A Software Fault-Tolerance Mechanism for Mixed-Critical Real-Time Applications on Consumer-Grade Many-Core Processors

vorgelegt von
Peter Munk, M.Sc.
geb. in Ostfildern

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Ben Juurlink
Gutachter:	Prof. Dr. Hans-Ulrich Heiß
Gutachter:	Prof. Dr.-Ing. Michael Glaß
Gutachterin:	Prof. Dr. Sabine Glesner

Tag der wissenschaftlichen Aussprache: 22. Juli 2016

Berlin 2016

Abstract

The number of cores per processor continues to increase due to higher integration rates and smaller feature sizes. This development of many-core processors leads to a higher susceptibility to soft errors, which are caused by high-energy particle strikes.

At the same time, the complexity and computational demand of automotive applications is rising steadily, following the vision of highly automated driving. Due to their computational performance and their comparatively low costs, commercial off-the-shelf many-core processors become attractive in cost-driven mass markets such as the automotive domain. In order to execute safety-critical automotive applications on such processors and fulfill the obligatory safety standards, fault-tolerance mechanisms have to be installed. However, hardware-implemented fault-tolerance mechanisms increase the unit costs and are uncommon in consumer-grade many-core processors, which are designed for applications without safety requirements.

In this thesis, we present an adaptive software-implemented fault-tolerance mechanism that is based on the N modular redundancy principle, leveraging the spatial redundancy of many-core processors. In order to eliminate the voter as a single point of failure, our mechanism employs two fail-silent voters that check and repair each other. The fail-silent behavior is achieved by an encoded voting procedure. Our fault-tolerance mechanism includes a state-conserving repair procedure to recover from replica and voter failures. In contrast to related work, we consider failures in all software components, including the OS kernel.

In order to meet the real-time requirement of automotive applications, our fault-tolerance mechanism utilizes well-known scheduling policies. However, consumer-grade many-core processors do not commonly provide an inter-core communication with latency and bandwidth guarantees. Therefore, we propose a software-based approach that guarantees a bounded latency for dynamic communication. The main principle of our approach is to limit the packet injection rate for all sources.

Furthermore, we present a framework that eases the development of safety-critical real-time applications on consumer-grade many-core processors. The framework adapts our fault-tolerance mechanism for each task of a mixed-critical task set such that the task's reliability and availability requirements are satisfied with minimum resource usage. A discrete Markov chain model of our mechanism is used to determine the resulting reliability and availability of each task. Measurements on a cycle-accurate and bit-accurate many-core processor simulator with realistic fault injections show the advantage of our adaptive software fault-tolerance mechanism and demonstrate the trade-off between resource usage and fault-tolerance level.

Zusammenfassung

Aufgrund schrumpfender Strukturbreiten und eines steigenden Integrationsgrads wächst die Anzahl der Rechenkerne pro Prozessor kontinuierlich. Diese Entwicklung von Many-Core Prozessoren bedingt allerdings eine höhere Anfälligkeit für transiente Fehler, welche aus energiereichen Partikeleinschlägen resultieren.

Getrieben von der Vision des hochautomatisierten Fahrens nehmen die Komplexität und der Rechenbedarf von Software im Automobilbereich stetig zu. Aufgrund ihrer Rechenleistung und ihrer vergleichsweise geringen Stückkosten wird der Einsatz von auf dem Markt erhältlichen Many-Core Prozessoren in stückkostengetrieben Massenmärkten wie dem Automobilereich zunehmend attraktiv. Um sicherheitskritische Anwendungen auf solchen Prozessoren auszuführen und die entsprechenden Sicherheitsnormen zu erfüllen, werden Fehlertoleranzmechanismen benötigt. In Hardware umgesetzte Mechanismen erhöhen die Stückkosten und sind nicht in gängigen Many-Core Prozessoren verbaut, welche für Anwendungen ohne Sicherheitsanforderungen ausgelegt sind.

In dieser Dissertation präsentieren wir einen adaptiven, in Software umgesetzten Fehlertoleranzmechanismus, welcher aufgrund des Prinzips der N Modular Redundancy die räumliche Redundanz von Many-Core Prozessoren ausnutzt. Um die Vergleichseinheit als singuläre Fehlerursache auszuschließen, enthält unser Mechanismus zwei Vergleichseinheiten, welche sich gegenseitig überprüfen und reparieren. Wir verwenden ein kodiertes Vergleichsverfahren, um sicherzustellen, dass jede Vergleichseinheit im Fehlerfall keine falschen Ergebnisse ausgibt. Unser Fehlertoleranzmechanismus enthält eine zustandserhaltende Reparaturprozedur, um Ausfälle von Repliken und Vergleichseinheiten zu kompensieren. Im Gegensatz zu verwandten Arbeiten berücksichtigt unser Ansatz Fehler in allen Softwarekomponenten, einschließlich dem Betriebssystem.

Um die Echtzeitanforderungen der Anwendung zu erfüllen, arbeitet unser Fehlertoleranzmechanismus mit bestehenden Scheduling-Verfahren. Allerdings bieten gängige Many-Core Prozessoren typischerweise keine Hardwareunterstützung für den Informationsaustausch zwischen den Rechenkernen mit garantierter Latenz und Bandbreite. Deshalb stellen wir einen softwarebasierten Ansatz vor, welcher die Latenzen dynamischer Kommunikation zwischen den Rechenkernen garantiert. Das zugrundeliegende Prinzip unseres Ansatzes ist es, die Paketinjektionsrate für alle Quellen zu beschränken.

Zudem präsentieren wir ein Framework, welches die Umsetzung von sicherheitskritischen Echtzeitanwendungen auf gängigen Many-Core Prozessoren erleichtert. Das Framework passt unseren Fehlertoleranzmechanismus für jede Softwareeinheit so an, dass deren Zuverlässigkeits- und Verfügbarkeitsanforderungen mit minimalem Ressourceneinsatz erreicht werden. Ein Modell des Fehlertoleranzmechanismus als diskrete Markow-Kette ermöglicht die Berechnung der sich ergebenden Zuverlässigkeit und Verfügbarkeit jeder Softwareeinheit. Messungen auf unserem zyklen- und bitakkurat simulierten Many-Core Prozessor mit realistischer Fehlerinjektion zeigen den Vorteil unseres adaptiven Software-Fehlertoleranzmechanismus und demonstrieren den Kompromiss zwischen Ressourcenverbrauch und Fehlertoleranzniveau.

Acknowledgment

Writing this thesis would not have been possible without the support of many people. First and foremost, I want to thank Prof. Dr. Hans-Ulrich Heiß for supervising my thesis and providing insightful comments and helpful advice. I thank my co-advisors Prof. Dr. Sabine Glesner and Prof. Dr. Michael Glaß for their valuable feedback on my thesis. Special thanks go to Prof. Dr. Jan Richling, Dr. Helge Parzjegla, and Prof. Dr. Reinhard Karnapke for all the fruitful discussions we had and their helpful comments. I would like to thank all team members of the KBS research group at the TU Berlin, especially Anselm Busse and Matthias Druve, for the answers to all my questions and their support, and Dr. Matthias Diener for proof-reading this thesis.

As an industrial PhD student at the Robert Bosch GmbH, I had the opportunity to gain insights into academia and industry at the same time. I thank my supervisor Dr. Björn Saballus for his guidance on my way as a PhD student and his kind way to increase my motivation and improve the focus of my work. I would like to thank all members of the ManyCore Project and the CR/AEA department, especially Dr. Jochen Härdtlein and Dr. Dirk Ziegenbein, for giving me the opportunity to work in such an open-minded and friendly environment. Special thanks go to Dr. Rai Devendra, Dr. Dakshina Dasari, and Dr. Matthias Freier for proof-reading this thesis. I feel lucky to share my workplace with my fellow PhD students Hendrik Röhm, Alexander Biewer, Martin Lowinski, Dr. Matthias Freier, Felix Rützel, and Sören Braunstein. Thank you for the inspiring discussions about work and about life in general as well as the good times we had together.

I am grateful for having received a fellowship from the Software Campus development program, which is funded by Germany's Federal Ministry of Education and Research. This fellowship initiated a strong and fruitful collaboration with Dr. Mohammad Shadi Alhakeem and Ralphael Lisicki, both of which I owe a debt of heartfelt gratitude. Without their support, this thesis would not have been possible in the way it is today.

Last but not least, I want to thank my parents Roland and Mechthild Munk and my brother Dr. Martin Munk for paving me the way to where I am today and their continuous encouragement. These acknowledgements would be incomplete without wholeheartedly thanking my wife Sandra for her never-ending support, including proof-reading this thesis, and all the joyful moments in my life.

Contents

The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers;[a]

DIONYSIUS LARDNER, 1834, [Lar34]

[a] Note that the computers here refer to human operators.

1 Introduction

In the past, new processor generations increased performance mainly by higher clock frequencies, deeper pipelines, and larger caches [MHL+15]. This was accompanied by a rising complexity and power dissipation, which at a certain point was no longer technologically sustainable [Vaj11]. Nevertheless, the number of transistors per chip continued to increase according to Moore's law. Since the performance benefits of building larger monolithic processors declined, hardware vendors started to produce chips with multiple processor cores [Bor07].

As a result, the first *many-core* processor with more than one thousand independent processor cores has been commercially available since 2014 [PEZ12], as predicted by the 2011's International Technology Roadmap for Semiconductors[1] [The11] and several other authors [Bor07, MVdWF08, WA09, Vaj11].

At the same time, the computational demand in the embedded real-time safety-critical domain increases. Embedded systems are information processing systems embedded into an enclosing product [Mar11]. These embedded systems must often comply with real-time constraints, since their correctness depends not only on the logical correctness of the computation, but also on the physical time at which these results are produced [BW01, But11]. In many cases, real-time embedded systems are used in a safety-critical context, where a failure has catastrophic consequences on the user and the environment [ALRL04, Kop11]. Typical examples of embedded real-time safety-critical systems are vehicles, airplanes, nuclear power plants, or medical devices.

1.1 Motivation

Due to the growing complexity and rising number of supported features of embedded real-time safety-critical systems, their performance requirements continue to rise. Consider for example the automotive domain: Power-train functionalities such as combustion engine control and transmission management or electric motor control and battery management, comfort functions such as traffic sign recognition and lane keeping assistance, and safety

[1] Names used without any reference to copyrights or trademarks may still be legally protected.

features such as collision avoidance systems are becoming more and more sophisticated and complex. The development of such advanced driver assistance systems (ADAS) is propelled by the vision of a highly automated driving (HAD) vehicle. The goal is to reach an autonomous driving level and completely control the vehicle without any human interaction or responsibility [Bun12, Nat13].

Driver-less cars are not only attractive for millions of commuters world-wide, but also promise attractive business cases. Figure 1.1 shows an HAD prototype vehicle of the Robert Bosch GmbH, which is based on Tesla's Model S. The HAD functionality is implemented on a computer installed in the trunk of the car, as shown in Figure 1.1a. The computer contains a commercial off-the-shelf (COTS) processor with 16 cores and 256 GB main memory. Only few fault-tolerance mechanisms have been included in the prototypical implementation, since a test driver is always supervising the vehicle as shown in Figure 1.1b.

In order to leave the prototypical state, the spatial dimensions of the computer in the trunk have to be reduced. Many-core processors potentially offer the computational power to execute software that implements applications like HAD within given real-time constraints. For this reason, many-core processors are not only attractive in the high performance computing and consumer electronic domain, but are also gaining more and more attention in the safety-critical real-time embedded domain [AEF+14, RCM14, SEH+12, MTK+11, RPM+15, SEU+15].

(a) The trunk of the HAD prototype contains the main computer and measurement equipment [TDa].

(b) The HAD prototype drives autonomously on a highway and is supervised by a test driver [TDb].

Figure 1.1: An HAD prototype vehicle based on Tesla's Model S.

1.2 Problem Statement

The rising number of transistors per processor is the result of higher integration rates and smaller feature sizes. The drawback of such highly integrated many-core processors is their susceptibility to soft errors caused by high-energy particle strikes [Bor05, Bau05, Nic11]. Unless countermeasures are in place, soft errors can result in bit flips and eventually jeopardize the correctness of the system. Due to their physical nature, we assume soft errors to be stochastically independent and to occur with a constant rate. In contrast to permanent errors, soft errors are of transient nature and can be repaired, e. g., by re-execution or reloading the affected memory cell. In this thesis, we only consider soft errors since the probability of a permanent error is at least three orders of magnitude lower than the probability of a soft error during the useful operation phase of a processor [Bau05, Nic11].

Due to the safety-critical nature of most embedded real-time systems, they have to be developed carefully and qualified to certain dependability standards, for instance the IEC 61508 [Int10] and its derivatives, e. g., the ISO 26262 [Int11] for the automotive industry. Consequently, appropriate fault-tolerance mechanisms are required to achieve the specified system dependability measures, e. g., the reliability and availability. Fault tolerance is the ability of a system to continue providing the correct service in presence of faults and soft errors, respectively [Joh84]. This ability is necessary for example in HAD, where a safe state cannot always be reached immediately as soon as an error is detected.

So far, most fault-tolerance mechanisms are implemented in hardware, e. g., error-correcting code (ECC) protection for memories or triple modular redundancy (TMR) setups [Sor09, KK10]. On the one hand, these hardware countermeasures increase the chip area or the number of devices. Thus, hardware countermeasures increase the unit costs and are therefore not common in consumer-grade many-core processors [GRSRV06]. On the other hand, these COTS many-core processors are especially attractive in cost-driven mass markets such as the automotive industry [GRSRV06].

In this thesis, we consider COTS consumer-grade many-core processors. These processors typically contain homogeneous cores connected by multiple Networks-on-Chip (NoCs). Each core comprises an independent central processing unit (CPU) and a core-local memory protected by a memory protection unit (MPU). Each wormhole-switching NoC with 2D mesh topology implements the dimension-order (XY) routing policy with round-robin early access arbitration. Furthermore, we assume a reliable off-chip memory to be connected to the many-core processor.

Software-implemented hardware fault-tolerance approaches do not share the drawback of high unit costs, since they leverage the computational performance of COTS consumer-grade processors [GRSRV06, KK10]. Additionally, software-based mechanisms can be applied exclusively to the safety-critical software entities, i. e., tasks. In typical automotive software applications, only a subset of tasks has safety requirements. For example, the failure of a task that calculates the brake force potentially leads to an hazardous event, while the failure of a task that merely protocols system variables is not considered critical. Hence, safety requirements are only allocated to the former task. Any protection of the latter task is superfluous and only increases costs. For this reason, software-based fault-tolerance approaches that support such *mixed-critical* task sets are advantageous.

Several software-based fault-tolerance approaches have been presented in literature. For single-core processors, a common fault-tolerance approach is based on compiler optimizations. These compiler-based solutions replicate data and instructions and add comparison routines [OSM02, RCA07, RKSH14]. However, compiler-based solutions increase the execution time, which can render it difficult to apply them under real-time constraints. The problem of combining real-time and fault-tolerance requirements is considered by several scheduling algorithms, which mostly reserve re-execution slots in case of a task failure [Kri14]. However, most of these scheduling approaches assume a reliable fault detection mechanism as well as a fault-tolerant scheduling policy implementation and task dispatcher in the operating system (OS). These assumptions are difficult to implement on COTS processors, since all homogeneous cores and hence all software components are affected by soft errors.

1.3 Research Questions

The main research question that we are concerned with in this thesis is:

> How, to which extent, and with which overhead can the fault-tolerance requirements of mixed-critical task sets with real-time constraints on consumer-grade many-core processor be achieved using only software means?

From this main research question, we deduce the following subordinated research questions: Due to the real-time constraints of the application, the system has to meet its implied deadlines at all times, even under the presence of errors. This especially affects the parallel scheduling of replicated tasks as well as the inter-task communication between the cores of the processor. However, most COTS consumer-grade many-core processors contain a NoC that does not provide bandwidth and latency guarantees in hardware.

> How to increase the system's fault-tolerance level while guaranteeing the real-time constraints without modifying the hardware?

As mentioned in the previous section, most software fault-tolerance mechanisms proposed in literature assume a fault-free comparison unit or a reliable OS kernel. However, all cores of a COTS many-core processor and hence all software entities including the OS kernel are affected by soft errors. Therefore, the proposed solution has to consider failures of the comparison unit and the OS kernel as well.

> How to design a fault-tolerance mechanism only from software components that potentially fail?

The software fault-tolerance mechanism has to guarantee the specific fault-tolerance requirement of each individual task of a mixed-critical task set. In order to achieve each task's fault-tolerance requirement with the minimum overhead in terms of resources and system load, the mechanism needs to be adaptable.

> How to determine the fault-tolerance level achieved by a specific adaptation of the software fault-tolerance mechanism at design time?

1.4 Proposed Solution

In order to answer the research questions identified in the previous section, we require an artifact whose properties can be measured and evaluated. For this reason, we propose an adaptive software-implemented hardware fault-tolerance mechanism that is based on the well-known *N modular redundancy (NMR)* principle [Bir14, Kop11, KK10]. Under NMR, a functional entity is replicated N times and a comparison unit, i. e., a voter, tallies the replica results. The large number of independent cores of COTS many-core processors offers to implement NMR in software by assigning replicated tasks to separate cores. Since the voter represents a *single point of failure (SPOF)*, NMR is only beneficial if the voter is reliable. However, as mentioned above, all cores and thus all software entities are affected by soft errors. Therefore, we propose a voting scheme with two fail-silent voters that increases the fault-tolerance of the voter without any hardware modification. Each fail-silent voter either provides the correct result or it remains silent and forwards no result. In order to achieve this fail-silent behavior, we extend the encoded voting approach of Ulbrich et al. [UHK+12]. Figure 1.2 presents a schematic overview of the NMR mechanism on a processor with 3×3 cores.

The fault-tolerance mechanism must respect the real-time requirements of the system. The implied deadlines of the task set are adjusted such that well-known scheduling policies for single-core processors can be employed to guarantee all real-time constraints. In order to guarantee the real-time constraints of the dynamic inter-core communication between the task replicas and the voter via the many-core processor's interconnect fabric, we propose a software-based mechanism that limits the packet injection rate to achieve a bounded latency of the dynamic communication.

Due to the transient nature of soft errors, a software-based repair mechanism allows to regain the functionality of a previously failed task replica. In this thesis, we present a repair mechanism based on spare replicas that preserves the state of the protected task.

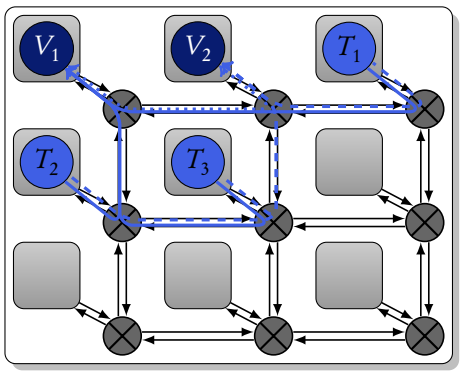


Figure 1.2: Schematic overview of the basic NMR mechanism for one task on a 3×3 many-core processor. Task T is replicated $N = 3$ times and forwards its result to both fail-silent voters V_1 and V_2, which compare the results and check each other. Solid lines represent messages sent from the task replicas to voter V_1, dashed lines represent messages sent from the task replicas to voter V_2, and dotted lines are messages that are exchanged between both voters.

The number of replicas and spares can be adapted for each safety-critical task. We propose a framework that allows the software developer to specify the fault-tolerance requirements of each individual task of a mixed-critical hard real-time application. In this thesis, we use reliability and availability as fault-tolerance measures. The framework employs a mathematical model to determine the resulting reliability and availability for a given number of replicas and spares. Hence, it is able to adapt the software fault-tolerance mechanism such that the target reliability and availability are achieved with a minimum of resources.

1.5 Main Contributions

The primary contributions of this thesis to the state-of-the-art are as follows:

- A software-implemented mechanism that guarantees the latency of dynamic inter-core communication on 2D mesh wormhole-switching NoCs, which implement the dimension-order (XY) routing policy with round-robin early access arbitration and thus support only best-effort communication. The main principle behind the proposed mechanism is a common limited packet injection rate for all sources. In contrast to related approaches, which derive the worst-case latencies of a static traffic pattern, our solution supports dynamic guaranteed service (GS) communication that changes at runtime, e. g., due to the reaction to a component failure. This work is published in [MFRC15].
- An adaptive software fault-tolerance mechanism that increases the reliability and availability of a task on a COTS many-core processor with homogeneous cores. The mechanism is based on NMR to leverage the spatial redundancy of the processor. The fault-tolerance level of the voter as a SPOF in an NMR setup is increased by employing two fail-silent majority voters that check and repair each other. To achieve the fail-silent behavior, we extend the encoded voting approach of Ulbrich et al. [UHK+12]. A state-preserving repair procedure leverages the reliable off-chip memory to recover from task and voter failures caused by soft errors. Unlike other software-based hardware fault-tolerance approaches, the proposed mechanism guarantees the real-time requirements of the task and considers failures in the underlying OS kernel. Experiments on a cycle-accurate and bit-accurate (CABA) many-core processor simulator show that the proposed fault-tolerance mechanism is able to increase the reliability of a task by a factor of up to 2.22 compared to an unprotected execution. The adaptive software fault-tolerance mechanism is published in [AML+15, MAL+15, MAL+16].
- A fault-tolerance analysis algorithm that computes the reliability and availability achieved by a specific configuration of our adaptive software fault-tolerance mechanism. The algorithm employs a discrete time Markov chain (DTMC) model of the fault-tolerance mechanism, which is calibrated with measurements conducted on our CABA many-core processor simulator. Compared to related fault-tolerance analyses, our model is based on a physically accurate fault model. The DTMC model of our software fault-tolerance mechanism is published in [MAL+16].

- A software fault-tolerance framework that eases the development of embedded real-time safety-critical applications of COTS consumer-grade many-core processors by providing interfaces to specify a mixed-critical task set, its safety and real-time requirements, as well as the processor's properties. Our framework uses the fault-tolerance analysis algorithm to adapt our software-based hardware fault-tolerance mechanism for each task of a mixed-critical task set such that the task's reliability and availability requirements are met with minimum resource usage. An example derived from a real-world application demonstrates the functionality of our framework. The software fault-tolerance framework is published in [AML+15, MAL+15].

1.6 Context

This work has been performed in the context of the CR/SP16-008 "ManyCore" research project in the Corporate Sector Research and Advance Engineering of the Robert Bosch GmbH. The Robert Bosch GmbH is a leading global supplier of technology and services in the automotive industry. The company actively develops HAD as a profitable future product. The goal of the CR/SP16-008 "ManyCore" research project is to enable the usage of consumer-electronic many-core processors in the automotive domain. The work described in this thesis represents an important step to achieve this goal from the fault-tolerance point of view.

1.7 Outline

The remainder of this thesis is organized as follows:

Chapter 2 introduces the fundamental background of this thesis. First, we focus on the properties of embedded real-time safety-critical systems and their properties. Next, we specify many-core processors and their hardware features and limitations. Details of the NoC as the inter-core communication fabric and our fault hypothesis are included as well. Afterwards, we introduce and combine important real-time and many-core OS concepts.

Chapter 3 discusses the related work of this thesis. This chapter contains two main sections. In the first section, we present hardware-based solutions and mathematical approaches to bound the latency of communication on a NoC. In the second section, we survey fault-tolerance mechanisms implemented in hardware and then focus on different software-based approaches.

Chapter 4 presents our approach to guarantee the latency of dynamic communication on a NoC. The underlying idea is to limit the packet injection rate in software in order to limit the upper bound of delays experienced by each packet in the NoC.

Chapter 5 introduces our software fault-tolerance mechanism. We justify the choice of NMR as foundation of the proposed mechanism and present the encoding mechanism that is used to design fail-silent voters in software and the repair mechanism that employs spare replicas. Finally, we explain how the proposed fault-tolerance mechanism is integrated with the real-time many-core OS.

Chapter 6 presents our software fault-tolerance framework. We first provide the mathematical background of DTMCs. Afterwards, we present the DTMC model of our software

fault-tolerance mechanism. The presented software fault-tolerance framework analyzes the DTMC model to adapt the mechanism such that the reliability and availability target of each task is achieved with minimum resource usage.

Chapter 7 experimentally evaluates our approaches. We use a simulated many-core processor to experimentally verify that a limited packet injection rate can guarantee a latency bound for dynamic communication. With the help of a fault injection mechanism, we measure the reliability and availability that is achieved by our software fault-tolerance mechanism. Finally, we compare the results of the DTMC model with the measured reliability and availability values.

Chapter 8 summarizes our conclusions and gives an outlook on possible directions for future work.

2 Fundamentals

This chapter introduces the background terminology and fundamental concepts used in this thesis. It also establishes a common notation by defining important terms.

First, we present embedded real-time safety-critical systems, which are in the center of attention of this thesis and whose properties, requirements, and constraints have to be carefully considered when selecting or designing the hardware architecture and developing the application software. Next, we specify the hardware architecture of many-core processors and enumerate its features and limitations. This includes details about the NoC, i. e., the inter-core communication infrastructure common to many-core processors, as well as our fault hypothesis. Finally, we combine real-time OS concepts with many-core OS designs to manage the considered system.

2.1 Embedded Systems

The scientific inventions and technological advancement in the past decades have lead to ubiquitous electronic and computing devices that influence the daily lives of human beings. These devices are trusted to control important and safety-critical products, e. g., a pace maker that monitors and controls heartbeats, an industrial robot that builds products next to or in cooperation with human workers, or an HAD vehicle. The electronic computing devices or systems that control these products are an indivisible part of the product itself and are therefore called *embedded systems*. In other words, an embedded system is an "information processing system embedded into enclosing products" [Mar11, p. xiii]. In this thesis, we focus on embedded systems.

Embedded systems usually need to conform various constraints including size and weight limits, power consumption, as well as real-time and safety-critical requirements, while meeting tight cost budgets [Koo96]. Size and weight limits as well as power consumption are out of scope of this thesis, since they are considered less critical for an HAD vehicle that serves as an example system through this thesis. In the following, we first consider the real-time constraints and afterwards the safety-critical requirements of embedded systems in detail.

2.1.1 Real-Time Systems

In this section, we summarize the basic concepts and models of real-time systems. Unless stated otherwise, this summary is based on Buttazzo's book on hard real-time computing systems [But11].

A *real-time system* is defined as "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay" [BW01, p. 13]. Hence, the correctness of a real-time system depends not only on the logical correctness of the computation, but also on the physical time at which these results are produced [BW01, But11, Kop11].

The functionality of a real-time system is implemented in software that controls the hardware and connected peripherals such as sensors and actuators. The software is typically divided into several encapsulated modules, so called *tasks* $T_i \in \mathcal{T}$. "A task is a software entity or program intended to process some specific input or to respond in a specific manner to events conveyed to it" [Nis97, p. 180]. In other words, a task is an abstract unit of scheduling that models a single-threaded *process* in the context of the OS, which is executed in a sequential fashion. Note that in this thesis, we use the term process when referring to a program and its abstraction in the context of an OS. Furthermore, we use the term task when referring to the abstract notion of a process in the context of real-time scheduling.

A task $T_i \in \mathcal{T}$ becomes *active* as soon as it can potentially execute on the processor. Active tasks are kept in the *ready queue* by the OS and the *scheduler*, respectively. The scheduler decides which active task is *running*, i. e., executed on the processor. The decision follows a specific *scheduling policy* that is implemented by the *scheduling algorithm*, i. e., a set of rules that determines the task execution order. The procedure of allocating the selected task to the processor is referred to as *dispatching*.

If the scheduler supports *preemption*, it can suspend the currently running task and insert it into the ready queue when a more urgent task becomes active, such that the more urgent task can be dispatched immediately. The process of exchanging the running task is called *context switch*. A *schedule* denotes the assignment of tasks to the processor, such that each task is executed until completion. A schedule is *feasible* if all tasks in the task set $\mathcal{T}$ can be completed according to a set of specified timing constraints. A set of tasks $\mathcal{T}$ is said to be *schedulable* if there exists at least one scheduling algorithm that can produce a feasible schedule.

The timing constraints of the real-time system have to be met by the software and the executed tasks, respectively. Therefore, each task $T_i \in \mathcal{T}$ is assigned a *relative deadline* D_i that specifies the time by which the execution of the task must have finished with respect to its activation time. The consequence of failing to provide a correct result within a given deadline depends on the real-time system and its environment. Therefore, deadlines are commonly differentiated in *hard*, *firm*, and *soft*:

Hard deadline. If a task misses a hard deadline, the consequences are hazardous and can lead to the loss of human lives. A typical example of a system with hard deadlines is an airbag electronic control unit (ECU).

Firm deadline. A firm deadline is allowed to be missed every once in a while, but the result of a task is useless for the system after the deadline. An example of a system with firm deadlines is the combustion engine ECU, where the physical inertia of the engine forgives one missed injection.

Soft deadline. If a task misses a soft deadline, the usefulness of a result decreases with the delay after the deadline, but no severe consequences are to be feared. A typical example of a system with soft deadlines is an video playback device, where a delayed frame is unpleasant but acceptable.

In this thesis, we consider tasks with hard deadlines.

Each task becomes active either in a *periodic* or an *aperiodic* manner. The activation of a periodic task $T_i \in \mathscr{T}$ is repeated with a fixed time interval, the *period* P_i. For example, a periodic task is activated with a specific sampling rate for a sensor signal. In contrast, aperiodic tasks are activated at a priori unknown times, e. g., triggered by an interrupt from an external sensor. An aperiodic task whose consecutive activations are separated by a minimum inter-arrival time is called a *sporadic* task. There is a large body of real-time scheduling theory that allows to include sporadic tasks in periodic schedules, e. g., the *Polling Server* or the *Deferrable Server*. For this reason, we consider periodic tasks in this thesis.

Current automotive software contains tasks with harmonic periods, i. e., the period of each task is an integer multiple of the period of any other task, so $\forall T_i, T_j \in \mathscr{T} : P_i = kP_j,\ k \in \mathbb{N}^+$ [KZH15]. The *hyperperiod* $\mathring{P}$ is the minimum interval of time after which a schedule repeats itself. The advantage of harmonic periods is that the hyperperiod $\mathring{P}_H = \max_{\forall T_i \in \mathscr{T}} P_i$ is generally smaller than the hyperperiod of unconstrained task sets $\mathring{P} = \operatorname{lcm}\left(P_i, P_j\right), \forall T_i, T_j \in \mathscr{T}$ if synchronously activated. We consider task sets with harmonic periods in this thesis.

The relation between a task's relative deadline and its period is categorized as follows [SLL07]:

- *implicit* if $D_i = P_i$,
- *constrained* if $D_i \leq P_i$, and
- *arbitrary* if there is no constraint between D_i and P_i.

In this thesis, we consider tasks with constrained deadlines, since they are most common in automotive software.

The j^{th} activation of a task T_i is called *job* $T_{i,j}$. In other words, a periodic task releases an infinite sequence of jobs. The job $T_{i,j}$ becomes active at time $a_{i,j} = O_i + j \cdot P_i$, started by the scheduler at time $s_{i,j}$, and completes execution at time $c_{i,j}$. The first activation of a task $T_i \in \mathscr{T}$ can be delayed by a phase or *offset* O_i.

In order for a task to fulfill its real-time constraints, the completion time of each job has to be before its absolute deadline, so $c_{i,j} \leq a_{i,j} + D_i,\ \forall j \in \mathbb{N}^+$. An overview of the task execution over time is given in Figure 2.1 on the following page.

The *worst-case execution time (WCET)* $\hat{E}_i$ of a task $T_i \in \mathscr{T}$ denotes the upper bound of the duration of the task's execution without preemption, i. e., the maximum execution time of all jobs $T_{i,j}$ [WEE+08]. For simple processor architectures, a safe upper bound of this theoretical value can be estimated by commercial static timing analysis tools such as Absint's aiT [HF13] ,Tidorum's Bound-T [Tid13], and Rapita System's RapiTime [Rap13].

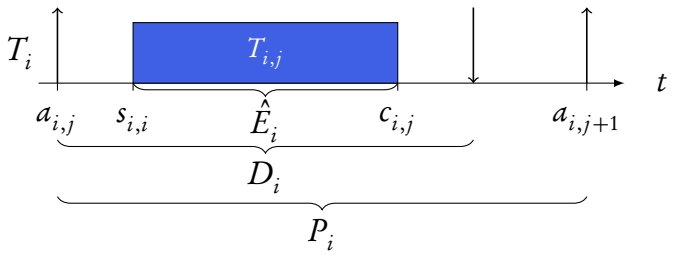


Figure 2.1: Temporal parameters of the job $T_{i,j}$ of task T_i.

The *scheduling analysis* is a procedure to verify that the schedule of a given task set $\mathscr{T}$ is feasible. Each scheduling policy and its respective scheduling algorithm needs a customized scheduling analysis. Typically, the scheduling analysis method determines the *worst-case response time (WCRT)* $\hat{R}_i$ of each task $T_i \in \mathscr{T}$. The WCRT denotes the maximum time for executing a task considering the worst-case interference with the other tasks in the task set, so $\hat{R}_i = \max_{\forall j \in \mathbb{N}^+} c_{i,j} - a_{i,j}$. If the WCRT of each task $T_i \in \mathscr{T}$ is smaller than its relative deadline , $\hat{R}_i < D_i$, the schedule is feasible.

A plethora of scheduling policies for real-time systems is proposed in literature. The policies can be classified by the following properties:

offline / online. Offline scheduling policies determine the complete schedule statically before runtime. They are also known as cyclic or static scheduling policies. Typical examples are list scheduling and time-triggered scheduling.

An online scheduling policy provides a scheduling algorithm that decides at runtime which task is executed next. They are also known as dynamic scheduling policies. Prominent examples are fixed-priority or earliest deadline first (EDF) scheduling.

static / dynamic priority assignment. Online scheduling policies generally select the task to be executed next depending on its priority. Under a static priority assignment scheme, the priority of each task is determined statically before runtime. Typical examples are rate monotonic and deadline monotonic priority assignment schemes. Under these schemes, priorities are selected inversely proportional to a task's period (rate) and absolute deadline, respectively, such that the task with the shortest period and closest deadline, respectively, has the highest priority.

In a dynamic priority assignment scheme, the task's priority is determined at runtime. A common dynamic priority assignment scheme is included in the EDF scheduling policy, where the task with the closest absolute deadline is assigned the highest priority.

preemptive / non-preemptive. A preemptive scheduling policy pauses the execution of a running task if a higher-priority task becomes active.

A non-preemptive scheduling policy always finishes the execution of the currently running task before starting the next one. Hence, non-preemptive scheduling policies require less context switches in general.

partitioned / global. This classification is only applicable for systems with multiple processing units, e. g., many-core processors. A partitioned scheduling policy employs a local scheduling algorithm per processing unit and core, respectively. Tasks are *mapped* to one core statically and cannot be executed on another core. Note that the procedure of task-to-core mapping is also known as task *assignment* or *binding*.

A global scheduling policy uses a global ready queue and does not only select which tasks are executed next, but also on which cores. For this reason, global scheduling policies require a task migration mechanism that allows to dynamically map tasks to different cores.

Note that there are also semi-partitioned or hybrid solutions, where tasks can be scheduled only on a subset of cores.

Offline scheduling policies are less flexible to react to environmental changes such as tasks that are added at runtime. Additionally, their real-time guarantees strongly rely on the hypotheses and assumptions of the system that are derived before runtime, e. g., the WCETs. Therefore, online scheduling policies are commonly used by the automotive industry.

Amongst the online scheduling policies, EDF is proven to be *optimal* in the sense of feasibility, i. e., if there exists a feasible schedule for a given task set $\mathcal{T}$, then EDF is able to find it. However, its scheduling algorithm is more complex and more difficult to implement compared to fixed-priority scheduling policies. Additionally, it exhibits a domino effect once a task misses its deadline. For this reason, fixed-priority scheduling policies are generally preferred to EDF by the automotive industry [AUT15].

Global scheduling policies require task migrations, which introduce runtime overhead. As a result, partitioned scheduling policies are preferred by the automotive industry.

Since we focus on the automotive domain in this thesis, we assume an *online partitioned fixed-priority preemptive scheduling policy* with a *deadline-monotonic priority assignment scheme*. Under this scheduling policy, the WCRT of each task $T_i \in \mathcal{T}$ can be calculated by the fixed-point iteration method of Audsley et al. [ABR+93]:

$$\hat{R}_i^{n+1} = \hat{E}_i + \sum_{\forall D_j \leq D_i} \left\lceil \frac{\hat{R}_i^n}{P_j} \right\rceil \hat{E}_j. \tag{2.1}$$

Inter-Task Communication

In a realistic real-time system such as an HAD vehicle, tasks exchange information and communicate with each other. In this thesis, we denote the information that is exchanged between tasks as a *message* $m_i \in \mathcal{M}$. Each message $m_i \in \mathcal{M}$ has one source task $\sigma : \mathcal{M} \rightarrow \mathcal{T}$ and one destination task $\delta : \mathcal{M} \rightarrow \mathcal{T}$. The size of a message m_i is denoted as $|m_i|$.

In general, inter-task communication is influenced by the schedule, i. e., the task execution order. For example, consider two tasks T_1 and T_2 which become active at the same time. Task T_1 has the higher priority and receives a message from task T_2. Under the fixed-priority scheduling policy, task T_1 is executed first and receives either an old message from the previous execution of task T_2 or no message at all. Whether task T_1 is able to

provide its functionality with an old message depends on the application. The fact that the application requires a new message, so task T_2 has to be scheduled before task T_1, is called a *precedence constraint*.

A system behaves in a *logically deterministic* manner "if, given an initial state and a set of ordered inputs, then the subsequent states and the values of subsequent outputs are entailed" [Kop11, p. 125]. A deterministic behavior of a system is beneficial to test its components, to mask a faulty channel, and helps to understand the system in general [Kop11]. In case the precedence constraints are not exactly specified and tasks read and write messages at arbitrary times, the system behaves non-deterministically due to the varying runtimes of jobs.

In order to enforce a deterministic behavior, we assume that the communication follows the concept of *logical execution time (LET)* [HHK03, KS12]. For any task $T_i \in \mathscr{T}$, the beginning of the LET of the job $T_{i,j}$ is equal to its activation time $a_{i,j}$. The end of the LET of the job $T_{i,j}$ is equal to its absolute deadline $a_{i,j} + D_i$. All communication and message transmission happens logically instantaneous at the beginning and the end of the LETs. Thus, the actual execution time of a job within the LET does not influence the inter-task communication. In other words, if the job finishes execution before its absolute deadline, writing its output is delayed until its deadline and the end of its LET, respectively.

Consider any message $m_i \in \mathscr{M}$. Let $T_i = \sigma(m_i)$ denote the sending task and let $T_k = \delta(m_i)$ denote the receiving task. From the implementation point of view, the message passing service has to ensure that the message m_i is transmitted between the completion $c_{i,j}$ of the sending job $T_{i,j}$ and before the start $s_{k,l}$ of the receiving job $T_{k,l}$ that is activated at the same time or after the LET sending job, so $a_{k,l} \geq a_{i,j} + D_i$.

We assume a *last-is-best semantic* for all communication, i. e., only the last message is relevant for the receiving task. This property allows to reduce the amount of messages in general: If the sending task T_i has a smaller period than the receiving task T_k, i. e., $P_i < P_k$, then only the last job $T_{i,j}$ within the period of the receiving task P_k needs to send a message.

The latest possibility for the job $T_{i,j}$ to send a message is its absolute worst-case response time $a_{i,j} + \hat{R}_i$. The earliest possibility for the job $T_{k,l}$ to receive a message is the best-case start time, i. e., its activation time $a_{k,l}$. Since both times are known a priori, the *worst-case communication time (WCCT)* $\hat{C}_i$ of any message $m_i \in \mathscr{M}$ can be statically determined before runtime by $\hat{C}_i = a_{k,l} - (a_{i,j} + \hat{R}_k)$. This WCCT $\hat{C}_i$ of each message $m_i \in \mathscr{M}$ has to be guaranteed by the message passing service and the hardware platform, respectively.

Figure 2.2 on the next page presents an example schedule of two tasks T_1 and T_2 that exchange messages following the LET concept and the last-is-best semantic. Message m_2 is transmitted logically instantaneously from job $T_{2,0}$ to job $T_{1,2}$ (and not job $T_{1,1}$) at time $a_{2,0} + D_2$. The WCCT of message m_2 is $\hat{C}_2 = a_{1,2} - (a_{2,0} + \hat{R}_2)$. Note that in a feasible schedule, the maximum possible WCRT equals the deadline, so $\hat{R}_2 \leq D_2$, and Figure 2.2 only shows the absolute deadline of job $T_{2,0}$. The second job of this task $T_{2,1}$ has a longer execution time and is preempted twice by job $T_{1,2}$ and job $T_{1,3}$. However, the communication is still deterministic due to the LET concept.

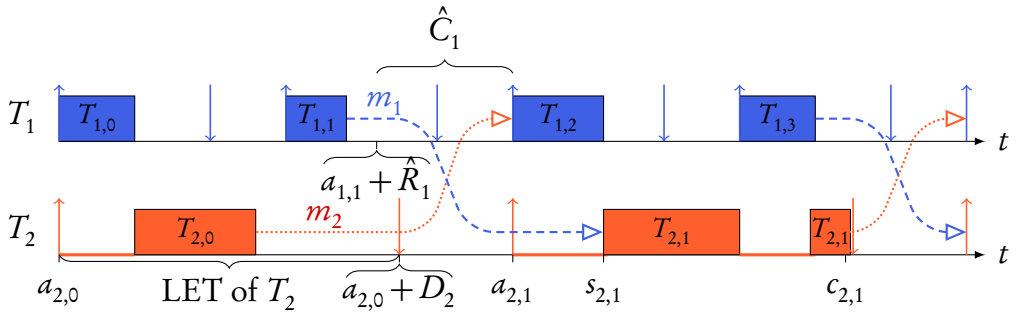


Figure 2.2: A schedule extract of two tasks T_1 (blue) and T_2 (red) that exchange messages according to the LET concept. Task T_1 has half the period and a shorter deadline than task T_2. According to the deadline monotonic priority assignment scheme, task T_1 has the higher priority. Both tasks are executed on the same core, so task T_1 preempts task T_2 (preemption time is shown as red line). Note that the actual, varying runtimes are shown.

In the example shown in Figure 2.2, message m_1 is transmitted logically instantaneously at $a_{1,1}+D_1$ from job $T_{1,1}$ to job $T_{2,1}$. The WCCT of message m_1 is $\hat{C}_1 = a_{2,1} - (a_{1,1}+\hat{R}_1) = 1 \cdot P_2 - (1 \cdot P_1 + \hat{R}_1)$. The preemption of job $T_{2,1}$ leads to a later start time, so $s_{2,1} > a_{2,1}$. Therefore, the actual communication time in this period is larger than the WCCT.

Note that communication between tasks at arbitrary times might be faster compared to the LET concept, especially on a single-core processor. However, as soon as tasks are executed on separate cores, the delay of communication via global variables is non-deterministic and depends on the runtimes of higher-priority tasks, while it is exactly the same delay as on a single core under the LET concept.

Apart from inter-task communication, tasks also communicate with external resources to read sensor values or write actuator signals. Following the LET concept, we assume that all required sensor signals are available at the beginning of the job's LET. Similarly, the actuator signals are forwarded only at the end of the job's LET. This functionality is implemented by so-called *drivers* within the OS, which are described in detail in Section 2.3.3.

2.1.2 Safety-Critical Systems

In this section, we introduce safety-critical systems and define common measures of their properties.

A *safety-critical system* has to be *dependable* [Mar11], i. e., it has "to avoid service failures that are more frequent and more severe than is acceptable" [p. 13][ALRL04]. Note that in contrast to the real-time system definition, service failures here include incorrect results with a correct timing, i. e., produced before the deadline. Typical examples of safety-critical systems are nuclear reactors, medical devices, and vehicles.

The definition of dependability is based on the term *service failure*, abbreviated failure hereafter. A failure occurs when the service delivered by the system deviates from the *intended service* of the system [ALRL04, Kop11]. For example, an HAD vehicle failed if it

does not response to changes in the environment anymore. Another example of a failure is storing or forwarding an incorrect result, which is also known as *silent data corruption (SDC)* [Kop11]. Note that the intended service is the service documented in the system's specification, which does not necessarily represent the *correct service* [Kop11].

The failure is the result of an unintended state within the system, which is called *error* [Kop11]. Errors that are present in the system but not detected are *latent* errors [ALRL04]. Typical examples of errors are bit flips in memory cells or a lost connection between two components.

An error is caused by an adverse phenomenon, which is called a *fault* [Kop11]. Faults can be internal or external of a system [ALRL04]. Examples of internal faults are wear-out effects such as electromigration [Sor09]. An example of an external fault is a high-energy particle strike [Bau05]. A fault is active when it causes an error, otherwise it is dormant [ALRL04].

The duration of faults is classified as permanent, transient, or intermittent [KK10, Sor09]. A *permanent* fault is persistent in time, e. g., the result of a wear-out effect or a systematic fault such as a hardware design flaw or a software bug. A *transient* fault is present only for a bounded time interval, e. g., the high-energy particle strike. An *intermittent* fault oscillates between active and dormant, so it occurs sporadically in a bursty manner, e. g., due to a loose connection. The manifestation of a transient fault is also known as *soft error* [Sor09]. Similarly, the effects of permanent faults are known as *hard errors* [Sor09].

The notion of fault, error, and failure always depends on the point of view and the system boundaries. The effects of faults, errors, and failures influence the dependability of a system. The following measures and metrics are used to evaluate the dependability of a system [ALRL04]:

Reliability is the probability of a system to provide the correct service until the *mission time* t_m, given that the system was operational at the time t_0 [KK10]. Let $\mathscr{S}$ denote the lifetime of the system (the time until the system fails) that is operational at $t = 0$ and let $F(t)$ denote the cumulative distribution function of $\mathscr{S}$ [KK10]. Then [KK10],

$$R(t) = \Pr\{\mathscr{S} > t\} = 1 - F(t). \tag{2.2}$$

Quantitatively, reliability specifies the probability that no operational interruptions will occur during a stated time interval [Bir14]. This does not mean that redundant parts may not fail, such parts can fail and be repaired without operational interruption at system level [Bir14].

The *failure rate* $\Lambda(t)$ is the conditional probability that a system will fail at time t when it provided the correct service at least until time t [KK10]. As a conditional probability, the failure rate $\Lambda(t)$ of the system can be calculated as [KK10]

$$\Lambda(t) = \frac{f(t)}{1 - F(t)}, \tag{2.3}$$

where $f(t)$ is the probability density function of the system's lifetime $\mathscr{S}$, so $f(t) = \frac{dF(t)}{dt}$ [KK10]. Thus, the reliability $R(t)$ of the system can be expressed by the differential equation [KK10]

$$\frac{dR(t)}{dt} = -\Lambda(t)R(t). \tag{2.4}$$

For a failure rate $\Lambda(t) = \Lambda$ and $t_0 = 0$, the solution of Equation (2.4) is

$$R(t_m) = e^{-\Lambda t_m}, \tag{2.5}$$

which defines the reliability after a mission time t_m [Bir14].

Safety denotes the absence of catastrophic consequences on the user(s) and the environment [ALRL04]. Examples of catastrophic consequences, also known as *malign* failures, are a meltdown in a nuclear power plant, the loss of human life caused by failed medical equipment, or an accident caused by an HAD vehicle. In many cases, the safety-critical system has to be developed according to safety standards, e. g., the IEC 61508 [Int10] and the ISO 26262 [Int11] in the automotive industry. These safety standards specify a hazard and risk analysis to deduce the safety level of a system and its functions. This analysis includes an assessment of the severity of the consequences and likelihood of a failure. Hence, safety can be expressed as reliability regarding malign failures [Kop11]. In other words, safety can be considered as an extension of reliability with respect to failures that may create safety hazards [Dub13]. Consequently, increasing the reliability also increases the safety, since less failures occur in the system. However, while reliability assurance deals with measure to minimize the total number of failures, safety assurance examines measures which allow an item to be brought into a safe state in the case of failure [Bir14]. In consequence, increasing the safety can reduce the reliability, e. g., if a system is brought into safe state by shutting it down when a component failure is detected, although it is unclear whether the component failure will cause a system failure.

Maintainability is defined as the ability of a system to undergo modifications and repairs [ALRL04]. These modification and repairs can be carried out either in a preventive manner at predetermined intervals in order to reduce wearout failures, or in a corrective manner after a failure has be recognized [Bir14]. In this thesis, we focus on corrective maintenance, referred to as maintenance in the following. Maintainability is expressed by the probability that a repair of the system or parts of it will be performed within a stated time interval after the occurrence of a non-catastrophic or *benign* failure [Bir14].

Availability is a broad term that expresses the ratio of delivered to expected service of a system [Bir14]. In contrast to reliability, which does not allow any failure of the system, availability considers system failures and the continuance of the system after a repair. The point availability $A_P(t)$ denotes the probability that the system provides the correct service at a stated instant of time t [Bir14], so

$$A_P(t) = \Pr\{\text{correct service at } t | \text{new at } t = 0\}. \tag{2.6}$$

The average availability $A(t_m)$ is defined as the expected proportion of time in which system provides the correct service in $(t_0, t_m]$ given a new system at t_0 [Bir14, Dub13], so

$$A(t_m) = \frac{1}{t_m} \int_{t_0}^{t_m} A_P(t)\, dt. \tag{2.7}$$

The long-term availability is the stationary or steady-state of the point availability or average availability [KK10, Bir14]. This is the common meaning of the term availability. The long-term availability is defined as

$$A_L = \lim_{t \to \infty} A_P(t) = \lim_{t_m \to \infty} A(t_m). \tag{2.8}$$

However, Equation (2.8) only holds if fault coverage, switching, and logistic support are ideal [Bir14]. This means that any failure of a system component is detected and a maintenance mechanism is always started. As we show in Section 5.2 on page 66, this assumption does not hold for our realistic fault model. Therefore, in this thesis we consider the average availability $A(t_m)$, referred to as availability hereafter.

In this thesis, we focus on the system's reliability and availability, since both of them have positive impact on the safety measure of a system. We also consider corrective maintenance in terms of a software implemented repair process.

Fault Tolerance

In case a failure occurs in a safety-critical system, a *safe state* that is identified a priori has to be reached. If this safe state can quickly be reached, the system is called *fail-safe* [Kop11]. For example, a typical safe state of an industrial robot is to stop moving and power off. Hence, detecting a system failure is sufficient for fail-safe systems.

However, there are environments where a safe state cannot be reached quickly and without further operation, e. g., an aircraft or an HAD vehicle. Such safety-critical systems are called *fail-operational* systems [Kop11]. After a system failure is detected, a fail-operational system has to continue to provide a possible degraded service until a safe state is reached. For example, after a system failure the HAD vehicle has to operate until the driver has taken control or the car is stopped in a safe position. In this thesis, we focus on fail-operational systems.

Fault prevention is part of general engineering and development methods and aims to reduce the number of faults introduced in the produced system [ALRL04]. In order to create a fail-operational system, fault prevention mechanisms are not sufficient. Instead, a *fault tolerance* approach with active redundancy to mask component failures is required [Kop11]. Fault tolerance is the ability of a system to continue providing the correct service in presence of faults [Joh84]. Note that self-repair, self-healing, and resilience are used as synonyms for fault tolerance [ALRL04].

The most important and most common fault-tolerance strategy is *triple modular redundancy (TMR)* [Kop11, Dub13]. Under TMR the results of three redundant *fault containment units (FCUs)* are compared by a *voter*, as shown in Figure 1.2 on page 5. An FCU is a system component that fails independently, i. e., error propagation to other components

is prevented. The voter accepts and outputs the final result only if at least two FCUs deliver the same result [Par94]. The reliability of the voter has a significant impact on the system's reliability, since it is in series with the redundant FCUs and thus a *single point of failure (SPOF)* [Dub13]. Although a failure of an FCU is masked by the voter, it reduces or eliminates any further *fault-masking* capability [Kop11]. Therefore, a maintenance mechanism that repairs the failed FCU when notified by the voter is beneficial for the system's fault-tolerance.

A *fail-silent* FCU either produces a correct result or no result at all [Kop11]. If the FCUs exhibit fail-silent behavior, a *dual modular redundancy (DMR)* setup is sufficient to construct a fault-tolerant system.

N modular redundancy (NMR) is the generalization of the TMR strategy for N redundant FCU. The voter compares all FCUs results and generates an output only if $M \leq N$ FCUs delivered the same result. The NMR fault-tolerance mechanism has the following prerequisites [Kop11]:

- Deterministic behavior of the FCUs. Without a deterministic behavior of the FCUs, two fault-free FCUs that started in the same state might at some point provide different results for the same input value.
- Independently failing FCUs. If a fault causes more than one FCU to fail, it can be shown that the correlated failure has a tremendous impact on the overall reliability.
- Communication infrastructure with overload prevention. Without further assumptions about the fault model, a failed FCU could overload the communication infrastructure and delay the results of the working FCUs.
- Fault-tolerant global time base. Without a global time base, the FCUs have to reach consensus over the time, i. e., they have to synchronize. Independent of the fault model, this requires to tolerate *Byzantine* or *malicious* faults, e. g., a failed FCU that sends a correct result or timestamp to one part of the remaining FCUs, and an incorrect result or timestamp to the other part.

Note that *security* measures partly overlap with dependability measures and include availability, integrity, and confidentiality [ALRL04]. Integrity is defined as the absence of improper system alterations and confidentiality denotes the absence of unauthorized disclosure of information [ALRL04]. Security is an important topic, especially considering the increasing connectivity trend in embedded systems. However, security is orthogonal to our work and out of scope of this thesis.

2.2 Many-Core Processors

Higher integration rates and shrinking feature sizes allowed hardware manufactures to increase the amount of processing elements or cores per chip. Such many-core processors can provide higher computational power compared to single-core processors at the same operating frequency. In this section, we introduce the specific hardware properties of many-core processors. After a general distinction between multi-core and many-core

processors, we present the components of a single core. Multiple cores are connected by a NoC, whose structure we detail afterwards. Finally, we present our fault hypothesis for the many-core processor hardware.

2.2.1 General Many-Core Design

Today's multi-core processors or Multi-Processor Systems-on-Chip (MPSoCs) comprise up to 18 cores, such as Intel's Xeon E7-8890 v3 processor [Int15]. Such general-purpose multi-core processors contain homogeneous complex high-performance cores and provide a shared address space between these cores. In order to maintain a cache-coherent memory system, the last-level cache (LLC) is typically shared by all cores, while the highest level (L1) cache is core-private [Vaj11]. A cache system is said to be coherent if and only if all cores, at any point in time, have a consistent view of what is the last globally written value to each location [Vaj11]. The cache-coherence protocol that ensures this property is implemented on top of an interconnect fabric such as a crossbar or ring bus that connects the cores and higher-level caches, respectively, with the LLC [Vaj11].

The scalability of these multi-core processors is limited by the memory bandwidth and throughput of the interconnect fabric as well as the power consumption [Vaj11]. Therefore, researchers in academia and industry are working on tiled architectures and many-core processors, which are "loosely defined as chips with several tens, but more likely hundreds, or even thousands of processor cores" [Vaj11, p. 13].

In contrast to multi-core processors, many-core processors abandon a cache-coherence mechanism due to its communication overhead [MVdWF08, Rut14]. Instead, they provide core-local memories that can be used as scratchpad memories and include a scalable communication fabric such as an on-chip network, which allows direct access to the core-local memories [Vaj11]. Many-core processors eventually contain heterogeneous specialized cores with different performance characteristics and instruction set architectures (ISAs) [Rut14]. Additionally, while general-purpose multi-core processors often include a speculative dynamic instruction scheduling, the trend in many-core processors is to revert to simple single-issue in-order execution [Vaj11, Rut14].

In contrast to graphics processing units (GPUs) such as Nvidea's Tesla series [NVI14] or AMD's FirePro series [Swi14], which belong to the class of single instructions, multiple data (SIMD) of parallel computers, many-core processors belong to the multiple instructions, multiple data (MIMD) class. Similar to a distributed system, each core of a many-core processor can execute a different instruction.

Typical examples of COTS many-core processors are

- Adapteva's Epiphany [Ada13]
- Kalray's Massively Parallel Processor Array (MPPA) [dDAB+13, dDAPL14]
- Mellanox's TILE-GX processors [Mel15]
- PEZY Computing's PEZY SC [PEZ12]

- Intel's TeraFLOPS processor [VHR+08], Single-chip Cloud Computer (SCC) [Int12, Bar10a, MVdWF08], and Many Integrated Core (MIC) architecture with the Xeon Phi coprocessor product family

- ST(microelectronics) Heterogeneous lOwpoweR Many-core (STHORM) academic processor [MBF+12, BFFM12]

To improve the power efficiency and scalability of many-core processors, researchers have introduced the globally asynchronous, locally synchronous (GALS) approach. In a GALS architecture, each core is managed by its own clock and represents a synchronous block of logic in an independent timing domain [TGL07]. The clocks of separate cores potentially run at different frequency and phases. The communication fabric is either asynchronous (clockless), mesochronous (hierarchical clock tree with same frequency but different phases), or synchronous but driven by a separate clock. The core interfaces synchronize both clock domains, e. g., via dual-clock first in, first out (FIFO) buffers. The GALS approach is implemented e. g., in the STHORM, which contains an asynchronous NoC [MBF+12, BFFM12].

The non-deterministic hardware timing properties of GALS many-core processors render their usage for embedded applications with real-time requirements difficult. For this reason, in this thesis we assume that all cores and the NoC operate at the same frequency and that the hardware timers of different cores do not drift, i. e., exhibit a bounded clock skew. This assumptions follows the synchronous processor design with a global clock implemented in Adapteva's Epiphany, Mellanox's TILE-GX, Kalray's MPPA, Intel's SCC with dynamic voltage and frequency scaling (DVFS) disabled.

2.2.2 Components of Each Core

In this thesis, we consider many-core processors that consist of $y \cdot x$ cores $C_{j,i} \in \mathscr{C}$ with $j = 1,2,\ldots,y$ and $i = 1,2,\ldots,x$. The number of cores in a many-core processor is denoted as $|\mathscr{C}| = yx$. All cores are *homogeneous*, i. e., all cores are of a common type, support the same instruction set architecture (ISA), and operate at the same frequency.

Each core consists of a CPU, a timer, an inter-core reset (ICR) mechanism, a scratch-pad random access memory (RAM) protected by an MPU, and a network interface (NI), as shown in Figure 2.3 on the following page. Following the typical System-on-Chip (SoC) communication standards, the communication between these components via the core-local crossbar is specified by a protocol such as the Virtual Component Interface (VCI) [On-01], the Open Core Protocol (OCP) [Acc13], or the Advanced eXtensible Interface (AXI) [ARM04]. In this thesis, we assume that the protocol of the core-local crossbar specifies transactions based on requests and responses.

The CPUs in a many-core processor typically follow the Reduced Instruction Set Computing (RISC) scheme and have a lower complexity than the high-performance cores of general-purpose multi-core processors. For example, Adapteva's Epiphany contains 32 bit RISC CPUs with a flexible pipeline length between five and eight stages [Ada13]. STHORM contains STxP70-V4 CPUs that have a 32 bit RISC architecture with a seven-staged pipeline [BFFM12]. Note that simple CPU architectures without out-of-order

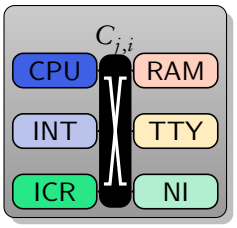


Figure 2.3: Architectural model of a core of a many-core processor containing a CPU, a timer (INT), an ICR mechanism, an uncached scratch-pad RAM protected by an MPU, and an NI connected by a local crossbar (depicted as X).

execution, branch prediction, or caches are advantageous to determine a tightly bound WCET estimation [WEE+08].

The core-local timer can be configured in software to provide a periodic interrupt after a specific amount of cycles. This interrupt is required to drive the real-time scheduler of the OS.

Each core is able to trigger the reset of another core by the ICR mechanism. The reset interrupt uses separate signal lines independent of the NoC. A core-local reset mechanism is implemented e. g., in Adapteva's Epiphany and in Intel's SCC.

The core-local scratch-pad RAM is integrated on the same die and operates with the same performance as a cache. It provides at least two banks that can be accessed in parallel. One bank is reserved for the `.mailbox` section, a specific memory area used for communication between cores. The remaining banks are used for the remaining sections containing instructions and data. This assumption follows the design of Adapteva's Epiphany, which provides four banks that can be accessed in parallel each cycle.

The CPU and the NI potentially access the `.mailbox` section of the RAM in parallel. Without a dual-port RAM, which is uncommon in COTS hardware due to its higher costs, these parallel accesses result in conflicts that have to be handled by the core-local crossbar. In this thesis, we assume that the core-local crossbar implements a round-robin arbitration policy to solve such conflict. Thus, a read or write request of the CPU or the NI is blocked in the worst-case. Due to the round-robin arbitration policy, this blocking delay is bounded and can be included in the WCET. Since the CPU is only delayed when accessing the RAM bank with the `.mailbox` section, the increase in the WCET is within reasonable bounds.

To prevent a propagating failure when one failed core corrupts the instructions or data of another core and ultimately cause a failure of the other core, we assume a memory protection unit (MPU) is available in each core. The software configures the MPU to prevent all write accesses to a specific address space from all but the local core. Thus, the MPU prohibits all requests of the NI to memory sections other than the `.mailbox` section and respective RAM bank. This assumption follows the design of Adapteva's Epiphany, which provides an MPU for eight predefined sections, Kalray's MPPA, which provides a full-fledged memory management unit (MMU), and Mellanox's TILE-GX, which provides a configurable "hardwall" in the NoC.

The NI connects a core with the NoC of the many-core processor. It consists of two parts, the initiator NI and the target NI. The initiator NI translates a CPU's read or

write requests to non-local memories into request packets and injects these request packets into the NoC. It also receives response packets from the NoC and translates them into responses for the requesting local CPU. These responses contain the data that was read or information about the success of a write operation. The target NI receives request packets from the NoC and translates them into requests for the local RAM. It also receives the RAM response, converts it into a response packet, and sends it back to the requesting CPU via the NoC.

The CPU either stalls until a response packet is received, or it continues execution and deals with the response packet when it is received. The former case is known as *synchronous* or blocking communication. The latter case is known as *asynchronous* or non-blocking communication. In this thesis, we consider both cases.

2.2.3 Network-on-Chip

In this section, we present the architecture of the NoC including the routing mechanism for the packets.

A packet-switching NoC is a scalable on-chip communication infrastructure, which is the result of adapting concepts such as routers and packets from computer networks into chip design [DT03, DMB06, BM06, MHL+15]. Due to their limited scalability, point-to-point connections, crossbars, and shared (ring) buses are replaced by NoCs in many-core processors [BM06, ZKCS02, MHL+15]. Hence, a NoC as communication fabric is an important differentiator between the multi-core and many-core processors [Nik15]. Since all cores are connected by the NoC, the many-core processor provides a shared memory abstraction with non-uniform memory access (NUMA) latencies. Following Adapteva's Epiphany and Intel's SCC, we assume a global address space where the highest bits of the memory address encode the core ID.

In this thesis, we consider a many-core processor that contains two NoCs, as shown in Figure 2.4 on the next page. Two separate NoCs are required to break the request-response dependency [HGR07]. If both request and response packets were sent in the same NoC, the request packet of one message could hold resources required by the response packet of another message and vice versa. Thus, deadlocks could still arise due to dependencies at message level [HGR07]. For this reason, COTS many-core processors typically contain multiple NoCs, e. g., Adapteva's Epiphany, Kalray's MPPA, and Mellanox's TILE-GX.

The request and response NoC have the same architecture and are identical except for the width of the links. For this reason, unless explicitly stated, we describe one NoC that represents the request or the response NoC in the following.

NoC Architecture

The NoC contains $y \cdot x$ routers $r_{j,i} \in \mathscr{R}$ with $j = 1, 2, \ldots, y$ and $i = 1, 2, \ldots, x$. The routers are connected in a 2D mesh topology, as depicted in Figure 2.4 on the following page. This topology is very common in many-core processor NoCs [BM06, MHL+15], e. g., in Adapteva's Epiphany, Mellanox's TILE-GX, and Intel's SCC.

In the 2D mesh topology, two adjacent routers are connected by a point-to-point connection. Hence, when the NI of core $C_{j,i}$ injects a packet $p_i \in \mathscr{P}$ in the NoC, the

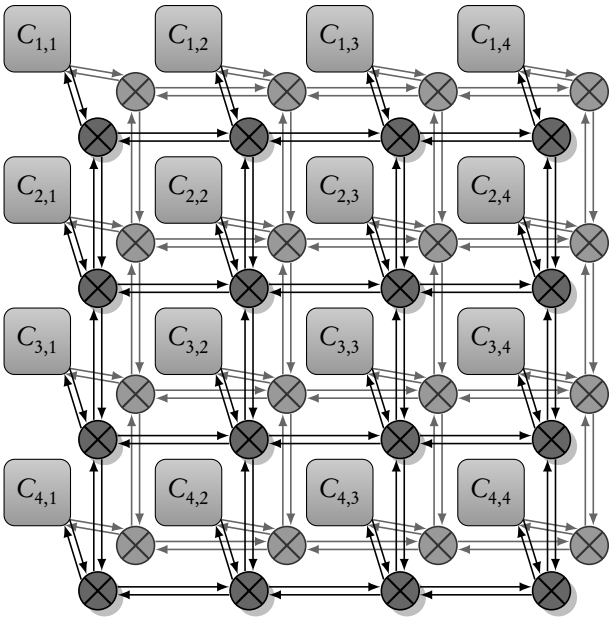


Figure 2.4: Architectural model of the request (dark) and response (light) 2D mesh NoCs with links and routers in a $4 \cdot 4$ many-core processor.

packet is forwarded to the router $r_{j,i}$ via the point-to-point connection. If the packet's destination is core $C_{j+1,i+2}$ and router $r_{j+1,i+2}$, respectively, the packet p_i traverses 4 routers, as shown in Figure 2.6 on page 27 for the red route (dashed line). The function $h : \mathscr{P} \to \mathbb{N}^+$ denotes the number of routers on the route of a packet from its source to its destination, i. e., the number of hops. The functions $\sigma : \mathscr{P} \to \mathscr{C}$ and $\delta : \mathscr{P} \to \mathscr{C}$ denote the corresponding source and destination core of each packet. In this thesis, we consider only *unicast* packets with exactly one source and one destination.

The connection between two adjacent routers as well as the connection between a router and an NI is implemented by a link $l \in \mathscr{L}$. A link is a bidirectional (full-duplex) connection, so two neighboring routers can exchange packets in opposite directions without a conflict. It is physically implemented by multiple wires for data transfers and the router-level flow control mechanism.

A flow control unit (flit) is an atomic entity that is transmitted over any link $l \in \mathscr{L}$ in one clock cycle. Thus, the size of a flit (in bits) equals the number of data wires in a link. The only difference between the request and the response NoC is that the flit size in the request network is larger than in the response network, which results from the SoC communication protocol. In both networks, all links have the same bandwidth $b = 1 \frac{\text{flit}}{\text{cycle}}$.

The size of a packet $p_i \in \mathscr{P}$ is denoted as $|p_i|$ and specifies the natural number of flits that belong to the packet p_i. When a read or write request is injected into the request network, the NI translates it into a request packet and three flits, respectively. Similarly, when a response is injected into the response network, the NI translates it into a response packet and three flits, too. Thus, all packets have a common fixed size of three flits, so $|p_i| = 3$ flits. The first (header) flit contains the destination address. The second flit decodes the request type, the source address, and other configuration information. The third (tail)

flit contains the actual data to be written. The flits are injected into the network in an ordered manner.

Router

A schematic overview of a router $r_{j,i}$ is presented in Figure 2.5 on the following page. Following the 2D mesh topology, each router has five input and five output ports. Four ports are connected to the northern, eastern, southern, and western router or left unconnected if the router is placed a the edge of the chip. The fifth port of router $r_{j,i}$ is connected to the NI of the core $C_{j,i}$. Note that the NI of each core is connected to both NoCs, i. e., to one router of the request NoC and one router of the response NoC.

Within a router, each input and output port is connected to a link controller (LC). The input LC stores the received flits in the input FIFO buffer and notifies the upstream router via the router-level flow control wire in case the input FIFO buffer is full. The output LC forwards the flits in the output FIFO buffer unless the flow control signals no capacity in the downstream router. To forward the flits from the input buffer to the output buffer, each router contains an internal crossbar. The crossbar allows to forward flits from different input buffers to mutually different output buffers in parallel. In this thesis, L_r denotes the fixed amount of time it takes for one flit to be forwarded from the input to the output port of a router without a conflict. The internal crossbar is controlled by the routing and arbitration logic.

Routing Policy

The routing logic implements the dimension-order (XY) routing policy. Under this policy, a packet is first transmitted along the horizontal axis. When the column of the destination core is reached, the packet is transmitted along the vertical axis until the destination core is reached. An example of two routes following this routing policy is shown in Figure 2.6 on page 27. The dimension-order (XY) routing policy is known to be minimal, deadlock-free, and deterministic [DMB06]. A packet always traverses the network with a minimal number of hops under a minimal routing policy [DMB06]. Under a deterministic routing policy, the same path between any source and destination pair is always chosen [DT03]. It is a subset of oblivious routing, where the route is agnostic of the network's present state [DT03]. Hence, deterministic routing policies ensure the in-order delivery of packets [DMB06]. Since the dimension-order (XY) routing policy is simple to implement [DT03], it is commonly used in COTS many-core processors, e. g., in Adapteva's Epiphany, Mellanox's TILE-GX, and Intel's SCC.

The arbitration logic resolves collisions that occur if multiple packets need to be forwarded to the same output port at the same time. Early access means that a packet is forwarded immediately if there is no collision. To resolve a collision, the arbitration logic applies a round-robin early access scheme to select one packet that is transmitted. All remaining packets are stalled and their flits are stored in the input buffers. When the tail flit of the selected packet is transmitted, the conflict between the remaining packets is resolved in the same manner. In this thesis, $\hat{L_C}$ denotes the upper bounded amount of time in which a packet is stalled by a collision with another packet in one router, i. e., the

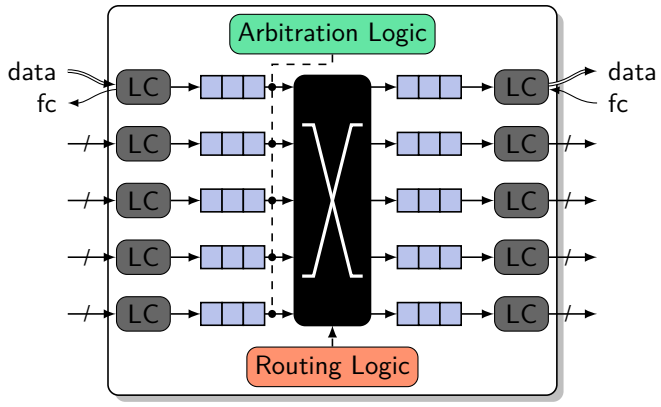


Figure 2.5: Architectural model of a router with five input and output link controllers (LCs) connecting the data and flow control (fc) signals to the FIFO input and output buffers, and an internal crossbar controlled by the routing and arbitration logic.

worst-case time it takes a router to arbitrate a collision and process all flits of the selected packet.

Consider one packet $p_i \in \mathscr{P}$ in an empty NoC, i. e., no other packets can collide with packet p_i. The routing logic determines the requested output port from the destination address stored in the header flit of packet p_i. If there is no collision, the router reserves the connection from input to output port until the tail flit of the packet p_i is forwarded. If there is enough space in the bounded FIFO input buffer of the next router, each following flit of packet p_i is forwarded immediately in a pipeline manner. This way of forwarding packets is known as *wormhole switching* [DMB06]. Wormhole switching is the prevailing design choice in NoCs due to its reduced buffer requirements [BM06], and is implemented e. g., in Kalray's MPPA, Mellanox's TILE-GX, and STHORM.

In contrast to store-and-forward switching that is commonly used in computer networks and stores packets completely in one router, a packet can span multiple routers in a wormhole switching network, as shown in Figure 2.6 on the facing page for the three flits of the red packet [BM06]. This property leads to additional collisions, e. g., if a router reserved an output port for the remaining flits of packet p_i and the header flit of newly arriving packet p_j contains an address that requires the same output port. Since the output port is reserved for packet p_i already, the newly arriving packet p_j is stalled until the tail flit of the packet p_i has passed and no arbitration is needed.

The bounded input FIFO buffer might not be large enough to store all flits of a stalled packet and newly arriving packets. To prevent a buffer overflow, the LC employs a router-level flow control mechanism to notify the upstream router that it does not accept more flits. Since the buffers of the upstream routers will eventually be full as well, the router-level flow control mechanism propagates the blockage through the NoC. This mechanism is known as back-pressure [DMB06]. If the back-pressure reaches a core and its NI, respectively, no further packets can be injected into the NoC.

An example of the described behavior is given in Figure 2.6 on the next page. The route of packet p_i is shown as red dotted line, its flits are represented as filled red input buffers.

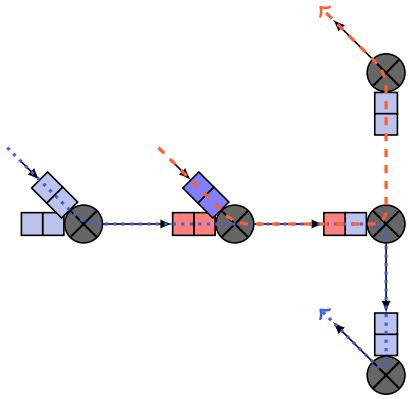


Figure 2.6: A schematic representation of a NoC with only a selection of routers, links, and FIFO buffers with a capacity for two flits are shown. The routes of the two packets, red and blue, are represented by dotted lines and follow the XY routing policy. The header flits of the red and blue packet collided in the center router. The round robin arbitration selected the red packet. The blue packet is stalled and currently causes back-pressure.

The route and flit of packet p_j are painted in blue color. Since the red packet p_i either was selected by the arbitration logic in the previous clock cycle or arrived one clock cycle before the blue packet p_j, its header flit was already forwarded to the downstream router and the output port is reserved for the red packet p_i. The blue packet p_j is stalled and causes back-pressure in the NI, since the last flit cannot be injected into the NoC.

Guaranteed Service Communication

In contrast to point-to-point or crossbar communication fabrics, the communication latency in a NoC is affected by collisions and the resulting back-pressure. Hence, when a CPU accesses a memory via the NoC, the latency of the access depends on the packets currently present in the NoC and ultimately on the instructions executed by the other cores in the processor. However, *GS* communication is required in real-time systems in order to ensure the WCCTs and verify the application's real-time constraints, respectively. GS communication describes a Quality of Service (QoS) class that includes a guaranteed performance of the traffic, i. e., a bounded latency and a guaranteed bandwidth [BM06].

Without the a priori knowledge of the executed instructions on all cores, the state of the round-robin arbitration logic at any point in time, and other low-level details, a logical separation of the communication on the NoC is required to derive the latency of the communication. The logical separation of communication can be achieved by reserving dedicated channels in a NoC and implementing virtual circuit switching [BM06]. In contrast to a packet-switched network, a circuit switched network first establishes a connection between source and destination that is used exclusively afterwards, as known from the landline telephone network.

Virtual circuits use logically independent resources in order to avoid packet collisions [BM06]. Resource independence is either achieved by applying a time-division

multiplexing (TDM) scheme for the complete NoC, as e. g., in the Æthereal NoC [GDR05, GH10], or by introducing *virtual channels*. Virtual channels share a physical link but use independent and individual input buffers per virtual channel in the router ports. If the virtual channels are arbitrated by priority levels and a preemption mechanism, concepts from scheduling theory can be adapted to determine an upper bound of the communication latency [BM06].

However, NoCs with GS communication are typically not available in COTS many-core processors [DNNP14, AJEF15]. Except for Kalray's MPPA, all mentioned many-core processors provide merely best-effort (BE) communication. Kalray's MPPA includes a hardware-implemented traffic shaper that allows to enforce GS communication for a statically known communication by applying network calculus (see Section 3.1.2 on page 42). Note that the NoC in Intel's SCC provides eight virtual channels, but it does not support reserving virtual channels for GS communication [Bar10a]. Therefore, in this thesis we assume that the NoC does *not* provide any hardware support for GS communication.

2.2.4 External Reliable Memory

Focusing on consumer-grade many-core processors, we do not assume any fault-tolerance hardware features such as ECCs protecting the memories [Dub13], fingerprinting the instruction execution history [SGK+04], or lock-stepped cores [Sor09].

However, we assume a reliable off-chip memory to be available and connected to the many-core processor. This off-chip memory is necessary to load the instructions and data into the many-core processor's RAMs after the power is switched on. All mentioned COTS many-core processors provide interfaces to connect external memories to the processor's NoC. An example of a reliable off-chip memory is a flash memory with ECC protection. Note that ECC is a typically implemented in NAND flash memories to compensate wear-out effects [CYM+12, YLJY15].

We further assume that the CPU of each core fetches the first instructions from this external reliable memory.

2.2.5 Fault Hypothesis

In this section, we introduce our fault hypothesis, i. e., the nature and origin of faults. In this thesis, the system boundary is the many-core processor. Therefore, we do not consider faults outside the processor, e. g., a short circuit due to water on the printed circuit board (PCB) or a power surge in the ECU due to a lightning strike.

Fault Rate

The notion of faults, errors, and failures introduced in Section 2.1.2 on page 15 depends on the level of detail and scope in which they are used. For example, at the physical level of an SRAM cell, a fault denotes the impact of a high-energy particle, an error occurs if the impact was large enough to flip the stored value, and a hardware failure arises if the cell is read, since the SRAM component fails to provide its correct service. At software level, a fault denotes a bit flip in a memory cell, an error occurs if the cell is read, and

a software failure arises if the wrong value is not masked and results in a wrong output. Finally, at system level, a fault denotes reading a flipped memory cell, an error occurs if the software executed on one core produces a wrong output, and a system failure arises only if no other means are taken and the wrong output is not checked, e. g., by comparing it with the results produced by other cores, but forwarded immediately.

In order to specify our fault hypothesis, we first assume a physical level of detail. Here, a failure is the hardware failure of an electronic component of the system. For these hardware components, the failure rate over time is commonly assumed to follow the well-known *bathtub curve*, plotted in Figure 2.7 [KK10, Bir14].

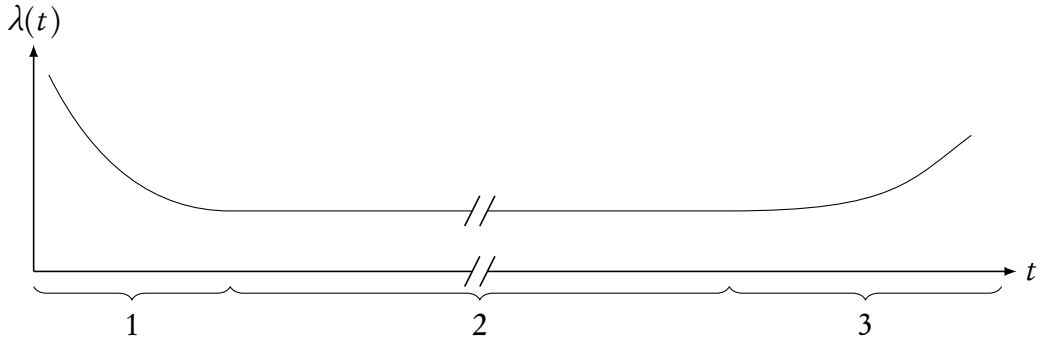


Figure 2.7: Schematic plot of a bathtub curve, representing the hardware failure rate over time in three phases.

The bathtub curve can be divided into three phases [Bir14]:

1. Early hardware failures that typically result from production defects. They are eliminated by stress and burn-in tests.

2. A constant hardware failure rate during operational lifetime.

3. Wearout failures caused by aging effects. For many electronic devices, these failures occur after ten years.

In this thesis, we only consider the useful operation phase. It is commonly assumed that in the useful operation phase, the *hardware failure rate* λ is constant [KK10, Nic11, Bir14].

Fault Types

Considering the system level of detail, the hardware failure of an electronic component is the result of a fault at the physical level. As mentioned in Section 2.1.2 on page 15, these physical faults are either of permanent, transient, or intermittent nature.

Permanent faults are persistent in time. Typical examples of permanent faults are disconnected wires or broken transistors that are the results of electromigration and time-dependent dielectric breakdown caused by stress migration and thermal cycling [Sor09]. These are typical examples of aging and wearout effects. During the useful operation phase of most systems, the permanent fault rate is three orders of magnitude lower the fault rate

of transient faults [Bau05, Nic11]. Therefore, we do not consider permanent faults in this thesis.

An intermittent fault oscillates between an active and a dormant state, so it occurs occasionally in a bursty manner. They are typically caused by wearout effects that have not yet resulted in a permanent fault [Kop11]. Hence, an increasing intermittent fault rate is an indication for the beginning of the wear-out phase [Kop11]. Since we focus only on the useful operation phase of a system, we do not consider intermittent faults in this thesis.

Transient faults are only present for a bounded time interval. Due to the physical nature of these transient faults, we assume that they are stochastically independent. In this thesis, we consider transient faults.

A common cause of transient faults is electromagnetic interference, either from outside the device or from long parallel signal wires [Sor09]. The latter is also known as a cross-talk [Sor09]. Furthermore, transient faults originate from cosmic radiation and radioactive impurities in the chip and its packaging [Bau05]. The impact of cosmic rays with the atmosphere causes high-energy ionizing particles, e. g., neutrons, whereas radioactive impurities produce alpha particles [Bau05]. If a high-energy particle or an alpha particle strikes the chip, it can dislodge a significant amount of charge within the semiconductor material, which is referred to as single event effect (SEE) [Nic11]. If this charge exceeds the critical charge of the semiconductor, it can change the state of a logic element such as an SRAM cell and become an error, which is known as single event upset (SEU) [Bau05, Sor09, Bir14]. If a particle strike affects combinational logic, it is called a single event transient (SET) and becomes an error only if latched in a logic element [ME02].

Additionally, high-energy ionizing particle strikes can turn on the parasitic bipolar transistors of a complementary metal-oxide-semiconductor (CMOS) transistor, which is then latched-up [DSSH03]. This fault is called single event latch-up (SEL) and exhibits no transient behavior [Bau05]. Instead, the recovery from an SEL requires a full power cycle (pseudo-permanent behavior) or is completely impossible (permanent behavior) [DSSH03, IO11]. Dodd et al. [DSSH03] showed that in contrast to SEUs and SETs, whose rate increases with a lower device voltage, the probability of SELs reduces with a lower voltage. As modern processors operate at decreasingly low voltages [Nic11], we do not consider SELs in this thesis.

Fault Effects

Transient faults can cause soft errors, e. g., a logical circuit malfunction or a single or multiple bit flips in memory cells, registers, or NoC router buffers. In contrast to permanent faults, whose effects are often modeled as so-called stuck-at errors, the damage of soft errors is not permanent [Nic11]. Thus, a soft error can be repaired by software by re-executing the instruction, rewriting the memory cell, or resending the packet.

Since a soft error at the system level of detail is equivalent to a failure at the level of detail of a hardware component, the hardware failure rate λ is also known as soft error rate (SER). The SER is typically given in *failure in time (FIT)*, where one FIT is one failure in 10^9 device hours. A device hour is an hour of the device operating. In 2005, Baumann stated that for advanced computer chips, the SER λ can exceed $50,000$ FIT [Bau05].

A soft error can propagate through the system and cause other errors such as a software task exhibiting incorrect behaviors, either by producing erroneous results or no result at all (omission fault). In case fault-tolerance mechanisms are not protecting the system and the soft error is not detected and corrected or masked, it results in a potentially hazardous system failure.

Vulnerability Factor

A fault can be masked and not become an error even if no fault-tolerance mechanism is present, e. g., in case a faulty memory cell is not used at all. Such faults that do not cause a failure are called *benign*. Similarly, errors can be masked and not become failures, e. g., if an ECC detects and masks a faulty memory cell. The ratio of faults that become failures is commonly known as *vulnerability factor*.

Vulnerability factors were defined for microprocessor components [MWE+03], for ISAs [RSKH11], and for the complete system stack including hardware and software [Sri10]. Simulations showed that the vulnerability factors highly depend on the specific hardware and software implementation [Sri10].

In this thesis, we define the vulnerability factor v as the fraction of faults that cause a failure of a software component executed on one core. For example, a task T with a vulnerability factor v_T executed on a core with an SER λ is expected to fail with the *task failure rate* $\Lambda_T = v_T \lambda$.

Common Mode Faults

Common mode faults denote faults that occur simultaneously in different components of a system [Dub13]. These faults often originate from the same root cause, therefore they also known as *common cause faults* [TS09].

The first reason for common mode faults are systematic faults such as hardware design flaws and software bugs. This source of common mode faults can be eliminated using diverse hardware designs and N-version programming, respectively [Dub13]. However, we consider homogeneous many-core processors with a common hardware design for all cores. Therefore, we rely on test and verification methods that show or prove the correctness of the hardware design or software behavior, and do not consider common mode faults caused by systematic faults in this thesis.

The second reason for common mode faults is the shared environmental condition, e. g., the temperature, pressure, or electromagnetic interference that affects the complete chip [Dub13]. In this thesis, we assume that the processor is operated within its specified conditions. For COTS many-core processors in the automotive domain, this typically means they are installed within the passenger cabin. Therefore, we do not consider common mode faults caused by environmental conditions.

The third reason for common mode faults are shared resources such as the power supply as well as the clock and reset tree in the many-core processor [Dub13]. The chip can be designed to protect against such faults by including redundant power supplies and voltage monitoring, core-local clock signals according to the GALS approach, and a specific pattern as a reset signal. However, except for the academic STHORM processor, such features

are typically not implemented in COTS many-core processors. Thus, common mode faults caused by shared resources need to be considered. Due to their nature of affecting the complete processor, they cannot be protected against by software running on the same processor, which is the proposed solution to protect against transient faults in this thesis. Hence, we do not specifically consider common mode faults caused by shared hardware resource in the following. Instead, we propose to compensate for these common mode faults in shared resources by increasing the vulnerability factor v of each component respectively.

Byzantine Faults

The term Byzantine fault originates from the Byzantine Generals Problem described by Lamport et al. [LSP82]. The authors investigate the problem of a distributed system reaching consensus when component failures are observed differently by the remaining components of the system. These component failures, which result in an inconsistence within the system, are known as Byzantine faults [ALRL04].

A common way to interpret Byzantine faults is to consider them as malicious, i. e., a component sends different messages to the other components on purpose. As mentioned in Section 2.1.2 on page 18, we do not focus on security issues in this thesis. Therefore, we assume that task replicas do not send different results to both voters on purpose.

Byzantine faults can be the effects of soft errors as well. However, our fault-tolerance mechanism does not resemble a distributed system and the task replicas do not need to reach consensus. Instead our mechanism includes two fail-silent voters that compare the results of all replicas. Instead of negotiating the final result, the inactive voter merely serves as backup of the active voter. Therefore, no Byzantine faults can occur between both voters.

Consider the case where a task replica sends a correct result to one voter and an incorrect result to the other voter. If the inactive voter receives the incorrect result, the correct result will be forwarded by the active voter. If the active voter gets the incorrect result, it will detect the incorrect result. in this case, Even if the active voter is unable to determine the majority of results, the inactive voter will output the final result correctly. Hence, the voters are able to tolerate Byzantine faults of the task replicas.

The voters compare the results available at given instants of time. As mentioned at the beginning of this section, all cores of the many-core processor operate with a common frequency. Hence, the processor provides a global time base for all task replicas and voters and no consensus over time needs to be found.

2.3 Embedded Many-Core Operating System

The general job of an OS is "to control all the computer's resources and provide a base upon which the application programs can be written" [TW06, p. 1]. In the scope of this thesis, the OS has to provide the means to execute embedded real-time safety-critical applications on a many-core processor. Thus, the embedded many-core OS has to adapt concepts from the world of well established, commercially available real-time operating systems (RTOSs) as well as concepts from the world of academic, scalable many-core OSs. In this section, we first present important concepts of RTOSs. Then, we introduce the main architecture of many-core OSs, before we propose how both worlds can be combined. Finally, we detail the OS message passing mechanism and present means to ensure fault isolation between cores.

2.3.1 Real-Time OSs

An RTOS must provide all essential general-purpose OS capabilities. This includes the management of resources such as the CPU, memory, interrupts, and system timer as well as providing a hardware abstraction layer to the application and the management of processes including the scheduling and inter-process communication (IPC) [TW06]. These mandatory OS functionalities are implemented by the *kernel* [Mar11].

If the kernel provides only the essential functionalities and all other OS services are implemented as processes in user space, the OS architecture is called *microkernel* [RDJ+09]. Examples of microkernel-based OSs is SYSGO's PikeOS, whose design is based on the L4 Microkernel [KW08].

In contrast, if all OS services are implemented within the kernel, the OS architecture is called *monolithic kernel* [RDJ+09]. Examples of a monolithic kernel architecture are Lynx Technologies' Lynx OS [Ble11] and Linux, which was adapted for real-time systems by several projects, amongst others the Preempt RT patch [RH07], Litmus^{RT} [BCA08], Xenomai [OB12], and Real-Time Application Interface (RTAI) [MBD+00].

In addition to common OS features, an RTOS must provide a notion of physical time, have a predictable timing behavior, i. e., the execution times of services must be upper bounded, and include a real-time scheduling algorithm, which guarantees that all tasks meet their deadlines [Mar11].

The field of application of RTOSs starts at deeply embedded systems with a small microcontroller, a few kilobytes of memory, and a few megahertz of clock frequency and ends at MPSoCs with gigabytes of attached memory and a clock frequency in the range of a gigahertz. Especially at the low-performance end, a low context switching latency, a small memory footprint, and low overheads in general are important features of an RTOS [BM05]. In such simple systems without virtual memory and an MMU, the RTOS design is often "kernel-less" and the OS is linked together with the application [RDJ+09]. Thus, the software is contained in a single executable which is executed in a single address space and no program loader is required [RDJ+09]. Additionally, OS service calls can be implemented as function calls and do not require a context switch [RDJ+09]. The OS can still protect itself from unwanted modifications if an MPU is available in hardware. Examples of RTOSs that are linked together with the application are AUTOSAR

basis software [AUT15], Express Logic's ThreadX [Exp06], and Mentor Graphics' Nucleus [Men15].

Additionally, an RTOS often has to be certified when used in a safety-critical context [Kop11]. However, certification is difficult if it contains dynamic control structures and thus behaves non-deterministically [Kop11]. Examples of RTOSs that are certified to the IEC 61508 standard [Int10] are Green Hills's Integrity [Gre13] and Wittenstein's SafeRTOS [Bar10b]. A sister RTOS of the latter, namely FreeRTOS, is also available as open-source software [Bar10b].

Due to their historical dominance in embedded systems, RTOSs are typically designed for single-core processors. However, some RTOSs also support *symmetric multi-processor (SMP)* systems. An SMP system is a homogeneous multi-core processor or a computer with multiple homogeneous processors [Vaj11]. The cores or processors access a shared main memory with a global address space [Vaj11]. If the SMP system has a uniform memory access (UMA) architecture, all tasks and processes can be executed by any core or processor without additional overhead. In an SMP system, the RTOS kernel typically resides on one core or processor and controls all cores or processors [Vaj11]. An example of an RTOS that supports SMP systems is Embedded Configurable Operating System (eCos) [Mas03].

Some RTOSs also support *asymmetric multi-processor (AMP)* systems, which have separate address spaces per core or processor [Vaj11]. The cores or processors potentially have heterogeneous architectures [Vaj11]. In such systems, each core or processor is managed by its own kernel and the RTOSs provides a message passing service for IPC [Vaj11]. An example of a RTOS that supports AMP systems is WindRiver's VxWorks [Win16].

The latter setup is closely related to the typical architecture of scalable many-core OSs, which is presented in the following. Note that there also exist hybrid SMP/AMP setups, also known as bound multi-processors (BMPs) systems [Vaj11]. BMPs systems are supported by some RTOSs, namely Enea's Operating System Embedded (OSE) [Str09], eSOL's eT-Kernel [Gon07], and QNX Neutrino [QNX12].

2.3.2 Many-Core OSs

Many-core processors contain hundreds of cores, abandoned cache-coherence, and represent a NUMA architecture. These processors propel the research and exploration of novel and scalable OS designs. Hence, apart from the hardware architecture, the OS design provides a clear differentiation between multi-core and many-core processor [Rut14]. As a result, several academic projects have proposed concepts for an OS for many-core processors, some of which are discussed in the following:

Barrelfish [BBD+09, BPS+09] is an OS with a *multikernel* architecture. The multikernel manages each core by a so-called CPU driver in privileged mode and a monitor process in user mode. Together, CPU driver and monitor provide typical microkernel functions, e. g., time-sliced scheduling, IPC, and low-level resource allocation. Similar to a microkernel, device drivers and higher-level system services run as user-level processes. Barrelfish's design is based on three principles:

- explicit inter-core communication
- hardware-neutral OS structure

- share nothing, replicate instead

Corey [BWCC+08] is an OS with a *exokernel* architecture. In an exokernel architecture, the resources are provided to the application and its libraries with little abstractions. Corey allows applications to dedicate cores and address rages to kernel functions and data. As a result, the inter-core communication is reduced and optimized for the application.

Factored OS (fos) [WA09] puts scalability as the primary design constraint and replaces time with space sharing. Each service of the OS is implemented by one or more so-called servers. The servers are spatially distributed across the many-core processor such that no time multiplexing on any core is necessary. The OS's functionality is provided as a collaboration of the servers, which communicate via a message passing service. Servers are executed on top of a microkernel that provides a protection mechanism and the communication infrastructure. The applications are executed on cores separate from the OS services. The authors give the following recommendations and design principles for many-core OSs:

- Avoid use of hardware locks
- Separate OS and application resources
- Avoid global cache-coherent shared memory

HeliOS [NHM+09] introduces *satellite kernels*, which provide a single uniform set of abstractions on heterogeneous many-core processors. Each NUMA domain is managed by one satellite kernel, which has few hardware requirements in order to run on a variety of core architectures. Satellite kernels communicate via message passing. The applications are compiled to a common intermediate language (CIL) and are later compiled to the ISA of the core they are executed on. Each application specifies an affinity value as part of its meta data, which indicates the amount of communication with other applications, other instances of itself, or specific hardware. Helios was designed under the following principles:

- Avoid unnecessary remote communication
- Require minimal hardware primitives
- Require minimal hardware resources
- Avoid unnecessary local IPC

Tesselation [LKB+09] is an OS that focuses on space-time partitioning. A spatial partition is an isolated unit containing a subset of physical machine resources. Resources are divided amongst interacting software components either exclusively or with soft real-time (RT) guarantees. A global partition manager virtualizes spatial partitions by time-multiplexing. The space-time partitions are multiplexed in coarse-grain and allow user-level scheduling inside the component. The software components have unrestricted access to their resources while they are active and communicate with other components via message passing.

In the following, we deduce the common trends of many-core OSs, following the work of Vadja [Vaj11]:

1. Each core is managed by a separate kernel. The kernel provides only the fundamental services such as IPC and resource management. Higher-level OS services are implemented on top of this microkernel.

2. The many-core OSs implement IPC via a message passing mechanism.

3. Due to the decreasing number of processes per core and the increasing number of cores per chip, the trend shifts from time-sharing to space-sharing approaches. Whereas Tesselation and Barrelfish still allow multiple applications to be scheduled on the same core, fos already executes each application exclusively on each core. The latter is beneficial for the performance, since less context switches are required and fewer cache misses are expected.

4. Support for heterogeneous architectures is becoming more important. Starting from cores with the same ISA but different clock frequencies, over different internal architectures such as pipeline depth, to cores with different ISAs or even re-programmable field-programmable gate array (FPGA) cores.

2.3.3 Combination of Real-Time and Many-Core OSs

In this section, we combine the concepts of RTOSs with the design of many-core OSs. Many-core OSs propose to manage each core by a local microkernel that provides only the fundamental services. This design is similar to microkernel-based RTOSs. The simple CPU architecture and small size of core-local memory advocates to manage each core by a RTOS kernel with a low memory footprint.

Scheduling

The RTOS kernel provides a fixed-priority preemptive scheduling algorithm and resource management services. As mentioned in Section 2.1.1 on page 9, a feasibility test for this scheduling algorithm exists.

Since all cores of the many-core processor are driven by a common clock signal, we assume that all timers and schedulers, respectively, are synchronous and exhibit no drift.

We assume a *global* deadline monotonic priority assignment, i. e., all priority levels are valid for all local schedulers. Hence, if the same set of tasks and processes, respectively, is scheduled by two RTOS kernels on separate cores, the jobs will be executed at the same time on both cores.

Mapping

The RTOS kernel and all processes scheduled by the kernel are contained in the core-local memory in order to reduce memory access latencies for instruction fetches and local data accesses. However, the memory capacity and the computational power of each core is limited. Thus, not all processes can be executed on a single core in general and a

feasible process-to-core mapping that fulfills all constraints has to be found. In general, the underlying *constraint satisfaction problem* is NP-complete. Note that the complexity of this problem might be increased by additional constraints, such as subsets of processes that have to be executed on mutually exclusive cores for fault-tolerance reasons.

The problem to find a feasible mapping of processes to cores has been extensively researched and several algorithms to solve the problem have been proposed [SC13]. These approaches are either of heuristic or exact nature. Examples of heuristic approaches are tabu search, simulated annealing, ant colony optimization, genetic algorithms, or particle swarm optimization [SC13]. Exact algorithms are commonly based on symbolic approaches such as *satisfiability modulo theories (SMT)* or *mixed integer linear programming (MILP)*. The former approach includes a pseudo-Boolean satisfiability (SAT) solver to decide the satisfiability of a decision problem and incorporates a background theory solver to interpret formulae [BSST09]. The latter approach only accepts linear constraints but ties to find an optimal solution, e. g., regarding the overall performance, the communication latency, or the energy efficiency.

In hard real-time systems, the solution of a mapping algorithm has to result in a feasible schedule on all cores and a guaranteed communication between all processes. Note that in this thesis, we assume a partitioned scheduling policy that requires a *static* mapping, so processes are not migrated between cores at runtime. Furthermore, the processes are single-threaded and therefore mapped to exactly one core.

Several authors have addressed the mapping problem in the context of hard real-time systems. Liu et al. [LGX+11] use a latency computation as background theory and perform optimization by a branch and bound strategy incorporated into an SMT solver. Kumar et al. [KCT13] create an SMT solver by combining a SAT solver and real-time calculus as background theory solver. Biewer et al. [BGH14, BAG+15] propose an SMT-based approach, where the logic solver performs static binding and routing while the background theory solver computes global time-triggered schedules. Reimann et al. [RGH+10, RLG+11] present an SMT-based system synthesis with hard real-time constraints, which includes a fixed-priority preemptive scheduling policy.

The presented approaches are required to map complex applications with a large number of processes on processors with barely enough resources. However, in this thesis we present a case study for which a feasible process mapping can easily be determined manually. Therefore, the inclusion of a solver to find a feasible mapping is left as future work.

Distributed Message Passing Service

The distributed message passing service (DMPS) provides the IPC between processes that are managed by separate kernels on different cores. It keeps the mapping of a process transparent to its communication partners. Unless the RTOSs already includes a DMPS, e. g., because it supports AMP systems, we assume that it is added to the RTOS.

As defined in Section 2.1.1 on page 13, the IPC adheres a last-is-best semantic and follows the LET concept in order to decouple communication from computation. Due to the last-is-best semantic, only one input buffer per receiving message per task is required. We assume the input buffer is located in the core-local memory of the receiving task and can be accessed by the receiving task without contention from other processes. From the

LET concept follows that the DMPS has to transmit all messages and fill all input buffers read by a process before the process is scheduled. The WCCTs for the transmission of all messages are statically derived and have to be guaranteed by the DMPS and the NoC hardware. In Chapter 4 on page 55, we show how an upper bound for the communication over the NoC can be guaranteed.

External Communication Driver

In order to implement the LET concept and to achieve logical determinism, all communication between tasks and external resources such as sensors and actuators has to be independent of the task's actual execution time. Therefore, the OS contains so-called *drivers*. Drivers are low-level routines that are executed as exactly as possible at the tasks' LET borders with deactivated interrupts [HHK03]. When a task reads a sensor value, a sensor driver reads the sensor value at the task's activation time and stores the value at the task's input buffer similar to a message. As soon as a task writes an actuator signal, an actuator driver reads the task's output buffer and forwards the signal to the actuator at the task's deadline.

Since external resources are not part of the many-core processor, we consider them to be reliable in the following. Most sensors include a digital/analog converter that contains a sample and hold logic. This sample and hold logic keeps the signal stable for a specific amount of time. Since drivers are scheduled precisely, we assume that replicated drivers on separate cores read the same value from a common sensor.

Note that a typical setup to increase the reliability of a system if only unreliable sensors are available is to connect multiple unreliable sensors to the processor. This way, each task replica can read values from a different sensor. However, since sensor values are typically noisy, the signals from different sensors will slightly differ. As a result, the task replica results will differ and a more complex voting procedure is required [Par94]. Nevertheless, we are confident that the software fault-tolerance mechanism proposed in this thesis can be adapted for systems with unreliable sensors.

We further assume that new data can always be read from or written to resources in an atomic way. In other words, we do not consider complex state-full external resources that can only be accessed in a mutually exclusive way and therefore require resource access protocols to be in place.

Fault Isolation

According to our fault hypothesis in Section 2.2.5, faults occur independently. Consider a bit flip in the memory area of one process. As a result, the process can fail and outputs a wrong result or no result at all. In case the kernel mode and MPU are configured correctly, the core-local OS kernel ensures that the remaining processes are still scheduled correctly. However, the OS kernel is stored in the core-local memory, too, and is subject to faults as well. In case a fault affects the scheduler and dispatcher or other critical kernel routines, all processes scheduled on the core with a failed OS kernel fail as well. Therefore, there is no fault isolation on one core.

In order to prevent a failed process or OS kernel to overwrite data in the memory of another core, we leverage the core-local MPUs. During initialization, the OS kernel configures the local MPU to prevent all write accesses from other cores to the local RAM except for the `.mailbox` section that is required to implement the DMPS. We assume that the MPU can be configured such that each core can only write to a specific part of the `.mailbox` section. Thus, a failed core cannot overwrite messages from other cores in the memory of another core. To prevent dual-point failures, i. e., a failed MPU and the failure of another core that abuses the missing protection, the OS periodically checks the state of the MPU. Note that the MPU cannot prevent soft errors in the core-local memories, it is merely required to establish fault isolation between cores.

In order to prevent a failed OS kernel from triggering another core's reset via the ICR mechanism, the ICR mechanism's interface requires a special authentication, e. g., a specific sequence of at least two instructions or a specific bit pattern that has to be written to at least two memory cells. Therefore, a single bit flip in any instruction of the privileged OS code cannot trigger the reset of another core. Additionally, the respective ICR function contains a plausibility check to prevent a single bit flip to alter the destination core, e. g., by comparing the function's parameter with a value stored at a specific memory location. To prevent dual-point failures, the OS function is periodically compared with the correct version stored in the reliable external memory.

There still exists a residual probability of a core failure that is not isolated to one core. For example a failure that disables the MPU in combination with a failure that abuses the missing protection before it is detected by the periodical check or a failure in the ICR function that circumvents the plausibility check in combination with a failure that calls the ICR function with a wrong parameter before the periodical comparison. However, such combination of failures within the check or comparison period have a very low probability and are neglected in the following. In other words, we assume that fault isolation between cores can be established.

2.4 Summary

In this chapter, we have presented the relevant background of this thesis. First, we have introduced the fundamental constraints and concepts of real-time embedded systems. In this thesis, we consider tasks sets with harmonic periods, hard deadlines, no phases, constraint deadlines, and a global deadline-monotonic priority assignment. Next, we have defined important measures of safety-critical systems, such as the reliability and the availability. Afterwards, we have described the common design principles of many-core processors. This thesis focuses on COTS many-core processors with homogeneous cores and two NoCs. Each core contains CPU, a timer, an ICR mechanism, a scratch-pad RAM protected by an MPU, and an NI. Each wormhole-switching NoC with a 2D mesh topology implements the dimension-order (XY) routing policy with round-robin early access arbitration and connects a reliable off-chip memory to the many-core processor. The fault hypothesis of this thesis considers transient faults in the cores and the NoCs of the many-core processor. Finally, we have combined the concepts of real-time and

many-core OSs. In this thesis, each core is managed by a local RTOS kernel that provides an online partitioned fixed-priority preemptive scheduling algorithm and a DMPS.

3 Related Work

This chapter discusses the related work of this thesis. First, we survey existing approaches to provide dynamic guaranteed service (GS) communication on NoCs. Afterwards, we present an overview of fault-tolerance mechanisms focusing on software-implemented mechanisms for many-core processors.

3.1 Dynamic GS Communication on NoCs

One possibility to provide dynamic GS communication on NoCs is based on hardware support. In the following, we will provide an overview of such possibilities. If the NoC hardware does not support GS communication, mathematical models can be used to determine the upper bound of the packet latencies. We discuss such mathematical approaches afterwards.

3.1.1 Hardware-based Approaches

A common hardware-based approach to provide dynamic GS communication is to construct contention-less NoCs, so packets cannot be blocked by other packets and the worst-case latencies are trivial to determine.

Several authors propose to prevent contention by TDM [GDR05, KSWJ06, PK08, SBSK12, WGO+13]. Under TDM, time slots are assigned to the packets of a static traffic pattern at design time. If the traffic pattern is not known at design time, the assignment of packets to time slots has to be modified at runtime. Winter and Fettweis [WF11] present a hardware-implemented solution to allocate GS virtual channels at runtime. Other hardware solutions employ a probe packet that tries to reserve free time slots [KB10, LJL14]. Stefan et al. [SNG12] propose an online time slot allocation algorithm based on backtracking in software.

Apart from TDM, another approach to prevent contention in NoCs is to introduce virtual channels with different priority levels [ST11, BS05]. Offline feasibility algorithms allow to determine whether all latency requirements of a static traffic pattern are guaranteed [SB08, Shi09, BIS10, IHB15]. In order to support dynamic traffic patterns, some authors propose an application programmable interface (API) in hardware to allow the software to reserve a new route in the network at runtime [CASM11, CCM14, RCM13, RCM15].

Nikolić et al. [NYP14] present an algorithm to determine the worst-case communication delays on NoCs with priority virtual channels, for applications that are implemented

according to the Limited Migrative Model (LMM) [NP12, Nik15]. The LMM is a semi-partitioned approach, where each task is encapsulated in a dispatcher and a allocated to a statically selected subset of cores. At runtime, the dispatchers communicate with each other and choose a master dispatcher, which executes the tasks.

Kalray's MPPA [dDAPL14] many-core processor contains NIs with a traffic shaper implemented in hardware. This traffic shaper enforces a software-specified packet injection rate and hence allows to determine the worst-case delays and buffer sizes with the help of network calculus [LMJ+09]. With this hardware-based solution, the MPPA is an exception from the commercially available many-core processors, since it provides means to ensure GS communication in its NoC. Most other consumer-grade COTS many-core processors do not contain a NoC with hardware support for GS communication, since multiple physical channels, traffic shapers, virtual channels with priority levels, or a TDM scheme typically require more buffers or chip area and hence result in larger unit costs.

3.1.2 Mathematical Approaches

Most commercially available many-core processors are designed to provide best-effort service and contain a NoC without virtual channels [DNNP14, AJEF15].

Kiasari et al. [KJL13] provide an overview of mathematical analysis methods to determine the latency of packets in such NoCs, namely queuing theory, data-flow analysis, scheduling analysis, and network calculus. Queuing theory considers only the average properties of the network. Data-flow analysis is able to determine the worst-case latency and additionally allows to model dependent flows. However, restrictive models of computation (MOCs) such as synchronous data-flow (SDF) or cyclo-static data-flow (CSDF) have to be used. As mentioned above, scheduling analysis provides feasibility tests for flows in a NoC if the NoC contains virtual channels with different priority levels. Network Calculus is a methodology to derive worst-case latencies and backlog bounds, which uses an elegant abstraction with arrival curves and service curves on top of a min-plus algebra [KJL13].

Network calculus is designed for forward networks in which the service curve of a router is independent of the service curves of the other routers. However, the router-level flow-control in NoCs introduces back-pressure, hence the service curve of a router depends on the service curve of the next router [DNNP14]. Qian et al. [QLD09, QLD10] extend network calculus to calculate the end-to-end delay bound of a flow in a NoC with wormhole switching and flow control. The extension models the flow control as another service curve. It operates based on a contention tree that covers direct and indirect contention by constructing complex contention scenarios from three basic contention patterns.

Ferrandiz et al. [FFF11] showed that modeling the flow control as another service curve is pessimistic. The authors introduce a so-called wormhole section, that is a set of consecutive routers shared by a static set of flows. They compute the service curve of a section offered to an input flow by concatenating all contained routers and subtracting the arrival curve of all intersecting flows.

As mentioned by Dasari et al. [DNNP14], the idea of wormhole sections does not scale with an increasing number of traffic flows. If the number of flows with short intersecting parts increases, the wormhole sections get more fragmented. In the end, every router is treated as an individual wormhole section and the results are pessimistic again [DNNP14].

The problem of determining the worst-case delays in networks with wormhole switching was addressed by several authors without using network calculus as well [Lee03, RMB+13, FFF09]. The common approach is to recursively analyze the contention in each router on the path of each flow of packets [FFF12]. To determine the worst-case scenario, all sources are considered to inject packets at the maximum rate and all buffers are assumed to be full initially [DNNP14].

Dasari et al. [DNNP14, Das14] enhance this recursive calculus method by leveraging the input arrival patterns known from network calculus. Due to the blocking semantics, they identify a minimum inter-release time of packets and incorporate this into the recursive algorithm. This allows the analysis to prune packets that will not be present at the time the analyzed packet reaches a router. Additionally, Dasari et al. parametrized the algorithm to be able to adjust for the trade-off between complexity and tightness.

Abdallah et al. [AJEF15] focus on the same problem and present an algorithm to compute the worst-case traversal times of static flows in the NoC. In contrast to Dasari et al., the authors do not consider the application-level arrival patterns but leverage the fact that two indirectly conflicting flows are not necessarily blocking each other, but can be forwarded both in a pipeline manner, as described by Lu et al. [LJS05].

While the discussed mechanisms allow to determine the worst-case transfer latencies (WCTLs) on NoCs without hardware support for GS communication, they all assume a static set of flows captured in a static traffic pattern. However, soft errors in cores and their repair require a dynamic modification of the traffic pattern, especially if data has to be copied from the external reliable memory.

Tagel et al. [TEHJ11] present a communication model and synthesis approach to guarantee the communication latencies on NoCs without hardware support. Using a heuristic approach, the authors propose to execute the communication synthesis method not offline but at run-time whenever the SoC is reconfigured. However, this approach requires to know changes in the traffic pattern in advance, in order to compute all communication latencies and check if all real-time constraints are still met. Since this assumption does not hold for reactions to soft errors, a mechanism that guarantees the communication latencies of all possible traffic patterns is needed.

3.2 Fault-Tolerance Mechanisms for Many-Core Processors

The subject of fault-tolerance and reliability is of great interest in academia and industry [GRSRV06, Sor09, KK10, Bir14]. With the advent of multiple cores per processor, their inherent spatial redundancy was soon leveraged by fault-tolerance mechanisms. An excellent overview of fault-tolerance mechanisms for multi-core processors is provided by Gizopoulos et al. [GPA+11]. In the following, we provide an overview of hardware-based fault-tolerance mechanisms with a focus on multi- and many-core processors. Afterwards, we detail software solutions that are applicable on COTS processors as well.

3.2.1 Hardware-based Fault-Tolerance Approaches

Baleani et al. [BFM+03] present different SoC architectures for safety-critical automotive applications, including lock-step, TMR, and dual lock-step configurations. The authors state the overhead in terms of chip area for configurations is 6%–30%. Similar works propose configurable [ARJS07] or dynamically adjustable [LIMM07] redundancy configurations, or support mixed-critical applications [WCS09]. Smolens et al. [SGK+04] propose a fingerprinting technique that summarizes the history of executed instructions into a cryptographic signature. The fingerprints of two processors can be compared in a DMR configuration and reduce the error-detection effort.

The mentioned approaches have in common that they require a reliable voter, which compares and tallies the results of the replicated executions. To increase the reliability of the voter, Patooghy et al. [PMJ+06] propose a hardware-based mechanism that leverages temporal redundancy. A disagreement unit, which was originally proposed to switch in spare replicas, compares the replicas' results again and switches in a spare voter if a failure of the first voter is detected. Sloan and Kumar [SK09] present a fault-tolerance mechanism for many-core processors in which the voting logic is integrated into the NoC routers to allow distributed voting.

Li et al. [LRS+08] studied the effects of faults and state that 95% of the unmasked faults are detected via simple symptoms. Thus, the authors propose low-cost hardware monitors to detect software anomalies such as fatal hardware traps, high OS load, and hangs. When an anomaly is detected, the control is transferred to the firmware that initiates further diagnosis. This method was adapted by Hari et al. [HLR+09] for multi-threaded applications.

However, all discussed solutions share the drawback of specific hardware prerequisites. While the chip area overhead might be small compared to COTS chips, the cost of producing such specialized chips is high if not a large number of chips is produced.

Hardware/Software Co-Design Approaches

Several authors addressed fault-tolerance and robustness in hardware/software co-design approaches. Hardware/software co-design combines the design of hardware and software components in a single effort [Sch13]. Depending on given optimization criteria, hardware components are allocated (selected), software tasks are bound (mapped) to them, and a task schedule is determined.

Xie et al. [XLK+04, XLK+07] incorporate reliability into a hardware/software co-design paradigm for embedded systems. They introduce an allocation and scheduling algorithm that selectively duplicates critical tasks to detect soft errors. The task criticality is specified by the user or derived from the number of preceding tasks that require the output. The authors focus only on error detection and lack to derive the resulting reliability of the system.

Reimann et al. [RGL+08] present a system synthesis approach for dependable embedded systems, which automatically performs redundant task binding, namely simplex, duplex, and TMR, as well as the voter placement. The voter is implemented either in hardware or in software. The design space exploration evaluates the introduced dependability in

terms of the mean time to failure (MTTF), to quantify reliability, and the mean time to unsafe failure (MTTUF), to quantify safety, for different task replications schemes together with additional objectives like power consumption and latency. The reliability of the voter depends on the design space of its implementation, i. e., the number of FPGA slices and RAM used. However, the approach does not target homogeneous COTS many-core processors with same SERs on all cores, including the voter core.

Sun et al. [SGD15] propose a highly-automated formal approach that provides fault-tolerance in a distributed multi-processor system. Their approach is based on discrete controller synthesis. For a given definition of the system, a controller code is generated that detects processor failures from missing heartbeats and performs offline determined task migrations to continue the execution of the system. The authors assume that all processors are fail-silent, which is typically not the case for consumer-grade devices.

Similar to hardware/software co-design is the idea to consider fault-tolerance over multiple layers, namely from the hardware features to the application itself [HBD+13, HBZ+14, SAB+15]. Sabry et al. [SAC14] propose such a cross-layer reliable system design that combines the error resiliency at different layers of abstraction. The authors integrate a fault-tolerance buffer into the on-chip SRAMs. Information on reliability from the hardware level is propagated to the application level, which contains a checkpointing and rollback mechanism. The authors optimize buffer size and checkpointing frequency to minimize the energy overhead with given performance and size constraints.

Considering the combination of hardware and software to increase the fault-tolerance is certainly a promising approach. However, the hardware of the COTS consumer-grade processors we consider in this thesis cannot be changed and hence, the discussed solutions are not applicable.

3.2.2 Software-based Fault-Tolerance Approaches

Software-based fault-tolerance mechanisms for consumer-grade COTS processors do not share the high unit costs of hardware-based approaches. A good overview of software-implemented hardware fault-tolerance is given by Goloubeva et al. [GRSRV06]. In the following, we first introduce arithmetic coding techniques and compiler-based solutions. Next, we discuss approaches that combine fault-tolerance and real-time scheduling policies. Finally, software-based methods that leverage the spatial redundancy of many-core processors are presented.

Arithmetic Coding

Information redundancy in form of coding is a common way to increase the fault-tolerance of hardware and software [Dub13]. While parity codes and linear codes such as Hamming and cyclic redundancy check (CRC) codes only protect data, arithmetic codes protect operations as well [Dub13]. The simplest representative of arithmetic code is the AN code. AN encoding is the multiplication of the data N with the code A, with $N, \mathsf{A} \in \mathbb{N}^+$, hence the name of the coding scheme. Note that the symbol N denotes the number of replicas under the NMR scheme in this thesis. Therefore, we substitute N with X in the following but still refer to the arithmetic coding scheme as ANBD coding.

Errors are detected by checking whether the division of an encoded value by A has a reminder. The multiplication is an *invariant* with respect to the addition and subtraction ($AX_1 + AX_2 = A(X_1 + X_2)$), so errors during addition of two encoded values can be detected. The underlying idea is that a faulty arithmetic operation does not preserve the code with a high probability, so a faulty operation most likely results in a non-valid code word [Avi71]. Thus, AN coding protects against modified operands and faulty operation.

However, an error that exchanges the operands or the operator itself leads to a wrong value but a valid code word, so it remains undetected. For this reason, Forin [For90] introduced ANB coding by adding a per-variable static signature B. The arithmetic operations are adapted such that they ensure that the result is again correctly encoded and has a signature that depends only on the signatures of the input variables. Since these variables and their signatures are known at compile time, the resulting signature after the operation is known as well and can be compared at run time. For example, consider the variable X_i that is assigned the static signature B_i and the variable X_j with the static signature B_j. Each variable is encoded by $X' = A \cdot X + B$. Hence, if both variables are added, the resulting signature $B_i + B_j$ is statically known and reveals if an operand or the operator was exchanged.

Additionally, errors can result in a lost update of one or both operands. Forin [For90] proposes to protect against outdated operands by introducing a sequence counter D that counts variable updates, resulting in ANBD coding. Under ANBD coding, the variable X is encoded as $X' = A \cdot X + B + D$. The expected value of the sequence counter D after and operation with ANBD-encoded variables depends on the operation and has to be computed dynamically.

Forin included ANBD coding in the *vital coded processor*. The vital coded processor executes applications that are completely ANBD-coded but requires special hardware features to store and check signatures.

As Schiffel [Sch11] pointed out, Forin lacks to discuss implementation of operations other than addition and branch statements. Schiffel and Fetzer [WF07] propose an ANBD-coded interpreter for binary programs, which encodes the executed program at load time. The interpreter detects corrupted data and faulty executions of the executed program without special hardware requirements.

In [SSSF10a], Schiffel et al. replaced the binary interpreter by a compiler that automatically encodes an application either with AN-, ANB, or ANBD-code. Note that only register variables are encoded, all values loaded or stored in the memory are not encoded. The same authors compare the runtime overhead with compiler-based approaches, cf. Section 3.2.2 on the facing page. The measured throughput of compiler-based approaches is higher than the throughput of ANBD coding [SSSF10b]. However, the error detection rate of the latter is higher [SSSF10b].

ANBD coding protects against errors in faulty operations, modified operands, capabilities exchanged operands, exchanged operators, and lost updates [Sch11]. Thus, ANBD coding is capable of protecting against data and control flow errors [Sch11]. However, encoding the complete application significantly prolongs the execution time since more computational effort is required for the encoded operations, which renders the application

of information redundancy methods difficult under real-time constraints. Additionally, the hardware redundancy of many-core processors is left unused.

Compiler-based Approaches

Duplicating code and data at the source code level was proposed by Rebaudengo et al. [RRTV99]. At instruction level, the idea of Error Detection by Duplicated Instructions (EDDI) was presented by Oh et al. [OSM02]. The proposed software fault-tolerance mechanism replicates individual instructions while storing the results in separate registers of a single CPU. Comparing the results of the registers allows to detect errors. Rice et al. [RCV+05] build on the work of Oh et al. and present a compiler-based approach named *Swift*. Swift employs unused instruction-level resources to decrease the runtime overhead. Rice et al. [RCA07] extended their work and included *Swift-R* and *Trump*. Swift-R adds recovery to the error detection functionality of Swift by creating two copies of an instruction and by employing a TMR scheme. The authors report an execution time of 1.99 normalized to an unprotected execution. To reduce this overhead, Rice et al. present Trump that employs AN encoding to store the redundant data in two instead of three registers. Trump has a normalized runtime of 1.36. To further improve the reliability, the authors propose to statically analyze the source code, in order to find invariants and add mask instructions that ensure them. The authors state that the additional performance costs for masking are negligible.

In order to reduce the performance overhead of EDDI, Borodin et al. [BJV07, BJHV09] propose to apply no or only weak instruction-level fault-tolerance techniques to uncritical parts of an application and use the time and resource gains to apply stronger fault-tolerance techniques to the more critical parts.

Shafik et al. [SRP+13] propose a software modification tool that adds a combination of three software-only fault-detection mechanisms. First, it replaces the original data types and arithmetic operations with their encoded versions at the source code level. After compilation, jump instructions are protected by label signatures and check-pointing routines are added at assembly level. The authors report a performance overhead of 83%.

The discussed compiler-based approaches are designed for single-core processors and significantly increase the runtime of the application. The increased runtime can hinder the application of compiler-based approaches in real-time constraint embedded systems, where the computational power is low and deadlines have to be guaranteed.

Rehman et al. [RSKH11, RKSH14] propose to estimate the effect of hardware faults on software errors by introducing the instruction vulnerability index (IVI). The IVI considers spatial and temporal vulnerabilities from the processor architecture, e. g., the pipeline stages. Based on this value, high-level source code reliability transformations such as data type optimization and loop unrolling are performed. This decreases the number of critical instruction executions and can be combined with instruction-level approaches discussed above. Rehman et al. use an instruction set simulator (ISS) to simulate a Leon-II processor with ECC protected data and instruction caches. Faults are injected based on a physical fault model that includes the neutron flux rate. Rehman et al. suggest 10, 50, and 100 faults/10MCycles, which is equivalent to an SER λ of $1 \cdot 10^{-6}$, $5 \cdot 10^{-6}$, and $1 \cdot 10^{-5}$. The fault injection rate for the different components of the processor (register

file, PC, ALU, etc.) are determined from the component's chip area. In [RTK+13], Rehman et al. combine the compiler-based reliability optimization mechanisms with real-time constraints. They profile the functional (failures) and temporal (missed deadlines) reliability of multiple versions of functions compiled with different performance and reliability trade-offs. Based on the measured profiles, scheduling tables are constructed at compile time. These scheduling tables are used by the OS to dynamically select a suitable version of a function depending on the behavior of its predecessors. While the approach of Rehman et al. considers real-time constraints, it is designed for single-core processors. Therefore, it is orthogonal to our approach of leveraging the spatial redundancy of many-core processors.

Compiler-based approaches require the software's source code to be available for recompilation. This is often impossible as many vendors distribute software as binaries [Döb14]. The fault-tolerance mechanism presented in this thesis does not share this requirement and can protect binary software entities, as long as the task's interfaces are specified accordingly.

Scheduling Approaches

Most discussed compiler-based fault-tolerance mechanisms extend the execution time of the software and do not consider the effects of this behavior on the real-time constraints of the system. In the following, we discuss scheduling policies that combine fault-tolerance and real-time requirements. An excellent overview of such scheduling policies is given by Krishna [Kri14].

Burns et al. [BDP96] present a feasibility analysis of fault-tolerant real-time task sets. The authors assume that a fault can only affect one task at a time and is detected at the end of that task's execution. If a fault is detected, the task is re-executed.

Tchamgoue et al. [TSH+15] present a compositional real-time scheduling framework which uses time redundancy to tolerate faults. Each component contains a fault manager which detects the failure of a task and starts the task's associated backup strategy, i. e., a backup task. The authors analyzed the schedulability considering a rate monotonic scheduling algorithm.

Izosimov et al. [IPEP08] present a hybrid real-time fault-tolerant scheduling approach for single core processors. If a fault is detected, the task is re-executed. Offline schedules guarantee that the deadlines of hard real-time tasks are always met, while the execution of soft real-time tasks might be canceled under the presence of faults. The approach is extended for distributed embedded systems which enable spatial redundancy by Pop et al. [PIEP09].

Saraswat et al. [SPM09] propose a fault-tolerance mechanism for mixed-critical task sets. Tasks can have either hard or soft real-time constraints. Soft real-time tasks are scheduled by a constant bandwidth server (CBS). Task dependability requirements are either none, only permanent, or transient and permanent faults. Transient faults are tolerated using checkpointing with roll-back recovery, which is accounted for in each task's WCET. Permanent faults are handled by restarting affected tasks on other processors. The authors present a greedy online heuristic to ensure all deadlines are met and to select the destination processor in case of permanent faults.

Haque et al. [HAZ13] optimize the energy consumptions for reliable real-time applications on multi-core processors with dynamic voltage scaling (DVS). A lower voltage saves energy, but also increases the SER. By replicating tasks on several cores, the authors propose a heuristic that achieves a certain reliability target with minimum energy. The authors assume that each task copy conducts an acceptance or sanity check, and as long as one task finishes successfully, the reliability target is achieved.

Axer et al. [ASE11] present a reliability analysis for mixed-critical real-time task sets. The execution of each periodic task instance is separated by checkpoints and replicated on multiple cores depending the task's criticality. Faults are detected by comparing the hardware-generated execution fingerprints generated of the task replicas. The presented analysis considers temporal failures that stem from re-execution starting from the last checkpoint in case of an error as well as logical failures.

Mottaghi and Zarandi [MZ14] propose a EDF-based scheduling policy that tolerates single and multiple transient faults. Depending on the core utilization and a task's defined criticality threshold, the scheduling policy marks runnable tasks either critical or non-critical. The criticality is then used to assign a fault-tolerance method, namely task replication on two cores or checkpointing with roll-back recovery. The authors assume fault detection is performed by the task itself using sanity checks.

Girault et al. [GKSS03] present an algorithm for automatically obtaining distributed and fault-tolerant static schedules. The algorithm is based on a list scheduling heuristic with an active task replication strategy and includes the scheduling of inter-task messages. The fault-tolerance is specified as the maximum number of processor failures. The authors assume fail-silent behavior of the processors, which eases fault detection.

Eles et al. [EIPP08] present a system synthesis approach for fault-tolerant hard real-time systems. They focus on the handling of transient faults using both checkpointing with rollback recovery and active replication. The system synthesis algorithm decides on the assignment of fault-tolerance policies to processes, the optimal placement of checkpoints and the mapping of processes to processors, while satisfying the timing constraints of the application. The authors assume the software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant.

Bagheri and Jervan [BJ14] extend the system-level design frameworks for NoC-based multi-core processors of Tagel et al. [TEHJ11] to support mixed-critical task sets. The framework considers only a subset of tasks to have predictability and dependability requirements. Fault tolerance is achieved by check-pointing, fault detection, rollback, and re-execution. The schedule is generated with adequate recovery slacks to enable re-execution without deadline misses. The authors do not provide any details about the fault-detection mechanism.

The presented scheduling policies mainly rely on re-execution to increase the fault-tolerance of the system. An orthogonal approach is to increase the reliability of a multi-core system by adjusting the task mapping [SSKH13]. While static task mappings, which try to balance the temperature between the cores, can increase the lifetime of the system, they cannot overcome the faults incurred in the system [SSKH13]. Therefore, researchers have investigated global scheduling policies that adapt the task mapping at runtime based on precomputed mappings [LKP+10, DKV13] or completely dynamically [AFAL07,

CM11, RDMDY15]. These task remapping strategies are difficult to apply under real-time constraints, since the migration effort has to be considered carefully.

All discussed fault-tolerant real-time scheduling and mapping policies assume a reliable fault detection mechanism. However, if implemented in software, the fault detection mechanism is subject to errors as well. Furthermore, most discussed approaches require a fault-tolerant OS kernel that implements the scheduling policy.

Fault-tolerant OS. David and Campbell [DC07] combine techniques such as exception handling, code reloading, component isolation and restarts, service restarts, watchdog timers, transactional components, and process level checkpointing and restore to build a self-healing OS named *Choices*. The authors tested the recovery rate of the named techniques on a simulated hardware platform where faults were injected. However, Choices does not support real-time applications. Additionally, most named techniques require additional runtime and are therefore difficult to apply under real-time constraints.

Borchert et al. [BSS13] present a software-based memory error protection approach that is used to harden OSs. The authors use aspect-oriented programming to add replication and encoding schemes to OS data structures. While single bit flip fault injections experiments show that the number of failures can be reduced significantly, the approach is only possible for object-orient OSs such as eCos.

Hoffmann et al. [HBD+14] evaluated the fault-tolerance of statically and dynamically configured OSs. In contrast to dynamically configured OSs such as eCos or FreeRTOS, where tasks can be added and removed at runtime, the task set and its properties is defined at compile time in statically configured OSs such as OSEK/OS and AUTOSAR/OS [HBD+14]. The authors show that statically configured OSs are inherently more reliable, since less instructions are executed and the static configuration data can be stored in a read-only memory. Hardening the dynamic OS with software-based methods similar to the ones proposed by Borchert et al. and with hardware-based measures such as an MPU and watchdog allows to achieve the same robustness level as the static OS. However, the overhead in terms of memory consumption and runtime are significant.

Task-Replication Approaches

Most discussed arithmetic coding concepts, compiler extensions, scheduling policies, and fault-tolerance OS designs leverage information and temporal redundancy. Adapting hardware fault-tolerance concepts such as NMR has been proposed at large scale for distributed systems [KDK+89, KWQ+12] and ECU networks [AFK05], at machine scale for SMP machines with multiple processors [SMR+07], and for single-core processors with simultaneous multi-threading[1] [MKR02, GSVP03]. With the advent of multi- and many-core processors, leveraging the inherent spatial redundancy to implement fault-tolerance mechanism on a single chip became attractive.

Derin et al. [DCT+13] present the MADNESS project, which includes an adaptive run-time application mapping approach to allow graceful degradation in the presence of faults. Applications have to follow the polyhedral process network (PPN) MOC. Faults

[1] Also known as *hyper-threading*

are detected by self-testing routines in hardware. For critical applications, an NMR task replication scheme is proposed as well.

Bolchini et al. [BMS12] present a dynamic scheduling solution to achieve fault tolerance in many-core architectures. The TMR-based fault-tolerance mechanism replicates a task on three cores within a cluster of STMicroelectronics's P2012 many-core processor. Each cluster has a cluster controller, which is a core that is particularly resistant to failures. A comparison unit is instantiated on the cluster controller to analyze the results of the tasks on demand. The authors consider transient and permanent faults and dynamically schedule task replicas on healthy cores. In [BCM13], the authors add duplication with comparison (DWC) and DWC plus re-execution as possible reliability mechanisms. A software layer above the operating system dynamically selects an appropriate reliability mechanism based on run-time metrics in order to achieve the best possible balance between reliability and performance. However, the reliability depends on the specific hardware architecture of the P2012 with the hardened cluster controller for the comparison unit. Additionally, the fault model aims at handling a maximum number of concurrent faults instead of considering randomly distributed faults.

Döbel et al. [DHE12] present an OS service that provides software-implemented redundant multi-threading to unmodified binary-only applications called *Romain*. The OS service is part of the L4/Fiasco.OC microkernel OS. The framework allows to detect and recover from single-event upsets in hardware. The Romain service itself is implemented within a *reliable computing base (RCB)*. The RCB is an analogy to the trusted computing base (TCB) in security research. The RCB is defined as "a subset of software and hardware components that ensures the operation of software-based fault-tolerance methods and that we distinguish from a much larger amount of components that can be affected by faults without affecting the program's desired results" [ED12]. In [DH12], Döbel and Härtig propose to execute the RCB on radiation-hardened cores. Döbel et al. [DMH14] also explicitly consider the effect of replica and RCB placement for heterogeneous many-core processors. Döbel and Härtig [DH13] investigate the runtime overhead of Romain and suggest to protect the RCB using compiler-based mechanisms, namely SWIFT or ANBD encoding. The insightful work of Döbel [Döb14] shows that the fault-tolerance of applications can be increased by software-implemented OS-level mechanisms that are based on process replication. In his experiments, Döbel [Döb14] employs the fault injection framework FAIL* [SHD+15] and measures fault coverage of single bit flips.

Höller et al. [HRIK15] focus on the complete spectrum of dependability, which includes not only reliability and availability but also security. For this reason, the authors propose an approach based on software replication that compiles the replicas diversely [HKR+15].

All approaches discussed so far do not consider real-time requirements of the system. The following two works include this as well.

Huang et al. [HBR+11] present an analysis and optimization method for fault-tolerant task scheduling on multi-core processors considering real-time constraints. Fault-tolerance is achieved using a combination of replication and re-execution. The system-level reliability is computed by a binary tree analysis considering permanent and transient faults. A multi-objective evolutionary algorithm optimizes the fault-tolerance policy assignment, the mapping of tasks to processing elements, and the static time-triggered schedule of tasks as well as messages. Huang [Hua14] uses the evolutionary algorithm as a design space

exploration engine for fault-tolerant embedded system design on MPSoCs. The software application is represented as a Kahn process network (KPN). In order to consider the fault coverage of the comparison unit in the reliability analysis as well, Huang refers to the results of Schiffel et al. [SSSF10b], which state that the fault coverage grows exponentially with a linearly growing runtime overhead. Huang evaluates the relationship between the number of replicas and the voter coverage with the number of detected unrecoverable faults and the SDCs. Note that the fault coverage results of Schiffel et al. only specifies the number of replica failures that are not detected by the voter or the ANBD code, respectively. However, the fault coverage does not include the reliability of the voter, i. e., faults inside the voter with lead to an SDC even though all replica results are correct. In his experiments, Huang assumes an abstract task-level fault rate. Additionally, Huang assumes the OS including the scheduler is fault-free.

Similar to our goal, Yoneda et al. [YISK15] aim to increase the reliability of an automotive ECU that includes a many-core processor. The authors propose the Duplication with Temporary TMR and Reconfiguration (DTTR) scheme: two task replicas are executed on two separate cores and their results are compared. In case of a mismatch, the execution is repeated together with a previously inactive third replica on another core to detect the failed replica. It is assumed that if the fault was transient, all three replicas will deliver the same result again. Otherwise, the faulty core is disabled and the system is reconfigured by activating spare replicas. Faults such as bit flips that require a repair mechanism are not considered. The authors assume dependable inter-core communication and that two or more cores cannot fail within two cycle times. A cycle time is similar to the hyperperiod. Yoneda et al. propose to implement the voter either in hardware, or use two voters that encode the final result. In the latter case, the actuator uses the first correctly encoded result it receives. However, the voters receive unencoded values from the replicas. Thus, the voting procedure might fail and the actuator uses a correctly encoded but wrong result. Yoneda et al. model the DTTR scheme as a Markov chain and compare the resulting reliability after a mission time with a dual lock-step configuration and a TMR plus spare configuration. The constant fault rate of the voter is assumed to be significantly smaller than the fault rate of the replicas. The authors evaluate the reliability for different cycle times. A shorter cycle time limits the reexecution possibilities and constrains the reconfiguration possibilities. Hence, a shorter cycle time results in a lower probability. In contrast, we assume that the real-time constraints are derived from the system's properties and its environment and do not allow task reexecution.

Ulbrich et al. [UHK+12, Ulb14] present the *Combined Redundancy (CoRed)* architecture, a software-based fault-tolerance approach for mixed-criticality control applications. CoRed protects an application by TMR and employs ANBD coding to increase the fault-tolerance of the software voter. The authors provide extensive experiments including a quadrocopter to analyze their approach. Similar to Döbel, Ulbrich et al. employ FAIL* [SHD+15] to inject single bit flips in the CPU's registers with the help of its hardware debugger. Hoffmann et al. [HUD+14a, HUD+14b] present insightful details about the implementation of the CoRed architecture and show that complete fault coverage for single and dual bit flips can be achieved. However, the complete fault coverage holds only within their *protection domain*, which ends by forwarding an encoded tallied result and a dynamic signature. All units that use this result as an input, e. g., another software

task or an actuator, have to decode the result and compare the dynamic signature with the static signature stored locally. Especially if the result is used by a hardware actuator, the implementation of this decode and comparison routine results in additional costs. Furthermore, errors that affect the decode and comparison routine are not considered by CoRed. The CoRed architecture also does not consider errors at OS level, e. g., within the scheduler. While the ANBD coding approach is able to detect control-flow errors within the voter, it is unable to do so in case the OS kernel failed.

3.3 Summary

In this chapter, we have presented and discussed the related work of this thesis.

There are several approaches to provide dynamic GS communication in NoCs by adapting the hardware. However, most COTS many-core processors do not offer such specialized NoCs. Mathematical methods such as network calculus or recursive calculus allow to model best-effort NoCs of such processors. The mathematical model can be analyzed to determine the WCTLs of a static traffic pattern. However, as soon as the traffic pattern changes dynamically, e. g., due to reactions to a soft error, the analysis has to be repeated and the result has to be checked according to the real-time constraints. Since this prolongs the reaction time to soft error and might even prevent a repair process completely if the real-time constraints are not met, a software-based solution to allow dynamic GS communication in best-effort NoCs is proposed in this thesis.

The literature contains a variety of hardware-based fault-tolerance mechanisms such as NMR, lock-stepping, and fingerprinting. Similar to hardware-software co-design approaches, these fault-tolerance mechanisms are not applicable for COTS consumer-grade many-core processors due to their specialized hardware requirements. Software-implemented hardware fault-tolerance approaches do not share these specialized hardware requirements. Information redundancy methods such as ANBD coding and compiler-based approaches such as EDDI and Swift increase the execution time of an application and do not leverage the spatial redundancy of many-core processors. Fault-tolerant real-time scheduling policies combine reliability and real-time requirements. However, most scheduling policies assume a reliable fault detection mechanism and scheduler implementation, i. e., a fault-tolerant OS. Some researchers have investigated the robustness and fault-tolerance of OSs, but assume only single bit flip fault models. The inherent spatial redundancy of many-core processors is exploited by several fault-tolerance mechanisms. However, most of these approaches do not consider real-time requirements or assume a fault-tolerant OS. A comparison of software-implemented hardware fault-tolerance mechanisms that leverage the spatial redundancy with respect to the research questions of this thesis is presented in Table 3.1 on the next page.

Table 3.1: Comparison of software-implemented hardware fault-tolerance mechanisms that leverage spatial redundancy of many-core processors with respect to the research questions of this thesis.

Objective	Derin et al. [DCT+13]	Bolchini et al. [BMS12, BCM13]	Döbel et al. [DHE12, DH13, Döb14]	Höller et al. [HRJK15]	Huang et al. [HBR+11, Hua14]	Yoneda et al. [YISK15]	Ulbrich et al. [UHK+12, Ulb14]	This work
Considers real-time constraints	—	—	—	—	—	(✓)[1]	✓	✓
Considers faults in the voter	—	(✓)[2]	(✓)[3]	—	(✓)[4]	—	✓	✓
Considers faults affecting the OS	—	—	—	—	—	—	—	✓
Uses physically accurate fault model[5]	—	—	—	—	(✓)[6]	—	—	✓
Includes a repair mechanism	✓	✓	✓	—	✓	—	—	✓
Supports mixed-critical task sets	—	—	—	—	—	(✓)[7]	—	✓
Is adaptable	—	—	—	—	✓	✓	—	✓
Provides a fault-tolerance analysis[8]	—	—	—	—	✓	✓	—	✓

[1] The real-time constraints influence the reliability.
[2] In terms of mapping the voter to the cluster controller.
[3] In terms of encapsulating the voter in the reliable computing base (RCB).
[4] Only considers voter coverage, does not provide an implementation.
[5] Assumes exponential fault distribution with multiple parallel bit flips.
[6] While the fault rate is exponentially distributed, the experiments are performed at high level and assume the task's failure rate to be known.
[7] Tasks with different reliability requirements are scheduled on different cores.
[8] A model of mechanism that allows to estimate resulting fault-tolerance level.

4

Dynamic Guaranteed Service Communication

COTS consumer-grade many-core processors do not provide hardware support for guaranteed service (GS) communication on the NoC. However, in order to use such many-core processors in systems with real-time constraints, a guaranteed bandwidth and bounded end-to-end latency for all packets on the NoC is essential. Without GS support by the NoC, it cannot be guaranteed that communicating tasks mapped to different cores meet all deadlines and receive all messages in time.

In this chapter, we propose a method to achieve GS communication without special hardware support such as virtual channels or a TDM scheme. We first introduce the communication model and review how packets are generated. Then, we prove that our central idea, a limited packet injection rate for all sources, indeed limits congestion in the NoC. Finally, we derive the worst-case transfer latency (WCTL) of any packet, discuss our approach, and combine it the with DMPS of the OS.

The work presented in this chapter is published in [MFRC15].

4.1 Communication Model

In order to write data to or read data from other components via the NoC, any function internally uses one or multiple store word and load word instructions with an address that belongs to the component located on another core or even outside the many-core processor. When executed by the CPU, the instruction is forwarded to the NI, which transcodes the request into a request packet $\overrightarrow{p_i} \in \overrightarrow{\mathscr{P}}$ and three flits, respectively, that are injected in the request NoC. The request packet is routed through the request network and processed by the destination component. As a result, a corresponding response packet $\overleftarrow{p_i} \in \overleftarrow{\mathscr{P}}$ is injected into the response NoC and finally returned to the requesting CPU.

The set of all request packets $\overrightarrow{\mathscr{P}}$ represents the *traffic pattern*, since it includes all read and write requests that are injected into the request NoC by all cores. Note that each request packet requires a corresponding response packet to be injected into the response NoC due to the request-response protocol of the hardware components.

4.2 Limited Packet Injection Rate Approach

We are interested to bound the time between injecting the request packet $\overrightarrow{p_i}$ and receiving the response packet $\overleftarrow{p_i}$. We denote this time as transfer latency L_i. The transfer latency L_i is composed of three parts:

1. The time required for the request packet $\overrightarrow{p_i}$ to traverse the request NoC from source core $\sigma(\overrightarrow{p_i})$ to destination core $\delta(\overrightarrow{p_i})$.
2. The time the destination needs to process the request packet $\overrightarrow{p_i}$ and to inject the corresponding response packet $\overleftarrow{p_i}$.
3. The time required for the response packet $\overleftarrow{p_i}$ to traverse back in the response NoC from the destination core $\delta(\overrightarrow{p_i})$ to the source core $\sigma(\overrightarrow{p_i})$.

In other words,

$$L_i = L_p(\overrightarrow{p_i}) + L_\delta + L_p(\overleftarrow{p_i}), \tag{4.1}$$

where $L_p(p_i)$ is the latency of the packet p_i in the request or the response NoC and L_δ is the time the destination component needs to process the request packet $\overrightarrow{p_i}$ and inject the response packet $\overleftarrow{p_i}$. An overview of the various latencies that lead to the traversal latency L_i of the packet p_i is presented Figure 4.1 on the facing page.

Our goal is to determine an upper bound of all transfer latencies in an arbitrary traffic pattern. This WCTL $\hat{L}$ is specified as the maximum transfer latency $\hat{L} = \max_{\forall \overrightarrow{p_i} \in \overrightarrow{\mathscr{P}}} L_i$.

According to Equation (4.1), the WCTL $\hat{L}$ is only achieved if the latency of the destination L_δ as well as the latency of the request packet $L_p(\overrightarrow{p_i})$ and the latency of the response packet $L_p(\overleftarrow{p_i})$ are all maximal. The request and response NoCs are identical except for the link width and flit size, respectively. However, both networks offer the same bandwidth $b = 1\frac{\text{flit}}{\text{cycle}}$ and all packets have the same size in both networks, so $\left|\overrightarrow{p_i}\right| = \left|\overleftarrow{p_i}\right| = 3\,\text{flit}$. Therefore, unless explicitly stated, we do not differentiate between the request and the response NoC in the following and denote a packet in any network as $p_i \in \mathscr{P}$. In any network the maximal latency of a packet is defined as the worst-case packet latency (WCPL) $\hat{L_p} = \max_{\forall p_i \in \mathscr{P}} L_p(p_i)$.

Hence, the WCTL is

$$\hat{L} = 2\hat{L_p} + \hat{L_\delta}, \tag{4.2}$$

where $\hat{L_\delta}$ is the upper bound of L_δ, the latency of the destination core. Note that Equation (4.2) represents an upper bound since the request packet and the corresponding response packets do not both experience the WCPL in each NoC in general.

Our goal is to guarantee the WCTL for a dynamically changing traffic pattern, i. e., we want to allow request packets to be removed from, added to, or modified in the set of request packets $\overrightarrow{\mathscr{P}}$ without any restriction of the source and destination. However, this can change the transfer latency L_i of existing packets as a result of collisions in the shared medium NoC. In the following, we show that an upper bound of the WCTL $\hat{L}$ in any

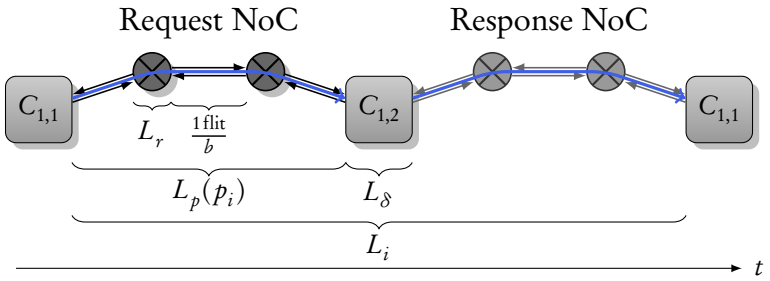


Figure 4.1: A schematic overview of the latencies in the request and response NoCs.

traffic pattern can be found if the injection rate F of all sources $C_{y,x} \in \mathscr{C}$ is limited by a certain bound. This maximum injection rate F is defined as $F = \frac{1}{\hat{L}}$. Thus, the minimum time each source has to wait before injecting the next request packet is at least $\hat{L}$. Note that due to the definition of the WCTL $\hat{L}$, any source will receive a response packet before injecting the next request packet. Hence, our approach is applicable to both synchronous and asynchronous NoCs.

For the sake of brevity, we define the system $\mathscr{S}$ that consists of two 2D-mesh wormhole switching NoCs, each with the dimension order (XY) routing policy, $y \times x$ cores, bidirectional (full-duplex) links, FIFO buffers, and a router-level flow control mechanism. Furthermore, each source in the system $\mathscr{S}$ obeys the maximum packet injection rate F.

In the following, we use some properties of the system $\mathscr{S}$ to determine the latency of a packet without any collisions. By adding the worst-case latency introduced by collisions with other packets, we derive the WCPL $\hat{L_p}$.

4.2.1 Traversal Latency

We start by analyzing the traversal latency $L_{\not c}(p_i)$, i. e., the latency of the packet p_i in one otherwise empty NoC of the system $\mathscr{S}$. The traversal latency $L_{\not c}(p_i)$ is the time required for the packet p_i to traverse the request or the response NoC from $\sigma(p_i)$ to $\delta(p_i)$ without any collisions.

Let L_r denote the latency of a router, i. e., number of clock cycles required by the router to process one flit, and b denote the bandwidth of any link $l \in \mathscr{L}$. The traversal latency is derived by adding up the latency of each router L_r and each link $\frac{1\,\text{flit}}{b}$ passed by the first flit of packet p_i on its route through the network. Since the number of hops $h(p_i)$ is defined as the number of routers passed by the packet p_i, one more link latency $\frac{1\,\text{flit}}{b}$ has to be added to reach the NI of the destination core, as can be seen in Figure 4.1. Under the wormhole switching mechanism, the remaining flit cannot be delayed by any router and follow the first flit immediately, so $\frac{|p_i|-1\,\text{flit}}{b}$ has to be added until the complete packet traversed the network, where $|p_i|$ is the size of the packet p_i in flits. Thus, the complete traversal latency is

$$L_{\not c}(p_i) = h(p_i)(L_r + \frac{1\,\text{flit}}{b}) + \frac{|p_i|}{b}. \tag{4.3}$$

Consequently, the worst-case traversal latency $\hat{L_{\not\subset}}$ of any packet $p_i \in \mathscr{P}$ in one otherwise empty NoC of the system $\mathscr{S}$ is

$$\hat{L_{\not\subset}} = \max_{\forall p_i \in \mathscr{P}} L_{\not\subset}(p_i) = (x+y-1)(L_r + \frac{1\,\text{flit}}{b}) + \frac{|p_i|}{b}. \tag{4.4}$$

Due to the dimension-order (XY) routing protocol, the maximum number of routers to be traversed in each 2D-mesh NoC with $y \times x$ routers is $y+x-1$, namely from one corner to the diagonally opposite corner. Note that the worst-case traversal latency $\hat{L_{\not\subset}}$ holds for the request and for the response NoC.

4.2.2 Blockage Latency

As soon as there is more than one packet present in one NoC, the latency of each packet potentially increases due to collisions and blockages by other packets. In the following, we use some properties of the system $\mathscr{S}$ to derive the maximum blockage latency of any packet in one NoC of the system $\mathscr{S}$.

Lemma 4.1. *Two packets of an arbitrary traffic pattern collide at most once in one NoC of the system $\mathscr{S}$.*

Proof. Suppose two packets p_a and p_b collide once, i. e., they arrive in a router r and request the same link l_c in the same cycle. The router r resolves this conflict by blocking one packet and forwarding the other one. The dimension order (XY) routing policy of the NoC implies that the remaining route of packet p_a and the remaining route of packet p_b in the NoC each have at most two parts A and B. Part A of both remaining routes is identical. It contains at least the link l_c. In part A, no further collisions of packet p_a and packet p_b are possible due to the FIFO buffers and the order-preserving property of the dimension order (XY) routing policy. Part B of the remaining route of each packet exists if $\delta(p_a) \neq \delta(p_b)$. The dimension order (XY) routing policy ensures that part B of both routes is different, thus no further collisions of packet p_a and packet p_b are possible. Without loss of generality, since the remaining routes of the packets p_a and p_b have at most two parts A and B which cannot lead to further collisions, two packets collide at most once in any router of one NoC. □

Lemma 4.2. *For an arbitrary traffic pattern in the system $\mathscr{S}$, a packet collides at most once with other packets injected by any other source in one NoC.*

Proof. Let the packet p_a be injected at t_a. Let $p_i \in \mathscr{P}_b$ be a packet in a set of packets $\mathscr{P}_b$ injected by another source at time t_i. Since all sources obey the maximum packet injection rate, the injection time of any other packet $p_n \in \mathscr{P}_b$ from the same source $\sigma(p_n) = \sigma(p_i)$ is

$$t_n \geq t_i + n\hat{L},\ n \in \mathbb{Z}. \tag{4.5}$$

Suppose the packet p_a collides with the packet p_i at t_C. According to the definition of the WCPL $\hat{L_p}$, the packet p_a and any packet $p_n \in \mathscr{P}_b$ is present in one NoC of the

system $\mathscr{S}$ for at most the time $\hat{L_p}$. Therefore, the packet p_a can only collide with the packet p_i if

$$t_a < t_C < t_a + \hat{L_p} \text{ and} \tag{4.6}$$

$$t_i < t_C < t_i + \hat{L_p} \tag{4.7}$$

hold.

According to Lemma 4.1, p_a collides at most once with p_i in one NoC. However, the packet p_a could collide with another packet $p_n \in \mathscr{P}_b$ from the same source as packet p_i. From Equation (4.6) and Equation (4.7) follows that $t_i - \hat{L_p} < t_a < t_i + \hat{L_p}$. From Equation (4.5) follows that any previous packets $p_j \in \mathscr{P}_b$ with $j < i$ is present in the NoC at most until $t_i - \hat{L_p} < t_a$. Likewise, any subsequent packet $p_k \in \mathscr{P}_b$ with $k > i$ is not injected into the NoC before $t_i + \hat{L} > t_a + \hat{L_p}$. Hence, all previous and subsequent packets of p_i cannot collide with the packet p_a in the same NoC. Therefore, packet p_a collides at most once with any packet $p_n \in \mathscr{P}_b$ injected by another source in one NoC. □

Figure 4.2 on the following page visualizes the proof of Lemma 4.2 on page 58. All depicted packets experience their worst-case latencies. By shifting the time at which packet p_a is injected by core $C_{1,1}$ into the request network back and forth, it can be seen that packet p_a collides only with one packet from the other core $C_{2,1}$ in the request network.

Due to the definition of the packet injection rate F, the number of packets present in the NoCs of the system $\mathscr{S}$ is limited. In the following, we prove that this also limits the maximum number of times any packet can collide with other packets and potentially be blocked.

Theorem 4.1. *For an arbitrary traffic pattern in the system $\mathscr{S}$, any packet $p_i \in \mathscr{P}$ collides at most $xy-2$ times with other packets in one NoC.*

Proof. In one NoC of the system $\mathscr{S}$, there exist xy sources. According to Lemma 4.2, any packet $p_i \in \mathscr{P}$ collides at most once with any packet from another source in one NoC of the system $\mathscr{S}$. Due to the dimension order (XY) routing policy and bidirectional links, the destination core $\delta(p_i)$ of the packet p_i cannot inject packets that collide with the packet p_i, since all common links are traversed in the opposite direction if there are any. Furthermore, the packet p_i cannot collide with any other packet $p_n \in \mathscr{P}$ injected by the same source $\sigma(p_i) = \sigma(p_n)$ if all sources obey the packet injection rate F. Thus, any packet $p_i \in \mathscr{P}$ collides at most $xy-2$ times with other packets in one NoC. □

The latency added by a collision $L_C(p_i)$ of the packet p_i with another packet does not only depend on the traffic pattern, but also on the internal state of the router. Since an exact analysis of the collision latency has to include details like the state of the router-internal round-robin arbitration logic, which are not statically known for a dynamic traffic pattern, we consider the worst-case behavior and assume a packet is always blocked in case of a collision. The worst-case collision latency $\hat{L_C}$ is the upper bound of the latency experienced by any packet when colliding with another packet, namely the time required by a router to arbitrate a conflict and process all three flits of the other packet.

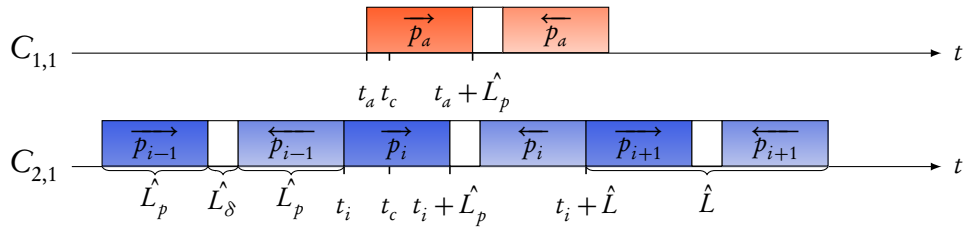


Figure 4.2: Visualization of the proof of Lemma 4.2. The core $C_{1,1}$ sends one packet p_a. The core $C_{2,1}$ sends three consecutive packets with the maximum injection rate. The time during which a request packet is present in the request network is shown as a dark colored area. Similarly, the time during which the response packet traverses the response network is depicted as a light colored area. All packets experience the worst-case latencies.

Corollary 4.1. *For an arbitrary traffic pattern in the system $\mathcal{S}$, the worst-case blocking latency $\hat{L}_B$ of any packet $p_i \in \mathcal{P}$ in one NoC is at most*

$$\hat{L}_B = (xy-2)\hat{L}_C. \tag{4.8}$$

Proof. Based on Theorem 4.1, any $p_i \in \mathcal{P}$ collides at most $xy-2$ times with other packets in one NoC. Hence, the worst-case blocking latency $\hat{L}_B$ of any $p_i \in \mathcal{P}$ is the number of collisions of the packet multiplied with the worst-case collision latency $\hat{L}_C$, i. e., the maximum amount of time it takes a router to arbitrate the conflict and process the winning packet. □

The WCPL $\hat{L}_p$ in the system $\mathcal{S}$ is the sum of the worst-case traversal latency $\hat{L}_{\not\subset}$ and the worst-case blockage latency $\hat{L}_B$ of any packet, so

$$\hat{L}_p = \hat{L}_{\not\subset} + \hat{L}_B. \tag{4.9}$$

For a given set of parameters of the system $\mathcal{S}$, Equation (4.2) calculates the WCTL $\hat{L}$ of any transfer in the system $\mathcal{S}$. The WCTL $\hat{L}$ defines the maximum packet injection rate $F = \frac{1}{\hat{L}}$. By definition, the injection rate limits the number of packets present in the system $\mathcal{S}$ at any instant of time, which again limits the number of collisions and bounds the WCTL $\hat{L}$.

4.2.3 Dynamic Traffic Pattern

In the following, we prove that the definition of the packet injection rate F allows to dynamically modify the traffic pattern, i. e., that the set of request packets $\overrightarrow{\mathcal{P}}$ can be modified at runtime without violating the guaranteed WCTL $\hat{L}$.

Theorem 4.2. *In the system $\mathcal{S}$, the source and destination of any unprocessed request packet $\overrightarrow{p_i} \in \overrightarrow{\mathcal{P}}$ can be changed without increasing the WCTL $\hat{L}$. Furthermore, in the system $\mathcal{S}$,*

new request packets can be added to the traffic pattern $\overrightarrow{\mathscr{P}}$ *and existing request packets can be removed from the traffic pattern* $\overrightarrow{\mathscr{P}}$ *without increasing the WCTL* $\hat{L}$.

Proof. The WCTL $\hat{L}$ calculated in Equation (4.2) is upper bounded if all parts of Equation (4.2) are upper bounded. The latency of the destination L_δ is upper bounded by definition. The WCPL $\hat{L_p}$ is defined by Equation (4.9). According to Corollary 4.1, any packet $p_i \in \mathscr{P}$ is blocked for at most $\hat{L_B}$ time units in the system $\mathscr{S}$. The worst-case traversal latency $\hat{L_{\not c}}$ of any packet $p_i \in \mathscr{P}$ in one otherwise empty NoCs of the system $\mathscr{S}$ is upper bounded, since the maximum number of hops in each NoC is fix. Thus, $\hat{L_p}$ and $\hat{L}$ cannot increase in the system $\mathscr{S}$, independent of the modifications in the traffic pattern and the set of request packets $\overrightarrow{\mathscr{P}}$, respectively. □

4.3 Discussion

The presented approach is designed for systems with separate NoCs for request and response packets. Separate NoCs are commonly used to prevent message-level deadlocks [HGR07]. However, in case every request packet is completely received by the destination before the response packet is created, a message-level deadlock cannot occur and it is possible to transfer packets of both types in one NoC. In this case, the presented limited packet injection rate approach can still be applied, since the WCTL $\hat{L}$ already considers each packet being blocked by one request and one response packet from every other source in the worst case.

The limited packet injection rate approach is inverse to the comprehensive analysis methods mentioned in Section 3.1.2 on page 42. Instead of taking a fixed traffic pattern as given, we shape the traffic pattern in such way that the latency is inherently bounded. In contrast to methods based on TDM, our solution is more flexible since it does not restrict injections to specific time slots but merely bounds the injection rate.

The real-time many-core OS enforces that all traffic in the NoCs is created by OS functions: The DMPS that implements the IPC between cores, the drivers that read sensor values and write actuator signals, and the initialization and repair routines, which copy all necessary instructions and data from the external memory to the core-local memory. Since they are part of the OS, the packet injection rate of these functions can be controlled and adjusted.

For example, consider the DMPS that has to guarantee that each message $m_i \in \mathscr{M}$ is delivered within its WCCT $\hat{C}_i$. The presented limited packet injection rate approach guarantees the WCTL $\hat{L}$ for every packet. The first two flits of each packet encode the destination address and the request type, respectively. Thus, only the last flit contains the actual data to be transferred. Assume the size of the message $|m_i|$ is given in bytes and the effective flit size to be 32 bits in the request and the response network. When $\hat{C}_i < \frac{|m_i|}{4}\hat{L}$ holds, the WCCT $\hat{C}_i$ of the message m_i is guaranteed and the real-time constraints are met.

According to our fault hypothesis presented Section 2.2.5 on page 28, the DMPS is subject to soft errors, too. However, a failed DMPS potentially violates the limited injection rate and hence affects the guaranteed latency of independent inter-core communication. For this reason, we follow our approach for the ICR mechanism and the MPU configuration presented in Section 2.3.3 on page 38: The DMPS is periodically checked by the OS, e. g., by computing a checksum of the respective code section or comparing the section with the correct version stored in the reliable external memory. In case a failure in the DMPS is detected, the OS kernel prevents further packet injections by resetting the local core.

4.4 Summary

In this chapter, we have presented a software solution that allows GS communication on NoCs that support only best-effort communication. Our central idea is to limit the packet injection rate of all cores. The injection rate is the inverse of the time between sending two consecutive packets. An upper bound of the packet injection rate can be implemented in software and enforced by the OS without additional hardware support. The proposed method does not depend on a static and a priori known communication. As long as no source exceeds the maximum packet injection rate, our solution supports a dynamic GS communication that changes at runtime. This is an important requirement for our system, since faults occur at arbitrary points in time and the ensuing repair process requires GS communication between the cores and the external memory. In Section 7.2 on page 114, we present an experimental evaluation of our approach for different traffic patterns.

5 Software Fault-Tolerance Mechanism

The main motivation of this thesis is to leverage the computational power of many-core processors in safety-critical systems. In order to comply with the mandatory safety standards and to achieve the respective reliability and availability targets, fault-tolerance mechanisms have to be employed. However, the hardware of low-cost COTS consumer-grade processors is not specifically designed to provide a high level of fault-tolerance and cannot be modified. Therefore, a software-based mechanism has to be found.

In this chapter, we present our software-based hardware fault-tolerance mechanism. Our mechanism is based on the well-known NMR scheme, can be adapted for each task separately, and is designed to comply with the system's real-time constraints. First, we compare different software-implemented hardware fault-tolerance approaches and justify the choice of an NMR-based mechanism. Next, we focus on the voter, the SPOF in any NMR-based mechanism. In order to improve the voter's reliability, we propose an approach with two fail-silent voters that check each other. Afterwards, we introduce our repair procedure that restores failed task replicas or voters and thus additionally improves the system's reliability and availability. Finally, we explain how the proposed fault-tolerance mechanism is integrated with the real-time many-core OS and provide a comprehensive example of the mechanism's functionality and real-time capability.

The work presented in this chapter is published in [MAL+15, MAL+16].

5.1 NMR Approach

In Section 3.2, we introduced several concepts to increase the fault-tolerance of many-core processors. Amongst the fault-tolerance mechanisms that do not depend on hardware modifications and thus can be applied on COTS processors, there are

- arithmetic coding,
- compiler-based approaches,
- scheduling policies,
- and task-replication methods.

Arithmetic coding such as ABND coding and compiler-based approaches such as EDDI and Swift increase the execution time of the software and the tasks, respectively. An extended WCET of a subset of tasks can result in a real-time system to miss its deadlines. Furthermore, both methods do not utilize the spatial redundancy of many-core processors.

Fault-tolerant scheduling policies combine real-time and reliability requirements. These policies often reserve time slots for reexecuting a task in case its failure was detected. For multi-core processors, researchers have proposed global scheduling policies that adapt the task mapping in case of a task failure. However, all policies discussed in Section 3.2.2 require a reliable fault detection mechanism as well as a fault-tolerant scheduler and dispatcher implementation. On a homogeneous many-core processor where all cores exhibit the same SER, such reliable software components require additional protection.

Task-replication methods like NMR are inherently parallel and therefore well suited for many-core processors. As explained in Section 2.1.2, an FCU failure is not only detected but can also be masked if the redundancy level is larger than two, so $N \geq 3$. Hence, no reexecution is required, which eases the application of the NMR mechanism under real-time constraints.

Furthermore, the NMR scheme is adaptable: with a reliable voter, a higher redundancy level N results in a higher system reliability for realistic SERs [KK10]. Thus, tasks with different safety criticalities and target reliabilities, respectively, can all be handled by the same NMR mechanism if a different redundancy level N for each task is chosen.

Consequently, we select NMR as the basis of our software fault-tolerance mechanism. Each safety-critical task T_i is replicated N_i times. Each *task replica* T_i^j with $j = 1, 2, \ldots, N_i$ is mapped to a mutually different core. All task replicas are executed in parallel and send their results to two common *voter* tasks V_i^1 and V_i^2, which are executed on separate cores, as shown in Figure 5.1. Note that task replicas or voters of different tasks can be mapped to the same core.

Majority Voting. In an NMR mechanism, the voter compares the results of all replicas in order to determine the correct result and to identify all failed task replicas. Several voting algorithms have been proposed in literature. The most common generic voting strategies are [LBB04]:

unanimity voting: all results have to be equal ($M_i = N_i$)

majority voting: at least $M_i = \left\lceil \frac{N_i+1}{2} \right\rceil$ task replicas deliver the correct result

plurality voting: relaxed form of majority voter, implements M_i-out-of-N_i voting, e. g., 2-out-of-5 or 3-out-of-7

median voting: selects the mid-value of replica results

(weighted) average voting: calculates the (weighted) average of replica results

The former three voting strategies are *exact* and require at least M_i replicas to deliver exactly the same result. The latter two voting strategies are *inexact* and can handle replica results that differ within a small application-specific threshold, e. g., due to sensor noise [LBB04].

The most suitable voting algorithm depends on the application and its environment, e. g., the cardinality of the voter input and output space as well as the voting algorithm's time and space complexity [LBB04]. We focus on exact voting algorithms, since they

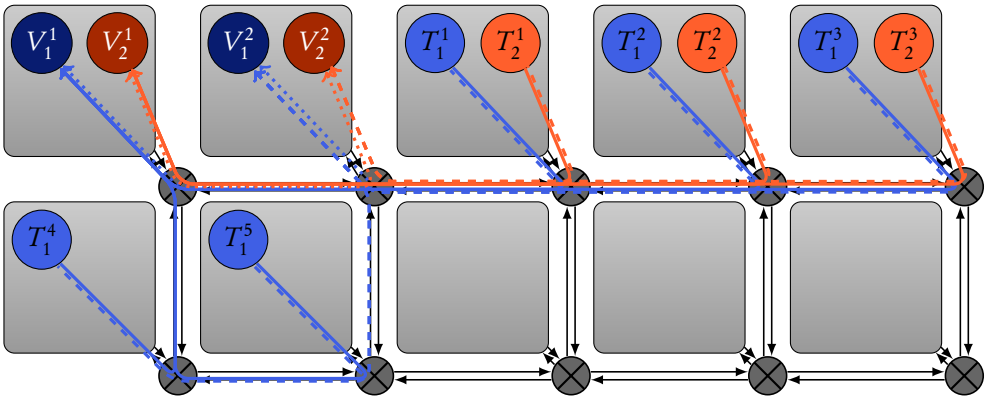


Figure 5.1: Schematic overview of the NMR mechanism on a 2×5 many-core processor. Task T_1 is replicated $N_1 = 5$ times and forwards its result to voters V_1^1 and V_1^2. Task T_2 is replicated $N_2 = 3$ times and forwards its result to voters V_2^1 and V_2^2. Solid lines represent messages sent from the task replicas to voter V_i^1, dashed lines represent messages sent from the task replicas to voter V_i^2, and dotted lines are messages that are exchanged between both voters.

are less complex compared to inexact algorithms [Par94]. Furthermore, inexact voting strategies are less suitable in safety-critical systems, since they do not fail silently in case no consensus is found, but still provide a potentially catastrophic output [LBB04].

Exact voting algorithms require task replicas that use identical input data and are deterministic at their interfaces. The input data of each task replica T_i^j, i. e., the messages and sensor values read by the task T_i, are provided by the local OS kernel's DMPS and drivers. These OS functions are designed to ensure that all replicas work with the same input data, as explained in Section 2.3.3.

Note that task replicas might perform floating point operations, which have a limited precision and introduce inaccuracies, and provide a floating point result. However, as long as all replicas use the same deterministic functions with the identical input data, their results are identical at binary level, since the same inaccuracies occur in all replicas. Hence, an exact voting algorithm can still compare the results bit by bit.

Amongst the exact voting algorithms, the unanimity voter is very strict and does not mask any task replica failure. Hence, it does only provide fault detection but no fault tolerance. The plurality voting strategy can produce a tie, i. e., it can determine two correct results. For example, in a 3-out-of-7 configuration, 3 task replicas deliver result X_1, another 3 replicas generate result $X_2 \neq X_1$, and the last replica outputs another obviously wrong result. The plurality voter is unable to decide if result X_1 or result X_2 is correct. The majority voter does not share this problem. Therefore, we use a majority voting strategy in this thesis.

If $M_i = \left\lceil \frac{N_i+1}{2} \right\rceil$ or more task replicas fail, which is the case in the previous example, the majority voter cannot decide on a correct result. This case is denoted as a *detected system failure* or detected uncorrectable failure. In this thesis, we consider M_i as constant in order to achieve a higher reliability. In other words, if some of the N_i task replicas failed, M_i

is not adapted to the smaller N_i. For example, if $N_i = 5$ and 2 task replicas have failed already, all $M_i = 3$ remaining replicas have to provide correct results.

5.2 Reliable Voter

The reliability of an NMR system $R_{\mathscr{S}}(t_m)$ after the mission time t_m without a repair procedure is given by [KK10]

$$R_{\mathscr{S}}(t_m) = R_V(t_m) \sum_{i=M}^{N} \binom{N}{i} \big(R_T(t_m)\big)^i \big(1 - R_T(t_m)\big)^{N-i}, \tag{5.1}$$

where $R_V(t_m)$ is the reliability of the voter after the mission time t_m, $R_T(t_m)$ is the reliability of a task replica after the mission time t_m, and $\binom{N}{i}$ denotes the binomial coefficient $\frac{N!}{i!(N-i)!}$.

Since the voter is in series to the task replicas and represents a SPOF, the absence of errors in the voter is of paramount importance. However, we consider consumer-grade processors with homogeneous cores that all exhibit the same SER. Like any other task, the voter task is affected by bit flips in its code and data sections, by failures in the core-local OS, and by faults in the NoC or the logic and the register file during its execution. Thus, the voter task exhibits a failure rate that is approximately similar to the failure rate of a task replica, so $R_V(t_m) \approx R_T(t_m)$. From Equation (5.1) follows that the reliability of the system $R_{\mathscr{S}}(t_m)$ is worse compared to the reliability of an unprotected simplex system without a reliable voter, i. e., $R_{\mathscr{S}}(t_m) = R_T(t_m)$.

Consider two safety-critical tasks T_1 and T_2. Both are replicated on the processor and task T_1 sends its results to task T_2. In this case, the problem of the voter as a SPOF can be circumvented by adding one majority voter for each task replica T_2^j of the receiving task, as shown in Figure 5.2. In this well-known setup, a single voter failure does not cause more harm than a single task replica failure [KK10]. However, this solution cannot by applied if the receiver of the result generated by the safety-critical task T_1 is not replicated. For example, the receiver of the result the safety-critical task T_2 might be an external actuator that is not replicated.

Therefore, we require a fault-tolerance mechanism that provides one reliable output. Hence, we have to harden the implementation of a single voter in software. For selecting an appropriate software fault-tolerance mechanism for the voter, we follow the same reasoning as for tasks: we use two voters that run in parallel on separate cores and leverage the spatial redundancy of the many-core processor. However, adding a voter-voter to compare the results of both voters just introduces another SPOF. Therefore, in contrast to the fault-tolerance mechanism for tasks, the voters have to check each other.

Both voters receive and vote on the replicas' results. However, at any time, only one voter is *active*, forwards the tallied result, and starts the repair procedure for failed replicas. In addition, each voter sends a notification to the other voter after it has successfully finished the voting procedure. The absence of this notification tells a voter that the other voter has failed. When a failure of the inactive voter is detected by the active voter, the active voter starts the repair procedure of the failed voter. In case the active voter failed,

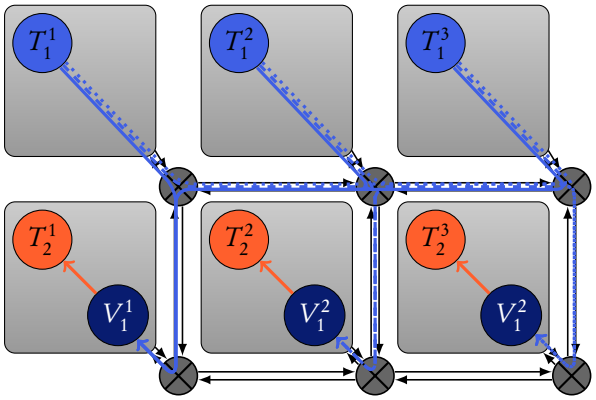


Figure 5.2: Visualization of two safety-critical tasks T_1 and T_2 on a 2×3 many-core processor. Task T_2 receives a message from T_1. Since both tasks are replicated $N_1 = N_2 = 3$ times, three majority voters can be added in between the tasks. Each replica T_1^j sends its result to all voters V_1^k. Solid lines represent messages sent from the task replicas to voter V_1^1, dashed lines represent messages sent from the task replicas to voter V_1^2, and dotted lines represent messages sent from the task replicas to voter V_1^3.

the inactive voter becomes active, forwards its tallied result, and starts the repair procedure for the failed task replicas and the failed voter. Once the failed voter is repaired, it sends notifications again but it does not become active unless the active voter fails.

Note that for the time a failed voter is repaired, the active voter remains a SPOF. However, in total the system's reliability and availability are increased since only a combination of two failures, one in both voters, can result in a system failure. In contrast to the detected system failure that occurs when a voter detects that M_i or more task replicas failed, the failure of both voters is an *critical system failure* or an SDC, since the system is unaware of its failure and cannot reach a safe state.

5.2.1 ANBD Encoding

The notification that a voter finished the voting procedure is not enough to decide whether the voter has failed. A failure can still cause a voter to skip the replica results comparison routine and send a notification immediately without having determined the correct result.

To ensure that a notification is only sent after the voting procedure was successfully executed, we depend on a mechanism to guarantee a *fail-silent* behavior of both voters. If a voter is fail-silent, it will only send a notification to the other voter and forward the tallied result if no failure occurred. To achieve this fail-silent behavior, we extend the *CoRed Encoded Voter* approach [UHK+12], which originally is designed only for the TMR mechanism.

The encoded voter approach relies on ANBD coding, which represents a combination of AN coding, per value signatures, and sequence counters. ANBD codes detect data, addressing, and sequence faults, as introduced in Section 3.2.2.

Multiplying the integer value X with a constant integer number A is the widely known AN arithmetic coding scheme. With a carefully selected constant A, it efficiently detects manipulations of the encoded value. To check if the encoded variable X' represents a valid code and was not manipulated, it has to be tested whether

$$X' \mod A = 0. \tag{5.2}$$

Typically, large prime numbers are used as A [Sch11]. However, as Hoffmann et al. [HUD+14a, HUD+14b] pointed out, the selection of A should also consider the characteristics of the hardware architecture. In our implementation, we use the *Super-As* proposed by Hoffmann et al. [HUD+14a, HUD+14b].

Let X'_j and X'_k denote two AN encoded values that are operands of an addition. The multiplication with A is an invariant with respect to addition and subtraction. Other operations such as integer multiplication and division can be adapted to ensure the result is encoded correctly afterwards, e. g., by dividing and multiplying the result with A [Sch11]. Hence, errors during the operation with encoded values can be detected, since a faulty arithmetic operation does not preserve the code with a high probability. Therefore, the result $X'_l = X'_j + X'_k = A(X_j + X_k)$ is still a valid code word when the addition itself was fault-free, since $X'_l \mod A = 0$ holds.

However, if a fault causes one of the operands or the operator to be exchanged, the resulting code word will still be valid although an error occurred. In order to detect such operator and operand errors, a unique static signature B_j for each variable X_j is added to the plain value. Since these signatures are statically assigned, the expected signature after the operation with the chosen operands is also known. Let B_j be the signature of X'_j and B_k be the signature of X'_k. The expected signature of the result X'_l of the addition of X'_j and X'_k is $B_j + B_k$. Exchanged operands or another operator would obviously result in a different signature. Hence, each variable X_j is encoded by

$$X' = A \cdot X_j + B_j \tag{5.3}$$

under this coding scheme, which is known as ANB coding.

When a fault causes a variable update to be lost, e. g., a bit flip in the memory address that causes the new value is stored in a wrong memory cell, the expected signature after an operation would be correct and the code word would be valid. Therefore, a cyclic sequence counter $D = 0, 1, \ldots, \hat{D}$ that counts variable updates is added to the plain value, which reveals outdated values. This coding scheme is known as ANBD coding and encodes each variable X by

$$X' = A \cdot X + B_j + D. \tag{5.4}$$

The expected sequence counter after an operation with ANBD encoded values has to be computed dynamically. Note that $B_j + \hat{D} < A$ has to be fulfilled when selecting the signatures B_j and the static maximum sequence counter value $\hat{D}$ [WF07].

The AN, ANB, and ANBD coding schemes require all values to be integer. However, only the voting procedure itself is encoded. The task's function can include floating point

computations since it is protected by replication. Therefore, the task replicas can encode the binary alias of their floating point result before sending it to the voter. The encoded voting algorithm operates on the encoded values and forwards the tallied and decoded result, which still represents a floating point value for the receiving task or actuator. Hence, the ANBD encoded voting procedure does not impose a restriction on the application.

5.2.2 Encoded Voting Algorithm

To protect the voting algorithm with the ANBD coding scheme, the voting algorithm has to be implemented using only encoded operations that are applied directly on the encoded values and conserve the arithmetic code. Hence, data integrity can be ensured during the execution of the encoded voting algorithm.

Let X_j and X_k represent the results of two task replicas T_i^j and T_i^k. Before each task replica sends its result to both voters, it encodes its result with the ANB code and Equation (5.3), respectively. The task replicas do not add the sequence counter D, since this requires all task replicas and both voters of one task to maintain a common sequence counter value. As soon as a repair procedure is introduced to the system, a repaired task replica or voter has to gather the current value of D from other task replicas or voters. This introduces common cause failures, as other task replicas or voters may provide a corrupt value of D. In order to circumvent another voting procedure to determine the correct sequence counter value after each repair, we design the DMPS to delete an incoming message when it is read. Therefore, when the voter collects a replica's result and this replica fails silently in the next period, the voter will not be able to read the old result again and is able to detect the replica's failure. A lost update is still possible if the DMPS and the task replica fail at the same time. However, the probability of this combination of failures is comparable to the combination of a failed task replica and a failure in the checking routine of D inside an encoded operation. Note that in contrast to the sequence counter D, the signature B_j and the integer constant A are statically known.

Both voters V_i^1 and V_i^2 receive the ANB encoded results X'_j and X'_k from the two task replicas T_i^j and T_i^k. Once each voter reads the replica value, it adds a voter-internal sequence counter D to the encoded value. This leads to an ANBD-encoded replica result, which allows to detect lost updates within the encoded voting procedure itself. Since the sequence counter D is maintained by each voter separately, no agreement policy between both voters in required.

To compare the two ANBD-encoded values, the correctness of the relation $\mathsf{X}_j = \mathsf{X}_k$ has to be evaluated. According to Equation (5.4), this relation can be written as:

$$\frac{1}{\mathsf{A}}(\mathsf{X}'_j - \mathsf{B}_j - \mathsf{D}) = \frac{1}{\mathsf{A}}(\mathsf{X}'_k - \mathsf{B}_k - \mathsf{D}),$$

which can be simplified to

$$\mathsf{X}'_j - \mathsf{X}'_k = \mathsf{B}_j - \mathsf{B}_k. \tag{5.5}$$

Hence, all comparison operations within the voting algorithm are replaced with Equation (5.5), as shown in line 7 of Listing 5.1.

```
1  int j,k,v;
2  for (j=0; j<=N_i − M_i && v<M_i; j+=1) {
3     D = B_j;
4     E = {j};
5     v = 1;
6     for (k=j+1; k<N_i && v<M_i; k+=1) {
7        if (X'_k − X'_j == B_k − B_j) {
8           E = E ∪ {k};
9           D = D + ((X'_k − X'_j) + B_j);
10          v+=1;
11       }
12    }
13 }
```

Listing 5.1: Encoded voting algorithm—Part 1

The encoded voting algorithm presented in Listing 5.1 tries to find M_i task replicas that deliver the same result. The IDs of these M replicas with an equal result are stored in the *equality set* $\mathscr{E}$, so $\forall j,k \in \mathscr{E} : X_j = X_k$. The voting algorithm ends as soon as M_i equal results have been found. The remaining replica results are not compared and the respective replica IDs are not added to the equality set even if the replicas have sent the same (correct) value. Unless M_i or more replicas failed, which is equivalent to a detected system failure, the equality set contains exactly M_i replicas at the end of the voting algorithm. The encoded result of any of the M_i replicas referred to by the equality set $\mathscr{E}$ can be considered as correct.

However, during the execution of the encoded voting algorithm, soft errors may cause false branch decisions, jumps in the instruction sequence, or incorrect values to be stored in temporary variables. In order to guarantee the correctness of the execution and the control flow of the encoded voting procedure itself, we combine the ANBD coding with *signatured instruction streams* [SS87]. Under this approach, each possible path in the program's control flow, i. e., the sequence of operations, is characterized by a unique *static signature* $\mathbf{S}$ that can be computed offline. At runtime, a *dynamic signature* $\mathbf{D}$ is generated dynamically depending on the executed control flow. After the voting algorithm, both signatures are compared and an inequality indicates a failure in the execution.

In order to guarantee the correct execution of our voting procedure, it is implemented such that there is only one valid path to each decision. The result of the encoded voting algorithm is the equality set $\mathscr{E}$ containing the IDs of M_i replicas that delivered the same result. The voter's decision and the corresponding control flow have to be mapped to a constant static signature $\mathbf{S}$ and the selection of this static signature has to be unique for each decision path. We use the sum of the signatures of the replicas referred to by the equality set $\mathscr{E}$ as the static signature $\mathbf{S}$ to represent this path, so

$$\mathbf{S} = \sum_{j \in \mathscr{E}} \mathsf{B}_j. \tag{5.6}$$

To ensure the uniqueness of this static signature, Equation (5.6) must result in a different value for each equality set. Selecting the values B_j plays the main role in the fulfillment of this requirement. Since each replica is either contained in the equality set or not, we choose 2^j as the value of the signature B_j. Hence, the summation of all B_j in the equality set is exclusive and satisfies the uniqueness requirement. For small N and a 32 bit architecture, it is feasible to determine A such that $\mathsf{A} > \mathsf{B}_j + \hat{\mathsf{D}}$. Hoffmann et al. [HUD+14b] showed that 2^j is bad choice for the parameter A, since it results in a shift operation and a low Hamming distance between code words. However, the signatures B_j are added to the code word, so they do not influence encoding with the parameter A.

During the voter's execution, a *dynamic signature* $\mathbf{D}$ is computed depending on the comparisons and the branch decisions that lead to the resulting equality set $\mathscr{E}$. As shown in lines 2 and 8 in Listing 5.1, the dynamic signature $\mathbf{D}$ is calculated during the execution of the voting procedure as

$$\mathbf{D} = B_{e_1} + \sum_{j=2}^{M_i} (\mathsf{X}'_{e_j} - \mathsf{X}'_{e_1}) + \mathsf{B}_{e_1}, \tag{5.7}$$

where $e_1, \ldots, e_{M_i} \in \mathscr{E}$ are the elements of the equality set $\mathscr{E}$. In case of an error-free execution of the voting procedure, it follows from Equation (5.5) and Equation (5.7) that

$$\mathbf{D} = \sum_{j \in \mathscr{E}} \mathsf{B}_j, \tag{5.8}$$

which equals the static signature of the equality set $\mathbf{S}$.

Any failure in the execution of the voting procedure affects the dynamic signature $\mathbf{D}$, which does not equal the static signature for the resulting equality set anymore and thus reveals the failure.

For example, consider a soft error in the comparison of two encoded values in line 7 of Listing 5.1, which causes the algorithm to execute lines 8-10 and k to be added to the equality set $\mathscr{E}$, although $\mathsf{X}_j \neq \mathsf{X}_k$. Since $(\mathsf{X}'_k - \mathsf{X}'_j) + \mathsf{B}_j$ is added to the dynamic signature $\mathbf{D}$ and $\mathsf{X}'_k - \mathsf{X}'_j \neq \mathsf{B}_k - \mathsf{B}_j$, the dynamic signature $\mathbf{D}$ will not match the static signature $\mathbf{S}$ that is computed from the equality set $\mathscr{E}$.

As another example, imagine a soft error in the program counter causes it to skip the computation of the dynamic signature in lines 9 and 10 after it successfully added the correct replica k to the equality set $\mathscr{E}$. Again, the static signature $\mathbf{S}$ will not be equivalent with the dynamic signature $\mathbf{D}$.

Let a soft error affect the temporary variables `j` or `k`. This causes the algorithm to skip the comparison of some replica results. In case this results in the algorithm to find less than M_i replicas with equal results, the encoded voter wrongly assumes a detected system failure and fails silently. This is the wished behavior in case of a voter failure and allows the other voter to jump in. In case the algorithm skipped the comparison of some results but still finds an equality set $\mathscr{E}$ with M_i equal results, the soft error is benign and can be ignored.

Listing 5.2 on the following page shows all checks that are performed before the decoded result is forwarded (line 21) and the other voter is notified (line 23). First, we ensure that

```
1  if (|ℰ| != M_i) {
2      return; //fail silently
3  }
4
5  S = 0;
6  for (e_j ∈ ℰ) {
7      S = S + B_{e_j};
8  }
9  if (S != D) {
10     return;
11 }
12
13 int t = X'_{e_0} - B_{e_0} - D;
14 if (t mod A != 0) {
15     return;
16 }
17
18 if (mode == active) {
19     outputResult(receiver, t / A);
20 }
21 notifyOtherVoter();
```

Listing 5.2: Encoded voting algorithm—Part 2

M_i replicas with equal results have been found and added to the equality set $\mathscr{E}$ in line 1. If this is not the case, the voter fails silently since it can be the result of a soft error in the voting algorithm and the other voter determined a correct result.

Next, the static signature is computed and compared with the dynamic signature. In contrast to computing the static signature $\mathbf{S}$ at runtime, it can as well be computed at compile time for each equality set and stored in a look-up table. Both design choices differ only in execution time and memory consumption.

Finally, the encoded result of the first replica referred to by the equality set is selected. As mentioned above, any other result is equally valid. In line 16, it is checked if the result is still correctly encoded and no soft error has affected the value in the meanwhile, so it tests whether $(X'_j - B_j - D) \mod A = 0$ holds. In case this check fails, the voter does not check the next replica but instead fails silently and relies on the other voter to jump in. If the encoding is valid, the active voter decodes the tallied result and sends it to the receiver. Both voters notify each other if all checks are passed.

5.2.3 Residual SPOFs

The encoded voting algorithm allows to determine the data integrity of the result as well as the correctness of the execution and the control flow. In order to fail silently if an error

occurred, the data integrity and the correct execution have to be evaluated, as shown in Listing 5.2. However, this evaluation is subject to soft errors, too.

A soft error in the comparison of the static and the dynamic signature can deem the correct result of a voting procedure as incorrect, which still results in a fail-silent behavior. In case the voting procedure failed before, an error in the comparison routine can prevent the voting failure to be detected and thus break the fail-silent assumption. However, this requires the occurrence of two soft errors within the same period and therefore has a low probability.

In contrast to Ulbrich et al. [UHK+12], we do not follow the assumption that an actuator receiving the voter's result can handle encoded results. Therefore, the voter has to decode the tallied result before forwarding it to an external device. However, a soft error can affect the decoded result after it has passed this integrity check, see line 20 and 21 in Listing 5.2. This residual SPOF has a low probability, as the decoded and checked result is subject to faults only for a minimal duration, before forwarding it to its final destination.

The underlying OS kernel including the scheduler and dispatcher can also be affected by soft errors. As a result, the voter task might not be executed any more, which results in fail-silent behavior. The voter task could also be scheduled too early or too late. If executed too early, the replica results are not generated yet, which will be detected by the wrong sequence counter D and cause the voter to fail silently. If executed too late, the other voter will detect the missing notification and start the repair procedure.

Furthermore, soft errors can affect the communication infrastructure between the cores. In most cases, such soft errors in the NoC lead to erroneous replica results or missing notifications and cause unnecessary repairs. However, with a low probability a soft error in the NoC can change the voter's output before reaching its destination or cause an omission failure. With a low probability, a soft error can also affect the DMPS such that wrong messages are sent to the destination of the voter output.

The discussed residual SPOFs have a low probability but are evidence that complete reliability cannot be achieved within realistic system boundaries. Therefore, the probability of their occurance is considered in the model of the fault-tolerance mechanism in Section 6.1.

5.3 Repair

In our fault hypothesis we consider only transient faults, since they occur at much higher rates compared to permanent faults. Transient faults are caused by cosmic radiation, radioactive impurities in the chip, or electromagnetic interference. As a result of their physical nature, transient faults are present in the system only for a bounded time interval.

Soft errors are the manifestation of transient faults. Due to the large fraction of chip area occupied by the core-local SRAMs, 90% of soft errors are bit flips in memory [EET+14, EET+15]. While the transient fault that causes a bit flip is only present for a bounded time interval, the soft error is only repaired if the failed memory cell is re-written. If this is the case, the soft error that caused a task replica failure does not occur again when the task replica is executed in the next period. However, in case a memory cell is written only

once and always read afterwards, the transient fault at hardware level can have permanent behavior from the software's point of view.

A simple experiment with our cycle-accurate simulator, which is explained in detail in Chapter 7, with an unprotected task executed on a single core showed that out of 3,952,726 periods with erroneous results, only 42,123 soft errors vanished and the next period delivered a correct result again. Hence, in 99% of the measured failures the transient hardware fault caused a permanent software failure. The reason for this large fraction of soft errors that cause permanent software failures is the great amount of instructions and static data stored in the core-local memories, which are only read but not rewritten.

From this simple experiment we conclude that an active repair procedure is necessary to restore a failed task replica or voter. While the software fault-tolerance mechanism is able to operate with $N_i - M_i$ failed task replicas and one failed voter, any additional task replica failure will cause a detected system failure and any additional voter failure will cause a critical system failure. Therefore, adding the repair procedure to the software fault-tolerance mechanism increases the system's reliability and availability.

5.3.1 Spares

Once a task replica failure is detected, we want to replace it with a functional task replica and operate with the all N_i task replicas as soon as possible. Therefore, each safety-critical task T_i has not only N_i task replicas T_i^j with $j = 1, 2, \ldots, N_i$ but additionally S_i *spares* $\widetilde{T}_i^k$ with $k = 1, 2, \ldots, S_i$. A spare is another replication of the original task that is mapped to a different core than the other spares, replicas, and voters of the respective task. Hence, a task or its instructions and static data, respectively, is replicated and statically mapped to $N_i + S_i$ cores in total, as shown in Figure 5.3. Note that similar to task replicas, spares can be mapped to the same cores with spares and replicas of other tasks.

Spares are scheduled by the core-local OS and its scheduler in the same way as task replicas. However, in contrast to task replicas, spares do not execute the task's original code, but only inform both voters about their existence in every period. This is achieved by sending a special message to the voter.

If necessary, a spare can be started by the active voter. When a spare is started, it becomes a task replica, executes the task's code, and sends the obtained result to both voters.

Informing the voter about the spare's existence requires less time than executing the original task's code and sending the result to the voter. Hence, the WCET of a spare is generally shorter than the WCET of a task replica. In order to ensure that spares meet their deadlines even after they were started by the voter and became a task replica, the WCET of a spare is set to the WCET of the respective task replica during the scheduling analysis.

Since spares are not executed, starting a spare can uncover dormant errors in its code or static data. In this case, the voter detects the failure of the new task replica in the next period and activates another spare.

The advantage of using spares compared to using $N_i + S_i$ task replicas all the time is a lower runtime overhead. The free runtime can be used for scheduling uncritical tasks. The disadvantage is a lower system reliability and availability. However, the reliability

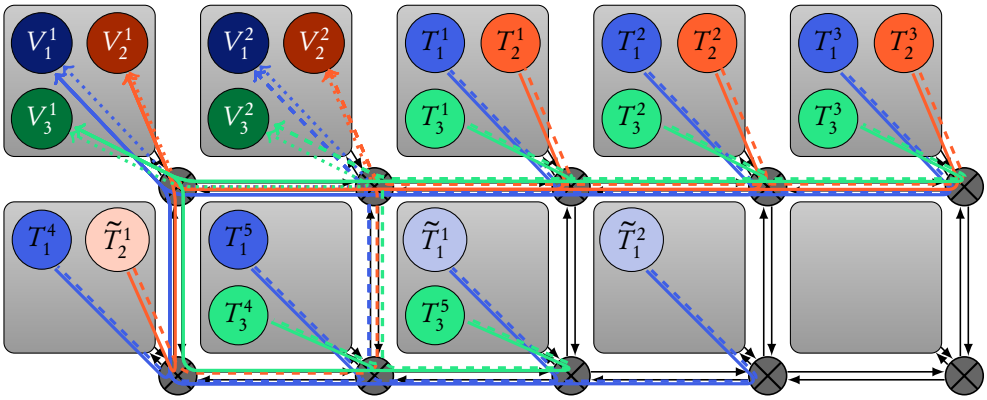


Figure 5.3: Overview of the NMR mechanism with spares on a 2×5 many-core processor. Task T_1 has $N_1 = 5$ replicas and $S_1 = 2$ spares, task T_2 has $N_2 = 3$ replicas and $S_2 = 1$ spare, and task T_3 has $N_3 = 5$ replicas but no spares, so $S_3 = 0$. Solid lines represent messages sent from the task replicas to voter V_i^1, dashed lines represent messages sent from the task replicas to voter V_i^2, and dotted lines are messages that are exchanged between both voters.

and availability is still better than using only N_i task replicas without any spares, since it takes longer to repair a task replica than to activate a spare in general and therefore the system has to operate with less than N_i task replicas for a longer time. Note that a task can also be configured to have no spares, so $S_i = 0$, if the target reliability and availability can be achieved without using spares.

5.3.2 State Duplication

When the voter detects a task replica failure, it actives a spare of the respective task. The voter knows about the spare's existence and location since it was notified by the spare in the same period.

Tasks can have an *internal state*, e. g., a variable for the integration of a sensor value or a mode setting. This internal state is used by all jobs of a replica. It is not to be confused with the temporary variables used by each job during its execution, which are stored in the register file and the stack, but are deleted after the job is finished. Note that the internal state is the reason why repairing all task replicas in case of a detected system failure is not possible. Under majority voting, a detected system failure occurs if less than M_i replicas deliver the same result. In this case, a fault-free task replica with a correct state cannot be identified.

All task replicas have their own internal state in order to ensure fault isolation. Since all task replicas are deterministic and process the same input values, their internal states are identical. However, a spare does not execute the original task's function and does not know about its current internal state. Hence, the internal state of a fault-free task replica has to be copied to the spare's core before the spare can become a new task replica.

Long latency errors are errors that are present in a component but cause the components failure only after a long time, e. g., an error that only affects a rarely used branch or mode.

These errors might be present in the internal state and cause a task replica to fail only when the rarely used branch is executed or the mode is entered. Otherwise, the replica provides a correct result and is deemed fault-free by the voter. Therefore, the long latency errors present in the internal state can be copied to a spare when the spare is activated. As a result, two task replicas will fail at the same time once the long latency error is triggered. However, long latency errors can be detected and eliminated at design time by the *CrashFinder* technique [LLP15]. This technique combines static analysis methods with fault injection tests. Therefore, we assume that a correct result is equivalent to a fault-free internal state in this thesis.

To copy the internal state from a fault-free task replica to a spare is the job of the *state duplication service (SDS)*. The SDS is part of the core-local OS and hence it is present on all cores. When the voter starts the repair procedure for a failed task replica, it activates the SDS on a core where a spare of the respective task is located. The voter informs the SDS about the location of a fault-free task replica, e. g., one of the replicas in the equality set. Once started, the SDS copies the internal state of the fault-free replica to the local core and activates the local spare, so the spare becomes a new task replica.

The SDS is executed after the task's voter but before the next period of the respective task. Therefore, it is ensured that no task replica is running in parallel to the SDS and the copy of the internal state is consistent. Note that there exists an SDS for each task replica and spare mapped to the local core. Considering typical embedded applications and the throughput of the NoC, we assume that the SDS is able to copy the complete internal states and activate all local spares before the SDS's deadline.

Due to the SDS, the system operates with N_i task replicas in the next period as long as enough spares are available. Otherwise, the number of active replicas is decreased until enough spares are available to restore all N_i replicas.

5.3.3 Code Duplication

When the voter detects a task replica failure, it does not only active a spare but also starts the repair of the failed replica. In order to do so, the voter notifies the *code duplication service (CDS)* on the core with the failed task replica. Similar to the SDS, the CDS is part of the OS and therefore it is present on all cores.

Once started, the CDS deletes the failed task from the OS and the scheduler, respectively, such that the failed replica is not executed anymore. Afterwards, it overwrites the replica's code and static data with the task's code and data stored in the reliable external memory. When all code and data is completely restored, the CDS registers the repaired task in the OS again. Since all tasks are initially configured as spares, the repaired task replica has to be started by the voter before it actually executes the restored code.

5.3.4 Core Reset

According to our fault hypothesis, the OS kernel and its components, the scheduler, dispatcher, DMPS, and drivers are subject to faults as well. As a result, a task replica failure can also be caused by a failed underlying OS kernel function. For example, if the scheduler

fails, a task replica might be terminated too early or not be started at all. In this case, the voter will not get any result from this replica.

In order to repair an OS failure, a mechanism that operates independently of the core's OS is required. The ICR mechanism fulfills this requirement. After a reset, any core of the many-core processor fetches its instructions from the reliable external memory. These instructions copy the code and data of the OS and all tasks mapped to the restarted core from the reliable external memory to the core-local memory. This way, all transient faults in code or data in all tasks as well as in the OS and its components are cleared. After the core-local memory content is completely restored, the execution continues with instructions from the local memory. By default, all task replicas are configured as spares after booting.

Repairing a core with the reset mechanism is coarse grained and takes a large amount of cycles compared to rewriting the unknown memory cells that caused the failure. Additionally, it affects all replicas and spares from different tasks that are mapped to the core, although the error might only affect one replica. Therefore, a core is only repaired by a reset when the task replica repair procedure with the CDS is not possible or has failed.

5.3.5 Repair Procedure

After the encoded voting procedure, both voters compare the encoded results of all replicas not referred to by the equality set with the tallied encoded result. All task replicas that did not sent a correct result are repaired. In order to not repair a task or core twice, only the active voter actually starts the SDS and the CDS or triggers a core reset.

The kernel components that implement the repair procedure, i. e., the SDS and the CDS, are also affected by soft errors. In case of a CDS failure, the failed task replica is not repaired and is not becoming a new spare. For this reason, both voters keep track of the number of periods they do not get a spare notification from the respective task and core after starting the repair procedure. Each voter stores this information in its internal state. If a voter does not receive a spare notification after a certain timeout, which is derived from the CDS's WCRT, it assumes the OS has failed and the active voter uses the ICR mechanism to reset the respective core.

A failure of the SDS will be detected by both voters in the next period when the new task replica delivers no or a wrong result. Hence, both voters search a new spare and the active voter starts the CDS for the failed task replica. However, the effect of an SDS failure equals the effect of a dormant error in the spare, which also causes a failure once the spare is activated and becomes a task replica. Since the voters cannot distinguish between an SDS failure and a dormant error in the spare, the CDS always repairs the SDS together with the failed task replica.

When a task replica fails on a core where the CDS is currently repairing another task replica, it is likely that not the task replica but the underlying OS caused the failure of both task replicas. Therefore, a voter first determines whether the CDS is already executed on a core with a failed task replica. If this is the case, the voter immediately uses the reset mechanism to repair the core.

Soft errors can occur during the boot process, e. g., when data is copied over the unreliable NoC and stored in the unreliable core-local memory. For this reason, the voter also

monitors the number of periods it does not get a notification from a spare after starting the core reset routine. If the number of periods exceeds the maximum boot timeout, the core reset routine is started again.

In case a voter does not receive a notification from the other voter, it repairs the failed voter, too. For an efficient implementation, the voters of different tasks are typically mapped to one core and share a large amount of instructions. Therefore, all voters mapped to the same core are likely to fail simultaneously. Consequently, the fault-free voter immediately triggers a core reset in case of a voter failure and we do not use spare voters.

After booting, all voters on the restarted core are inactive. Since each voter is inactive after a reset and does not forward the tallied result or trigger the repair mechanism, it does not need to copy the potentially corrupt internal state of the active voter but can build up its internal state over multiple periods. Note that the internal state is only used to monitor the timeouts of the repair procedures. Therefore, if a voter becomes active before it completely gathered the current timeouts in its internal state, it might restart a running CDS or boot process or delay the restart of a failed repair process. This is acceptable in contrast to copying the potentially corrupt internal state of the active voter, which can result in common cause failures that ultimately causes a critical system failure.

In addition to the SDS, CDS and the boot process, the described repair procedure executed by each voter is subject to soft errors, too. In most cases, an error in the repair procedure of the active voter only leads to unnecessary repairs, while an error in the inactive voter is only reflected in its internal state. However, in some rare cases a failure in the repair procedure of the a voter can lead to a system failure, e. g., if the active voter wrongly triggers the repair of a fault-free task replica and only M_i fault-free task replicas are present in the system, if the active voter issues the SDS to copy the internal state of a faulty task replica to a spare and therefore less than M_i fault-free task replicas are available in the next period, or if the inactive voter wrongly triggers the repair of the active voter but does not become active itself. In other words, the fact that the voters are able to repair task replicas breaches the fault isolation assumption between cores. However, the analysis of a model of the software fault-tolerance mechanism shows that adding an error-prone repair procedure to the mechanism still results in a higher reliability and availability compared to a mechanism without repairs, cf. Section 7.4.3 on page 134.

5.4 Real-Time Integration

In this section, we describe how the proposed software fault-tolerance mechanism can be applied for a mixed-critical task set $\mathcal{T}$ and integrated with the system's real-time constraints.

5.4.1 Task Wrapper

The original function of each safety-critical task $T_i \in \mathcal{T}$ is encapsulated within a *task wrapper*, as shown in Listing 5.3. The task wrapper can be configured only by the SDS to act either as task replica or as spare by the `active` variable, which equals `0` initially. Note

that the value of B_j is different on each core and the `input_data` structure is updated by the DMPS and the drivers according to the LET principle.

The wrapped task is mapped to $N_i + S_i$ mutually different cores. The parameters N_i and S_i are selected depending on the task's safety criticality, as is shown in the next chapter. For each safety critical task $T_i \in \mathcal{T}$, two voters V_i^1 and V_i^2 are mapped to separate cores as well, such that no voter is present on the same core as any task wrapper T_i^j .

When the system is started, all task replicas are initially configured as spares, since this is the desired behavior after a core reset. When both voters are executed for the first time, they check whether they receive more notifications from spares than configured for the respective task. If this is the case, each voter assumes not just its local core was reset but the complete system was started. Therefore, one voter becomes active while the other stays inactive, which is decided e. g., depending on the mapping. The active voter starts the first N_i spares of the respective task such that in the next period, the system is correctly configured.

The set of distributed task wrappers and voters represents a reliable task for the rest of the system. Therefore, all components together have to comply with the original task's real-time requirements.

5.4.2 Communication

Inter-task communication follows the LET concept, so all messages between tasks or the periphery are exchanged logically instantaneously at the task's activation and deadline, which is implemented by the DMPS and the drivers. However, the communication between the task wrappers and both voters cannot follow the LET concept, since this would delay the output of the result by an additional period. Therefore, the task wrappers send their results directly to both voters and do not use the LET. Note that the voter forwards the tallied result with the DMPS or respective driver again.

If all cores obey the limited packet injection rate approach presented in Chapter 4, the WCTL $\hat{L}$ of any packet is guaranteed independently of the packet's source and destination. This allows to determine the WCCT $\hat{C}_i$ of any message m_i. For example, consider a many-core processor with a flit size of 32 bits in the request and the response network and the size of a message m_i sent from any task wrapper $T_i^j = \sigma(m_i)$ to any voter $V_i^j = \delta(m_i)$ to be $|m_i| = 8$ bytes. Thus, the WCCT $\hat{C}_i$ of this message is $\hat{C}_i = 2\hat{L}$, as long as all sources obey the limited injection rate.

5.4.3 Scheduling

Each task wrapper is periodically executed by the OS scheduler of the respective core with the original task's period P_i. In order for the SDS to have sufficient time to activate a spare before its next periodical activation, the relative deadline each task wrapper D_i^j needs to be advanced in case the difference between the period and the original deadline of the task is smaller than the WCET of the SDS, so $D_i^j = \min(D_i, P_i - \hat{E}_{SDS})$. The fixed-point iteration method presented in Section 2.1.1 is used to determine the WCRTs $\hat{R}_i^j$ of each

```
1 void wrapper() {
2    if (active) {
3       X = original_function(*input_data);
4       X'_j = X·A+B_j;
5       sendMsgToVoters(X'_j);
6    } else {
7       sendMsgToVoters(SPARE_CODE);
8    }
9 }
```

Listing 5.3: Task wrapper

task wrapper on all cores. If the WCRT of any task wrapper is larger than its potentially advanced deadline, the real-time constraints are not guaranteed and the task set cannot be scheduled together with the software fault-tolerance mechanism. In this case, the task mapping has to be adjusted. If a modified task mapping does not lead to a feasible schedule, a processor with more cores or a higher performance is the only option left.

Both voter tasks are instantiated on separate cores and scheduled with the same period as the respective task, so $P^j_{V_i} = P_i$. However, each voter is activated with an offset of the WCCT and the maximum WCRT of all task wrappers, so $O_{V_i} = \hat{C}_i + \max_{\forall j} \hat{R}^j_i$. The voter must be able to trigger the SDS and activate a spare before the next period. For this reason, the deadline of each voter task $D^j_{V_i}$ is the original task's deadline advanced by the WCET of the SDS $\hat{E}_{SDS}$ and the WCCT $\hat{C}_j$ of the respective message to start the SDS, so $D^j_{V_i} = D_i - \hat{E}_{SDS} - \hat{C}_j$. The fixed-point iteration method is used again to check whether all advanced deadlines of both voters are met.

5.4.4 OS components

Apart from the task wrappers and voters, the OS components that implement the repair procedure have to comply with the system's real-time constraints as well.

State Duplication Service

The SDS is part of the OS kernel on each core. Its job is to copy the internal state of a fault-free task replica to the local spare and activate it. In order to activate the spare before the next period, the SDS has to be executed after the voter but before the next periodical activation of the respective task.

For this reason, the deadlines of all task wrappers are advanced such that a free time frame between each task wrapper's deadline and its next periodic activation exists. This time frame is reserved for the execution of the SDS. Note that the SDS is only executed when activated by a voter and it is able to activate the spares of all tasks mapped to the local core. Although the SDS has to copy data from another core, its WCET is known due to the limited packet injection rate approach.

Code Duplication Service

Similar to the SDS, the CDS is part of the OS kernel. Its job is to restore a failed task replica. Since the SDS ensures that a new task replica is available in the next period, the runtime of the CDS is less critical. Thus, the CDS consumes the remaining execution time when activated by the voter and is preemptable by all core-local real-time tasks. For example, the CDS uses the time frame reserved for the SDS when the SDS is not activated. Note that the WCRT of the CDS is known nevertheless.

The CDS runs only when activated by the voter, which is checked during the SDS time window. Since the active voter resets the core as soon as two task replicas failed on the same core, the CDS has to repair only one task replica at any time.

Reset Service

When the voter triggers the reset of a core, the restarted OS has to schedule all local tasks in synchronization with the tasks on other cores. For this reason, the boot duration of all cores is prolonged to match an integer multiple of the hyperperiod $\mathring{P}$. The maximum of this boot duration and the WCET of the CDS is denoted as worst-case repair time $\hat{B}$.

The voter does not directly trigger the restart of another core but uses a reset service provided by the OS in order to schedule a core reset exactly at the next hyperperiod $\mathring{P}$. This way, the reset core is in synchronization with the other cores after booting.

5.5 Mixed-Critical Task Set Example

In order to demonstrate the functionality of our software fault-tolerance mechanism and illustrate its real-time capability, we provide an example with a mixed-critical real-time task set in the following.

The parameters of the chosen task set are given in Table 5.1. The task priorities for the fixed-priority scheduling policy are assigned according to the deadline-monotonic priority assignment scheme and criticality of the task.

The number of task replicas N_i and spares S_i is chosen depending on the criticality level, as is explained in detail in the next chapter. The function of each original task T_1, T_2, and T_3 is encapsulated in a task wrapper, which is then replicated $N_i + S_i$ times. Note that task T_4 is uncritical, so it is not protected by the software fault-tolerance mechanism and not included in a task wrapper.

The four mixed-critical tasks have different but harmonic periods. Note that 10 ms and 20 ms are typical periods in automotive applications [KZH15]. For the sake of simplicity, we neglect non-periodic tasks in our case study. However, non-periodic tasks could be included by reserving separate cores for interrupt-driven task replicas and voters.

In the example, the WCETs of the SDS $\hat{E}_{SDS}$ and any voter $\hat{E}^j_{V_i}$ equal 1.5 ms. The WCCT $\hat{C}_i$ of any message sent between task wrappers, voters, and the SDS or CDS is 0.25 ms.

Task	$P_i = D_i$	$\hat{E}_i$	Priority	N_i	S_i
T_1	10 ms	2.5 ms	4	5	2
T_2	10 ms	2.5 ms	3	3	1
T_3	20 ms	4.5 ms	2	5	0
T_4	10 ms	2.5 ms	1	1	0

Table 5.1: The timing and mapping parameters of the mixed-critical real-time task set example.

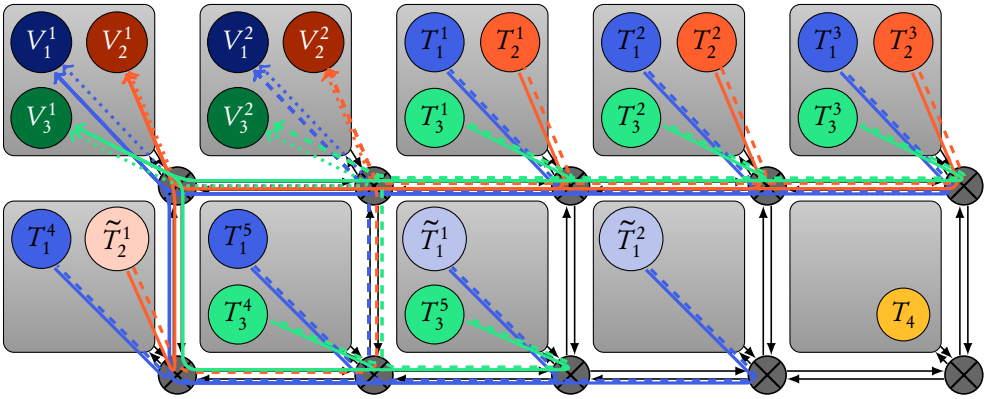


Figure 5.4: A manually derived mapping of the task set example on a 2×5 many-core processor. The solid, dashed, and dotted arrows represent messages that are sent to the first voter, the second voter, and between both voters, respectively.

5.5.1 Mapping

All task replicas, spares and voters of the same task have to be mapped to mutually exclusive cores. As mentioned in Section 2.3.3 on page 36, there exist heuristic and solver-based solutions to determine a feasible task mapping. In this example, the task mapping is derived manually and follows a first fit approach, except that task replicas T_3^4 and T_3^5 are placed one core further to the right. The unprotected T_4 is mapped to its own core in order to improve the load balancing and to prevent its state from being affected by a reset of its core due to another task's failure. The task mapping on a 2×5 many-core processor is depicted in Figure 5.4.

5.5.2 Scheduling

For the given task set and mapping, Figure 5.5 on the next page presents a worst-case schedule of cores $C_{1,1}$, $C_{1,3}$, and $C_{2,2}$. In the worst-case schedule, the execution time of all tasks equals their WCETs. Additionally, the SDS and the CDS are activated. Since the worst-case schedule represents the complete hyperperiod $\mathring{P} = \max_{\forall i} P_i = 20\,\text{ms}$ of the task set, it includes the WCRTs of all tasks on the depicted cores.

All wrapper tasks and voters are scheduled on their local cores with the original task's period, so $P_i^j = P_{V_i}^j = P_i$. Since all tasks in the example task set have implicit deadlines,

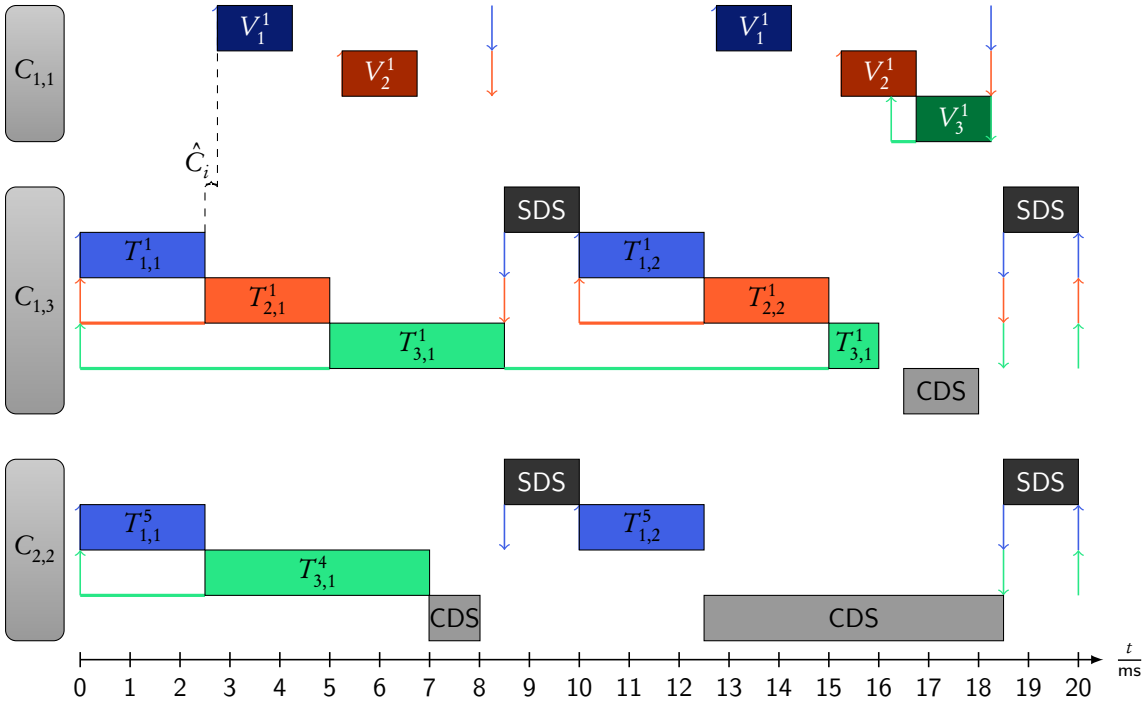


Figure 5.5: Schematic representation of the worst-case schedule on cores $C_{1,1}$, $C_{1,3}$, and $C_{2,2}$. Task activations are indicated by arrows pointing upwards, absolution deadlines by arrows pointing downwards. Colored horizontal lines designate tasks that are activated but currently preempted by higher priority tasks.

so $P_i = D_i$, the relative deadline of each task wrapper is advanced by the WCET of the SDS. As shown in Figure 5.5, the relative deadline of all task wrappers of task T_1 is $D_1^j = D_1 - \hat{E}_{SDS} = (10 - 1.5)\,\text{ms} = 8.5\,\text{ms}$. Similarly, the relative deadline of each task wrapper of task T_2 also equals $D_2^j = 8.5\,\text{ms}$ and the relative deadline of each task wrapper of task T_3 is $D_3^j = 18.5\,\text{ms}$.

The WCRTs $\hat{R}_i^j$ of all task wrappers on all cores are calculated using the fixed-point iteration method. As shown in Figure 5.5, all advanced deadlines are met although task wrapper job $T_{3,1}^1$ on core $C_{1,3}$ is preempted twice by task wrappers T_1^1 and T_2^1 and once by the SDS. Its WCRT $\hat{R}_3^1 = \hat{E}_3 + 2(\hat{E}_1 + \hat{E}_2) + \hat{E}_{SDS} = 4.5 + 2(2.5 + 2.5) + 1.5 = 16\,\text{ms}$. Note that task wrappers T_1^1 and T_2^1 cannot be preempted by the SDS, since SDS is only executed after their advanced deadline, which is chosen such that the SDS finishes before their next periodic activation.

For each task, both voters are offset by the maximum WCRT of any task wrapper of the respective task. As shown in Figure 5.5, the WCRT of the task wrapper for task T_3 on core $C_{2,2}$ is $\hat{R}_3^4 = \hat{E}_3 + \hat{E}_1 = 7\,\text{ms}$. This is shorter than the WCRT $\hat{R}_3^1 = 16\,\text{ms}$ of the task wrapper on core $C_{1,3}$, since only two tasks are scheduled on core $C_{2,2}$. Hence, the offset of both respective voters $O_{V_3}^1 = O_{V_3}^2$ is $\hat{C}_i + \max_{\forall j} \hat{R}_3^j = 16.25\,\text{ms}$. The offsets of the voters of task T_1 and T_2 are derived in the same way.

In order to be able to trigger the SDS and activate a spare before the next period, the deadline of each voter $D^j_{V_i}$ is advanced by the WCET of the SDS $\hat{E}_{SDS}$ and the WCCT $\hat{C}_i$ of the respective message to start the SDS. For the given example task set, the relative deadline of the voter of task T_3 mapped to core $C_{1,1}$ is $D^1_{V_3} = D_3 - \hat{E}_{SDS} - \hat{C}_i = (20 - 1.5 - 0.25)\,\text{ms} = 18.75\,\text{ms}$. The fixed-point iteration method is used again to determine the WCRTs $\hat{R}^j_{V_i}$ of all voters and check whether all voters meet their advanced deadlines. As shown in Figure 5.5, the voter of task T_3 is preempted by the voter task T_2 but still meets its deadline in the worst-case. Thus, the complete task set is feasible.

5.5.3 Fault-Tolerance

Once the mixed-critical real-time task set is integrated with the software fault-tolerance mechanism, it is protected from $N_i - M_i$ task replica failures and single voter failures. Figure 5.6a on the facing page depicts the schedule of the running system under the presence of faults. As mentioned before, the boot duration is prolonged to match one hyperperiod $\mathring{P} = 20\,\text{ms}$. Hence, all task wrappers and voters are scheduled first after 20 ms.

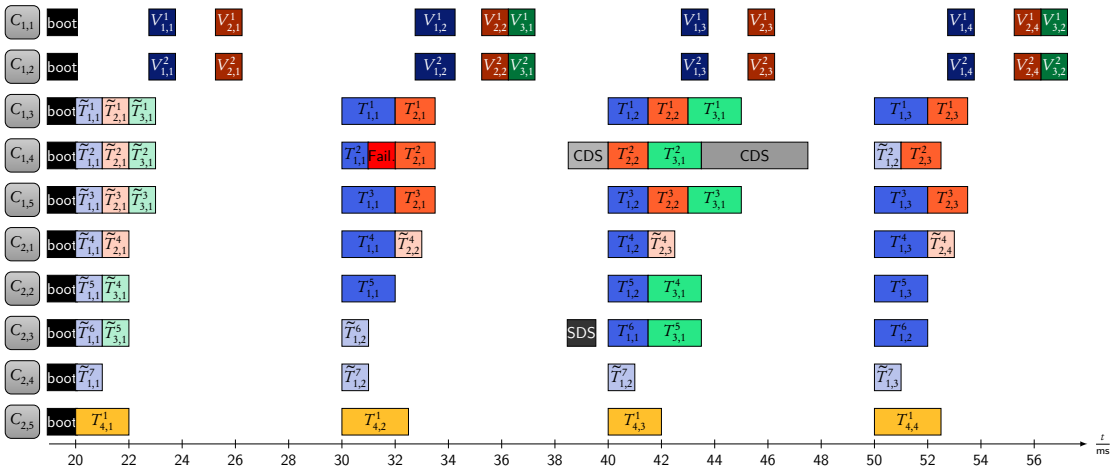
All task wrappers are initially configured as spares and notify their respective voters about their existence. Since there are more spares present in the system than configured, the voters detect the system start. In this case, one voter becomes active, e. g., depending on the core ID, and starts the first N_i replicas of each replicated task. This mechanism will initialize the software fault-tolerance mechanism as long as less than the configured number of spares fail during the initialization boot process of the system.

In the next period, after 31 ms, a fault causes the task wrapper of task T_1 on core $C_{1,4}$ to fail. This failure is detected by both voters. The active voter starts the CDS on core $C_{1,4}$. The CDS deletes the failed task replica from the scheduler table, copies the code and static data from the reliable external memory to the local core, and registers a new task wrapper afterwards. Therefore, a new spare $\widetilde{T}^2_1$ is available after 50 ms again.

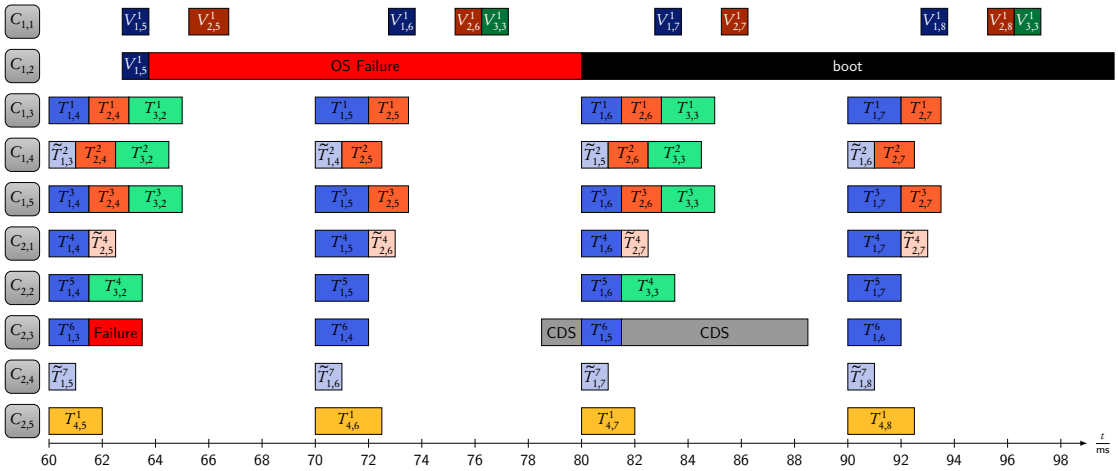
The active voter additionally triggers the SDS on core $C_{2,3}$. The SDS copies the internal state from the fault-free replica it was informed about by the voter. Afterwards, the SDS activates the local spare $\widetilde{T}^6_1$ such that in the next period, after 40 ms, $N_1 = 5$ task replicas are active.

The schedule is continued in Figure 5.6b. A soft error causes an OS failure in core $C_{1,2}$ after 63.25 ms. As a result, the voter $V^1_{2,5}$ on core $C_{1,1}$ is not notified by $V^2_{2,5}$. Therefore, it becomes active and forwards its tallied result in case it has not been active already. Additionally, the voter uses the reset service of the OS to reset the core $C_{1,2}$ at the next hyperperiod, i. e., at 80 ms. The boot duration is extended to match the hyperperiod $\mathring{P} = 20\,\text{ms}$. Hence, all voters on core $C_{1,2}$ are executed again after 100 ms, but are inactive. Note that this is not shown in Figure 5.6b.

After 62 ms, the task replica job $T^5_{3,2}$ fails on core $C_{2,3}$. Since this task has the period $P_3 = 20\,\text{ms}$, the CDS is only started at 78.5 ms, after the voter $T^1_{3,3}$ has detected the failure. The CDS restores the task wrapper of task T_3. Hence, the voters of T_3 are notified by a new spare on core $C_{2,3}$ afterwards. Note that this is not shown in Figure 5.6b anymore. Since the system operates with $4 < N_3$ task replicas when the new spare becomes available,



(a) After 31 ms, task replica job $T^2_{1,1}$ fails. The SDS activates spare $\widetilde{T}^6_1$ and the CDS repairs the failed replica such that the second spare $\widetilde{T}^2_1$ is available after 50 ms again.



(b) After 62 ms, the task replica job $T^5_{3,2}$ fails on core $C_{2,3}$. Since task T_3 is configured without spares, the failed replica first becomes a spare after 100 ms before all five replicas are active again after 120 ms (not shown). After 63.75 ms, the OS on core $C_{1,2}$ fails. As a result, the other voters on core $C_{1,1}$ reset core $C_{1,2}$ at the next hyperperiod at 80 ms. All voters on core $C_{1,2}$ are executed again in passive mode after 100 ms (not shown).

Figure 5.6: Schematic representation of an example schedule of the mixed-critical real-time task set protected by the software fault-tolerance mechanism under the presence of faults. The communication between the components is omitted for the sake of clarity.

the active voter immediately starts the SDS and the failed task replica is finally restored at 120 ms after two periods P_3.

Consider the case that instead of the task replica job $T^5_{3,2}$ the OS of $C_{2,3}$ fails after 62 ms. Then the task replica job $T^6_{1,4}$ does not deliver a result neither. This is detected by the voter job $T^1_{1,6}$, which starts the CDS on core $C_{2,3}$. Hence, the voter job $T^1_{3,3}$ detects that

the CDS is running already and issues the reset of the core $C_{2,3}$ at the beginning of the next hyperperiod.

5.6 Summary

In this chapter, we have presented our software-based hardware fault-tolerance mechanism. We have reasoned that a mechanism based on the NMR scheme is most suitable to increase the reliability and availability of software executed on a many-core processor and can be integrated with the system's real-time constraints. Furthermore, the level of replication can be adapted for each task such that the fault-tolerance mechanism supports mixed-critical task sets. Additionally, we have selected a majority voter due to its low complexity and since it does not produce ties.

The voter represents a SPOF in the system, thus it has to be reliable. We have proposed to increase the reliability by using two fail-silent voters that check each other. The fail-silent behavior is achieved by the encoded voting procedure that operates only on ANBD-encoded replica results. The procedure returns the equality set that refers to M task replicas that delivered a correct result as well as a dynamic signature $\mathbf{D}$. Comparing the dynamic signature with the static signature $\mathbf{S}$, which is unique for each equality set, uncovers failures in the control flow of the encoded voting procedure. The result of any replica that is referred to by the equality set represents the tallied result. The integrity of the tallied result is ensured since it stays ANBD-encoded all the time. However, the comparison of the dynamic and the static signature as well as the final decoding of the tallied result are subject to faults, too. We have discussed the residual SPOFs and include them in our model of the fault-tolerance algorithm, which is presented in the next chapter.

To further increase the system's fault tolerance, we have extended the NMR scheme by a repair procedure. The repair procedure is activated by a voter when it detects a failed task replica or a failure of the other voter. In order to operate with N task replicas even after a replica failure, we have added spares to the systems. Spares are task replicas that are not executed but can be activated by the voter. The code duplication service (CDS) and state duplication service (SDS) are OS services that repair the failed task replica and copy the internal state of a fault-free replica to activate the spare, respectively. In case these OS services fail as well, the voter uses the inter-core reset (ICR) mechanism to restart the respective core.

To integrate the software fault-tolerance mechanism with a mixed-critical task set, the original task function is encapsulated in a task wrapper that is mapped to mutual exclusive cores together with both voters. We have shown how to select the voter offset and advance the deadlines of task wrappers such that the system complies with its real-time constraints. The fixed-point iteration method for the partitioned fixed-priority preemptive scheduling policy is used to provide the respective real-time guarantees.

We have illustrated the functionality and the real-time capability of the proposed software fault-tolerance mechanism in a comprehensive example. The example shows that the mechanism tolerates component failures to protect mixed-critical task sets and meets all deadlines. An experimental evaluation and implementation of the presented software fault-tolerance mechanism is given in Section 7.3 on page 120.

6 Software Fault-Tolerance Framework

The implementation of a mixed-critical application on an unreliable many-core processor is challenging and time-consuming, especially when real-time constraints have to be guaranteed. This task typically requires the cooperation of hardware experts determining the SER of the many-core processor, safety analysts specifying the reliability and availability targets depending on the obligatory safety standards, real-time specialists defining the task set and its parameters, and software developers implementing the functionality of the system and configuring the OS accordingly.

We propose a software fault-tolerance framework that provides all involved roles with an interface to specify a mixed-critical task set, its safety and real-time requirements, and the properties of the many-core processor. Based on this specification, the framework employs a fault-tolerance analysis to derive the resulting reliability and availability of each task on the target hardware. These results are used to adapt the software fault-tolerance mechanism for each task such that the task's fault-tolerance requirements are met with minimum overhead in terms of system utilization, i. e., with the minimum required number of replicas and spares. Thus, the proposed fault-tolerance framework alleviates the integration of our software fault-tolerance mechanism with a mixed-critical real-time task set.

In this chapter, we first present an overview of approaches to model a system in order to assess its fault-tolerance properties. Next, we detail discrete time Markov chains (DTMCs) as the underlying mathematical model of our fault-tolerance analysis. We show how to model and analyze the proposed software fault-tolerance mechanism as a DTMC using the PRISM probabilistic model checker. Afterwards, we present how our fault-tolerance framework integrates the fault-tolerance analysis and its results with the fault-tolerance mechanism. Finally, we demonstrate the functionality of our fault-tolerance framework with a mixed-critical real-time task set example derived from a real-world application.

The work presented in this chapter is published in [AML+15, MAL+15, MAL+16].

6.1 Fault-Tolerance Models

In order to adapt the software fault-tolerance mechanism such that it achieves the target reliability $\check{R}$ and target availability $\check{A}$ after the mission time t_m with minimum overhead in terms of system utilization, the actual reliability R and availability A for a given parametrization of the mechanism have to be known.

One way to determine these values is to implement the mechanism and measure the actual reliability and availability of the system. However, realistic SERs are low and

realistic mission times are long. Hence, it takes a huge amount of long measurements until statistically significant values can be obtained.

Another way to determine the actual reliability and availability is to analyze a model of the software fault-tolerance mechanism. Once the mechanism is modeled and all parameters are known, the actual reliability R and availability A of the system can be analyzed faster compared to measurements. For this reason, we present a model of the software fault-tolerance mechanism in the following.

One approach to model a system in order to assess its reliability is to partition it into independent components and represent it as a reliability block diagram [Bir14, p. 28]. If the reliabilities of the components in the block diagram are known, the system's reliability can be derived from combinatorial calculations of series and parallel connections between these components. For example, Equation (5.1) on page 66 calculates reliability of an NMR system from the reliabilities of the components, namely the replicas and the voter.

Note that the reliability block diagram derives the failure rate of the system but does not account for the mode and effect (consequence) of a failure [Bir14, p. 72]. In reliability engineering, these cause-to-effect chains are typically derived by a fault tree analysis (FTA) or by failure mode, effects and criticality analysis (FMECA). In an FMECA all possible failure modes, their effects, and their criticality are systematically assessed in an inductive (bottom-up) way [Bir14, p. 72]. In an FTA the failure-causes-to-effects are investigated in deductive (top-down) way as a logic combination of faults [Bir14, p. 76]. In this thesis, we focus on the system's reliability and availability. While we differentiate between two modes, namely detected and undetected system failures, we do not consider the application-specific effect of these system failures.

If a system contains a repair procedure, imperfect voters, or different failure modes, it is *complex* in the scope of reliability analysis [Bir14]. For complex systems, the reliability cannot be derived by a reliability block diagram and combinatorial arguments anymore [Bir14]. For example, in an NMR system with a repair procedure, the number of replicas changes over time. This dynamic value of N_i is not captured by Equation (5.1) on page 66, which therefore does not model the system correctly. Since our software fault-tolerance mechanism contains a repair procedure, too, it cannot be modeled as a reliability block diagram.

6.2 Markov Chain

The common ways to model reliability of complex systems are *stochastic processes*. In the following, we give a basic overview of mathematical foundation of stochastic processes based on [Ros14] and [Bir14]. Note that in this thesis, we do not contribute to the field of stochastic processes but leverage existing approaches to model and analyze our system.

A stochastic process is a collection of random variables $\{X_t\}$. The *state space* $\mathcal{X}$ is the collection of all possible values the random variables can take on, i. e., the sample space. A sample *path* or *realization* of a stochastic process is the collection of values from the state space that are assumed by the random variables in one realization of the random process, e. g., $X_1 = x_1, X_2 = x_2, \ldots$ with $x_i \in \mathcal{X}$. The index t denotes the time and is either continuous or discrete.

6.2.1 Discrete Time

If the index t is discrete and the state space $\mathcal{X}$ is a discrete, finite, and countable set, the stochastic processes is a *discrete time Markov chain (DTMC)* if it satisfies the Markov property. The Markov property is

$$\Pr(X_t = x_t | X_0 = x_0, X_1 = x_1, \ldots, X_{t-1} = x_{t-1}) = \Pr(X_t = x_t | X_{t-1} = x_{t-1}), \tag{6.1}$$

where $x_i \in \mathcal{X}$ are the states of the process. In other words, in a DTMC the transition to the next state only depends on the previous state, which means it is *memory-less*. This memory-less property implies that in order to predict the future states reached by the DTMC, it is sufficient to know the current state. Note that the state changes only occur at discrete time steps t.

The one-step *transition probability* π is the probability that the process, when in state x_i at time t, will next transition to state x_j at time $t+1$, so

$$\pi_{x_i,x_j}^{t,t+1} = \Pr(X_{t+1} = x_j | X_t = x_i). \tag{6.2}$$

Since the transition probabilities are probabilities, $0 \leq \pi_{x_i,x_j}^{t,t+1} \leq 1$ holds. Furthermore, since the chain must transition somewhere, $\sum_{x_j \in \mathcal{X}} \pi_{x_i,x_j}^{t,t+1} = 1$ holds.

If all one-step transition probabilities do not depend on time, so $\pi_{x_i,x_j}^{t,t+1} = \pi_{x_i,x_j}$ for all t, then the DTMC is called *time homogeneous*. In the following, we consider only time homogeneous DTMCs.

A state $x_i \in \mathcal{X}$ is called *absorbing state* if $\pi_{x_i,x_i} = 1$. In other words, once a realization of the DTMC enters an absorbing state, it never exits this state again.

A (time homogeneous) DTMC is fully specified by the one-step transition probability matrix $\mathbf{P}$ that contains all one-step transitions and the initial state distribution $\Pr(X_0 = x_i) = k$ for all $x_i \in \mathcal{X}$ and $0 \leq k \leq 1$.

The n-step transition probability is

$$\pi_{x_i,x_j}^{(n)} = \Pr(X_{t+n} = x_j | X_t = x_i). \tag{6.3}$$

for $n \geq 0$. Note that the superscript (n) is an index and not an exponent. The n-step transition probabilities satisfy the Chapman-Kolmogorov equations

$$\pi_{x_i,x_j}^{(n+m)} = \sum_{x_k \in \mathcal{X}} \pi_{x_i,x_k}^{(n)} \pi_{x_k,x_j}^{(m)} \tag{6.4}$$

for all $n, m \geq 0$. This can be also written as $\mathbf{P}^{(n+m)} = \mathbf{P}^{(n)}\mathbf{P}^{(m)}$. By induction follows that the transition probabilities after the n^{th} step can be calculated as $\mathbf{P}$ to the power of n, so $\mathbf{P}^{(n)} = \mathbf{P}^n$. As a result, it is possible to compute the probability of reaching any state x_j after a particular discrete time t, given the one-step transition probability matrix and the initial state distribution of the DTMC.

6.2.2 Continuous Time

According to our fault hypothesis, failures occur at exponentially distributed, random points on a continuous time axis. Hence, they cannot be modeled directly with a DTMC.

If the time index t of a stochastic process is continuous and the state space $\mathcal{X}$ is a discrete, finite, and countable set, the stochastic processes is a *continuous time Markov chain (CTMC)* if it satisfies the Markov property. Note that CTMCs are also known as Markov processes in literature. CTMCs are conceptually similar to DTMCs. However, the transition probabilities π_{x_i,x_j} are replaced by *transition rates* ρ_{x_i,x_j}. In order to fulfill the Markov property, the time spent in each state has to be exponentially distributed [Ros14, p. 373]. Hence, the time spent in a state x_i before transitioning to state x_j is exponentially distributed with mean $\frac{1}{\rho_{x_i,x_j}}$.

This property fits well to our fault hypothesis and allows to model a replica failure as a transition between two states with the constant failure rate Λ. However, the periodic executions that cause a fault to become active and lead to a component failure as well as the repair times in our system is not exponentially distributed. Instead, we assume a constant period P_i and worst-case repair time $\hat{B}$. Since this property requires a deterministic distribution to model the time spent in a subset of states, our software fault-tolerance mechanism cannot be accurately modeled by a CTMC.

6.2.3 DTMC for Fault-Tolerance Analysis

In order to incorporate exponentially distributed SERs with deterministic repair rates, we leverage the periodic nature of the considered real-time system. The periodic nature allows to discretize the time by considering a transition to the next state as the next period. For example, we calculate the probability of a failure during one period but are not interested in the exact time of the fault that caused the failure. At each discrete time step, i. e., at each period, the state is changed only if a failure occurred. For this reason, we are able to use a DTMC to model our software fault-tolerance mechanism and analyze the resulting reliability and availability.

Note that there exist extensions of Petri nets such as discrete deterministic and stochastic Petri nets (DDSPNs) [ZCH97] and stochastic activity networks (SANs) [SM00], which support exponential and deterministic distributions for transitions between places and states, respectively. However, compared to our approach these approaches are more complex.

Reliability. In the following, we introduce how a DTMC model is analyzed and how its reliability can be derived based on the example depicted in Figure 6.1 on the facing page. The example system contains two task replicas. Hence, the number of alive replicas can be modeled as the states of a DTMC. In state ②, both replicas are alive, in state ① one replica failed while the other is alive, and in state ⓪ both replicas failed.

Let f denote the probability of a failure of a task replica within the duration of one period P_i, so $f = 1 - e^{-\lambda P_i}$. Note that the complete duration of the period is captured in the task failure probability f although the task execution will be shorter in most cases.

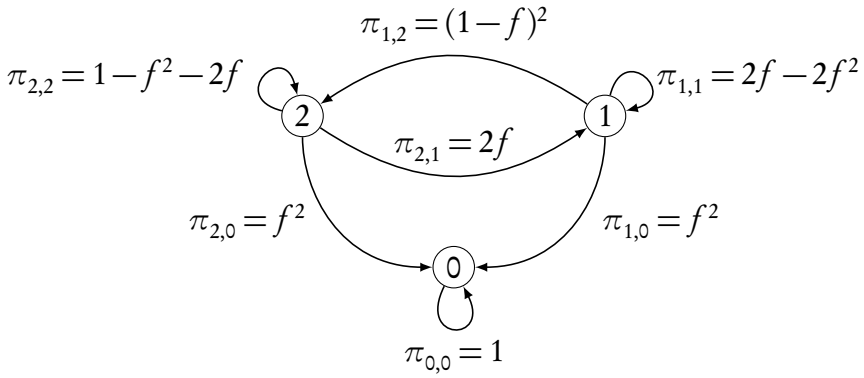


Figure 6.1: A DTMC model of a system with two replicas and a repair procedure that repairs one replica as long as the other replica is fault-free. The repair procedure fails with the same probability as a task replica f. The state number represents the number of fault-free replicas. State ⓪ is an absorbing state since no repair is possible anymore.

The reason is that failures in the OS kernel as well as bit flips in the task's memory area also result in task failures but occur during the complete period.

Assume that a failed replica is repaired within one period, as long as the other replica is alive. The repair procedure fails with the same probability f. If both replicas fail, the system cannot be repaired anymore.

The probability that either of the two fault-free replicas in state ② fails is $f+f$, thus the transition from state ② to state ① has the probability $\pi_{2,1} = 2f$. The probability that both alive replicas in state ② fail within the same period is $f \cdot f$, so the transition from state ② to state ⓪ has the probability $\pi_{2,0} = f^2$. The probability of staying in state ② is $\pi_{2,2} = 1-f^2-2f$.

When one replica failed, the realization of the DTMC is in state ①. The failed replica is repaired within one period, but the repair procedure is subject to faults as well. The probability that the remaining fault-free replica and the repair of the failed replica both fail is $f \cdot f$, hence transition from state ① to state ⓪ has the probability $\pi_{1,0} = f^2$. The other way round, the probability that the remaining fault-free replica as well as the repair procedure of the other replica are both fault-free is $(1-f)^2$, hence transition from state ① to state ② has the probability $\pi_{1,2} = (1-f)^2$. In case the repair is successful and the remaining replica fails or the repair fails and the remaining replica stays fault-free the state is not changed. Thus, the probability of staying in state ① is $\pi_{1,1} = 1-(1-f)^2-f^2 = 2f-2f^2$.

In case both replicas have failed, the system is not repaired anymore. Thus, state ⓪ is an absorbing state and the probability of staying in state ⓪ is $\pi_{0,0} = 1$, therefore $\pi_{0,1} = \pi_{0,2} = 0$.

All state transition probabilities can be represented in the one-step transition matrix $\mathbf{P}$, which is

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ f^2 & 2f-2f^2 & (1-f)^2 \\ f^2 & 2f & 1-f^2-2f \end{pmatrix}.$$

Both task replicas are initially fault-free, thus the initial state distribution is $\Pr(X_0 = 0) = 0$, $\Pr(X_0 = 1) = 0$, and $\Pr(X_0 = 2) = 1$ or $(0,0,1)$ if represented as a vector.

With the transition matrix $\mathbf{P}$ and the initial state distribution, it is now possible to calculate the probability of each state after an arbitrary number of discrete time steps, i. e., periods. In reliability theories, the reliability $R(t)$ is the probability of the DTMC not having reached the failed state until time t. So, the reliability of the system is the sum of all possible realizations of the DTMC until the given number of periods that do not contain state ⓪.

Since the failed state is an absorbing state in our example, no realization that contains state ⓪ will ever leave it. Thus, we can calculate the reliability of the system as the probability of not being in state ⓪ at the specified time. The reliability of the modeled system after $t = nP_i$ is $R(nP_i) = 1 - \Pr(X_n = 0) = \sum_{x_i \in \mathcal{X}} \pi^{(n)}_{x_i,0} \Pr(X_0 = x_i)$. This can also be represented in matrix notation, where $(\Pr(X_n = 0), \Pr(X_n = 1), \Pr(X_n = 2)) = (0,0,1)\mathbf{P}^n$.

For example, consider a period $P_i = 10\,\text{ms}$ and an (unrealistic) failure rate $\lambda = 10.536\,\frac{1}{\text{s}}$. Thus, the probability of a failure per period $f = 1 - e^{-\lambda P_i} = 0.1$, so the one-step transition matrix is

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0.01 & 0.18 & 0.81 \\ 0.01 & 0.2 & 0.79 \end{pmatrix}.$$

Furthermore, assume we are interested in reliability of the modeled system after $t = 100\,\text{ms} = 10P_i$, so $n = 10$. With the initial state distribution vector $(0,0,1)$, the state probability vector after $n = 10$ discrete time steps is

$$(0,0,1)\mathbf{P}^{10} = (0.0956, 0.179, 0.725)$$

and the system is in state ⓪ with a probability of 9.56%. Thus, the reliability of the system after $t = 100\,\text{ms}$ is $R(10P_i) = 1 - 0.0956 = 90.4\%$.

In general it is not sufficient to calculate the probability of not reaching the failed state at the specified time in order to determine reliability of the system. If the failed state is not an absorbing state, the probability will include realizations that contain the failed state but end in a fault-free state. However, the definition of the reliability is specified as the probability that no operational interruptions will occur during a stated time interval.

Availability. In contrast to the reliability, the availability considers system failures and the continuance of the system after a repair. So far, there is no difference between the availability and the reliability in the example, since the system is either alive or it failed and cannot be repaired anymore.

In order to demonstrate the difference between reliability and availability, assume that the example system depicted in Figure 6.1 on page 91 fails as soon as one replica fails. In other words, the system is only working if both replicas are fault-free, i. e., if the system is in state ②. In state ①, the system is now considered to have failed, but the repair procedure allows the system to become fault-free again.

The point availability $A_P(t)$ denotes the probability that the system provides the correct service at a stated instant of time t, as defined in Section 2.1.2 on page 15. The point

availability can be calculated as the reliability in the previous example, since it explicitly includes realizations of the DTMC that contain the failed states ⓪ and ①. In other words, the point availability is $A_P(nP_i) = \Pr(X_n = 2) = \sum_{x_i \in \mathcal{X}} = \pi^{(n)}_{x_i,2} \Pr(X_2 = x_i)$.

The average availability $A(t_m)$ is defined as the expected proportion of time in which the system provides the correct service in $(t_0, t_m]$. Let the mission time $t_m = nP_i$ and $t_0 = 0$. Then, the average availability is $A(t_m) = \frac{1}{t_m} \sum_{t=t_0}^{t_m} A_P(t) = \frac{1}{n} \sum_{m=0}^{n} \Pr(X_m = 2)$.

The long-term availability A_L is defined as the stationary or steady-state of the point availability or average availability. It is calculated as $A_L = \lim_{t \to \infty} A_P(t)$. Since the example system still contains the absorbing state ⓪, which is never left once entered, the probability of being in state ⓪ is $\Pr(X_n = 0) = 1$ for $n \to \infty$ and probability of being in any other state is $\Pr(X_n = 1) = \Pr(X_n = 2) = 0$ for $n \to \infty$. Thus, the long-term availability A_L for a system that is able to fail completely is always $A_L = 0$. For this reason, we focus on the average availability in this thesis and abbreviate it as availability in the following.

6.3 PRISM Model

For a simple system such as the previous example, the DTMC can be derived manually. However, for more complex systems such as our proposed software-based hardware fault-tolerance mechanism with its fail-silent voters and spares, a manual derivation of a DTMC model is time-consuming and error-prone. Therefore, we generate and analyze the DTMC model of our fault-tolerance mechanism using the PRISM probabilistic model checker version 4.3 [KNP11, PRI15].

6.3.1 The PRISM Probabilistic Model Checker

The PRISM probabilistic model checker is a free and open-source software tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior. It supports DTMCs, CTMCs, Markov decision processes, and probabilistic timed automata as mathematical models.

PRISM requires the system to be described in the state-based PRISM language. The PRISM language allows to define several modules. Each module can have local variables. The state of the system is defined by the local variables of all modules. The local variables within each module are changed by a set of commands. Each command consists of a guard and one or more updates. The guard is a predicate over all variables that has to be satisfied in order to enable the updates of the command. Each update describes the transition the module can make by assigning new values to the local variables. Each update is also assigned a probability. The probability of all updates in one command has to sum up to one. Each command can also be labeled by an action. Commands with the same action in different modules are *synchronized*, so they make their transitions simultaneously. When multiple commands are enabled in different modules, PRISM makes a probabilistic choice and combines the probabilities of the commands' updates. The PRISM language also supports the use of global constants, e. g., for defining failure probabilities. An example of a system described in the PRISM language is given in Listing 6.1 on page 95.

6.3.2 Model of the Software Fault-Tolerance Mechanism

The PRISM model of our software fault-tolerance mechanism uses the following constants:

- `Fr`: the failure rate Λ_T of the replicas
- `Fs`: the failure rate $\Lambda_{\tilde{T}}$ of the spares
- `Fv`: the failure rate Λ_V of the voter
- `Fc`: the failure rate Λ_C of the voter caused by residual SPOFs
- `T`: the mission time t_m
- `P`: the task period P_i
- `Ni`: the number of task replicas N_i
- `Si`: the number of spares S_i

Based on these constants, the PRISM model first calculates the number of periods within the mission time `Tp = T/P` and the minimum number of active replicas required by the NMR scheme `Mi=floor(Ni/2)+1`. Note that all constants have the same time base, e. g., hours, milliseconds, or cycles.

For each failure rate, the PRISM model calculates the probability of a failure after one period and the probability of a correct period. Note that the complete duration of the period is used to calculate the probability of a correct period and the probability of a failure after one period, since bit flips in the memory as well as failures in the OS kernel occur during the complete period. The probability of a task replica failure after one period is $\texttt{dr} = 1 - e^{-\texttt{Fr}\cdot\texttt{P}}$ and the probability of a failure-free period of the replica is $\texttt{cr} = 1 - \texttt{dr} = e^{-\texttt{Fr}\cdot\texttt{P}}$. Similarly, the probability of a spare failure after one period is $\texttt{ds} = 1 - e^{-\texttt{Fs}\cdot\texttt{P}}$ and the probability of a correct period of the spare is $\texttt{cs} = 1 - \texttt{ds}$.

For the voter, the probability of a silent failure after one period is $\texttt{dv} = 1 - e^{-\texttt{Fv}\cdot\texttt{P}}$. As explained in Section 5.2.3 on page 72, there exist residual SPOFs in each voter that can cause a voter to fail in a non-silent way. In the following, we denote such non-silent voter failures as *critical*. The probability of a critical failure after one period is $\texttt{dc} = 1 - e^{-\texttt{Fc}\cdot\texttt{P}}$. As a result, the probability of a correct period of the voter is $\texttt{cv} = 1 - (\texttt{dv} + \texttt{dc})$.

The worst-case repair time $\hat{B}$ is specified as the maximum of the boot duration and the WCET of the CDS in Section 5.4.4 on page 81. This time is a multiple of the task's period P_i, since a repaired task is not activated before its next period. The ratio of errors that affect only one task replica to errors that affect the OS kernel and therefore all replicas on the same core is unknown. Therefore, we conservatively assume the worst-case repair time $\hat{B}$ is always required in the model of the software fault-tolerance mechanism. In the following, let B denote the worst-case repair time in periods, so $B = \frac{\hat{B}}{P_i}$.

The PRISM model of our software fault-tolerance mechanism contains four modules:

```
module voter1
v1 : [0..Bv] init Bv;
[V] v1=Bv -> cv:(v1'=v1) + dv:(v1'=1) + dc:(v1'=0);
[V] v1<Bv & v1>=2 -> cv:(v1'=v1+1) + dv:(v1'=1) + dc:(v1'=0);
[V] v1<2 -> true;

[R] v1>=2 -> true;
[R] v1=1 & v2=Bv -> (v1'=2);
[R] v1=1 & v2<Bv -> true;
[R] v1=0 -> true;
endmodule

module voter2 = voter1 [v1=v2, v2=v1] endmodule
```

Listing 6.1: The two voter modules of the PRISM model. The second voter is a copy of the first voter.

1. The first module contains the variables for the number of alive replicas `r`, failed replicas `fr`, alive spares `s`, and failed spares `fs`. Additionally, the first module contains the variables `b`i for the number of tasks that have been under repair for i periods, with $i = 1,\ldots,B$. All variables have to be contained within one module, since the repair procedure requires to modify all of them and updates can only assign new values to the local variables in PRISM.

2. The second module specifies the first voter. For simplicity, the module introduces the constant `Bv` $= B + 2$. It contains a variable `v1` that denotes the state the first voter is in: `Bv` for working, `Bv-1` to 2 when under repair, 1 for failed silently, and `0` for failed critically. The complete voter module is shown in Listing 6.1.

3. The third module specifies the second voter. The PRISM language supports module renaming, i. e., the duplication of modules and substitution the local variable names. We use this possibility and define the third module as a copy of the second module that substitutes the variable `v1` with the variable `v2`, as shown in Listing 6.1.

4. The fourth module defines the order of the synchronized commands of the previous three modules.

If two commands within the same module are enabled at the same time, local nondeterminism occurs. For DTMC models, PRSIM solves this local nondeterminism by randomly selecting *one* transition of the module. However, replica, spare, and voter failures as well as failures during the repair procedure occur simultaneously within each period. Therefore, we split each period into the following $B + 4$ steps. All commands that belong to one step are synchronized by an action.

Replica failures (`RF`): The number i of replicas that fail in the current period depends on the probability of a replica failure after one period `dr` and the number of alive replicas `r`. In order to reduce the number of states of the resulting DTMC, we calculate the number of failed replicas directly in the model, following the binomial

```
[RF] r=3 ->     pow(cr,3)     : (r'=3)
             + 3*pow(cr,2)*dr  : (r'=2)&(fr'=fr+1)
             + 3*cr*pow(dr,2)  : (r'=1)&(fr'=fr+2)
             +     pow(dr,3)   : (r'=0)&(fr'=fr+3);
[RF] r=2 ->   pow(cr,2)   : (r'=2)
             +    2*cr*dr  : (r'=1)&(fr'=fr+1)
             + pow(dr,2)   : (r'=0)&(fr'=fr+2);
[RF] r=1 ->   cr : (r'=1)
             + dr : (r'=0)&(fr'=fr+1);
[RF] r=0 -> true;
```

Listing 6.2: Code excerpt of the replica module of the PRISM model with the commands that calculate the failure probabilities of 0-3 alive replicas following the binomial distribution $\mathcal{B}(\mathtt{r},\mathtt{fr})$.

distribution $\mathcal{B}(\mathtt{r},\mathtt{dr})$. An excerpt of the PRISM code for 0-3 alive replicas is shown in Listing 6.2. Note that the probability of a failure-free period of the replica is $\mathtt{cr} = 1 - \mathtt{dr}$. The number of failed replicas i is subtracted from the number of alive replicas in the next period, so $\mathtt{r}' = \mathtt{r} - i$, and added to the number of failed replicas in the next period, so $\mathtt{fr}' = \mathtt{fr} + i$.

Spare failures (SF): In a next step, the number j of spares that fail in the current period is calculated in the same way as the number of failing replicas. The number of failed spares j is subtracted from the number of alive spares in the next period, so $\mathtt{s}' = \mathtt{s} - j$, and added to the number of failed spares in the next period, so $\mathtt{fs}' = \mathtt{fs} + j$.

Last repair period (RB): Previously failed replicas are completely repaired after B periods. Hence, the number stored in each variable bB is added to the number of available spares `f'` or to the number of failed spares `fs'`, depending on the probability of a failure that occurred within the worst-case repair time $\hat{B}$. The probability of at least one failure during the B periods is calculated following the binomial distribution $\mathcal{B}(\mathtt{b}B, \mathtt{ds}^B)$. Note that we use the probability of a spare failure after one period ds to determine whether the repair procedure was successful, since like a spare the task is not executed when under repair. While this probability does not include the probability of a CDS failure explicitly, it includes the probability of OS kernel failure to which the CDS belongs.

i^{th} repair period (Ri): All replicas under repair since $i = 1,\ldots,B-1$ periods are added to the next repair phase, so bj'=bi and bi'=0 with $j = i + 1$.

Voter failures (V): In the second to last step of each period, the transitions of both voters are evaluated. Each voter fails either silently (v'=1) with probability dv or it fails critically (v'=0) with probability dc. Note that each voter is contained in a separate module and the respective commands are labeled with the same action, so they are active simultaneously. Thus, PRISM combines the probabilities of either of them or both failing in this period. If the voter is not working and the repair procedure was started in a previous period, so v<Bv & v>=2, the voter is repaired

unless it fails again with probability dv or dc, respectively, so cv:v'=v+1. The voter module is shown in Listing 6.1 on page 95.

Repair procedure (R): Finally, in the last step the number of active replicas and voters is evaluated and the repair procedures are started if the system has not failed completely. If sufficiently many replicas are still alive, so r > Mi, at least one voter is alive v1=Bv | v2=Bv, and no voter failed critically v1>0 & v2>0, then each failed replica is repaired by adding it to the first boot phase, so b1'=fr and fr'=0. For each missing replica Ni − r, a failed or alive spare is chosen randomly with probabilities fs/(s+fs) and s/(s+fs), respectively. A spare becomes a replica r'=r+1, s'=s−1 and a failed spare becomes a failed replica fr'=fr+1, fs'=fs−1. An excerpt from the repair commands in the replica module is shown in Listing 6.3 on the following page. If a voter failed silently and the other voter has not failed, e. g., v1=1 & v2=Bv, the repair procedure of the failed voter is started, e. g., v1=2. If the other voter also failed e. g., v1=1 & v2<2 or the voter failed critically, e. g., v1=0, it stays in the respective state forever. The voter repair commands are shown in Listing 6.1 on page 95.

6.3.3 Model Analysis

From the software fault-tolerance mechanism modeled in the PRISM language, the PRISM model checker generates a DTMC model. The model checker supports the automated analysis of quantitative properties of the DTMC model and provides four different model-checking engines. In order to describe the properties of the model that should be evaluated by the model checking engine, PRISM provides the PRISM property specification language that subsumes several well-known temporal logics.

We are interested in the reliability and availability of the software fault-tolerance mechanism. From the definition of reliability in Section 2.1.2 on page 15 follows that it represents the probability of all paths through the state space that do not contain a failed state. Hence, these paths contain only states with more than Mi replicas that are still alive, at least one voter that is working, and no voter that has failed in a critical way. In the PRISM property specification language, this is defined as P=?[G<=Tp*(*B*+4) r>=Mi & v1>0 & v2>0 & (v1=Bv | v2=Bv)], where Tp is the number of periods in the mission time. The first operator P=? states that we are interested in the probability of the path property stated in square brackets. The path property is a formula that evaluates to either true or false for a single path in a model. The first element of the path property is the *global* temporal operator G<=Tp*(*B*+4). The global temporal operated is an invariance, so it evaluates to true only if the following path property remains true in all Tp*(*B*+4) states along the path. Note that model divides each period in $B + 4$ steps, so the complete mission time is Tp*(*B*+4).

In order to evaluate the availability, we employ PRISM's reward mechanism. This mechanism allows to add a specific value to a reward variable whenever a guard evaluates to true. As shown in Listing 6.4 on the next page, we add the value 1/Tp and the end of each period (synchronized with the last step [R]) in case the fault-tolerance mechanism produced a result, i. e., more than Mi replicas are still alive, at least one voter is working,

```
[R] r<Ni   & s=0 & fs=0 & (r>=Mi) & (v1=Bv|v2=Bv) & v1>0 & v2>0 ->
    (b1'=fr)&(fr'=0);
    //No spare available, repair failed replica to get a new spare
[R] r<Ni   & s=0 & fs>0 & (r>=Mi) & (v1=Bv|v2=Bv) & v1>0 & v2>0 ->
    (b1'=fr)&(fr'=min(N-r,fs))&(fs'=fs-min(N-r,fs));
    //Only failed spares available, they become failed replicas
[R] r<Ni   & s>0 & fs=0 & (r>=Mi) & (v1=Bv|v2=Bv) & v1>0 & v2>0 ->
    (b1'=fr)&(fr'=0)&(s'=max(0,s-(N-r)))&(r'=min(N,r+s));
    //Only alive spares available, so use as many as required or needed
[R] r=Ni-1 & s>0 & fs>0 & (r>=Mi) & (v1=Bv|v2=Bv) & v1>0 & v2>0 ->
    (s/s+fs) :
        (b1'=fr)&(fr'=0)&(fs'=fs)&(s'=max(0,s-1))&(r'=min(N,r+1))
  + (fs/s+fs) : (b1'=fr)&(fr'=1)&(fs'=max(0,fs-1))&(s'=s)&(r'=r);
    //One replica failure, failed and alive spares available:
    //choose randomly if a failed or an alive spare is used
```

Listing 6.3: Code excerpt of the replica module of the PRISM model with the commands that repair replica failures depending on the number of available alive and failed spares.

```
rewards "alive"
   [R] (r>=Mi) & (v1>0) & (v2>0) & ((v1=Bv) | (v2=Bv)): 1/Tp; //Add
      1/Tp for every complete period which a correct result
endrewards
```

Listing 6.4: The reward in the PRISM model used to calculate the average availability.

and no voter failed in a critical way. In the PRISM property specification language, the statement `R"alive"=?[C<=Tp*(`B`+4)]` calculates the cumulative reward for the complete mission time `Tp*(`B`+4)`. Since the mission time comprises `Tp` periods, the reward variable `"alive"` represents the average number of periods without a failure, which equals the definition of the average availability given in Section 2.1.2 on page 15.

Analysis of an Unprotected Task Execution. In order to compare the resulting reliability and availability of a task protected by our software fault-tolerance mechanism with an unprotected execution of the same task, the reliability and availability of the unprotected execution have to be calculated. Although the PRISM model provides results for a respective configuration $N_i = 1$ and $S_i = 0$, these values include the probabilities of voter failures, which do not exist when a task is executed in an unprotected manner. Therefore, the reliability and availability of the unprotected execution are calculated as follows:

As defined in Section 2.1.2 on page 15, the reliability of an unprotected task T_i after the mission time t_m is

$$R(t_m) = e^{-\Lambda_T t_m} \tag{6.5}$$

on a processor core with a task failure rate Λ_T. As mentioned in Section 2.2.5 on page 31, the task failure rate $\Lambda_T = v_T \lambda$ depends on the vulnerability factor v_T of the task and the SER λ of the core.

For a continuous and unprotected system, the average availability is computed according to Equation (2.7) on page 18. The periodic nature of the real-time system allows to discretize Equation (2.7) over time. Together with the definition of the point availability in Equation (2.6) on page 17, the average availability in a periodic system is

$$A(t_m) = \frac{1}{n} \sum_{i=1}^{n} \Pr\{\text{correct service after } i \text{ periods} | \text{system new at } i = 0\},$$

where n is the number of completed periods in the mission time, so $n = \lfloor \frac{t_m}{P_i} \rfloor$. In a system without a repair procedure, the probability of a correct service after i periods is the reliability after one period $R_P = R(P_i)$ to the power of i. Note that the reliability after one period $R_P < 1$ for realistic failure rates and periods. Thus, the average availability of an unprotected task T_i in a periodic system is a geometric series, so

$$A(t_m) = \frac{1}{n} \sum_{i=1}^{n} R_P^i = \frac{R_P(1-R_P^n)}{n(1-R_P)}. \tag{6.6}$$

6.4 Workflow

In this section, we present our software fault-tolerance framework, which integrates the analysis with our fault-tolerance mechanism. First we describe an algorithm that leverages the PRISM model to determine the minimum number of necessary task replicas and spares. Afterwards, we present the workflow, interfaces, and components of the framework before we provide a comprehensive example.

6.4.1 Fault-Tolerance Analysis

The fault-tolerance analysis employs the PRISM model to determine the minimum number of task replicas and spares that are required to achieve the target reliability $\breve{R}_i$ and the target availability $\breve{A}_i$ of a task T_i with the period P_i after the mission time t_m. The failure rate of a task replica Λ_T, a spare $\Lambda_{\tilde{T}}$, a voter Λ_V, and the rate of critical voter failures Λ_C depend on SER λ and the vulnerability v of the respective component. The SER λ of any core of the many-core processor is typically specified by the hardware manufacturer or derived from fault injection experiments. The vulnerability factor v of a component depends on various properties including the size of the component in the memory, the hardware's robustness to faults, and the software implementation. Its value can be conservatively approximated by fault injection experiments and field data or estimated by experienced experts.

The fault-tolerance analysis algorithm is presented in Listing 6.5 on the next page. It starts by checking whether the task's execution has to be protected by the software fault-tolerance mechanism at all. Therefore, the algorithm first calculates the reliability and availability of an unprotected execution using Equation (6.5) and Equation (6.6). In case the resulting reliability and availability are higher than the target reliability and target availability, so $R_i(t_m) \geq \breve{R}_i(t_m)$ and $A_i(t_m) \geq \breve{A}_i(t_m)$, the task is executed without protection from the software fault-tolerance mechanism.

```
1  (N_i,S_i) faultToleranceAnalysis (Ř_i,Ǎ_i,P_i,t_m,Λ_T,Λ_T̃,Λ_V,Λ_C) {
2      //compare R and A of unprotected execution
3      R_i = e^(−Λ_T t_m);
4      n = ⌊t_m/P_i⌋;
5      R_P = e^(−Λ_T P_i);
6      A_i = R_P(1−R_P^n)/n(1−R_P);
7      if ((R_i ≥ Ř_i) && (A_i ≥ Ǎ_i)) {
8          return (1,0);
9      }
10     //compare R and S with increasing N and S
11     for (N_i = 3; N_i + S_i + 2 ≤ |𝒞|; N_i+=2) {
12         M = ⌈(N_i + 1)/2⌉;
13         for (S_i = 0; ((S_i ≤ M) && (N_i + S_i + 2 ≤ |𝒞|)); S_i++) {
14             (R_i,A_i) =
                   runPRISMModelChecker(Λ_T,Λ_T̃,Λ_V,Λ_C,t_m,P_i,N_i,S_i);
15             if ((R_i ≥ Ř_i) && (A_i ≥ Ǎ_i)) {
16                 return (N_i,S_i);
17             }
18         }
19     }
20     return (NOT_FOUND);
21 }
```

Listing 6.5: Fault-tolerance analysis algorithm.

Otherwise, the algorithm initially selects the parameters $N_i = 3$ and $S_i = 0$, which represent the smallest number of task replicas and spares required by the software fault-tolerance mechanism. The algorithm uses the PRISM model to calculate the reliability $R_i(t_m)$ and availability $A_i(t_m)$ of the task for the selected parameter setting. If the resulting reliability and availability are higher than the target reliability and target availability, so $R_i(t_m) \geq \check{R}_i(t_m)$ and $A_i(t_m) \geq \check{A}_i(t_m)$, the current parameter setting is returned.

Otherwise, the number of spares S_i is increased until $S_i = M_i$. We set the maximum number of spares to $M_i = \lceil \frac{N_i+1}{2} \rceil$, since the majority voter is unable to determine a result if more than M_i replicas are under repair in the same period. If the target reliability and target availability are not achieved with the maximum number of spares M_i, the number of task replicas N_i is increased by 2 and the number of spares S_i is reset to 0.

The fault-tolerance algorithm continues to increase the number of spares and replicas until the target reliability and target availability are achieved or all cores of the many-core processor are used by the two voters, the task replicas, and the spares, so $N_i + S_i + 2 > |\mathscr{C}|$. In the latter case, the required fault-tolerance level cannot be achieved by the software fault-tolerance mechanism on the given many-core processor and a respective information is returned.

As a result of the fault-tolerance analysis, the parameters N_i and S_i are selected such that the task achieves the target reliability and target availability with the minimum number of task replicas and spares, and thus with minimum overhead.

6.4.2 Fault-Tolerance Framework

The software fault-tolerance framework combines the fault-tolerance mechanism with the fault-tolerance analysis. The components and the workflow of the framework are presented in Figure 6.2. The variables depicted on the left side of Figure 6.2 represent the interface for the domain experts. For example, the task T_i and its period P_i are defined by a real-time specialist, the target reliability $\breve{R}_i$, the target availability $\breve{A}_i$, as well as the mission time t_m are specified by a safety analyst, and the failure rates of all involved components Λ_T, $\Lambda_{\tilde{T}}$, Λ_V, and Λ_C are determined by a hardware expert from the processor's SER λ and the respective component's vulnerability factor v_T, $v_{\tilde{T}}$, v_V, and v_C.

The fault-tolerance analysis is carried out for each task $T_i \in \mathcal{T}$. The target reliability $\breve{R}_i(t_m)$ and target availability $\breve{A}_i(t_m)$ of each task T_i as well as the mission time t_m are specified by the developer. These values are typically derived from the system's specification, its environment, and the safety standards the system has to fulfill. As mentioned before, the failure rate of a task replica Λ_T, a spare $\Lambda_{\tilde{T}}$, a voter Λ_V, and the rate of critical voter failures caused by the residual SPOFs Λ_C are typically derived from fault injection experiments, field data, or estimated by experienced experts.

As a result of the fault-tolerance analysis, the number of task replicas N_i and spares S_i are known or the analysis showed that the specified fault-tolerance level cannot be achieved with the given hardware and the software fault-tolerance mechanism, in which case the workflow is canceled with an error.

The implementation of the elements of the software fault-tolerance mechanism presented in the previous chapter are contained within the framework. For each task, the framework encapsulates the original task function in the task wrappers and it creates the respective voters. Furthermore, it adds the CDS, SDS, the reset service as well as the DMPS and drivers to the OS kernel.

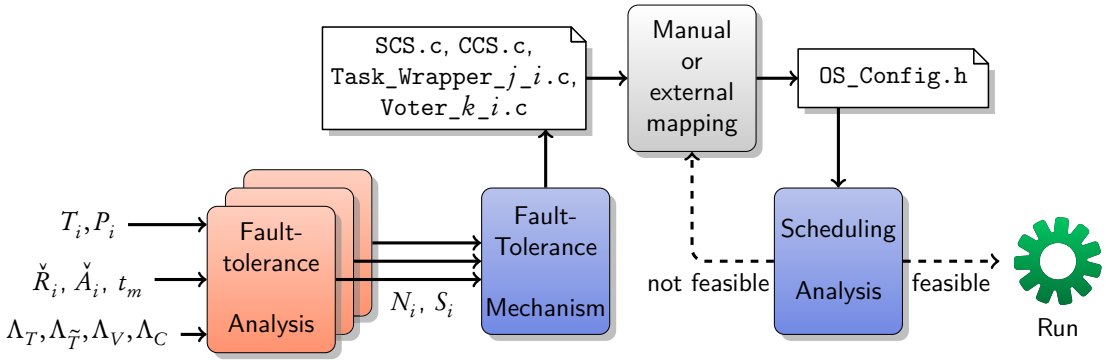


Figure 6.2: Overview of the software fault-tolerance framework. The data flows are shown as solid lines, the control flows are shown as dashed lines. The input variables on the left hand side represent the interfaces for the domain experts.

The complete software is then mapped to the many-core processor, following the constraints of the software fault-tolerance mechanism. The mapping is performed either manually or by using existing approaches described in Section 2.3.3 on page 36. As a result, each component is mapped to one core of the many-core processor, which is depicted as `OS_config.h` in Figure 6.2 on page 101.

If a mapping is found, the mapped software is input to the scheduling analysis described in Section 5.4.3 on page 79. In case the WCRTs of all task wrappers and voters are shorter than their respective deadlines, the schedule is feasible and the system is ready to be executed. If the schedule is infeasible, the task mapping has to be adjusted accordingly.

6.5 Real-World Application Example

Highly automated driving (HAD) is a prominent example of a safety-critical application with high computational demands and real-time constraints. HAD applications typically comprise several parts [LAB+11]. First, various sensor data, e. g., from cameras, radar devices, ultrasonic sensors, or Global Positioning System (GPS) units, is analyzed to detect objects and features such as obstacles and the road surface. This detection is often executed within the sensor itself and the detected objects and features are sent to a main controller.

In the main controller, the features and objects detected by separate sensors are merged, a map of the environment is generated, and the position of the vehicle within this map is estimated. At the same time, the trajectory is planned. The trajectory is a combination of the high-level route to the destination and low-level maneuvers, e. g., due to recognized traffic lights or obstacles.

Finally, the trajectory is used to calculate the steering angle and the acceleration or breaking signal. These signals are sent to the ECUs that control the electronic steering, the combustion engine or the electric motor, and the breaking system. As a real-time system, the various parts of the HAD application are typically encapsulated in periodically executed tasks.

6.5.1 Application Model

The goal of this example is to show that the proposed software fault-tolerance framework can be applied to real-world applications. However, most HAD applications are currently not designed for many-core processors but assume a shared memory architecture in the main controller and an SMP OS. For example, most software implementations make extensive use of look-up tables that are accessed by several tasks from different cores. On a many-core processor, the look-up table would either be replicated to decrease the access latency or the values would be computed during runtime, leveraging the additional computational power. Additionally, a complete HAD application is too complex to be executed within reasonable time on a simulated many-core processor, which is presented in the next chapter.

Therefore, this example contains only a subset of four mixed-critical tasks. These four mixed-critical tasks are already known from the example of the previous chapter and Section 5.5 on page 81. The example is limited to a single many-core processor, which is

equivalent to the scope of our software fault-tolerance mechanism. Note that a common goal in the automotive industry is to combine the functionality of multiple ECUs in a single or a few distributed control systems to reduce hardware costs.

6.5.2 Target Fault-Tolerance

The safety-criticality of the system and its components is derived from the safety standard for the automotive industry, the ISO 26262 [Int11]. Note that this thesis does not argue whether the ISO 26262 is applicable for HAD applications. We merely use the safety standard to exemplify the functionality of our software fault-tolerance framework.

The ISO 26262 describes how hazard analysis and risk assessment determine the safety goal and automotive safety integrity level (ASIL) of a safety item. In case the failure of a safety item has a high severity, is difficult to control by the driver, and has a high probability to occur, the safety item has the highest safety goal ASIL D. A safety item is a system or array of systems that implement a function at the vehicle level [Int11]. The ASIL of a safety item is inherited by all systems, which include elements such as a software unit or task, respectively [Int11].

Considering transient hardware faults, the ISO 26262 specifies a *probabilistic metric for random hardware failures (PMHF)* that has to be achieved. For the safety goal ASIL D, the ISO 26262 defines the *random hardware failure target* $\breve{\lambda} < 10^{-8}\,\mathrm{h}^{-1}$. For the safety goals ASIL C and ASIL B, the random hardware failure targets are both $\breve{\lambda} < 10^{-7}\,\mathrm{h}^{-1}$. For ASIL A, the ISO 26262 does not require to evaluate the safety goal violations due to random hardware failures. However, the ISO 26262 is derived from the IEC 61508, the international standard for electrical, electronic, and programmable electronic safety-critical systems [Int10]. Similar to the ISO 26262, the IEC 61508 defines four safety integrity levels (SILs). The IEC 61508 requires the *average probability of a failure* of a safety-critical function to be $< 10^{-8}\,\mathrm{h}^{-1}$ for the highest SIL 4 as well as $< 10^{-5}\,\mathrm{h}^{-1}$ for the lowest SIL 1, respectively [Int10]. While the random hardware failure target of the ISO 26262 must not be confused with the average probability of a failure of a safety-critical function required by the IEC 61508, we nevertheless assume a random hardware failure target $\breve{\lambda} < 10^{-5}\,\mathrm{h}^{-1}$ for ASIL A in this example.

The ASIL requirement of each task of the example task set is defined in Table 6.1 on the following page. Note that the ASIL requirements are chosen to represent a mixed-critical application such as HAD. Task T_4 has no safety-critical functionality, which is denoted as class Quality Management (QM) by the ISO 26262.

For each of the remaining three tasks, we derive the target reliability $\breve{R}(t_m)$ and target availability $\breve{A}(t_m)$. We select the mission time $t_m = 24\,\mathrm{h}$, since we assume this the maximum amount of time an HAD vehicle operates without any interruptions. Any interruption leads to a restart of the application and many-core processor, respectively, which causes all unprotected core-local memories to be reloaded and results in the fault-free state.

As mentioned in Section 2.2.5 on page 31, only a subset of faults causes a task to fail. Therefore, we calculate the target failure rate of a task $\breve{\Lambda}_T = v_T \breve{\lambda}$ depending on the vulnerability factor v_T of a task and the random hardware failure target $\breve{\lambda}$. Without

Task	$P = D$	ASIL	$\check{R}$	$\check{A}$
T_1	10 ms	D	0.999999985	0.9999999927
T_2	10 ms	A	0.999985	0.9999927
T_3	20 ms	C	0.99999985	0.999999927
T_4	10 ms	QM	–	–

Table 6.1: A specified safety goal (ASIL) and the required target reliability $\check{R}$ and target availability $\check{A}$ of the mixed-critical real-time task set example after a mission time $t_m = 24\,\text{h}$.

anticipating the results of Section 7.4.1 on page 130, we set the task vulnerability factor v_T to 0.061. For example, task T_1 with the safety goal ASIL D and the random hardware failure target $\check{\lambda} < 10^{-8}\,\text{h}^{-1}$ has a target failure rate $\check{\Lambda}_{T_1} = 6.1 \cdot 10^{-10}\,\text{h}^{-1}$. With Equation (6.5) on page 98 and Equation (6.6) on page 99, we calculate its target reliability $\check{R}_1 > 0.999999985$ and its target availability $\check{A}_1 > 9999999927$ after the mission time $t_m = 24\,\text{h}$. The respective value for T_2 and T_3 are calculated in the same way and are presented in Table 6.1.

6.5.3 Hardware Properties

In this thesis, we aim to increase the fault-tolerance of COTS consumer-grade many-core processors in software. One example of such a processor is Adapteva's Epiphany III, which is produced in a 65 nm process and has 32 kB local SRAM memory per core [Ada13]. A single SRAM cell has an SER of 10^{-4} to 10^{-2} FIT [Sla11]. This SER is constant for processes from 250 nm down to 50 nm [Sla11]. The reason is that on the one hand, the critical charge of particle strikes decreases with smaller cells, which increases the SER, and on the other hand, the cell area shrinks, which decreases the cross section for particle strikes and thus the SER [Sla11].

Hence, the memory of each core of the considered COTS many-core processor has an SER λ between $2.62 \cdot 10^{-8}\,\frac{1}{\text{h}}$ and $2.62 \cdot 10^{-6}\,\frac{1}{\text{h}}$. Around 10% of all soft errors in an architecturally similar OpenRISC 1200 processor occur in the CPU's register file, combinational logic, and the flip-flops [EET+14, EET+15]. Thus, the SER of each core is set to $\lambda \leq 2.87 \cdot 10^{-6}\,\frac{1}{\text{h}}$ in this example. Note that soft errors in the NoC are not considered in the SER λ, since we focus on the SER of a single core.

Considering the task vulnerability factor $v_T = 0.061$, the failure rate of the task executed on one core of the COTS many-core processor is $\Lambda \leq 1.75 \cdot 10^{-7}\,\frac{1}{\text{h}}$. With this failure rate Λ, the unprotected execution of any task on one core of the COTS many-core processor results in the reliability $R = 0.99999974$ and the availability $A = 0.99999987$ after the mission time $t_m = 24\,\text{h}$.

6.5.4 Software Fault-Tolerance Framework

The software fault-tolerance framework uses the fault-tolerance analysis algorithm for each task of the example task set to determine the minimum number of task replicas and spares that are required to achieve the target reliability and target availability.

Since task T_4 is not safety-critical, it is executed without any protection. The previous calculation of the reliability $R = 0.99999974$ and availability $A = 0.99999987$ of an unprotected execution of any task on one core of the COTS many-core processor shows that target reliability $\breve{R}_2 = 0.999985$ and target availability $\breve{A}_2 = 0.9999927$ of task T_2 with the safety goal ASIL A are achieved without the software fault-tolerance mechanism. However, the target reliabilities and availabilities of task T_1 with ASIL D and task T_3 with ASIL C are not met without a fault-tolerance mechanism.

Based on the fault injection experiments presented in Section 7.4.1 on page 130, we assume a task vulnerability factor $v_T = 0.061$, a spare vulnerability factor $v_{\tilde{T}} = 0.05484$, a voter vulnerability factor $v_V = 0.7031$, and a vulnerability factor $v_C = 4.881 \cdot 10^{-5}$ of critical voter failures. With these vulnerability factors and the SER $\lambda = 2.87 \cdot 10^{-6} \frac{1}{\text{h}}$ of one core of the considered COTS many-core processor, the analysis of the PRISM model of our software fault-tolerance mechanism with $N_i = 3$ task replicas and $S = 0_i$ spares results in the reliability $R = 0.9999999965$ and the availability $A = 0.9999999931$ after a mission time $t_m = 24\,\text{h}$. Hence, the fault-tolerance analysis determines that the target reliability $\breve{R}_1 = 0.999999985$ and target availability $\breve{A}_1 = 0.9999999927$ of task T_1 with safety goal ASIL D as well as the target reliability $\breve{R}_3 = 0.99999985$ and target availability $\breve{A}_3 = 0.999999927$ of task T_3 with the safety goal ASIL C can both be achieved with the lowest possible number of replicas $N_3 = 3$ and spares $S_3 = 0$.

With this result of the fault-tolerance analysis algorithm, the software fault-tolerance framework replicates the respective tasks three times, encapsulates them in the task wrapper, initializes the necessary voters, and generates all components of the repair procedure, as shown in Figure 6.2 on page 101. The resulting task set is mapped to the cores of the many-core processor with the additional constraint that task replicas, voters, and spares of the same task have to be mapped to mutually exclusive cores. Note that the replicas and voters of separate tasks can be mapped to the same core.

In order to evaluate different configurations of the software fault-tolerance mechanism in the following and to create a more interesting mapping problem, we arbitrarily modify the parameters N_i and S_i of the task set as follows: Task T_1 has $N_1 = 5$ replicas and $S_1 = 2$ spares, task T_2 has $N_2 = 3$ replicas and $S_2 = 1$ spare, and task T_3 has $N_3 = 5$ replicas and no spares, so $S_3 = 0$. Note that these configurations achieve a higher reliability and availability than required by the safety standard. The task set example with modified parameters is shown in Table 5.1 on page 82. The task mapping of the mixed-critical task set is presented in Section 5.5.1 on page 82.

6.6 Summary

In this chapter, we have presented our software fault-tolerance framework. The goal of the framework is to ease the application of the software-based hardware fault-tolerance

mechanism described in the previous chapter. Therefore, the framework provides interfaces to describe the application as a mixed-critical real-time task set, to specify the safety requirements in form of the target reliability as well as the target availability of each task, and to provide the hardware properties as the SER per core and the vulnerability factors of the software components. The functionality of the framework is demonstrated by an example derived from a real-world application. The example shows that the safety requirements of some tasks of a mixed-critical task set can only be achieved on a COTS consumer-grade many-core processor when the software fault-tolerance framework is employed.

We have presented a fault-tolerance analysis algorithm to determine the minimum numbers of task replicas and spares that are required to achieve the target reliability and target availability of each task. Our fault-tolerance analysis employs the PRISM probabilistic model checking engine to determine the resulting reliability and availability of a task protected by a specific configuration of the software fault-tolerance mechanism. The input of the probabilistic model checking engine is a model of our fault-tolerance mechanism in the PRISM language. The underlying mathematical concept of this model is a DTMC. In Section 7.4 on page 130, we compare the results of our PRISM model with the values obtained from an experimental evaluation of the proposed software fault-tolerance mechanism.

7 Evaluation

In this chapter, we experimentally evaluate the proposed mechanism presented in the previous chapters. First, we introduce the experimental setup, i. e., the many-core processor simulator and the fault injection method. Then, we experimentally verify the dynamic guaranteed service (GS) communication concept presented in Chapter 4 on this simulated many-core processor. Next, we detail the implementation of our software-based hardware fault-tolerance mechanism presented in Chapter 5 and measure the achieved reliability and availability as well as the generated overhead in terms of system load. Finally, we compare the measured fault-tolerance with the results obtained by the fault-tolerance analysis method presented in Chapter 6.

The work presented in this chapter is published in [MFRC15, MAL+15, MAL+16].

7.1 Experimental Setup

In the following, we present the cycle-accurate and bit-accurate (CABA) many-core processor simulator. Afterwards, we introduce our fault injection mechanism that is part of this simulator.

7.1.1 Hardware Simulator

In contrast to higher-level simulations such as instruction-level or task-level, all timing effects such as the contention on the NoC are accurately simulated at CABA level. This allows us to precisely evaluate the real-time capability of the software. However, this high accuracy comes with the disadvantage of a slow simulation speed. An FPGA implementation of the many-core processor and the fault injection extension can decrease the runtime of a single experiment significantly. However, various experiments with different injected faults are necessary to generate statistically meaningful results. In contrast to the FPGA implementation, the single-threaded CABA simulation can easily be executed in parallel on modern multi-core servers or server clusters. Therefore, all experiments were conducted on a CABA many-core processor simulator.

Our simulator is based on SoCLib[1] [PBdM+09]. SoCLib is an open-source platform for virtual prototyping that includes a library with several virtual intellectual property (IP) components, such as CPUs, timers, caches, memories, crossbars, NoCs, and TTY interfaces. These virtual components communicate with the VCI protocol [On-01]. The

[1] Subversion (SVN) revision 2581 from January 20, 2015

VCI protocol specifies the communication between an initiator and target component based on requests and responses.

SoClib is based on SystemC, an American National Standards Institute (ANSI) standard C++ class library for system and hardware design which includes an event-driven simulation kernel [IEE12]. To speed up the simulation, we use the open-source SystemCASS library[2] [BPG04] instead of the original SystemC simulation engine. In contrast to SystemC, SystemCASS was optimized for SoCLibs CABA components and uses static scheduling, which is enabled by a statically derived signal dependency graph [BPG04]. The many-core processor simulator that uses the SoClib and SystemCASS libraries is compiled with GCC version 4.7.3.

Core Design

The simulated processor is configured in the style of a COTS many-core processor, as defined in Section 2.2 on page 19. Each core is comprised of a CPU with a 32 Bit little endian Microprocessor without Interlocked Pipeline Stage (MIPS) architecture, 64 KiB RAM, a timer, an inter-core reset (ICR) mechanism, a TTY interface, and a network interface (NI) connected by a local crossbar, as shown in core $C_{1,0}$ of Figure 7.1 on the facing page.

The MIPS32El CPU provides a RISC instruction set and has a pipeline with five stages [Uhl05]. The on-chip memories are uncached since they are directly implemented on the chip and can be accessed with low latency. The floating point unit is turned off since we do not assume a floating point unit (FPU) to be available at each core of a many-core processor. For this reason, the application uses software-implemented floating point operations. The timer is provided by SoCLib's XICU component. The core-local crossbar implements a round-robin arbitration policy in case of conflicts, i. e., in case several initiators try to reach the same target. Since SoCLib's RAM component does not support multiple banks, we added two RAMs to the chip. One RAM component contains the `.mailbox` section and the other component the remaining sections.

Network-on-Chip

The simulated processor contains $y \cdot x$ cores $C_{j,i} \in \mathscr{C}$ with $j = \{1,2,\ldots,y\}$ and $i = \{1,2,\ldots,x\}$, which are connected by the Distributed, Scalable, Predictable Interconnect Network (DSPIN). The DSPIN [MPGS06] is a 2D packet switching NoC that uses a dimension order (XY) routing policy with wormhole switching and round-robin arbitration. The DSPIN implementation provided as a virtual component in SoCLib does neither support virtual channels nor does it support different clock domains.

Following the VCI protocol for an interface to another communication fabric, the NI consists of two parts, the initiator NI and the target NI. As explained in Section 2.2.3 on page 23, if request and response packets were transmitted in the same NoC, deadlocks could arise due to dependencies at message level [HGR07]. Therefore, the processor contains two NoCs, one for the requests and one for the responses, as depicted in Figure 2.4 on page 24. The flit size in the request network is 39 bits and the flit size in the response network is 32 bits. The initiator NI translates the VCI requests, e. g., from CPU on core

[2] SVN revision 58 from October 14, 2015

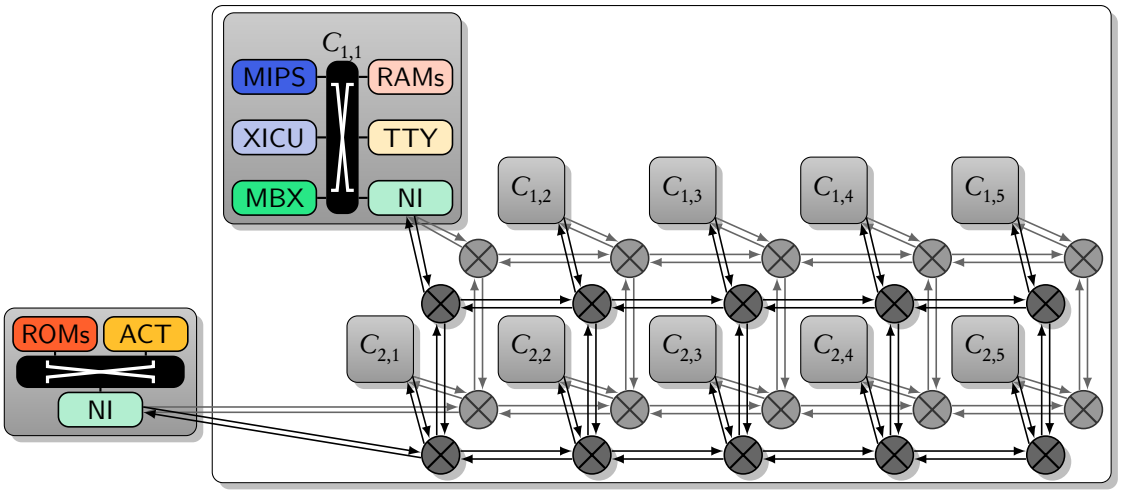


Figure 7.1: The architecture of the simulated many-core processor with $5 \cdot 2$ cores. The cores are connected by the request NoC (dark) and the response NoC (light). Both NoCs also connect the external read-only memory (ROM) and actuator (ACT) interface. Core $C_{1,1}$ is shown with its internal components, a MIPS32El CPU, two RAMs, a timer (INT), an ICR mechanism (MBX), a TTY interface, and an NI.

$C_{1,1}$ into a packet comprised of multiple flits, which are sent through the request NoC. When the target NI receives all flits of a packet, it translates it back to a VCI request and forwards it to the local target component, e. g., the RAM on core $C_{2,5}$. The local target component receives the request and creates a response, e. g., the requested data or information about the success of a write operation. This response is then sent to the response port of the target NI that translates it into flits and sends them through the response NoC. Finally, when the initiator NI received all flits from the response network, it forwards the VCI response to the requesting component, the CPU on core $C_{1,1}$ in this example.

Figure 7.1 depicts the architecture of a simulated many-core processor with $5 \cdot 2$ cores as well as the external ROM and actuator interface. Core $C_{1,1}$ is shown with its internal components. All cores are connected by two DSPIN NoCs, one for transferring requests and one for transferring the corresponding responses.

Memory Map

The simulated processor provides a 32 bits flat global address space. Each CPU can access the components of the other cores via the NoC. The complete memory map is shown in Table 7.1 on the following page. In the address, the first two nibbles decode the row and column of the core. We extended the instruction set to map all addresses of the very first core, i. e., all addresses that start with `0x00`, to the local memory of the executing core, so one executable can be loaded on any core. The third nibble of the address decodes the core-local component. To implement the MPU functionality in the simulator, we adapted the NI to allow write requests only to the `.mailbox` section in any other core's RAM.

Address From	Address To	Element
0x0000 0000	0x00ff ffff	Mapped to the local core
0x1100 0000	0x1100 f000	External-write-protected RAM bank of core $C_{1,1}$
0x1110 0000	0x1110 1000	Unprotected RAM bank (the `.mailbox` section)
0x1120 0000	0x1120 0010	Memory-mapped TTY interface
0x1130 0000	0x1130 1000	Timer interface (SoCLib XICU component)
0x1140 0000	0x1140 0300	Reset interrupt trigger (SoCLib Mailbox component)
0x1150 0000	0x1150 0004	Output interface for RTOS trace; data is converted into a trace file by simulator
0xji00 0000	0xjiff ffff	Core $C_{j,i}$ with $j = \{1,2,\ldots,y\}$ and $i = \{1,2,\ldots,x\}$, same setup as core $C_{1,1}$
0x2000 0000	0x2000 a000	ROM bank 0
0x2010 0000	0x2010 a000	ROM bank 1
0x2020 0000	0x2020 a000	ROM bank 2
0x2030 0000	0x2030 0020	Virtual actuator interface; data is stored with a time-stamp externally

Table 7.1: The memory map of the simulated many-core processor.

In order to be able to determine the load of each core, we employ the tracing facility of the RTOS on each core. The tracing events are stored in a special memory section. All data written to this section is stored in a trace file outside the simulator together with the current number of simulated cycles.

To implement the ICR mechanism, a mailbox component was added to each core. Note that the mailbox component is not to be confused with the `.mailbox` section in the RAM. The mailbox component is provided by the SoCLib library and provides memory-mapped interrupt triggers. It is configured to provide as many triggers as cores, such that each core is able to reset another core. Note that the interrupt signals are point-to-point connections and independent of the NoC.

In our hardware model, we assume a reliable external memory to be connected to the processor. To mimic a ROM with multiple banks, we attached three uncached ROMs of size 40 KiB to the NoC. When the simulator starts, each ROM component loads the sections of a specified file in the Executable and Linkable Format (ELF).

Depending on its ID, the CPU fetches the first instructions from the respective external ROM over the NoC. These first instructions copy the respective `.text` and `.data` sections for a voter, a task replica, or the unprotected task core from the ROM into the unreliable core-local RAM. After the copy routine, the execution of the OS kernel is started by setting the program counter (PC) to a start address in the core-local RAM. Note that the selection of which core loads which sections can be done completely in software, too.

In order to analyze the reliability and availability of the simulated system, an additional virtual actuator interface (ACT) is added to the NoC. All data written to this virtual

actuator interface is stored outside the simulator together with a timestamp given in simulated cycles. Since the application and its timing requirements are statically known, the timestamp and value of the generated result can be compared with its deadline and the expected correct value. This way, we can calculate the fault-tolerance of the simulated system.

7.1.2 Fault Injection

Under normal operating conditions, the occurrence rate of faults is very low and requires extensive experiments, e. g., measuring the soft errors in 64 devices with 7 Gbits SRAM cells in total at 2552 m altitude for 6702 h of operation [ASM+12]. In order to decrease the experimental effort and speed up the experiment duration, fault injection is a commonly used technique [HTI97].

Faults can be injected either at hardware-level or in software. Hardware-implemented fault injection methods based on heavy-ion radiation, pin-level injection, or electromagnetic interference require a dedicated equipment and a complex setup [HTI97]. In contrast, software-based fault injection approaches do not require specialized hardware and allow repeatable experiments by adding fault injecting code to the target software. However, these modifications influence the system under test and can cause side effects. By injecting faults in a simulated hardware platform that executes the software under test, researchers have limited the side effects without requiring specialized equipment [DJPI96].

The fault space of all possible bit flips at all possible points in time is too large to be evaluated completely within reasonable time, even if only single bit flips are considered [SHD+15]. Following the physical nature soft errors, we choose a random sampling approach: faults are injected in random locations at random points in time. The spatial fault distribution follows a uniform probability, whereas the temporal fault injections are exponentially distributed. Hence, the more experiments are performed the better is the statistical approximation of the reliability and availability.

Compared to the related work presented in Chapter 3 on page 41, our fault injection mechanism does not assume an abstract task-level fault rate but is based on the SER of the physical hardware. Furthermore, our fault injection mechanism is not restricted to single bit flips but generates multiple faults distributed randomly in time and over components. We extend the SoCLib-based many-core processor simulator by a fault injection mechanism. As a CABA simulator, the simulated level of detail allows to inject all faults specified in our fault hypothesis present in Section 2.2.5 on page 28. This statistical fault injection on microarchitectural simulators can provide early and accurate reliability characterization [KTCG15]. In the following, we describe the fault injection implementation in the memories, register files, and the NoC.

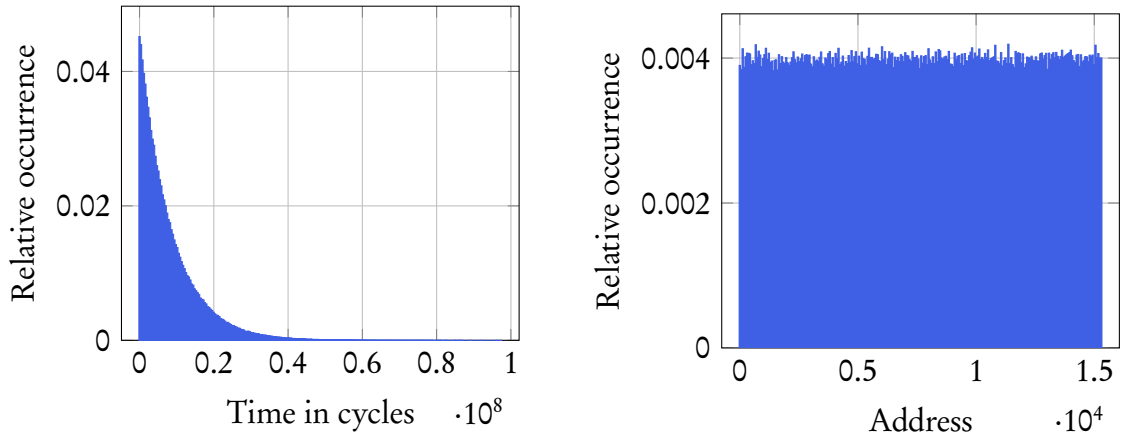
Core-Local RAM

To inject soft errors in the core-local memories, we extended the RAM component used by each core. The SER λ per core is provided as input parameter to the simulator. It is used to calculate the fault injection cycle t_f at which a soft error is injected. Follow the inverse transform sampling method, this point in time t_f is calculated by (7.1) [Dev86],

where U is a uniformly distributed random number within the interval $(0,1)$ generated by the C++ rand()/RAND_MAX statement.

$$t_f = \frac{-\ln(1-U)}{\lambda} \tag{7.1}$$

When the simulator reached the fault injection cycle t_f, the error is injected by flipping one randomly selected bit of a randomly selected memory cell following a uniform distribution. Additionally, the next fault injection cycle t_f is computed. Note that soft errors affecting two memories in separate cores can occur in parallel, since each core has its own memory component. A visualization of the temporal and spatial distribution of the faults is given in figure Figure 7.2.



(a) In time **(b)** In space

Figure 7.2: A histogram with 250 bins of the fault injection times and core-local memory locations with an SER $\lambda = 1.1 \cdot 10^{-7}$ after 819,340 injections.

Core-Local Register File

Soft errors also affect the CPU and its internal components. To include such errors in our simulation, we inject bit flips in the register file. Similar to the core-local memory, the fault injection mechanism waits for an exponentially distributed random number of cycles and then flips a random bit in a randomly selected register. Note that the PC as part of the register file is subject to these bit flips as well.

Ebrahimi et al. [EET+14, EET+15] analyzed the soft error distribution for the OpenRISC 1200 processor, which is architecturally very similar to the simulated MIPS32El with its 32 bit RISC architecture and the five-staged pipeline. For this processor with 4 KiB cache and 32 registers the authors found that for different embedded workloads 90% of soft errors affect the cache. Less than 1% of faults affected the combinational logic and the flip-flops. Since the effects of such faults, e. g., in pipeline, the arithmetical logical unit (ALU), and the integer multiplication and division unit, lead to wrong register values in most cases, no additional faults were injected there. Ebrahimi et al. found that the

remaining 9% of faults affected the register file. Following these results, we set the SER for faults in the register file to 10% of the SER λ for the core-local memories.

Network-on-Chip

Apart from the core-local components like RAMs and CPUs, soft errors also affect the NoC. They can result in bit flips in the flits currently stored in the router and NI buffers. A bit flip affecting the first flit results in the packet being delivered to the wrong address. In most cases, this address either does not exist, is protected by the MPU, or is badly aligned. Thus, the probability of a bit flip causing a packet to overwrite existing data stored in a `.mailbox` section is low. A bit flip in the second flit that decodes the VCI request is either masked or causes the NI to reject the packet. A soft error affecting the third flit causes wrong data to be written to the correct memory address.

The combinational logic and the flip-flops of the NI and the router with its internal crossbar are affected by soft errors, too. These soft errors in logic can cause a flit to be dropped, forwarded to the wrong output port, or manipulate its data. As a result, the effects are similar to soft errors affecting the occupied NoC buffers.

Apart from particle strikes, soft errors in the long inter-router wires are caused by crosstalk coupling [RFZJ13, FA12]. Crosstalk coupling is the result of electromagnetic interference that occurs under specific transition patterns between parallel wires [RFZJ13, FA12]. Crosstalk coupling causes glitches and delays, which again result in bit flips in the flits currently sent via the wire. However, an analysis of the crosstalk coupling in the inter-router wires would require a complete synthesis of the simulated many-core processor and therefore is out of scope of this thesis.

From a packet point of view, the end-to-end fault rate depends on its size, its route through the NoC, and the router design [VWOH15]. Hence, fault injection at the target NIs has to determine the fault rate for each packet separately. Additionally, the traffic pattern and contention it causes within the NoC affects the fault rate of each packet, since a packet that is blocked and stored in a router buffer is subject to soft errors for a longer period.

Therefore, we inject faults directly within the DSPIN routers. Similar to the fault injection method in the core-local RAMs and register files, a random bit in a randomly chosen element of a randomly selected FIFO input buffer is flipped after an exponentially distributed random amount of cycles.

In the simulated many-core processor, every router has 5 input ports and each port has a FIFO buffer for 4 flits. The flit size in the request network is 39 bits and 32 bits bits in the response network. Thus, a router of the request network provides 98 bytes of SRAM, which is equivalent to 0.146% of the size of the core-local SRAM. In order to include faults from crosstalk coupling in the wires as well as in the combinational logic and the flip-flops of the router and the NI, we set the SER per router to 1% the memory fault rate λ.

7.2 Dynamic GS Communication

In this section, we use the CABA simulated many-core processor described in the previous section to evaluate our limited packet injection rate approach. This approach is presented in Chapter 4 and guarantees GS communication in the DSPIN NoC, which supports only best-effort communication in hardware.

The experiments are performed on the many-core processor simulator with deactivated fault injection. The parameters of the simulated hardware are as follows: The simulated processor is configured to contain 16 cores with $x = y = 4$. The packet size in the request and the response network is $|p_i| = 3\,\text{flits}$, the latency of a router is $L_r = 3\,\text{cycles}$, the blockage latency in case of a collision is $\hat{L_C} = 4\,\text{cycles}$, the latency of the destination core is $\hat{L_\delta} = 2\,\text{cycles}$, and the bandwidth of any link $l_i \in \mathscr{L}$ is $b = 1\,\frac{\text{flit}}{\text{cycle}}$.

In order to evaluate our approach, we are interested in the transfer latencies and the load on the links. Both values are obtained for the SystemC trace generated by the DSPIN NoC. In order to decrease the simulation runtime and to minimize side effects, all software is implemented at assembler level without an OS.

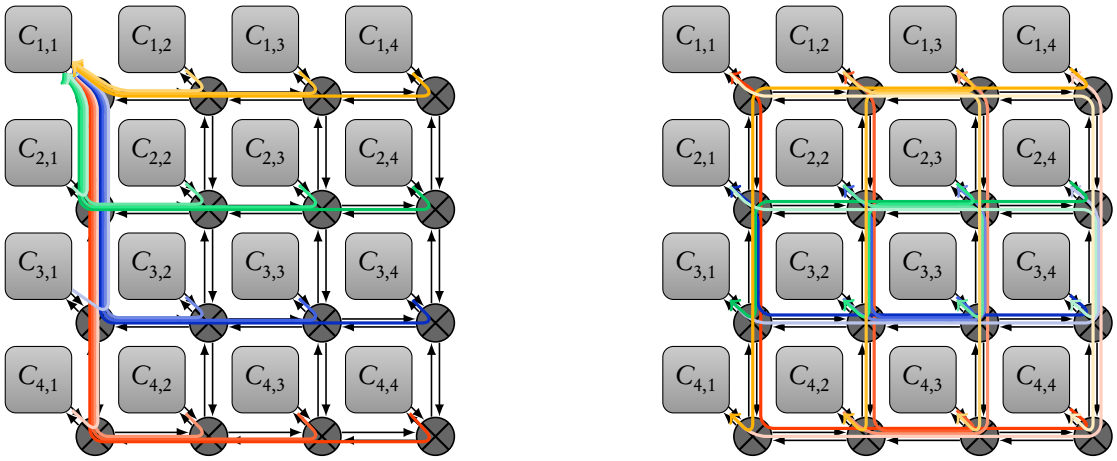
In this section, we first present different traffic patterns that are used for the evaluation of the limited packet injection rate approach. Next, we measure the transfer latencies of a selected traffic pattern under different packet injection rates. These measurements are compared to the WCTL and corresponding packet injection rate in the following. In order to indicate the level of pessimism introduced by our approach, we finally measure the load on the links for different injection rates.

7.2.1 Traffic Patterns

First, we describe the traffic patterns, i. e., the set of all request packets $\overrightarrow{\mathscr{P}}$, that are used to measure the transfer latencies in the following experiments.

Unfortunately, the traffic pattern that results in a transfer latency as long as the theoretical WCTL $\hat{L}$ is unknown. Such a traffic pattern requires a request packet to be blocked by one packet from all other sources in the request NoC and the corresponding response packet to be blocked by one packet from all other sources in the response NoC as well. This behavior is difficult to achieve since packets for all sources have to collide in each network. It is trivial to create a traffic pattern in which all packets collide in the request network by forcing all packets to use a single common link. However, due to the dimension order (XY) routing policy, this single common link is located between a router and an NI of a core. Thus, all response packets are injected one after the other and do not collide in the response network. Note that in order to derive the WCTL, we assume there is a traffic pattern where all packets collide in the request NoC *and* in the response NoC.

In the following experiments, we use two well-known traffic patterns: The first traffic pattern maximizes the transfer latency in the request NoC. This *latency* traffic pattern is achieved if all sources send packets to the same destination with their maximum allowed packet injection rate F, as illustrated in Figure 7.3a on the facing page. Since all sources inject packets with a common destination, all request packets potentially collide in the last link between the request NoC and the destination core's NI. The request packets with the



(a) Latency (all-to-one) traffic pattern

(b) Throughput (uniform all-to-all) traffic pattern

Figure 7.3: Visualization of the packet routes in the request NoC for two traffic patterns.

highest number of hops from source to destination arrive at the last link after all other request packets and hence are potentially blocked by all other packets.

The opposite extreme traffic pattern balances the traffic optimally in the request NoC. Since this maximizes the network's throughput, it is also known as the worst-case *throughput* traffic pattern [TD02]. In the throughput traffic pattern, each source core $C_{j,i}$ sends packets to the destination core $C_{y-j+1,x-i+1}$, as shown in Figure 7.3b.

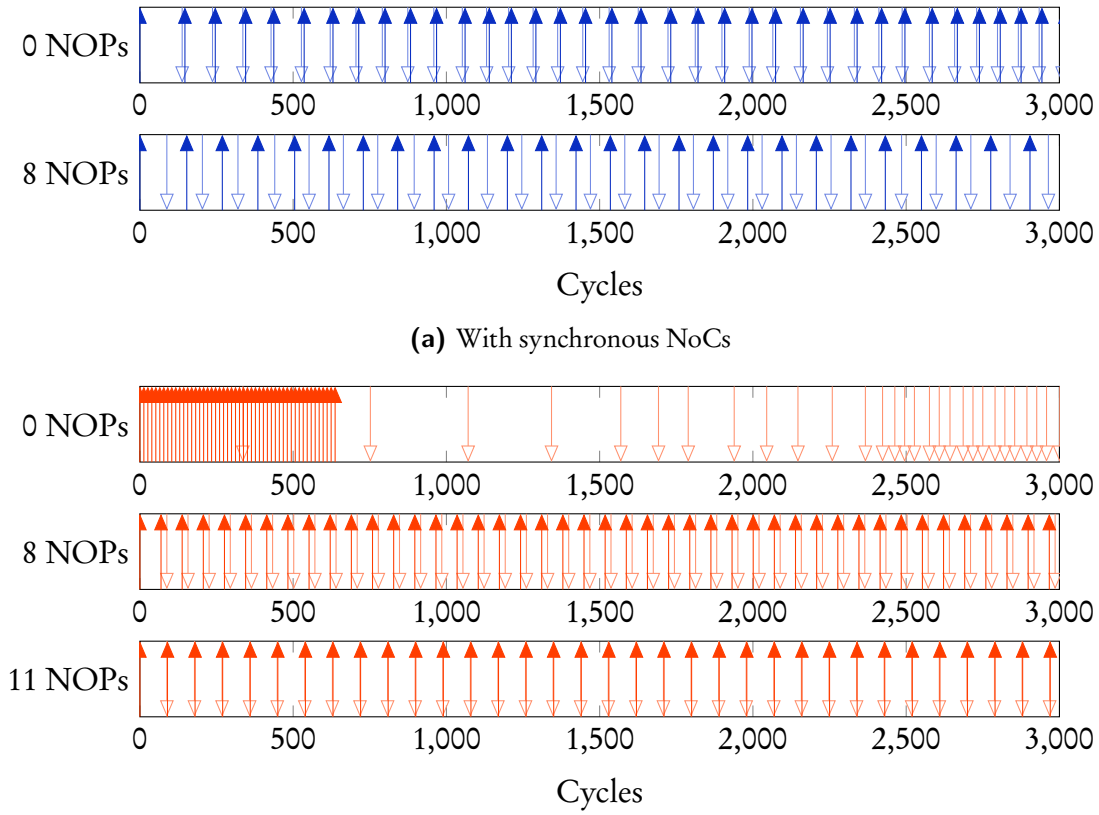
The third traffic pattern we employ in the following experiments is the *random* traffic pattern. In this traffic pattern, each core sends packets to other randomly selected cores. Two consecutive request packets can be sent to different cores. The random destination addresses are computed offline.

7.2.2 Transfer Latency Measurements

In a first experiment, we measure the transfer latencies under the latency traffic pattern, so all cores send request packets to core $C_{1,1}$. Due to the router-level flow control, an NI cannot inject new request packets into the request NoC if the input buffer of the first router is full. Since we want to capture the maximum latency of each packet, we set the buffer size of both NoCs to 50 packets and 150 flits, respectively. Each source sends 50 request packets, so all NIs are always able to inject new packets.

Figure 7.4 on the next page shows the packet injection times and the corresponding response packet receive times of core $C_{4,4}$. Note that core $C_{4,4}$ sends packets with the highest number of hops to the destination, which results in the worst-case traversal latency $\hat{L_{\not\subset}}$.

Figure 7.4a on the following page shows the latencies in the original, synchronous version of DSPIN, where the NIs wait for the response packet while the CPU is stalled. Without decreasing the packet injection rate (0 NOPs), the transfer latency of the first packet sent from $C_{4,4}$ is 138 cycles. This is due to cores closer to the destination which



(a) With synchronous NoCs

(b) With asynchronous NoCs

Figure 7.4: Trace of 50 request packet injection times and the corresponding response packet receive times under the latency traffic pattern within the first 3,000 cycles of source core $C_{4,4}$. Dark blue filled up arrows represent injected request packets and light blue empty down arrows represent arriving response packets. Each trace represents a different packet injection rate implemented by a different number of NOP instructions between two consecutive packet injections.

have a shorter traversal latency and inject multiple packets before the request packet of first packet sent from source $C_{4,4}$ arrives at the destination.

By decreasing the packet injection rate for all sources, this effect is prevented and the measured traversal latency is reduced. After a waiting time equivalent to 8 NOPs, the traversal latency of the first packet sent from $C_{4,4}$ is reduced to 88 cycles. Hence, we found an injection rate that bounds the blockage of the first packet sent from $C_{4,4}$ for the latency traffic pattern. Adding more NOP instructions between packet injections does not decrease the transfer latency of the first packet anymore.

The synchronous nature of the DSPIN NoC has the disadvantage that the CPU is stalled during the transfer, while it could potentially execute memory-independent instructions. In order to achieve a common packet injection rate for all sources, the time each CPU stalls has to be subtracted from the additional waiting time added by software. Note that

the time each CPU stalls is different for each source, since it depends on the transmission time of the injected packet and hence on the traffic pattern.

Therefore, we adapted DSPIN's NIs to support asynchronous behavior and alleviate the mentioned disadvantages of the synchronous NoC. Figure 7.4b on page 116 shows the latencies in the asynchronous version of DSPIN.

Without decreasing the packet injection rate (0 NOPs), the request and the response NoC are both severely overloaded. In order to reduce the blocking effects, we decrease the injection rate of all sources by adding NOP instructions between two packet injections. As a result, the transfer latency of the first packet sent from source $C_{4,4}$ is reduced. When 8 or more NOP instructions are inserted, the transfer latency of this packet is constantly 88 cycles. Note that an injection rate equivalent to 8 NOP in an asynchronous network is higher compared to a synchronous network, since the CPU processes the first NOP instruction immediately instead of being stalled until the response packet is received.

When the packet injection rate is equivalent to 11 or more NOP instructions, the response packet corresponding to the first request packet is received before the second request packet is sent. The reason for transfer latency of 88 cycles with a lower injection rate is that there are no collisions in the response network under the latency traffic pattern. Thus, the next request packet can be injected a certain time before the previous response packet is received without creating more collisions.

Both experiments show that a lower packet injection rate reduces the blocking effects, reduces the transfer latency, and thus maximizes the number of transferred packets. Additionally, we measured that a packet injection rate F equivalent to 8 NOP instructions is sufficient to limit the blockage delay in the asynchronous and in the synchronous version of the NoC and to guarantee a constant transfer latency under the latency traffic pattern.

7.2.3 Worst-Case Transfer Latency Evaluation

Next, we calculate the WCTL and compare it to the maximum measured transfer latencies (MMTLs) of the different traffic patterns. We use the parameters of the simulation platform and Equation (4.2) on page 56 to calculate the WCTL $\hat{L}$ as follows:

$$\begin{aligned}
\hat{L} &= 2\hat{L_p} + \hat{L_\delta} \\
&= 2(\hat{L_{\not c}} + \hat{L_B}) + \hat{L_\delta} \\
&= 2((x+y-1)(L_r + \frac{1\,\text{flit}}{b}) + \frac{|p_i|}{b} + (xy-2)\hat{L_C}) + \hat{L_\delta} \\
&= (2((4+4-1)(3+1)+3+(4\cdot 4-2)\cdot 4)+2)\,\text{cycles} \\
&= 176\,\text{cycles}
\end{aligned}$$

The definition of the packet injection rate $F = \frac{1}{\hat{L}}$ requires each core to wait 176 cycles before injecting the next packet. Due to disabled caches and the latency of local memory accesses, each NOP instruction takes 7 cycles. Hence, a waiting time of 176 cycles requires 26 NOP instructions.

In Figure 7.5 on page 119, the MMTLs for different traffic patterns are plotted. When the packet injection rate F is obeyed and consecutive injections are separated by 26 NOPs,

none of the simulated traffic patterns had an MMTL higher than 176 cycles of the WCTL. Hence, the calculated WCTL is indeed an upper bound for all simulated traffic patterns.

Under the throughput traffic pattern with 1,000 injected request packets, the MMTLs are at most 53 cycles in synchronous NoCs and 49 cycles in asynchronous NoCs, as shown in Figure 7.5a on the facing page. Furthermore, the packet injection rate does *not* influence the measured transfer latency in asynchronous NoCs and only very little in synchronous NoCs. The reason is that the throughput traffic pattern is designed to cause few collisions.

Compared to the throughput traffic pattern, the latency traffic pattern with 50 injected request packets causes large MMTLs, as shown in Figure 7.5b on the next page. Especially if the packet injection rate is high and the NoCs are asynchronous, the MMTL is 2,791 cycles. The reason is the design of the latency traffic pattern, which causes many collisions since all packets are sent to a common destination.

In order to find a traffic pattern that causes even higher transfer latencies, we performed 800 simulations with a random traffic pattern. In the random traffic pattern, each core injects 1,000 packets to random destinations cores. The MMTLs of all 800 repetitions are plotted in Figure 7.5c on the facing page. However, none of the 800 runs with the random traffic pattern resulted in a latency in the range of latency traffic pattern.

7.2.4 Load Measurements

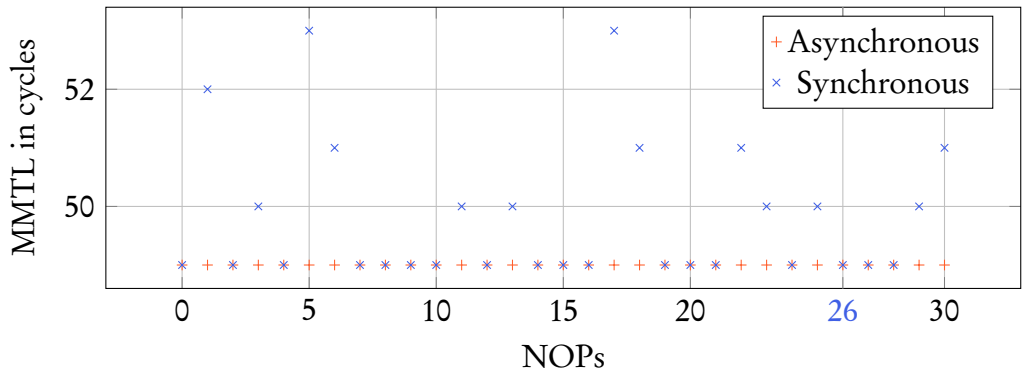
In this experiment, we compare the average load in the links of the request and the response NoC under different packet injection rates. The load of each link is calculated by dividing the number of transferred flits by the number of cycles of the entire experiment. In the following, we use the average link load that is the sum of the loads of all links in both NoCs divided by the total number of links.

Figure 7.6 on page 120 shows the measured average link load under the latency traffic pattern with different packet injection rates. The bend of the load curve at 7 NOPs in the asynchronous NoCs and 3 NOPs in the synchronous NoCs is due to links that are 100% loaded and router-level flow-control causing back-pressure.

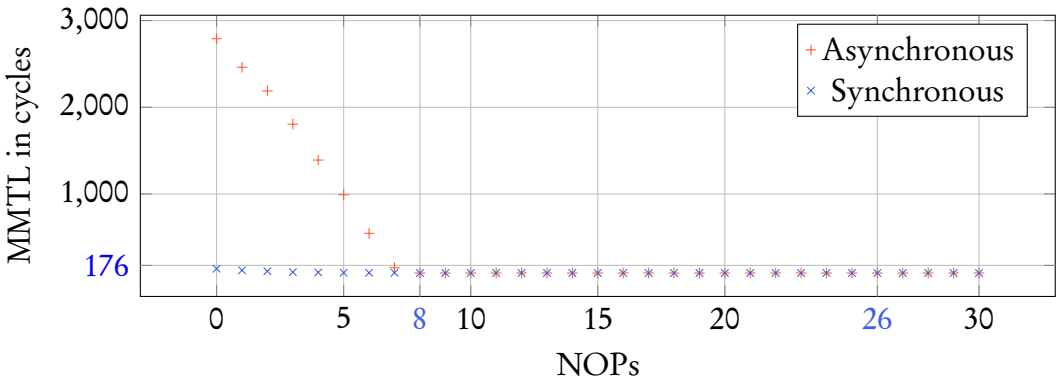
With an injection rate $F = \frac{1}{176\,\text{cycles}}$ and 26 NOPs between consecutive packet injections, respectively, the load is 1.06% in asynchronous NoCs and 0.869% in synchronous NoCs. The first experiment in Section 7.2.2 on page 115 shows that a packet injection rate F equivalent to 8 NOPs is sufficient to achieve a bounded MMTL. With a delay of 8 NOPs, the link load is 2.94% in asynchronous NoCs and 1.81% in synchronous NoCs. Hence, for the latency traffic pattern our approach is pessimistic by a factor of 2.77 in asynchronous NoCs and 2.08 in synchronous NoCs for the latency traffic pattern.

The origin of the lower average link load as well as the discrepancy between the MMTL and the WCTL is manifold:

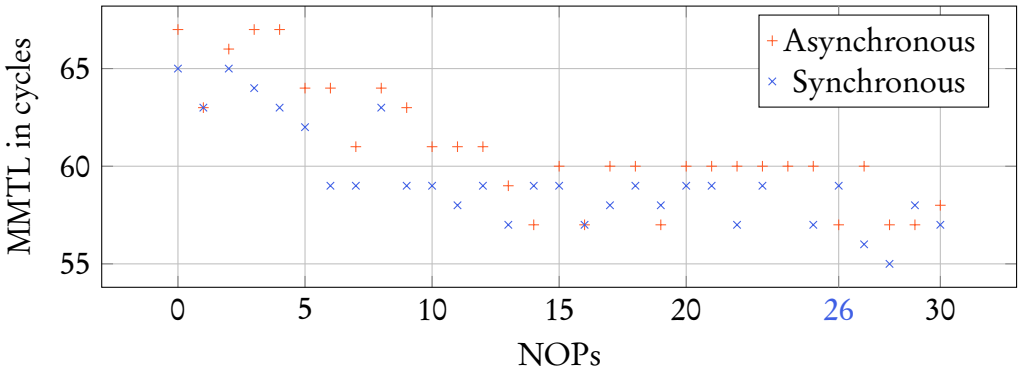
1. The traffic pattern that results in the WCTL is unknown. Amongst the three investigated traffic patterns, the latency traffic pattern causes the highest MMTL. Under the latency traffic pattern all packets are routed over one common link in the request network. However, no collisions are expected in the response NoC under the latency traffic pattern, since the responses are injected one after another by a single source only.



(a) Throughput traffic pattern with 1,000 request packets per source



(b) Latency traffic pattern with 50 request packets per source



(c) Maximum of 800 repetitions of a random traffic pattern with 1,000 request packets per source

Figure 7.5: Maximum measured transfer latencies (MMTLs) of various traffic patterns with different packet injection rates in synchronous and asynchronous NoCs.

2. The worst-case behavior is assumed for each collision, i. e., a packet is always assumed to be blocked for the duration to forward an entire packet. In reality, a packet can also win the arbitration or it is blocked only be the last flit of another packet.

3. In the worst-case, a packet is blocked by packets from all other sources. In our implementation of the latency traffic pattern, all sources start injecting packets at

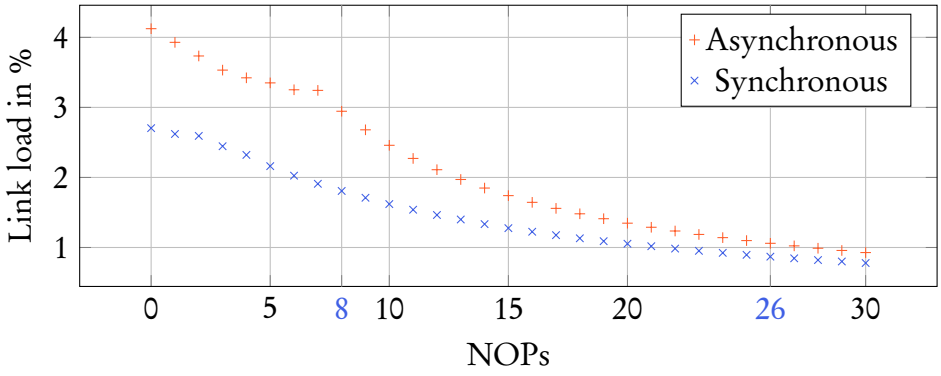


Figure 7.6: Measured average load on the links of synchronous and asynchronous NoCs for different packet injection rates under the latency traffic pattern.

the same time. Hence, the request packet $\overrightarrow{p_i}$ with $\sigma(\overrightarrow{p_i}) = C_{4,4}$ is blocked by only 9 packets. The other packets are transferred before they can collide with packet $\overrightarrow{p_i}$.

4. The latency of a router L_r is only 2 cycles if the link was free before. We calculated with $L_r = 3$ cycles, since it takes one additional cycle to detect whether a previously occupied link is free again.

7.3 Fault-Tolerance Evaluation

In this section, we experimentally verify the proposed software fault-tolerance mechanism. First, we describe the implementation details of the mixed-critical task-set and the real-time many-core OS used in the experiment. Then, we present and discuss the measured reliability and availability of the task-set. Finally, we evaluate the overhead of our fault-tolerance mechanism in terms of system load.

7.3.1 Implementation

For the evaluation of the software fault-tolerance mechanism, we reuse the well-known task set with four mixed-critical tasks described in Table 5.1 on page 82 and Table 6.1 on page 104. To summarize: Task T_1 is replicated $N_1 = 5$ times and has $S_1 = 2$ spares, task T_2 is replicated $N_2 = 3$ times and has $S_2 = 1$ spare, task T_3 is replicated $N_3 = 5$ times but has no spares ($S_3 = 0$), and task T_4 is not safety-critical, so it is not replicated at all. All tasks except task T_3 have a period of $P_1 = P_2 = P_4 = 10\,\text{ms}$, task T_3 has a period of $P_3 = 20\,\text{ms}$. The worst-case repair time is 2 hyperperiods, so $\hat{B} = 2\mathring{P}$.

Sample Algorithm

A task's vulnerability to soft errors depends on the algorithm it executes. Each instruction and static data element of the compiled algorithm is subject to soft errors while it is executed by the CPU. The task's vulnerability to soft errors also depends on the number of compiled instructions and the size of the static data that are stored in the core-local

memory, which is affected by soft errors, too. Hence, an algorithm that only waits for a specific execution time, which was measured before in a real system, and finally outputs a statically configured result is not representatively affected by soft errors. A comparison of the reliability and availability of tasks that execute different algorithms would include their different vulnerabilities. Therefore, all four mixed-critical tasks in the task set execute the same sample algorithm in order to allow a fair comparison of different parameter settings of our fault-tolerance mechanism.

The sample algorithm is designed to represent a mixture of typical workloads of embedded software components. It consists of the following parts: First, the task counts the bits of a given sensor value using a look-up table. The bit counting routine is selected from the automotive category of MiBench benchmark suit for embedded applications [GRE+01]. Next, the task executes stack-intensive recursive procedure calls to calculate the Fibonacci number of the number of bits of the sensor value. Finally, the resulting Fibonacci number is the input of three software-implemented floating point operations, namely multiplication, division, and square root taken from the Newlib C 2.2.0 library. The resulting floating point number is truncated to integer and sent to the voter or outputted by the actuator driver, respectively.

Operating System

As described in Section 2.3.3 on page 36, we combine the concepts of RTOSs with the design of many-core OSs by managing each core by its own RTOS kernel. As RTOS kernel, we select FreeRTOS V8.2.1 [Bar10b]. FreeRTOS is a memory-efficient, configurable, and open-source RTOS. It supports dynamic task management, so the task execution can be stopped and re-initiated at runtime. To port FreeRTOS to the specific setup of our simulated many-core processor, we added a new hardware abstraction layer (HAL) to the kernel. The complete OS kernel and all processes executed on the local core reside completely inside the core's local RAM.

On each core, the processes are scheduled by FreeRTOS's fixed-priority preemptive scheduler. FreeRTOS uses a periodic timer interrupt, i. e., a tick, to keep track of the time and switch the execution of processes. We set the tick interrupt to 25,000 cycles, which is equivalent to 0.25 ms assuming a processor speed of 100 MHz.

FreeRTOS provides macros and hooks that allow to trace events of several components, including the scheduler and dispatcher [Bar10b]. In order to determine the task runtimes and the system load of each core, we employ this tracing facility. All tracing events are stored in a special memory section such that the simulator is able to generate an external trace file, as described in Section 7.1.1 on page 107.

Since FreeRTOS does not provide an IPC mechanism for processes on separate cores managed by different kernels, we extended the OS by a DMPS. The DMPS obeys the maximum packet injection rate. It provides an interface for each task that allows to specify the core and a slot to which a message should be sent. The message size is set to 32 bits, so a message is read as an atomic entity and no locking mechanism is required. The receiver process of the message is not interrupted but has to check for new messages itself. When a task reads the message from a specific core and slot, the message is deleted. Using this message passing mechanism, the replicas of one task running on separate cores send their

results to a statically assigned slot number on both voter cores. When the voter of this task is executed, it reads all messages from replica cores by checking the respective slots. The scheduling and the WCCT ensures that all messages for the replicas are transmitted by the DMPS before the voter task is scheduled.

In order to provide input data for the sample algorithm of each task, we added virtual sensor drivers to FreeRTOS. On each core, the virtual sensor driver delivers a sequence of input values to all task replicas. These input values have to be the same for all tasks in order to allow a fair comparison of their reliability and availability. However, the sequence of inputs starts with a different value for each task in order to detect a failed voter that overwrites the output of the voter of another task. The input values are calculated by each virtual sensor driver in a loop that repeats itself after 50 values. Hence, the virtual sensor driver's loop counter of each task represents a variable state that has to be copied from a fault-free replica during a replica repair. Solely because the input values and the sample algorithm of the tasks are statically known, failures can be detected as wrong outputs after the simulation.

Compilation and Boot Process

To compile the RTOS together with the task set, we use the GCC version 4.8.3 for the 32 bit little endian MIPS architecture with floating point emulation and optimization level 2.

Three separate files in the ELF are generated: one for the core that executes the task T_4 without protection, one for the cores that execute the voters, and one for cores that execute the task replicas of T_1, T_2, and T_3. For simplicity, replicas of all tasks are contained in the latter executable file. Depending on the task mapping and the ID of the CPU that executes the software, only a subset of replica tasks is instantiated by the local OS.

As mentioned in Section 7.1.1 on page 107, the many-core simulator loads the compiled ELF files into the external ROMs prior to the simulation. When a core of the many-core processor is reset, the CPU fetches the first instructions from one of the external ROM over the NoC depending on its ID. Note that for simplicity, we statically configured which core fetches its first instructions from which external ROM in the simulated hardware. This configuration can as well be included in the first instructions.

These first instructions initiate the boot process and copy the `.text` and `.data` sections that include the OS kernel and the tasks from the external ROM to the core-local RAM. After the copy procedure, the local RAM of the core executing task T_4 is occupied to 79.4%, the RAM of a core executing all voter tasks is occupied to 69.8%, and the RAM containing replicas of all tasks except task T_4 is occupied to 91.8%.

Furthermore, the boot process clears all remaining sections, namely the `.mailbox`, `.stack`, and `.bss` sections. Using the tracing facility of the simulated many-core processor, we measure the maximum duration of the boot process to be 2,883,197 cycles. Assuming a processor speed of 100 MHz, the boot process takes 28.8 ms. Since the boot duration is not a multiple of the hyperperiod $\mathring{P} = 20$ ms of the task set, it has to be extended in order to continue execution in synchronization with the other cores after a reset. Therefore, the OS waits by polling a hardware cycle counter after all data is copied to the core-local RAM

and all remaining sections are cleared. After the repair time $\hat{B} = 2\mathring{P} = 40\,\text{ms}$ has passed, the boot process finishes by enabling the scheduler and the tick interrupt, respectively.

Execution Times

A histogram of the measured execution times of each task is shown in Figure 7.7. The execution times are measured using the tracing facility of FreeRTOS. No faults were injected during the measurement of the execution time.

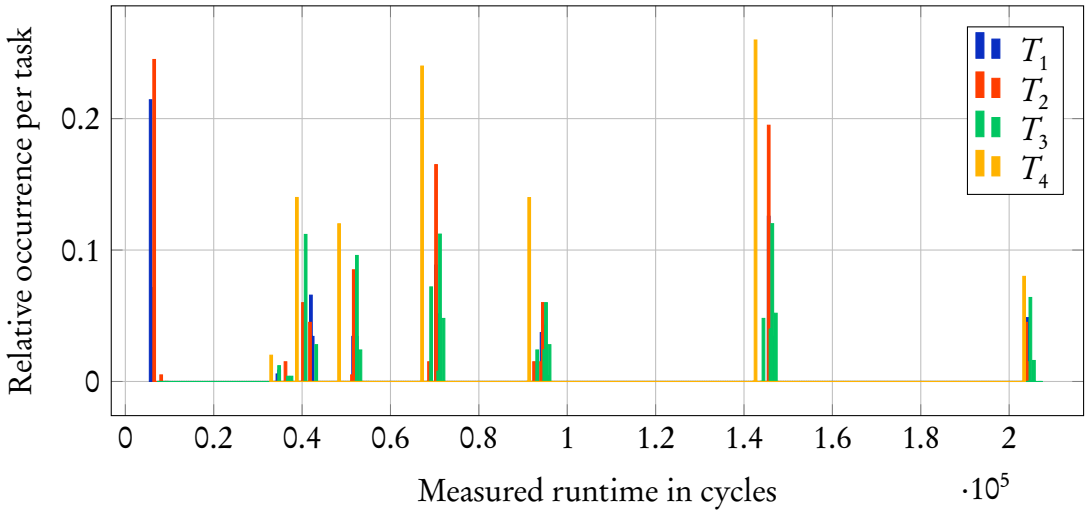


Figure 7.7: A histogram with 500 bins of the measured execution times of all tasks after 100,315 periods of task T_1 with $P_1 = 10\,\text{ms}$ or 16.7 min of simulated real-world time, respectively. The bins are derived from the execution times of each task separately.

Figure 7.7 shows that the measured execution times of the unprotected task T_4 are slightly lower compared to the execution times of the remaining tasks. The reason is that in contrast to the remaining tasks, task T_4 is not encapsulated by the task wrapper. The histogram also reveals that 21.4% and 25.0% of measured execution times of tasks T_1 and T_2 take only 6,814 and 6,888 cycles, respectively. This is due to tasks T_1 and T_2 being configured with $\widetilde{T}_1 = 2$ and $\widetilde{T}_2 = 1$ spares, respectively, and spares do not execute the original task's algorithm but merely check whether they have been activated by the SDS.

The longest measured execution time of all tasks is 206,843 cycles. With a clock frequency of 100 Mhz, this equals 2.06 ms. Note that in Table 5.1 on page 82, the WCETs of all tasks except task T_3 are 2.5 ms, which represents a safe upper bound of the maximum measured execution time. However, the WCET of task T_3 is 4.5 ms in Table 5.1 on page 82. This value was chosen to demonstrate the scheduling analysis for the mixed-critical task set. However, as mentioned before, all tasks have to execute the same sample algorithm in order to be able to compare the reliability and availability of different parametrization of the software fault-tolerance mechanism.

7.3.2 Measured Fault-Tolerance

The software described in the previous section is executed on the simulated many-core processor. The simulator injects faults with a constant SER λ, as explained in Section 7.1.2 on page 111.

In order to measure the reliability and average availability of each of the four tasks, we compare all results generated by each task with the expected results. Furthermore, we check whether each result was received by the virtual actuator before its respective deadline. This is possible since the sample algorithm of all tasks as well as the input values are statically known. Thus, even if more than M_i replicas fail with the same result and the majority voter forwards this wrong result, we are able to detect the system failure.

In case all results generated by a task within the mission time t_m are correct and received in time, the reliability of the task is 1, otherwise it is 0. The fraction of correct and timely results generated by each task represents the task's average availability.

In order to cope with computational effort of the CABA processor simulation, the mission time is set to $t_m = 10^8$ cycles, which is equivalent to 1 s assuming a clock frequency of 100 Mhz. With this mission time, the average runtime of a single-threaded simulation experiments is 53.7 ± 1.67 min on an Intel Xeon E3-1270v3 processor. To compensate the short mission time, the SER λ is increased respectively. For comparison: an SER $\lambda = 10^{-7} \frac{1}{\text{cycles}}$ is equivalent to $36,000 \frac{1}{\text{h}}$ when assuming a clock frequency of 100 Mhz.

Failure Types

In a first experiment, we investigate the different failure types that are caused by the injected faults. The different failure types are explained in the following:

No result. No result was received by the virtual actuator before the task's original deadline. This failure type represents silent failures, which are also known as detected system failures or detected uncorrectable failures.

Wrong result. A wrong result was received by the virtual actuator. This failure type represents critical failures, undetected system failures, and SDCs, respectively.

More than two results. We accept receiving two correct results from both voters of the same task within the same period, since this might occur in case one voter core is freshly repaired and the other voter core fails because of a different task. However, if more than two results are received in one period, this is considered as a critical failure since it indicates the failure of at least one voter.

Two results from one voter. Two results from the same voter were received within one period. This is considered a critical failure since each voter is expected to deliver only one result per period.

Voter results differ. Both voters of a single tasks send a result to the virtual actuator but these results differ. This failure type represents a critical failure.

In this experiment, SER is set to $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$. In order to generate statistically meaningful results, the experiment was repeated 10,000 times. Note that the specific fault injection times and locations are randomly selected in each experiment.

Figure 7.8 on the next page plots the average absolute number of measured failures and their respective types for each of the four tasks of the mixed-critical task set. Note that the absolute number of failures does not directly represent a task's reliability and availability. For example, Figure 7.8 on the following page shows that task T_3 has the lowest number of silent failures. However, task T_3 also has a longer period than the other tasks, thus the number of expected results is lower. Nevertheless, the results already indicate that the software fault-tolerance mechanism is able to decrease the number of failures.

Reliability and Availability Distribution

Knowing the various failure types, we are now interested in the distribution of the measured reliability and availability.

For a single experiment with a constant SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$, the measured reliability of a task is only 1 if all results are correct and in time. Otherwise, the measured reliability of the task is 0. Hence, the distribution of the measured reliability is bipolar.

The empirical mean of the measured reliability converges to the theoretical reliability for the number of experiment $n \to \infty$ [Bir14, p. 4]. Therefore, the experiment was repeated $n = 10,000$ times with a constant SER but random fault injection times and locations. In the following, we denote the mean measured reliability of all experiments as the reliability. Since the reliability distribution is bipolar, other measures than the mean such as the variance and standard deviation are not applicable.

Figure 7.9 on page 127 plots the measured bipolar reliability distribution of each task of the mixed-critical task set after $n = 10,000$ experiments. The figure shows that tasks with a higher number of replicas N and spares S have a higher average reliability. For example, the reliability of task T_1 with $N = 5$ and $S = 2$ is $R_1 = 93.3\%$ and the reliability of the unprotected task T_4 is $R_4 = 66.1\%$ after the mission time $t_m = 10^8$ cycles and with an SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$.

The measured availability of each experiment is computed as the number of correct and timely generated results divided by the total number of expected results. Figure 7.10 on page 127 shows a histogram with 100 bins of each task's measured availability on the simulated many-core processor with an SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$. The histogram represents the distribution of the measured availability after 10,000 experiments. Note that for the depicted distribution, measures such as variance and standard deviation are not applicable.

Figure 7.10 shows that low availability values were measured less often for tasks with a higher number of replicas N and spares S. Thus, tasks with a high number of replicas N and spares S have a higher average measured availability. For example, task T_1 with $N = 5$ and spares $S = 2$ has a measured availability $A > 0.99$ in 93.3% of experiments, while the unprotected task T_4 has a measured availability $A > 0.99$ only in 66.1% of experiments. Note that an experiment with a measured availability $A = 0$ means that no correct and timely result was generated, most likely because both voters failed before they could

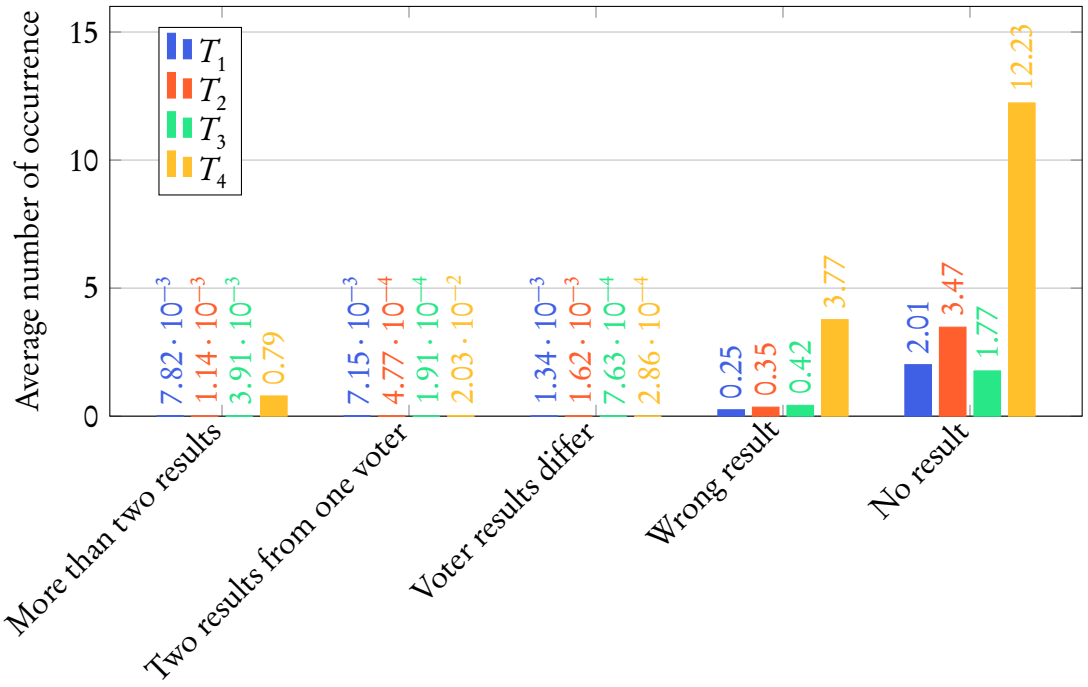


Figure 7.8: The average absolute number of failures and their respective types that each task T_i experiences within the mission time on the simulated many-core processor with an SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$.

initially activate the task replicas. Following the same reasoning as for the reliability, we consider the empirical mean of the measured availabilities as availability in the following.

Fault-Tolerance over Mission Time

Figure 7.11 on page 128 plots the reliability and availability of each task after various mission times t_m. The minimum mission time $t_m = 10^7$ cycles is equivalent to 100 ms and the maximum mission time $t_m = 10^8$ cycles is equivalent to 1 s assuming a clock frequency of 100 MHz. The SER is $\lambda = 1.1 \cdot 10^{-7} \frac{1}{\text{cycles}}$. Each experiment was repeated 10,000 times.

The measured results show that for an increasing mission time, the reliability and availability decreases less steeply for a larger number of replicas N and spares S. In all cases, the reliability and availability of tasks that are protected by the software fault-tolerance mechanism is higher than the reliability and availability of the unprotected task T_4. However, the figure also indicates that for a mission time $t_m \to \infty$, the reliability and availability of any task $R_i = A_i = 0$.

Fault-Tolerance over SER

Figure 7.12 on page 128 plots the reliability R availability A of each task on the simulated many-core processor with various SERs λ. For each SER λ, the experiment was repeated 10,000 times. The mission time is constant $t_m = 10^8$ cycles.

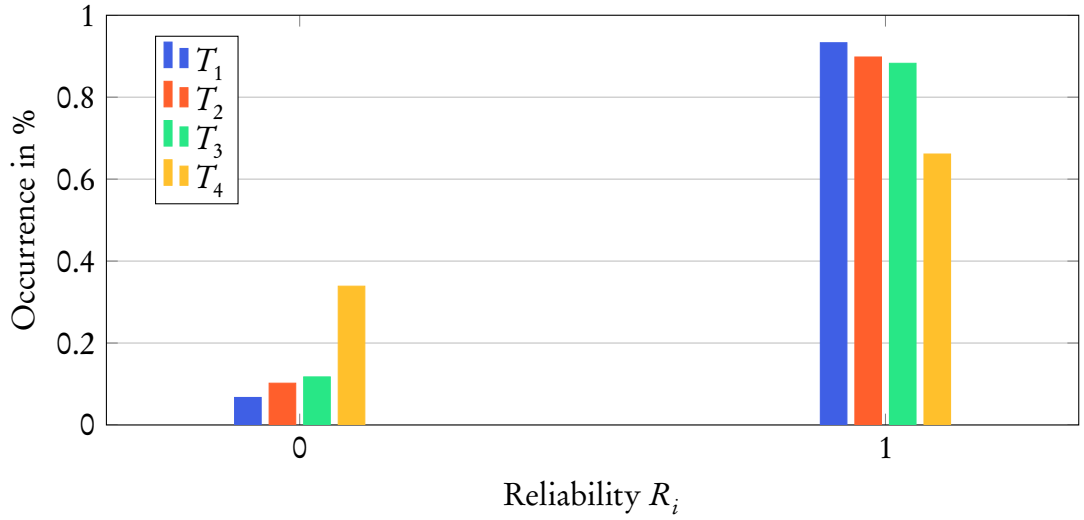


Figure 7.9: The bipolar distribution of the measured reliability R_i distribution of each task T_i after the mission time $t_m = 10^8$ cycles on the simulated many-core processor with an SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$.

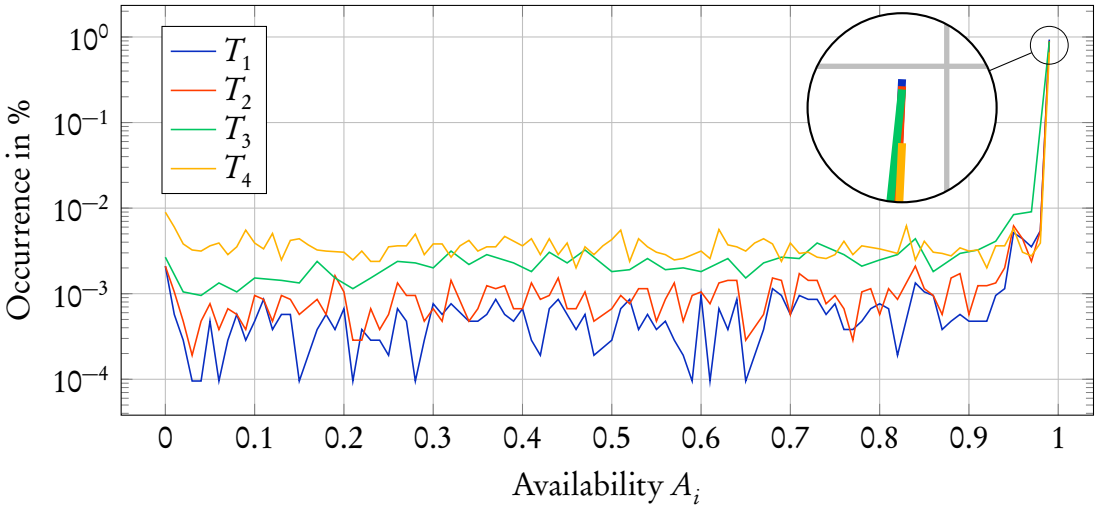


Figure 7.10: Histogram with 100 bins of the measured availability A_i distribution of each task T_i after the mission time $t_m = 10^8$ cycles on the simulated many-core processor with an SER $\lambda = 6.87 \cdot 10^{-8} \frac{1}{\text{cycle}}$.

Figure 7.12 shows that the fault-tolerance increases with a larger number of replicas N and spares S. In most cases, the reliability and availability of the tasks protected by our software fault-tolerance mechanism are higher than the reliability and availability of the unprotected task T_4. However, for high SERs, this effect is inverse. The reason is that for such high SERs the combined probability of a failure of both voters or a failure of more than M_i replicas is larger than the probability of a failure of a single task. For lower and more realistic SERs, the measurements reveal that the software fault tolerance mechanism

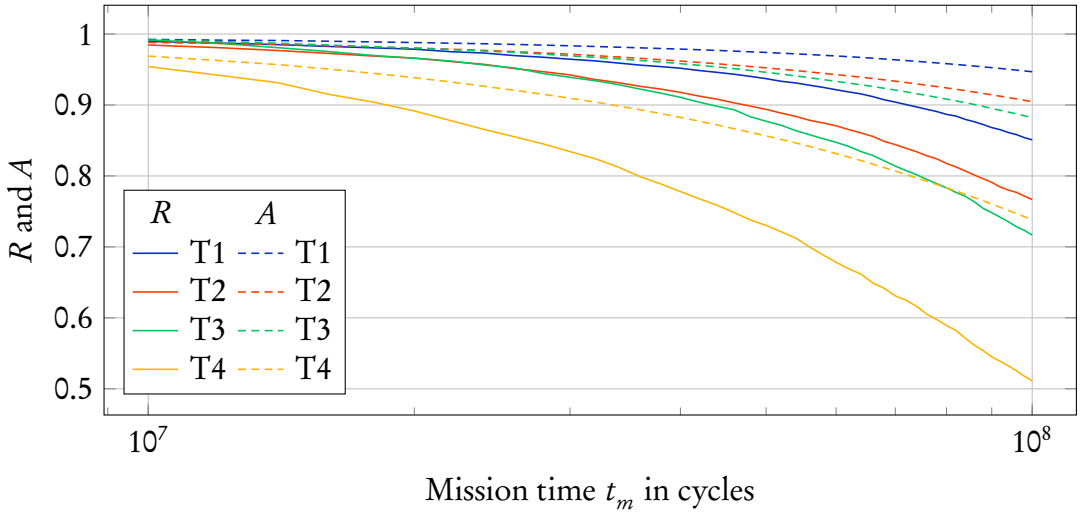


Figure 7.11: The reliability R and availability A of each task T_i after various mission times t_m on the simulated many-core processor with a constant SER $\lambda = 1.1 \cdot 10^{-7} \frac{1}{\text{cycles}}$.

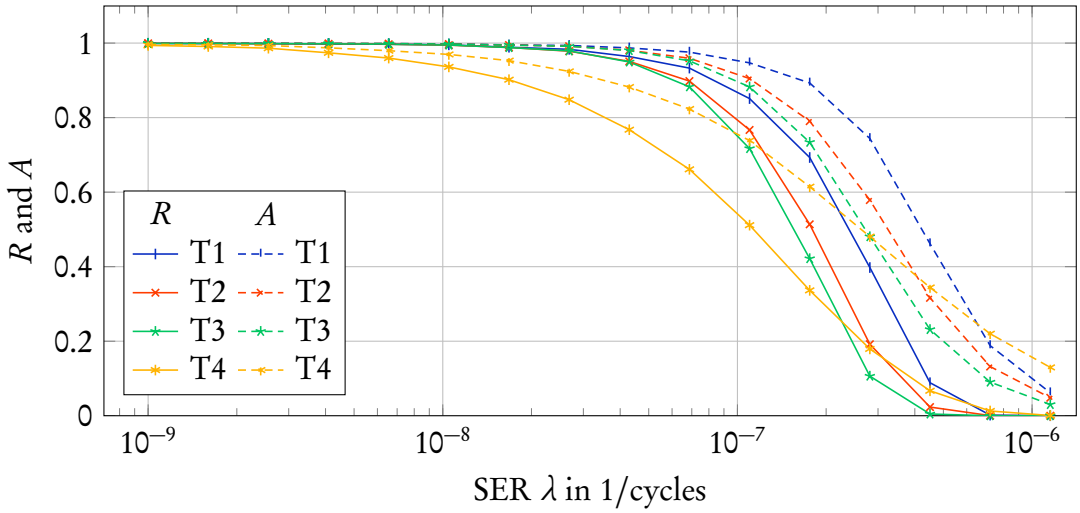


Figure 7.12: The reliability R availability A of each task T_i after the constant mission time $t_m = 10^8$ cycles on the simulated many-core processor with various SERs λ.

is able to increase the reliability of a task by a factor of up to 2.22 (with $N = 5$ replicas and $S = 2$ spares and an SER $\lambda = 2,81 \cdot 10^{-7} \frac{1}{\text{cycles}}$).

7.3.3 Overhead Evaluation

The fault-tolerance increment achieved by the proposed software fault-tolerance mechanism takes its toll on the system's resources. In this section, we present the overhead in terms of system load that is introduced by the fault-tolerance mechanism.

We define the system load as the percentage of time in which the idle task is not running, i. e., the time in which each core is booting or executing tasks and OS kernel functions, divided by the total elapsed runtime. Note that the maximum load of the simulated many-core processor with 10 cores is 10, since the maximum load of each core is 1. We employ the tracing facility of FreeRTOS and the simulated many-core processor in order to determine the system load of the mixed-critical task set example. The following values represent the average system load after 10,000 runs of the same experiment.

Under an SER $\lambda = 10^{-9} \frac{1}{\text{cycles}}$, where only few repairs are executed, the average system load of the unprotected task T_4 is 0.089. When protected by the software fault-tolerance mechanism, each task is replicated on multiple cores. Thus, the load of task T_1 with $N_1 = 5$ replicas and $S_1 = 2$ spares is 0.475, the load of task T_2 with $N_2 = 3$ replicas and $S_2 = 1$ spares is 0.287, and the load of task T_3 with $N_3 = 5$ replicas no spares ($S_3 = 0$) is 0.226. Note that task T_3 has a longer period than the other tasks, in fact $P_3 = 2P_1 = 2P_2$, thus its contribution to the system load is lower. For each task, the software fault-tolerance mechanism instantiates two voters. The load of both voters V_1^1 and V_1^2 for task T_1 is 0.118, the load of both voters for task T_2 is 0.0813, and the load of both voters for task T_3 is 0.0451. Again, the contribution of both voters for task T_3 to the system load is lower than the other tasks due to the longer period of task T_3. Additionally, the software fault-tolerance mechanism requires an SDS and a CDS on each core, which add a load of 0.0284 and 0.0641 to the system, respectively. Considering the additional load of the SDS and the CDS as shared between all three protected tasks, the load of task T_1 is 7 times higher and the load of task T_2 is 4.5 times higher than the load of the unprotected task T_4. For a fair comparison, we assume task T_3 runs with the same period as the remaining tasks and we double its load. As a result, the load of task T_3 is 6.8 times higher, than the load of the unprotected task T_4.

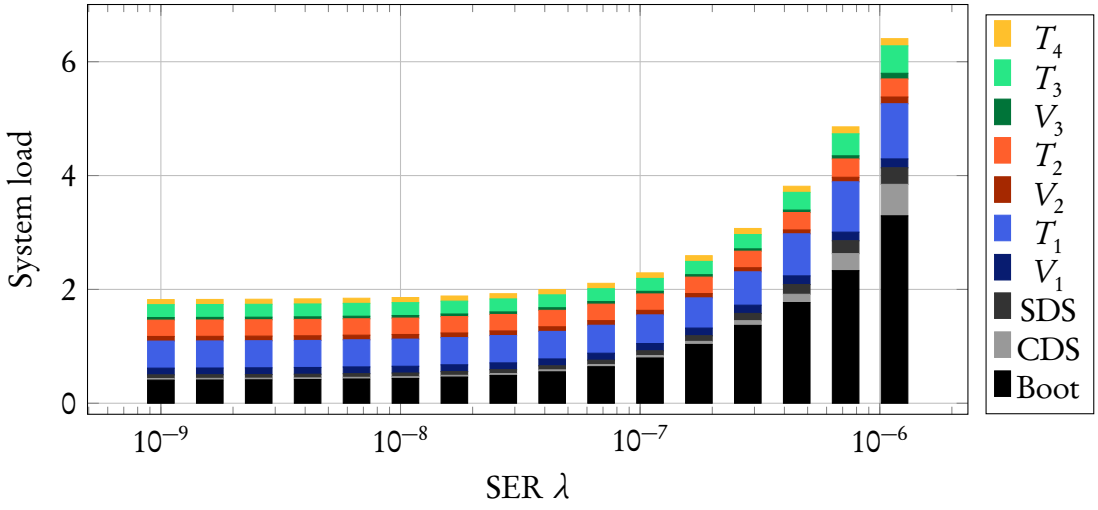


Figure 7.13: Average measured system load of the mixed-critical task set example per SER λ.

Figure 7.13 on page 129 plots the measured average system load per SER λ and the portion each task contributes to the system load. The values show that for a higher SER, the system load increases since more repairs are necessary.

The measured load has to be considered carefully, since the OS and its tracing facility are affected by soft errors, too. For example, the name of the task that is currently scheduled might be modified or the tracing routine stops working correctly due to soft errors. In both cases, the load measurement is disturbed. As a result, the standard deviation of the measurement increases especially for high SERs. For example, the standard deviation of the load of task T_1 is 0.0048 for an SER $\lambda = 10^{-9}$ but increases to 0.0879 for an SER $\lambda = 1.1 \cdot 10^{-7}$ and up to 0,306 for an SER $\lambda = 1.1 \cdot 10^{-6}$.

7.4 Comparison of Theory and Measurement

In this section, we evaluate our fault-tolerance analysis and the underlying DTMC model of our software fault-tolerance mechanism by comparing the measured reliability and availability with the results of the analysis. In order to compare the theoretical and the measured results, all parameters of the model have to be known. Therefore, we first calibrate our model and determine the vulnerability factors of the software components. Next, we present the fault-tolerance analysis results for the same mission times and SERs as in the measurement. Furthermore, we remove the repair procedure from the model in order to determine its influence on the reliability and investigate the scalability of the model-based analysis. Finally, we evaluate the error between the fault-tolerance analysis and the measured results.

7.4.1 Calibration

The PRISM model and the underlying DTMC model operate with software component failure rates instead of the hardware's SER. As mentioned in Section 2.2.5 on page 31, the failure rate $\Lambda = v\lambda$ of any software component depends on the vulnerability factor v of the component and the SER λ. The vulnerability factors of the software components, namely the vulnerability factors of the task replica v_T, the spare $v_{\tilde{T}}$, the voter v_V as well as the vulnerability factor of the voter for critical failures caused by the residual SPOFs v_C are unknown in general.

One way to determine these vulnerability factors are exhaustive fault injection campaigns. In such a campaign, every single bit at every possible cycle is flipped and each time it is checked whether the software component still works as expected. Consider for example task T_4 with period $P_4 = 10\,\text{ms}$, which is equivalent to $100,000$ cycles assuming a clock frequency of 100 MHz. Together with the OS kernel that also affects the task's correct execution, the size of task T_4 in the core-local memory is 52 kB. To determine the vulnerability factor of task T_4, an exhaustive fault injection campaign requires $4.2 \cdot 10^{10}$ experiments. Note that there exist methods to prune the search space by combining faults with the same effect [SHD+15]. However, the vulnerability factor has to consider failures caused by combinations of faults as well, which leads to a combinatorial explosion of the search space even with pruning.

Minimizing Error Approach

Therefore, we take another approach and leverage the fact that the measured results are already known and include the vulnerability factors. We compute the absolute reliability error as the difference between the reliability derived by the fault-tolerance analysis and the measured reliability. We sum up the squared absolute reliability error for all 16 SERs that were used in the previous experiments. Finally, we search the vulnerability factors that yield the minimum squared error sum between the reliability of the fault-tolerance analysis and the measured reliability.

To find the minimum factors, we use the *Nelder-Mead algorithm* [NM65], since it does not need to know the derivative of the n-dimension function of which it searches the minimum. Note that the Nelder–Mead algorithm is also known as Downhill Simplex method. It calculates the function values of a simplex, the geometrical object connecting $n+1$ points, e. g., a triangle for $n=2$ dimensions. The function values are sorted. The point of the simplex that yields the worst function value is iteratively replaced by *reflection* at the centroid of the remaining points. If the function value of the reflected point is better than the best point of the simplex, the simplex is *expanded* in the respective direction. If the function value of the reflected point is not better than any of the remaining points, the reflected point is *contracted* closer to the centroid. If the contracted point does not yield a better function value than any of the remaining points, all points of the simplex are *reduced* around the centroid. Hence, the simplex moves in the direction of the (local) minimum and finally contract around it.

In the following, we use the implementation of the Nelder–Mead algorithm provided by Perl's `Math::Amoeba` module version 0.05. The maximum number of iterations is set to 100 but was never reached to determine the vulnerability factors with a convergence tolerance of 10^{-6}.

To determine the vulnerability factor of a task replica v_T, we leverage the fact that the task T_4 is executed without protection on a separate core $C_{2,5}$. The Nelder-Mead algorithm finds the minimum square error sum between the measured reliability of task T_4 and the estimate of the model with the task vulnerability factor $v_T = 0.061$. Note that the reliability of an unprotected execution is not derived by the DTMC model but calculated from Equation (6.5) on page 98 and Equation (6.6) on page 99. As mentioned in Section 7.3.1 on page 120, all tasks execute the same algorithm and are implemented the same way. Hence, the task vulnerability factor v_T also holds for T_1, T_2, and T_3.

Since spares are not executed but only reside in the core-local memory until activated by the voter, they are only affected by faults in the core-local memory but do not fail due to CPU faults. As stated in Section 7.1.2 on page 111, 90% of faults are injected into the core-local memory while 10% of faults affect the CPU. Hence, we conclude that the spare vulnerability factor $v_{\tilde{T}} = 0.9 v_T = 0.0549$. The spare vulnerability factor $v_{\tilde{T}}$ is valid for each task $T_i \in \mathcal{T}$ for the same reason as for the common task vulnerability factor.

In order to derive the vulnerability factor of the voter v_V and the vulnerability factor of critical voter failures caused by the residual SPOFs v_C, we use the Nelder-Mead algorithm to find the minimum error between the measured reliability and availability of each task T_1, T_2, and T_3 separately. Note that T_4 is executed without protection by a voter. An overview

of the vulnerability factors of each task as determined by our calibration approach is given in Table 7.2.

Task	v_T	$v_{\tilde{T}}$	v_V	v_C
T_1	0.06094	0.05484	0.4184	$7.487 \cdot 10^{-4}$
T_2	0.06094	0.05484	0.5609	$6.190 \cdot 10^{-4}$
T_3	0.06094	0.05484	0.7031	$4.881 \cdot 10^{-5}$
T_4	0.06094	–	–	–

Table 7.2: Overview of the vulnerability factors of a task replica v_T, spare $v_{\tilde{T}}$, voter v_V, and the vulnerability factor of critical voter failures v_C caused by the residual SPOFs of each task $T_i \in \mathcal{T}$.

Discussion of vulnerability factors

The results show that the vulnerability of the voter to critical failures caused by faults in the residual SPOFs is more than 50 times smaller than the vulnerability of the voter to silent failures, as expected in Section 5.2.3 on page 72.

However, in contrast to a task replica that is only vulnerable to 6.1% of faults, voter failures are caused by at least 41.8% of faults. One reason of this high voter vulnerability factor is the repair procedure contained in each voter, which is subject to faults as well but not checked by the other voter and only repaired if the voting procedure fails, too. While the DTMC considers failures during the repair process of a voter or task replica, it does not account for the fact that the repair procedure in the voter which triggers the repair process can fail. By minimizing the difference between the fault-tolerance analysis results and measured values, we account for this fact to the voter's vulnerability factor.

Table 7.2 reveals that the voters of the different tasks do not have the same vulnerability factor. This unexpected outcome is explained as follows: The first voter of each task is mapped to core $C_{1,1}$, as shown in Figure 5.4 on page 82. Additionally, all first voters share their implementation of the voting procedure. The same holds for the second voter of each task. Both voters check each other and start the repair of a failed voter at the hyperperiod. As a result, the voter V_1 of task T_1, which has the highest priority and is executed first, has the lowest probability of faults in its code and data. The voter V_3 of task T_3, which is executed last, has the highest chance to be affected by the accumulated faults. Additionally, all faults that cause previous voters to get stuck and occupy the CPU cause a failure of voter V_3 due to its lower priority. Note that an execution time monitor as part of the OS kernel can prevent such failures but has not been implemented in our experiments, since we focus on our software fault-tolerance mechanism.

7.4.2 Analyzed Fault-Tolerance

We employ PRISM's *hybrid* probabilistic model checking engine to analyze the PRISM model of our software fault-tolerance mechanism and to calculate the resulting reliability R and availability A. The hybrid model checking engine provides the best compromise

between runtime and memory space [PRI15]. To increase the precision of the returned values, we set the termination epsilon of the iterative numerical method of the model checking engine to 10^{-32} using the `-epsilon` switch.

To calculate the reliability R and availability A of the unprotected task T_4, we implement Equation (6.5) and Equation (6.6) in a Perl script and use the `Math::BigFloat` module to achieve arbitrary precision.

Fault-Tolerance over Mission Time

Figure 7.14 on the next page plots the results of the four tasks of the mixed-critical task set for a varying mission time t_m. The SER λ is constantly set to $1.1 \cdot 10^{-7} \frac{1}{\text{cycles}}$. Based on this SER, the vulnerability factors presented in Table 7.2 on page 132 are used to calculate the failure rate Λ of each software component.

Figure 7.14 shows that a larger number of task replicas and spares results in a higher reliability and availability after the same mission time and under the same SER. The results also indicate that for an increasing mission time, the fault-tolerance of the system decreases. The reason of this behavior is that the system does not recover from a detected or undetected system failure. These system failures occur with a low probability. However, they cause an absorbing state in the underlying DTMC model. Since any realization of the DTMC model eventually enters the absorbing state at some point in time, the reliability and availability of the system are both $R(t_m) = 0$ and $A(t_m) = 0$ for an infinite mission time $t_m \to \infty$.

Fault-Tolerance over SER

Figure 7.15 on the next page plots the resulting reliability R and availability A of the four tasks under different SERs λ. The mission time t_m is constantly set to 10^8 cycles.

Figure 7.15 shows that a higher number of replicas and spares allows to tolerate a higher SER. For example, for a mission time $t_m = 10^8$ cycles and under the SER $\lambda = 10^{-8} \frac{1}{\text{cycles}}$, the reliability of task T_1 with $N_1 = 5$ and $S_1 = 2$ is $R_1 = 99.736\%$ and the reliability of task T_3 with $N_3 = 5$ and $S_3 = 0$ is $R_3 = 99.602\%$. For the same mission time t_m and SER λ, the reliability of the unprotected task T_4 is $R_4 = 93.81\%$.

The results show that for high SERs the unprotected execution results in a higher reliability and availability. This effect is due to the fact that the combined probability of a failure of both voters or more than M_i task replicas gets larger than the probability of the failure of a single task. However, for lower and more realistic SERs, our software fault-tolerance mechanism is able to significantly increase the resulting reliability and availability of a task. Consistent with the measured results in Section 7.3.2 on page 126, we find that the fault tolerance mechanism achieves a reliability increment of a factor of up to 2.20 with $N = 5$ replicas and $S = 2$ spares and an SER $\lambda = 2,81 \cdot 10^{-7} \frac{1}{\text{cycles}}$. A detailed comparison between the theoretical results and the measured fault-tolerance values is given in Section 7.4.5 on page 136.

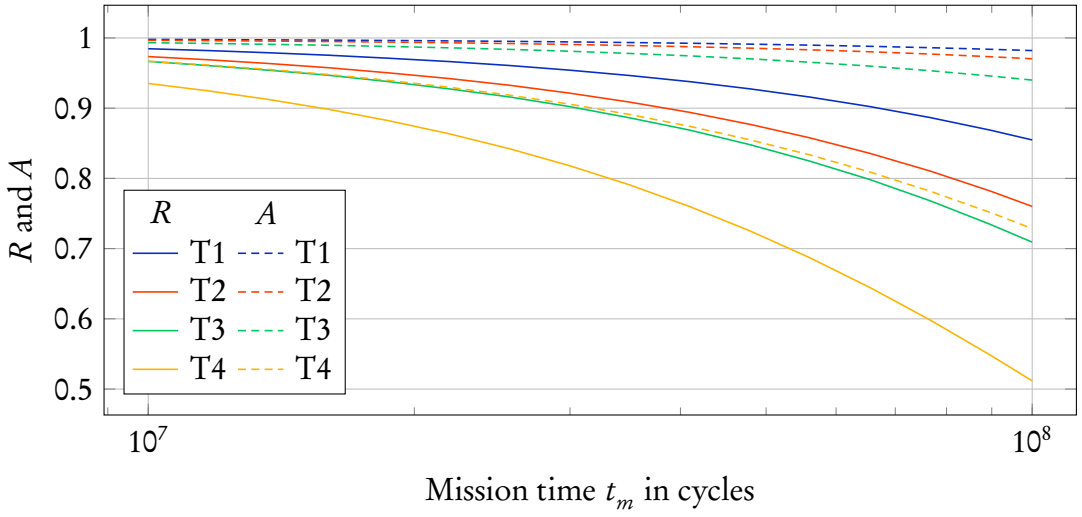


Figure 7.14: The reliability R and availability A computed from the model for a varying mission time t_m. The SER λ is $1.1 \cdot 10^{-7} \frac{1}{\text{cycles}}$.

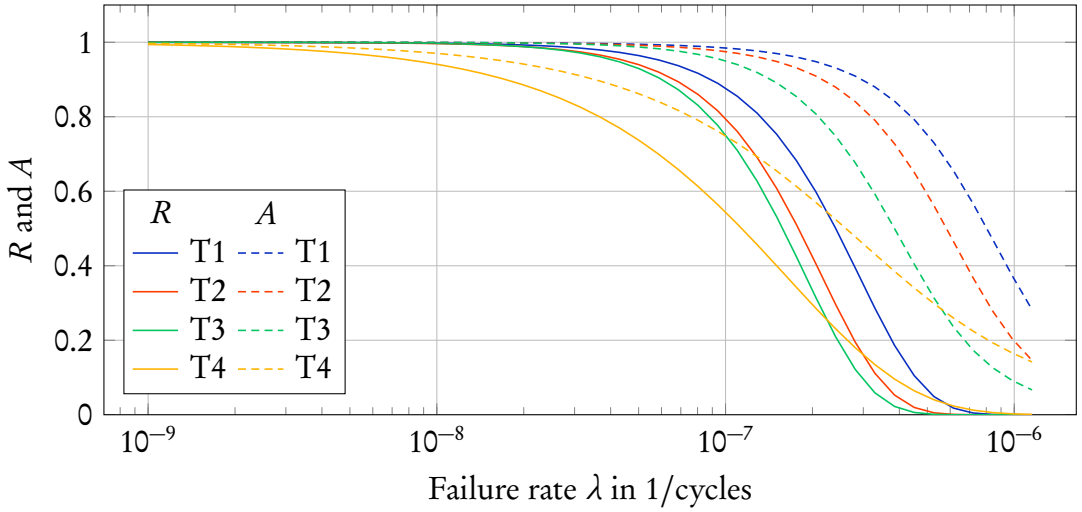


Figure 7.15: The reliability R and availability A computed from the model for different SERs λ. The mission time t_m is 10^8 cycles.

7.4.3 Repair Procedure Influence

In order to analyze the reliability and availability of our software fault-tolerance mechanism *without* the repair procedure, all boot and repair steps are removed from the PRISM model.

The modified PRISM model is analyzed for different SERs λ. The mission time t_m is constantly set to 10^8 cycles. The resulting reliability R and availability A of the tasks are plotted in Figure 7.16 on the facing page. Note that the resulting reliability and availability of the unprotected task are exactly the same as in the Figure 7.15, since the unprotected execution does not contain a repair procedure.

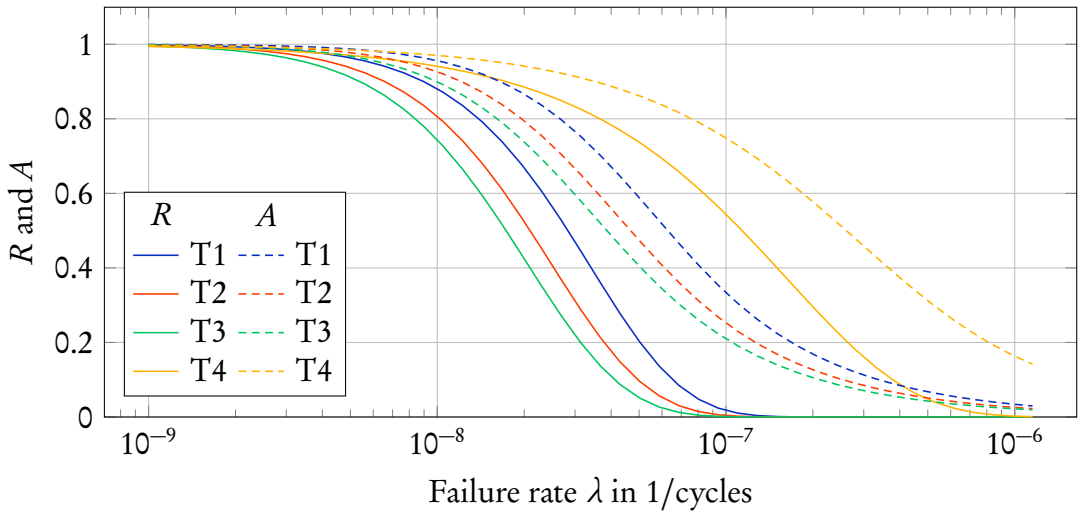


Figure 7.16: The reliability R and availability A for varying SERs λ computed from a model of the software fault-tolerance mechanism without the repair procedure. The mission time t_m is 10^8 cycles.

Compared to the reliability and availability of the original software fault-tolerance mechanism with a repair procedure, the resulting values computed from the PRISM model without a repair procedure are worse for any configuration. In fact, the reliability is only improved for an SER $\lambda < 3{,}5 \cdot 10^{-9}$ for task T_1, an SER $\lambda < 1.87 \cdot 10^{-9}$ for task T_2, and an SER $\lambda < 1.17 \cdot 10^{-9}$ for task T_3.

7.4.4 Model Scalability

An overview of the number of states and transitions of the DTMC model generated from our PRISM model is given in Table 7.3 on the following page. Two configurations of the software fault-tolerance mechanism that are not used in the task set are added to improve the significance of the results, namely $N = 7$ replicas with $S = 0$ and $S = 3$ spares and a period of 10 ms each.

Table 7.3 shows that a higher number of task replicas and spares leads to a larger number of states and transitions contained in the DTMC model. Additionally, the number of states and transitions is significantly lower without the repair procedure. Note that the number of states of the configuration $N_3 = 5$ and $S_3 = 0$ is significantly lower, since the respective task T_3 has twice the period of the other tasks and the other configurations given in Table 7.3. Hence, the number of steps in resulting PRISM model is only half as high as in the other models but the probability of a failure after one period is higher.

We measured the runtime of PRISM's hybrid model checking engine version 4.3 on an Intel Xeon E3-1270v3 processor with 32 GB RAM. The results given in Table 7.3 indicate that the model checking time increases exponentially with the number of task replicas and spares as well as the complexity of the fault-tolerance mechanism itself. The SER has no significant influence on the measured model checking time. However, a longer mission time increases the model checking time, since the length of each realization of the DTMC

N_i	S_i	Rep.	States	Transitions	λ in $\frac{1}{\text{cycles}}$	t_m in cycles	Reliability analysis time in s	Availability analysis time in s
3	1	—	216	488	$1.1 \cdot 10^{-7}$	10^8	0.011	0.006
		✓	10,721	18,159	$1 \cdot 10^{-9}$	10^8	0.432	0.819
					$1.1 \cdot 10^{-7}$	10^8	0.464	0.756
					$1.1 \cdot 10^{-7}$	10^9	2.895	7.425
5	0	—	162	393	$1.1 \cdot 10^{-7}$	10^8	0.005	0.003
		✓	3,495	7,036	$1 \cdot 10^{-9}$	10^8	0.072	0.132
					$1.1 \cdot 10^{-7}$	10^8	0.07	0.1
					$1.1 \cdot 10^{-7}$	10^9	0.401	0.921
5	2	—	486	1,341	$1.1 \cdot 10^{-7}$	10^8	0.014	0.018
		✓	82,392	156,050	$1 \cdot 10^{-9}$	10^8	4.795	23.48
					$1.1 \cdot 10^{-7}$	10^8	4.084	21.43
					$1.1 \cdot 10^{-7}$	10^9	18.31	71.26
7	0	—	216	596	$1.1 \cdot 10^{-7}$	10^8	0.009	0.008
		✓	6,837,504	15,555,064	$1 \cdot 10^{-9}$	10^8	772.7	2,155
					$1.1 \cdot 10^{-7}$	10^8	789.0	2,138
					$1.1 \cdot 10^{-7}$	10^9	6,106	15,031
7	3	—	864	2,816	$1.1 \cdot 10^{-7}$	10^8	0.021	0.022
		✓	11,530,898	29,377,772	$1 \cdot 10^{-9}$	10^8	2,092	4,941
					$1.1 \cdot 10^{-7}$	10^8	2,249	4,935
					$1.1 \cdot 10^{-7}$	10^9	15,288	39,222

Table 7.3: The size of the DTMC model generated by PRISM's single-threaded hybrid model checking engine and the time to analyze the model measured on an Intel Xeon E3-1270v3 processor.

model increases. Compared to 10,000 repetitions of a simulation, where each run takes 53.7 min in average, the model-based fault-tolerance analysis of all tasks is faster by a factor of $1.2 \cdot 10^6$.

7.4.5 Model Precision

Finally, we evaluate the precision of the model-based analysis of the software fault-tolerance mechanism. Therefore, we compare the results obtained from the DTMC model with the measured values.

The fault-tolerance analysis determined that task T_1 with $N = 5$ replicas and $S = 2$ spares has a reliability $R = 85.48\%$ and availability $A = 98.20\%$ under a SER $\lambda = 1.1 \cdot 10^{-7} \frac{1}{\text{cycles}}$ and after a mission time t_m of 10^8cycles. For the same SER and mission time, we measured

the reliability $R = 85.11\%$ and availability $A = 94.69\%$ of task T_1 after 10,000 runs on our simulated many-core processor. Thus, the absolute error between the estimated and the measured value is 0.367 percentage points for the reliability R and 3.51 percentage points for the availability A.

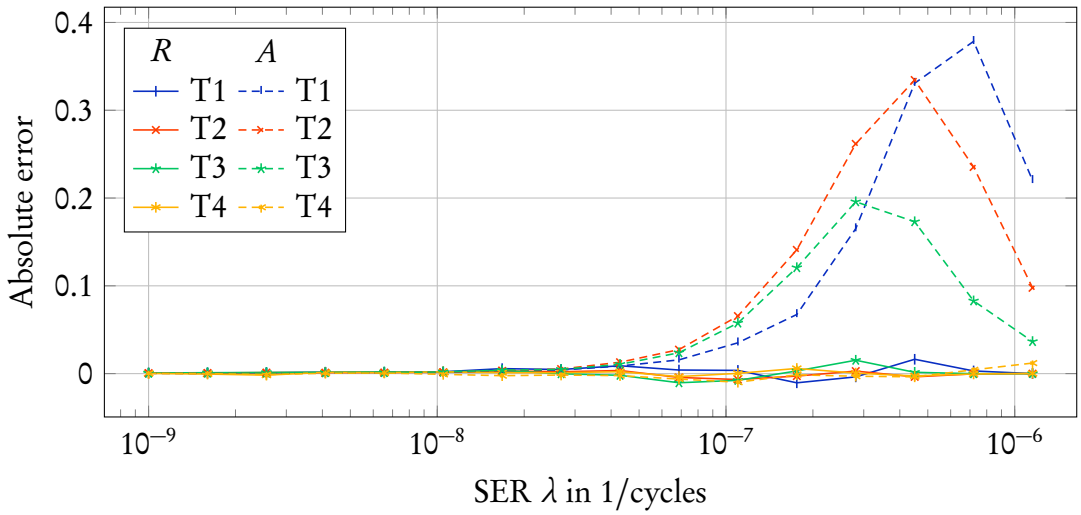


Figure 7.17: The absolute error between the theoretical estimate of the reliability R and the availability A computed from the DTMC model and the measured reliability and availability values.

Figure 7.17 plots the absolute error of the theoretical and measured reliability and availability. The absolute errors of the reliability are below 1.63 percentage points. However, our fault-tolerance analysis overestimates the resulting availability with the selected vulnerability factors. The absolute errors of the availability is up to 37.85 percentage points. This reveals that the vulnerability factors are chosen by minimizing the reliability error.

The comparatively large absolute error of the availability is caused by the fact that the PRISM model of our software fault-tolerance mechanism considers each task separately. Therefore, it does not consider the effects when replicas from different tasks are mapped to the same core. For example, if two task replicas on the same core fail, the voter resets this core and the third, potentially fault-free replica is terminated as well. Similarly, we manually mapped the voters of all tasks to the same core and let them share the voting procedure. As it turns out, this is a suboptimal design decision since the independence between the voters and hence their fault-tolerance is sacrificed to an efficient implementation. However, modeling the complete system with all tasks in a single DTMC model would significantly increase the number of states and transitions and thus the model checking time.

7.5 Summary

In this chapter, we have employed a many-core processor simulator to evaluate the approaches presented in the previous chapters. The configuration of the simulated processor

follows the common design principles of COTS consumer-grade many-core processors as defined in Section 2.2 on page 19. The cycle-accurate and bit-accurate (CABA) simulator includes a fault injection mechanism to emulate faults in the CPU, core-local memory, and the NoC.

In order to evaluate the dynamic GS communication approach presented in Chapter 4, we have measured the transfer latencies of three traffic patterns (latency, throughput, and random) with different packet injection rates. When enforcing the limited packet injection rate, the measured results provide evidence that our approach indeed guarantees an upper bound of the WCTL. With the latency traffic pattern, we have compared the average link load generated by the packet injection rate that was calculated to guarantee the WCTL with the average link load resulting from a packet injection rate that was deemed sufficient to achieve a bounded MMTL. The results show that our dynamic GS communication approach is pessimistic by a factor of 2.77 in asynchronous NoCs and 2.08 in synchronous NoCs for the latency traffic pattern.

Next, we have experimentally evaluated our software-based hardware fault-tolerance mechanism with an implementation of the mixed-critical task set known from the examples of the previous chapters. The results show that tasks protected by the fault-tolerance mechanism have a higher reliability and availability than the unprotected tasks except for very high (and unrealistic) SERs. For $N = 5$ replicas and $S = 2$ spares our mechanism is able to increase the reliability of a task by a factor of up to 2.22. As expected, this increase takes its toll on the system load, which is 7 times higher compared to an unprotected execution of the same algorithm. Considering the rising number of cores in a many-core processor and the increasing SER of the hardware, we argue that this overhead becomes less and less important when a certain fault-tolerance level has to be ensured.

In order to calibrate parameters and determine the vulnerability factors required by our fault-tolerance framework and analysis, respectively, we have minimized the error between results of the underlying PRISM model and the measured reliability value. The fault-tolerance analysis is used to derive the reliability and availability of the mixed-critical task set example without simulation. We have shown that if the repair mechanism is removed from our software fault-tolerance mechanism and the PRISM model, respectively, the fault-tolerance level decreases and is only superior to the unprotected execution for low (but more realistic) SERs. The model checking time increases significantly with an increasing number of replicas and spares, with a longer mission time, and when the repair mechanism is included. However, compared to the 10,000 repetitions of the CABA simulation, the theoretical fault-tolerance analysis is still faster by 6 orders of magnitude. Finally, we have compared the absolute error between the fault-tolerance analysis and the measured reliability and availability. Due to the chosen calibration process, the error of the reliability values is below 1.63 percentage points. However, the error of the availability values is up to 37.85 percentage points. This is caused by the fact that the PRISM model considers each task separately and the manual mapping is suboptimal. Nevertheless, the model-based analysis and the measurements both provide evidence that our software fault-tolerance mechanism is able to increase fault-tolerance of a task compared to an unprotected execution on a many-core processor.

8 Conclusion

In this chapter, we summarize the results of this thesis and discuss them with respect to the research questions defined in the introduction. Finally, we outline potential future research directions.

8.1 Summary and Discussion of Results

In this thesis, we have presented a software-based hardware fault-tolerance framework for mixed-critical task sets on consumer-grade commercial off-the-shelf (COTS) many-core processors. The proposed framework provides interfaces for all roles involved in designing a safety-critical real-time application, such as hardware experts, safety analysts, real-time specialists, and software developers. Based on the requirements and properties specified through these interfaces, the framework selects the parameters of our software fault-tolerance mechanism for each task such that the task's fault-tolerance requirement is achieved with the minimum amount of resources, i. e., cores. In order to perform this parameter selection, the fault-tolerance level in terms of reliability and availability of each task must be known a priori. To solve this problem, we have proposed a fault-tolerance analysis based on a discrete time Markov chain (DTMC) model of our software fault-tolerance mechanism. The DTMC model operates on the failure rates of the software components instead of the soft error rate (SER) of the hardware. Therefore, we have calibrated the model and the respective vulnerability factors based on the measurements we have conducted on our cycle-accurate and bit-accurate (CABA) many-core processor simulator. Hence, we have answered the question of how to determine the fault-tolerance level achieved by a specific adaptation of the software fault-tolerance mechanism at design time.

The proposed software fault-tolerance mechanism is able to increase the reliability and availability of a task using only software means. In order to leverage the spatial redundancy of many-core processors, our mechanism is based on the well-known N modular redundancy (NMR) principle. However, without a reliable majority voter, redundant and parallel execution of the same task on separate cores is not beneficial. We have solved this problem without any hardware modification using two fail-silent voters that check and repair each other. To achieve the fail-silent behavior, we have extended the encoded voting procedure proposed by Ulbrich et al. [UHK+12] to support more than three replicas. The encoded voting procedure compares ANBD-encoded results without decoding them, such that any error in the voting procedure or the replica results is detected before forwarding the tallied result. In contrast to Ulbrich et al. [UHK+12], our

fault-tolerance mechanism provides a decoded result that can be used by external devices without further calculations. However, this includes some residual single point of failures (SPOFs) which we have discussed and evaluated in this thesis.

In order to further increase a task's fault tolerance level without additional runtime overhead, we have added a repair procedure and spare replicas to our software fault-tolerance mechanism. When a voter detects a failed task replica, the repair procedure copies the state of a fault-free task replica to a spare replica. Afterwards, the spare is activated and replaces the failed replica. Thus, our fault-tolerance mechanism supports state-full tasks and is able to operate with the original number of replicas as soon as possible. In contrast to most related fault-tolerance mechanisms, we have explicitly considered failures in all software components, including the voter and the repair procedure itself as well as the underlying OS kernel and its scheduler. In case one of both voters, a component of the repair procedure, or the OS kernel has failed, the respective core is restarted and reloads its core-local memory from an external reliable memory, e. g., an ECC-protected flash memory. Hence, the proposed software fault-tolerance mechanism answers the question of how to design a fault-tolerance mechanism only from software components that potentially fail.

We have investigated the combination of many-core and real-time OSs in order to comply with the real-time requirements of the application. One key design principle of many-core processors is a core-local memory and an inter-core communication fabric such as a Network-on-Chip (NoC). Therefore, we have proposed to manage each core with a core-local kernel that provides a priority-based preemptive real-time scheduler as well as a distributed message passing service (DMPS).

The DMPS requires an inter-core communication with a guaranteed latency and bandwidth. However, most consumer-grade many-core processors do not provide a NoC that provides latency and bandwidth guarantees in hardware. We have solved this problem in software with our limited packet injection rate approach. The main idea of this approach is that each core has to wait at least the duration of the worst-case transfer latency (WCTL) between sending two consecutive packets. In contrast to related work such as network calculus or recursive calculus, our limited packet injection rate approach also provides guarantees for dynamic traffic patterns that are unknown at design time. Such traffic patterns are caused by repair processes that occur at unpredictable points in time as a reaction to task replica failures. With the help of our CABA many-core processor simulator, we have shown that our derivation of the packet injection rate indeed guarantees an upper bound of the WCTL. While we have been unable to find a traffic pattern where the WCTL is actually reached, we have determined that for the latency traffic pattern, our approach is pessimistic by a factor of 2.77 in asynchronous NoCs and a factor of 2.08 in synchronous NoCs. Together with our fault-tolerance mechanism, the proposed limited packet injection rate approach provides an answer to the question of how to increase the system's fault-tolerance level while guaranteeing the real-time constraints without modifying the hardware.

In this thesis, we have evaluated the software fault-tolerance framework with an example task set containing four tasks with different real-time constraints and fault-tolerance requirements, which are derived from a realistic application. We have shown how to select the voter offset and advance the deadlines of the task replicas such that well-established

scheduling policies are able to guarantee the application's real-time requirements. The example task set has been executed on our many-core processor simulator. This simulator is able to inject faults in the core-local memories, the CPU registers, and the NoC buffers. We have measured that our software fault-tolerance mechanism is able to increase the reliability of a task with five replicas and two spares by a factor of up to 2.22 compared to an unprotected execution of the same task. As expected, the overhead in terms of system load is 7 times higher in this configuration compared to the unprotected execution of the task. Considering the rising number of cores and the increasing SER of many-core processors, we argue that this overhead becomes less and less important when certain fault-tolerance requirements have to be ensured. With these results we have answered the main research question of this thesis. We have shown how to achieve the fault-tolerance requirements of mixed-critical task sets with real-time constraints on consumer-grade many-core processor using only software means and we have quantified the resulting reliability and availability as well as the generated overhead.

8.2 Future Research Opportunities

The software fault-tolerance framework presented in this thesis eases the implementation of safety-critical applications with real-time requirements on consumer-grade many-core processors, which provide the necessary computational performance at attractive costs. During our research, we have identified the following questions that are worth further investigation.

Fault Hypothesis

While the fault injection mechanism of our many-core processor simulator is able to inject faults in separate components at the same time, it is limited to single bit flips. Chatterjee et al. [CNM+14] show that with a decreasing feature size the amount of multi-cell upsets (MCUs) in SRAMs increases. Although we assume only minor effects on the vulnerability factors, adding such multi bit flips to our fault injection mechanism would increase its accuracy.

Furthermore, our fault hypothesis considers only transient hardware faults. Although the permanent fault rate is significantly lower [Bau05], it would be interesting to include such faults to the fault hypothesis and extend the fault-tolerance mechanism accordingly.

Fault-Tolerance Mechanism

The voters of the presented software fault-tolerance mechanism perform an exact majority voting scheme. Depending on the system, other voting schemes might be beneficial. For example, the majority scheme can be replaced by a 2-out-of-5 or 4-out-of-5 scheme given a configuration with five replicas. Another example is the median voter, which belongs to the group of inexact voters.

Our fault-tolerance mechanism uses exactly two fail-silent voters that check and compare each other. The number of voters could be increased in order to evaluate its benefit on the fault-tolerance level and its overhead in terms of system load. A setup with more than two

voters requires to establish a hierarchy amongst the voters to ensure that only one voter forwards the final result. Similar to this idea, spare voters could be introduced, since the current repair procedure for voters is based on the inter-core reset mechanism only.

In this thesis, each task replica and spare is statically assigned to a specific core. In case of a replica failure, the repair procedure chooses the first available spare and the closest fault-free task replica. Instead, the repair procedure could select the spare in an intelligent way, e. g., depending on the accumulated number of failures or the current load of a core. This is especially useful if permanent faults are considered, too, and cores with permanently failed hardware components need to be avoided. Furthermore, the static assignment of spares could be completely abandoned and instead, the new core on which the replacement of a failed task replica is instantiated could be determined dynamically. In this case, the repair procedure would adopt the behavior of a load balancer and is able to reduce the thermal stress between the cores. Eventually, we imagine task replicas are relocated proactively before a failure occurs.

Complex Applications

In this thesis, we consider typical automotive applications that consist of periodically executed tasks with comparatively small results. Applications from other domains might include long-running multi-threaded processes that generate large results. In order to support such kind of applications, adjustments of our software fault-tolerance framework might increase its efficiency. The well-known concepts of checkpoints and checksums indicate a potential way to deal with long-running processes and large results. Instead of replicating tasks, it might be necessary to replicate threads.

Real Hardware

All experiments of this thesis are conducted on a CABA many-core processor simulator. This simulation-based approach allows to inject faults without modifying the application software or the OS. Furthermore, multiple simulators can be executed in parallel to generate a large number of results. However, the CABA simulation is computationally expensive and thus limited to small applications with low complexity. For this reason, it would be interesting to evaluate the scalability our software-based fault-tolerance mechanism on a real many-core processor with carefully implemented fault injection in software or a field-programmable gate array (FPGA) implementation of a many-core processor with fault injection extensions.

Fault-Tolerance Framework

The presented software fault-tolerance framework adapts the number of replicas and spares based on the result of our fault-tolerance analysis and the underlying DTMC model. This model of the fault-tolerance mechanism considers only a single task and currently does not include side effects between replicas or voters of different tasks that are mapped to the same core. If such side effects would be included in the model, the precision of its results is likely to increase. However, the complexity of the model and hence the number of states in the DTMC model and the analysis time will increase as well. Other modeling methodologies

such as stochastic activity networks (SANs) are computational less intensive to analyze but are limited to simulation-based solvers [MAL+15].

In this thesis, we employ the partitioned fixed-priority preemptive scheduling policy for which a feasibility test exists. Our limited packet injection rate approach guarantees an upper bound of the communication delay. Both of these properties ease the search of a feasible task-to-core mapping. However, additional constraints are introduced by the fault-tolerance mechanism that requires a subset of task to be mapped to mutually exclusive cores. For complex applications with many communicating tasks and a high system utilization, the mapping problem becomes difficult to be solved manually. Therefore, we suggest to adapt and include existing task mapping solutions to our framework.

Our framework should not be limited to the presented software fault-tolerance mechanism. Instead, we see great potential in including other fault-tolerance mechanisms to our framework. We envision our framework to be capable of selecting a suitable subset from a catalog of hardware and software mechanism such that the fault-tolerance requirements are achieved in an optimal way.

Bibliography

[ABR+93] Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993. doi:10.1049/sej.1993.0034. Cited on page 13.

[Acc13] Accellera Systems Initiative Inc. Open Core Protocol Specification 3.0, 2013. URL: http://www.accellera.org/images/downloads/standards/ocp/OCP_3.0_Specification.zip [accessed March 28, 2016]. Cited on page 21.

[Ada13] Adapteva, Inc. Epiphany architecture reference, 2013. URL: http://www.adapteva.com/docs/epiphany_arch_ref.pdf [accessed March 28, 2016]. Cited on pages 20, 21, and 104.

[AEF+14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82:1–82:37, 2014. doi:10.1145/2560033. Cited on page 2.

[AFAL07] Luis Almeida, Sebastian Fischmeister, Madhukar Anand, and Insup Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, pages 67–74, 2007. doi:10.1145/1289927.1289942. Cited on page 50.

[AFK05] Joakim Aidemark, Peter Folkesson, and Johan Karlsson. A framework for node-level fault tolerance in distributed real-time systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DNS)*, pages 656–665, 2005. doi:10.1109/DSN.2005.7. Cited on page 50.

[AJEF15] Laure Abdallah, Mathieu Jan, Jérôme Ermont, and Christian Fraboul. Wormhole networks properties and their use for optimizing worst case delay analysis of many-cores. In *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 59–68, 2015. doi:10.1109/SIES.2015.7185041. Cited on pages 28, 42, and 43.

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33,

2004. doi:10.1109/TDSC.2004.2. Cited on pages 1, 15, 16, 17, 18, 19, and 32.

[ARJS07] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 470–481, 2007. doi:10.1145/1250662.1250720. Cited on page 44.

[ARM04] ARM Ltd. AMBA AXI Protocol Specification, 2004. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html [accessed March 28, 2016]. Cited on page 21.

[ASE11] Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 149–158, 2011. doi:10.1145/2039370.2039396. Cited on page 49.

[ASM+12] J.L. Autran, S. Serre, D. Munteanu, S. Martinie, S. Semikh, S. Sauze, S. Uznanski, G. Gasiot, and P. Roche. Real-time soft-error testing of 40nm SRAMs. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.5.1–3C.5.9, 2012. doi:10.1109/IRPS.2012.6241814. Cited on page 111.

[AUT15] AUTOSAR. General specification of basic software modules, Release 4.2.2, 2015. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/standard/AUTOSAR_SWS_BSWGeneral.pdf [accessed March 28, 2016]. Cited on pages 13 and 34.

[Avi71] Algirdas Avižienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions on Computers*, C-20(11):1322–1331, 1971. doi:10.1109/T-C.1971.223134. Cited on page 46.

[BAG+15] Alexander Biewer, Benjamin Andres, Jens Gladigau, Torsten Schaub, and Christian Haubelt. A symbolic system synthesis approach for hard real-time systems based on coordinated SMT-solving. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 357–362, 2015. URL: http://dl.acm.org/citation.cfm?id=2755834 [accessed March 28, 2016]. Cited on page 37.

[Bar10a] Max Baron. The single-chip cloud computer—Intel networks 48 Pentiums on a chip. *Microprocessor Report*, 2010. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf [accessed March 28, 2016]. Cited on pages 21 and 28.

[Bar10b] Richard Barry. *Using the FreeRTOS Real Time Kernel—Standard Edition*. Real Time Engineers Ltd., 2010. Cited on pages 34 and 121.

[Bau05] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, 2005. doi:10.1109/TDMR.2005.853449. Cited on pages 3, 16, 30, and 141.

[BBD+09] Andrew Baumann, Paul Barhamy, Pierre-Evariste Dagandz, Tim Harrisy, Rebecca Isaacsy, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009. doi:10.1145/1629575.1629579. Cited on page 34.

[BCA08] Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 157–169, 2008. doi:10.1109/RTSS.2008.23. Cited on page 33.

[BCM13] Cristiana Bolchini, Matteo Carminati, and Antonio Miele. Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing*, 29(2):159–175, 2013. doi:10.1007/s10836-013-5367-y. Cited on pages 51 and 54.

[BDP96] Alan Burns, Robert I. Davis, and Sasikumar Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the 8th Euromicro Workshopon Real-Time Systems (RTS)*, pages 29–33, 1996. doi:10.1109/EMWRTS.1996.557785. Cited on page 48.

[BFFM12] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 983–987, 2012. doi:10.1109/DATE.2012.6176639. Cited on page 21.

[BFM+03] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto L. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 170–177, 2003. doi:10.1145/951710.951734. Cited on page 44.

[BGH14] Alexander Biewer, Jens Gladigau, and Christian Haubelt. A novel model for system-level decision making with combined ASP and SMT solving. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in*

Europe (DATE), pages 217:1–217:4, 2014. doi:10.7873/DATE.2014.230. Cited on page 37.

[Bir14] Alessandro Birolini. *Reliability Engineering: Theory and Practice*. Springer, 7th edition, 2014. doi:10.1007/978-3-642-39535-2. Cited on pages 5, 16, 17, 18, 29, 30, 43, 88, and 125.

[BIS10] Alan Burns, Leandro Soares Indrusiak, and Zheng Shi. Schedulability analysis for real time on-chip communication with wormhole switching. *International Journal of Embedded and Real-Time Communication Systems*, 1(2):1–22, 2010. doi:10.4018/jertcs.2010040101. Cited on page 41.

[BJ14] Mehrdad Bagheri and Gert Jervan. Fault-tolerant scheduling of mixed-critical applications on multi-processor platforms. In *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 25–32, 2014. doi:10.1109/EUC.2014.13. Cited on page 49.

[BJHV09] Demid Borodin, Ben H.H. Juurlink, Said Hamdioui, and Stamatis Vassiliadis. Instruction-level fault tolerance configurability. *Journal of Signal Processing Systems*, 57(1):89–105, 2009. doi:10.1007/s11265-008-0175-9. Cited on page 47.

[BJV07] Demid Borodin, Ben H.H. Juurlink, and Stamatis Vassiliadis. Instruction-level fault tolerance configurability. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pages 110–117, 2007. doi:10.1109/ICSAMOS.2007.4285741. Cited on page 47.

[Ble11] John Blevins. Enhancing application performance on multicore systems. White paper, 2011. URL: http://mil-embedded.com/articles/enhancing-application-performance-multicore-systems/ [accessed March 28, 2016]. Cited on page 33.

[BM05] Sanjeev Baskiyar and Natarajan Meghanathan. A survey of contemporary real-time operating systems. *Informatica (Slovenia)*, 29(2):233–240, 2005. URL: http://www.informatica.si/index.php/informatica/article/view/36 [accessed March 28, 2016]. Cited on page 33.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1):1–51, 2006. doi:10.1145/1132952.1132953. Cited on pages 23, 26, 27, and 28.

[BMS12] Cristiana Bolchini, Antonio Miele, and Donatella Sciuto. An adaptive approach for online fault management in many-core architectures. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 1429–1432, 2012. doi:10.1109/DATE.2012.6176589. Cited on pages 51 and 54.

[Bor05] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005. doi:10.1109/MM.2005.110. Cited on page 3.

[Bor07] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, pages 746–749, 2007. doi:10.1145/1278480.1278667. Cited on page 1.

[BPG04] Richard Buchmann, Frédéric Pétrot, and Alain Greiner. Fast cycle accurate simulator to simulate event-driven behavior. In *Proceedings of the International Conference on Electrical, Electronic and Computer Engineering (ICEEC)*, pages 35–38, 2004. doi:10.1109/ICEEC.2004.1374374. Cited on page 108.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 12–17, 2009. URL: http://www.usenix.org/events/hotos09/tech/full_papers/baumann/baumann.pdf [accessed March 28, 2016]. Cited on page 34.

[BS05] Tobias Bjerregaard and Jens Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the 11th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 34–43, 2005. doi:10.1109/ASYNC.2005.7. Cited on page 41.

[BSS13] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd Annual International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013. doi:10.1109/DSN.2013.6575308. Cited on page 50.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825. Cited on page 37.

[Bun12] Bundesanstalt für Straßenwesen. Rechtsfolgen zunehmender Fahrzeugautomatisierung. *Berichte der Bundesanstalt für Straßenwesen, Unterreihe "Fahrzeugsicherheit"*, F83:1–2, 2012. URL: http://www.bast.de/DE/Publikationen/Foko/2013-2012/2012-11.html [accessed March 28, 2016]. Cited on page 2.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011. doi:10.1007/978-1-4614-0676-1. Cited on pages 1, 9, and 10.

[BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: Ada 95, real-time Java and real-time POSIX*. Addison-Wesley, 3rd edition, 2001. Cited on pages 1 and 10.

[BWCC+08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 43–57, 2008. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/boyd-wickizer/boyd_wickizer.pdf [accessed March 28, 2016]. Cited on page 35.

[CASM11] Everton Alceu Carara, Gabriel Marchesan Almeida, Gilles Sassatelli, and Fernando Gehm Moraes. Achieving composability in NoC-based MPSoCs through QoS management at software level. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 407–412, 2011. doi:10.1109/DATE.2011.5763071. Cited on page 41.

[CCM14] Everton Alceu Carara, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Differentiated communication services for NoC-based MPSoCs. *IEEE Transactions on Computers*, 63(3):595–608, 2014. doi:10.1109/TC.2012.123. Cited on page 41.

[CM11] Chen-Ling Chou and Radu Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 673–678, 2011. doi:10.1109/DATE.2011.5763113. Cited on page 50.

[CNM+14] I. Chatterjee, B. Narasimham, N.N. Mahatme, B.L. Bhuva, R.A. Reed, R.D. Schrimpf, J.K. Wang, N. Vedula, B. Bartz, and C. Monzel. Impact of technology scaling on SRAM soft error rates. *IEEE Transactions on Nuclear Science*, 61(6):3512–3518, 2014. doi:10.1109/TNS.2014.2365546. Cited on page 141.

[CYM+12] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the 30th International Conference on Computer Design (ICCD)*, pages 94–101, 2012. doi:10.1109/ICCD.2012.6378623. Cited on page 28.

[Das14] Dakshina Dasari. *Timing Analysis of Real-Time Systems Considering the Contention on the Shared Interconnection Network in Multicores*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2014. URL: www.cister.isep.ipp.pt/docs/893 [accessed April 25, 2016]. Cited on page 43.

[DC07] Francis M. David and Roy H. Campbell. Building a self-healing operating system. In *Proceedings of the 3rd International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, pages 3–10, 2007. doi:10.1109/DASC.2007.22. Cited on page 50.

[DCT+13] Onur Derin, Emanuele Cannella, Giuseppe Tuveri, Paolo Meloni, Todor Stefanov, Leandro Fiorin, Luigi Raffo, and Mariagiovanna Sami. A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project. *Microprocessors and Microsystems*, 37(6–7):515–529, 2013. doi:10.1016/j.micpro.2013.07.007. Cited on pages 50 and 54.

[dDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of the High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013. doi:10.1109/HPEC.2013.6670342. Cited on page 20.

[dDAPL14] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 97:1–97:6, 2014. doi:10.7873/DATE.2014.110. Cited on pages 20 and 42.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. doi:10.1007/978-1-4613-8643-8. Cited on page 111.

[DH12] Björn Döbel and Hermann Härtig. Who watches the watchmen? – protecting operating system reliability mechanisms. In *Presented as part of the 8th Conference on Hot Topics in System Dependability (HotDep)*, pages 1–6, 2012. URL: https://www.usenix.org/conference/hotdep12/workshop-program/presentation/d%C3%B6bel [accessed April 25, 2016]. Cited on page 51.

[DH13] Björn Döbel and Hermann Härtig. Where have all the cycles gone? – investigating runtime overheads of os-assisted replication. In *Proceedings of the 2nd Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES)*, pages 2534–2547, 2013. URL: http://os.inf.tu-dresden.de/~doebel/papers/2013-sobres-overhead.pdf [accessed April 25, 2016]. Cited on pages 51 and 54.

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proceedings of the 10th International Conference on Embedded Software (EMSOFT)*, pages 83–92, 2012. doi:10.1145/2380356.2380375. Cited on pages 51 and 54.

[DJPI96] Todd A. DeLong, Barry W. Johnson, and Joseph A. Profeta III. A fault injection technique for VHDL behavioral-level models. *IEEE Design & Test of Computers*, 13(4):24–33, 1996. doi:10.1109/54.544533. Cited on page 111.

[DKV13] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Communication and migration energy aware design space exploration for multicore systems with intermittent faults. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 1631–1636, 2013. doi:10.7873/DATE.2013.331. Cited on page 49.

[DMB06] Giovanni De Micheli and Luca Benini. *Networks on Chips*. Morgan Kaufmann, 2006. doi:10.1016/B978-012370521-1/50000-X. Cited on pages 23, 25, and 26.

[DMH14] Björn Döbel, Robert Muschner, and Hermann Härtig. Resource-aware replication on heterogeneous multicores: Challenges and opportunities. In *Presented at 1st Workshop on Resource Awareness and Adaptivity in Multi-Core Computing(RACING)*, pages 1–6, 2014. URL: http://arxiv.org/abs/1405.2913 [accessed April 25, 2016]. Cited on page 51.

[DNNP14] Dakshina Dasari, Borislav Nikolić, Vincent Nélis, and Stefan M. Petters. NoC contention analysis using a branch-and-prune algorithm. *ACM Transactions on Embedded Computing Systems*, 13(3s):113:1–113:26, 2014. doi:10.1145/2567937. Cited on pages 28, 42, and 43.

[Döb14] Björn Döbel. *Operating System Support for Redundant Multithreading*. PhD thesis, Technischen Universität Dresden, 2014. URL: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-157190 [accessed April 25, 2016]. Cited on pages 48, 51, and 54.

[DSSH03] Paul E. Dodd, Marty R. Shaneyfelt, James R. Schwank, and Gerald L. Hash. Neutron-induced latchup in SRAMs at ground level. In *Proceedings of the 41st International Annual Reliability Physics Symposium*, pages 51–55, 2003. doi:10.1109/RELPHY.2003.1197720. Cited on page 30.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003. Cited on pages 23 and 25.

[Dub13] Elena Dubrova. *Fault-Tolerant Design*. Springer, 2013. doi:10.1007/978-1-4614-2113-9. Cited on pages 17, 18, 19, 28, 31, and 45.

[ED12] Michael Engel and Björn Döbel. The reliable computing base: A paradigm for software-based reliability. In *Presented at the 1st Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES)*, 2012. URL: http://www.danceos.org/sobres/2012/papers/SOBRES-7-Engel.pdf [accessed April 25, 2016]. Cited on page 51.

[EET+14] Mojtaba Ebrahimi, Adrian Evans, Mehdi Baradaran Tahoori, Razi Seyyedi, Enrico Costenaro, and Dan Alexandrescu. Comprehensive analysis of alpha and neutron particle-induced soft errors in an embedded processor at nanoscales. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 1–6, 2014. doi:10.7873/DATE.2014.043. Cited on pages 73, 104, and 112.

[EET+15] Mojtaba Ebrahimi, Adrian Evans, Mehdi Baradaran Tahoori, Enrico Costenaro, Dan Alexandrescu, Vikas Chandra, and Razi Seyyedi. Comprehensive analysis of sequential and combinational soft errors in an embedded processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1586–1599, 2015. doi:10.1109/TCAD.2015.2422845. Cited on pages 73, 104, and 112.

[EIPP08] Petru Eles, Viacheslav Izosimov, Paul Pop, and Zebo Peng. Synthesis of fault-tolerant embedded systems. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 1117–1122, 2008. doi:10.1109/DATE.2008.4484825. Cited on page 49.

[Exp06] Express Logic Inc. ThreadX – the high-performance embedded kernel. User guide version 5.0, 2006. Cited on page 34.

[FA12] Bo Fu and Paul Ampadu. *Error Control for Network-on-Chip Links*. Springer, 2012. doi:10.1007/978-1-4419-9313-7. Cited on page 113.

[FFF09] Thomas Ferrandiz, Fabrice Francès, and Christian Fraboul. A method of computation for worst-case delay analysis on SpaceWire networks. In *Proceedings of the International Symposium on Industrial Embedded Systems (SIES)*, pages 19–27, 2009. doi:10.1109/SIES.2009.5196187. Cited on page 43.

[FFF11] Thomas Ferrandiz, Francès Frances, and Christian Fraboul. Using network calculus to compute end-to-end delays in spacewire networks. *ACM SIGBED Review – Work-in-Progress Session of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 8(3):44–47, 2011. doi:10.1145/2038617.2038627. Cited on page 42.

[FFF12] Thomas Ferrandiz, Francès Frances, and Christian Fraboul. A sensitivity analysis of two worst-case delay computation methods for spacewire networks. In *Proceedings of the 24th Conference on Real-Time Systems (ECRTS)*, pages 47–56, 2012. doi:10.1109/ECRTS.2012.35. Cited on page 43.

[For90] P. Forin. Vital coded microprocessor principles and application for various transit systems. In J.-P. Perrin, editor, *Control, Computers, Communications in Transportation*, IFAC Symposia Series, pages 79–84. Pergamon, 1990. doi:10.1016/B978-0-08-037025-5.50017-7. Cited on page 46.

[GDR05] Kees Goossens, John Dielissen, and Andrei Rădulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005. doi:10.1109/MDT.2005.99. Cited on pages 28 and 41.

[GH10] Kees Goossens and Andreas Hansson. The Æthereal network on chip after ten years: goals, evolution, lessons, and future. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 306–311, 2010. doi:10.1145/1837274.1837353. Cited on page 28.

[GKSS03] Alain Girault, Hamoudi Kalla, Mihaela Sighireanu, and Yves Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *Proceedings of the InternationalConference on Dependable Systems and Networks (DSN)*, pages 159–168, 2003. doi:10.1109/DSN.2003.1209927. Cited on page 49.

[Gon07] Masaki Gondo. Blending asymmetric and symmetric multiprocessing with a single OS on ARM11 MPCore. White paper, 2007. URL: http://www.esol.com/embedded/download/pdf/whitepaper_multicore.pdf [accessed April 30, 2016]. Cited on page 34.

[GPA+11] Dimitris Gizopoulos, Mihalis Psarakis, Sarita V. Adve, Pradeep Ramachandran, Siva Kumar Sastry Hari, Daniel J. Sorin, Albert Meixner, A. Biswas, and Xavier Vera. Architectures for online error detection and recovery in multicore processors. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 533–538, 2011. doi:10.1109/DATE.2011.5763096. Cited on page 43.

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC)*, pages 3–14. IEEE Computer Society, 2001. doi:10.1109/WWC.2001.15. Cited on page 121.

[Gre13] Green Hills Software. INTEGRITY RTOS, 2013. URL: http://www.ghs.com/download/datasheets/INTEGRITY_RTOS.pdf [accessed April 30, 2016]. Cited on page 34.

[GRSRV06] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. doi:10.1007/0-387-32937-4. Cited on pages 3, 43, and 45.

[GSVP03] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83, 2003. doi:10.1109/MM.2003.1261390. Cited on page 50.

[HAZ13] Mohammad A. Haque, Hakan Aydin, and Dakai Zhu. Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms. In *Proceedings of the International Green Computing Conference (IGCC)*, pages 1–11, 2013. doi:10.1109/IGCC.2013.6604518. Cited on page 49.

[HBD+13] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 99:1–99:10, 2013. doi:10.1145/2463209.2488857. Cited on page 45.

[HBD+14] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing(ISORC)*, pages 230–237, 2014. doi:10.1109/ISORC.2014.26. Cited on page 50.

[HBR+11] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 247–256, 2011. doi:10.1145/2039370.2039409. Cited on pages 51 and 54.

[HBZ+14] Jörg Henkel, Lars Bauer, Hongyan Zhang, Semeen Rehman, and Muhammad Shafique. Multi-layer dependability: From microarchitecture to application level. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pages 47:1–47:6, 2014. doi:10.1145/2593069.2596683. Cited on page 45.

[HF13] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. White paper, 2013. URL: http://www.absint.com/aiT_WCET.pdf [accessed April 30, 2016]. Cited on page 11.

[HGR07] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 95859:1–10, 2007. doi:10.1155/2007/95859. Cited on pages 23, 61, and 108.

[HHK03] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003. doi:10.1109/JPROC.2002.805825. Cited on pages 14 and 38.

[HKR+15] Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. Evaluation of diverse compiling for software-fault detection. In

Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE), pages 531–536, 2015. URL: http://dl.acm.org/citation.cfm?id=2755873 [accessed April 30, 2016]. Cited on page 51.

[HLR+09] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture (MICRO)*, pages 122–132, 2009. doi:10.1145/1669112.1669129. Cited on page 44.

[HRIK15] Andrea Höller, Tobias Rauter, Johannes Iber, and Christian Kreiner. Towards dynamic software diversity for resilient redundant embedded systems. In *Proceedings of the 7th International Workshop Software Engineering for Resilient Systems (SERENE)*, pages 16–30, 2015. doi:10.1007/978-3-319-23129-7_2. Cited on pages 51 and 54.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997. doi:10.1109/2.585157. Cited on page 111.

[Hua14] Jia Huang. *Towards an Integrated Framework for Reliability-Aware Embedded System Design on Multiprocessor System-on-Chips*. PhD thesis, Technische Universität München, 2014. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20141104-1178696-0-9 [accessed April 30, 2016]. Cited on pages 51 and 54.

[HUD+14a] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Experiences with software-based soft-error mitigation using AN codes. *Software Quality Journal*, pages 87–113, 2014. doi:10.1007/s11219-014-9260-4. Cited on pages 52 and 68.

[HUD+14b] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. A practitioner's guide to software-based soft-error mitigation using an-codes. In *Proceedings of the 15th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 33–40, 2014. doi:10.1109/HASE.2014.14. Cited on pages 52, 68, and 71.

[IEE12] IEEE Computer Society. 1666-2011 - IEEE standard for standard SystemC language reference manual, 2012. doi:10.1109/IEEESTD.2012.6134619. Cited on page 108.

[IHB15] Leandro Soares Indrusiak, James Harbin, and Alan Burns. Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip. In *Proceedings of the 27th Conference on Real-Time Systems (ECRTS)*, pages 47–56, 2015. doi:10.1109/ECRTS.2015.12. Cited on page 41.

[Int10] International Electrotechnical Commission (IEC). 61508-1: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements, 2010. Cited on pages 3, 17, 34, and 103.

[Int11] International Standard Organization (ISO). ISO 26262-5: Road vehicles – functional safety – Part 5: Product development at the hardware level, 2011. Cited on pages 3, 17, and 103.

[Int12] Intel Crop. The SCC platform overview, revision 0.8, 2012. URL: https://communities.intel.com/docs/DOC-5512 [accessed April 30, 2016]. Cited on page 21.

[Int15] Intel Crop. Intel Xeon processor E7-8890 v3 (45M cache, 2.50 GHz), 2015. URL: http://ark.intel.com/products/84685 [accessed April 30, 2016]. Cited on page 20.

[IO11] Eishi Ibe and Kenichi Osada. *Low Power and Reliable SRAM Memory Cell and Array Design*, chapter Reliable Memory Cell Design for Environmental Radiation-Induced Failures in SRAM, pages 89–124. Springer, 2011. doi:10.1007/978-3-642-19568-6_6. Cited on page 30.

[IPEP08] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 915–920, 2008. doi:10.1109/DATE.2008.4484791. Cited on page 48.

[Joh84] Barry W. Johnson. Fault-tolerant microprocessor-based systems. *IEEE Micro*, 4(6):6–21, 1984. doi:10.1109/MM.1984.291277. Cited on pages 3 and 18.

[KB10] Tim Kranich and Mladen Berekovic. NoC switch with credit based guaranteed service support qualified for GALS systems. In *Proceedings of the 13th Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 53–59, 2010. doi:10.1109/DSD.2010.30. Cited on page 41.

[KCT13] Pratyush Kumar, Devesh B. Chokshi, and Lothar Thiele. A satisfiability approach to speed assignment for distributed real-time systems. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 749–754, 2013. doi:10.7873/DATE.2013.160. Cited on page 37.

[KDK+89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25–40, 1989. doi:10.1109/40.16792. Cited on page 50.

[KJL13] Abbas Eslami Kiasari, Axel Jantsch, and Zhonghai Lu. Mathematical formalisms for performance evaluation of networks-on-chip. *ACM Computing Surveys*, 45(3):38:1–38:41, 2013. doi:10.1145/2480741.2480755. Cited on page 42.

[KK10] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2010. Cited on pages 3, 5, 16, 17, 18, 29, 43, 64, and 66.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011. doi:10.1007/978-3-642-22110-1_47. Cited on page 93.

[Koo96] Philip Koopman. Embedded system design issues (the rest of the story). In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 310–317, 1996. doi:10.1109/ICCD.1996.563572. Cited on page 9.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2nd edition, 2011. doi:10.1007/978-1-4419-8237-7. Cited on pages 1, 5, 10, 14, 15, 16, 17, 18, 19, 30, and 34.

[Kri14] C. Mani Krishna. Fault-tolerant scheduling in homogeneous real-time systems. *ACM Computing Surveys*, 46(4):48:1–48:34, 2014. doi:10.1145/2534028. Cited on pages 4 and 48.

[KS12] Christoph M. Kirsch and Ana Sokolova. The logical execution time paradigm. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, pages 103–120. Springer, 2012. doi:10.1007/978-3-642-24349-3_5. Cited on page 14.

[KSWJ06] Nikolay Kavaldjiev, Gerard J.M. Smit, Pascal T. Wolkotte, and Pierre G. Jansen. Providing QoS guarantees in a NoC by virtual channel reservation. In *Proceedings of the International Workshop on Applied Reconfigurable Computing*, volume 3985, pages 299–310, 2006. doi:10.1007/11802839_38. Cited on page 41.

[KTCG15] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. Accelerated microarchitectural fault injection-based reliability assessment. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 47–52, 2015. doi:10.1109/DFT.2015.7315134. Cited on page 111.

[KW08] Robert Kaiser and Stephan Wagner. The PikeOS concept–histroy and design. White paper, 2008. URL: https://www.sysgo.com/nc/news-events/document-center/whitepapers/pikeos-history-and-design-jan-2008 [accessed April 30, 2016]. Cited on page 33.

[KWQ+12] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th Conference on Operating Systems Design and Implementation (OSDI)*, pages 237–250. USENIX Association, 2012. URL: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-190.pdf [accessed April 30, 2016]. Cited on page 50.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015. URL: http://ecrts.eit.uni-kl.de/forum/viewtopic.php?f=20&t=23 [accessed April 30, 2016]. Cited on pages 11 and 81.

[LAB+11] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Sören Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, Michael Sokolsky, Ganymed Stanek, David Michael Stavens, Alex Teichman, Moritz Werling, and Sebastian Thrun. Towards fully autonomous driving: Systems and algorithms. In *Proceedings of the Intelligent Vehicles Symposium (IV)*, pages 163–168, 2011. doi:10.1109/IVS.2011.5940562. Cited on page 102.

[Lar34] Dionysius Lardner. Babbage's calculating engine. *Edinburgh Review*, LIX(CXX):263–327, 1834. URL: http://www.archive.org/details/Dionysius_Lardner_-_Babbages_Calculating_Engine [accessed April 30, 2016]. Cited on page 1.

[LBB04] Gholam Reza Latif-Shabgahi, Julian M. Bass, and Stuart Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328, 2004. doi:10.1109/TR.2004.832819. Cited on pages 64 and 65.

[Lee03] Sunggu Lee. Real-time wormhole channels. *Journal of Parallel and Distributed Computing*, 63(3):299–311, 2003. doi:10.1016/S0743-7315(02)00055-2. Cited on page 43.

[LGX+11] Weichen Liu, Zonghua Gu, Jiang Xu, Xiaowen Wu, and Yaoyao Ye. Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1382–1389, 2011. doi:10.1109/TPDS.2010.204. Cited on page 37.

[LIMM07] Christopher LaFrieda, Engin Ipek, Joé F. Martínez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 37th Annual International Conference on Dependable Systemsand Networks (DSN)*, pages 317–326, 2007. doi:10.1109/DSN.2007.100. Cited on page 44.

[LJL14] Shaoteng Liu, Axel Jantsch, and Zhonghai Lu. Parallel probe based dynamic connection setup in TDM NoCs. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, 2014. doi:10.7873/DATE.2014.252. Cited on page 41.

[LJS05] Zhonghai Lu, Axel Jantsch, and Ingo Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, volume 2, pages 960–964, 2005. doi:10.1109/ASPDAC.2005.1466499. Cited on page 43.

[LKB+09] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st Conference on Hot Topics in Parallelism (HotPar)*, 2009. URL: https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/liu/liu.pdf [accessed April 30, 2016]. Cited on page 35.

[LKP+10] Chanhee Lee, Hokeun Kim, Hae-woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 307–316, 2010. doi:10.1145/1878961.1879014. Cited on page 49.

[LLP15] Guanpeng Li, Qining Lu, and Karthik Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *Proceedings of the 45th Annual International Conference on Dependable Systems and Networks (DSN)*, pages 450–461, 2015. doi:10.1109/DSN.2015.36. Cited on page 76.

[LMJ+09] Zhonghai Lu, Mikael Millberg, Axel Jantsch, Alistair Bruce, Pieter van der Wolf, and Tomas Henriksson. Flow regulation for on-chip communication. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 578–581, 2009. doi:10.1109/DATE.2009.5090731. Cited on page 42.

[LRS+08] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276, 2008. doi:10.1145/1346281.1346315. Cited on page 44.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. doi:10.1145/357172.357176. Cited on page 32.

[Mar11] Peter Marwedel. *Embedded System Design*. Springer, 2nd edition, 2011. doi:10.1007/978-94-007-0257-8. Cited on pages 1, 9, 15, and 33.

[Mas03] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2003. Cited on page 34.

[MBD+00] P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes, and D. Beal. RTAI: Real-time application interface. *Linux Journal*, 72:1–10, 2000. URL: http://www.linuxjournal.com/article/3838 [accessed April 30, 2016]. Cited on page 33.

[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 1137–1142, 2012. doi:10.1145/2228360.2228568. Cited on page 21.

[ME02] David G. Mavis and Paul H. Eaton. Soft error rate mitigation techniques for modern microcircuits. In *Proceedings of the 40th Annual Reliability Physics Symposium*, pages 216–225, 2002. doi:10.1109/RELPHY.2002.996639. Cited on page 30.

[Mel15] Mellanox Technologies Ltd. TILEPro64 processor – product brief, 2015. URL: http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf [accessed May 1, 2016]. Cited on page 20.

[Men15] Mentor Graphics Crop. Nucleus RTOS, 2015. URL: http://s3.mentor.com/public_documents/datasheet/embedded-software/nucleus-rtos-ds.pdf [accessed April 30, 2016]. Cited on page 34.

[MHL+15] Sheng Ma, Libo Huang, Mingche Lai, Wei Shi, and Zhiying Wang. *Networks-on-Chip – From Implementations to Programming Paradigms*. Morgan Kaufmann, 2015. doi:10.1016/B978-0-12-800979-6.09985-6. Cited on pages 1 and 23.

[MKR02] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–110, 2002. doi:10.1109/ISCA.2002.1003566. Cited on page 50.

[MPGS06] Ivan Miro Panades, Alain Greiner, and Abbas Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the GALS approach. In *Proceedings of the 1st International Conference and Workshops on Nano-Networks (Nano-Net)*, pages 1–5, 2006. doi:10.1109/NANONET.2006.346219. Cited on page 108.

[MTK+11] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Ha Soonhoi, Chanhee Lee, Qiang Xu, and Lin Huang. Mapping of applications to MPSoCs. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 109–118, 2011. doi:10.1145/2039370.2039390. Cited on page 2.

[MVdWF08] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the Conference on Supercomputing (SC)*, pages 38:1–38:11, 2008. doi:10.1145/1413370.1413409. Cited on pages 1, 20, and 21.

[MWE+03] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 29–41, 2003. doi:10.1109/MICRO.2003.1253181. Cited on page 31.

[MZ14] Mohammad H. Mottaghi and Hamid R. Zarandi. DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors. *Microprocessors and Microsystems*, 38(1):88–97, 2014. doi:10.1016/j.micpro.2013.11.013. Cited on page 49.

[Nat13] National Highway Traffic Safety Administration. Preliminary statement of policy concerning automated vehicle, 2013. URL: http://www.autoalliance.org/index.cfm?objectid=CC9678B0-A415-11E5-997E000C296BA163 [accessed April 30, 2016]. Cited on page 2.

[NHM+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, pages 221–234, 2009. doi:10.1145/1629575.1629597. Cited on page 35.

[Nic11] Michael Nicolaidis, editor. *Soft Errors in Modern Electronic Systems*, volume 41 of *Frontiers in Electronic Testing*. Springer, 2011. doi:10.1007/978-1-4419-6993-4. Cited on pages 3, 29, and 30.

[Nik15] Borislav Nikolić. *Many-Core Platforms in the Real-Time Embedded Computing Domain*. PhD thesis, CISTER Research Center, 2015. URL: http://www.cister.isep.ipp.pt/docs/many_core_platforms_in_the_real_time_embedded_computing_domain/1110/view.pdf [accessed April 30, 2016]. Cited on pages 23 and 42.

[Nis97] Nimal Nissanke. *Realtime Systems*. Prentice Hall, 1997. Cited on page 10.

[NM65] John Ashworth Nelder and Roger Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965. URL: http://comjnl.oxfordjournals.org/content/7/4/308.full.pdf [accessed April 30, 2016]. Cited on page 131.

[NP12] Borislav Nikolić and Stefan M. Petters. Towards network-on-chip agreement protocols. In *Proceedings of the 10th international conference on Embedded Software (EMSOFT)*, pages 207–216, 2012. doi:10.1145/2380356.2380395. Cited on page 42.

[NVI14] NVIDIA Crop. Tesla GPU accelerators, 2014. URL: http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf [accessed April 30, 2016]. Cited on page 20.

[NYP14] Borislav Nikolić, Patrick Meumeu Yomsi, and Stefan M. Petters. Worst-case communication delay analysis for many-cores using a limited migrative model. In *Proceedings of the 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2014. doi:10.1109/RTCSA.2014.6910498. Cited on page 41.

[OB12] Pierre Olivier and Jalil Boukhobza. A hardware time manager implementation for the Xenomai real-time kernel of embedded linux. *ACM SIGBED Review – 2nd Workshop on Embed With Linux (EWiLi)*, 9(2):38–42, 2012. doi:10.1145/2318836.2318843. Cited on page 33.

[On-01] On-Chip Bus (OCB) Development Working Group. Virtual Component Interface Standard Version 2 (OCB 2 2.0. Technical report, VSI Alliance, Inc., 2001. Cited on pages 21 and 107.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002. doi:10.1109/24.994913. Cited on pages 4 and 47.

[Par94] Behrooz Parhami. Voting algorithms. *IEEE Transactions on Reliability*, 43(4):617–629, 1994. doi:10.1109/24.370218. Cited on pages 19, 38, and 65.

[PBdM+09] Nicolas Pouillon, Alexandre Becoulet, Aline Vieira de Mello, Francois Pecheux, and Alain Greiner. A generic instruction set simulator API for timed and untimed simulation and debug of MP2-SoCs. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, pages 116–122, 2009. doi:10.1109/RSP.2009.11. Cited on page 107.

[PEZ12] PEZY Computing Ltd. PEZY-SC, 2012. URL: http://pezy.co.jp/en/products/pezy-sc.html [accessed April 30, 2016]. Cited on pages 1 and 20.

[PIEP09] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):389–402, 2009. doi:10.1109/TVLSI.2008.2003166. Cited on page 48.

[PK08] Christian Paukovits and Hermann Kopetz. Concepts of switching in the time-triggered network-on-chip. In *Proceedings of the 14th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 120–129, 2008. doi:10.1109/RTCSA.2008.18. Cited on page 41.

[PMJ+06] Ahmad Patooghy, Seyed Ghassem Miremadi, A. Javadtalab, Mahdi Fazeli, and Navid Farazmand. A solution to single point of failure using voter replication and disagreement detection. In *Proceedings of the 2nd International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, pages 171–176, 2006. doi:10.1109/DASC.2006.15. Cited on page 44.

[PRI15] *PRISM Manual Version 4.3*, 2015. URL: http://www.prismmodelchecker.org/manual/Main/AllOnOnePage [accessed April 30, 2016]. Cited on pages 93 and 133.

[QLD09] Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *Proceedings of the 3rd International Symposium on Networks-on-Chip (NoCs)*, pages 44–53, 2009. doi:10.1109/NOCS.2009.5071444. Cited on page 42.

[QLD10] Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for on-chip packet-switching networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(5):802–815, 2010. doi:10.1109/TCAD.2010.2043572. Cited on page 42.

[QNX12] QNX Software Systems Ltd. Multicore processing user's guide, 2012. URL: http://support7.qnx.com/download/download/26177/Multicore_Processing_Users_Guide.pdf [accessed April 30, 2016]. Cited on page 34.

[Rap13] Rapita Systems Ltd. Rapitime explained. White paper, 2013. URL: https://www.rapitasystems.com/system/files/RapiTime%20Explained.pdf [accessed May 1, 2016]. Cited on page 11.

[RCA07] George A. Reis, Jonathan Chang, and David I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, 2007. doi:10.1109/MM.2007.4. Cited on pages 4 and 47.

[RCM13] Marcelo Ruaro, Everton Alceu Carara, and Fernando Gehm Moraes. Adaptive QoS techniques for NoC-based MPSoCs. In *Proceedings of the International Symposium on System on Chip (SoC)*, pages 1–6, 2013. doi:10.1109/ISSoC.2013.6675274. Cited on page 41.

[RCM14] Marcelo Ruaro, Everton Alceu Carara, and Fernando Gehm Moraes. Runtime QoS support for MPSoC: a processor centric approach. In *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 43:1–43:7, 2014. URL: http://www.inf.pucrs.br/~moraes/my_pubs/papers/2014/sbcci_ruaro.pdf, doi:10.1145/2660540.2661011. Cited on page 2.

[RCM15] Marcelo Ruaro, Everton Alceu Carara, and Fernando Gehm Moraes. Runtime adaptive circuit switching and flow priority in NoC-based MPSoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(6):1077–1088, 2015. doi:10.1109/TVLSI.2014.2331135. Cited on page 41.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 243–254, 2005. doi:10.1109/CGO.2005.34. Cited on page 47.

[RDJ+09] Franz Rammig, Michael Ditze, Peter Janacik, Tales Heimfarth, Timo Kerstan, Simon Oberthuer, and Katharina Stahl. Basic concepts of real time operating systems. In Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer, editors, *Hardware-dependent Software*, pages 15–45. Springer, 2009. doi:10.1007/978-1-4020-9436-1_2. Cited on page 33.

[RDMDY15] Laura A. Rozo Duque, Jose M. Monsalve Diaz, and Chengmo Yang. Improving MPSoC reliability through adapting runtime task schedule based on time-correlated fault behavior. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, pages 818–823, 2015. URL: http://dl.acm.org/citation.cfm?id=2755753.2755939 [accessed April 25, 2016]. Cited on page 50.

[RFZJ13] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys*, 46(1):8:1–8:38, 2013. doi:10.1145/2522968.2522976. Cited on page 113.

[RGH+10] Felix Reimann, Michael Glaß, Christian Haubelt, Michael Eberl, and Jürgen Teich. Improving platform-based system synthesis by satisfiability modulo theories solving. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 135–144, 2010. doi:10.1145/1878961.1878986. Cited on page 37.

[RGL+08] Felix Reimann, Michael Glaß, Martin Lukasiewycz, Joachim Keinert, Christian Haubelt, and Jürgen Teich. Symbolic voter placement for dependability-aware system synthesis. In *Proceedings of the 6th International Conference onHardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 237–242, 2008. doi:10.1145/1450135.1450190. Cited on page 44.

[RH07] Steven Rostedt and Darren V. Hart. Internals of the RT patch. In *Proceedings of the Linux Symposium*, pages 161–172, 2007. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-161-172.pdf [accessed May 1, 2016]. Cited on page 33.

[RKSH14] Semeen Rehman, Florian Kriebel, Muhammad Shafique, and Jörg Henkel. Reliability-driven software transformations for unreliable hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(11):1597–1610, 2014. doi:10.1109/TCAD.2014.2341894. Cited on pages 4 and 47.

[RLG+11] Felix Reimann, Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. Symbolic system synthesis in the presence of stringent real-time constraints. In *Proceedings of the 48th Design Automation Conference (DAC)*, pages 393–398, 2011. doi:10.1145/2024724.2024817. Cited on page 37.

[RMB+13] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 62(3):452–467, 2013. doi:10.1109/TC.2011.240. Cited on page 43.

[Ros14] Sheldon M. Ross. *Introduction to Probability Models*. Elsevier Science, 11th edition, 2014. Cited on pages 88 and 90.

[RPM+15] Tiana A. Rakotovao, Diego P. Puschini, Julien Mottin, Lukas Rummelhard, Amaury Negre, and Christian Laugier. Intelligent vehicle perception: Toward the integration on embedded many-core. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA-DITAM)*, pages 7–12, 2015. doi:10.1145/2701310.2701313. Cited on page 2.

[RRTV99] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 210–218, 1999. doi:10.1109/DFTVS.1999.802887. Cited on page 47.

[RSKH11] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Proceedings of the 7th International Conferenceon Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 237–246, 2011. doi:10.1145/2039370.2039408. Cited on pages 31 and 47.

[RTK+13] Semeen Rehman, Anas Toma, Florian Kriebel, Muhammad Shafique, Jian-Jia Chen, and Jörg Henkel. Reliable code generation and execution on

unreliable hardware under joint functional and timing reliability considerations. In *Proceeedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 273–282, 2013. doi:10.1109/RTAS.2013.6531099. Cited on page 48.

[Rut14] Jochem Hendrik Rutgers. *Programming models for many-core architectures: a co-design approach*. PhD thesis, University of Twente, 2014. URL: http://doc.utwente.nl/90661/ [accessed May 1, 2016]. Cited on pages 20 and 34.

[SAB+15] Muhammad Shafique, Philip Axer, Christoph Borchert, Jian-Jia Chen, Kuan-Hsun Chen, Björn Döbel, Rolf Ernst, Hermann Härtig, Andreas Heinig, Rüdiger Kapitza, Florian Kriebel, Daniel Lohmann, Peter Marwedel, Semeen Rehman, Florian Schmoll, and Olaf Spinczyk. Multi-layer software reliability for unreliable hardware. *it - Information Technology*, 57(3):170–180, 2015. doi:10.1515/itit-2014-1081. Cited on page 45.

[SAC14] Mohamed M. Sabry, David Atienza, and Francky Catthoor. OCEAN: An optimized HW/SW reliability mitigation approach for scratchpad memories in real-time SoCs. *ACM Transactions on Embedded Computing Systems - Special Issue on Real-Time and Embedded Technology and Applications, Domain-Specific Multicore Computing, Cross-Layer Dependable Embedded Systems, and Application of Concurrency to System Design (ACSD)*, 13(4s):138:1–138:26, 2014. doi:10.1145/2584667. Cited on page 45.

[SB08] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the 2nd International Symposium on Networks-on-Chip (NOCS)*, pages 161–170, 2008. doi:10.1109/NOCS.2008.11. Cited on page 41.

[SBSK12] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, 2012. doi:10.1109/NOCS.2012.25. Cited on page 41.

[SC13] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture - Embedded Systems Design*, 59(1):60–76, 2013. doi:10.1016/j.sysarc.2012.10.004. Cited on page 37.

[Sch11] Ute Schiffel. *Hardware Error Detection Using AN-Codes*. PhD thesis, Technische Universität Dresden, 2011. URL: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-69872 [accessed May 1, 2016]. Cited on pages 46 and 68.

[Sch13] Patrick R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2nd edition, 2013. doi:10.1007/978-1-4614-3737-6. Cited on page 44.

[SEH+12] Christian El Salloum, Martin Elshuber, Oliver Höftberger, Haris Isakovic, and Armin Wasicek. The ACROSS MPSoC—a new generation of multi-core processors designed for safety-critical embedded systems. In *Proceedings of the 15th Conference on Digital System Design (DSD)*, pages 105–113, 2012. doi:10.1109/DSD.2012.126. Cited on page 2.

[SEU+15] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 220–229, 2015. doi:10.1109/CODESISSS.2015.7331385. Cited on page 2.

[SGD15] Wei-Tsun Sun, Alain Girault, and Gwenaël Delaval. A formal approach for the synthesis and implementation of fault-tolerant industrial embedded systems. In *Proceedings of the 10th International Symposium on Industrial Embedded Systems (SIES)*, pages 264–272, 2015. doi:10.1109/SIES.2015.7185068. Cited on page 45.

[SGK+04] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas Nowatzyk. Fingerprinting: bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22–29, 2004. doi:10.1109/MM.2004.72. Cited on pages 28 and 44.

[SHD+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC)*, pages 245–255, 2015. doi:10.1109/EDCC.2015.28. Cited on pages 51, 52, 111, and 130.

[Shi09] Zheng Shi. *Real-Time Communication Services for Networks on Chip*. PhD thesis, University of York, 2009. URL: https://cs.york.ac.uk/rts/documents/thesis/shi09.pdf [accessed May 1, 2016]. Cited on page 41.

[SK09] Joseph Sloan and Rakesh Kumar. Towards scalable reliability frameworks for error prone CMPs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 261–270, 2009. doi:10.1145/1629395.1629432. Cited on page 44.

[Sla11] Charles Slayman. Soft error trends and mitigation techniques in memory devices. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–5, 2011. doi:10.1109/RAMS.2011.5754515. Cited on page 104.

[SLL07] Sang H. Son, Insup Lee, and Joseph Y.-T. Leung, editors. *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007. doi:10.1201/9781420011746. Cited on page 11.

[SM00] William H. Sanders and John F. Meyer. *Stochastic Activity Networks: Formal Definitions and Concepts*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343. Springer, 2000. doi:10.1007/3-540-44667-2_9. Cited on page 90.

[SMR+07] A. Shye, T. Moseley, V.J. Reddi, J. Blomstedt, and D.A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306, 2007. doi:10.1109/DSN.2007.98. Cited on page 50.

[SNG12] Radu Stefan, Ashkan Beyranvand Nejad, and Kees Goossens. Online allocation for contention-free-routing NoCs. In *Proceedings of the Interconnection Workshop on Network Architecture: On-Chip, Multi-Chip (INA-OCMC)*, pages 13–16, 2012. doi:10.1145/2107763.2107767. Cited on page 41.

[Sor09] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Morgan & Claypool, 2009. doi:10.2200/S00192ED1V01Y200904CAC005. Cited on pages 3, 16, 28, 29, 30, and 43.

[SPM09] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *ACM SIGBED Review - Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 6(3):6:1–6:5, 2009. doi:10.1145/1851340.1851348. Cited on page 48.

[Sri10] Vilas Keshav Sridharan. *Introducing abstraction to vulnerability analysis*. PhD thesis, Northeastern University, 2010. URL: http://www.vilas-sridharan.com/personal/Publications_files/VKS_Dissertation_Final.pdf [accessed May 1, 2016]. Cited on page 31.

[SRP+13] Rishad A. Shafik, Gerard Rauwerda, Jordy Potman, Kim Sunesen, Dhiraj K. Pradhan, Jimson Mathew, and Ioannis Sourdis. Software modification aided transient error tolerance for embedded systems. In *Proceedings of the Conference on Digital System Design (DSD)*, pages 219–226, 2013. doi:10.1109/DSD.2013.32. Cited on page 47.

[SS87] Michael A. Schuette and John Paul Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, 36(3):264–276, 1987. doi:10.1109/TC.1987.1676899. Cited on page 70.

[SSKH13] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: Survey of current and emerging

trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 1:1–1:10, 2013. doi:10.1145/2463209.2488734. Cited on page 49.

[SSSF10a] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: Detecting hardware errors in software. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 169–182, 2010. doi:10.1007/978-3-642-15651-9_13. Cited on page 46.

[SSSF10b] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-implemented hardware error detection: Costs and gains. In *Proceedings of the 3rd International Conference on Dependability (DEPEND)*, pages 51–57, 2010. doi:10.1109/DEPEND.2010.16. Cited on pages 46 and 52.

[ST11] Yahia Salah and Rached Tourki. Design and FPGA implementation of a QoS router for networks-on-chip. In *Proceedings of the 3rd International Conference on Next Generation Networks and Services (NGNS)*, pages 84–89, 2011. doi:10.1109/NGNS.2011.6142551. Cited on page 41.

[Str09] Patrik Strömblad. ENEA multicore: High performance packet processing enabled with a hybrid SMP/AMP OS technology. White paper, 2009. URL: http://www.enea.com/Embedded-hub/whitepapers/white-papers/Multicore-High-performance-packet-processing-enabled-with-a-hybrid-SMPAMP-OS-technology [accessed May 1, 2016]. Cited on page 34.

[Swi14] John Swinimer. AMD FirePro awarded top spot on the Green500 list, 2014. URL: http://www.amd.com/en-us/press-releases/Pages/amd-firepro-awarded-2014nov20.aspx [accessed May 1, 2016]. Cited on page 20.

[TDa] Gabriela Tucher-Denkinger. Erprobungsfahrzeug Tesla, Kofferaum. URL: http://www.bosch-mediaspace.de/mediaspace/media/object/view/details.htm?objectId=54265 [accessed March 28, 2016]. Cited on page 2.

[TDb] Gabriela Tucher-Denkinger. Tesla auf Autobahn A81, von innen. URL: http://www.bosch-mediaspace.de/mediaspace/media/object/view/details.htm?objectId=54256 [accessed March 28, 2016]. Cited on page 2.

[TD02] Brian Towles and William J. Dally. Worst-case traffic for oblivious routing functions. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–8, 2002. doi:10.1145/564870.564872. Cited on page 115.

[TEHJ11] Mihkel Tagel, Peeter Ellervee, Thorsten Hollstein, and Gert Jervan. Communication modelling and synthesis for NoC-based systems with real-time constraints. In *Proceedings of the 14th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 237–242, 2011. doi:10.1109/DDECS.2011.5783086. Cited on pages 43 and 49.

[TGL07] Paul Teehan, Mark R. Greenstreet, and Guy G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design Test of Computers*, 24(5):418–428, 2007. doi:10.1109/MDT.2007.151. Cited on page 21.

[The11] The International Technology Roadmap for Semiconductors. System drivers, 2011. URL: http://www.itrs2.net/itrs-reports.html [accessed May 1, 2016]. Cited on page 1.

[Tid13] Tidorum Ltd. Bound-T time and stack analyzer: User guide, 2013. URL: http://www.bound-t.com/manuals/user-guide.pdf [accessed May 1, 2016]. Cited on page 11.

[TS09] Peter Tummeltshammer and Andreas Steininger. Power supply induced common cause faults-experimental assessment of potential countermeasures. In *Proceedings of the International Conference on Dependable Systems Networks (DSN)*, pages 449–457, 2009. doi:10.1109/DSN.2009.5270308. Cited on page 31.

[TSH+15] Guy Martin Tchamgoue, Junho Seo, Jongsoo Hyun, Kyong Hoon Kim, and Yong-Kee Jun. Supporting fault-tolerance in a compositional real-time scheduling framework. *ACM SIGBED Review - Special Issue on the 7th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 12(2):7–15, 2015. doi:10.1145/2782753.2782754. Cited on page 48.

[TW06] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Pearson Prentice Hall, 3rd edition, 2006. Cited on page 33.

[UHK+12] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. Eliminating single points of failure in software-based redundancy. In *Proceedings of the 9th European Dependable Computing Conference (EDCC)*, pages 49–60, 2012. doi:10.1109/EDCC.2012.21. Cited on pages 5, 6, 52, 54, 67, 73, and 139.

[Uhl05] Michael Uhler. *See MIPS Run*. Computer Architecture and Design. Morgan Kaufmann, 2nd edition, 2005. doi:10.1016/B978-012088421-6/50000-8. Cited on page 108.

[Ulb14] Peter Ulbrich. *Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:

de:bvb:29-opus4-50561 [accessed May 1, 2016]. Cited on pages 52 and 54.

[Vaj11] András Vajda. *Programming Many-Core Chips*. Springer, 2011. doi:10.1007/978-1-4419-9739-5. Cited on pages 1, 20, 34, and 36.

[VHR+08] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008. doi:10.1109/JSSC.2007.910957. Cited on page 21.

[VWOH15] Michael Vonbun, Stefan Wallentowitz, Andreas Oeldemann, and Andreas Herkersdorf. An analytic approach on end-to-end packet error rate estimation for network-on-chip. In *Proceedings of the Conference on Digital System Design (DSD)*, pages 621–628, 2015. doi:10.1109/DSD.2015.82. Cited on page 113.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009. doi:10.1145/1531793.1531805. Cited on pages 1 and 35.

[WCS09] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode multicore reliability. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 169–180, 2009. doi:10.1145/1508244.1508265. Cited on page 44.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389. Cited on pages 11 and 22.

[WF07] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 356–369, 2007. doi:10.1007/978-3-540-75101-4_34. Cited on pages 46 and 68.

[WF11] Markus Winter and Gerhard P. Fettweis. Guaranteed service virtual channel allocation in NoCs for run-time task scheduling. In *Proceedings of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*,

pages 419–424, 2011. doi:10.1109/DATE.2011.5763073. Cited on page 41.

[WGO+13] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 583–594, 2013. doi:10.1145/2485922.2485972. Cited on page 41.

[Win16] Wind River Systems Inc. VxWORKS product overview, 2016. URL: http://www.windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf [accessed May 1, 2016]. Cited on page 34.

[XLK+04] Yuan Xie, Lin Li, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Reliability-aware co-synthesis for embedded systems. In *Proceedings of the 15th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 41–50, 2004. doi:10.1109/ASAP.2004.1342457. Cited on page 44.

[XLK+07] Yuan Xie, Lin Li, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Reliability-aware co-synthesis for embedded systems. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(1):87–99, 2007. doi:10.1007/s11265-007-0057-6. Cited on page 44.

[YISK15] Tomohiro Yoneda, Masashi Imai, Hiroshi Saito, and Kenji Kise. Dependable real-time task execution scheme for a many-core platform. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 197–204, 2015. doi:10.1109/DFT.2015.7315162. Cited on pages 52 and 54.

[YLJY15] Liu Yuan, Huaida Liu, Pingui Jia, and Yiping Yang. An adaptive ECC scheme for dynamic protection of NAND flash memories. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1052–1055, 2015. doi:10.1109/ICASSP.2015.7178130. Cited on page 28.

[ZCH97] Robert Zijal, Gianfranco Ciardo, and Günter Hommel. Discrete deterministic and stochastic Petri nets. In *Proceedings of the 9th ITG/GI-Fachtagung zu Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen (MMB)*, pages 103–117, 1997. URL: https://www.researchgate.net/publication/221441049_Discrete_Deterministic_and_Stochastic_Petri_Nets [accessed May 1, 2016]. Cited on page 90.

[ZKCS02] Cesar A. Zeferino, Márcio E. Kreutz, Luigi Carro, and Altamiro A. Susin. A study on communication issues for systems-on-chip. In *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 121–126, 2002. doi:10.1109/SBCCI.2002.1137647. Cited on page 23.

Publications by the Author

The following publications (sorted in chronological order) contain material relevant to this thesis.

[MFRC15] Peter Munk, Matthias Freier, Jan Richling, and Jian-Jia Chen. Dynamic guaranteed service communication on best-effort networks-on-chip. In *Proceedings of the 3rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 353–360, 2015. doi:10.1109/PDP.2015.47. Cited on pages 6, 55, and 107.

[AML+15] Mohammad Shadi Alhakeem, Peter Munk, Raphael Lisicki, Helge Parzyjegla, and Gero Mühl. A framework for adaptive software-based reliability in COTS many-core processors. In *Workshop Proceedings of the 28th International Conference on Architecture of Computing Systems (ARCS)*, pages 1–4, 2015. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7107113 [accessed March 28, 2016]. Cited on pages 6, 7, and 87.

[MAL+15] Peter Munk, Mohammad Shadi Alhakeem, Raphael Lisicki, Helge Parzyjegla, Jan Richling, and Hans-Ulrich Heiß. Toward a fault-tolerance framework for COTS many-core systems. In *Proceedings of the 11th European Dependable Computing Conference (EDCC)*, 2015. doi:10.1109/EDCC.2015.32. Cited on pages 6, 7, 63, 87, 107, and 143.

[MAL+16] Peter Munk, Mohammad Shadi Alhakeem, Raphael Lisicki, Hendrik Röhm, Helge Parzyjegla, and Hans-Ulrich Heiß. A software fault-tolerance mechanism for real-time applications on many-core processors. In *Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH)*, 2016. doi:10.13140/RG.2.1.3929.0004. Cited on pages 6, 63, 87, and 107.

The following papers were published during the course of my PhD but are not that part of this thesis.

[MR14] Peter Munk and Jan Richling. Migration-aware WCET estimation for heterogeneous multi-cores. *ACM SIGBED Review – Special Issue on the 6th Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 11(3):22–25, 2014. doi:10.1145/2692385.2692388. Not cited.

[MSRH15] Peter Munk, Björn Saballus, Jan Richling, and Hans-Ulrich Heiß. Position paper: Real-time task migration on many-core processors. In *Workshop Proceedings of the 28th International Conference on Architecture of Computing Systems (ARCS)*, pages 1–4, 2015. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7107098 [accessed March 28, 2016]. Not cited.

[BMGH15] Alexander Biewer, Peter Munk, Jens Gladigau, and Christian Haubelt. On the influence of hardware design options on schedule synthesis in time-triggered real-time systems. In *Proceedings of the 18. Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 105–114, 2015. URL: `http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-164197` [accessed March 28, 2016]. Not cited.

List of Figures

List of Tables

List of Listings

List of Symbols

a	activation time
A	availability
A_P	point availability
A_L	long-term availability
$\check{A}$	target availability
A	integer constant A from ANBD
b	bandwidth
B	worst-case repair time in Periods
$\hat{B}$	worst-case repair time
B	signature B from ANBD
$\mathscr{B}$	binomial distribution
c	completion time
C	core
$\mathscr{C}$	core set
$\lvert\mathscr{C}\rvert$	number of cores
$\hat{C}$	worst-case communication time
D	relative deadline
$\mathbf{D}$	dynamic signature of encoded voting algorithm
D	sequence counter D from ANBD
$\hat{\mathsf{D}}$	maximum sequence counter value
δ	destination
$\mathscr{E}$	equality set of encoded voting algorithm
$\hat{E}$	worst-case execution time
f	probability of fault per period
F	packet injection rate
h	number of hops
l	link
$\mathscr{L}$	link set
L	latency
$\hat{L}$	worst-case latency

L_p	packet latency
L_δ	latency destination
$\hat{L}_\delta$	worst-case latency destination
$\hat{L}_p$	worst-case packet latency
L_r	latency router
$L_\not{\epsilon}$	traversal latency
$\hat{L}_\not{\epsilon}$	worst-case traversal latency
L_C	latency collision
$\hat{L}_C$	worst-case latency collision
$\hat{L}_B$	worst-case latency blockage
λ	soft error rate (SER)
$\breve{\lambda}$	target SER
Λ	failure rate
Λ_V	voter failure rate
Λ_T	task failure rate
$\breve{\Lambda}_T$	target task failure rate
$\Lambda_{\tilde{T}}$	spare failure rate
Λ_C	critical voter failure rate
m	message
$\lvert m_i \rvert$	message size
$\mathscr{M}$	message set
M	number of replicas that have to deliver the same result
$\mathbb{N}^+$	natural numbers (without 0)
N	number of task replicas
O	offset
P	period
$\mathring{P}$	hyperperiod
p	packet
$\mathscr{P}$	set of packets
$\lvert p_i \rvert$	packet size
$\overrightarrow{p_i}$	request packet
$\overrightarrow{\mathscr{P}}$	set of request packets
$\overleftarrow{p_i}$	response packet
$\overleftarrow{\mathscr{P}}$	set of response packets

π	transition probability
$\mathbf{P}$	transition probability matrix
Pr	probability operator
r	router
$\mathscr{R}$	set of routers
$\hat{R}$	worst-case response time
R	reliability
$\check{R}$	target reliability
ρ	transition rate
s	start time
$\mathscr{S}$	system
S	number of task spares
$\mathbf{S}$	static signature of encoded voting algorithm
σ	source
t	time
t_m	mission time
T	task
$\mathscr{T}$	task set
$\widetilde{T}$	spare task
V	voter
v	vulnerability factor
v_V	voter vulnerability factor
v_T	task vulnerability factor
$v_{\widetilde{T}}$	spare vulnerability factor
v_C	critical voter vulnerability factor
x	state
$\mathscr{X}$	state space
X	random variable
X	task replica result
X'	encoded task replica result
$\mathbb{Z}$	integer numbers

List of Abbreviations

ADAS	advanced driver assistance systems
ALU	arithmetical logical unit
AMD	Advanced Micro Devices
AMP	asymmetric multi-processor
ANSI	American National Standards Institute
API	application programmable interface
ASIL	automotive safety integrity level
AXI	Advanced eXtensible Interface
BE	best-effort
BMP	bound multi-processor
CABA	cycle-accurate and bit-accurate
CBS	constant bandwidth server
CDS	code duplication service
CIL	common intermediate language
CMOS	complementary metal-oxide-semiconductor
CoRed	Combined Redundancy
COTS	commercial off-the-shelf
CPU	central processing unit
CRC	cyclic redundancy check
CSDF	cyclo-static data-flow
CTMC	continuous time Markov chain
DDSPN	discrete deterministic and stochastic Petri net
DMPS	distributed message passing service
DMR	dual modular redundancy
DSPIN	Distributed, Scalable, Predictable Interconnect Network
DTMC	discrete time Markov chain
DTTR	Duplication with Temporary TMR and Reconfiguration
DVFS	dynamic voltage and frequency scaling
DVS	dynamic voltage scaling
DWC	duplication with comparison
ECC	error-correcting code
eCos	Embedded Configurable Operating System
ECU	electronic control unit
EDDI	Error Detection by Duplicated Instructions
EDF	earliest deadline first

ELF	Executable and Linkable Format
FCU	fault containment unit
FIFO	first in, first out
FIT	failure in time
flit	flow control unit
FMECA	failure mode, effects and criticality analysis
fos	Factored OS
FPGA	field-programmable gate array
FPU	floating point unit
FTA	fault tree analysis
GALS	globally asynchronous, locally synchronous
GCC	GNU's Not Unix (GNU) compiler collection
GNU	GNU's Not Unix
GPS	Global Positioning System
GPU	graphics processing unit
GS	guaranteed service
HAD	highly automated driving
HAL	hardware abstraction layer
ICR	inter-core reset
IEC	International Electrotechnical Commission
IP	intellectual property
IPC	inter-process communication
ISA	instruction set architecture
ISO	International Organization for Standardization
ISS	instruction set simulator
IVI	instruction vulnerability index
KPN	Kahn process network
LC	link controller
LET	logical execution time
LLC	last-level cache
LMM	Limited Migrative Model
MCU	multi-cell upset
MIC	Many Integrated Core
MILP	mixed integer linear programming
MIMD	multiple instructions, multiple data
MIPS	Microprocessor without Interlocked Pipeline Stage
MMTL	maximum measured transfer latency
MMU	memory management unit

MOC	model of computation
MPPA	Massively Parallel Processor Array
MPSoC	Multi-Processor System-on-Chip
MPU	memory protection unit
MTTF	mean time to failure
MTTUF	mean time to unsafe failure
NI	network interface
NMR	N modular redundancy
NoC	Network-on-Chip
NUMA	non-uniform memory access
OCP	Open Core Protocol
OS	operating system
OSE	Operating System Embedded
PC	program counter
PCB	printed circuit board
PMHF	probabilistic metric for random hardware failures
PPN	polyhedral process network
QM	Quality Management
QoS	Quality of Service
RAM	random access memory
RCB	reliable computing base
RISC	Reduced Instruction Set Computing
ROM	read-only memory
RT	real-time
RTAI	Real-Time Application Interface
RTOS	real-time operating system
SAN	stochastic activity network
SAT	satisfiability
SCC	Single-chip Cloud Computer
SDC	silent data corruption
SDF	synchronous data-flow
SDS	state duplication service
SEE	single event effect
SEL	single event latch-up
SER	soft error rate
SET	single event transient
SEU	single event upset
SIL	safety integrity level
SIMD	single instructions, multiple data

SMP	symmetric multi-processor
SMT	satisfiability modulo theories
SoC	System-on-Chip
SPOF	single point of failure
SRAM	static random access memory (RAM)
STHORM	ST(microelectronics) Heterogeneous lOwpoweR Many-core
SVN	subversion
TCB	trusted computing base
TDM	time-division multiplexing
TMR	triple modular redundancy
TTY	teletypewriter
UMA	uniform memory access
VCI	Virtual Component Interface
WCCT	worst-case communication time
WCET	worst-case execution time
WCPL	worst-case packet latency
WCRT	worst-case response time
WCTL	worst-case transfer latency