Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems

Viacheslav Izosimov

Linköping University
INSTITUTE OF TECHNOLOGY

ISBN 91-85643-72-6 ISSN 0280-7971
PRINTED IN LINKÖPING, SWEDEN
BY LINKÖPINGS UNIVERSITET

Scheduling and Optimization of Fault-Tolerant Embedded Systems

by

Viacheslav Izosimov

November 2006
ISBN 91-85643-72-6
Linköping Studies in Science and Technology
Thesis No. 1277
ISSN 0280-7971
LiU-Tek-Lic-2006:58

ABSTRACT

Safety-critical applications have to function correctly even in 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. These techniques are considered in the context of distributed real-time systems with non-preemptive static cyclic scheduling.

Safety-critical applications have 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 consideration of fault tolerance are required.

The 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, and checkpoint distribution.

Dedicated scheduling techniques and mapping optimization strategies are also proposed to handle customized transparency requirements associated with processes and messages. By providing fault containment, transparency can, potentially, improve testability and debugability of fault-tolerant applications.

The efficiency of the proposed scheduling techniques and design optimization strategies is evaluated with extensive experiments conducted on a number of synthetic applications and a real-life example. The experimental results show that considering fault tolerance during system-level design optimization is essential when designing cost-effective fault-tolerant embedded systems.

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

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

To My Parents

Abstract

SAFETY-CRITICAL APPLICATIONS HAVE to function correctly even in 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. These techniques are considered in the context of distributed real-time systems with non-preemptive static cyclic scheduling.

Safety-critical applications have 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 consideration of fault tolerance are required.

The 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, and checkpoint distribution.

Dedicated scheduling techniques and mapping optimization strategies are also proposed to handle customized transparency requirements associated with processes and messages. By providing fault containment, transparency can, potentially, improve testability and debugability of fault-tolerant applications.

The efficiency of the proposed scheduling techniques and design optimization strategies is evaluated with extensive experiments conducted on a number of synthetic applications and a real-life example. The experimental results show that considering fault tolerance during system-level design optimization is essential when designing cost-effective fault-tolerant embedded systems.

Acknowledgements

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

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 also would like to express many thanks to my current and former colleagues at ESLAB and IDA for creating a nice friendly working environment.

I am also grateful to my sister, my brother, and all my friends who have supported me during writing of the thesis.

Finally, I devote this thesis to my parents who have been encouraging me during my 20 years of studies. Last, but not least, my deepest gratitude is towards my girlfriend, Yevgeniya Kyselova, for her love, patience, and support.

Contents

Chapter 1
Introduction

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

Faults in a distributed embedded system can be permanent, intermittent or transient (also known as soft errors). Permanent faults cause long-term malfunctioning of components. Transient and intermittent faults appear for a short time. Causes of intermittent faults are within system boundaries, while causes of transient faults are external to the system. The effects of transient and intermittent faults, even though they appear for a short time, can be as devastating as the effects of permanent faults. They may corrupt data or lead to logic miscalculations, which can result in a fatal failure.

Due to their higher rate, transient and intermittent faults cannot be addressed in a cost-effective way by applying traditional *hardware-based* fault tolerance techniques suitable for tolerating permanent faults. In this thesis we deal with tran-

sient and intermittent faults and consider several *software-based* fault tolerance techniques, including re-execution, software replication, and rollback recovery with checkpointing.

Embedded systems with fault tolerance have to be carefully designed and optimized, in order to satisfy strict timing requirements without exceeding a certain limited amount of resources. Moreover, not only performance 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 a set of design optimization problems and present the contributions of our work. An overview of the thesis with short descriptions of the chapters is also presented.

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: high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [Mah04, Con03, Har01].

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 [Cor04, Han02] and proper fault tolerance techniques against transient faults are needed.

Intermittent faults are also very common in automotive systems. 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 faults produced by other components [Kim99].

Causes of transient and intermittent faults can vary a lot. At first, we will list possible causes of transient faults, which are outside of system boundaries and may include several external factors such as:

- *(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];
- *electromagnetic interference* by mobile phones, wireless communication equipment [Str06], power lines, and radar [Han02];
- *lightning storms* that can affect power supply, current lines, or directly electronic components [Hei05].

In contrast to transient faults, the causes of intermittent faults are within the system boundaries. They can be triggered, for example, by one device affecting other components through radio emission or via the power supply. One such component can create several intermittent faults at the same time. There are several possible causes of intermittent faults listed in literature:

- *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];
- *software errors* (also called *Heisenbugs*) that manifest themselves under rare circumstances and, therefore, are difficult to find during software testing [Kop04].

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, 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, the JAS 39 Gripen [Als01] architecture contains seven hardware replicas that can tolerate up to three transient faults. However, such a solution 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, Yin06], and re-execution [Kan03a]. However,

if applied in a straightforward manner to an existing design, techniques against transient faults introduce significant time overheads, which can lead to unschedulable solutions. On the other hand, using faster components or a larger number of resources may not be affordable due to cost constraints. Therefore, efficient design optimization techniques are required in order to meet time and cost constraints in the context of fault tolerant systems.

Transient faults are also common for communication channels, even though we do not deal with them explicitly. Fault tolerance against multiple transient faults affecting communications have already been studied. Solutions such as a cyclic redundancy code (CRC) are implemented in communication protocols available on the market [Kop93, Fle04].

1.2 Contributions

In our approach, an embedded system is represented as a set of processes communicating by sending messages. Processes are *mapped* on computation nodes connected to the communication infrastructure. The mapping of processes is decided using optimization algorithms such that the performance is maximized. Processes and communication schedules are determined off-line by *static cyclic scheduling*. Our design optimization thoroughly considers the impact of communications on the overall system performance.

To provide resiliency against transient faults, processes are *assigned* with re-execution, replication, or recovery with checkpointing. Design optimization algorithms consider various overheads introduced with fault tolerance techniques. In addition to performance and cost-related requirements, debugability and testability of embedded systems are also taken into account during design optimization. In this thesis, we relate the two latter properties to transparency, which provides fault containment

and, thus, potentially, can improve the debugability and testability of the system.

The main contributions of this thesis are:

- **a static cyclic scheduling framework** [Izo05] to schedule processes and messages, providing fault isolation of computation nodes;
- **a conditional static scheduling framework** [Izo06b] that creates more efficient schedules than the ones generated with the above mentioned technique. This approach also allows to trade-off between transparency and schedule length;
- **a technique for schedule length estimation of conditional schedules** [Izo06a] that evaluates design solutions in terms of performance without the need of extensive computation;
- **mapping and fault tolerance policy assignment strategies** [Izo05, Izo06a] for mapping of processes to computation nodes and assigning of a proper combination of fault tolerance techniques to processes, such that the performance is maximized;
- **an approach to the optimization of checkpoint distribution in rollback recovery** [Izo06c].

1.3 Thesis Overview

The thesis is structured as follows:

- **Chapter 2** introduces basic concepts of fault tolerance in the context of system-level design and optimization algorithms. It also provides reference sources of related work.
- **Chapter 3** presents our hardware architecture, application model, and fault model. We introduce the notion of transparency and frozenness, related to testability and debugability requirements of applications. This chapter also presents how

to model fault tolerance techniques in the context of static cyclic scheduling.

- **Chapter 4** presents two static cyclic scheduling techniques with fault tolerance requirements, including scheduling with transparency/performance trade-offs. These scheduling techniques are used by design optimization strategies presented in the later chapters to derive fault-tolerant schedule tables.
- **Chapter 5** discusses mapping and policy assignment optimization issues. First, we propose a mapping and fault tolerance policy assignment strategy that combines software replication with re-execution. Second, we present a mapping optimization strategy that can handle transparency properties during design optimization and supports transparency/performance trade-offs. An efficient schedule length estimation technique used as a cost function for the mapping optimization strategy with transparency is proposed.
- **Chapter 6** introduces our checkpoint distribution strategies. We also present mapping and policy assignment optimization with checkpointing and rollback recovery.
- **Chapter 7**, finally, presents our conclusions and directions for future work.

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 classic fault tolerance techniques. Finally, we present relevant research work on design optimization for fault tolerant systems and suggest a possible design flow enhanced with fault tolerance techniques.

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 first, "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 next, "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. 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 certain design constraints are satisfied [Pra94c, Pop04c, Pop04a]. These constraints can include memory constraints, power constraints, as well as security- and safety-related constraints. To further improve performance, 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, Lak99].

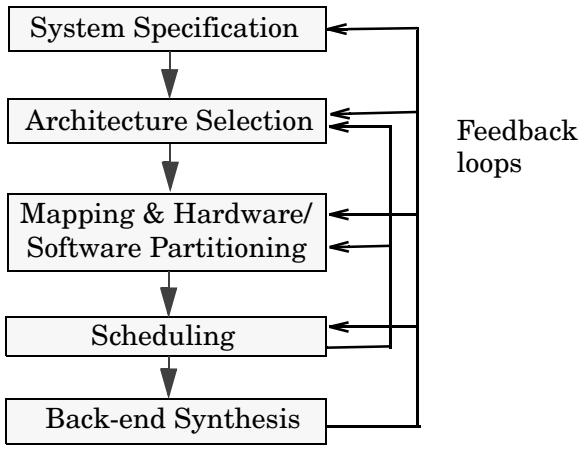


Figure 2.1: Generic Design Flow

After mapping and partitioning, the execution order and start times of processes are analysed in the "scheduling" step. Scheduling can be either static or dynamic. In the case of dynamic scheduling, start times are obtained on-line 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 the form of *schedule tables*. In this thesis we focus on *non-preemptive static cyclic scheduling*. Researchers have developed several algorithms to efficiently produce static schedules off-line. Many of the algorithms are based on list scheduling heuristics [Cof72, Deo98, Jor97, Kwo96].

If, according to the resulted schedule, deadlines are not satisfied, then either mapping or partitioning should be changed at first (see a feedback line in Figure 2.1). If no schedulable solution can be found by optimizing process mapping and/or scheduling, then the hardware architecture needs to be modified and the optimization will be performed again.

Eventually, a schedulable solution will be found and the actual back-end system synthesis of a prototype will begin in 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 with efficient design optimization on early design stages (with mapping and scheduling) to reduce design costs.

2.1.1 OPTIMIZATION HEURISTICS

In general, design optimization by mapping and partitioning is an NP-hard problem [Gar03]. Therefore, exact approaches, producing optimal solutions, such as constraint-logic programming

(CLP) [Hen96], integer-linear programming (ILP) [Pra94c], or branch-and-bound approaches [Kas84], are very time-consuming and impractical for many real-life applications.

To overcome the complexity of mapping and partitioning optimization, various heuristics that provide near-optimal but efficient design solutions were proposed. Usually, in these heuristics, mapping and partitioning are changed incrementally with small modifications, or *moves*, until a schedulable design solution is found.

There are several *general-purpose optimization heuristics* [Ree93] that can be used for system-level design optimization, such as *simulated annealing* [Met53, Col95, Rab93, Ele97], *tabu search* [Glo86, Han86, Ele97, Man04], and *genetic algorithms* [Hol75, Gol89, Con05, Bax95]. Many researchers either adapt general-purpose heuristics or develop custom algorithms that are often greedy-based and, possibly, with some methods to recover from local optima.

At first, system-level design optimization heuristics were applied to solve simple hardware/software partitioning of an application mapped on a monoprocessor system, where some functions were implemented on ASICs or FPGAs for acceleration [Cho95, Gup95, Axe96, Ele97, Ern93]. More advanced approaches consider the design of complex and heterogeneous systems [Bol97, Dav98, Dav99, Dic98, Lak99]. Later, these approaches were extended towards the design of distributed embedded systems [Pop03]. For example, the design of multi-cluster distributed systems was considered in [Pop04a] and [Pop04b]. Moreover, issues related to the design optimization of fault-tolerant systems have recently received a close attention from the research community. In Section 2.4, after presenting basic fault tolerance techniques, we will discuss related work on design optimization with fault tolerance, highlighting the limitations of approaches proposed so far.

2.2 Fault Tolerance Techniques

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

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: watchdogs, assertions, signatures, duplication, memory protection codes, and few others.

Signatures. Signatures [Nah02a, Jie92, Mir95, Sci98, Nic04] are one of the most powerful error detection techniques. In this technique, a set of logic operations can be assigned with precomputed "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 errors. The simplest watchdog schema, *watchdog timer*, monitors the execution time of processes, whether it exceeds a certain limit [Mir95]. Another approach is to incorporate 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]. EDAC code separately protects each memory block to avoid propagation of errors. A common schema is "single-error-correcting, double-error-detecting" (SEC-DEC) [Pen95] that can correct one error and detect two errors 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* α, which is the time needed for detecting faults. The error-detection overhead can vary a lot with the error-detection tech-

nique 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

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

Figure 2.2 shows 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-

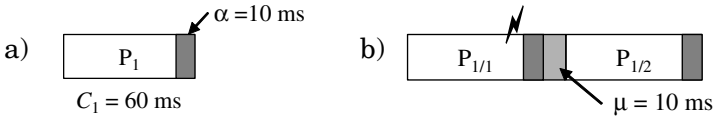


Figure 2.2: Re-execution

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 in Chapter 6.

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 overhead due to re-execution can be reduced with more complex fault tolerance techniques such as *rollback recovery with checkpointing* [Pun97, Yin06, Ora94]. The main principle of this technique is to restore the last non-faulty state of the failing process, i.e., to *recover* from faults. The last non-faulty state, or *checkpoint*, has to be saved in advance in the static memory and will be restored if the process fails. The part of the process between two checkpoints or between a checkpoint and the end of the process is called *execution segment*.

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, which is easier for system design and optimization [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 check-

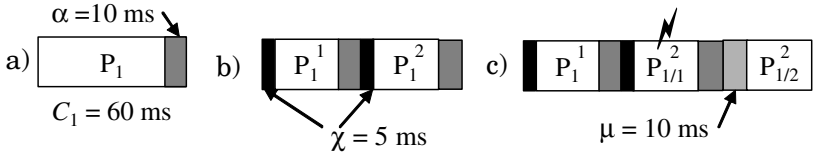


Figure 2.3: Rollback Recovery with Checkpointing

points 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 faults 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 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 simple re-execution.

2.2.4 ACTIVE AND PASSIVE REPLICATION

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

In the case of active replication [Xie04], all replicas of processes are executed independently of fault occurrences. In the case of passive replication, also known as *primary-backup* [Ahn97], 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, 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*

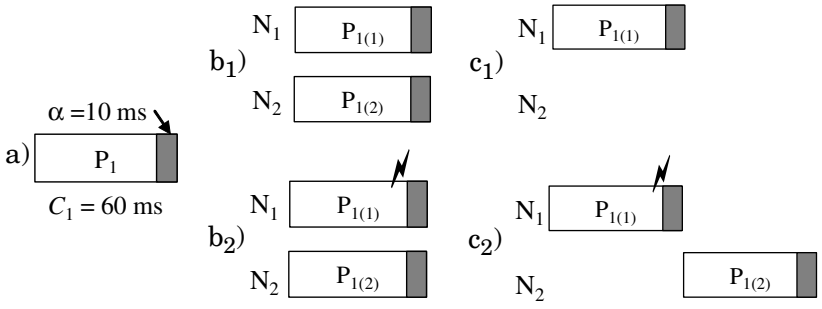


Figure 2.4: Active Replication (b) and Primary-Backup (c)

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$_2$.

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

2.3 Transparency

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, debugging, verification and testing become very difficult. A possible solution against this problem is *transparency*.

Originally, Kandasamy et al. [Kan03a] propose *transparent* re-execution, where recovering from a transient fault on one computation node is hidden from other nodes. Transparency has the advantage of fault containment and increased debugability. Since the occurrence of faults in certain process does not affect

the execution of other processes, the total number of execution scenarios is reduced. Therefore, less number of execution alternatives have to be considered during debugging, testing, and verification. However, transparency can increase the worst-case delay of processes, reducing performance of the embedded system.

2.4 Design Optimization with Fault Tolerance

Fault-tolerant embedded systems have to be optimized in order to meet time and cost constraints. Researchers have shown that schedulability of an application can be guaranteed for pre-emptive on-line scheduling under the presence of a single transient fault [Ber94, Bur96, Han03, Yin06].

Liberato et al. [Lib00] propose an approach for design optimization of monoprocessor systems in the presence of multiple transient faults and in the context of pre-emptive earliest-deadline-first (EDF) scheduling.

Hardware/software co-synthesis with fault tolerance is addressed in [Sri95] in the context of event-driven fixed priority scheduling. Hardware and software architectures are synthesized simultaneously, providing a specified level of fault tolerance and meeting the performance constraints. 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.

Xie et al. [Xie04] propose a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Introducing redundant processes into a pre-designed schedule is used in [Con05] in order to improve error detection. Both approaches only consider one single fault.

Power-related optimization issues in fault-tolerant applications are tackled in [Yin04] and [Jia05]. Ying Zhang et al.

[Yin04] study fault tolerance and dynamic power management. Rollback recovery with checkpointing is used in order to tolerate multiple transient faults in the context of message-passing distributed systems. Fault tolerance is applied on top of a predesigned system, whose process mapping ignores the fault tolerance issue.

Kandasamy et al. [Kan03a] propose constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. The work was later extended with fault-tolerant transmission of messages on a time-division multiple access bus [Kan03b]. Both papers consider only one fault per computation node. 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] propose a simple heuristic for combining several static schedules in order to mask fault patterns. Passive replication is used in [Alo01] to handle a single failure in multiprocessor systems so that timing constraints are satisfied. Multiple failures are 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 is considering optimal assignment of fault tolerance policies. Several other limitations of previous research, which we will also overcome in this thesis, are the following:

- design optimization of embedded systems with fault tolerance is very limited, as, for example, process mapping is not considered together with fault tolerance issues;
- multiple faults are not addressed in the framework of static cyclic scheduling; and
- transparency, if at all addressed, is restricted to a whole computation node and is not flexible.

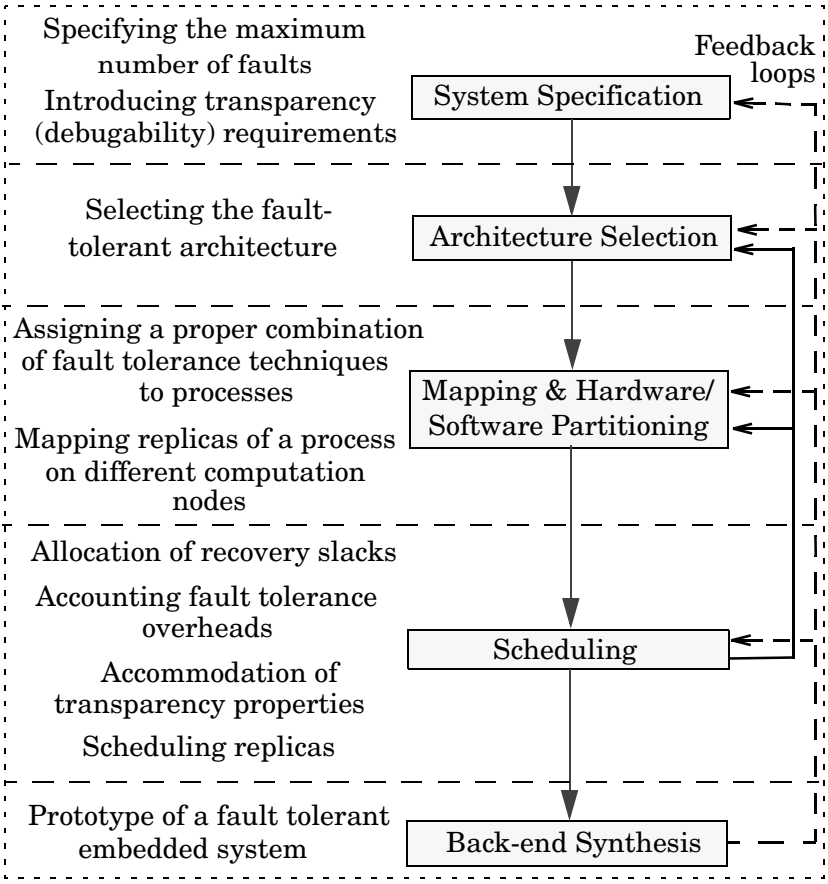


Figure 2.5: Design Flow with Fault Tolerance

2.4.1 DESIGN FLOW WITH FAULT TOLERANCE TECHNIQUES

In Figure 2.5 we enhance the generic design flow presented in Figure 2.1, with the consideration of fault tolerance techniques.

In the "System Specification" step, designers specify the maximum number of faults, which have to be tolerated. They introduce transparency (debugability) requirements in order to improve debugability and testability of the system.

In the second step, the fault-tolerant architecture with the sufficient level of redundancy needs to be chosen. For example, in order to tolerate a single permanent fault, designers can decide to duplicate computation nodes and the bus, as in the Time-Triggered Architecture (TTA) [Kop03].

In the "Mapping & Hardware/Software Partitioning" step, processes are assigned with fault-tolerance techniques against transient faults. We call the assignment of fault tolerance techniques to processes *fault-tolerance policy assignment*. For example, some processes can be assigned with re-execution, some with active replication, and some with a combination of re-execution and replication. Designers also choose such a mapping that replicas of a process are mapped on different computation nodes.

Besides the classical scheduling task, our scheduling algorithms for fault tolerance perform

- an allocation of recovery slacks on computation nodes for re-execution and rollback recovery;
- accounting for recovery, checkpointing, and error-detection overheads;
- an accommodation of transparency properties of processes and messages into schedules;
- scheduling replicas of a process and their outputs.

In the last step, "Back-end Synthesis", the fault-tolerant design is synthesized into a prototype.

Chapter 3
Preliminaries

IN THIS CHAPTER we introduce our application model, hardware architecture, and fault model. We also present our approach to process recovery.

3.1 System Model

In this section we present details regarding our application model and system architecture.

3.1.1 APPLICATION MODEL

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

Processes are non-preemptable and cannot be interrupted by other processes. Processes send their output values encapsulated in messages, when completed. All required inputs have to arrive before activation of the process. Precedence constraints, e.g. that one process cannot start before the others terminate, are introduced with edges without messages. Figure 3.1a shows a simple application represented as a graph composed of five nodes (processes P_1 to P_5) connected with five edges (messages m_1, m_2, and m_3, plus two precedence constraints).

In this thesis, we will consider *hard real-time* applications, common for safety-critical systems. Time constraints are imposed with a global hard deadline D, which is an interval of time within which the application $\mathcal{A}$ *has to* complete. Some processes may also have local deadlines d_{local}. We model such deadlines by inserting a dummy node between a process, that has a local deadline, and the sink node of process graph $\mathcal{G}$. The dummy node is a process with execution time $C_{dummy} = D - d_{local}$, which, however, is not allocated to any resource [Pop03].

3.1.2 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

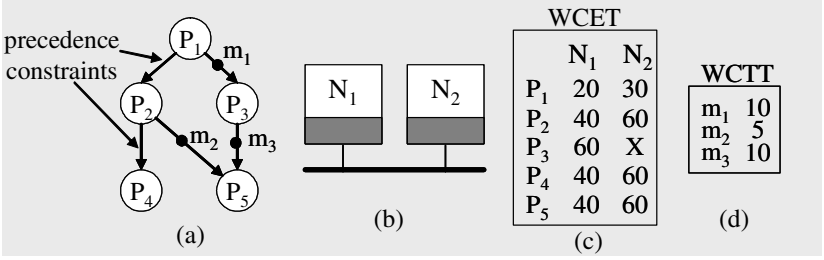


Figure 3.1: A Simple Application and a Hardware Architecture

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* (allocated) 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. We consider that the mapping of the application is not fixed and has to be determined as part of the design optimization.

We consider that for each process its worst-case execution time (WCET) is given. Using WCET guarantees predictable behaviour, which is important for safety-critical systems. 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].

Figure 3.1c shows the worst-case execution times of processes of the application depicted in Figure 3.1a. For example, process P_2 has the worst-case execution 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.

In the case of processes mapped on the same computation node, message transmission time between them is accounted for in the worst-case execution time of the sending process. If processes are mapped on different computation nodes, then messages between them are sent through the communication network. We consider that the worst-case transmission time (WCTT) of messages is given. The worst-case transmission time of messages, however, does not include waiting time in the queue of the communication controller. Figure 3.1d shows the worst-case transmission times of messages for the application depicted in Figure 3.1a.

In this thesis, we consider a *static non-preemptive* scheduling approach, where both communications and processes are statically scheduled. The start times of processes and sending times of messages 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 real-time 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 3.2b we depict a static schedule for the application and the hardware architecture presented in Figure 3.1 in the case where this application is mapped as shown in Figure 3.2a. 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 90 ms, respectively. The schedule table of N_2 contains start times of P_2 and P_4, 20 and 80 ms, plus sending time of message m_2, which is 80 ms. According to the static schedule, the application will complete at 140 ms, which satisfies the deadline D of 200 ms.

Although, so far, we have illustrated the generation of a single schedule for a single execution scenario, in general, an application can have different execution scenarios. 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

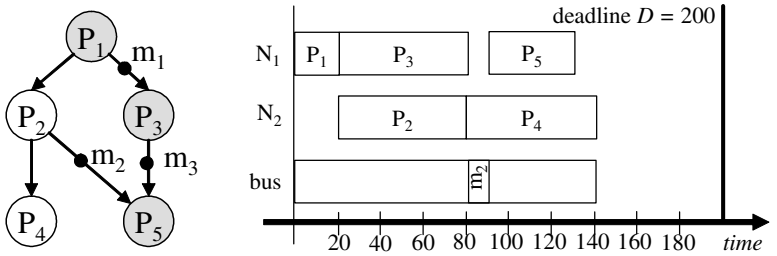


Figure 3.2: A Static Schedule

real-time scheduler will choose the appropriate schedule that corresponds to the actual conditions. If the conditions change, the real-time scheduler will accordingly switch to the appropriate schedule. This mechanism will be exploited in the following sections for capturing the behaviour of fault-tolerant applications. In the case of fault-tolerant systems and if the alternative scenarios are due to fault occurrences, the corresponding schedules are called *contingency schedules*.

3.2 Fault Model and Basic Fault Tolerance Techniques

We assume that a maximum number k of transient faults can happen during a system period T. This model is an extension of the single fault model proposed in [Kan03a]. For example, in Figure 3.3a we show a simple application of two processes that has to tolerate the maximum number of two faults, e.g. $k = 2$.

Overheads due to fault tolerance techniques have to be reflected in a system architecture. 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 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

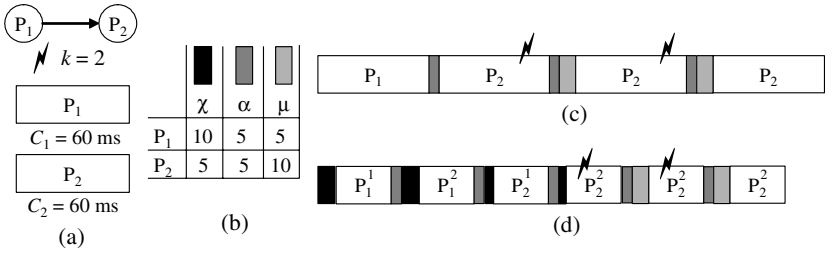


Figure 3.3: Fault Model and Fault Tolerance Techniques

the time needed to restore the process state. Rollback recovery with checkpointing is also characterized by a checkpointing overhead χ_i, which is related to the time needed to store intermediate process states.

We consider that the worst-case time overheads related to the particular fault tolerance techniques are given. For example, Figure 3.3b shows recovery, detection and checkpointing overheads associated with the processes of the simple application depicted in Figure 3.3a. 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 3.3c and Figure 3.3d, respectively.[1] As can be seen, the overheads related to the fault tolerance techniques have a significant impact on the overall system performance. In Figure 3.3d, for example, overheads contribute to a delay of 75 ms, while the execution of processes without overheads would take only 180 ms.

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[2]. We con-

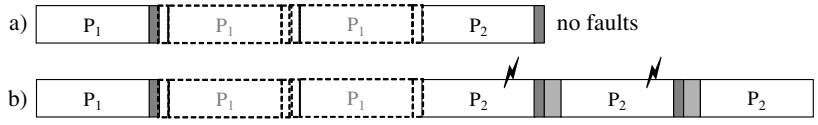


Figure 3.4: Transparency and Frozenness

1. The overhead due to saving process inputs is ignored in the context of re-execution, as discussed in Section 2.2.2.
2. External, i.e., outside the frozen process or message.

sider also that transparency requirements are given with a function $\mathcal{T}: \mathcal{W} \rightarrow \{Frozen, Regular\}$, where $\mathcal{W}$ is the set of all processes and messages sent over the bus.

For example, Figure 3.4 shows the non-fault scenario and the worst-case fault scenario of the application depicted in Figure 3.3a, if re-execution is applied and process P_2 is frozen. Process P_2 is scheduled at 145 ms in both execution scenarios independently of *external* fault occurrences, e.g., faults in process P_1. However, if faults occurrences are *internal*, i.e., within process P_2, process P_2 has to be re-executed as shown in Figure 3.4b.

3.3 Recovery in the Context of Static Cyclic Scheduling

In the context of static cyclic scheduling, each execution scenario has to be explicitly modelled [Ele00]. Tolerating transient faults with re-execution and rollback recovery with checkpointing leads to a large number of execution scenarios. In this section, we present our approach to model re-execution and rollback recovery with checkpointing in the context of static scheduling.

3.3.1 RE-EXECUTION

In the case of re-execution, faults lead to different execution scenarios that correspond to a set of alternative *contingency schedules*. For example, considering the same application as in Figure 3.1a, with a maximum number of faults $k = 1$, re-execution will require three alternative schedules as depicted in Figure 3.5a. 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 real-time scheduler switches from the non-fault schedule S_0 to the schedule S_2 corresponding to a fault in process P_2.

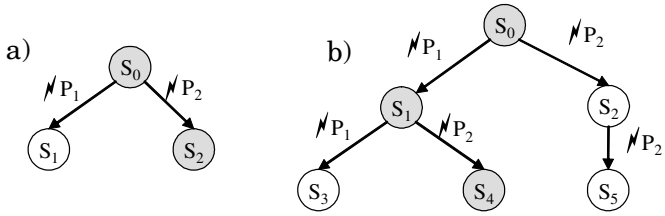


Figure 3.5: Contingency Schedules for Re-execution

Similarly, in the case of multiple faults, every fault occurrence will trigger a switching to the corresponding alternative contingency schedule. Figure 3.5b represents a tree of constructed alternative schedules for the same application of two processes, if two transient faults can happen at maximum, i.e. $k = 2$. For example, as depicted with shaded circles, if process P_1 experiences a fault, the real-time scheduler switches from the non-fault schedule S_0 to contingency schedule S_1. Then, if process P_2 experiences a fault, the real-time scheduler switches to schedule S_4.

3.3.2 ROLLBACK RECOVERY WITH CHECKPOINTING

In a static schedule, similar to re-execution, every recovery action of rollback recovery with checkpointing will lead to different execution scenarios and will correspond to a set of alternative *contingency schedules*.

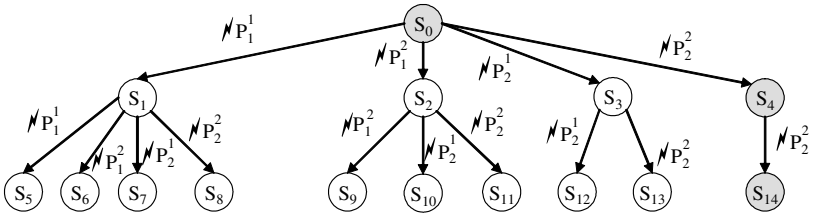


Figure 3.6: Contingency Schedules for Rollback Recovery with Checkpointing

Figure 3.6 represents a tree of constructed alternative schedules for the application in Figure 3.1a and the rollback recovery schema with two checkpoints as in Figure 3.3d. In the schedule, every execution segment P_i^j is considered as a "small process" that is recovered in case of fault occurrences. Therefore, the number of contingency schedules is larger than it is in the case of pure re-execution. In Figure 3.6 we highlight the fault scenario presented in Figure 3.1d with shaded circles. The real-time scheduler switches between schedules S_0, S_4, and S_{14}.

Chapter 4
Scheduling with Fault Tolerance Requirements

In this chapter we propose two scheduling techniques for fault-tolerant embedded systems, 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 contingency schedules and fast schedule generation time.

Both scheduling techniques are based on a *fault-tolerant conditional process graph* (FT-CPG) representation, which is used to generate fault-tolerant schedule tables.

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 the mechanism of masking fault occurrences. The notion of transparency has been introduced with the notion 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 debug and, in principle, safer. Moreover, since transparency reduces the number of execution scenarios, the amount of memory required to store contingency schedules corresponding to these scenarios is less. 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.

Let us illustrate such a trade-off with the example in Figure 4.1, where we have an application consisting of four processes, P_1 to P_4 and three messages, m_1 to m_3, mapped on an architecture with two computation nodes, N_1 and N_2. Messages m_1 and m_2 are sent from P_1 to processes P_4 and P_3, respectively. Message m_3 is sent from P_2 to P_3. The worst-case execution times of each process are depicted in the figure, and the deadline of the application is 210 ms. We consider a fault scenario where two transient faults ($k = 2$) can occur.

Whenever a fault occurs, the faulty process has to be re-executed. As discussed in Section 3.3, in the context of static cyclic scheduling, each fault scenario will correspond to an alternative static schedule. Thus, the real-time scheduler in a computation node that experiences a fault has to switch to another schedule with a new start time for that process. For example, according to the schedule in Figure 4.1a$_1$, the processes are scheduled at times indicated by the white rectangles in the Gantt chart. Once a fault occurs in P_3, the scheduler on node N_2 will have to switch to another schedule. In this schedule, P_3 is delayed with $C_3 + \mu$

to account for the fault, where C_3 is the worst-case execution time of process P_3 and μ is the recovery overhead. If, during the second execution of P_3, a second fault occurs, the scheduler has to switch to another schedule illustrated in Figure 4.1a$_2$.

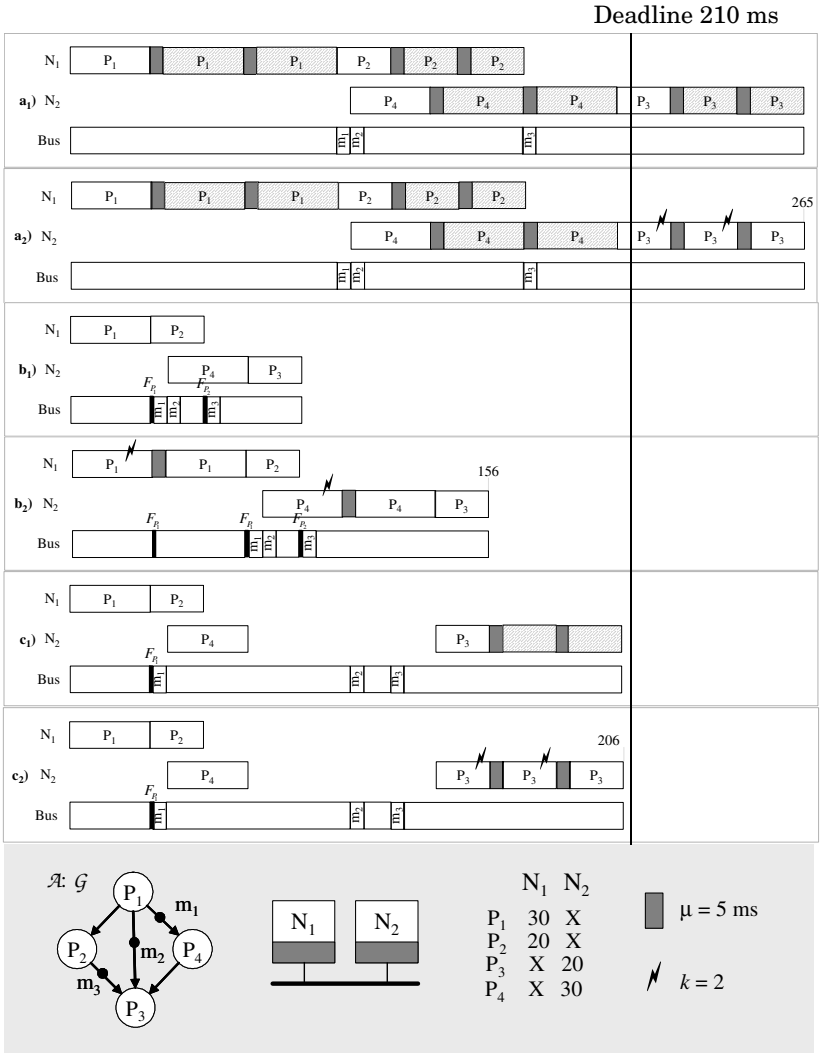


Figure 4.1: Trading-off Transparency for Performance

In Figure 4.1a_1, we have constructed the schedule such that each execution of a process P_i is followed by a *recovery slack*, which is idle time on the computation node, needed to recover (re-execute) the process, in the case that it fails. For example, for P_3 on node N_2, we introduce a recovery slack of $k \times (C_3 + \mu) = 50$ ms to make sure that we can recover P_3 even in the case it experiences the maximum number of faults (Figure 4.1a_2). Thus, a fault occurrence that leads to the re-execution of any process P_i will impact only P_i. We call such an approach *fully transparent* because fault occurrences in a process are transparent to *all* other processes on the same or other computation nodes.

In Figure 4.1 we illustrate three alternative scheduling strategies, representing different transparency/performance trade-offs. For each alternative, we show the schedule when no faults occur (a_1–c_1) and depict the corresponding 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, since this is a situation that can happen in the worst-case scenario. Thus, we would like to have in a_2–c_2 schedules of the worst-case scenario that meet the deadline of 210 ms depicted with a thick vertical line.

In general, a *fully transparent* approach, as depicted in Figure 4.1a_1 and 4.1a_2, has the drawback of producing unnecessarily large delays. In the case of full transparency, the largest delay is produced by the scenario depicted in Figure 4.1a_2, which has to be activated when two faults happen in P_3. Faults in the other processes are masked within the recovery slacks allocated between processes. The worst-case end-to-end delay in the case of full transparency is 265 ms, which will miss the deadline.

To meet the deadline, another approach, depicted in Figure 4.1b_1 and 4.1b_2, is not to isolate the effect of fault occurrences at all. Figure 4.1b_1 shows the execution scenario if no fault occurs. In this case, a fault occurrence in process P_i can affect the schedule of another process P_j. 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 is done via the error message F_{P_1}, depicted as a black rectangle on the bus, which broadcasts the error occurrence on P_1 to other computation nodes. This would lead to a worst-case scenario of only 156 ms, depicted in Figure 4.1b$_2$, that meets the deadline.

However, transparency (masking fault occurrences) is highly desirable because it makes the application easier to debug, and a designer would like to introduce as much transparency as possible without violating the timing constraints. Thus, an approach is required, which allows to fine-tune the application properties such that the deadlines are satisfied and transparency is preserved as much as possible. An example of such an approach is depicted in Figure 4.1c$_1$ and 4.1c$_2$, where tolerating faults in the other processes is transparent to process P_3 and its input messages m_2 and m_3, but not to P_1, P_2, P_4 and m_1. In this case, P_3, m_2 and m_3 are said to be *frozen*, i.e., they have the same start time in all schedules. The debugability is improved because it is easier to observe the behaviour of P_3 in the alternative schedules. Its start time does not change due to the occurrence and handling of faults. Moreover, the memory needed to store the alternative schedules is also reduced with transparency, since there are less start times to store. In this case, the worst-case end-to-end delay of the application is 206, as depicted in Figure 4.1c$_2$, and the deadline is met.

In Figure 4.2, we present fault scenarios to illustrate changes of start times of processes and messages in the case of the *customized* transparency depicted in Figure 4.1c$_1$ and 4.1c$_2$. Figure 4.2a repeats the fault-free scenario as in Figure 4.1c$_1$. In Figure 4.2b, we show that, if process P_1 is affected by faults, the start times of regular process P_2 and regular message m_1 are changed. However, the start times of frozen messages m_2 and m_3, as well as process P_3, are not affected. In Figure 4.2c, process P_4 is affected by faults, however, the start time of frozen

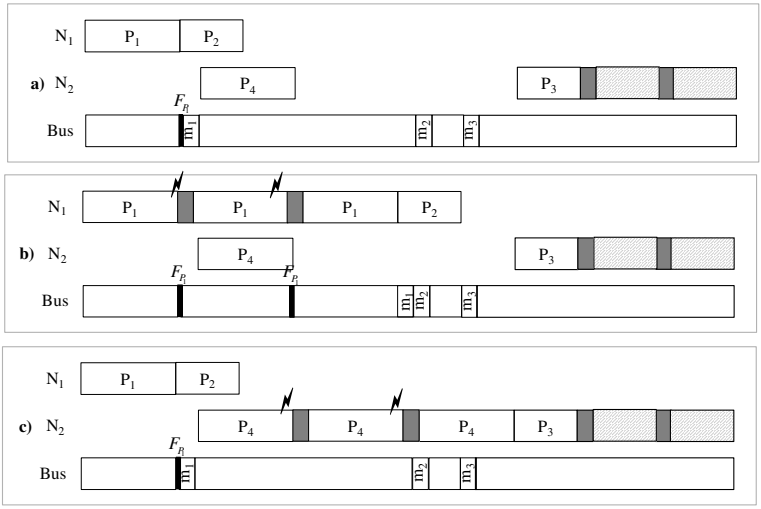


Figure 4.2: Fault Scenarios for Customized Transparency

process P_3 is calculated such that the fault occurrences in process P_4 cannot disrupt it.

In this thesis, we propose an approach to transparency, which offers the designer the possibility to trade-off transparency with performance. Given an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ we will capture the transparency using the function $\mathcal{T}: \mathcal{W} \rightarrow \{Frozen, Regular\}$, where $\mathcal{W}$ is the set of all processes and messages sent over the bus. In a fully transparent system, all messages and processes are frozen. Our approach allows the designer to specify the frozen status for individual processes and messages considering, for example, the difficulty to trace them during debugging, achieving thus a desired transparency/performance trade-off.

The conditional scheduling will handle these transparency requirements by allocating the same start time[1] for v_i in all the alternative schedules of application $\mathcal{A}$. For example, to handle

1. A frozen process P_i with a start time t_i, if affected by a fault, will be re-executed at a start time $t_i^* = t_i + C_i + \mu$.

the situation in Figure 4.1c, where P_3 and its inputs m_2 and m_3 are not affected by faults, $\mathcal{T}(m_2)$, $\mathcal{T}(m_3)$ and $\mathcal{T}(P_3)$ will have to be set to "frozen".

The shifting-based scheduling will consider only one transparency set-up, where all messages sent on the bus and only those messages are set to "frozen", e.g. $\mathcal{T}(m_1)$, $\mathcal{T}(m_2)$, and $\mathcal{T}(m_3)$ for the set-up in Figure 4.1. The same start time will be preserved for the "frozen" messages in the schedule table.

4.2 Fault-Tolerant Conditional Process Graph

The scheduling techniques presented in this chapter are based on a fault-tolerant conditional process graph (FT-CPG) representation. FT-CPG 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. Graphically, an FT-CPG is a fork-and-join structure, where each branch corresponds to a change of conditions. These branches would meet in the case of frozen processes and messages.

Definition. Formally, an FT-CPG 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 FT-CPG with P_i^m that will correspond to the m^{th} copy of process $P_i \in \mathcal{A}$. Each node $P_i^m \in V_P$ with simple edges at the output is a regular node. A node $P_i^m \in V_C$ with *conditional edges* at the output is a *conditional process* that produces a condition.

Each node $v_i \in V_T$ is a *synchronization node* and represents the synchronization point corresponding to a frozen process or message (i.e., $\mathcal{T}(v_i) = frozen$). We denote with P_i^S the synchronization node of process $P_i \in \mathcal{A}$ and with m_i^S the synchronization node of message $m_i \in \mathcal{A}$. Synchronization nodes will take zero time to execute.

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

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

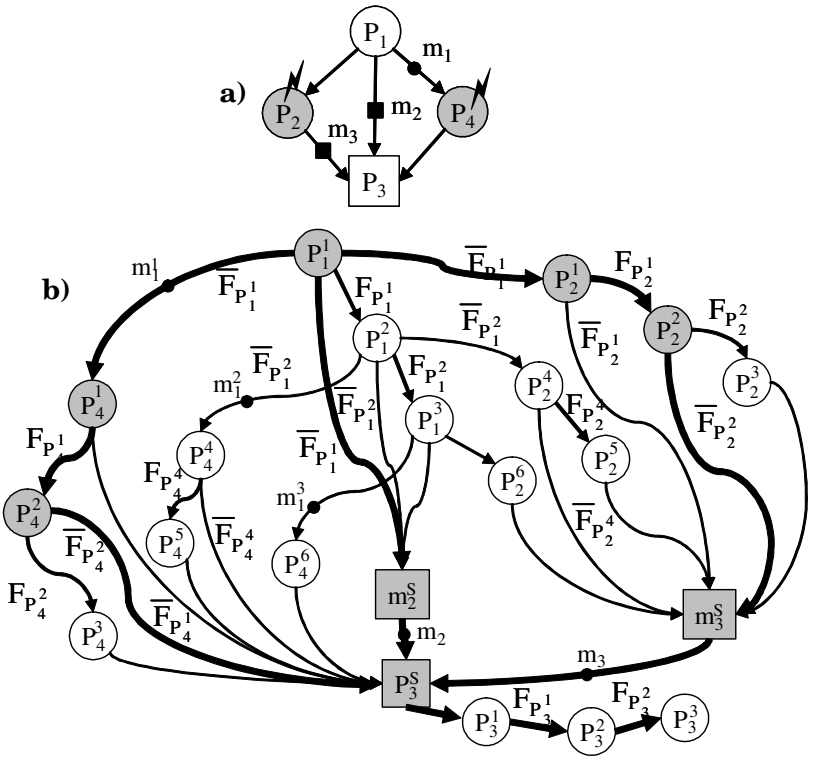


Figure 4.3: Fault-Tolerant Conditional Process Graph

Edges $e_{ij}^{mn} \in E_C$, $e_{ij}^{mS} \in E_C$, and $e_{ij}^{mS_m} \in E_C$ are *conditional edges* and have an associated condition value. The condition value produced is "true" (denoted with $F_{P_i^m}$) if P_i^m experiences a fault, and "false" (denoted with $\overline{F}_{P_i^m}$) if P_i^m does not experience a fault. Alternative paths starting from such a process, which correspond to complementary values of the condition, are disjoint[1]. Note that edges e_{ij}^{Sn}, $e_{ij}^{S_m n}$, e_{ij}^{SS}, $e_{ij}^{S_m S}$, $e_{ij}^{SS_m}$, and $e_{ij}^{S_m S_m}$ coming from a synchronization node cannot be conditional.

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

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, during the execution of processes P_2 and P_4). Transparency requirements are depicted with rectangles on the application graph, where process P_3, message m_2 and message m_3 are set to be frozen. For scheduling purposes we will convert the application $\mathcal{A}$ to a fault-tolerant conditional process graph (FT-CPG) G, represented in Figure 4.3b. In an FT-CPG the fault occurrence information is represented as *conditional edges* and the frozen processes/messages are captured using *synchronization nodes*. One of the conditional edges is P_1^1 to P_4^1 in 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 FT-CPG in Figure 4.3b captures all the fault scenarios that can happen during the execution of application $\mathcal{A}$ in 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. We will refer to every such subgraph corresponding to a particular execution scenario as an alternative path of the FT-CPG. The fault

1. They can only meet in a synchronization node.

scenario for a given process execution, for example P_4^1, the first execution of P_4, is captured by the conditional edges $F_{P_4^1}$ (fault) and $\overline{F}_{P_4^1}$ (no-fault). The transparency requirement that, for example, P_3 has to be frozen, is captured by the synchronization node P_3^S inserted before the conditional edge with copies of process P_3. In 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, like P_3^S, are synchronization nodes (depicted with a rectangle).

Regular and conditional processes are activated when all their inputs have arrived. A synchronization node can be activated after inputs coming on one of the alternative paths, corresponding to a particular fault scenario, have arrived. For example, one alternative path with faults in processes P_2 and P_4 is marked with thicker edges in Figure 4.3b. In this path, a transmission on the edge $\overline{F}_{P_2^1}$ will be enough to activate m_2^S.

A guard is associated to each node in the graph. In Figure 4.3b, an example of a guard associated to a node is, for example, $K_{P_2^2} = \overline{F}_{P_1^1} \wedge F_{P_2^1}$, which means that P_2^2 will be activated in the fault scenario where P_2 experienced a fault, while P_1 did not. A node is activated only when the value of the associated guard is true.

4.2.1 GENERATION OF FT-CPG

In Figure 4.3b we depict the FT-CPG G, which is the result of transforming the application $\mathcal{A}$ in Figure 4.3a. Each process P_i is transformed into a structure which models all possible fault occurrence scenarios with P_i, consisting of a number of conditional nodes and their corresponding conditional edges, and a set of regular nodes.

In Figure 4.4 we outline the **BuildFTCPG** algorithm that traces processes in the merged graph $\mathcal{G}$ with transparency requirements $\mathcal{T}$ in the presence of k faults, transforming them into corresponding sub-structures of the FT-CPG G. In the first

step, **BuildFTCPG** places the root process into the FT-CPG (line 2). Then, re-executions of the root process are inserted, connected through "faulty" conditional edges with the "true" condi-

```
BuildFTCPG(G, T, k)
1  G = ∅
2  P_i = RootNode(G); Insert(P_i^1, G); AssignFaults(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}); AssignFaults(P_i^{k–f+1}, f)
5  end for
6  L = ∅ -- add successors of the root node to the ready list
7  for ∀Succ(P_i) ∈ G do L = L ∪ Succ(P_i)
8  while L ≠ ∅ do -- trace all processes in the merged graph G
9    P_i = ExtractProcess(L)
10   VC = GetValidPredCombinations(P_i, G)
11   for vc_n ∈ VC do -- insert copies of process P_i
12     Insert(P_i^n,G); Connect(P_i^n, ∀P_x^m ∈ vc_n); AssignFaults(P_i^n,k–jointfaults(vc_n))
13   end for
14   for ∀m_j ∈ InputMessages(P_i) if T(m_j) do -- transform frozen messages
15     Insert(m_j^S, G); ReConnect(∀e_xi^mn (m_j), m_j^S) -- insert synchronization node
16     {P_i^n} ←MergeCopies({P_i^n}); Connect({P_i^n}, m_j^S); AssignFaults({P_i^n}, k)
17   end for
18   if T(P_i) then -- if process P_i is frozen, then insert corresponding synchronization node
19     Insert(P_i^S, G); ReConnect(∀e_xi^mn, P_i^S)
20     P_i^1 ← MergeAllCopies({P_i^n}); Connect(P_i^1, P_i^S); AssignFaults(P_i^1, k)
21   end if
22   for ∀P_i^n do -- add re-executions of copies P_i^n of process P_i
23     for f = faults(P_i^n)–1 downto 0 do
24       Insert(P_i^{n(f)}, G); Connect(P_i^{n(f)}, P_i^{n(f–1)}); AssignFaults(P_i^{n(f)}, f)
25     end for
26   end for
27   ReLabel({P_i^n}∪{P_i^{n(f)}}) -- re-label copies P_i^n of process P_i and their re-executions
28   for ∀Succ(P_i) ∈ G do -- add successors of process P_i to the ready list
29     if ∀Succ(P_i) ∉ L then L = L ∪ Succ(P_i)
30   end for
31 end while
32 return G
end BuildFTCPG
```

Figure 4.4: Generation of FT-CPG

tion value (lines 3-5). The copy of the root process and its re-executions are assigned with k, $k-1$, $k-2$, ..., 0 faults, respectively, which will be used in the later steps. In Figure 4.5, we show an intermediate state of generation of the FT-CPG depicted in Figure 4.3b. The result after the first step is highlighted with a dashed line. After the first step, processes P_1^1, P_1^2 and P_1^3 are inserted, connected with two conditional edges e_{11}^{12} and e_{11}^{23}.

In the next step, **BuildFTCPG** places successors of the root process into the ready list $\mathcal{L}$ (line 7). For generation of the FT-CPG, the order of processes in the ready list $\mathcal{L}$ is not important and **BuildFTCPG** extracts the first available process P_i for transformation (line 9).

For each process P_i, extracted from the ready list $\mathcal{L}$, **BuildFTCPG** prepares a set of *valid* combinations $\mathcal{VC}$ of predecessor processes (line 10). The combinations are valid if the predecessor processes in each combination $vc_n \in \mathcal{VC}$ correspond to a common set of condition values and, all together, do not experience more than k faults. For example, in Figure 4.5, the combination of predecessors of process P_3 consisting of processes P_2^1 and P_4^1 is valid, while combination of P_2^1 and P_4^4 is not. P_2^1 and P_4^1 belong to a fault scenario (a set of conditions), where process P_1 does not experience faults. However, P_2^1 and P_4^4 belong to two contradictory fault scenarios: (1) *no* faults happen in P_1 and (2) a fault, marked as $F_{P_1^1}$, happens in the first execution of P_1.[1] Similarly, the combination consisting of P_2^2 and P_4^2 is valid but the combination of P_2^4 and P_4^6 is not. However, for example, the combination of P_2^2 and P_4^3 is also not valid, even though processes P_2^2 and P_4^3 belong to a common set of conditions, e.g. fault occurrences do not contradict with each other. This combination is not

1. In Figure 4.5, "faulty" edges with "true" conditions $F_{P_i^n}$ are thick, while the "non-faulty" edges are thin. For the actual names of "false" conditions $\bar{F}_{P_i^n}$ associated to "non-faulty" edges, look into Figure 4.3b.

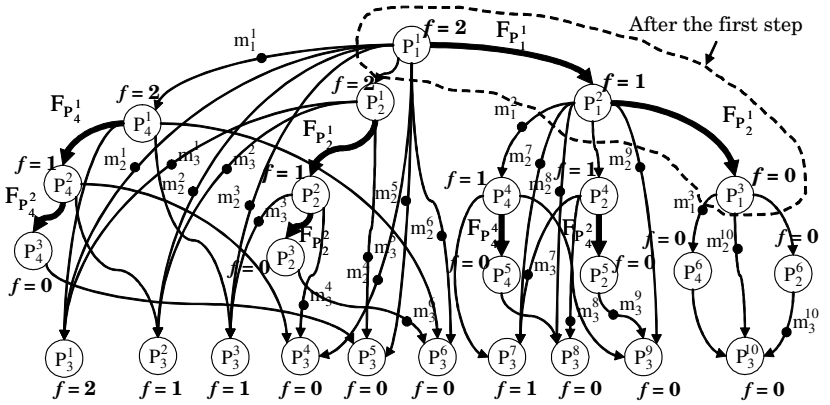


Figure 4.5: FT-CPG Generation without Synchronization Nodes

valid because P_2^2 and P_4^3, taken together, experience 3 faults, which is more than $k = 2$.

For each valid combination $vc_n \in \mathcal{VC}$, **BuildFTCPG** inserts a copy P_i^n of process P_i and assign it to all the predecessors in vc_n with corresponding conditional and unconditional edges (lines 11-13). The conditional edges are "non-faulty" with the "false" conditional value. These conditional and unconditional edges can carry copies of input messages to process P_i. For example, in Figure 4.5, conditional edge e_{14}^{11}, connecting P_1^1 and P_4^1, carries a copy of message m_1.

Each copy P_i^n can experience at most $f = k - joint\_faults(vc_n)$ faults (see **AssignFaults** function, line 12). For example, the copy P_3^1 of process P_3, corresponding to the combination of P_2^1 and P_4^1, can experience $f = 2 - 0 = 2$ faults, but the copy P_3^4 of P_3, corresponding to the combination of P_2^2 and P_4^2, will not experience faults because $f = 2 - 2 = 0$.

If any of the input messages m_j to process P_i is frozen, then a synchronization node m_j^S is introduces (lines 14-17). Edges containing the message m_j are re-connected to m_j^S (both conditional and unconditional); message m_j is also removed from these

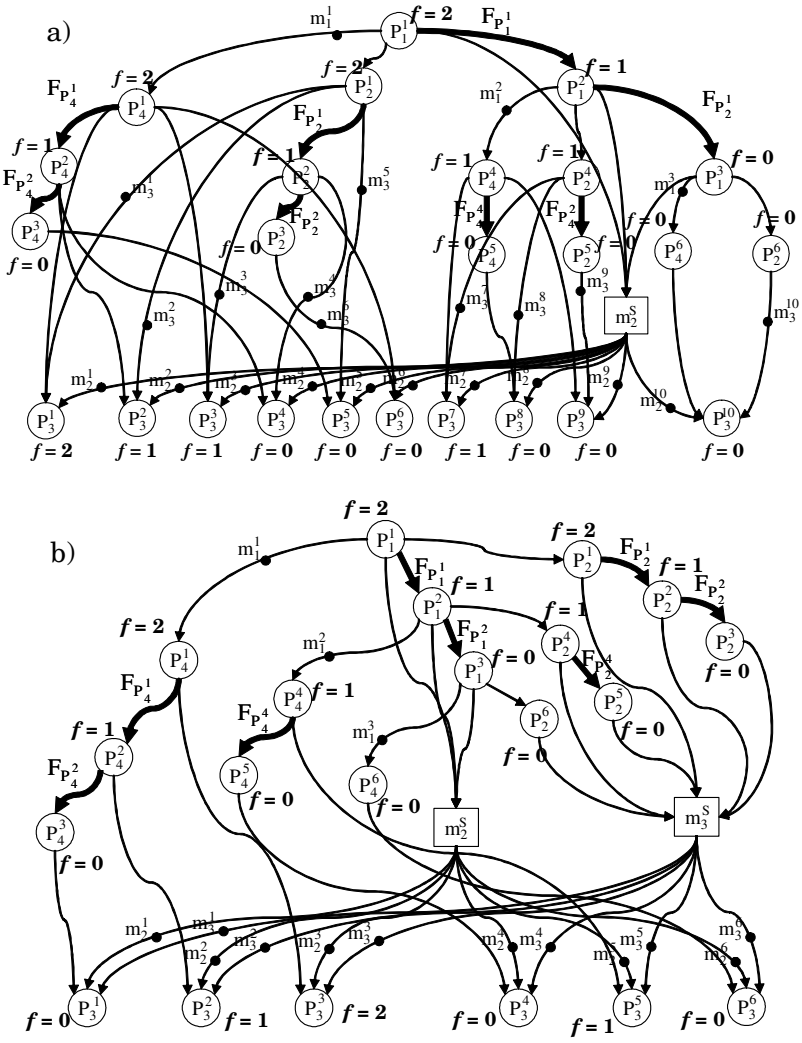


Figure 4.6: Inserting Synchronization Nodes

edges (see **ReConnect** function, line 15). Copies P_i^n of process P_i, that differ only in the conditions passed through those edges (with m_j), are merged (**MergeCopies** function, line 16). The remaining copies P_i^n of P_i are connected to m_j^S with uncondi-

tional edges carrying message m_j. **BuildFTCPG** performs this operation for every frozen input message. In Figure 4.6, we show how frozen messages m_2 and m_3 are introduced for the application $\mathcal{A}$ depicted in Figure 4.3a. First, for message m_2, synchronization node m_2^S is inserted into the structure depicted in Figure 4.5, as shown in Figure 4.6a, and copies of process P_3 are re-connected to this synchronization node. Note that none of the copies is merged in this case because the conditions associated to all of them are different. Second, for message m_3, we introduce synchronization node m_3^S, as illustrated in Figure 4.6b. In this case, 10 copies of process P_3, in Figure 4.6a, are merged into 6 copies, which are, then, connected to the synchronization node m_3^S.

If process P_i is frozen, then synchronization node P_i^S is introduced (lines 18-21). All edges that connect copies P_i^n of P_i to combinations of valid predecessors $vc_n \in \mathcal{VC}$ and synchronization nodes m_j^S are re-connected to P_i^S (**ReConnect** function, line 19). All copies P_i^n of P_i are merged into a single copy P_i^1, which is connected to the synchronization node P_i^S through unconditional edge (**MergeAllCopies** function, line 19). The maximum number of faults $f = k$ is associated to P_i^1. For example, the application $\mathcal{A}$ in Figure 4.3a has frozen process P_3. Once this process is reached by the FT-CPG generation algorithm, after introducing frozen message m_3, synchronization node P_3^S is created. Then, all copies of process P_3, in Figure 4.6b, are merged into single copy P_3^1 connected to P_3^S, as shown in the FT-CPG in Figure 4.3b.

After copies of process P_i are introduced and, if needed, merged, **BuildFTCPG** adds re-executions of P_i's copies against possible fault occurrences (lines 22-26). The number of faults f has been associated to each copy P_i^n of process P_i in the previous steps. To tolerate these faults, f re-executions are inserted, connected through "faulty" conditional edges. Note that the first re-execution $P_i^{n(f-1)}$ is connected to the copy P_i^n of process P_i, while every other re-execution is connected to the previous process re-

execution, e.g. the second re-execution is connected to the first, the third to the second, and so on. The number of faults is accordingly associated to the re-executions, e.g. first re-execution $P_i^{n(f-1)}$ is associated with $f-1$ faults, second re-execution $P_i^{n(f-2)}$ with $f-2$ faults, and the last re-execution $P_i^{n(0)}$ with 0 faults. The numbers of faults associated to re-executions will be later used for transforming successors of process P_i.

After re-labelling (line 27), re-executions of copy P_i^n will have indices following the index of copy P_i^n, e.g. P_i^{n+1}, P_i^{n+2}, and so on. The original copies P_i^n are re-labelled accordingly. In Figure 4.5, for example, the copy P_4^1 of process P_4 can be re-executed two times with re-executions labelled as P_4^2 and P_4^3; the copy P_2^4 can be re-executed once with re-execution P_2^5; the copy P_4^6 of process P_4 is not re-executed.

When process P_i has been finally transformed, its successors are placed into the ready list $\mathcal{L}$ (lines 28-30). **BuildFTCPG** continues until all processes in the merged graph $\mathcal{G}$ are transformed into corresponding sub-structures of the FT-CPG G.

4.3 Conditional Scheduling

The problem that we are addressing with conditional 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, and a set of transparency requirements on the application $\mathcal{T}(A)$, we are interested to determine the schedule table $\mathcal{S}$ such that the worst-case end-to-end delay δ_G, by which the application completes execution is minimized, and the transparency requirements captured by $\mathcal{T}$ are satisfied. If the resulting delay is smaller than the deadline, the system is schedulable.

4.3.1 SCHEDULE TABLE

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

Only one part of the table has to be stored in each node, namely, the part concerning decisions that are taken by the corresponding scheduler. Figure 4.7 presents the schedules for the nodes N_1 and N_2 produced by the conditional scheduling algorithm in Figure 4.8 for the FT-CPG in Figure 4.3. In each table there is one row for each process and message from application $\mathcal{A}$. A row contains activation times corresponding to different values of conditions. In addition, there is one row for each condition whose value has to be broadcasted to other computation nodes. Each column in the table is headed by a logical expression constructed as a conjunction of condition values. Activation times in a given column represent starting times of the processes and transmission of messages when the respective expression is true.

According to the schedule for node N_1 in Figure 4.7, process P_1 is activated unconditionally at the time 0, given in the first column of the table. Activation of the rest of the processes, in a certain execution cycle, depends on the values of the conditions, i.e., the unpredictable occurrence of faults during the execution of certain processes. For example, process P_2 has to be activated at $t = 30$ if $\overline{F}_{P_1^1}$ is true, at $t = 100$ if $F_{P_1^1} \wedge F_{P_1^2}$ is true, etc.

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

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

Figure 4.7: Conditional Schedule Tables

At a certain moment during the execution, when the values of some conditions are already known, they have to be used to take the best possible decisions on process activations. Therefore, after the termination of a process that produces a condition, the value of the condition is broadcasted from the corresponding computation node to all other computation nodes. This broadcast is scheduled as soon as possible on the communication channel, and is considered together with the scheduling of the messages. The scheduler in a node knows from its schedule table when to expect a condition message.

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

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

4.3.2 CONDITIONAL SCHEDULING ALGORITHM

According to our application model, some processes can only be activated if certain conditions (i.e., fault occurrences), produced by previously executed processes, are fulfilled. 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 on which process to activate and at which time has to be taken without knowing which values the conditions will later get. On the other side, at a certain moment during execution, when the values of some conditions are already

known, they have to be used in order to take the best possible decisions on when and which process to activate, in order to reduce the schedule length.

Optimal scheduling has been proven to be an NP-complete problem [Ull75] in even simpler contexts. Hence, heuristic algorithms have to be developed to produce a schedule of the processes such that the worst case delay is as small as possible. Our strategy for the synthesis of fault-tolerant schedules is presented in Figure 4.8. The **FTScheduleSynthesis** function takes as input the application $\mathcal{A}$ with the transparency requirements $\mathcal{T}$, the number k of transient faults that have to be tolerated, the architecture consisting of computation nodes $\mathcal{N}$ and bus B, the mapping $\mathcal{M}$, and produces the schedule table $\mathcal{S}$.

Our synthesis approach employs a *list scheduling* based heuristic, **FTCPGScheduling**, presented in Figure 4.10, for scheduling each alternative fault-scenario. However, the fault scenarios cannot be independently scheduled: the derived schedule table has to fulfil the requirements (1) to (3) presented in Section 4.3.1, and the synchronization nodes have to be scheduled at the same start time in all alternative schedules.

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

Figure 4.8: Fault-Tolerant Schedule Synthesis Strategy

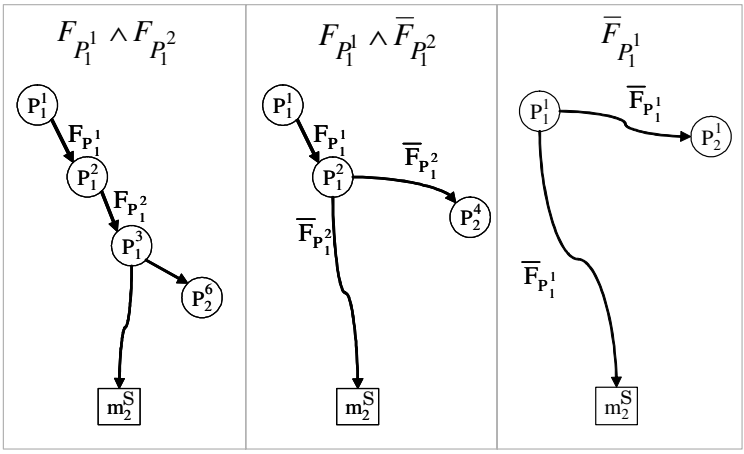


Figure 4.9: Alternative paths investigated by FTCPGScheduling for the synchronization node m_2^S

In the first line of the **FTScheduleSynthesis** algorithm, we initialize the schedule table $\mathcal{S}$ and build the FT-CPG G as presented in Section 4.2.[1] List scheduling heuristics use priority lists from which ready processes are extracted in order to be scheduled at certain moments. A process is ready if all its predecessors have been scheduled. We use the *partial critical path* (PCP) priority function [Ele00] in order to assign priorities to processes (line 3).

A synchronization node S_i, in order to mask fault occurrences, must have the same start time t_i in the schedule $\mathcal{S}$, regardless of the guard K_{S_i} under which it is scheduled. For example, the synchronization node m_2^S in Figure 4.9 has the same start time of 105, in each corresponding column of the table in Figure 4.7. In order to determine the start time t_i of a synchronization node

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

$S_i \in \mathcal{L}_S$, where $\mathcal{L}_S$ is the list of synchronization nodes, we will have to investigate all the alternative fault-scenarios (modelled as different alternative paths through the FT-CPG) that lead to S_i. Figure 4.9 depicts the three alternative paths that lead to m_2^S for the graph in Figure 4.3b. These paths are generated using the **FTCPGScheduling** function (called in line 6, Figure 4.8), which records the maximum start time t_{max} of S_i over the start times

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

Figure 4.10: Conditional Scheduling

in all the alternative paths. In addition, **FTCPGScheduling** also records the guards $\mathcal{K}_{S_i}$ under which S_i has to be scheduled. The synchronization node S_i is then inserted into the schedule table in the columns corresponding to the guards in the set $\mathcal{K}_{S_i}$ at the unique time t_{max} (line 10 in Figure 4.10). For example, m_2^S is inserted at time $t_{max} = 105$ in the columns corresponding to $\mathcal{K}_{m_2} = \{\overline{F}_{P_1^1}, F_{P_1^1} \wedge F_{P_1^2}, F_{P_1^1} \wedge \overline{F}_{P_1^2}\}$.

The **FTCPGScheduling** function is recursive and calls itself for each conditional node in order to separately schedule the nodes in the *faulty* branch, and those in the *true* branch (lines 21 and 23, Figure 4.10). Thus, the alternative paths are not activated simultaneously and resource sharing is correctly achieved. During the exploration of the FT-CPG it is important to eliminate alternative paths that are not possible to occur. This requirement is handled by introducing the parameter f, which represents the number of faults that still can occur. f is decremented for each call of **FTCPGScheduling** that explores a faulty (true) branch. Thus, only if $f > 0$ (line 19), we will continue to investigate branches through recursions.

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

modate P_i. For example, m_2 is scheduled first at 105 on the bus, thus time 0–105 is idle time on the bus. We will later schedule m_1 at times 66, 31 and 100, within this idle segment. The scheduling of P_i will be done under the currently known conditions K, determined at line 6 on the resource R. Our approach eliminates from K those conditions that, although known to R at time t, will not influence the execution of P_i.

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 are addressing 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 $\mathcal{S}$ 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 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, e.g. *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 FT-CPG representation, on top of which an additional ordering is introduced.

Let us illustrate the ordering with an example in Figure 4.11a 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 computation node N_2. Message m_1 is frozen since it is transmitted through the bus. In this example, we introduce the following order 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 FT-CPG with introduced order relations, when $k = 2$ transient faults can happen at maximum, is presented in Figure 4.11b. The introduced precedence constraints order the execution of processes assigned to the computation node. This order can be obtained using a priority function, for example, the partial critical path (PCP) [Ele00].

For the shifting-based scheduling, the FT-CPG is not explicitly generated. Instead, only a root schedule is obtained off-line. The root schedule consists of start times of processes in the non-

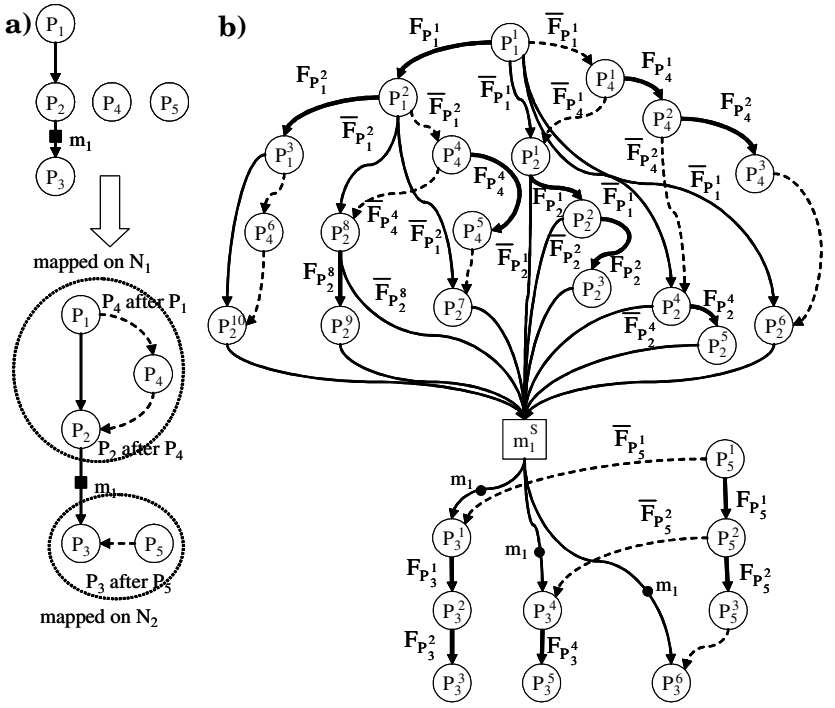


Figure 4.11: Ordered FT-CPG

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 on-line scheduler for extracting the execution scenario corresponding to a particular fault occurrence (which corresponds to a branch of the ordered FT-CPG). 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 4.12 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 $\mathcal{RS}$.

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

Figure 4.12: Generation of Root Schedules

At first, the order of process execution is introduced with the partial critical path (PCP) priority function [Ele00] (line 2). Initial recovery slacks for all processes $P_i \in \mathcal{A}$ are calculated as $s_0(P_i)=k\times (C_i + \mu)$ (lines 3-5). Then, recovery slacks of processes mapped on the same computation node are merged to reduce timing overhead.

The process graph $\mathcal{A}$ is traversed starting from the root node (line 7). Process p is selected from the ready list $\mathcal{L}$ according to the priority function (line 9). The last scheduled process r on the computation node, on which p is mapped, is extracted from the root schedule $\mathcal{S}$ (line 11). Process p is scheduled and its start time

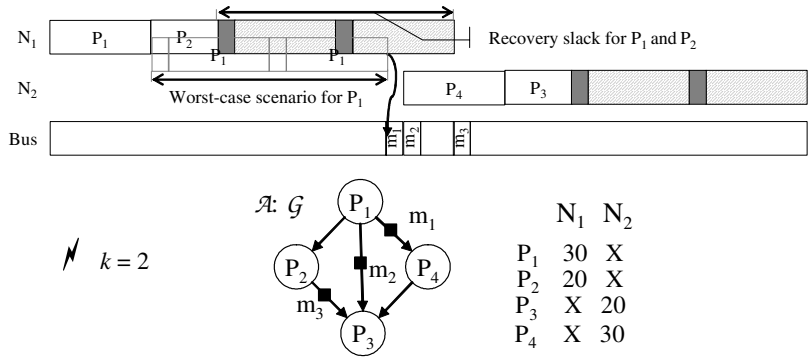


Figure 4.13: Example of a Root Schedule

is recorded in the root schedule. 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 13-14). The adjustment is performed in two steps:

1. The idle time b between process p and the last scheduled process r is calculated (line 13).
2. The recovery slack $s(p)$ of process p is changed, if recovery slack $s(r)$ of process r substracted with the idle time b is larger than the initial slack $s_0(p)$. Otherwise, the initial slack $s_0(p)$ is preserved as $s(p)$ (line 14).

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 recovery slack $s(p)$ (line 16).

After the adjustment of the recovery slack, process p is removed from the ready list $\mathcal{L}$ (line 17) and its successors are added to the list (lines 19-21). After scheduling of all the processes in the application graph $\mathcal{A}$, the algorithm returns a root schedule $\mathcal{RS}$ with start times of processes, sending times of messages, and recovery slacks (line 23).

In Figure 4.13 we present an example of the generated 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 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 95 and 125 ms and share a recovery slack of 70 ms. Messages m_1, m_2 and m_3 are sent at 90, 95, and 110 ms, respectively, at the end of the worst-case recovery intervals of sender processes.

Extracting Execution Scenarios. In Figure 4.14, we show an example, where we extract one execution scenario from the root schedule of the application $\mathcal{A}$, depicted in Figure 4.13. In this execution scenario, process P_4 experiences two faults. P_4 starts at 95 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 95+30+5 = 130 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 a fault and the start time of P_4's second re-execution P_4^2 is 130+30+5= 165 ms. Process P_3 will be delayed because of the re-executions of process P_4. The current time t_c of the scheduler, at the moment when P_2 is activated, is 165+30 = 195 ms, which is more than 125 ms that

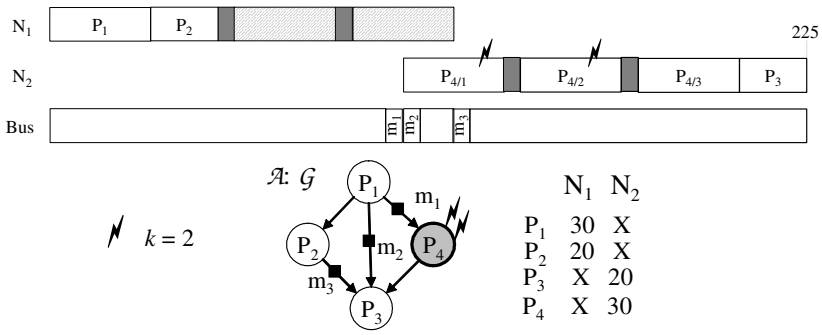


Figure 4.14: Example of Execution Scenario

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

Figure 4.15: Extracting Execution Scenarios

is the schedule time of P_2 according to the root schedule. Therefore, process P_2 will be executed at $t_c = 195$ ms. The application $\mathcal{A}$ will complete execution at 225 ms.

The run-time algorithm for extracting execution scenarios from the root schedule $\mathcal{RS}$ is presented in Figure 4.15. The real-time 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 t_c 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 at its start time in the root schedule (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 messages 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 against 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 4.12). 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 4.15).

4.5 Experimental Results

For the evaluation of our algorithms we used applications of 20, 40, 60, and 80 processes mapped on architectures consisting of 4 nodes. We have varied the number of faults, considering 1, 2, and 3 faults, which can happen during one execution cycle. The duration μ of the recovery time has been set to 5 ms. Fifteen examples were randomly generated for each application dimension, thus a total of 60 applications were used for experimental evaluation. Execution times and message lengths were assigned randomly within the 10 to 100 ms, and 1 to 4 bytes ranges, respectively. The experiments were done on Sun Fire V250 computers.

We were first interested to evaluate how well the 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 4.8. 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 $100 \times (\delta_{CS} - \delta_{NFT}) / \delta_{NFT}$.

For the experiments, we considered that the designer is interested to maximize the amount of transparency for the inter-processor messages, which are critical to a distributed fault-tolerant system. Thus, we have considered five transparency scenarios, depending on how many of the inter-processor messages have been set as frozen: 0, 25, 50, 75 or 100%. Table 4.1 presents the average fault-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. For example, for application graphs of 60 processes with three faults, we have obtained an 86% overhead for 100% frozen messages, which was reduced to 58% for 50% frozen messages.

Table 4.2 presents the average memory[1] space per computation node (in kilobytes) required by the schedule tables. Often, one entity has the same start time under different conditions. Such entries into the table can be merged into a single table entry, headed by the union of the logical expressions. Thus, Table 4.2 reports the memory required after such a straightforward compression. We can observe that as the transparency increases, the memory requirements decrease. For example, for 60 processes and three faults, increasing the number of frozen

Table 4.1: Fault-Tolerance Overheads (CS), %

	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

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

Table 4.2: Memory Requirements (CS), 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%	0.13	0.28	0.54	0.36	0.89	1.73	0.71	2.09	4.35	1.18	4.21	8.75
75%	0.22	0.57	1.37	0.62	2.06	4.96	1.20	4.64	11.55	2.01	8.40	21.11
50%	0.28	0.82	1.94	0.82	3.11	8.09	1.53	7.09	18.28	2.59	12.21	34.46
25%	0.34	1.17	2.95	1.03	4.34	12.56	1.92	10.00	28.31	3.05	17.30	51.30
0%	0.39	1.42	3.74	1.17	5.61	16.72	2.16	11.72	34.62	3.41	19.28	61.85

messages from 50% to 100% reduces the memory needed from 18K to 4K.

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 exponentially with the number of faults that have to be handled. However, even for graphs of 60 processes, for example, and three faults, the schedule synthesis algorithm finishes in under 10 minutes.

Shifting-based scheduling, discussed in Section 4.4, can only handle a setup with 100% transparency for inter-processor messages. As a second set of experiments, we have compared the conditional scheduling approach with the shifting-based scheduling approach, namely SBS, considering this 100% scenario. In order to compare the two algorithms, we have produced the end-to-end delay δ_{SBS} of the application when using SBS. When comparing the delay δ_{CS}, obtained with conditional scheduling, to δ_{SBS} in the case of $k = 2$, for example, 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 quota of a second and can pro-

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%	0.016			0.034			0.054			0.070		

duce root schedules for large graphs of 80, 100, and 120 processes with 4, 6, and 8 faults in that time. The schedule generation time does not exceed 0.2 sec. even for 120 processes and 8 faults. Therefore, shifting-based scheduling can be effectively used in design optimization strategies, 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 run even on small microcontrollers.

Finally, we considered a real-life example implementing a vehicle cruise controller (CC). The process graph that models the CC has 32 processes, and is described in [Pop03]. The CC was mapped on an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Breaking System (ABS) and Transmission Control Module (TCM). We have considered a deadline of 300 ms, $k = 2$ and $\mu = 2$ ms.

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

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 schedule tables.

The second scheduling approach handles only a fixed transparency set-up, transparent recovery, where all messages on the bus have to be sent at fixed times, regardless of fault occurrences. However, this scheduling approach is much faster than 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
Process Mapping and Fault Tolerance Policy Assignment

In this chapter we discuss two optimization problems regarding the design of fault-tolerant embedded systems: mapping with fault tolerance policy assignment and mapping with performance/transparency trade-offs.

For optimization of policy assignment, presented in the first part of this chapter, we combine re-execution that provides time redundancy with replication that 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.

In the second part of this chapter we present an approach for mapping optimization of fault-tolerant embedded systems with performance/transparency trade-offs. In order to speed-up the optimization process, we propose a schedule length estimation

heuristic that evaluates design solutions without the need to generate the complete schedule tables, thus reducing the runtime.

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 fault tolerance 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 active 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

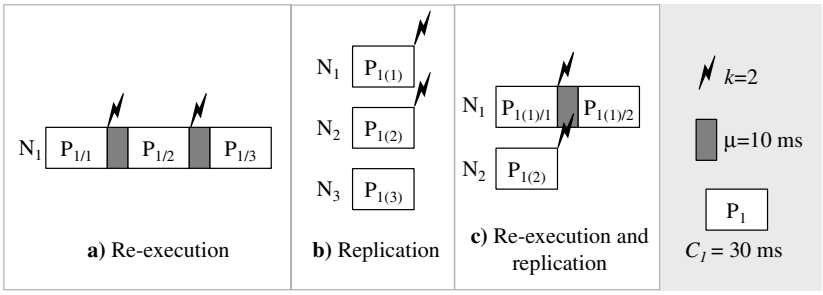


Figure 5.1: Policy Assignment: Re-execution + Replication

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]

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 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 is as follows:

- As an input we have an application $\mathcal{A}$ given as a set of process graphs (Section 3.1) and a system consisting of a set of nodes $\mathcal{N}$ connected to a bus B.
- The parameter k denotes the total number of transient faults that can appear in the system during one cycle of execution.

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

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

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

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.

fications, which will be discussed in Section 5.1.4, 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. The fault model assumes 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.

In Figure 5.2 we depict the schedules[1] for each node. Comparing the schedules in Figure 5.2a_1 and Figure 5.2b_1, we can observe that using (a_1) active replication the deadline is missed.

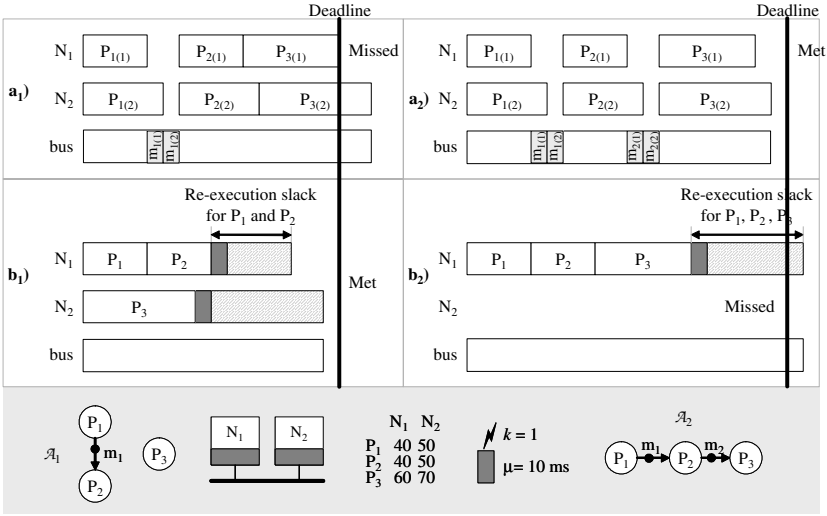


Figure 5.2: Comparison of Replication and Re-execution

1. The schedules depicted are optimal.

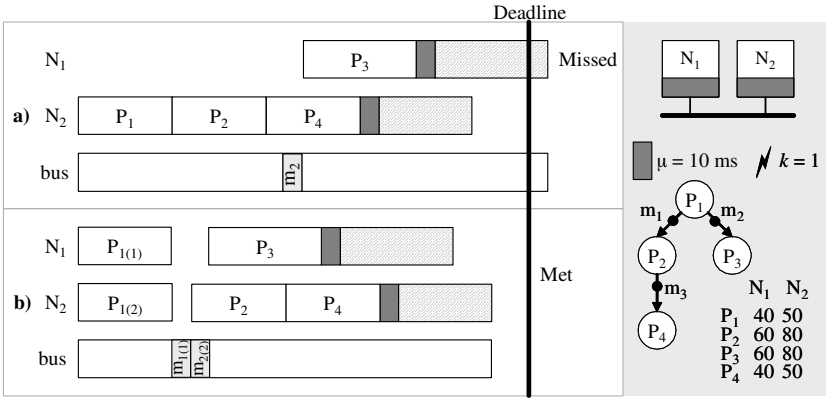


Figure 5.3: Combining Re-execution and Replication

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 re-execution slacks, have to finish before the deadline. Using (b_1) re-execution we are able to meet the deadline. 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 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. Moreover, the best result is most likely to be obtained when both techniques 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. However, combining re-execution with replication, as in Figure 5.3b where process P_1 is replicated, will meet the deadline. In this case, message m_2 does not have to be delayed to mask the failure of process P_1. Instead, 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.

5.1.2 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 algo-

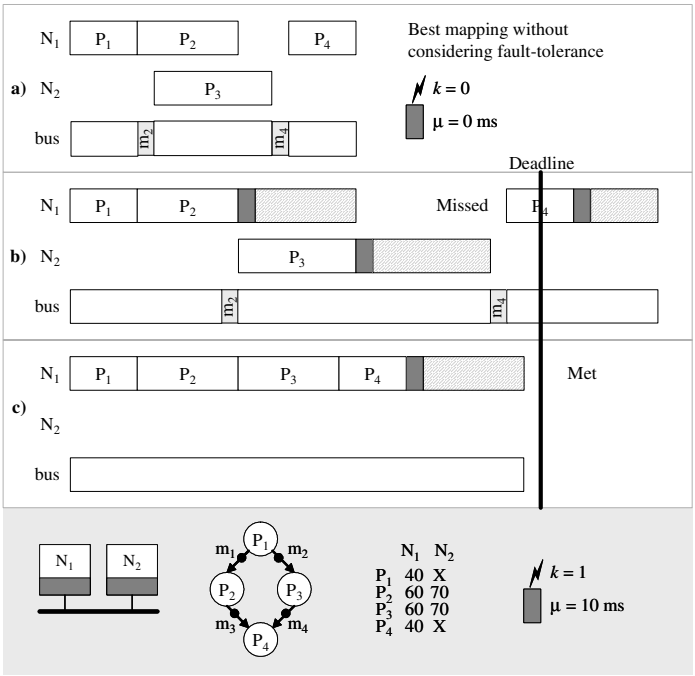


Figure 5.4: Mapping and Fault Tolerance

rithm 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.

The 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 order to reduce the re-execution slack and the delays due to the masking of faults. 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.

5.1.3 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 three 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 assignment

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

Figure 5.5: Design Optimization Strategy for Fault Tolerance Policy 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. The second step consists of a greedy heuristic GreedyMPA (line 3), discussed in Section 5.1.4, that aims to improve the fault-tolerance policy assignment and mapping obtained in the first step.
3. If the application is still not schedulable, we use, in the third step, a tabu search-based algorithm TabuSearchMPA presented in Section 5.1.4.

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

5.1.4 SCHEDULING AND REPLICATION

In Section 4.4, we presented the shifting-based scheduling algorithm for re-execution. 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).

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

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_2 can arrive. Therefore, P_3 has to be placed in the schedule, as illustrated in Figure 5.6a, after message $m_{2(2)}$ for replica $P_{2(2)}$ has arrived. Of course, we should also introduce recovery slack for process P_3 if 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 replicated processes. In this root schedule, process P_3 will start immediately after replica $P_{2(1)}$ in the case that no fault occurred in $P_{2(1)}$. If replica $P_{2(1)}$ fails, then, in the corresponding contingency 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

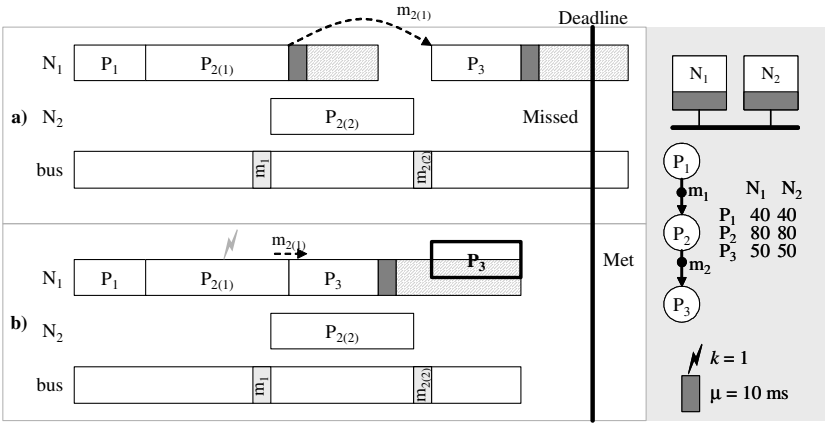


Figure 5.6: Scheduling Replica Descendants

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.

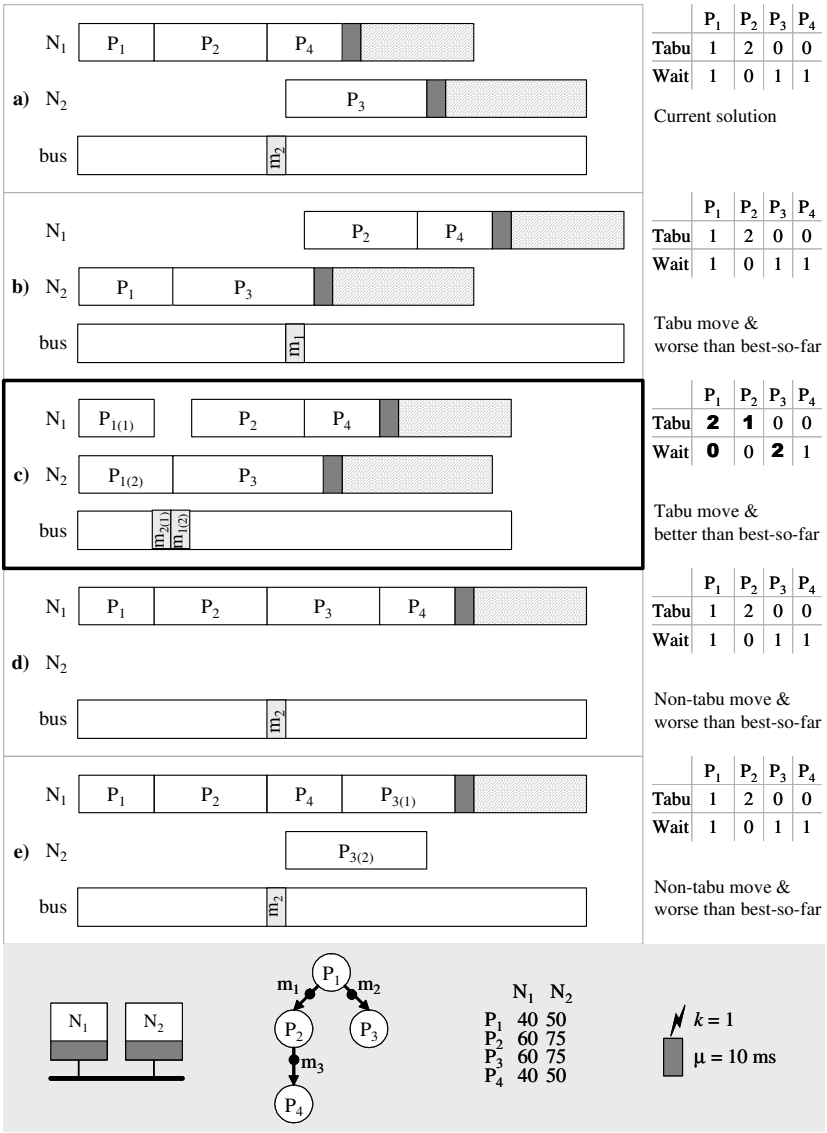
5.1.5 OPTIMIZATION ALGORITHMS

For the optimization of the mapping and fault-policy assignment we perform two steps, see Figure 5.5. One is based on a greedy heuristic, GreedyMPA. If this step fails, we use, in the next step, a tabu search-based approach, TabuSearchMPA.

Both approaches investigate in each iteration all the processes on the critical path of the merged application graph $\mathcal{G}$ (see Section 3.1), 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 neighbour solutions generated starting from Figure 5.7a, are the solutions presented in Figure 5.7b–5.7e. Out of these, the solution in Figure 5.7c is the best in terms of schedule length.

The greedy approach selects in each iteration the best move found and applies it to modify the design. The disadvantage of the greedy approach is that it can "get stuck" into a local opti-

mum. To avoid this, we have implemented a tabu search algorithm, presented in Figure 5.8.



	P1	P2	P3	P4
Tabu	1	2	0	0
Wait	1	0	1	1

	P1	P2	P3	P4
Tabu	1	2	0	0
Wait	1	0	1	1

	P1	P2	P3	P4
Tabu	**2**	**1**	0	0
Wait	**0**	0	**2**	1

	P1	P2	P3	P4
Tabu	1	2	0	0
Wait	1	0	1	1

	P1	P2	P3	P4
Tabu	1	2	0	0
Wait	1	0	1	1

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 a solution based on aspiration criteria
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

The tabu search 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. We define the critical path as the path through the merged graph $\mathcal{G}$ which corresponds to the longest delay in the schedule table. 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 encountered during the search. The selective history is implemented in our case through the use of two tables, *Tabu* and *Wait*. Each process has an entry in this tables. If $Tabu(P_i)$ is non-zero, it means that the process is "tabu", i.e., should not be selected for generating moves, while 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. Thus (lines 9 and 10 of the algorithm) a move will be removed from the neighbourhood solutions if it is tabu. However, tabu moves are also accepted if they are better than the best-so-far solution (line 10). In line 12 the search is diversified with moves which 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} is 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.

5.1.6 EXPERIMENTAL RESULTS

For the evaluation of our algorithms we 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. A time-division multiple access (TDMA) bus with the time-triggered protocol (TTP) [Kop03] has been used as a communication media. 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 were randomly generated for each application dimension, thus a total of 75 applications were used for experimental evaluation. We generated both graphs with random structure and graphs based on more regular structures like trees and groups of chains. Execution times and message lengths were assigned randomly using both uniform and exponential distribution within the 10 to 100 ms, and 1 to 4 bytes ranges, respectively. The experiments were 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

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

Number of processes	*k*	% maximum	% average	% minimum
20	3	98.36	70.67	48.87
40	4	116.77	84.78	47.30
60	5	142.63	99.59	51.90
80	6	177.95	120.55	90.70
100	7	215.83	149.47	100.37

have derived a reference non-fault tolerant implementation, NFT, which ignores the fault tolerance issues. The NFT implementation is produced as 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 5.2: 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	52.44	32.72	19.52
4	110.22	76.81	46.67
6	162.09	118.58	81.69
8	250.55	174.07	117.84
10	292.11	219.79	154.93

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, 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-

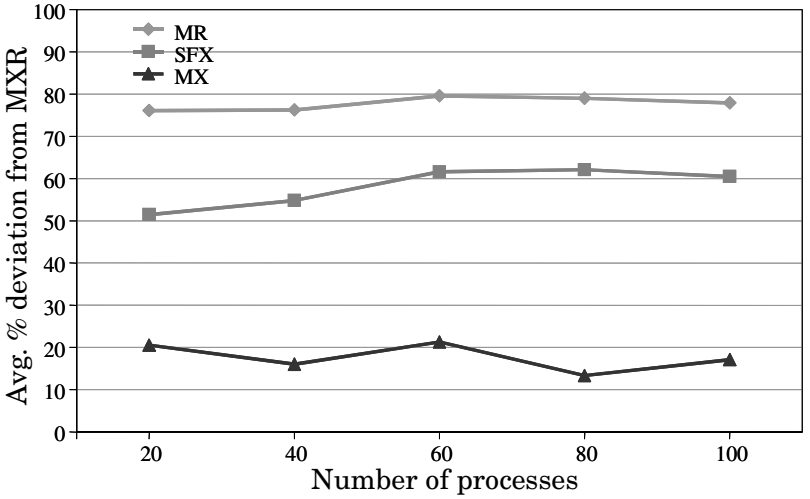


Figure 5.9: Comparing MXR with MX, MR and SFX

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 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 a real-life example implementing a vehicle cruise controller (CC), which was previously used to evaluate scheduling techniques in Chapter 4. We have consid-

ered 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.

5.2 Mapping Optimization with Transparency

In this part of the chapter, we discuss mapping optimization with the possibility of trading-off transparency for performance. As discussed in Section 4.1, increased transparency makes a system easier to debug. The amount of memory required to store contingency schedules is also reduced with increasing transparency. However, as a drawback, transparency increases the worst-case delays of processes.

The designer specifies the desired degree of transparency by customizing transparency properties with declaring certain processes and messages as *frozen*. As discussed earlier, a frozen process or message has a fixed start time regardless of the occurrence of faults in the rest of application. These customized transparency properties have to be taken into account during design optimization, particularly mapping optimization, because they introduce delays that can violate timing constrains of the application.

Thus, in this part of the chapter, we are interested to find a mapping of processes such that the delays introduced due to the frozen processes and messages are reduced. The design strategies proposed will be based on conditional scheduling, presented in Section 4.3, that can accommodate customized transparency properties into fault-tolerant schedules. In this section we assume that fault tolerance is achieved by re-execution.

5.2.1 MOTIVATIONAL EXAMPLES

To illustrate the issues related to mapping with transparency, we will discuss two examples, one with a frozen process, presented in Figure 5.10 and one with frozen messages, presented in Figure 5.11. As before, we depict frozen processes and messages using squares.

In Figure 5.10 we consider an application consisting of six processes, P_1 to P_6 that have to be mapped on an architecture consisting of two computation nodes connected to a bus. We assume that there can be at most $k = 2$ faults during one cycle of operation. The worst-case execution times for each process on each computation node are depicted in the table next to the architecture. Furthermore, let us assume that process P_2 is frozen. We impose a deadline of 310 ms for the application (a thick line crossing the figure). If we decide the mapping without considering the transparency requirement on P_2, we obtain the optimal mapping depicted in Figure 5.10a (processes P_2, P_4 and P_5 are mapped on node N_1; while P_1, P_3 and P_6 on node N_2). If transparency is ignored, the application is schedulable in all possible fault scenarios; it meets the deadline even in case of the worst-case fault scenario shown in Figure 5.10b.

If the same mapping determined in Figure 5.10a is used in the case of transparency, the obtained solution is depicted in Figure 5.10c, where the start time of P_2, which should be frozen, is delayed with respect to the worst-case fault scenario of P_4. However, in this case, the deadline will not be met due to the delay introduced by the frozen process P_2. A mapping that makes the system schedulable even in the worst-case fault scenario and with a frozen P_2 is shown in Figure 5.10d. According to this mapping, processes P_1, P_2 and P_5 are mapped on node N_1, while processes P_3, P_4 and P_6 are mapped on node N_2. Counterintuitively, this mapping is not balanced and communications are increased compared to the previous mapping, since we send message m_2, which is two times larger than message m_1. How-

ever, it gives a better solution that will meet the deadline even in the worst-case fault scenario depicted in Figure 5.10d.[1]

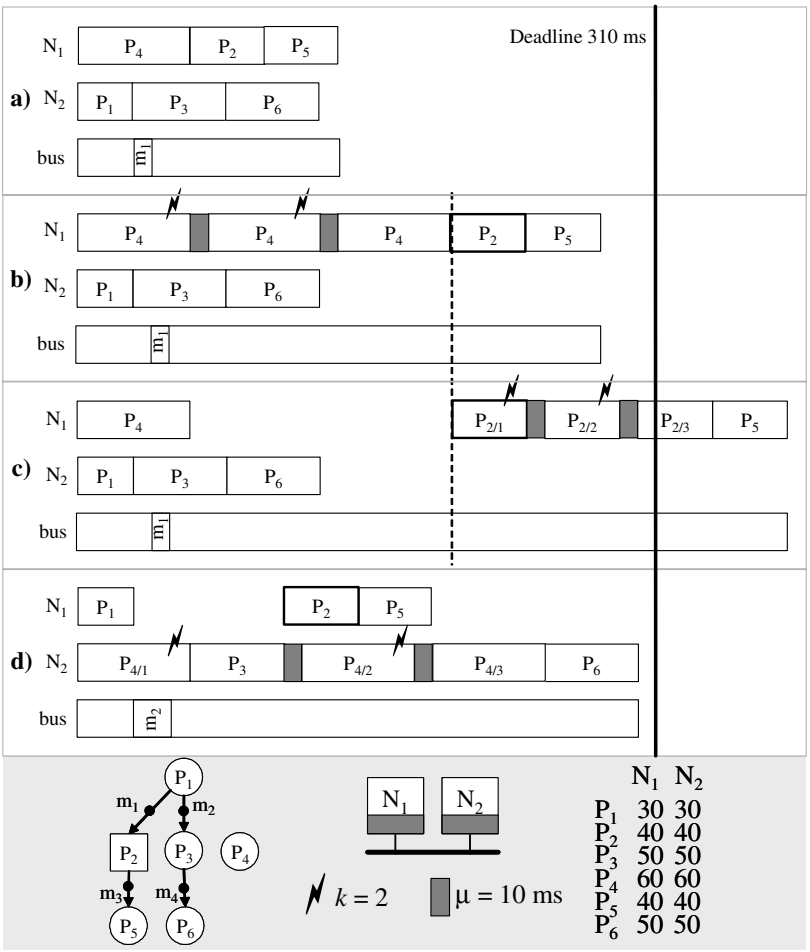


Figure 5.10: Mapping with Frozen Processes

1. Note that process P_3 is scheduled between the first execution $P_{4/1}$ of process P_4 and re-execution $P_{4/2}$ of P_4 because process P_3 has higher priority than process P_4, according to the partial critical path (PCP) [Ele00] priority function applied to list scheduling (Section 4.3).

Another example, illustrating the importance of considering frozen messages during the mapping process, is depicted in Figure 5.11. The application consists also of six processes, P_1 to P_6. We impose a deadline of 290 ms for the application. Let us consider that all messages transmitted on the bus are frozen.

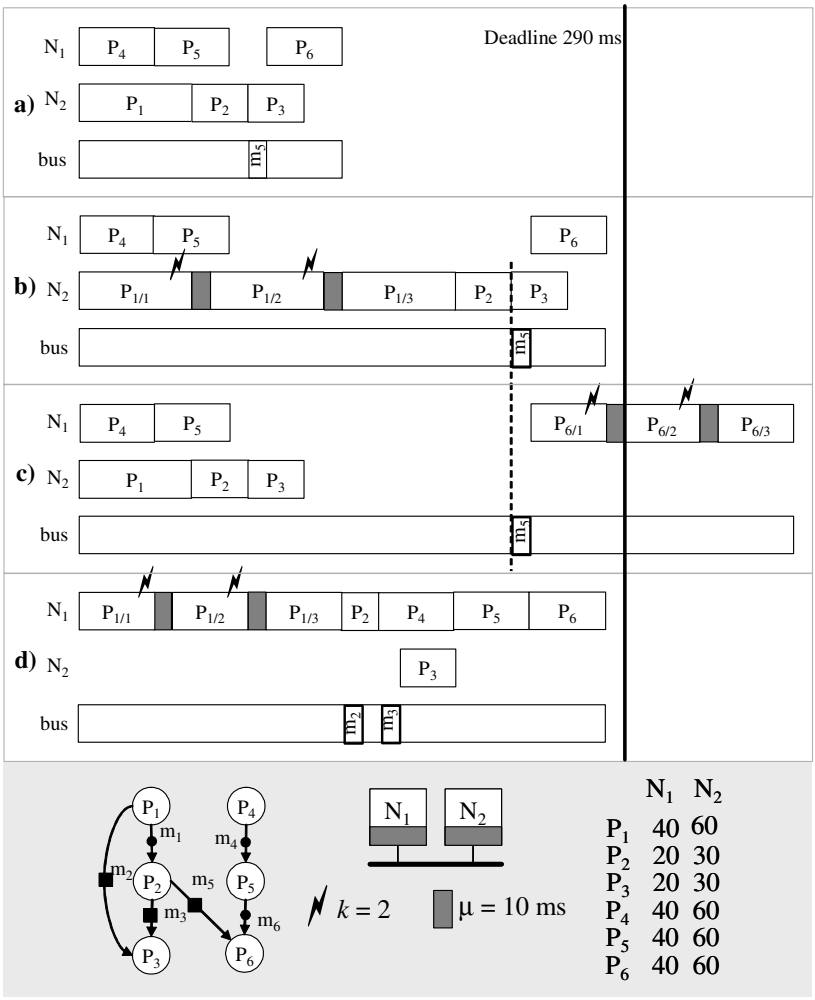


Figure 5.11: Mapping with Frozen Messages

This means that messages will have the same start time on the bus regardless of the particular fault scenario that happens. The optimal mapping, ignoring transparency, is presented in Figure 5.11a. The application meets the deadline even in the worst case fault scenario as shown in Figure 5.11b. This mapping is balanced and communications are minimized. Once we introduce transparency, the application becomes unschedulable, as illustrated in Figure 5.11c. However, considering the transparency requirements during mapping leads to a schedulable solution depicted, with its worst-case scenario, in Figure 5.11d. Counterintuitively, this solution is not balanced and, instead of one message, two messages are sent via the bus.

The examples presented have shown that transparency properties have to be carefully considered during mapping and that mapping alternatives which are optimal for non-transparent solutions can be inefficient when transparency is introduced.

5.2.2 OPTIMIZATION STRATEGY

Our mapping optimization strategy, outlined in Figure 5.12, determines a mapping $\mathcal{M}$ for application $\mathcal{A}$ on computation nodes $\mathcal{N}$ such that the application is schedulable and the transparency requirements $\mathcal{T}$ are satisfied. The optimization strategy receives as input the application graph $\mathcal{G}$, the maximum number of faults k in the system period, the architecture $\mathcal{N}$, transparency requirements $\mathcal{T}$, and deadline D. The output of the algorithm is a mapping $\mathcal{M}$ of processes to nodes, and a conditional schedule table for processes and messages, which is produced with the conditional scheduling algorithm presented in Section 4.3.

The design problem outlined above is NP complete [Ull75]. Our strategy, presented in Figure 5.12, is to address separately the mapping and scheduling. We start by determining an initial mapping $\mathcal{M}_{init}$ with the InitialMapping function (line 1). This is a straightforward mapping that balances computation node utilization and minimizes communications. The schedulability of the

resulted system is evaluated with the conditional scheduling algorithm (lines 2-3). If the initial mapping is unschedulable, then we iteratively improve the mapping of processes on the critical path of the worst-case fault scenario aiming at finding a schedulable solution (lines 4-9). For this purpose, we use a hill-climbing heuristic, which combines a greedy algorithm and a method to recover from local optima.

A new mapping alternative $\mathcal{M}_{new}$ is obtained with a greedy algorithm, IterativeMapping (line 5) which is presented in the next section. Since IterativeMapping is a greedy heuristic it will very likely end up in a mapping $\mathcal{M}_{new}$, which is a local minimum. In order to explore other areas of the design space, we will restart the IterativeMapping heuristic with a new initial solution $\mathcal{M}_{init}$ which should not lead to the same local minimum. As recommended in literature [Ree93], an efficient way to find such a new initial mapping is the following: Given the actual solution $\mathcal{M}_{new}$ we apply an optimization run using a new cost function different from the global schedule length, which is used as a cost function in the optimization so far. This optimization run will

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

Figure 5.12: Optimization Strategy for Mapping with Performance/Transparency Trade-Offs

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

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

5.2.3 ITERATIVE MAPPING

IterativeMapping depicted in Figure 5.13 is a greedy algorithm that incrementally changes the mapping $\mathcal{M}$ until no further improvement (line 3) is possible. Our approach is to tentatively change the mapping of processes on the critical path of the application graph $\mathcal{G}$. The critical path $\mathcal{CP}$ is found by the function FindCP (line 6). Each process $P_i \in \mathcal{CP}$ on the critical path is tentatively moved to each node in $\mathcal{N}$. We evaluate each move in terms of schedule length, considering transparency properties $\mathcal{T}$ and the number of faults k (line 11).

The calculation of the schedule length should, in principle, be performed by conditional scheduling (ConditionalScheduling function, see Section 4.3). However, conditional scheduling takes too long time to be used inside such an iterative optimization loop. Therefore, we have developed a fast schedule length estimation heuristic, ScheduleLengthEstimation, which is used to

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

Figure 5.13: Iterative Mapping Heuristic (IMH)

guide the InitialMapping heuristic. The estimation heuristic is presented in the next section.

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

5.2.4 SCHEDULE LENGTH ESTIMATION

The worst-case fault scenario consists of a combination of k fault occurrences that leads to the longest schedule. The conditional scheduling algorithm, presented in Section 4.3, examines all the fault scenarios captured by the fault-tolerant conditional process graph (FT-CPG), produces the conditional schedule table, and implicitly determines the worst-case fault scenario.

However, the number of alternative paths to investigate is growing exponentially with the number of faults. Hence, the execution time of the conditional scheduling algorithm is also growing, as our experiments in Section 5.1.6 show. On one hand, conditional scheduling is, therefore, too slow to be used inside the mapping optimization loop. On the other hand, mapping optimization does not require generation of complete schedule tables. Instead, only the schedule length is needed in order to evaluate the quality of the current design solution. Hence, in this section, we are proposing a worst-case schedule length estimation heuristic.

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

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

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

Figure 5.14: Schedule Length Estimation

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

The ScheduleLengthEstimation heuristic is outlined in Figure 5.14. The set of all synchronization nodes $\mathcal{L}_s$ is generated (line 1). Priorities are assigned to all synchronization nodes (line 2). For priority assignment we use the partial critical path function outlined in [Ele00]. The estimation chooses synchronization nodes according to the assigned priorities such that it can derive their fixed start time. For each synchronization node, ScheduleLengthEstimation selects a set of

processes that will potentially introduce the largest delay (line 6).

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

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

5.2.5 EXPERIMENTAL RESULTS

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

We were first interested to evaluate the proposed heuristic for schedule length estimation (ScheduleLengthEstimation in Figure 5.14, denoted with SE), in terms of monotonicity, relative to the ConditionalScheduling (CS) algorithm presented in Section 4.3. SE is monotonous with respect to CS if for two alternative mapping solutions $\mathcal{M}_1$ and $\mathcal{M}_2$ it is true that if $\text{CS}(\mathcal{M}_1) \leq \text{CS}(\mathcal{M}_2)$ then also $\text{SE}(\mathcal{M}_1) \leq \text{SE}(\mathcal{M}_2)$.

For the purpose of evaluating the monotonicity of SE, 50 random mapping changes were performed for each application. Each of those mapping changes was evaluated with both SE and CS. The results are depicted in Table 5.2. As we can see, in over 90% of the cases, SE correctly evaluates the mapping decisions, i.e. in the same way as CS. The rate of monotonicity decreases slightly with the application dimension. However, it is not influenced by increasing the number of faults.

Another important property of SE is its execution time, presented on Table 5.3. Execution time of the SE is growing linearly with the number of faults and application size. Over all graph dimensions, the execution time of SE is always less than 1 sec. In comparison, the execution time of CS is growing exponen-

Table 5.2: Monotonicity (%)

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

Table 5.3: Execution Time (sec)

Number of processes	2 faults		3 faults		4 faults	
	SE	CS	SE	CS	SE	CS
20	0.01	0.07	0.02	0.28	0.04	1.37
30	0.13	0.39	0.19	2.93	0.26	31.50
40	0.32	1.34	0.50	17.02	0.69	318.88

Table 5.4: Mapping Improvement (%)

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

tially with the number of processes and the number of faults and can reach 318.88 seconds for 40 processes and 4 faults. This shows that the conditional scheduling cannot be used inside the optimization loop, while the scheduling heuristic is well-suited for design space exploration.

We were also interested to evaluate our mapping optimization strategy. Table 5.4 shows the improvement by mapping optimization that considers fault tolerance with transparency over straightforward mapping (InitialMapping in Figure 5.13), which does not consider the fault tolerance aspects. Thus, we determined using ConditionalScheduling the schedule length for two mapping alternatives: InitialMapping and the mapping obtained by our OptimizationStrategy in Figure 5.12. Table 5.4 presents the percentage improvement in terms of schedule length of our mapping optimization compared to the straightforward solution. The schedule length obtained with our mapping optimization algorithm is 30% shorter on average. This confirms that considering the fault tolerance and transparency aspects leads to significantly better design solutions and that the ES heuristic can be successfully used inside an optimization loop.

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

Finally, we have considered a real-life example implementing a vehicle cruise controller (CC), which was previously used to evaluate scheduling techniques in Chapter 4. We have considered the same deadline of 300 ms, $k = 2$ and $\mu = 2$ ms. The straightforward solution was unschedulable even with only 25% frozen messages and no frozen processes. However, the applica-

tion optimized with our mapping strategy, was easily schedulable with 85% frozen messages. Moreover, we could additionally introduce 20% frozen processes without violating timing constraints.

5.3 Conclusions

In the first part of this chapter, we have proposed a strategy for fault tolerance policy assignment and mapping. We decided 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 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.

In the second part of the chapter, we have proposed a mapping optimization strategy that supports performance/transparency trade-offs during the design process. Since the conditional scheduling algorithm is computation-intensive and cannot be used inside an optimization loop, we have proposed a fast estimation heuristic which is able to accurately evaluate a given mapping decision. The proposed mapping algorithm based on the estimation heuristic is able to produce effective design solutions for a given transparency set-up.

Chapter 6
Checkpointing

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. Secondly, we extend the optimization strategy for fault tolerance policy assignment presented in Section 5.1 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

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 was 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 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

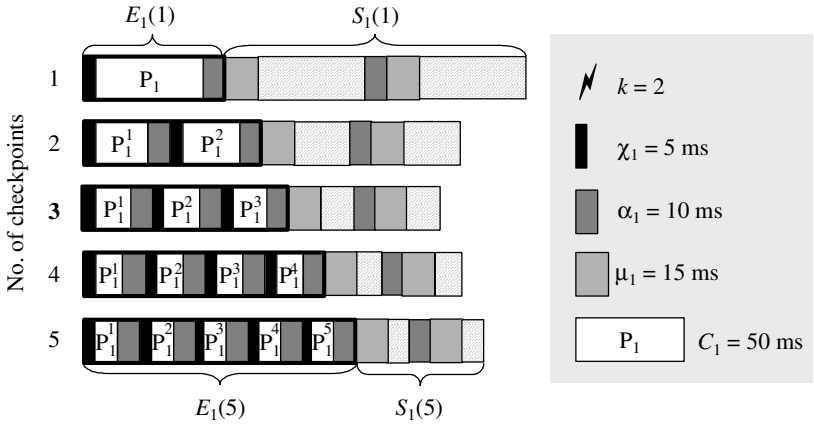


Figure 6.1: Locally Optimal Number of Checkpoints

first re-execution of the process. Thus, for one checkpoint, the recovery slack S_1 of process P_1 is (50 + 15) × 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) × 2 + 10 = 90 ms.

By introducing more checkpoints, the recovery slack S_1 can be thus 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:

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

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

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

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 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.

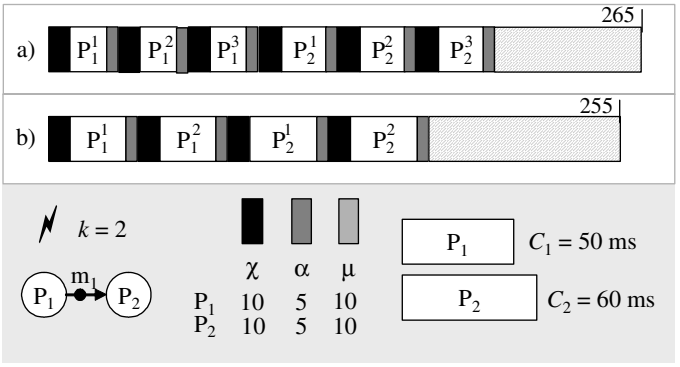


Figure 6.2: Globally Optimal Number of Checkpoints

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:

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

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

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

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 of checkpoints 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 6.2.4.

6.2 Policy Assignment with Checkpointing

In this section we extend the fault tolerance policy assignment algorithm presented in Section 5.1 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 active 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 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 $\mathcal{X}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow$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,

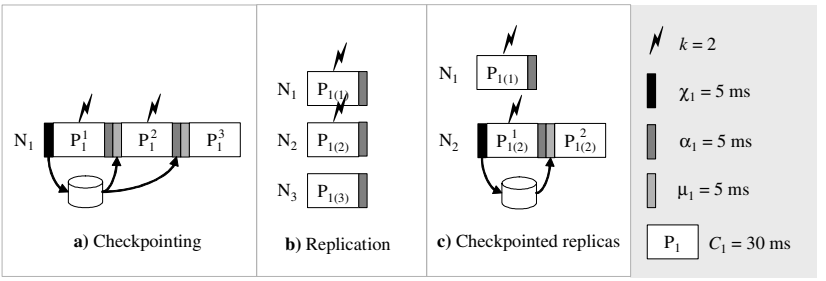


Figure 6.3: Policy Assignment: Checkpointing +Replication

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

thus the checkpoints are equally distributed throughout the execution time of the process. If process $P_i \in \mathcal{V}$ or replica $P_{i(j)} \in \mathcal{V}_R$ is not checkpointed, then we have $\mathcal{X}(P_i) = 0$ or $\mathcal{X}(P_{i(j)}) = 0$, respectively.

Each process $P_i \in \mathcal{V}$, besides its worst 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 set of process graphs (Section 3.1) 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 such that the k transient faults are tolerated and the imposed deadlines are guaranteed to be satisfied, within the constraints of the given architecture $\mathcal{N}$.

Determining a system configuration $\psi = <\mathcal{F}, \mathcal{X}, \mathcal{M}, \mathcal{S}>$ 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 $\mathcal{S}$ 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 were discussed in Section 5.1.4, 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. Some particular details of scheduling with checkpointing and replication are presented in Section 6.2.2.

6.2.1 MOTIVATIONAL EXAMPLES

Let us illustrate some of the issues related to policy assignment with checkpointing. In the example presented in Figure 6.4 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

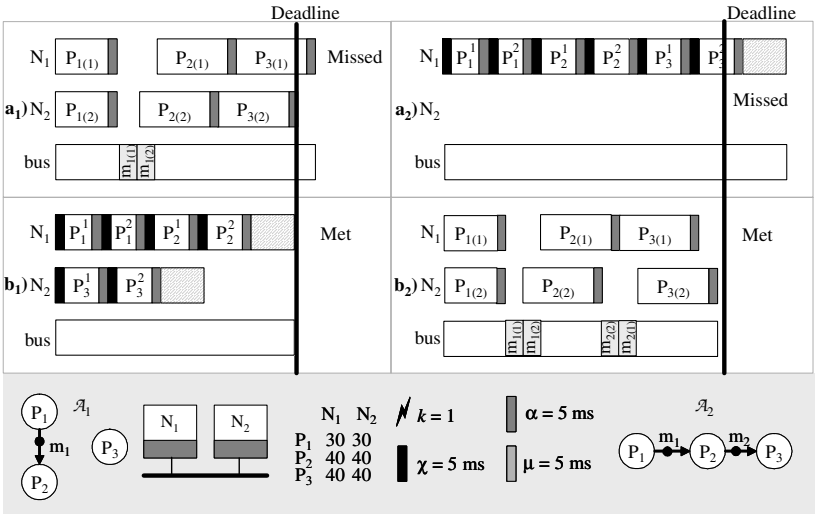


Figure 6.4: Comparison of Checkpointing and Replication

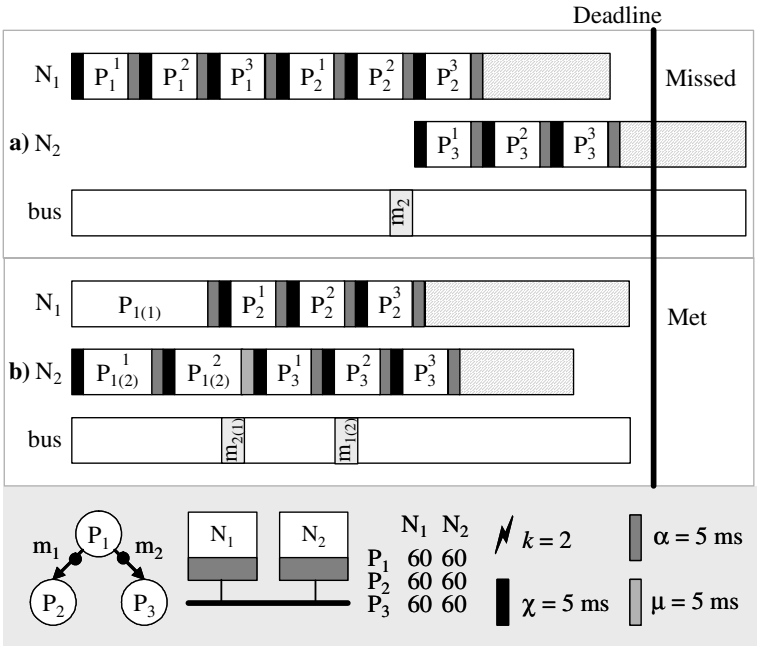


Figure 6.5: Combining Checkpointing and Replication

architecture, and processes can be mapped to any node. The fault model assumes a single fault, thus $k = 1$, and the fault-tolerance overheads are presented in the figure. The application $\mathcal{A}_1$ has a deadline of 140 ms depicted with a thick vertical line. We have to decide which fault-tolerance technique to use.

In Figure 6.4a_1, a_2, b_1 and b_2 we depict the root schedules[1] for each node and the bus. Comparing the schedules in Figure 6.4a_1 and Figure 6.4b_1, we can observe that using active replication (a_1) the deadline is missed. However, using checkpointing (b_1) we are able to meet the deadline. Each process has an optimal number of two checkpoints in Figure 6.4b_1. If we consider application $\mathcal{A}_2$, similar to $\mathcal{A}_1$ but with process P_3 data dependent on

1. The schedules depicted are optimal.

P_2, as illustrated in the right lower corner of Figure 6.4, the deadline of 180 ms is missed in Figure 6.4a$_2$ if checkpointing is used, and it is met when replication is used as in Figure 6.4b$_2$.

This example shows that the particular technique to use has to be carefully adapted to the characteristics of the application. Moreover, the best result is often to be obtained when both techniques are used together, some processes being checkpointed, while others replicated.

Let us now consider the example in Figure 6.5, where we have an application with three processes, P_1 to P_3, mapped on an architecture of two nodes, N_1 and N_2. The processes can be mapped to any node, and the worst-case execution times on each node are given in a table. In Figure 6.5a all processes are using checkpointing, and the depicted root schedule is optimal for this case. Note that m_2 has to be delayed to mask two potential faults of P_1 to node N_2. With this setting, using checkpointing will miss the deadline. However, combining checkpointing with replication, as in Figure 6.5b where process P_1 is replicated, will meet the deadline. $P_{1(1)}$ is a simple replica without checkpointing and message $m_{2(1)}$ from this replica is sent directly after completion of $P_{1(1)}$. In second replica $P_{1(2)}$ of process P_1, one fault has to be masked, which delays the message $m_{1(2)}$. However, the delay of message $m_{1(2)}$ is less then the delay of message m_2 in Figure 6.5a.

6.2.2 SCHEDULING WITH CHECKPOINTING AND REPLICATION

Some additional discussion is needed to understand generation of the root schedule in Figure 6.5b. As discussed in Section 5.1.4, scheduling of replica descendants is slightly different from regular scheduling of re-executed processes. The same applies for checkpointing and replication. In Figure 6.5b, process P_2 is scheduled directly after replica $P_{1(1)}$ of process P_1.

Initially, we suppose that the shared recovery slack on the node N_1 is equal to the individual recovery slack of process P_2. In

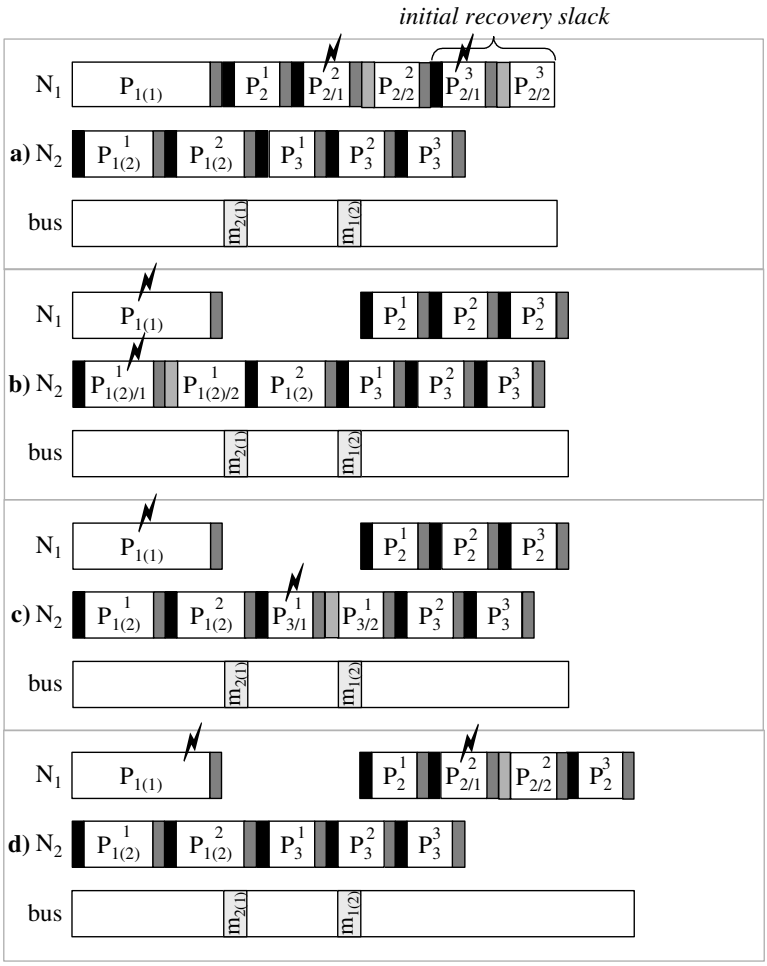


Figure 6.6: Generation of the Root Schedule Combining Replication and Checkpointing

Figure 6.6a, we show a fault scenario, where two faults happen in process P_2 and accommodated into the initial shared recovery slack. However, if a fault corrupts replica $P_{1(1)}$ of process P_1, process P_2 has to wait until message $m_{1(2)}$ arrives regardless of

whether and how faults occur on node N_2. This situation is illustrated in Figure 6.6b, where replica $P_{1(2)}$ is recovered from a fault, and in Figure 6.6c, where second execution segment P_3^2 of process P_3 is affected. If there is no fault occurrences on the node N_2, one more fault can happen on the node N_1 and affect execution of process P_2. Therefore, the shared recovery slack on the node N_1 has to be adjusted accordingly. In Figure 6.6d, we show a fault scenario where both replica $P_{1(1)}$ and process P_2 are affected by faults, which is the worst-case fault scenario of the considered application from Figure 6.5. This fault scenario has been accommodated into the adjusted shared recovery slack on the node N_1, depicted in Figure 6.5b.

6.2.3 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. Our optimization strategy which produces the configuration ψ leading to a schedulable fault-tolerant application is outlined in Figure 6.7 and has two steps:

1. In the first step (lines 1–3) we quickly decide on an initial fault-tolerance policy assignment given by $\mathcal{F}^0$, and an initial

```
OptimizationStrategy(A, N)
1     Step 1:  <M^0, F^0> = InitialMPA(A, N)
2              S^0 = ListScheduling(A, N, M^0, F^0)
3              if S^0 is schedulable then return ψ^0 end if
4     Step 2:  ψ = TabuSearchMPA(A, N, ψ^0)
5              if S is schedulable then return ψ end if
6     return ψ
end OptimizationStrategy
```

Figure 6.7: Design Optimization Strategy for Fault Tolerance Policy Assignment with Checkpointing

mapping $\mathcal{M}^0$. The initial mapping and fault-tolerance policy assignment algorithm (InitialMPA line 2 in Figure 6.7) assigns a checkpointing policy with a locally optimal number of checkpoints (using the equation in Figure 6.1b) 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 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.

6.2.4 OPTIMIZATION ALGORITHMS

For deciding the mapping and fault tolerance policy assignment with checkpointing we use a tabu search based heuristic approach, TabuSearchMPAChk, which is adaptation of the TabuSearchMPA algorithm, presented in Section 5.1.5. 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 schedule 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 merged application graph $\mathcal{G}$, the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}.

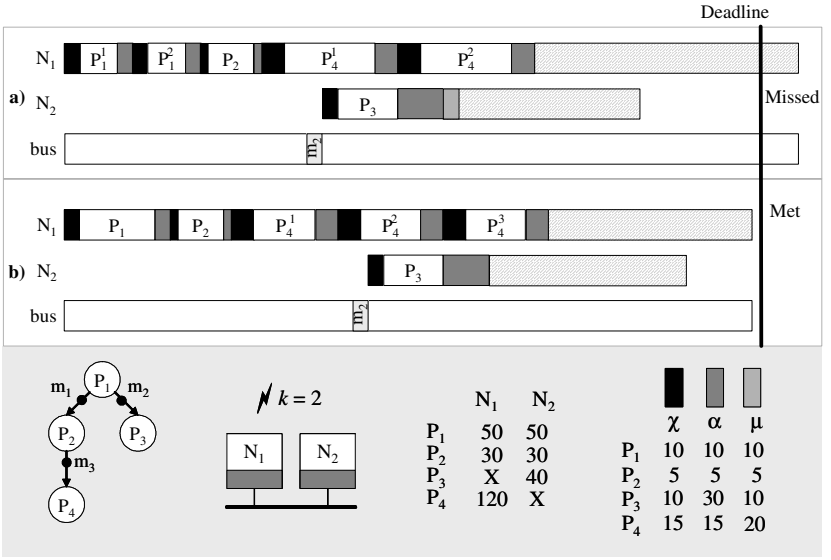


Figure 6.8: Restricting the Moves for Setting the Number of Checkpoints

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 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. We define the critical path as the path through the merged graph G which corresponds to the longest delay in the schedule table.

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 6.8 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 over-

heads are also given in the figure, and we can have 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 $n_4^0 = 3$. Let us assume that our current solution is the one depicted in Figure 6.8a, 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 6.8a, 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) 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 6.8b.

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.

6.2.5 EXPERIMENTAL RESULTS

For the evaluation of our algorithms we 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. A time-division multi-

ple access (TDMA) bus with the time-triggered protocol (TTP) [Kop03] has been used as a communication media. 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 was 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 were randomly generated for each application dimension, thus a total of 75 applications were used for experimental evaluation. The experiments were 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. For this, we have implemented each application without any fault-tolerance concerns. This non-fault-tolerant implementation, NFT, has been obtained using an approach similar to the algorithm in Figure 6.7 but without fault-tolerance techniques. The same applications have been implemented, on the same amount of resources, using the optimization strategy in Figure 6.7, with multiple checkpoints and replication (MCR). Together with the MCR approach we have also evaluated two extreme approaches: MC that considers only checkpointing, and MR which relies only on replication for tolerating faults. MC and MR use the same optimization approach as MCR, but besides the mapping moves, they consider assigning only checkpointing

Table 6.1: Fault Tolerance Overheads

Number of processes	% maximum overhead	% average overhead	% minimum overhead
20	98.36	70.67	48.87
40	116.77	84.78	47.30
60	142.63	99.59	51.90
80	177.95	120.55	90.70
100	215.83	149.47	100.37

(including the optimization of the number of checkpoints) or only replication, respectively. In addition, we have also implemented a checkpointing-only strategy, namely MC0, similar to MC, but where the number of checkpoints is fixed based on the formula (6.3), updated from [Pun97]. For these experiments, we have derived the shortest schedule within an imposed time limit for optimization: 1 minute for 20 processes, 10 for 40, 30 for 60, 2 hours and 30 minutes for 80 and 6 hours for 100 processes.

Let δ_{MCR} and δ_{NFT} be the lengths of the root schedules obtained using MCR and NFT, respectively. The overhead is defined as $100 \times (\delta_{MCR} - \delta_{NFT}) / \delta_{NFT}$. The fault-tolerance overheads of MCR compared to NFT are presented in Table 6.1.[1] The

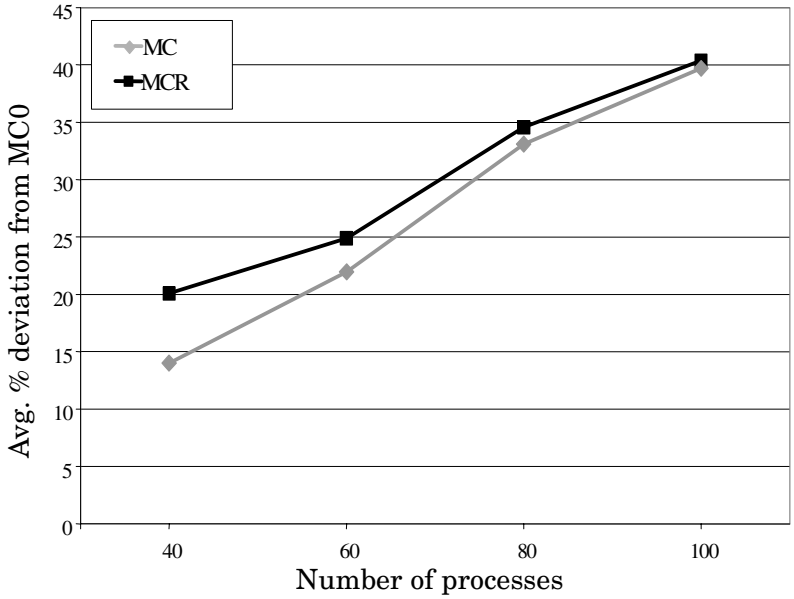


Figure 6.9: Deviation of MC and MCR from MC0 with Varying Application Size

1. Note that checkpointing overhead χ and error-detection overhead α are explicitly accounted for in the value of fault tolerance overheads presented in Table 6.1.

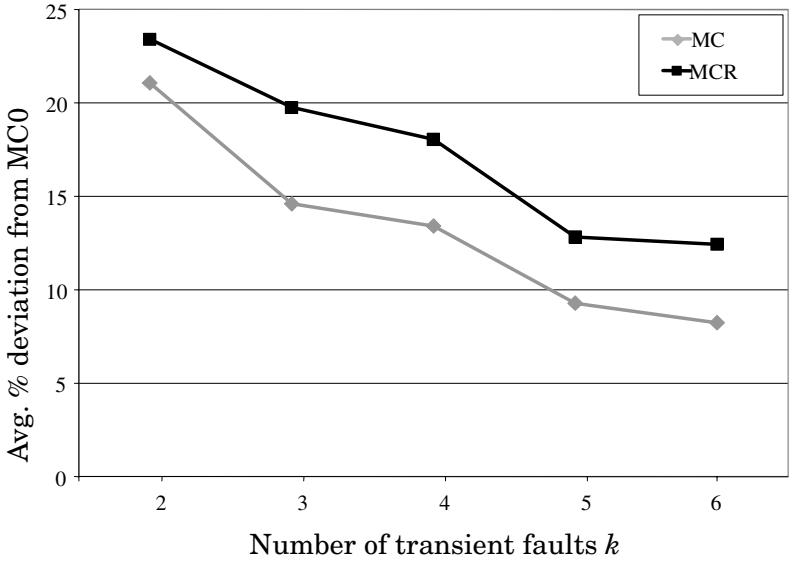


Figure 6.10: Deviation of MC and MCR from MC0 with Varying the Number of Transient Faults

MCR approach can offer fault-tolerance within the constraints of the architecture at an average overhead of 88.49%. In the case only replication is used (MR), the overheads compared to NFT are very large (e.g., 306.51% on average for applications 100 processes).

We were interested to compare the quality of MCR to MC0 and MC. In Figures 6.9-6.10 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 6.9-6.10 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 6.9 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 6.11, 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 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.

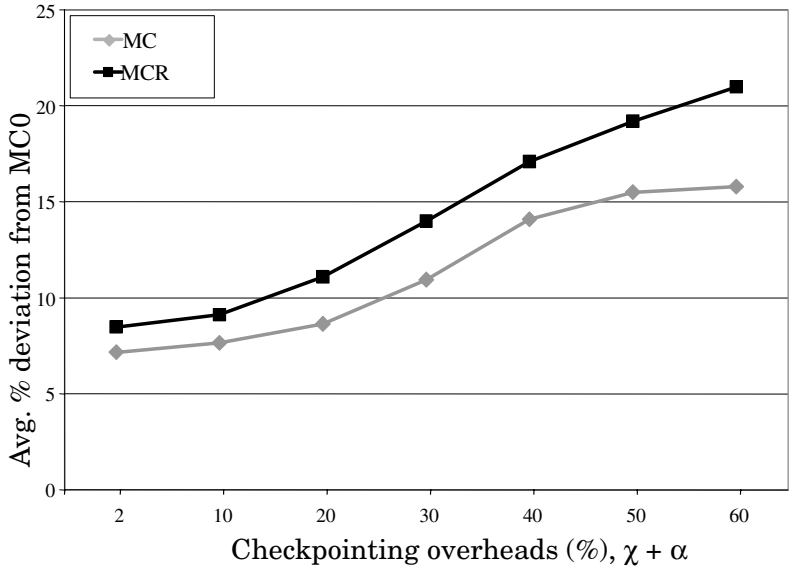


Figure 6.11: Deviation of MC and MCR from MC0 with Varying Checkpointing Overheads

We have also evaluated the MCR and MC approaches with increasing number of transient faults. We have considered applications with 40 processes mapped on 4 computation nodes, and varied k from 2 to 6, see Figure 6.10. 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 8.30%, while MCR improves with 10.29%).

Finally, we considered a real-life example implementing a vehicle cruise controller (CC), which was used to evaluate scheduling 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 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 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, as in the case of MR, the delay is 369 ms, which, again, is greater than the deadline.

6.3 Conclusions

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

Chapter 7
Conclusions and Future Work

IN THIS THESIS we have presented several strategies for scheduling, mapping and policy assignment of fault-tolerant embedded systems. Emphasis has been also placed on 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 this final chapter, we summarize the work presented in the thesis and point out ideas for future work.

7.1 Conclusions

In this thesis we have considered hard 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 by messages. The processes are scheduled by static

cyclic scheduling. To provide fault tolerance against transient faults, processes are assigned with re-execution, replication, or recovery with checkpointing.

Scheduling. 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 schedule tables.

The second scheduling approach handles only a fixed transparency set-up, transparent recovery, where all messages on the bus have to be sent at fixed times, regardless of fault occurrences. This scheduling approach is much faster than 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.

Mapping and fault tolerance policy assignment. We have developed several algorithms for policy assignment and 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 re-execution or 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.

Transparency is a very important property that makes a system easier to debug and, in principle, safer. The amount of memory required to store contingency schedules is also less. We have shown how performance/transparency trade-offs imposed by designers can be supported during the design process. The main contribution of this approach is the ability to consider the transparency requirements imposed by the designer during the mapping optimization. The mapping is driven by a schedule estimation heuristic which is able to accurately evaluate a given mapping decision. Considering the fault-tolerance and transparency requirements during the mapping optimization process we have been able to provide transparency-aware fault tolerance under limited resources.

Checkpoint optimization. We have also addressed the problem of checkpoint optimization. We have shown that by globally optimizing the number of checkpoints, as opposed to the approach when processes were 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.

All proposed algorithms have been implemented and evaluated on numerous synthetic applications and a real-life example from automotive electronics. The results obtained have shown the efficiency of the proposed approaches and methods.

7.2 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.

Soft real-time. The main focus in this thesis was on hard real-time embedded systems. However, soft real-time systems that do not need to strictly guarantee that the application meets deadlines are emerging in many industrial areas. The design optimization with fault tolerance can be important for providing the appropriate level of quality of service in such systems. Another interesting aspect is the integration between soft real-time and hard real-time applications. Design optimization with fault tolerance can be important for designing efficient and reliable embedded systems with mixed soft-and-hard real-time processes.

Probabilistic fault model. In this thesis, we have considered a deterministic fault model, where a maximum number of k faults can happen during one cycle of application execution. This model can be extended to capture variations of fault occurrence rates over time and different behaviour of faults on different computation nodes. Using a probability distribution function of fault occurrences is one possible suggestion. In the future, we can adapt approaches presented in this thesis to capture a probabilistic fault model, which can, potentially, lead to more efficient design solutions.

Fault-tree analysis. Some faults are not activated and, at the same time, one fault may cause several errors. This can be captured in a fault tree, where it is possible to investigate a complete chain of events from a fault to a failure. Moreover, faults and failures can be ranked according to their criticality. The fault tree will become different after re-designing the system by changing mapping and fault tolerance policy assignment, or customizing transparency properties. Considering the fault tree and its changes during design optimization can improve accuracy and reduce costs of design solutions. Non-critical faults can be ignored, while the critical will receive a close attention.

Appendix I

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

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

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.

References

[Als01] K. Alstrom and J. Torin, "Future Architecture for Flight Control Systems", *Proc. 20th 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. 15th 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.

[Aud95] N. C. Audsley et al., "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, 8, 173-198, 1995.

[Axe96] J. Axelsson, "Hardware/Software Partitioning Aiming at Fulfilment of Real-Time Constraints", *Systems Architecture*, 42, 449-464, 1996.

[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.

[Bax95] M. J. Baxter, M. O. Tokhi, and P. J. Fleming, "Task-Processor Mapping for Real-Time Parallel Systems Using Genetic Algorithms with Hardware-in-the-Loop", *Proc. First Intl. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 158-163, 1995.

[Ben03] A. Benso et al., "A Watchdog Processor to Detect Data and Control Flow Errors", *Proc. 9th 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 et al., "Hardware/Software Co-Design of Digital Telecommunication Systems", *Proc. of the IEEE*, 85(3), 391-418, 1997.

[Bur96] A. Burns et al., "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Proc. Euromicro Workshop on Real-Time Systems*, 29–33, 1996.

[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. Int. 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.

[Col95] M. Coli and P. Palazzari, "A New Method for Optimization of Allocation and Scheduling in Real-Time Applications", *Proc. 7^{th} Euromicro Workshop on Real-Time Systems*, 262-269, 1995.

[Col03] A. Colin and S. M. Petters, "Experimental Evaluation of Code Properties for WCET Analysis", *Proc. 24^{th} IEEE Real-Time Systems Symp.*, 190-199, 2003.

[Con05] J. Conner et al., "FD-HGAC: A Hybrid Heuristic/ Genetic Algorithm Hardware/Software Co-synthesis Framework with Fault Detection", *Proc. Asia and South Pacific Design Automation Conf.*, 709-712, 2005.

[Con03] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability", *IEEE Micro*, 23(4), 14-19, 2003.

[Cor04] F. Corno et al., "Evaluating the Effects of Transient Faults on Vehicle Dynamic Performance in Automotive Systems", *Proc. Intl. Test Conference*, 1332-1339, 2004.

[Dav98] B. P. Dave and N. K. Jha, "COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Systems", *IEEE Trans. on CAD*, 17(10), 900-919, 1998.

[Dav99] B. P. Dave, G. Lakshminarayana, and N. J. Jha, "COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems", *IEEE Trans. on VLSI Systems*, 7(1), 92-104, 1999.

[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.

[Dic98] R. P. Dick and N. K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Proc. Intl. Conf. on CAD*, 1998.

[Ele97] P. Eles et al., "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 et al., "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Trans. on VLSI Systems*, 8(5), 472-491, 2000.

[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.

[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.

[Gir03] A. Girault et al., "An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules", *Proc. Intl. Conf. on Dependable Systems and Networks*, 159-168, 2003.

[Glo86] F. Glover, "Future Paths for Integer Programming and Links to Artificial Intelligence", *Computers and Operations Research*, 13(5), 533-549, 1986.

[Gol89] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Gol03] O. Goloubeva et al., "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.

[Gup95] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, 1995.

[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.

[Han86] P. Hansen, "The Steepest Ascent Mildest Descent Heuristic for Combinatorial Programming", *Congress on Numerical Methods in Combinatorial Optimization*, 1986.

[Han02] H. A. Hansson et al., “Integrating Reliability and Timing Analysis of CAN-based Systems”, *IEEE Trans. on Industrial Electronics*, 49(6), 1240-1250, 2002.

[Har01] S. Hareland et al., “Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes”, *Proc. Symp. on VLSI Technology*, 73-74, 2001.

[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 et al., “Measured Faults during Lightning Storms”, *Proc. IEEE PowerTech’2005*, Paper 72, 5p., 2005.

[Hen96] P. van Hentenryck and V. Saraswat, “Strategic Directions in Constraint Programming”, *ACM Computing Surveys*, 28(4), 701-726, 1996.

[Her00] A. Hergenhan and W. Rosenstiel, “Static Timing Analysis of Embedded Software on Advanced Processor Architectures”, *Proc. Design, Automation and Test in Europe Conf.*, 552-559, 2000.

[Hil00] M. Hiller, “Executable Assertions for Detecting Data Errors in Embedded Control Systems”, *Proc. Intl. Conf. on Dependable Systems and Networks*, 24-33, 2000.

[Hol75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[Izo05] V. Izosimov et al., “Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems”, *Proc. Design Automation and Test in Europe Conf.*, 864-869, 2005.

[Izo06a] V. Izosimov et al., "Mapping of Fault-Tolerant Applications with Transparency on Distributed Embedded Systems", *Proc. 9th Euromicro Conf. on Digital System Design*, 313-320, 2006.

[Izo06b] V. Izosimov et al., "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Tradeoffs for Distributed Embedded Systems", *Proc. Design Automation and Test in Europe Conf.*, 706-711, 2006.

[Izo06c] V. Izosimov et al., "Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication", *Proc. 3rd IEEE Intl. Workshop on Electronic Design, Test & Applications*, 440-447, 2006.

[Jia00] Jia Xu and D. L. Parnas, "Priority Scheduling Versus Pre-Run-Time Scheduling", *Real Time Systems*, 18(1), 7-24, 2000.

[Jie92] Jien-Chung Lo et al., "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*, 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 et al., "A Hybrid Framework of Worst-Case Execution Time Analysis for Real-Time Embedded System Software", *Proc. IEEE Aerospace Conf.*, 1-10, 2005.

[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 et al., "Modeling and Simulation of Time Domain Faults in Digital Systems", *Proc. 10th 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.

[Kas84] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", *IEEE Trans. on Computers*, 33(11), 1023-1029, 1984.

[Kim99] K. Kimseng et al., "Physics-of-Failure Assessment of a Cruise Control Module", *Microelectronics Reliability*, 39, 1423-1444, 1999.

[Kop89] H. Kopetz et al., "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*, 9(1), 25-40, 1989.

[Kop90] H. Kopetz et al., "Tolerating Transient Faults in MARS", *Proc. 20th 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. 23rd 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 et al. "From a Federated to an Integrated Architecture for Dependable Embedded Real-Time Systems", *Tech. Report 22, Technische Universität Wien*, 2004.

[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.

[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.

[Lak99] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, "Wawesched: A Novel Scheduling Technique for Control-Flow Intensive Designs", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* 18(5), 1999.

[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 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.

[Man04] S. Manolache, P. Eles, and Z. Peng, "Optimization of Soft Real-Time Systems with Deadline Miss Ratio Constraints", *Proc. 10^{th} IEEE Real-Time and Embedded Technology and Applications Symp.*, 562-570, 2004.

[May78] T. C. May and M. H. Woods, "A New Physical Mechanism for Soft Error in Dynamic Memories", *Proc. 16^{th} Intl. Reliability Physics Symp.*, 33-40, 1978.

[Met53] N. Metropolis et al., "Equation of State Calculation by Fast Computing Machines", *Chemical Physics*, 21, 1087-1091, 1953.

[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.*, 524-533, 1998.

[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.

[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.

[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.*, 209-223, 2005.

[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.*, 1164–1169, 2004.

[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 et al., "Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications", *Proc. Design, Automation and Test in Europe Conf.*, 1028-1033, 2004.

[Pop04b] P. Pop et al., "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 et al., "Scheduling and Mapping in an Incremental Design Methodology for Distributed Real-Time Embedded Systems", *IEEE Trans. on VLSI Systems*, 12(8), 793-811, 2004.

[Pra94c] S. Prakash and A. Parker, "Synthesis of Application-Specific Multiprocessor Systems Including Memory Components", *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.

[Rab93] D. J. Rabideau and A. O. Steinhardt, "Simulated Annealing for Mapping DSP Algorithms onto Multiprocessors", *Proc. 27th Asilomar Conf. on Signals, Systems and Computers*, 668-672, 1993.

[Ree93] C. R. Reevs, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, 1993.

[Ros05] D. Rossi et al., "Multiple Transient Faults in Logic: An Issue for Next Generation ICs?", *Proc. 20th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 352-360, 2005.

[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.

[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.

[Str06] B. Strauss et al., "Unsafe at Any Airspeed?", *IEEE Spectrum*, 43(3), 44-49, 2006.

[Sun95] Sung-Soo Lim et al., "An Accurate Worst Case Timing Analysis for RISC Processors", *IEEE Trans. on Software Engineering*, 21(7), 593-604, 1995.

[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.

[Tsi01] Y. Tsiatouhas et al., "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.

[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.

[Wei04] Wei Huang et al., "Compact Thermal Modeling for Temperature-Aware Design," *Proc. 41st Design Automation Conf.*, 878-883, 2004.

[Xie04] Y. Xie et al., "Reliability-Aware Co-synthesis for Embedded Systems", *Proc. 15th IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors*, 41-50, 2004.

[Yin04] Ying Zhang, R. Dick, and K. Chakrabarty, "Energy-Aware Deterministic Fault Tolerance in Distributed Real-Time Embedded Systems", *Proc. 42nd Design Automation Conf.*, 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.

[Ziv97] A. Ziv and J. Bruck, "An On-Line Algorithm for Checkpoint Placement", *IEEE Trans. on Computers*, 46(9), 976-985, 1997.

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2006-11-15

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report: category

☒ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN 91-85643-72-6

ISRN LiU-Tek-Lic-2006:58

Serietitel och serienummer
Title of series, numbering

ISSN 0280-7971

Linköping Studies in Science and Technology

Thesis No. 1277

URL för elektronisk version

http://www.ida.liu.se/~eslab

Titel
Title

Scheduling and Optimization of Fault-Tolerant Embedded Systems

Författare
Author

Viacheslav Izosimov

Sammandrag
Abstract

Safety-critical applications have to function correctly even in 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. These techniques are considered in the context of distributed real-time systems with non-preemptive static cyclic scheduling.

Safety-critical applications have 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 consideration of fault tolerance are required.

The 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, and checkpoint distribution.

Dedicated scheduling techniques and mapping optimization strategies are also proposed to handle customized transparency requirements associated with processes and messages. By providing fault containment, transparency can, potentially, improve testability and debugability of fault-tolerant applications.

The efficiency of the proposed scheduling techniques and design optimization strategies is evaluated with extensive experiments conducted on a number of synthetic applications and a real-life example. The experimental results show that considering fault tolerance during system-level design optimization is essential when designing cost-effective fault-tolerant embedded systems.

Nyckelord
Keywords

Embedded systems, Real-Time Systems, Design optimization, Fault tolerance, Transient faults, Soft errors

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Licentiate Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598 **Rego Granlund:** C^{3}Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.

No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001.
Fif-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
Fif-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
Fif-a-51 **Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
Fif-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.
No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tesanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.

No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
Fif-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
Fif-a 71 **Emma Eliasson:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency im Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.
No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation-What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.