowest hardware cost and continue (line 20 and 22). The heuristic iter-

```
HardeningOpt(G, AR, M, ρ, H0): const Hmax, Hmin
1  REX0 = ReExecutionOpt(G, AR, M, ρ, H0)
2  SL = Scheduling(G, AR, M, REX0)
3  if SL ≤ D then
4    H = H0
5  else
6    H = IncreaseAllHardening(AR, H0)
7    if H > Hmax then
8      return unschedulable
9  end if
10 while true do
11   for all Nj ∈ N do
12     if hj > Hmin then
13       ReduceHardening(hj)
14       REX = ReExecutionOpt(G, AR, M, ρ, H)
15       SLj = Scheduling(G, AR, M, REX)
16       if SLj ≤ D thenHWCostj = GetCost(AR, H)
17       IncreaseHardening(hj)
18     end if
19   end for
20   best = BestCostResource(∀HWCostj)
21   if no improvement or H ≡ Hmin then break
22   else ReduceHardening(hbest)
23   end if
24 end while
25 REX = ReExecutionOpt(G, AR, M, ρ, H)
26 return {REX; H}
end HardeningOpt
```

Figure 10.5: Hardening Optimization Algorithm

ates as long as improvement is possible, i.e., there is at least one schedulable alternative, or until the minimum hardening level (H_{min}) is reached for all the nodes (line 21). In Figure 10.1a, the heuristic will stop once h-versions N_1^2 to N_2^2 have been reached since the solutions with less hardening are not schedulable.

The re-execution optimization heuristic ReExecutionOpt is called in every iteration of HardeningOpt to obtain the number of re-executions in software (line 14 and inside the IncreaseAllHardening function, line 6). The heuristic takes as an input the architecture $\mathcal{AR}$, mapping $\mathcal{M}$, and the hardening levels H. It starts without any re-executions in software at all. The heuristic uses the SFP analysis and gradually increases the number of re-executions until the reliability goal ρ is reached. The exploration of the number of re-executions is guided towards the largest increase in the system reliability. For example, if increasing the number of re-executions by one on node N_1 will increase the system reliability from $1- 10^{-3}$ to $1- 10^{-4}$ and, at the same time, increasing re-executions by one on node N_2 will increase the system reliability from $1- 10^{-3}$ to $1- 5{\cdot}10^{-5}$, the heuristic will choose to introduce one more re-execution on node N_2.

10.6 Scheduling

In our hardening optimization framework we use the shifting-based scheduling strategy, which we have proposed in Chapter 4, that uses “recovery slack” in order to accommodate the time needed for re-executions in the case of faults. After each process P_i we assign a slack equal to $(t_{ijh} + \mu) \times k_j$, where k_j is the number of re-executions on the computation node N_j with hardening level h, t_{ijh} is the worst-case execution time of the process on this node, and μ is the re-execution overhead. The recovery slack is also shared between processes in order to reduce the time allocated for recovering from faults.

The Scheduling heuristic is used by the HardeningOpt and the mapping optimization heuristics to determine the schedulability of the evaluated solution, and produces the best possible schedule for the final architecture.

10.7 Experimental Results

For the experiments, we have generated 150 synthetic applications with 20 and 40 processes. The worst-case execution times (WCETs) of processes, considered on the fastest node without any hardening, have been varied between 1 and 20 ms. The recovery overhead μ has been randomly generated between 1 and 10% of process WCET.

Regarding the architecture, we consider nodes with five different levels of hardening with randomly generated process failure probabilities. We have considered three fabrication technologies with *low*, *medium* and *high* transient fault rates, where the technology with the *high* fault rate (or, simply, the *high FR technology*) has the highest level of integration and the smallest transistor sizes. We have generated process failure probabilities within the following probability intervals for these technologies:
low FR technology: between $2.1 \cdot 10^{-9}$ and $2.4 \cdot 10^{-5}$;
medium FR technology: between $2.1 \cdot 10^{-8}$ and $2.4 \cdot 10^{-4}$; and
high FR technology: between $2.1 \cdot 10^{-7}$ and $2.4 \cdot 10^{-3}$.

The hardening performance degradation (*HPD*) from the *minimum* to the *maximum* hardening level has been varied from 5% to 100%, increasing linearly with the hardening level. For a *HPD* of 5%, the WCET of processes increases with each hardening level with 1, 2, 3, 4, and 5%, respectively; for *HPD* = 100%, the increases will be 1, 25, 50, 75, and 100% for each level, respectively. Initial costs of computation nodes (without hardening) have been generated between 1 and 6 cost units. We have assumed that the hardware cost increases linearly with the

hardening level. The system reliability requirements have been varied between $\rho = 1 - 7.5 \cdot 10^{-6}$ and $1 - 2.5 \cdot 10^{-5}$ within one hour. The deadlines have been assigned to all the applications independent of the transient fault rates and hardening performance degradation of the computation nodes. The experiments have been run on a Pentium 4 2.8 GHz processor with 1Gb memory.

In our experimental evaluation, we have compared our design optimization strategy from Section 10.3, denoted OPT, to two strategies, in which the hardening optimization step has been removed from the mapping algorithms. In the first strategy, denoted MIN, we use only computation nodes with the minimum hardening levels. In the second strategy, denoted MAX, only the computation nodes with the maximum hardening levels are used.

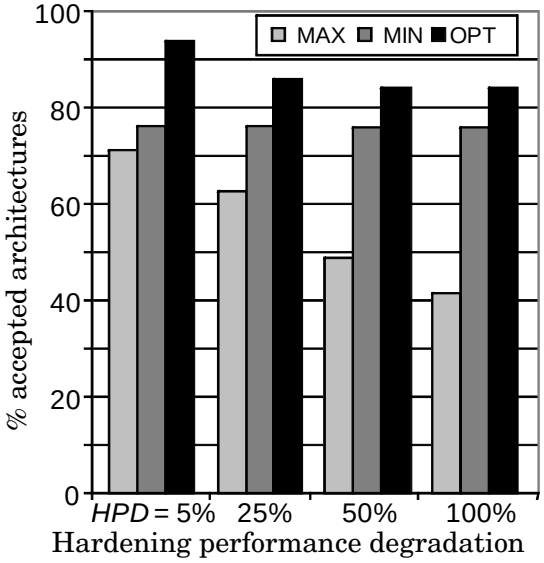


Figure 10.6: % Accepted Architectures as a Function of Hardening Performance Degradation

The experimental results are presented in Figure 10.6, Table 10.1 and Figure 10.7, which demonstrate the efficiency of our design approaches in terms of the applications (in percentage) *accepted* out of all considered applications. By the *acceptable* application we mean an application that meets its reliability goal, is schedulable, and does not exceed the maximum architectural cost (*ArC*) imposed. In Figure 10.6, for the *medium FR* technology and *ArC* = 20 units, we show how our strategies perform with an increasing performance degradation due to hardening. The MIN strategy always provides the same result because it uses the nodes with the minimum hardening levels and applies only software fault tolerance techniques. The efficiency of the MAX strategy is lower than for MIN and is further reduced with the increase of performance degradation. The OPT gives 18% improvement on top of MIN, if *HPD* = 5%, 10% improvement if *HPD* = 25%, and 8% improvement for 50% and 100%. More detailed results for *ArC* = 15 and *ArC* = 25 cost units are shown in Table 10.1, which demonstrate similar trends.

In Figure 10.7a and Figure 10.7b, we illustrate the performance of our design strategies for the three different technologies and their corresponding fault rates. The experiments in Figure 10.7a have been performed for *HPD* = 5%, while the ones in Figure 10.7b correspond to *HPD* = 100%. The maximum

Table 10.1: % Accepted Architectures with Different Hardening Performance Degradation (*HPD*) and with Different Maximum Architecture Costs (*ArC*) for the *Medium FR* Technology

	HPD = 5%			*HPD* = 25%			*HPD* = 50%			*HPD* = 100%		
ArC	**15**	**20**	**25**	**15**	**20**	**25**	**15**	**20**	**25**	**15**	**20**	**25**
MAX	35	71	92	33	63	84	27	49	74	23	41	65
MIN	76	76	82	76	76	82	76	76	82	76	76	82
OPT	92	94	98	86	86	92	80	84	90	78	84	90

architectural cost is 20 units. In the case of the *low FR* technology, the MIN strategy is as good as our OPT due to the fact that the reliability requirements can be achieved exclusively with only software fault tolerance techniques. However, for the *medium FR* technology, our OPT strategy outperforms MIN. For the *high FR* technology, OPT is significantly better than both other strategies since, in this case, finding a proper trade-off between the levels of hardening in hardware and the levels of software re-execution becomes more important.

The execution time of our OPT strategy for the examples that have been considered is between 3 minutes and 60 minutes.

We have also run our experiments on the vehicle cruise controller (CC) composed of 32 processes, previously used to evaluate several other optimization heuristics presented in the thesis.

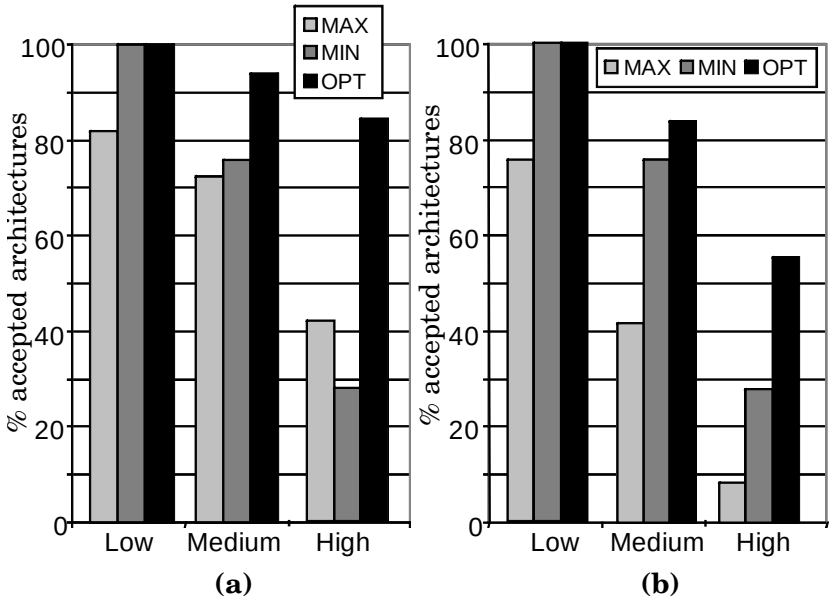


Figure 10.7: % Accepted Architectures for Different Fault Rates with *ArC* = 20 for **(a)** *HPD* = 5% and **(b)** *HPD* = 100%

The CC considers an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Braking System (ABS) and Transmission Control Module (TCM). We have set the system reliability requirements to $\rho = 1 - 1.2 \cdot 10^{-5}$ within one hour and considered μ between 1 and 10% of process average-case execution times. The process failure probabilities have been generated between $4.5 \cdot 10^{-8}$ and $5 \cdot 10^{-5}$; five *h*-versions of the computation nodes have been considered with *HPD* = 25% and linear cost functions. We have considered a deadline of 300 ms. We have found that CC is not schedulable if the MIN strategy has been used. However, the MAX and OPT approaches are able to produce a schedulable solution. Moreover, our OPT strategy with the trading-off between hardware and software fault tolerance levels has produced results 66% better than the MAX in terms of cost.

10.8 Conclusions

In this chapter we have considered hard real-time applications mapped on distributed embedded architectures. We were interested to derive the least costly implementation that meets imposed timing and reliability constraints. We have considered two options for increasing the reliability: hardware hardening and software re-execution.

We have proposed a design optimization framework for minimizing of the hardware cost by trading-off between hardware hardening and software re-execution. Our experimental results have shown that, by selecting the appropriate level of hardening in hardware and re-executions in software, we can satisfy the reliability and time constraints of the applications while minimizing the hardware cost of the architecture. The optimization relies on a system failure probability (SFP) analysis, which connects the level of hardening in hardware with the number of re-executions in software.

PART V
Conclusions and Future Work

Chapter 11
Conclusions and Future Work

IN THIS THESIS we have presented several strategies for design optimization and scheduling of distributed fault-tolerant embedded systems. We have considered hard real-time and mixed soft and hard real-time embedded systems.

In the context of hard real-time systems, we have proposed two scheduling techniques, as well as mapping and policy assignment approaches. We have also taken into account debugability and testability properties of fault-tolerant applications by considering transparency requirements. We have proposed scheduling and mapping approaches that can handle transparency as well as the trade-off transparency vs. performance.

In the context of mixed soft and hard real-time systems, we have proposed a value-based scheduling approach, which generates a set of trees that guarantee the deadlines for the hard processes even in the case of faults while maximizing the overall utility.

In the context of embedded systems with hardened hardware components, we have proposed a design optimization approach

to trade-off hardware hardening and software re-execution to provide cost-efficient, schedulable and reliable implementations. We have also proposed an analysis approach to support our design optimization, which connects hardening levels with the number of re-executions, and determines if the obtained system architecture meets the desired reliability goal.

In this final chapter, we summarize the work presented in the thesis and point out ideas for future work.

11.1 Conclusions

In this thesis we have considered real-time systems, where the hardware architecture consists of a set of heterogeneous computation nodes connected to a communication channel. The real-time application is represented as a set of processes communicating with messages. The processes are scheduled based on off-line generated quasi-static schedules. To provide fault tolerance against transient faults propagated to the software level, processes are assigned with re-execution, replication, or recovery with checkpointing. To increase the reliability of hardware components, hardening techniques are used.

All proposed algorithms have been implemented and evaluated on numerous synthetic applications and a real-life example.

In this section we will present conclusions for each part of the thesis in more details.

11.1.1 HARD REAL-TIME SYSTEMS

Scheduling. In the context of hard real-time systems we have proposed two novel scheduling approaches with fault tolerance: conditional scheduling and shifting-based scheduling. These approaches allow us to efficiently incorporate fault tolerance against multiple transient faults into static cyclic schedules.

The main contribution of the first approach is the ability to handle performance versus transparency and memory size trade-offs. Our conclusion is that this scheduling approach also generates the most efficient schedules.

The second scheduling approach handles only a fixed transparency setup, transparent recovery, where all messages on the bus have to be sent at fixed times, regardless of fault occurrences. The order of processes on computation nodes is also fixed in all alternative execution scenarios. Based on our investigations, we conclude that this scheduling approach is much faster than the conditional scheduling approach and requires less memory to store the generated schedule tables.

Mapping and fault tolerance policy assignment. We have developed several algorithms for policy assignment and process mapping, including mapping with performance/transparency trade-off.

At fault tolerance policy assignment, we decide on which fault tolerance technique or which combination of techniques to assign to a certain process in the application. The fault tolerance technique can be either rollback recovery, which provides time-redundancy, or active replication, which provides space-redundancy. We have implemented a tabu search-based optimization approach that decides the mapping of processes to the nodes in the architecture and the assignment of fault tolerance policies to processes.

According to our evaluations, the proposed approach can efficiently optimize mapping and policy assignment and is able to provide schedulable solutions under limited amount of resources.

Transparency/performance trade-off. In our scheduling and mapping optimization techniques, we can handle transparency requirements imposed on the application. Transparency is an important property that makes a system easier to observe and debug. The amount of memory required to store alternative

schedules is also smaller. However, transparency may lead to significant performance overheads. Thus, we have proposed a fine-grained approach to transparency, where transparency can be selectively applied to processes and messages. Our conclusion is that the performance/transparency trade-offs imposed by designers can be supported during the design process.

Checkpoint optimization. We have also addressed the problem of checkpoint optimization. The conclusion of this study is that by globally optimizing the number of checkpoints, as opposed to the approach when processes are considered in isolation, significant improvements can be achieved. We have also integrated checkpoint optimization into a fault tolerance policy assignment and mapping optimization strategy, and an optimization algorithm based on tabu search has been implemented.

11.1.2 MIXED SOFT AND HARD REAL-TIME SYSTEMS

In the context of mixed soft and hard real-time applications, the timing constraints have been captured using deadlines for hard processes and time/utility functions for soft processes.

We have proposed an approach to the synthesis of fault-tolerant schedules for mixed hard/soft applications. Our quasi-static scheduling approach guarantees the deadlines for the hard processes even in the case of faults, while maximizing the overall utility of the system. In the context of *distributed embedded systems*, depending on the current execution situation and fault occurrences, an online distributed scheduler chooses which schedule to execute on a particular computation node, based on the pre-computed switching conditions broadcasted with the signalling messages over the bus.

This study demonstrates that the obtained tree of fault-tolerant schedules can deliver an increased level of quality-of-service and guarantee timing constraints of safety-critical distributed applications under limited amount of resources.

11.1.3 EMBEDDED SYSTEMS WITH HARDENED HARDWARE COMPONENTS

In this thesis, we have also presented an approach, considering hard real-time applications, where the reliability of hardware components is increased with hardening. Re-executions have been used in software to further increase the overall system reliability against transient faults. We have proposed the system failure probability (SFP) analysis to evaluate if the system meets its reliability goal ρ with the given reliability of hardware components and the given number of re-executions in software.

We were interested to derive the least costly implementation that meets the imposed timing and reliability constraints. We have proposed a design optimization framework for minimizing of the overall system cost by trading-off between hardware hardening and software re-execution. The optimization relies on the system failure probability (SFP) analysis, which connects the level of hardening in hardware with the number of re-executions in software.

Based on our experimental results, we conclude that, by selecting the appropriate level of hardening in hardware and re-executions in software, we can satisfy the reliability and time constraints of the applications while minimizing the hardware cost of the architecture.

11.2 Future Work

The work that has been presented in this thesis can be used as a foundation for future research in the area of design optimization of fault-tolerant embedded systems. We see several directions for the future work based on this thesis.

Failure probability analysis. Our system failure probability (SFP) analysis is safe but is very time-consuming since it is based on exact mathematical computations.

Development of a fast probability calculation, which is not exact but still safe, could be an important contribution to hardware/software design for fault tolerance. Extension of the SFP analysis to replication and rollback recovery with checkpointing is another interesting direction of future work.

Relaxing assumptions regarding fault tolerance mechanisms. In this thesis we have considered that fault tolerance mechanisms are themselves fault-tolerant, i.e., they use their own internal fault tolerance techniques for self-protection against transient faults. However, the costs implied by these techniques can be high. This cost can be, potentially, reduced by considering "imperfect" fault tolerance mechanisms that would allow a fault to happen during, for example, recovering or storing a checkpoint. Trading-off reliability for the reduced cost of fault tolerance mechanisms, with additional measures to cope with "imperfect" fault tolerance, can potentially reduce the overall system cost while still satisfying the reliability goal.

Error detection. We consider error detection overheads as a given input to our design optimization and scheduling algorithms. Researchers have proposed a number of methods to optimize error detection, which, however, are considered in isolation. Thus, a joint design optimization of the error detection techniques with the fault tolerance techniques can be an interesting direction for future work.

Fault tolerance and power consumption. Researchers have proposed a number of techniques to optimize power consumption in the context of fault-tolerant embedded systems. One such design optimization framework, based on our shifting-based scheduling, that connects reliability and power consumption is, for example, discussed in [Pop07]. More work in

this area would be beneficial for the future development of reliable embedded systems that can deliver high performance at low power consumption.

Fault tolerance and embedded security. Security has become an important concern to modern embedded systems. Fault tolerance mechanisms against transient faults may, on one side, provide additional protection against attacks on the system. On the other side, these fault tolerance mechanisms may be exploited by an attacker to gain more information about the system or attempt to take control over it. Thus, studying fault tolerance in the context of embedded security and developing a design optimization framework for coupling security and fault tolerance techniques, based on the work presented in this thesis, can be an important contribution.

Appendix I

VEHICLE CRUISE CONTROLLER (CC). In this thesis we have used the vehicle cruise controller (CC) example from [Pop03]. In this appendix we briefly discuss the functionality of the CC and present its process graph.

The CC provides several functions: (a) maintaining a constant speed during driving at speeds between 35 and 200 km/h, (b) offering interface to the driver with the buttons to increase/decrease speed, (c) resuming a previous reference speed, and, finally, (d) it is suspended when the brake pedal has been pressed. The CC interacts with five distributed nodes: the Anti-lock Braking System (ABS), the Transmission Control Module (TCM), the Engine Control Module (ECM), the Electronic Throttle Module (ETM), and the Central Electronic Module (CEM). In this thesis we consider the architectures to implement the CC, where the CC functionality is mapped on three nodes, ABS, ETM and TCM.

In [Pop03] the CC functionality is modelled as a process graph, which consists of 32 processes, as depicted in Figure A.1. All processes in the application have to be executed, unless some functions are specified as optional (soft). For example, in Part III, we consider a CC implementation with 16 soft processes out

of 32. In Part II and Part IV, we consider that all processes in the CC process graph are mandatory (hard).

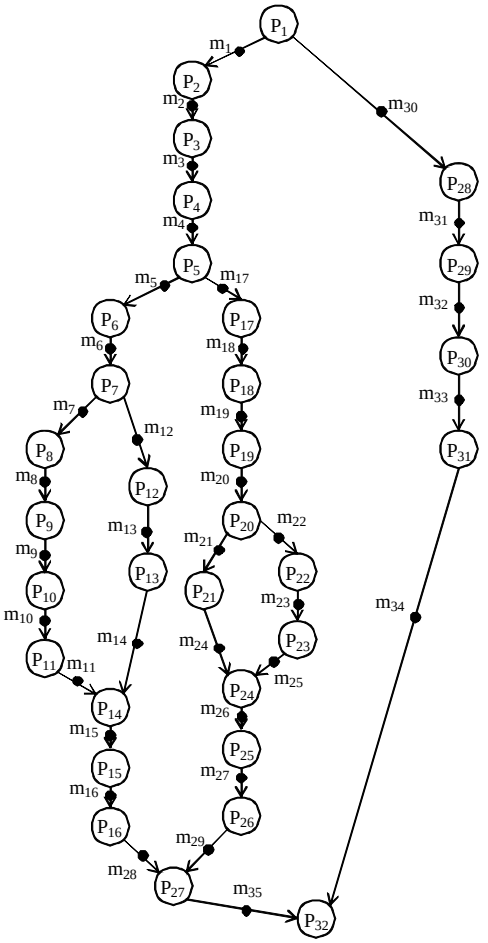


Figure A.1: The Cruise Controller Process Graph (Adapted from [Pop03])

Appendix II

FORMULA (6.3) IN CHAPTER 6. In the presence of k faults, for process P_i with the worst-case execution time C_i, error-detection overhead α_i, recovery overhead μ_i, and checkpointing overhead χ_i, the optimum number of checkpoints n_i^0, when process P_i is considered in isolation, is given by

$$n_i^0 = \begin{cases} n_i^- = \left\lfloor \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rfloor, \text{ if } C_i \le n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \\ n_i^+ = \left\lceil \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rceil, \text{ if } C_i > n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \end{cases}$$

Proof:[1] We consider process P_i in isolation in the presence of k faults. The execution time R_i of process P_i with n_i^0 checkpoints in the worst-case fault scenario is obtained with formula (6.1):

1. This proof, in general terms, follows the proof of Theorem 2 (formula (6.2) in Section 6.1) in [Pun97].

$$R_i(n_i^0) = E_i(n_i^0) + S_i(n_i^0) \longrightarrow$$

$$R_i(n_i^0) = (C_i + n_i^0 \times (\alpha_i + \chi_i)) + \left(\left(\frac{C_i}{n_i^0} + \mu_i\right) \times k + \alpha_i \times (k-1)\right)$$

The problem of finding the optimum number of checkpoints n_i^0 for process P_i, when we consider P_i in isolation, reduces to the following minimization problem:

Minimize

$$R_i(n_i^0) = (C_i + n_i^0 \times (\alpha_i + \chi_i)) + \left(\left(\frac{C_i}{n_i^0} + \mu_i\right) \times k + \alpha_i \times (k-1)\right)$$

with respect to n_i^0

The conditions for minima in a continuous system are

$$\frac{dR_i}{dn_i^0} = 0 \qquad \frac{d^2R_i}{d(n_i^0)^2} > 0$$

$$\frac{dR_i}{dn_i^0} = 0 \longrightarrow \alpha_i + \chi_i - \frac{C_i}{(n_i^0)^2} \times k = 0 \longrightarrow$$

$$\alpha_i + \chi_i = \frac{C_i}{(n_i^0)^2} \times k \longrightarrow n_i^0 = \sqrt{\frac{kC_i}{\alpha_i + \chi_i}}$$

Also,

$$\frac{d^2R_i}{d(n_i^0)^2} = \frac{2kC_i}{(n_i^0)^3} \longrightarrow \frac{d^2R_i}{d(n_i^0)^2} > 0 \text{ since } C_i > 0,\ k > 0, \text{ and } n_i^0 > 0$$

Since n_i^0 has to be integer, we have

$$n_i^- = \left\lfloor \sqrt{\frac{kC_i}{\alpha_i + \chi_i}} \right\rfloor \qquad n_i^+ = \left\lceil \sqrt{\frac{kC_i}{\alpha_i + \chi_i}} \right\rceil$$

We have to choose among these two values of n_i^0.

Let $n_i^- = \left\lfloor \sqrt{\frac{kC_i}{\alpha_i + \chi_i}} \right\rfloor$ $R_i(n_i^-)$ is better than $R_i(n_i^- + 1)$ if

$$(C_i + n_i^- \times (\alpha_i + \chi_i)) + \left(\left(\frac{C_i}{n_i^-} + \mu_i\right) \times k + \alpha_i \times (k-1)\right) <$$

$$(C_i + (n_i^- + 1) \times (\alpha_i + \chi_i)) + \left(\left(\frac{C_i}{n_i^- + 1} + \mu_i\right) \times k + \alpha_i \times (k-1)\right) \longrightarrow$$

$$\longrightarrow \frac{kC_i}{n_i^-} < (\alpha_i + \chi_i) + \frac{kC_i}{n_i^- + 1} \longrightarrow C_i < n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k}$$

which means that if $C_i < n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k}$, we select the floor value as locally optimal number of checkpoints. If $C_i > n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k}$, we select the ceiling.

When $C_i = n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k}$, the execution time R_i of process P_i in the worst-case fault scenario will be the same if we select n_i^- or if n_i^+. However, in this situation, we prefer to choose n_i^- because the lower number of checkpoints, caeteris paribus, reduces the number of possible execution scenarios and complexity.

Appendix III

PROBABILITY ANALYSIS FOR FIGURE 9.1.

(A). Let us compute the probability of no fault in process P_1 on the least hardened node N_1^1:[1]

$Pr(0;N_1^1) = \lfloor(1- 4{\cdot}10^{-2})\rfloor = 0.96.$

Formulae (9.4) and (9.5) are simplified in the case of a single node and can be used as follows:

$Pr(f > 0; N_1^1) = \lceil 1 - 0.96 \rceil = 0.04.$

The system period T is 360 ms, hence system reliability against transient faults is $(1 - 0.04)^{10000} = 0.00000000000$, or simply 0, i.e., the system is not reliable at all, and, thus, does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now try with a single re-execution, i.e., $k_1 = 1$:

$Pr(1;N_1^1) = \lfloor 0.96{\cdot}(4{\cdot}10^{-2})\rfloor = 0.0384.$

According to the simplified formulae (9.4) and (9.5),

$Pr(f > 1;N_1^1) = \lceil 1- 0.96 - 0.0384 \rceil = 0.0016.$

---

1. Here and later on, symbols $\lceil$ and $\rceil$ indicate that numbers are *rounded up* with 10^{-11} accuracy; $\lfloor$ and $\rfloor$ indicate that numbers are *rounded down* with 10^{-11} accuracy. This is needed in order to keep the safeness of the probabilities calculated.

Hence, the system reliability against transient faults is $(1 - 0.0016)^{10000} = 1.111 \cdot 10^{-7}$, which is quite far from the desired reliability goal $\rho = 1 - 10^{-5}$.

For $k_1 = 2$: $Pr(2;N_1^1) = \lfloor 0.96 \cdot 0.04 \cdot (4 \cdot 10^{-2}) \rfloor = 0.001536$, and

$Pr(f > 2;N_1^1) = \lceil 1 - 0.96 - 0.0384 - 0.001536 \rceil = 0.000064$.

The system reliability against transient faults is, thus, 0.52728162474, which is considerably better but still much less than $\rho = 1 - 10^{-5}$.

For $k_1 = 3$: $Pr(3;N_1^1) = \lfloor 0.96 \cdot 0.04 \cdot 0.04 \cdot (4 \cdot 10^{-2}) \rfloor = 0.00006144$, and $Pr(f > 3; N_1^1) = \lceil 1 - 0.96 - 0.0384 - 0.001536 - 0.00006144 \rceil = 0.00000256$. The system reliability against transient faults over the system period T is 0.97472486966, which is less than $\rho = 1 - 10^{-5}$.

For $k_1 = 4$: $Pr(4;N_1^1) = \lfloor 0.96 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot (4 \cdot 10^{-2}) \rfloor = 0.0000024576$, and $Pr(f > 4; N_1^1) = \lceil 1 - 0.96 - 0.0384 - 0.001536 - 0.00006144 - 0.0000024576 \rceil = 0.0000001024$. The system reliability over the system period T is already 0.99897652406, which is, however, less than $\rho = 1 - 10^{-5}$.

For $k_1 = 5$: $Pr(5;N_1^1) = \lfloor 0.96 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot (4 \cdot 10^{-2}) \rfloor = 0.00000009830$, and $Pr(f > 5; N_1^1) = \lceil 1 - 0.96 - 0.0384 - 0.001536 - 0.00006144 - 0.0000024576 - 0.00000009830 \rceil = 0.00000000410$. The obtained reliability over the system period T is 0.99995900084, which is now only slightly less than the required $\rho = 1 - 10^{-5}$.

Finally, for $k_1 = 6$: $Pr(6;N_1^1) = \lfloor 0.96 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot 0.04 \cdot (4 \cdot 10^{-2}) \rfloor = 0.00000000393$, and $Pr(f > 6; N_1^1) = \lceil 1 - 0.96 - 0.0384 - 0.001536 - 0.00006144 - 0.0000024576 - 0.00000009830 - 0.00000000393 \rceil = 0.00000000017$. The obtained reliability over the system period T is 0.99999830000, which meets the reliability goal $\rho = 1 - 10^{-5}$.

Thus, 6 re-executions must be introduced into process P_1 when running on the first h-version N_1^1 of computation node N_1, in order to satisfy the reliability goal ρ.

(B). Let us compute the probability of no fault in process P_1 on the second h-version N_1^2 of the node N_1:

$Pr(0;N_1^2) = \lfloor(1- 4{\cdot}10^{-4})\rfloor = 0.9996.$

Formulae (9.4) and (9.5) are simplified in the case of a single node and can be used as follows:

$Pr(f > 0; N_1^2) = \lceil 1 - 0.9996\rceil = 0.0004.$

The system period T is 360 ms, hence the system reliability against transient faults is $(1 - 0.0004)^{10000} = 0.01830098833$, which does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now try with a single re-execution, i.e., $k_1 = 1$:

$Pr(1;N_1^2) = \lfloor 0.9996{\cdot}(4{\cdot}10^{-4})\rfloor = 0.00039984.$

According to the simplified formulae (9.4) and (9.5),

$Pr(f > 1;N_1^2) = \lceil 1- 0.9996 - 0.00039984\rceil = 0.00000016.$

Hence, the system reliability against transient faults is $(1 - 0.00000016)^{10000} = 0.99840127918$, which still does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

For $k_1 = 2$: $Pr(2; N_1^2) = \lfloor 0.9996 \cdot 0.0004 \cdot (4 \cdot 10^{-4})\rfloor = 0.00000015993$, and

$Pr(f > 2; N_1^2) = \lceil 1- 0.9996 - 0.00039984 - 0.00000015993\rceil = 0.00000000007.$

The system reliability over the system period T is, thus, 0.99999930000, which meets the reliability goal $\rho = 1 - 10^{-5}$.

Thus, 2 re-executions must be introduced into process P_1 when running on the second h-version N_1^2 of computation node N_1, in order to satisfy the reliability goal ρ.

(C). Let us compute the probability of no fault in process P_1 on the most hardened version N_1^3 of node N_1:

$Pr(0;N_1^3) = \lfloor(1- 4{\cdot}10^{-6})\rfloor = 0.999996.$

Formulae (9.4) and (9.5) are simplified in the case of a single node and can be used as follows:

$Pr(f > 0; N_1^3) = \lceil 1 - 0.999996\rceil = 0.000004.$

The system period T is 360 ms, hence system reliability against transient faults is $(1 - 0.000004)^{10000} = 0.96078936228$, which does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now try with $k_1 = 1$:

$Pr(1;N_1^3) = \backslash 0.999996 \cdot (4 \cdot 10^{-6}) /= 0.00000399998$.

According to the simplified formulae (9.4) and (9.5),

$Pr(f > 1;N_1^3) = (1 - 0.999996 - 0.00000399998 \backslash = 2 \cdot 10^{-11}$.

Hence, the system reliability against transient faults is $(1 - 2 \cdot 10^{-11})^{10000} = 0.99999980000$, which meets the reliability goal $\rho = 1 - 10^{-5}$.

Thus, 1 re-execution must be introduced into process P_1 when running on the third h-version N_1^3 of computation node N_1, in order to satisfy the reliability goal ρ.

PROBABILITY ANALYSIS FOR FIGURE 10.1.

(A). Let us, at first, compute the probability of no faulty processes for both hardened nodes N_1^2 and N_2^2:

$Pr(0;N_1^2) = \backslash (1 - 1.2 \cdot 10^{-5}) \cdot (1 - 1.3 \cdot 10^{-5}) / = 0.99997500015$.

$Pr(0;N_2^2) = \backslash (1 - 1.2 \cdot 10^{-5}) \cdot (1 - 1.3 \cdot 10^{-5}) / = 0.99997500015$.

According to formulae (9.4) and (9.5),

$Pr(f > 0;\ N_1^2) = 1 - 0.99997500015 = 0.00002499985$.

$Pr(f > 0;\ N_2^2) = 1 - 0.99997500015 = 0.00002499985$.

$Pr((f > 0;\ N_1^2) \cup (f > 0;\ N_2^2)) = (1 - (1 - 0.00002499985) \cdot (1 - 0.00002499985) \backslash = 0.00004999908$.

The system period T is 360 ms, hence system reliability is $(1 - 0.00004999908)^{10000} = 0.60652865819$, which means that the system does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now consider $k_1 = 1$ and $k_2 = 1$:

$Pr(1;N_1^2) = \backslash 0.99997500015 \cdot (1.2 \cdot 10^{-5} + 1.3 \cdot 10^{-5}) /= 0.00002499937$.

$Pr(1;N_2^2) = \backslash 0.99997500015 \cdot (1.2 \cdot 10^{-5} + 1.3 \cdot 10^{-5}) /= 0.00002499937$.

According to formulae (9.4) and (9.5),

$Pr(f > 1;N_1^2) = (1 - 0.99997500015 - 0.00002499937 \backslash = 4.8 \cdot 10^{-10}$.

$Pr(f > 1;N_2^2) = (1 - 0.99997500015 - 0.00002499937 \backslash = 4.8 \cdot 10^{-10}$.

$Pr((f > 1; N_1^2) \cup (f > 1; N_2^2)) = 9.6 \cdot 10^{-10}$.

Hence, the system reliability is $(1 - 9.6 \cdot 10^{-10})^{10000} = 0.99999040004$ and the system meets its reliability goal $\rho = 1 - 10^{-5}$.

Thus, 1 re-execution must be introduced into all processes run on the second h-versions N_1^2 and N_2^2 of computation nodes N_1 and N_2, in order to satisfy the reliability goal ρ.

(B). Let us compute the probability of no faulty processes on the hardened node N_1^2:

$Pr(0;N_1^2) = (1-1.2 \cdot 10^{-5}) \cdot (1-1.3 \cdot 10^{-5}) \cdot (1-1.4 \cdot 10^{-5}) \cdot (1-1.6 \cdot 10^{-5}) = 0.99994500112$.

According to formulae (9.4) and (9.5), simplified for a single node,

$Pr(f > 0; N_1^2) = 1 - 0.99994500112 = 0.00005499888$.

The system period T is 360 ms, hence, system reliability is $(1 - 0.00005499888)^{10000} = 0.57694754589$, which means that the system does not satisfy the reliability goal $\rho = 1 - 10^{-5}$.

Let us now consider $k_1 = 1$:

$Pr(1;N_1^2) = 0.99994500112 \cdot (1.2 \cdot 10^{-5} + 1.3 \cdot 10^{-5} + 1.4 \cdot 10^{-5} + 1.6 \cdot 10^{-5}) = 0.00005499697$, and $Pr(f > 1; N_1^2) = 1 - 0.99994500112 - 0.00005499697 = 0.00000000191$.

Hence, the system reliability is $(1 - 1.89 \cdot 10^{-9})^{10000} = 0.99998090018$ and the system does not meet its reliability goal $\rho = 1 - 10^{-5}$, with a very small deviation though.

Let us now consider $k_1 = 2$:

$Pr(2; N_1^2) = 0.99994500112 \cdot (1.2 \cdot 10^{-5} \cdot 1.2 \cdot 10^{-5} + 1.2 \cdot 10^{-5} \cdot 1.3 \cdot 10^{-5} + 1.2 \cdot 10^{-5} \cdot 1.4 \cdot 10^{-5} + 1.2 \cdot 10^{-5} \cdot 1.6 \cdot 10^{-5} + 1.3 \cdot 10^{-5} \cdot 1.3 \cdot 10^{-5} + 1.3 \cdot 10^{-5} \cdot 1.4 \cdot 10^{-5} + 1.3 \cdot 10^{-5} \cdot 1.6 \cdot 10^{-5} + 1.4 \cdot 10^{-5} \cdot 1.4 \cdot 10^{-5} + 1.4 \cdot 10^{-5} \cdot 1.6 \cdot 10^{-5} + 1.6 \cdot 10^{-5} \cdot 1.6 \cdot 10^{-5}) = 0.00000000189$, and $Pr(f > 2; N_1^2) = 1 - 0.99994500112 - 0.00005499697 - 0,00000000189 = 0.00000000002$. Thus, the probability of system failure, which would require more than 2 re-executions, is $(1 - 2 \cdot 10^{-11})^{10000} = 0.99999980000$. Hence, the system meets its reliability goal $\rho = 1 - 10^{-5}$ with $k_1 = 2$ re-executions.

(C). Let us compute the probability of no faulty processes on the hardened node N_2^2:

$Pr(0;N_2^2) = \backslash(1–1\cdot10^{-5})\cdot(1–1.2\cdot10^{-5})\cdot(1–1.2\cdot10^{-5})\cdot(1–1.3\cdot10^{-5})/ = 0.99995300082$.

According to formulae (9.4) and (9.5), simplified for a single node,

$Pr(f > 0; N_2^2) = 1 – 0.99995300082 = 0.00004699918$.

The system period T is 360 ms, hence, system reliability is $(1 – 0.00004699918)^{10000} = 0.62500049017$, which means that the system does not satisfy the reliability goal $\rho = 1 – 10^{-5}$.

Let us now consider $k_2 = 1$:

$Pr(1;N_2^2) = \backslash0.99995300082 \cdot (1 \cdot 10^{-5} + 1.2 \cdot 10^{-5} + 1.2\cdot10^{-5} + 1.3 \cdot 10^{-5})/ = 0.00004699779$, and $Pr(f > 1; N_2^2) = (1 – 0.99995300082 – 0.00004699779\backslash = 0.00000000139$.

Hence, the system reliability is $(1 – 1.39\cdot10^{-9})^{10000} = 0.99998610009$ and the system does not meet its reliability goal $\rho = 1 – 10^{-5}$, with a very small deviation.

Let us now consider $k_2 = 2$:

$Pr(2; N_2^2) = \backslash0.99995300082 \cdot (1\cdot10^{-5}\cdot1\cdot10^{-5} + 1\cdot10^{-5}\cdot1.2\cdot10^{-5} + 1\cdot10^{-5}\cdot1.2\cdot10^{-5}+1\cdot10^{-5}\cdot1.3\cdot10^{-5}+1.2\cdot10^{-5}\cdot1.2\cdot10^{-5}+1.2\cdot10^{-5}\cdot1.2\cdot10^{-5} + 1.2\cdot10^{-5}\cdot1.3\cdot10^{-5} + 1.2\cdot10^{-5}\cdot1.2\cdot10^{-5} + 1.2\cdot10^{-5}\cdot1.3\cdot10^{-5} + 1.3\cdot10^{-5} \cdot 1.3\cdot10^{-5})/ = 0.00000000138$, and $Pr(f > 2; N_2^2) = (1 – 0.99995300082 – 0.00004699779 – 0.00000000138\backslash = 0.00000000001$. Thus, the probability of system failure, which requires more than 2 re-executions, is $(1 – 10^{-11})^{10000} = 0.99999990000$. Hence, the system meets its reliability goal $\rho = 1 – 10^{-5}$ with 2 re-executions.

(D). Let us compute the probability of no faulty processes on the hardened node N_1^3:

$Pr(0;N_1^3) = \backslash(1 – 1.2 \cdot 10^{-10}) \cdot (1 – 1.3 \cdot 10^{-10}) \cdot (1 – 1.4 \cdot 10^{-10}) \cdot (1 – 1.6 \cdot 10^{-10})/ = 0.99999999945$.

According to formulae (9.4) and (9.5), simplified for a single node,

$Pr(f > 0; N_1^3) = 1 - 0.99999999945 = 0.00000000055$.

The system period T is 360 ms, hence, system reliability is $(1 - 5.5 \cdot 10^{-10})^{10000} = 0.99999450001$, which means that the system meets the reliability goal $\rho = 1 - 10^{-5}$, even without any re-executions in software.

(E). Let us compute the probability of no faulty processes on the hardened node N_2^3:

$Pr(0;N_2^3) = \langle(1 - 1 \cdot 10^{-10}) \cdot (1 - 1.2 \cdot 10^{-10}) \cdot (1 - 1.2 \cdot 10^{-10}) \cdot (1 - 1.3 \cdot 10^{-10})\rangle = 0.99999999953$.

According to formulae (9.4) and (9.5), simplified for a single node,

$Pr(f > 0; N_2^3) = 1 - 0.99999999953 = 0.00000000047$.

The system period T is 360 ms, hence, system reliability is $(1 - 4.7 \cdot 10^{-10})^{10000} = 0.99999530001$, which means that the system meets the reliability goal $\rho = 1 - 10^{-5}$, even without any re-executions in software.

List of Notations

Application and Basic Architecture

$\mathcal{A}$	Application
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	Merged directed acyclic hypergraph of the application $\mathcal{A}$
$\mathcal{V}$	Set of processes (vertices)
$\mathcal{E}$	Set of messages (edges)
$P_i \in \mathcal{V}$	Process
$e_{ij} \in \mathcal{E}$	An edge that indicates that output of process P_i is an input of process P_j
m_i	Message
$d_i \in \mathcal{D}$	Deadline of process P_i
$\mathcal{D}$	Set of process deadlines
D	Global cumulative deadline
T	Hyperperiod of the application $\mathcal{A}$
$N_j \in \mathcal{N}$	Computation node
$\mathcal{N}$	Set of computation nodes
B	Communication bus

$r_j \in \mathcal{N} \cup \{B\}$ Resource (either a computation node $N_j \in \mathcal{N}$ or the bus B)

t_{ij}^{w} Worst-case execution time of process P_i executed on computation node N_j

C_i Worst-case execution time of process P_i (mapping is not specified)

t_{ij}^{b} Best-case execution time of process P_i executed on computation node N_j

t_{ij}^{e} Expected (average) execution time of process P_i executed on computation node N_j

$U_i(t)$ Utility function of soft process P_i

$\mathcal{R}$ Service degradation rule

σ_i Service performance degradation coefficient for process P_i (to capture the input "stale" values)

$U_i^{*}(t)$ Modified utility of soft process P_i capturing service performance degradation

U Overall utility

TF_i Tail factor of process P_i, $TF_i = WCET_i / (AET_i \times 2)$

Succ(P_i) Set of successors of process P_i

Pred(P_i) Set of predecessors of process P_i

Fault Tolerance Techniques

k The maximum number of faults that can happen in the worst case during one application run (or hyperperiod T)

f A current number of faults during one application run (execution scenario)

$P_{i/j}$ jth re-execution of process P_i

$P_{i(j)}$	jth replica of process P_i, where $P_{i(1)}$ is the original process
$m_{i(j)}$	jth replica of message m_i
P_i^k	kth execution (checkpointing) segment of process P_i
$P_{i/j}^k$	jth recovery of kth execution (checkpointing) segment of process P_i
α	Error detection overhead
μ	Recovery overhead
γ	Checkpointing overhead
O_i	Constant checkpointing overhead for process P_i, $O_i = \alpha_i + \gamma_i$
E_i	Actual execution time of process P_i
S_i	Recovery time of process P_i in the worst case
R_i	The total execution time of process P_i, $R_i = S_i + E_i$
n_i	The number of checkpoints in process P_i
n_i^0	The optimal number of checkpoints in process P_i if process P_i is considered in isolation

Fault-Tolerant Process Graph

$G(V_P \cup V_C \cup V_T, E_S \cup E_C)$	Fault-tolerant process graph corresponding to application $\mathcal{A} = \mathcal{G}(\mathcal{V}, \mathcal{E})$
V_P	Set of regular processes and messages
V_C	Set of conditional processes
V_T	Set of synchronization nodes
$v_i \in V_T$	Synchronization node
P_i^m	mth copy of process $P_i \in \mathcal{V}$

P_i^S	Synchronization node corresponding to process $P_i \in \mathcal{A}$
m_i^S	Synchronization node corresponding to message $m_i \in \mathcal{A}$
E_S	Set of simple edges
E_C	Set of conditional edges
$e_{ij}^{mn} \in E_S$	Simple edge that indicates that the output of P_i^m is the input of P_j^n
$e_{ij}^{mS} \in E_S$	Simple edge that indicates that the output of P_i^m is the input of P_j^S
$e_{ij}^{Sn} \in E_S$	Simple edge that indicates that the output of P_i^S is the input of P_j^n
$e_{ij}^{mS_m} \in E_S$	Simple edge that indicates that the output of P_i^m is the input of m_j^S
$e_{ij}^{S_m n} \in E_S$	Simple edge that indicates that the output of m_i^S is the input of P_j^n
$e_{ij}^{SS} \in E_S$	Simple edge that indicates that the output of P_i^S is the input of P_j^S
$e_{ij}^{S_m S} \in E_S$	Simple edge that indicates that the output of m_i^S is the input of P_j^S
$e_{ij}^{SS_m} \in E_S$	Simple edge that indicates that the output of P_i^S is the input of m_j^S
$e_{ij}^{mn} \in E_C$	Conditional edge that indicates that the output of P_i^m is the input of P_j^n
$e_{ij}^{mS} \in E_C$	Conditional edge that indicates that the output of P_i^m is the input of P_j^S
$e_{ij}^{mS_m} \in E_C$	Conditional edge that indicates that the output of P_i^m is the input of m_j^S

$F_{P_i^m}$	The "true" condition value ("fault" condition) if P_i^m experiences a fault
$\overline{F}_{P_i^m}$	The "false" condition value ("no fault" condition) if P_i^m does not experience a fault
$K_{P_i^m}$	Guard of P_i^m (a boolean expression that captures the necessary activation conditions (fault scenario) for the respective node)
K	Current guard
$\mathcal{K}$	Set of guards
$\mathcal{K}_{\vartheta_i}$	Set of guards of synchronization node ϑ_i
$vc_n \in \mathcal{VC}$	Valid combination of copies of the predecessor processes
$\mathcal{VC}$	Set of valid combination of copies of the predecessor processes

Scheduling

S	Schedule
S	Set of schedules (schedule tables)
Φ	A tree of fault-tolerant schedules
$S_i \in \Phi$	Schedule in the schedule tree Φ
$S_i^j \in \Phi$	Schedule in the schedule tree Φ that corresponds to jth group of schedules
S_{root}	First "root" schedule in the schedule tree, to which other schedules are attached
M	Size of the schedule tree
ϕ_{new}	New schedule in the schedule tree to switch to (and the new schedule produced by the scheduling algorithm)
S_{parent}	Parent schedule

t_c	Completion (finishing) time of a process
Δ	Evaluation step
$d\varepsilon$	Simulation error
$\mathcal{L}_R$	Ready list of processes [and messages]
$\mathcal{L}_\vartheta$	List of synchronization nodes
CRT	Process time counter
$\mathcal{RS}$	Root schedule (for shifting-base scheduling)
$\mathcal{T}$: $\mathcal{W} \rightarrow \{Frozen, Regular\}$	Transparency function
$\mathcal{W}$	Set of processes and messages
$w_i \in \mathcal{W}$	A process or a message
$\mathcal{T}(w_i)$	Transparency requirements for w_i
$\mathcal{T}(\mathcal{A})$	Transparency requirements for application $\mathcal{A}$
Sg_i	Signalling message of process P_i
CT_i	"True" condition upon completion of process P_i
CF_i	"False" condition upon completion of process P_i
rex	Number of re-executions in the schedule associated to a process
δ_G	The worst-case execution time (end-to-end delay) of scheduled application $\mathcal{A}$
δ_{CS}	The end-to-end delay of application $\mathcal{A}$ scheduled with conditional scheduling
δ_{SBS}	The end-to-end delay of application $\mathcal{A}$ scheduled with shifting-based scheduling
δ_{NFT}	The end-to-end delay of application $\mathcal{A}$ in case of no faults
U_{FTTree}, U_n	The overall utility produced by application $\mathcal{A}$ for a tree of fault-tolerant schedules

U_{f^N} The overall utility produced by application $\mathcal{A}$ for a single f^N schedule

Mapping and Fault Tolerance Policy Assignment

$\mathcal{M}: \mathcal{V} \rightarrow \mathcal{N}$ Process mapping function

$\mathcal{P}: \mathcal{V} \rightarrow \{Replication, Re\text{-}execution, Replication\ \&\ Re\text{-}execution\}$ Function which specifies whether a process is replicated, re-executed, or replicated and re-executed

$\mathcal{P}: \mathcal{V} \rightarrow \{Replication, Checkpointing, Replication\ \&\ Checkpointing\}$ Function which specifies whether a process is replicated, checkpointed, or replicated and checkpointed

$\mathcal{Q}: \mathcal{V} \rightarrow \mathrm{N}$ Function for assigning a number of replicas

$\mathcal{V}_R$ Set of process replicas

$\mathcal{R}: \mathcal{V} \cup \mathcal{V}_R \rightarrow \mathrm{N}$ Function for assigning a number of re-executions (recoveries) to processes and replicas

$\mathcal{X}: \mathcal{V} \cup \mathcal{V}_R \rightarrow \mathrm{N}$ Function for assigning a number of checkpoints to processes and replicas

$\mathcal{M}(P_i)$ Mapping of process P_i

$\mathcal{P}(P_i)$ Fault tolerance technique or a combination of fault tolerance techniques assigned to process P_i

$\mathcal{Q}(P_i)$ Number of replicas of process P_i

$\mathcal{R}(P_i)$, $\mathcal{R}(P_{i(j)})$ Number of re-executions (recoveries) for process P_i or replica $P_{i(j)}$

$\mathcal{X}(P_i)$, $\mathcal{X}(P_{i(j)})$ Number of checkpoints for process P_i or replica $P_{i(j)}$

ψ System configuration, $\psi = <\mathcal{F}, \mathcal{M}, S>$ (for re-execution) or $\psi = <\mathcal{F}, \mathcal{X}, \mathcal{M}, S>$ (for checkpointing)

$\mathcal{F}$	Fault tolerance policy assignment, $\mathcal{F} = <\mathcal{P}, \mathcal{Q}, \mathcal{R}>$ (for re-execution) or $\mathcal{F} = <\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{X}>$ (for checkpointing)
x^{best}	Best solution
N^{now}	Set of possible moves
$N^{waiting}$	Waiting moves
$N^{non\text{-}tabu}$	Non-tabu moves
x^{now}	Current solution
$x^{waiting}$	Waiting solution
$x^{non\text{-}tabu}$	Non-tabu solution
$Tabu$	Set of tabu counters
$Wait$	Set of waiting counters
$C\mathcal{P}$	Set of processes on the critical path

Hardened Architecture and Hardening Optimization

h	Hardening level of a computation node
N_j^h	Hardening version h of computation node N_j
C_j^h	Cost of version h of computation node N_j
t_{ijh}	Worst-case execution time of process P_i on h-version of computation node N_j
p_{ijh}	Process failure probability of process P_i on h-version of computation node N_j
ρ	Reliability goal, $\rho = 1 - \gamma$
γ	The maximum probability of a system failure due to transient faults on any computation node within a time unit, e.g. one hour of functionality
k_j	The maximum number of transient faults tolerated in software on computation node N_j during one iteration of the application (hyperperiod T)

$\{P_i, N_j^h\}$ Mapping of process P_i on h-version of computation node N_j

$\mathcal{AR}$ System architecture

$HWCost_j$ Hardening cost of node N_j

SL Schedule length

H Hardening level of the architecture

H_{min} Minimum hardening level of the architecture

System Failure Probability (SFP) Analysis

$Pr(0; N_j^h)$ Probability of no faults occurring (no faulty processes) during one iteration of the application (hyperperiod T) on the h-version of node N_j

$Pr_{S^*}(f; N_j^h)$ Probability of successful recovering from f faults in a particular fault scenario S^*

$Pr(f; N_j^h)$ Probability that the system recovers from all possible f faults during one iteration of the application

$Pr(f > k_j; N_j^h)$ The failure probability of the h-version of node N_j with k_j re-executions during one iteration of the application

$Pr\left(\bigcup_{j=1}^{n} (f > k_j; N_j^h)\right)$ Probability that the system composed of n computation nodes with k_j re-executions on each node N_j will not recover, in the case more than k_j faults have happened on any computation node N_j during one iteration of the application

List of Abbreviations

ABS	Anti-lock Braking System
AET	Expected (Average) Execution Time
ALU	Arithmetic Logic Unit
ArC	Architecture Cost
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
AUED	All Unidirectional Error Detecting
BCET	Best Case Execution Time
CC	Cruise Controller
CEM	Central Electronic Module
CP	Critical Path
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRT	Process Time Counter
CS	Conditional Scheduling
ECM	Engine Control Module

EDAC	Error Detection And Correction
EDF	Earliest Deadline First
EMI	Electromagnetic Interference
ET	Event Triggered
ETM	Electronic Throttle Module
FPGA	Filed Programmable Gate Array
FTPG	Fault-Tolerant Process Graph
HPD	Hardening Performance Degradation
IC	Integrated Circuit
MARS	Maintainable Real Time System
MC	Mapping with Checkpointing
MC0	Mapping with Locally Optimal Checkpoiting
MCR	Mapping with Checkpointing and Replication
MR	Mapping with Replication
MU	Multiple Utility
MX	Mapping with Re-execution
MXR	Mapping with Re-execution and Replication
NFT	Non-Fault-Tolerant
PCP	Partial Critical Path
QoS	Quality of Service
RM	Rate Monotonic
RS	Root Schedule
RT	Real-Time
SBS	Shifting-based Scheduling
SEC-DED	Single Error Correcting, Double Error Detecting
SEU	Single Event Upset

SFP	System Failure Probability
SFS	Strongly Fault Secure
SFX	Straightforward Re-execution
SP	Soft Priority
TCM	Transmission Control Module
TT	Time Triggered
WCET	Worst Case Execution Time
WCTT	Worst Case Transmission Time
XBW	X-by-Wire

Bibliography

[Aid01] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool", *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 83-88, 2001.

[Aid05] J. Aidemark, P. Folkesson, and J. Karlsson, "A Framework for Node-Level Fault Tolerance in Distributed Real-Time Systems", *Proc. Intl. Conf. on Dependable Systems and Networks*, 656-665, 2005.

[Als01] K. Alstrom and J. Torin, "Future Architecture for Flight Control Systems", *Proc. 20^{th} Conf. on Digital Avionics Systems*, 1B5/1-1B5/10, 2001.

[Alo01] R. Al-Omari, A. K. Somani, and G. Manimaran, "A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-Time Systems", *Proc. 15^{th} Intl. Parallel and Distributed Processing Symp.*, 23-27, 2001.

[Ahn97] KapDae Ahn, Jong Kim, and SungJe Hong, "Fault-Tolerant Real-Time Scheduling Using Passive Replicas", *Proc. Pacific Rim Intl. Symp. on Fault-Tolerant Systems*, 98-103, 1997.

[Aya08] T. Ayav, P. Fradet, and A. Girault, "Implementing Fault-Tolerance in Real-Time Programs by Automatic Program Transformations", *ACM Trans. on Embedded Computing Systems*, 7(4), 1-43, 2008.

[Aud95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, 8, 173-198, 1995.

[Axe96] J. Axelsson, "Hardware/Software Partitioning Aiming at Fulfillment of Real-Time Constraints", *Systems Architecture*, 42, 449-464, 1996.

[Ayd00] H. Aydin, R. Melhem, and D. Mosse, "Tolerating Faults while Maximizing Reward", *Proc. 12^{th} Euromicro Conf. on Real-Time Systems*, 219–226, 2000.

[Bal06] V. B. Balakirsky and A. J. H. Vinck, "Coding Schemes for Data Transmission over Bus Systems", *In Proc. IEEE Intl. Symp. on Information Theory*, 1778-1782, 2006.

[Bar08] R. Barbosa and J. Karlsson, "On the Integrity of Lightweight Checkpoints", *Proc. 11^{th} IEEE High Assurance Systems Engineering Symp.*, 125-134, 2008.

[Bau01] R. C. Baumann and E. B. Smith, "Neutron-Induced ^{10}B Fission as a Major Source of Soft Errors in High Density SRAMs", *Microelectronics Reliability*, 41(2), 211-218, 2001.

[Ben03] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A Watchdog Processor to Detect Data and Control Flow Errors", *Proc. 9^{th} IEEE On-Line Testing Symp.*, 144-148, 2003.

[Ber94] A. Bertossi and L. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real Time Systems", *Real Time Systems*, 7(3), 229–256, 1994.

[Bol97] I. Bolsens, H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/Software Co-Design of Digital Telecommunication Systems", *Proc. of the IEEE*, 85(3), 391-418, 1997.

[Bou04] P. Bourret, A. Fernandez, and C. Seguin, "Statistical Criteria to Rationalize the Choice of Run-Time Observation Points in Embedded Software", *In Proc. 1st Intl. Workshop on Testability Assessment*, 41-49, 2004.

[Bur96] A. Burns, R. Davis, and S. Punnekkat, "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Proc. Euromicro Workshop on Real-Time Systems*, 29–33, 1996.

[But99] G. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments", *IEEE Trans. on Computers*, 48(10), 1035–1052, 1999.

[Che99] P. Chevochot and I. Puaut, "Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies", *Proc. 6th Intl. Conf. on Real-Time Computing Systems and Applications*, 356-363, 1999.

[Cho95] P. H. Chou, R. B. Ortega, and G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", *Proc. Intl. Symp. on System Synthesis*, 22-27, 1995.

[Cla98] V. Claesson, S. Poledna, and J. Soderberg, "The XBW Model for Dependable Real-Time Systems", *Proc. Intl. Conf. on Parallel and Distributed Systems*, 130-138, 1998.

[Cof72] E. G. Coffman Jr. and R. L. Graham, “Optimal Scheduling for Two Processor Systems”, *Acta Informatica*, 1, 200-213, 1972.

[Col03] A. Colin and S. M. Petters, “Experimental Evaluation of Code Properties for WCET Analysis”, *Proc. Real-Time Systems Symp. (RTSS)*, 190-199, 2003.

[Con05] J. Conner, Y. Xie, M. Kandemir, R. Dick, and G. Link, “FD-HGAC: A Hybrid Heuristic/Genetic Algorithm Hardware/Software Co-synthesis Framework with Fault Detection”, *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 709-712, 2005.

[Con03] C. Constantinescu, “Trends and Challenges in VLSI Circuit Reliability”, *IEEE Micro*, 23(4), 14-19, 2003.

[Cor04a] F. Corno, M. Sonza Reorda, S. Tosato, and F. Esposito, “Evaluating the Effects of Transient Faults on Vehicle Dynamic Performance in Automotive Systems”, *Proc. Intl. Test Conf. (ITC)*, 1332-1339, 2004.

[Cor04b] L. A. Cortes, P. Eles, and Z. Peng, “Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks”, *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 1176-1181, 2004.

[Dav99] B. P. Dave, G. Lakshminarayana, and N. J. Jha, “COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems”, *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems*, 7(1), 92-104, 1999.

[Dav93] R. I. Davis, K. W. Tindell, and A. Burns, “Scheduling Slack Time in Fixed Priority Pre-emptive Systems”, *Proc. Real-Time Systems Symp. (RTSS)*, 222–231, 1993.

[Deo98] J. S. Deogun, R. M. Kieckhafer, and A. W. Krings, "Stability and Performance of List Scheduling with External Process Delays", *Real Time Systems*, 15(1), 5-38, 1998.

[Dim01] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, "Off-line Real-Time Fault-Tolerant Scheduling", *Proc. Euromicro Parallel and Distributed Processing Workshop*, 410–417, 2001.

[Ele97] P. Eles, Z. Peng, K. Kuchcinski and A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Design Automation for Embedded Systems*, 2(1), 5-32, 1997.

[Ele00] P. Eles, Z. Peng, P. Pop, and A. Doboli, "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Trans. on VLSI Systems*, 8(5), 472-491, 2000.

[Ema07] K. C. Emani, K. Kam, and M. Zawodniok, "Improvement of CAN BUS Performance by Using Error-Correction Codes", *In Proc. IEEE Region 5 Technical Conf.*, 205-210, 2007.

[Erm05] A. Ermedahl, F. Stappert, and J. Engblom, "Clustered Worst-Case Execution-Time Calculation", *IEEE Trans. on Computers*, 54(9), 1104-1122, 2005.

[Ern93] R. Ernst, J. Henkel, and T. Benner, "Hardware/Software Co-Synthesis for Microcontrollers", *IEEE Design & Test of Computers*, 10(3), 64-75, 1993.

[Fle04] FlexRay, Protocol Specification, Ver. 2.0, *FlexRay Consortium*, 2004.

[Fux95] W. Fuxing, K. Ramamritham, and J.A. Stankovic, "Determining Redundancy Levels for Fault Tolerant Real-Time Systems", *IEEE Trans. on Computers*, 44(2), 292–301, 1995.

[Gar03] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 2003.

[Gar06] R. Garg, N. Jayakumar, S. P. Khatri, and G. Choi, "A Design Approach for Radiation-Hard Digital Electronics", *Proc. Design Automation Conf. (DAC)*, 773-778, 2006.

[Gir03] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel, "An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules", *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 159-168, 2003.

[Gol03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error Detection Using Control Flow Assertions", *Proc. 18th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 581-588, 2003.

[Gom06] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection", *IEEE Micro*, 26(1), 92-99, 2006.

[Gus05] J. Gustafsson, A. Ermedahl, and B. Lisper, "Towards a Flow Analysis for Embedded System C Programs", *Proc. 10th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems*, 287-297, 2005.

[Han03] C. C. Han, K. G. Shin, and J. Wu, "A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults", *IEEE Trans. on Computers*, 52(3), 362–372, 2003.

[Han02] H. A. Hansson, T. Nolte, C. Norström, and S. Punnekkat, "Integrating Reliability and Timing Analysis of CAN-based Systems", *IEEE Trans. on Industrial Electronics*, 49(6), 1240-1250, 2002.

[Har01] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and Changhong Dai, "Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes", *Proc. Symp. on VLSI Technology*, 73-74, 2001.

[Hay07] J. P. Hayes, I. Polian, B. Becker, "An Analysis Framework for Transient-Error Tolerance", *Proc. IEEE VLSI Test Symp.*, 249-255, 2007.

[Hea02] C. A. Healy and D. B. Whalley, "Automatic Detection and Exploitation of Branch Constraints for Timing Analysis", *IEEE Trans. on Software Engineering*, 28(8), 763-781, 2002.

[Hei05] P. Heine, J. Turunen, M. Lehtonen, and A. Oikarinen, "Measured Faults during Lightning Storms", *Proc. IEEE PowerTech'2005*, Paper 72, 5p., 2005.

[Her00] A. Hergenhan and W. Rosenstiel, "Static Timing Analysis of Embedded Software on Advanced Processor Architectures", *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 552-559, 2000.

[Hil00] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems", *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 24-33, 2000.

[Izo05] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 864-869, 2005.

[Izo06a] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Mapping of Fault-Tolerant Applications with Transparency on Distributed Embedded Systems", *Proc. 9^{th} Euromicro Conf. on Digital System Design (DSD)*, 313-320, 2006.

[Izo06b] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems", *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 706-711, 2006.

[Izo06c] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication", *Proc. 3rd IEEE Intl. Workshop on Electronic Design, Test & Applications (DELTA)*, 440-447, 2006.

[Izo08a] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints", *Proc. Design, Automation, and Test in Europe Conf. (DATE)*, 915-920, 2008.

[Izo08b] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of Flexible Fault-Tolerant Schedules with Pre-emption for Mixed Soft and Hard Real-Time Systems", *Proc. 11th Euromicro Conf. on Digital System Design (DSD),* 71-80, 2008.

[Izo09] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, "Analysis and Optimization of Fault-Tolerant Embedded Systems with Hardened Processors", *Proc. Design, Automation, and Test in Europe Conf. (DATE)*, 682-687, 2009.

[Izo10a] V. Izosimov, P. Eles, and Z. Peng, "Value-based Scheduling of Distributed Fault-Tolerant Real-Time Systems with Soft and Hard Timing Constraints", *Submitted for publication*, 2010.

[Izo10b] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling and Optimization of Fault-Tolerant Embedded Systems with Transparency/Performance Trade-Offs", *Submitted for publication*, 2010.

[Jia00] Jia Xu and D. L. Parnas, "Priority Scheduling Versus Pre-Run-Time Scheduling", *Real Time Systems*, 18(1), 7-24, 2000.

[Jia05] Jian-Jun Han and Qing-Hua Li, "Dynamic Power-Aware Scheduling Algorithms for Real-Time Task Sets with Fault-Tolerance in Parallel and Distributed Computing Environment", *Proc. of Intl. Parallel and Distributed Processing Symp.*, 6-16, 2005.

[Jie92] Jien-Chung Lo, S. Thanawastien, T. R. N. Rao, and M. Nicolaidis, "An SFS Berger Check Prediction ALU and Its Application to Self-Checking Processor Designs", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (CAD)*, 11(4), 525-540, 1992.

[Jie96] Jie Xu and B. Randell, "Roll-Forward Error Recovery in Embedded Real-Time Systems", *Proc. Intl. Conf. on Parallel and Distributed Systems*, 414-421, 1996.

[Jon05] Jong-In Lee, Su-Hyun Park, Ho-Jung Bang, Tai-Hyo Kim, and Sung-Deok Cha, "A Hybrid Framework of Worst-Case Execution Time Analysis for Real-Time Embedded System Software", *Proc. IEEE Aerospace Conf.*, 1-10, 2005.

[Jon08] M. Jonsson and K. Kunert, "Reliable Hard Real-Time Communication in Industrial and Embedded Systems", *Proc. 3^{rd} IEEE Intl. Symp. on Industrial Embedded Systems*, 184-191, 2008.

[Jor97] P. B. Jorgensen and J. Madsen, “Critical Path Driven Cosynthesis for Heterogeneous Target Architectures”, *Proc. Intl. Workshop on Hardware/Software Codesign*, 15-19, 1997.

[Jun04] D. B. Junior, F. Vargas, M. B. Santos, I. C. Teixeira, and J. P. Teixeira, “Modeling and Simulation of Time Domain Faults in Digital Systems”, *Proc. 10^{th} IEEE Intl. On-Line Testing Symp.*, 5-10, 2004.

[Kan03a] N. Kandasamy, J. P. Hayes, and B. T. Murray, “Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems”, *IEEE Trans. on Computers*, 52(2), 113-125, 2003.

[Kan03b] N. Kandasamy, J. P. Hayes, and B. T. Murray “Dependable Communication Synthesis for Distributed Embedded Systems,” *Proc. Computer Safety, Reliability and Security Conf.*, 275–288, 2003.

[Kim99] K. Kimseng, M. Hoit, N. Tiwari, and M. Pecht, “Physics-of-Failure Assessment of a Cruise Control Module”, *Microelectronics Reliability*, 39, 1423-1444, 1999.

[Kop89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, “Distributed Fault-Tolerant Real-Time Systems: The MARS Approach”, *IEEE Micro*, 9(1), 25-40, 1989.

[Kop90] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, “Tolerating Transient Faults in MARS”, *Proc. 20^{th} Intl. Symp. on Fault-Tolerant Computing*, 466-473, 1990.

[Kop93] H. Kopetz and G. Grunsteidl, “TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems”, *Proc. 23^{rd} Intl. Symp. on Fault-Tolerant Computing*, 524-533, 1993.

[Kop97] H. Kopetz, *Real-Time Systems-Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[Kop03] H. Kopetz and G. Bauer, "The Time-Triggered Architecture", *Proc. of the IEEE*, 91(1), 112-126, 2003.

[Kop04] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri, "From a Federated to an Integrated Architecture for Dependable Embedded Real-Time Systems", *Tech. Report 22, Technische Universität Wien*, 2004.

[Kor07] I. Koren and C. M. Krishna, "Fault-Tolerant Systems", *Morgan Kaufmann Publishers*, 2007.

[Kwa01] S. W. Kwak, B. J. Choi, and B. K. Kim, "An Optimal Checkpointing-Strategy for Real-Time Control Systems under Transient Faults", *IEEE Trans. on Reliability*, 50(3), 293-301, 2001.

[Kri93] C. M. Krishna and A. D. Singh, "Reliability of Checkpointed Real-Time Systems Using Time Redundancy", *IEEE Trans. on Reliability*, 42(3), 427-435, 1993.

[Kwo96] Y. K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, 7(5), 506-521, 1996.

[Lib00] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems", *IEEE Trans. on Computers*, 49(9), 906-914, 2000.

[Lin00] M. Lindgren, H. Hansson, and H. Thane, "Using Measurements to Derive the Worst-Case Execution Time", *Proc. 7^{th} Intl. Conf. on Real-Time Computing Systems and Applications*, 15-22, 2000.

[Liu73] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, *J. of the ACM*, 20(1), 46-61, 1973.

[Mah04] A. Maheshwari, W. Burleson, and R. Tessier, “Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits”, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 12(3), 299-311, 2004.

[Mah88] A. Mahmood and E. J. McCluskey, “Concurrent Error Detection Using Watchdog Processors - A Survey”, *IEEE Trans. on Computers*, 37(2), 160-174, 1988.

[May78] T. C. May and M. H. Woods, “A New Physical Mechanism for Soft Error in Dynamic Memories”, *Proc. 16th Intl. Reliability Physics Symp.*, 33-40, 1978.

[Mel04] R. Melhem, D. Mosse, and E. Elnozahy, “The Interplay of Power Management and Fault Recovery in Real-Time Systems”, *IEEE Trans. on Computers*, 53(2), 217-231, 2004.

[Mel00] P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki, and P. Narasimhan, “Realize: Resource Management for Soft Real-Time Distributed Systems”, *Proc. DARPA Information Survivability Conf.*, 1, 281–293, 2000.

[Met98] C. Metra, M. Favalli, and B. Ricco, “On-line Detection of Logic Errors due to Crosstalk, Delay, and Transient Faults”, *Proc. Intl. Test Conf. (ITC)*, 524-533, 1998.

[Mir05] B. Miramond and J.-M. Delosme, “Design Space Exploration for Dynamically Reconfigurable Architectures”, *Proc. Design, Automation and Test in Europe (DATE)*, 366-371, 2005.

[Mir95] G. Miremadi and J. Torin, "Evaluating Processor-Behaviour and Three Error-Detection Mechanisms Using Physical Fault-Injection", *IEEE Trans. on Reliability*, 44(3), 441-454, 1995.

[Moh03] K. Mohanram and N. A. Touba, "Cost-Effective Approach for Reducing Soft Error Failure Rate in Logic Circuits", *Proc. Intl. Test Conf. (ITC)*, 893-901, 2003.

[Nah02a] Nahmsuk Oh, P. P. Shirvani, and E. J. McCluskey, "Control-Flow Checking by Software Signatures", *IEEE Trans. on Reliability*, 51(2), 111-122, 2002.

[Nah02b] Nahmsuk Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors", *IEEE Trans. on Reliability*, 51(1), 63-75, 2002.

[Nah02c] Nahmsuk Oh and E. J. McCluskey, "Error Detection by Selective Procedure Call Duplication for Low Energy Consumption", *IEEE Trans. on Reliability*, 51(4), 392-402, 2002.

[Nic04] B. Nicolescu, Y. Savaria, and R. Velazco, "Software Detection Mechanisms Providing Full Coverage against Single Bit-Flip Faults", *IEEE Trans. on Nuclear Science*, 51(6), 3510-3518, 2004.

[Nor96] E. Normand, "Single Event Upset at Ground Level", *IEEE Trans. on Nuclear Science*, 43(6), 2742-2750, 1996.

[Ora94] A. Orailoglu and R. Karri, "Coactive Scheduling and Checkpoint Determination during High Level Synthesis of Self-Recovering Microarchitectures", *IEEE Trans. on VLSI Systems,* 2(3), 304-311, 1994.

[Pat08] P. Patel-Predd, "Update: Transistors in Space", *IEEE Spectrum*, 45(8), 17-17, 2008.

[Pen95] L. Penzo, D. Sciuto, and C. Silvano, "Construction Techniques for Systematic SEC-DED Codes with Single Byte Error Detection and Partial Correction Capability for Computer Memory Systems", *IEEE Trans. on Information Theory*, 41(2), 584-591, 1995.

[Pet05] P. Peti, R. Obermaisser, and H. Kopetz, "Out-of-Norm Assertions", *Proc. 11th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 209-223, 2005.

[Pet06] P. Peti, R. Obermaisser, and H. Paulitsch, "Investigating Connector Faults in the Time-Triggered Architecture", *Proc. IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, 887-896, 2006.

[Pin04] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications", *Proc. Design, Automation and Test in Europe Conf. (DATE),* 1164–1169, 2004.

[Pin08] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-Tolerant Distributed Deployment of Embedded Control Software", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (CAD)*, 27(5), 906-919, 2008.

[Pir06] E. Piriou, C. Jego, P. Adde, R. Le Bidan, and M. Jezequel, "Efficient Architecture for Reed Solomon Block Turbo Code", *In Proc. IEEE Intl. Symp. on Circuits and Systems (ISCAS)*, 4 pp., 2006.

[Pop03] P. Pop, "Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems", *Ph. D. Thesis No. 833, Dept. of Computer and Information Science, Linköping University*, 2003.

[Pop04a] P. Pop, P. Eles, Z. Peng, V. Izosimov, M. Hellring, and O. Bridal, "Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications", *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 1028-1033, 2004.

[Pop04b] P. Pop, P. Eles, Z. Peng, and V. Izosimov, "Schedulability-Driven Partitioning and Mapping for Multi-Cluster Real-Time Systems", *Proc. 16th Euromicro Conf. on Real-Time Systems*, 91-100, 2004.

[Pop04c] P. Pop, P. Eles, Z. Peng, and T. Pop, "Scheduling and Mapping in an Incremental Design Methodology for Distributed Real-Time Embedded Systems", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 12(8), 793-811, 2004.

[Pop09] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "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, 2009.

[Pop07] P. Pop, K. Poulsen, V. Izosimov, and P. Eles, "Scheduling and Voltage Scaling for Energy/Reliability Trade-offs in Fault-Tolerant Time-Triggered Embedded Systems", *Proc. 5th Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 233-238, 2007.

[Pra94] S. Prakash and A. Parker, "Synthesis of Application-Specific Multiprocessor Systems Including Memory Components", *J. of VLSI Signal Processing*, 8(2), 97-116, 1994.

[Pun97] S. Punnekkat and A. Burns, “Analysis of Checkpointing for Schedulability of Real-Time Systems”, *Proc. Fourth Intl. Workshop on Real-Time Computing Systems and Applications*, 198-205, 1997.

[Ree93] C. R. Reevs, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, 1993.

[Ros05] D. Rossi, M. Omana, F. Toma, and C. Metra, “Multiple Transient Faults in Logic: An Issue for Next Generation ICs?”, *Proc. 20^{th} IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 352-360, 2005.

[Sav97] T. Savor and R. E. Seviora, “An Approach to Automatic Detection of Software Failures in Real-Time Systems”, *In Proc. 3^{rd} IEEE Real-Time Technology and Applications Symp. (RTAS)*, 136-146, 1997.

[Sci98] D. Sciuto, C. Silvano, and R. Stefanelli, “Systematic AUED Codes for Self-Checking Architectures”, *Proc. IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 183-191, 1998.

[Shi00] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, “Software-Implemented EDAC Protection against SEUs”, *IEEE Trans. on Reliability*, 49(3), 273-284, 2000.

[Shy07] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, “Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance”, *In Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 297-306, 2007.

[Sil07] V. F Silva, J. Ferreira, and J. A. Fonseca, "Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses", *In Proc. IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, 1107-1114, 2007.

[Sos94] J. Sosnowski, "Transient Fault Tolerance in Digital Systems", *IEEE Micro*, 14(1), 24-35, 1994.

[Sri95] S. Srinivasan and N. K. Jha, "Hardware-Software Co-Synthesis of Fault-Tolerant Real-Time Distributed Embedded Systems", *Proc. of Europe Design Automation Conf.*, 334-339, 1995.

[Sri96] G. R. Srinivasan, "Modeling the Cosmic-Ray-induced Soft-Error Rate in Integrated Circuits: An Overview", *IBM J. of Research and Development*, 40(1), 77-89, 1996.

[Sta97] R. P. Stanley, "Enumerative Combinatorics", Vol. I, Cambridge Studies in Advanced Mathematics 49, *Cambridge University Press*, 1997.

[Ste07] L. Sterpone, M. Violante, R.H. Sorensen, D. Merodio, F. Sturesson, R. Weigand, and S. Mattsson, "Experimental Validation of a Tool for Predicting the Effects of Soft Errors in SRAM-Based FPGAs", *IEEE Trans. on Nuclear Science*, 54(6), Part 1, 2576-2583, 2007.

[Sto96] N. Storey, "Safety-Critical Computer Systems", *Addison-Wesley*, 1996.

[Str06] B. Strauss, M. G. Morgan, Jay Apt, and D. D. Stancil, "Unsafe at Any Airspeed?", *IEEE Spectrum*, 43(3), 44-49, 2006.

[Sun95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and

Chong Sang Kim, “An Accurate Worst Case Timing Analysis for RISC Processors”, *IEEE Trans. on Software Engineering*, 21(7), 593-604, 1995.

[Sze05] D. Szentivanyi, S. Nadjm-Tehrani, and J. M. Noble, “Optimal Choice of Checkpointing Interval for High Availability”, *Proc. 11th Pacific Rim Dependable Computing Conf.*, 8pp., 2005.

[Tan96] H. H. K. Tang, “Nuclear Physics of Cosmic Ray Interaction with Semiconductor Materials: Particle-Induced Soft Errors from a Physicist’s Perspective”, *IBM J. of Research and Development*, 40(1), 91-108, 1996.

[Tin94] K. Tindell and J. Clark, “Holistic Schedulability Analysis for Distributed Hard Real-Time Systems”, *Microprocessing and Microprogramming*, 40, 117-134, 1994.

[Tri05] S. Tripakis, “Two-phase Distributed Observation Problems”, *In Proc. 5th Intl. Conf. on Application of Concurrency to System Design*, 98-105, 2005.

[Tro06] I. A. Troxel, E. Grobelny, G. Cieslewski, J. Curreri, M. Fischer, and A. George, “Reliable Management Services for COTS-based Space Systems and Applications”, *Proc. Intl. Conf. on Embedded Systems and Applications (ESA)*, 169-175, 2006.

[Tsi01] Y. Tsiatouhas, T. Haniotakis, D. Nikolos, and C. Efstathiou, “Concurrent Detection of Soft Errors Based on Current Monitoring”, *Proc. Seventh Intl. On-Line Testing Workshop*, 106-110, 2001.

[Ull75] D. Ullman, “NP-Complete Scheduling Problems,” *Computer Systems Science*, 10, 384–393, 1975.

[Vel07] R. Velazco, P. Fouillat, and R. Reis, (*Editors*) “Radiation Effects on Embedded Systems”, *Springer*, 2007.

[Vra97] H. P. E. Vranken, M. P. J. Stevens, and M. T. M. Segers, "Design-for-Debug in Hardware/Software Co-Design", *In Proc. 5th Intl. Workshop on Hardware/Software Codesign*, 35-39, 1997.

[Wan03] J. B. Wang, "Reduction in Conducted EMI Noises of a Switching Power Supply after Thermal Management Design," *IEE Proc. - Electric Power Applications*, 150(3), 301-310, 2003.

[Wat04] N. Wattanapongsakorn and S. P. Levitan, "Reliability Optimization Models for Embedded Systems with Multiple Applications", *IEEE Trans. on Reliability*, 53(3), 406-416, 2004.

[Wei04] Wei Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam, "Compact Thermal Modeling for Temperature-Aware Design," *Proc. 41st Design Automation Conf. (DAC)*, 878-883, 2004.

[Wei06] T. Wei, P. Mishra, K. Wu, and H. Liang, "Online Task-Scheduling for Fault-Tolerant Low-Energy Real-Time Systems", *In Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, 522-527, 2006.

[Wil08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puuat, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools", *ACM Trans. on Embedded Computing Systems (TECS)*, 7(3), 36.1-36.53, 2008.

[Xie04] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-Aware Co-synthesis for Embedded Systems", *Proc. 15th IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors*, 41-50, 2004.

[Yin03] Ying Zhang and K. Chakrabarty, "Fault Recovery Based on Checkpointing for Hard Real-Time Embedded Systems", *Proc. IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 320-327, 2003.

[Yin04] Ying Zhang, R. Dick, and K. Chakrabarty, "Energy-Aware Deterministic Fault Tolerance in Distributed Real-Time Embedded Systems", *Proc. 42nd Design Automation Conf. (DAC)*, 550-555, 2004.

[Yin06] Ying Zhang and K. Chakrabarty, "A Unified Approach for Fault Tolerance and Dynamic Power Management in Fixed-Priority Real-Time Embedded Systems", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(1), 111-125, 2006.

[Zha06] M. Zhang, S. Mitra, T. M. Mak, N. Seifert, N. J. Wang, Q. Shi, K. S. Kim, N. R. Shanbhag, and S. J. Patel, "Sequential Element Design With Built-In Soft Error Resilience", *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems*, 14(12), 1368-1378, 2006.

[Zho06] Q. Zhou and K. Mohanram, "Gate Sizing to Radiation Harden Combinational Logic", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (CAD)*, 25(1), 155-166, 2006.

[Zho08] Q. Zhou, M. R. Choudhury, and K. Mohanram, "Tunable Transient Filters for Soft Error Rate Reduction in Combinational Circuits", *Proc. 13th European Test Symp. (ETS)*, 179-184, 2008

[Zhu05] D. Zhu, R. Melhem, and D. Mosse, "Energy Efficient Configuration for QoS in Reliable Parallel Servers", *In Lecture Notes in Computer Science*, 3463, 122-139, 2005.

[Ziv97] A. Ziv and J. Bruck, "An On-Line Algorithm for Checkpoint Placement", *IEEE Trans. on Computers*, 46(9), 976-985, 1997.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.