Linköping Studies in Science and Technology

Licentiate Thesis No. 1771

Timing Predictability in Future Multi-Core Avionics Systems

by

Andreas Löfwenmark

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

Linköping 2017

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.

A doctor's degree comprises 240 ECTS credits (4 years of full-time studies).

A licentiate's degree comprises 120 ECTS credits.

Copyright © 2017 Andreas Löfwenmark

ISBN 978-91-7685-564-5

ISSN 0280-7971

Printed by LiU Tryck 2017

URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-134427>

Abstract

With more functionality added to safety-critical avionics systems, new platforms are required to offer the computational capacity needed. Multi-core platforms offer a potential that is now being explored, but they pose significant challenges with respect to predictability due to shared resources (such as memory) being accessed from several cores in parallel. Multi-core processors also suffer from higher sensitivity to permanent and transient faults due to shrinking transistor sizes.

This thesis addresses several of these challenges. First, we review major contributions that assess the impact of fault tolerance on worst-case execution time of processes running on a multi-core platform. In particular, works that evaluate the timing effects using fault injection methods. We conclude that there are few works that address the intricate timing effects that appear when inter-core interferences due to simultaneous accesses of shared resources are combined with the fault tolerance techniques. We assess the applicability of the methods to COTS multi-core processors used in avionics. We identify dark spots on the research map of the joint problem of hardware reliability and timing predictability for multi-core avionics systems.

Next, we argue that the memory requests issued by the real-time operating systems (RTOS) must be considered in resource-monitoring systems to ensure proper execution on all cores.

We also adapt and extend an existing method for worst-case response time analysis to fulfill the specific requirements of avionics systems. We relax the requirement of private memory banks to also allow cores to share memory banks.

This work has been supported by the Swedish Armed Forces, the Swedish Defence Materiel Administration and the Swedish Governmental Agency for Innovation Systems under grant number NFFP6-2013-01203.

Acknowledgements

First of all, I would like to thank my main advisor, Simin Nadjm-Tehrani, for her patience and constructive feedback. I also want to thank my secondary advisor at LiU, Christoph Kessler, for his valuable input and proofreading.

I want to thank Saab, and especially my industrial advisor Ingemar Söderquist, for giving me the opportunity to pursue this research. An extra thank you goes to Ingemar for proofreading and for our fruitful discussions.

Then I want to thank past and present members of RTSLAB, who all have pitched in to create a joyful and inspiring working environment where I have learned a lot. I have enjoyed our meetings, discussions and of course the *fika*.

Another big thank you goes to everyone in the lunch group for making (so many of) my lunches fun and interesting. The variety of discussion topics is endless and I learn new things every time. The quality of the jokes can be discussed though. ;-)

I also want to thank my girlfriend, *Sarah*, who means the world to me. I am very grateful for her endless love and support.

Finally, I want to thank all my other friends (who are not important enough to have been mentioned already ;-)) and family for their support.

Ride on!

*Andreas Löfwenmark
Linköping, Sweden
February 2017*

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem formulation	2
1.3	Contributions	3
1.4	List of publications	3
1.5	Thesis outline	4
2	Background	5
2.1	Multi-Core Basics	5
2.2	Shared Resources	5
2.2.1	Caches	6
2.2.2	Random Access Memory	7
2.2.3	Memory Controller	8
2.2.4	Interconnect	9
2.3	Avionic Architectures	10
2.3.1	Federated Architecture	10
2.3.2	Integrated Modular Avionics	11
2.3.3	Robust Partitioning	11
2.3.4	RTCA/DO-178C	12
2.3.5	ARINC 653	13
2.3.6	Safety and Certification	14
2.3.7	Mixed-Criticality Systems	14
2.4	Fault Management	15
2.4.1	Fault Tolerance	15
2.4.2	Fault Injection	16
2.5	Worst-Case Execution Time Estimation	17
3	Related Work	19
3.1	Resource Management	19
3.1.1	Execution Models	19
3.1.2	Caches	21
3.1.3	Scratchpad Memory	22
3.1.4	Shared Memory Banks	22
3.1.5	Memory Controller	23

3.2	Resource Monitoring	24
4	The Impacts of Faults on Timing	27
4.1	Impact of Faults on WCET	27
4.2	Hardware Error Detection and Correction	28
4.2.1	Faults in Cache	28
4.2.2	Faults in DRAM	29
4.2.3	Faults in General-Purpose Registers	29
4.3	Fault Tolerance	30
4.3.1	Fault Tolerance Methods	30
4.3.2	Fault-Aware Timing Analysis	33
4.4	Fault Injection	36
4.4.1	Hardware Fault Injection	36
4.4.2	Software Fault Injection	37
4.5	Discussion	38
5	Memory Request Monitoring	43
5.1	Motivation	43
5.2	Methodology	45
5.2.1	Hardware Platform	45
5.2.2	Software Platform	46
5.3	Measurement Results	48
5.4	Discussion	52
6	Shared Memory Bank Interference	55
6.1	DRAM Model	55
6.1.1	Inter- and Intra-Bank Delay	56
6.2	SCE Adaptation and Extension	58
6.2.1	Memory Request Monitoring	58
6.2.2	Memory Bank Partitioning	59
6.2.3	Cache Partitioning	59
6.3	Validation of the SCE Extension	59
6.3.1	Methodology	60
6.3.2	WCET Estimation	60
6.3.3	Response Time	61
6.3.4	Critical Processes	64
6.3.5	Applying the Method to an Earlier Benchmark	65
6.4	Discussion	66
7	Conclusions and Future Work	67
7.1	Conclusions	67
7.2	Future Work	69

List of Figures

2.1	Example of Multi-Core Architecture.	6
2.2	Various shared resources and their interference sources. . . .	6
2.3	Schematic view of a DRAM Module.	8
2.4	Schematic view of a DRAM Chip.	9
2.5	Small system – Federated Architecture.	10
2.6	Small system – IMA Architecture.	11
5.1	Effects of RTOS memory requests when counted by the run-time monitoring system.	44
5.2	Effects of RTOS memory requests when not counted by the run-time monitoring system.	45
5.3	Memory request characteristics for included applications. . .	48
5.4	Memory Requests per partition window.	49
5.5	Total number of memory requests.	50
5.6	Distribution of user mode memory requests per partition window (outliers have been grouped together into one if there are many with the same value).	51
6.1	Components of RD_p	57
6.2	Execution time measurements in isolation.	61
6.3	Comparison of memory requests.	62
6.4	Theoretical interference delay when using four cores.	63
6.5	Cores sharing banks causing interference delay.	63
6.6	Comparison of measured and estimated request time.	65

List of Tables

2.1	Failure rate per RTCA/DO-178C criticality level.	13
4.1	Classification of Surveyed Papers.	39
5.1	Distribution of total number of memory requests.	50
5.2	Proportion of supervisor mode requests per partition window.	51
6.1	DRAM timing parameters.	56
6.2	Characterization of partitions in isolation.	62
6.3	Worst-case service times.	62
6.4	Maximum response time of partitions.	64
6.5	Measured maximum response time of Mult with memory-intensive tasks in parallel.	65

Chapter 1

Introduction

Added functionality in future avionics systems brings complexities to both design and operation of these systems and requires new platforms that offer more computational capacity. Multi-core processing offers a potential that the industry is exploring, which opens up for new research questions in the context of safety-critical systems. The multi-core processors available commercially today, often referred to as commercial off-the-shelf (COTS), are inherently less predictable than single-core processors due to shared resources and the efforts of the chip manufacturers to optimize the throughput. This affects the analyses of worst-case execution time (WCET) and worst-case response time (WCRT). In the absence of new techniques, these analyses tend to result in very pessimistic estimates, which could negate the added computational capacity.

1.1 Motivation

Modern avionics system development is undergoing a major transition, from federated systems to Integrated Modular Avionics (IMA) where several aircraft functions can reside on the same platform. Moreover, there is a departure from today's single-core computing due to possible single-core obsolescence. We need to address the predictability problems of multi-core platforms (e.g., how to guarantee determinism (in time and space) for application tasks running on multiple cores and interacting through shared memory).

Multi-core platforms offer greater computational capacity with less size, weight and power (SWaP) compared to a single-core platform, and are used in diverse domains from mobile devices to supercomputing. However, safety-critical cyber-physical systems such as avionic and automotive [57] systems have not (yet) embraced the technology due to predictability issues. Aerospace systems are subject to costly and time-consuming certification processes, which require a predictable behavior under fault-free and

certain faulty conditions.

Predictability is also fundamental for establishing real-time correctness. In today's multi-core platforms, different cores share hardware resources such as caches and memory, which were essentially developed with a focus on maximizing the throughput, but when placed in the safety-critical context introduce challenges to predictability. However, these challenges have to be met as aerospace is moving towards higher exploitation of commercial-off-the-shelf (COTS), as well as aiming to exploit the low SWaP characteristics of multi-core. Although the use of concurrent processes and shared resources is well-studied in real-time systems, the estimation of worst-case execution time (WCET) and maximum response time for each process is based on well-defined regimes allowing mutually exclusive access to each shared resource. With applications running on multi-core platforms, this option no longer exists unless all caches are partitioned and the timing effects of requests to the shared memory are predictably estimated.

Avionics systems often operate at high altitude and are more exposed to cosmic radiation than electronics at ground-level. This cosmic radiation has sufficient energy to alter the states of circuit components, resulting in corrupted data (e.g., in caches and memory), often referred to as *soft errors*. Safety and reliability requirements of the system entail making a serious attempt to make it fault-tolerant by masking the soft errors.

To use multi-core processors in a safety-critical avionics system, both problems (i.e., timing predictability and fault tolerance) have to be addressed together. Safety-critical systems have to produce the correct output within the allotted time and different fault tolerance methods have different impact on the execution time of tasks in the system. The verifiability of the chosen fault tolerance mechanism and the WCET estimates must also be ensured. To create repeatable experiments, we also need appropriate fault injection methods.

1.2 Problem formulation

The purpose of this thesis is to identify challenges to the deployment of multi-core processors in safety-critical systems, and to provide mechanisms for meeting some of these challenges. The research goals are:

- **Predictability:** Study, devise and improve methods for assuring timing predictability and temporal isolation for avionics systems realized on multi-core platforms with shared resources.
- **Fault Tolerance:** Investigate how single event effects and their mitigation impact predictability and temporal isolation of avionics systems with certification constraints when realized on multi-core platforms with shared resources.

- **Maturity:** Determine if there exist methods considering predictability and fault tolerance on multi-core so as the deployment of multi-core platforms in safety-critical avionics systems can be achieved.

Assessing the maturity of methods does not, by itself, solve the aerospace problems, but it is a necessary step for formulating appropriate research goals meeting the challenges mentioned above. This thesis provides a partial addressing of the challenges and is a work-in-progress towards deploying safety-critical avionics systems on multi-core platforms.

1.3 Contributions

This thesis contributes techniques for bringing the aerospace industry closer to efficiently using multi-core processors in safety-critical avionics systems. The contributions of this thesis are as follows.

- (a) **Impact of memory requests from the operating system on inter-core interference:** We show the magnitude of memory requests issued by the Real-Time Operating System (RTOS) and that they should be considered when analyzing inter-core interference in a resource-monitoring system.
- (b) **Methodology for validation of worst-case response time with shared Dynamic Random Access Memory (DRAM) banks:** We extend an existing work with a timing model considering the inter-core interference in the DRAM and we relax the assumption of private DRAM banks to allow for using *shared* DRAM banks. We adapt the proposed Linux-specific mechanisms for use on an avionic relevant RTOS. We also devise a methodology for the validation of our approach.
- (c) **Review of state-of-the-art research addressing WCET estimation combined with fault tolerance on multi-core:** We investigate the joint problem of WCET estimation and fault tolerance on multi-core platforms to evaluate the maturity for deploying a certifiable fault-tolerant safety-critical real-time avionics system. Several research gaps are identified and we also identify directions for future research.

1.4 List of publications

The work in this thesis is the result of the following publications:

- A. Löfwenmark and S. Nadjm-Tehrani. **Challenges in future avionic systems on multi-core platforms.** In Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW 2014). IEEE, November 2014.

- A. Löfwenmark and S. Nadjm-Tehrani. **Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform.** In Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015). IEEE, November 2015.
- A. Löfwenmark and S. Nadjm-Tehrani. **Understanding Shared Memory Bank Access Interference in Multi-Core Avionics.** In Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016). OASiCs, July 2016.

The following report has also been prepared and submitted for evaluation:

- A. Löfwenmark and S. Nadjm-Tehrani. **Worst-Case Execution Time under Faulty Conditions on Multi-Core - A Survey.** Under Submission.

1.5 Thesis outline

The thesis is organized as follows:

Chapter 2 - Background provides the basic concepts relevant to understanding the rest of the thesis. We present common multi-core architectures and the various shared resources present in many multi-core platforms.

Chapter 3 - Related Work presents existing related works, addressing timing related (WCET) issues on multi-core platforms, on which our own work is built. Fault tolerance and its impact on WCET estimations is reviewed in more detail in Chapter 4.

Chapter 4 - The Impacts of Faults on Timing presents a survey identifying several gaps in research addressing multi-core WCET estimations and the impact fault tolerance mechanisms have on these estimates.

Chapter 5 - Memory Request Monitoring focuses on studying the magnitude of memory requests issued by an RTOS and the impact they have on the inter-core interference analysis.

Chapter 6 - Shared Memory Bank Interference provides a methodology for WCET estimation and validation while allowing multiple cores to share memory banks.

Chapter 7 - Conclusions and Future Work concludes the work presented in this thesis and presents potential future work.

Chapter 2

Background

This chapter provides the background material needed for understanding the contributions presented in this thesis.

2.1 Multi-Core Basics

A multi-core processor implements the multiple instruction multiple data (MIMD) technique [21,25] to achieve parallelism. There are multiple design choices to consider for a multi-core processor:

- Cores may or may not share caches.
- Cores may or may not share memory (private memory vs global or distributed shared memory).
- Cores may or may not be identical (homogeneous vs heterogeneous).
- Cores may be connected using different network topologies, such as bus, ring, mesh or crossbar.

In this thesis, we focus on shared-memory multi-core systems and a schematic view of the system used in this thesis is shown in Figure 2.1.

2.2 Shared Resources

In a single-core system only one task can access a resource at a time, but problems may arise in a multi-tasking system if the accesses from two tasks are interleaved or interrupted due to preemption (e.g., as a result of priority based scheduling). This is an extensively studied issue (e.g., [7,9,20,55,76]). On a multi-core system on the other hand, requests can be issued at the same time and may have to be arbitrated by the hardware somehow. This

Figure 2.1: Example of Multi-Core Architecture.

arbitration, which is not present on a single-core system, can impact the predictability. Next, the four components illustrated in Figure 2.2, where sharing can impact predictability will be briefly reviewed.

Figure 2.2: Various shared resources and their interference sources.

2.2.1 Caches

Parts of the cache architecture may affect the interaction between cores, e.g. *cache sharing* [26,48], *cache coherence* [26], and the *cache replacement policy* [2]. Cache sharing is present on platforms where the cores share one or more levels of cache. Cache coherence is needed for keeping shared data consistent among the core-private caches and the main memory. The cache replacement policy affects how cache lines are replaced when the cache is full.

Four types of cache interference may arise due to cache sharing: (a) intra-task interference, where a task evicts its own cache lines; (b) inter-task interference, where a task evicts cache lines belonging to another task; (c) cache pollution caused by asynchronous events such as interrupt service routines; and (d) inter-core interference, where tasks on different cores evict cache lines belonging to each other [58]. The first three are not new and exist in a multitasking single-core system as well, but the last one is new in multi-core systems. Another problem is that the cores may try to access the cache at the same time and one core will block the other core's access, which will introduce delays for the blocked core and timing-variability in the system. To achieve the required performance in IMA systems, using the L2 cache is important in time and space partitioned RTOSes [48] and the inter-core interference problem needs to be addressed.

The cache coherency protocol is responsible for keeping the shared data, cached in core-private caches, in sync between all cores storing the shared data and also with the main memory. If two cores (C1 and C2) operate on independent data, which are stored in main memory in such a way that the data ends up in the same coherency domain, the cache coherency protocol will invalidate C1's cache, when C2 writes to its data. C1 will suffer a delay when it tries to access its data because the cache has been invalidated and needs to be reloaded. This is known as *false sharing* and has the same impact as if the cores were accessing the same data [26]. Thus, the interference caused by the coherency protocol can add to the timing-variability of the system.

With respect to the cache replacement policy, the Least Recently Used (LRU) policy is the most analyzable policy as this policy does not suffer from domino effects [2]. Regardless of this, manufacturers replace the LRU policy with Pseudo-LRU (PLRU), which is a policy that almost always discards one of the least recently used items, or the FIFO policy. Both PLRU and FIFO are cheaper solutions in terms of implementation cost, but introduce domino effects affecting the WCET estimates.

2.2.2 Random Access Memory

Random access memory (RAM) is used in computer systems as main memory. The two main types of RAM used in a computer system are (a) static RAM (SRAM) (b) dynamic RAM (DRAM) where SRAM is based on a flip-flop circuit requiring very low power when not being accessed. It is more expensive and has low storage density, and is therefore mostly used for caches. DRAM is based around a capacitor that can be charged and discharged to represent '1' and '0'. However, the capacitor will slowly leak and requires periodic refreshes. It consumes more power, but can store more data at a lower cost. Therefore, DRAM is often used for main memory and in this thesis, we are primarily interested in main memory so we will not consider SRAM any further. The most common variant of DRAM nowadays,

is the synchronous DRAM (SDRAM) where the operation is coordinated by an externally-supplied clock signal (e.g., changes on control inputs are recognized after a rising edge of its clock input). In this thesis, the term DRAM will refer to the synchronous variant and is used interchangeably with the term SDRAM.

A memory module (such as a double data rate type three dual in-line memory module [DDR3 DIMM], which is found in many desktop computers) is built from several DRAM chips that make up the total data path width. Normal widths of each chip are four and eight bits, commonly referred to as “x4” (“by four”) and “x8” (“by eight”). To make up a 64-bit data path, eight “x8” chips are needed, see the example in Figure 2.3. If the DIMM supports error detection and correction, additional chips are added.

Figure 2.3: Schematic view of a DRAM Module.

DRAM memory devices are organized into ranks (set of DRAM chips connected to the same chip select signal (Figure 2.3 shows one rank)). Ranks can be accessed independently, but not simultaneously as they share the data bus. To improve performance, a rank contains a number of *banks* that can be accessed independently. Figure 2.4 shows a schematic view of DRAM chip. Each bank contains a *data array* organized as a matrix with a specified number of *rows* and *columns*. To read data from this data array, the row containing that data must be opened and the contents read into the *row buffer*, from which the column containing the data can be read. The row buffer is used to store the data from a row as the row is drained of data during the read due to implementation details (capacitors are discharged). Subsequent requests to the same row can be serviced with lower latency, as the row is already open (the data is in the row buffer). If a request requires another row to be opened, this will increase the latency as the currently open row must be closed and the data written back to the row before the new row can be opened. This will also influence the inter-core interference if different cores request data from different rows in the same bank.

2.2.3 Memory Controller

In addition to the memory devices, the DRAM memory system also contains a memory controller. A memory controller is a shared resource just like

Figure 2.4: Schematic view of a DRAM Chip.

the caches, and it has the same kind of issues. One core accessing the DRAM can be blocked because of another core accessing the DRAM at the same time. This interaction may prevent hardware synchronization of processors as is typically done [48] and can lead to schedule skid and jitter, resulting in timing-variability since synchronization has to be performed in software instead of hardware. The inter-core interference may be reduced by introducing additional memory controllers in the system [2], but this will not scale well with increasing number of cores.

The memory controller sends a number of commands to the DRAM devices: activate (ACT) to open a row; read/write (RD/WR) to read or write data to the row buffer; precharge (PRE) to close an open row. As described in Section 2.2.2, a row has to be read into the row buffer to be accessible. To read a row, the sequence (PRE, ACT, RD) is required to close the previously open row, read it into the row buffer and to read the wanted data. When using the open page policy, a subsequent access to the same row does not require the PRE-command, thus shortening the latency. The close page policy on the other hand always closes the row and another PRE-command is required even if the access is to the same row. This increases the latency, but the timings are more deterministic.

2.2.4 Interconnect

Multi-core processors are often part of a System on Chip (SoC) together with peripherals such as external memory, serial I/O and Ethernet. To handle all requests to the shared peripherals, an interconnect is implemented to arbitrate the requests. The internal workings of the interconnect and how it prioritizes the requests are often part of the manufacturer's intellectual property. In a safety-critical system the configuration of the interconnect cannot be ignored since it can cause a core to be blocked by another core accessing a resource in parallel [48].

The interconnect is a crucial component of the multi-core processor – making the alternative gold standard (see Section 2.3.3) difficult to realize if the hardware has not been developed for it. The behavior or performance in one partition may be affected by another partition, violating the alternative gold standard, which is incompatible with existing guidelines [2,41].

2.3 Avionic Architectures

Avionics systems are transitioning [94] from a federated architecture to one based on the Integrated Modular Avionics (IMA) concept [80]. Here we will briefly describe each and some differences. We will also describe other concepts relevant to the development of avionics systems.

2.3.1 Federated Architecture

Traditionally, avionics systems have been implemented following a federated design. Each avionic function is implemented in self-contained units, referred to as line-replaceable units (LRUs), that perform their unique tasks. The LRUs contain CPU, memory and I/O. Figure 2.5 shows a small federated system with three LRUs, sensors and actuators. Application tasks (circles labelled T1, . . . , T4 in the figure) execute on the CPUs. Each LRU is connected to each of the others by a point-to-point connection. As each LRU has its own unique design tailored for its unique tasks, adding system functionality often causes major system redesign, integration test and verification work. There are also high costs for maintenance due to the large number of spare LRUs (and spare parts to repair the LRUs) required.

Figure 2.5: Small system – Federated Architecture.

2.3.2 Integrated Modular Avionics

Avionics systems today are more often implemented using IMA [80]. IMA uses a high-integrity, partitioned environment that hosts multiple functions of different criticalities on a shared computing platform. The partitioning concept (see Section 2.3.3) ensures a fault containment level equivalent to its functionally equivalent federated system, which implies partitioning in both space and time [1,81]. Figure 2.6 shows the same small system as in Figure 2.5, but integrated on an IMA platform. Here, the application tasks have been consolidated on one CPU and a common network is used to connect the system.

Figure 2.6: Small system – IMA Architecture.

Work is ongoing in the European Union funded project ASHLEY [6] to develop the next generation, IMA2G.

2.3.3 Robust Partitioning

Robust partitioning is (re)defined and refined in several documents which makes it complicated to extract the official definition [41]. For example, RTCA/DO-297 [80] defines it by stating that it “ensures that any hosted application or function has no unintended effect on other hosted applications or functions”. Jean et al. refer to work by Rushby [82] defining the *Gold Standard for Partitioning*: “A robustly partitioned system ensures a fault containment level equivalent to its functionally equivalent federated system.”. Wilding et al. [96] define the *Alternative Gold Standard for Partitioning*: “The behavior and performance of software in one partition must be unaffected by the software in other partitions,” which is a stronger property and a sufficient condition to establish robust partitioning.

Robust partitioning consists of the following the concepts:

Fault Containment Functions should be separated in such a way that no

failure in one function (application) can cause another function to fail. Low criticality tasks should not affect high criticality tasks.

Space Partitioning No function may access the memory space of other functions (unless explicitly configured).

Temporal Partitioning A function's access to a set of hardware resources during a period of time is guaranteed.

Robust partitioning has traditionally been implemented in the federated architecture with dedicated hardware per application or function. With the introduction of IMA and multi-core, the robust partitioning property needs to be addressed and ensured [41].

ARINC 653 [1] is a software specification for space and time partitioning (see Section 2.3.5), aiming at an alternative gold standard. It contains its own interpretation of robust partitioning: “The objective of Robust Partitioning is to provide the same level of functional isolation as a federated implementation.”

The space partitioning concept can be implemented on multi-core systems as is, as long as the hardware can prevent invalid memory requests. The time partitioning concept, on the other hand, is problematic as ARINC 653 states that a partition should have exclusive access to its physical resources. Since the main memory is a shared resource (as we discussed in Section 2.2), achieving time partitioning in multi-core systems is not trivial.

2.3.4 RTCA/DO-178C

Development of safety-critical airborne software is guided by the de-facto standard RTCA/DO-178C [81], Software Considerations in Airborne Systems and Equipment Certification. It contains a number of objectives to be fulfilled. The software is assigned a design assurance level (DAL), ranging from level A (most critical) to E (non-critical), depending on the criticality as determined in the safety assessment process. In addition to RTCA/DO-178C the certification authorities have issued the Certification Authorities Software Team position paper (CAST 32A) [15] that identifies topics impacting the safety, performance and integrity of an airborne software system executing on multi-core processors.

It is required that probabilistic safety guarantees are provided to functionalities of different criticalities as shown in Table 2.1. Clearly, higher criticality levels require more stringent analyses of both hardware and software to ensure the assurance levels needed. WCET estimates need to be determined more thoroughly and tend to be larger (more pessimistic) in higher criticality levels.

Table 2.1: Failure rate per RTCA/DO-178C criticality level.

Level	Failure Condition	Failure Rate
A	Catastrophic	$10^{-9}/\text{h}$
B	Hazardous	$10^{-7}/\text{h}$
C	Major	$10^{-5}/\text{h}$
D	Minor	$10^{-3}/\text{h}$
E	No Effect	N/A

2.3.5 ARINC 653

ARINC 653 [1] (Avionics Application Standard Software Interface) is a software specification defining a general-purpose interface between the RTOS of an avionics computer and the application software. It allows multiple applications of different criticality levels (such as the RTCA/DO-178C DAL) to execute on the same computing platform, without affecting one another spatially or temporally (space and time partitioning), in the context of an IMA architecture. A partition is a program unit of the application to satisfy these partitioning constraints. Each partition has its own statically assigned memory space. ARINC 653 defines an Application Programming Interface (API) called Application/EXecutive (APEX), which enhances portability. The APEX API contains services to manage partitions and processes, to handle inter- and intra-partition communication, and also services for error handling. The API provides services in the following categories:

- Partition Management
- Process Management
- Time Management
- Inter-partition communication
- Intra-partition communication
- Error Handling

No services are provided for memory management; this has to be handled by the partition itself (within its statically assigned memory space).

A partition consists of one or more processes that combine to provide the functions associated with that partition. Processes within a partition share the same memory address space, and are scheduled according to a priority-based preemptive scheduling scheme. The naming convention using the terms partition and process is somewhat confusing as for general purpose operating systems the terms process and thread are often used, where the processes are separated in memory and the threads share the memory address space.

Partitions are scheduled in a static cyclic schedule, which prescribes when and for how long a partition can execute. The major frame time determines the length of the repeating schedule, and each major frame is divided into partition windows with a specified start time within the major frame and a duration. A partition can be scheduled in arbitrarily many partition windows within a major frame.

All resources needed by a partition are statically allocated in a configuration file. This includes for instance the partition window schedule, the memory requirements and inter-partition communication channels. During partition initialization, all resources used by the partition are created. When the normal mode execution starts, it is no longer possible to create additional processes.

The current revision of ARINC 653 is for single-core processors, but work is currently ongoing by the Airlines Electronic Engineering Committee (AEEC) APEX subcommittee¹ to extend the standard for use on multi-core processor architectures.

2.3.6 Safety and Certification

A safety-critical avionics system has to be certified by the authorities, e.g. the Federal Aviation Administration (FAA) in United States or the European Aviation Safety Agency (EASA) in Europe, covering both hardware and software.

The certificate, which is on the aircraft level, testifies that the type of aircraft including used technology meets the safety requirements. One important certification requirement is the development under the guidance of Aerospace Recommended Practice (ARP). ARP4754A [83] identifies relationships to other guidelines, such as RTCA/DO-297 [80] for IMA, RTCA/DO-254 (ED-80) [79] for airborne electronic hardware, and RTCA/DO-178C (ED-12C) [81] for airborne software.

For RTCA/DO-254 hardware certification, it is possible to rely on service experience in comparable applications if no problems have been discovered in previous systems [26]. The problem for multi-core processors is that the implementation has very limited service history [48].

Both RTCA/DO-178C and RTCA/DO-254 were written before multi-core processors were used in civil aircraft, and only addresses software hosted on single-core processors [23].

2.3.7 Mixed-Criticality Systems

Most embedded systems consist of many different functions. However, all functions in a system are not equally critical for correct service or mission (e.g., the flight control system is more critical than the infotainment system in a commercial airliner). A mixed-criticality system is a system

¹<http://www.aviation-ia.com/aeec/projects/apex/>

hosting several different aircraft functions assigned different DALs on the same computing platform. When estimating WCETs for the functions, the most critical ones may get a more stringent analysis than the less critical ones.

The multi-criticality task model was proposed by Vestal [92] after showing that existing scheduling theory could not address multiple criticality requirements. Multiple WCET estimates are specified for each task, one WCET for the criticality level the task belongs to and one for each lower level. Higher criticality levels demand greater levels of assurance and the WCET will therefore be more pessimistic than for the same task at a lower criticality level. Similarly, treatment of tasks in terms of resource allocation, and analysis of fault tolerance can be differentiated depending on the task criticality.

2.4 Fault Management

In this thesis, we use the fundamental concepts of faults, errors, failures, transient, intermittent, permanent and so on, in accordance with the well-known notions of dependability [8]. In this section, we briefly discuss concepts relevant for this thesis.

2.4.1 Fault Tolerance

Fault tolerance is the ability of a system to continue its intended operation, possibly at a reduced, but acceptable level, in the presence of faults [8] (so that faults do not affect the external state of a system). The basic concepts of fault tolerance are *faults*, *errors* and *failures*. A fault is the cause of an incorrect system state, errors are manifestations of faults, and when an error affects the external state of the system a failure has occurred.

The process of making a system fault-tolerant can be summarized in four steps [64]: *proactive fault management*, *error detection*, *fault diagnosis* and *recovery*. In the proactive step one tries to predict failures and also prevent them from happening. Examples of proactive fault management are software rejuvenation and online checks during run-time (such as memory scrubbing). A watchdog can be used to detect timing errors, which could be a result of implementation mistakes or bit-flips due to high-energy particles (cosmic radiation). Both of these effects could result in the system being stuck in an infinite loop. Other types of errors can be treated using redundancy at different levels in the system. When an error is detected, a way to locate the faulty component and also the type of the fault can help with the handling of the error. Using Triple Modular Redundancy (TMR) for instance can help identify a faulty component. It is important to mitigate the fault before a failure is triggered.

Faults, Errors and Failures

Faults can be classified as *permanent*, *intermittent* or *transient*, where transient faults are random, intermittent faults are repetitive, and a permanent fault is continuous in time. A fault can be either *active*, meaning it has manifested as an error, or *dormant*, where it has not yet produced an error.

Faults can originate from hardware or software. Software faults are faults (bugs) in the software, which could result from the design or implementation phase. They are permanent in nature, but the effects of the bugs can be transient [37]. In this thesis, we refer to these kinds of bugs as *transient software faults*. Other terms for software faults are Bohrbug (permanent), Mandelbug and Heisenbug (transient) [29,67]. The majority of software faults found in production software are transient as most of the other bugs, those that always fail, have hopefully been discovered in the process of design, review and testing [29].

A fault can produce many different types of errors, which can be either *detected* (i.e., an error message or an error signal exist) or *latent*. A system failure occurs if an error affects the external state of the system. Whether or not an error causes a failure depends on the structure and the behavior of the system. The system may contain redundancy to explicitly prevent an error from causing a failure, or the system may overwrite faulty data before a latent error causes a failure.

System failures occur when the delivered service deviates from the intended service and the duration or severity of service degradation is not acceptable. Depending on the type of failure and the system design, the system may either behave erratic or it may stop responding.

2.4.2 Fault Injection

To study the impact of transient faults and to determine how they should be treated one could wait for a fault to happen, but this would be very time consuming as transient faults are rare. Fault injection is a technique for evaluating the dependability of systems. It can be used to inject faults into a system with the aim of observing its dependability behavior and assessing the fault tolerance mechanisms.

Fault injection is a practical approach for achieving confidence in that faults cannot cause serious failures. This is done by intentionally introducing faults in the hardware on which the application runs. Fault injection can take many forms and can be introduced in different phases of system development. The faults are simulated using a fault model, and the selection of fault model is an issue that influences the potential outcomes of the injection experiments. Radiation beams is an example of a physical injection method that can be used to inject faults in a hardware circuit and can produce transient faults in random locations inside a hardware circuit, but it cannot reveal aging effects. Using radiation beams is also a time-consuming and expensive task. Alternatives have been developed, some hardware-based,

some software-based, and some simulation-based [75].

Hardware-based fault injection includes radiation-beam testing, risking permanent damage to the tested device and requires special hardware instrumentation. Software-based fault injection is cheap and is achieved by altering the contents of CPU registers or memory while running the relevant application software on the hardware being tested. Simulation-based fault-injection offers complete visibility inside the device under test provided the simulation model of the hardware is detailed enough.

Software fault injection (SFI) attempts to simulate software faults through code changes, either at compile-time by modifying the source code or at run-time by using a trigger and then changing data values (e.g., in memory or registers). Mutation testing modifies existing lines of code and can be used to simulate faults unintentionally introduced by programmers. SFI can be used to inject code changes, data errors and interface errors. Addressing the questions of when, where and what to inject are important to create an efficient fault injection method.

2.5 Worst-Case Execution Time Estimation

WCET (and WCRT) estimations are essential ingredients for establishing a predictable timing behavior in safety-critical systems, but the introduction of multi-core processors has made the estimations difficult to perform. Having to account for failures complicates the task even further. WCET estimation can be classified as static or measurement-based.

The static analysis methods compute the execution time of individual code blocks using a microarchitectural model of the target platform they will execute on. By design it will find the longest path and can thus provide a safe (overestimated) upper bound instead of the WCET. The amount of overestimation is dependent on the microarchitectural model, which can be very difficult to produce for complex platforms (e.g., multi-core with multi-threading, branch prediction, pipelining and multi-level caches).

The measurement-based analysis can provide accurate execution times as it is running on the target platform, but since it depends on actual execution it may be difficult to know if the WCET path has been covered and there is always the risk that the WCET estimation is not equal to the real WCET.

Hybrid WCET estimation techniques also exist, where the application is executed (several times) on the target hardware and execution traces are collected. With the execution traces, together with knowledge of the code structure, a WCET estimate can be produced. For this to function, all statements in the application source code have to be executed during trace collection.

A probabilistic timing analysis (PTA) [14,17] can be used to calculate a probabilistic WCET (pWCET). The failure rates specified for different DALs (Table 2.1) can be utilized during the pWCET calculations to find the relevant target probability. Two variants exist, static (SPTA) and mea-

surement based (MBPTA), as for the classical deterministic WCET estimations. These pWCET estimates are derived in such a way as to provide indications of likelihood (e.g., the estimates can be exceeded with a given probability [17]). Instead of requiring time-deterministic behavior of the platform, PTA is based on randomization of the timing behavior for some hardware (e.g., caches and shared buses).

Having covered the basic concepts and terminology required for understanding the rest of the thesis, we now move on to describe the related works in the next chapter.

Chapter 3

Related Work

Although a lot of research has been performed on multi-processor systems (e.g., [62,87,99]), focus for the great majority of these works is throughput and performance, not worst-case guarantees and predictability, thereby works with a focus on performance boost are not further discussed here.

How to utilize all cores in a safety-critical system while maintaining predictability has received a lot of attention. The focus here is predictability and timing aspects that are important in safety-critical applications. This chapter provides a review of the related works on fault-free timing predictability forming the basis for the work in this thesis. Fault tolerance and its impact on WCET estimations are reviewed in more detail in Chapter 4.

We roughly group the different methods into two categories, *resource management* and *resource monitoring*. Methods in these categories are reviewed in Section 3.1 and 3.2 respectively.

3.1 Resource Management

To address the timing variability problems due to the shared resources discussed in Section 2.2, resource management methods divide the shared resource among the users (e.g., partitioning caches, only allowing a task access during a certain phase in the execution).

3.1.1 Execution Models

Execution models attack the interferences by imposing restrictions on the applications, to ease the analysis and to allow control of the requests to the shared resources.

The superblock model is proposed for analyzing the worst-case response time (WCRT) by Schranzhofer et al. [84]. Applications are divided into a number of superblocks, where each superblock consists of an *acquisition*, an *execution* and a *replication* phase. The phases have a maximal execution

time and a maximal number of shared resource requests assigned. Three different access models are examined: *dedicated*, where resource requests are allowed only during the acquisition and replication phase; *general*, where resource requests are allowed at any time and the three phases are unused; and *hybrid*, where the three phases are used, but resource requests are still allowed at any time. The models are analyzed analytically for their worst-case performance (considering blocking/no-buffered requests to shared resources) to study schedulability and the impact on the execution time bounds. They show that separating computations and accesses to the shared resource is important for accurately computing the WCRT. They assume a time division multiple access (TDMA) arbiter for handling the requests to the shared resources. However, resource arbitration using TDMA may lead to under-utilization and may not be the best choice when both predictability and performance are important [45]. TDMA also requires special hardware support and if no such hardware support is available, other means of management are required.

The PRedictable Execution Model (PREM) [73] is another attempt to overcome the problems of non-predictable COTS components. The authors of this model also divide tasks into intervals (compatible and predictable), similar to superblocks above, where the predictable interval is further divided into a memory phase and an execution phase. During the memory phase, shared memory requests are issued to fill the cache with the data needed for the execution phase, which is performed without memory requests and cache misses. The I/O peripherals can access main memory during an execution phase without contention. The compatible interval is used for executing code that is too complex for the restrictions of the predictable interval, it is also used for executing operating system calls. Two new hardware devices are introduced, *real-time bridge* and *peripheral scheduler*, to make the scheduling of I/O peripherals more predictable.

The predictable interval (which is used with the same meaning as superblock) is also used to derive a modeling-based WCRT analysis method based on Timed Automata (TA) [49] and model checking is used for verification of the temporal properties of the system. To improve the scalability of the model checking, they also present a method that replaces parts of the model's abstract representations based on access request arrival curves [51]. The Uppaal toolbox [11] is used for modeling the TA.

Giannopoulou et al. [27] use the dedicated access model described above and present a scheduling policy for a mixed-criticality multi-core system with resource sharing without the need for special hardware support. The policy prevents timing interference among the tasks of different criticality levels by allowing only a statically known set of tasks with the same criticality level to execute on the cores at any time. To achieve more efficient resource utilization, static and dynamic barriers are used for synchronization on the global level, and on the core-level a flexible time-triggered scheduling strategy is used. This enforces timing isolation between criticality levels

and enables composable and incremental certifiability. The cost is run-time overhead for the clock and barrier synchronization between the cores.

An approach similar to Giannopoulou’s [27] is proposed by Anderson et al. [5] and further extended by Mollison et al. [63] who present a two-level hierarchical scheduling framework for mixed-criticality tasks. The top-level schedules container tasks, which in turn contain the “normal” tasks. Each container consists of tasks of the same criticality level and uses a given scheduling strategy.

3.1.2 Caches

The works in Section 3.1.1 all assume each core has its own private local memory or do not consider caches at all. However, shared caches introduce a number of problems and are a major source of non-determinism. Ward et al. [93], following Anderson et al., Mollison et al. and Herman et al. [5,34,63] add cache management to make the shared cache more predictable and reduce WCET estimates for high-criticality tasks. Page coloring is used to ensure that pages with different colors cannot cause cache conflicts. Since allocating the colors optimally is NP-hard in the strong sense, conflict-free color assignments may be impossible. To mitigate this, they propose treating the colors as shared resources to which requests must be arbitrated.

Mancuso et al. [58] address cache sharing interference by proposing a two-stage solution. The first stage profiles the critical real-time tasks to determine the memory request patterns, and the second stage is a deterministic cache allocation strategy named “Colored Lockdown”. It combines page coloring and cache locking. Page coloring is used to optimize the cache usage for frequently used memory pages and lockdown is used to override the cache replacement policy to ensure that the pages stay in the cache.

Special hardware mechanisms to reduce cache-related delays have also been proposed. A system for explicit reservation of cache memory [95], where the state of the cache memory is saved before a preemption and restored before the preempted task is resumed. The cache state is restored in a single operation before the task is resumed instead of multiple cache misses after the task has been resumed.

As mentioned in Section 2.5, PTA [14,17] can be used to calculate pWCETs. Instead of requiring time-deterministic behavior of the platform as most other methods, PTA is based on randomization of the timing behavior for the caches.

A method calculating pWCETs without the need to partition shared caches is presented by Slijepcevic et al. [90]. The method utilizes an additional hardware mechanism for controlling when evictions are allowed in the cache.

Altmeyer et al. [4] investigate different SPTA approaches for precision, correctness and optimality and derive a method using the number of distinct memory block accesses to make the analysis more accurate.

3.1.3 Scratchpad Memory

As the number of cores in a processor increases, scaling the traditional memory hierarchy becomes a problem since caches consume a lot of power and space on the chip. A platform architecture without caches and only a local scratchpad memory (SPM) for each core is more power-efficient and more scalable [56]. In a software managed multi-core system the code and data have to be moved to the scratchpad memory from main memory using direct memory access (DMA) before it can be executed. If all code and data fit in the SPM it is trivial to handle. However, if it doesn't fit it must be dynamically handled and this makes dynamic code management techniques a vital component.

Kim et al. [47] present two novel WCET-aware techniques for mapping data and code to the SPM (previous techniques have focused on the average-case execution time instead of WCET). They construct a variant of a control flow graph (CFG) called inline CFG, which is used as input to the analysis. They perform analysis based on integer linear programming to find an optimal function-to-region mapping leading to the lowest WCET, but the technique is not scalable when the number of functions grows. A heuristic algorithm that is scalable but sub-optimal is therefore proposed.

3.1.4 Shared Memory Banks

In addition to task (CPU) scheduling, the partitioning of task data in the shared memory is also important. A DRAM with several banks (as discussed in Section 2.2.2) enables tasks on different cores to access the memory in parallel without delaying each other provided that they access different banks. This is used by Giannopoulou et al. [28], where they propose a memory mapping optimization, applied at design time, to minimize the timing interference of tasks when accessing the memory. They also pinpoint that the memory mapping optimization is interdependent with the task scheduling and mapping used [27] and propose an integration of the two. An evaluation of an industrial application shows that memory mapping optimization can reduce the task response times.

The memory has often been regarded as a single resource (black box) and a constant time used for the access times, but the DRAM can be in one of several states affecting the time required to perform the access. PAL-LOC [102] is a DRAM bank-aware memory allocator, which is used to partition the memory banks found in DRAMs. It allows system designers to assign DRAM banks to cores, thus providing private banks to applications running on different cores. This way the inter-core interference is reduced. This will also eliminate any row collisions, where two cores request different rows in the same bank. Row collisions are more expensive than row hits as the currently open row has to be closed.

Wu et al. [97] and Kim et al. [46] both model the memory system more realistically than as a single resource. The memory controller has one re-

quest queue for each DRAM bank. Wu et al. only consider private DRAM banks for each core. Kim et al. who also include shared banks relax this limitation. Yun et al. [103] study interference arising in COTS platforms that can generate multiple outstanding memory requests and evaluate their approach on a simulation platform.

3.1.5 Memory Controller

There are also efforts to develop new memory controller hardware better suited for hard real-time systems. Several aspects of the controller can affect the access latency.

The analyzable memory controller (AMC) [70] uses round-robin to arbitrate between hard real-time tasks (HRTs), prioritizes the HRTs over the non-HRTs, uses one request queue per task and a close-page policy to derive an upper bound delay for memory requests. This upper bound delay can then be considered in the WCET analysis.

Instead of viewing the DRAM device as a single resource that is shared among all clients, Reineke et al. [77] propose a DRAM controller that views the DRAM device as several resources to be shared among clients individually. They interleave accesses to blocks in a time-triggered way to eliminate contention for the shared resources within the device.

A work-conserving time-division multiplexing (TDM) arbiter together with dynamic command scheduling is used in a memory controller presented by Li et al. [53] to provide tight bounds on the WCRT for real-time tasks, while also providing low average response time to non-real-time tasks. This memory controller is modeled using timed automata and analyzed using model checking to derive WCRT and worst-case bandwidth bounds for the memory controller [54].

There are many different memory arbiters that can be used in a system and many existing methods are tightly coupled to the arbitration policy. Dasari et al. [19] propose a general framework that considers this diversity to help system designers. The analysis is divided into an arbiter dependent step followed by an arbiter independent step.

DRAM modules utilizing multiple ranks is a flexible option for mixed critical workloads, but the rank switching time is becoming an issue for modern DRAM modules and Ecco et al. [22] present a DRAM memory controller minimizing the rank switching activity.

The approaches in this section all contain new hardware components. These can of course be used in the future if they make it into commercially available hardware platforms. In this thesis, we only consider readily available COTS hardware platforms.

3.2 Resource Monitoring

Another approach to the timing variability problems is *resource monitoring*, where the resource usage is monitored and requests can be restricted if a given limit is exceeded.

Resource monitoring can be used in two different ways: it can be used during development in order to estimate the parameters needed for WCRT estimation or interference bounding analysis. An example in our context is the work by Dasari et al. [18], where the number of bus accesses per task is measured and then used to perform schedulability tests. Monitoring can also be used at run-time to starve a task that demands more than the predefined share of resources during well-defined periods. We will refer to the latter as *run-time monitoring*.

For single-core systems based on ARINC 653 [1] the requests to shared resources (basically only the CPU) are monitored by the RTOS ensuring that the statically defined partition execution schedule is followed. One approach to avoid the shared resource interference while still using multiple cores [26] utilizes an ARINC 653 compliant partitioning model. A safety-critical partition executes on its own core, while the other cores are idle. When no critical application is running, several partitions may utilize the available cores. This eliminates shared resource interference on the critical applications, but it is also quite expensive since many cores may be unused in several partition windows.

In addition to considering only the CPU, the memory requests can also be monitored. This can be implemented using performance monitor counters found in most CPUs to track the consumed memory bandwidth and a scheduler (i.e., a run-time monitor) could use the performance monitor counters to prevent contention and to spread the memory requests [39], thereby improving predictability.

A mix of the two monitoring approaches is found in the work by Yun et al. [104] who use performance counters to get information on memory requests to separate real-time and non-real-time tasks. They concentrate the critical tasks to one core (the critical core) and the other cores (interfering cores) contain the non-critical tasks. Run-time monitoring is used to throttle the memory requests issued by the interfering cores if more memory requests than allotted are issued. The focus of the work is to schedule the critical tasks while the impact on the non-critical tasks is minimized.

In another work by Yun et al. [101], the concept of a memory-performance critical section is introduced, where a task uses an API to indicate that it requires a high memory bandwidth and the OS regulates the other cores. This is designed to protect soft real-time applications and can result in heavy penalties on co-running applications.

Reserving one core for critical tasks may result in under-utilization if the critical tasks do not execute very often. Also, the approach is not suitable if there are more than one critical application. A more general approach

is proposed by Inam et al. [38], where a concept of multi-resource servers (MRSs) [10,40] is presented. The server maintains two different budgets; one for the CPU usage and one for the number of memory requests, but the memory throttling is proposed per server instead of per core, and several servers can execute on each core. They show that the MRS can limit the interference between applications running on different cores, but also that cache pollution affects timing properties of tasks. Thus, MRS should be complemented with for instance cache management.

Nowotsch et al. [69] propose a very similar approach for integrating temporal partitioning and WCET analysis. Their proposed approach extends the existing single-core techniques with determining the maximum number of shared resource requests per shared resource. Based on these estimates they introduce a new phase, the interference-delay analysis, to account for the additional interference caused by shared resources. The temporal partitioning is split into a monitoring and a suspension task where the monitoring task uses performance counters to track the resource usage of each process, and the suspension task is invoked once a process exhausts its (preallocated) capacity.

Other similar approaches are presented by Fernandez et al. [24], who introduce resource usage signatures and templates to abstract the contention caused and experienced by tasks on different cores. These signatures and templates are used to determine an execution time bound instead of the actual tasks; and by Yun et al. [105], who propose MemGuard, a memory bandwidth reservation system that provides bandwidth reservation for temporal isolation and a reclamation component.

Biondi et al. [12] present a framework for component-based design in multi-processor real-time systems under partitioned scheduling. They map components to virtual processors implemented through a reservation server, named M-BROE. During integration, the virtual processors are assigned to physical processors. The framework enables resource sharing among independently developed real-time applications. The resource access protocol makes use of FIFO non-preemptive spinlocks to ensure mutual exclusion among the processors.

The Single Core Equivalence (SCE) framework [59,85] combines several of the previously discussed approaches and consists of three parts: Colored Lockdown [58] for managing the shared cache; MemGuard [105] for monitoring and limiting the number of DRAM requests; and PALLOC [102] for DRAM bank partitioning. Starting from single-core WCET estimations, they add interference bounds resulting from shared resource usage on a multi-core platform to minimize the effects from other cores.

Having presented related works addressing the predictability and timing aspects, we will continue to review the research area of fault tolerance combined with timing predictability on multi-core.

Chapter 4

The Impacts of Faults on Timing

In this chapter, we review the state of research addressing the joint study of timing predictability and fault tolerance in the multi-core setting. The elementary fault classes of interest are dimension (hardware, software) and persistence (transient, permanent) [8]. We analyze proposed fault tolerance methods targeting these fault classes and whether their impact on WCET analysis is addressed. We also assess the applicability of the surveyed methods to safety-critical avionics systems on multi-core platforms.

4.1 Impact of Faults on WCET

Our survey is motivated by the requirement to handle both predictability and fault tolerance in avionics systems and the fact that aircraft is more exposed to cosmic radiation at cruising altitude. We know from previous chapters that predictability (WCET/WCRT) is affected by the introduction of multi-core processors and the same may be true for fault tolerance methods. Existing fault tolerance methods may incur a lot of inter-core interference that does not have to be considered on single-core processors.

Specifically, we seek answers to the following questions in a multi-core context:

1. Can we bound WCET estimates in presence of fault tolerance?
2. Can we validate fault tolerance and timeliness claims using fault injection?
3. What fault tolerance methods can be implemented with current COTS processors?

Reviewing the existing research in the area creates a base for understanding what are the building blocks from both perspectives (timing and fault

tolerance), and for making informed architectural decisions when allocating applications to nodes, partitions, cores, as well as affecting other resource boundaries (e.g., communication buses).

Others have surveyed related research areas previously. Mushtaq et al. [64] survey fault tolerance techniques for shared memory multi-core systems, which is similar to our work. However, they do not include the effects fault tolerance techniques have on WCET estimations. Natella et al. [68] have performed a survey of software fault injection (SFI) for systems executing on single-core processors. Their survey does not cover multi-core platforms nor the applicability of fault injection to verify WCET estimation claims, which we aim to investigate.

4.2 Hardware Error Detection and Correction

Depending on the system, there are different ways to handle the fault (or error). One way is to shut down the system until the fault can be repaired, but for some systems this is not an option. In such cases triple modular redundancy (TMR) can be used to handle faulty components if the probability for all three redundant components failing at the same time is considered low. Other solutions use a checkpoint and repair methodology, which periodically saves the execution state (creating a checkpoint) and when a fault is detected the execution is rolled back to the checkpoint. Fault masking is also a way of recovering from faults. TMR is an example of fault masking that uses three redundant components and majority voting to mask deviating data.

4.2.1 Faults in Cache

Caches occupy a large area of the processor chip and are vulnerable to soft errors. Therefore, to mitigate soft errors the caches are equipped with error detecting (and sometimes also error correcting) codes. The simplest detection type is parity where (typically) one extra bit is used per byte to indicate whether the number of 1-bits in the byte is even or odd. This can later be used to detect an error. An error-correction code (ECC) is required to correct errors and a common error correcting code is the single-error correction and double-error detection (SECDED) Hamming code [75].

Parity on Tags and Data

Using only parity may be enough for write-through data caches and instruction caches as no modified data exist. Upon detection of a parity error the processor may automatically invalidate the corresponding cache line and the data (or instructions) will be fetched again from the next level in the memory hierarchy, which could be the memory and thus result in additional memory requests. The processor could also generate an exception to be

handled by the system software. Either way, the fault mitigation will affect the execution time of the task and has to be considered.

Parity on Tags and ECC on Data

ECC on data will ensure that no modified data is corrupted by a single-bit transient fault. Parity errors will result in cache line invalidation. If the number of ECC single-bit errors exceed a definable threshold, this could be an indication of many latent parity errors and the address tags in the cache should be invalidated in order to reduce the probability of multi-bit tag errors. This preemptive error detection will also lead to invalidation of cache lines, which could result in additional memory requests and do affect inter-core interference and WCET/WCRT.

ECC on Tags and Data

For caches with ECC on both tags and data, no additional fault mitigation overhead exists. There are hardware implementations of ECC for caches that entail only a constant overhead for error detection and correction regardless of whether a fault is found or not. The fault-free operation will suffer some additional latency (in the order of one cycle).

4.2.2 Faults in DRAM

As DRAM contains data that is modified, only error detection and correction methods are of value and ECC is often used. A constant overhead for error detection and correction can be expected here in the same way as for ECC on cache tags and data.

One detail that may have to be considered is that the corrected data may only be delivered to the core and not written back to the memory. If this is the case, additional care has to be taken to determine if the period of time until a new value is written to the same memory location is acceptable compared to the risk of another single-bit error in the same data word (resulting in a multi-bit error that is not recoverable).

4.2.3 Faults in General-Purpose Registers

The general-purpose registers (plus many special purpose registers, configuration registers and other control logic) are not protected by parity or other similar means, and the amount of chip area used by such logic is increasing with the complexity of the chips [52]. These faults have to be mitigated using means such as instruction duplication [52] or task re-execution (see discussions in Section 4.3).

4.3 Fault Tolerance

In this section, we aim to investigate Question 1, that is WCET bounds in presence of faults tolerance. The question can be viewed from two different perspectives: (a) Fault tolerance methods can consider WCET aspects; and (b) Timing analysis methods can be fault aware. We treat each perspective separately. In Section 4.3.1 we consider (a) and (b) is considered in Section 4.3.2.

To validate claims about fault tolerance, fault injection methods are used, but the injection method is not a contribution of the papers in this section. Fault injection (related to Question 2) is surveyed in Section 4.4.

4.3.1 Fault Tolerance Methods

Here we look at fault tolerance methods considering multi-core systems in presence of transient and/or permanent hardware faults. Some also consider software faults.

Multi-core platforms can be used to enhance fault tolerance where redundant processes can execute on different cores. Shye et al. [86] present a software-centric paradigm in transient fault tolerance on multi-core platforms. They use several cores to deploy redundant processes of the original (single-threaded) application process and then compare the results to detect transient faults. One master process is replicated several times to create the redundant slave processes (replicas). Since all processes execute the same instructions, care must be taken to ensure that any system state that is modified is only modified once as if the original process is executing by itself (preserving the semantics of the master process). This has been solved by a system call emulation unit that is inserted (using the LD_PRELOAD feature of Linux) between the process and the operating system. The emulation unit intercepts the application start (to create redundant processes as well as the original process) and the system calls, and ensures that system calls that return non-deterministic data (such as time of day) will be handled in a way that all replicas see the same value. This method works well on systems where throughput is not the primary goal. To be able to both detect and recover from transient faults, at least three redundant processes (the original process and two replicas) are needed.

Shye et al. [86] only cover single-threaded applications. Multi-threaded applications are addressed by Mushtaq et al. [65] by introducing a record-and-replay approach to make multi-threaded shared memory requests deterministic in addition to the use of redundant processes. The order of all shared memory requests performed by the original process is recorded and later replayed by the replicas. Recovery is handled by checkpointing and rollback. This method requires communication between the original process and the replicas, and the memory used for the communication can also become corrupted, which makes it less reliable. Therefore, Mushtaq et al. extend their work and introduce deterministic multi-threading [66]

instead of record/replay. The definition of deterministic multi-threading is that given the same input, the threads of a multi-threaded process always have the same lock interleaving. To ensure that the locks are acquired in the same order, they introduce logical clocks that are inserted at compile-time. When a thread is trying to acquire a lock, it is only allowed to do so if its logical clock has the minimal value. A number of optimizations are introduced to the logical clock handling to reduce the overhead. The fault tolerance method adds an average overhead of 49 percent to the execution time in absence of faults.

The fault tolerance methods discussed so far use homogeneous multi-cores with identical cores. Meisner and Platzner [61] investigate using a hybrid multi-core, which combines “normal” cores with cores that are implemented in a field-programmable gate array (FPGA). They propose a dynamic redundancy technique, named thread shadowing, which duplicates (shadows) a software or hardware thread during a time period. One advantage of using hybrid multi-cores is that a software thread can shadow a hardware thread or the other way around (which is called trans-modal). Hardware threads are often much faster than software threads and this can be utilized in a trans-modal round-robin shadowing (i.e., one hardware thread shadows a number of software threads for a time period one at a time). This requires only one extra core, but does not provide a continuous error detection. It is also possible to increase the number of cores to shadow each thread and provide a more comprehensive error detection, which is more suitable for transient faults. The former configuration can be used to detect permanent hardware faults.

Thread duplication and majority voting is suitable for masking transient faults, but it is more difficult to detect permanent faults. The question is also what to do when identifying a permanent fault. Peshave et al. [74] use a reconfigurable Chip Multi-Core Processor (CMP) to provide redundancy in order to improve reliability. They mask transient faults and tolerate core-level permanent faults. The framework consists of a TMR system that uses dual-core CMPs, where one of the cores is deactivated and used as a redundant core in case of faults in the running one. Each of the three CMPs execute a copy of the application and the output from each is used in a voting procedure to produce the final result. The voter is included in a separate hardware block that monitors the system and can for instance switch to the redundant core on any CMP if a permanent fault is detected. This system is compared to a standard TMR system using single-core processors and they conclude that the dual-core system can tolerate more core-level permanent and transient faults than the single-core based system.

Even though the semiconductor technology evolution resulting in smaller and smaller transistors increases the sensitivity to both permanent and transient faults, the fault-free operation is still the common case. Hari et al. [33] focus therefore on light-weight detection mechanisms, while the relative rare operation of fault diagnosis requires more capacity and is expensive. They

develop a diagnosis algorithm for multi-threaded applications on multi-core systems. The algorithm is based on repeated rollback/replay and can deterministically replay execution from a previous checkpoint for a multi-threaded application, which is a requirement for a proper diagnosis. Several replays may be necessary to distinguish between software bugs, transient and permanent hardware faults and to identify the faulty core in case of a permanent fault.

The majority of transient faults will be spatial multi-bit faults as the technology scaling continues towards smaller and smaller feature sizes [60]. A spatial multi-bit fault is a single-event upset (SEU) affecting more than one bit. If the affected bits belong to the same protection domain, the common single-error correction and double-error detection (SECDED) error correction code cannot be used for correction as it only corrects single-bit faults. Therefore, Manoochchri and Dubois [60] develop a formal model (PARMA+) to benchmark failure rates of caches with spatial multi-bit faults and different protection schemas. They focus on the cache because the caches occupy a large area on the processor chip and are vulnerable to soft errors, which will affect reliability. PARMA+ estimates the cache failures in time (FIT) rate. The FIT rate of a device is the number of failures that can be expected in one billion (10^9) device-hours of operation (e.g., 1000 devices for 1 million hours, or 1 million devices for 1000 hours each, or some other combination). A component having a failure rate of 1 FIT is then equivalent to having a mean time between failures (MTBF) of 1 billion hours. Most components have failure rates measured in hundreds or thousands of FITs. This is not really a fault tolerance method, but the model can help chip designers to configure reliability enhancing protection schemes in a more optimal way, resulting in more reliable components.

The fault tolerance mechanisms described so far are all designed to handle random sporadic faults (and permanent faults in caches), but faults can also be used to attack a system. In this case, the faults are injected by an adversary in a well-planned manner. Hence, the fault-tolerant mechanisms discussed so far are not enough for fault-based attack resistance. Yuce et al. [100] propose FAME (Fault-attack Aware Microprocessor Extensions), a hardware-based fault detection that is continuously monitoring the system and a software-based fault response mechanism (software trap handler) that is invoked when a fault is detected. A fault control unit (FCU) collects information to ensure that a recovery is possible from the software trap handler. The FCU also flushes the processor pipeline and disables write operations to the memory and registers to ensure that no faulty results are committed to the processor state and the software trap handler is restarted if the Fault Detection Unit (FDU) detects a fault during execution of the trap handler. This ensures the completion of the trap handler before resuming normal execution.

4.3.2 Fault-Aware Timing Analysis

The fault tolerance methods presented in the previous section do not consider the impact they have on the execution time and perform no timing analyses. The focus is on delivering the correct function and none of the methods can be used (without alterations) to answer Question 1. Many embedded systems will have strict requirements on the WCET of different functions, hence fault tolerance cannot ignore this impact on WCET and schedulability. These systems are also often safety-critical and contain functions of mixed criticalities. We proceed to review fault-aware timing analysis methods.

The fault-tolerant mixed-criticality (FTMC) scheduling algorithm for single-core systems proposed by Pathan [71] includes a fault tolerance perspective by using execution of backup tasks if faults are detected. Each task (both original and backup) has different WCETs on the different criticality levels (in this case LO, HI). The frequency of faults during a fixed time period is also considered for LO and HI criticality modes. The system will switch from LO to HI mode when either a task exceeds its LO-criticality WCET or the number of errors exceeds its threshold. Once the system has switched to HI-criticality mode all LO-criticality tasks are dropped. If any task errors are detected, a backup is executed and this could be re-execution of the original task (to handle transient hardware-faults) or execution of a diverse implementation (to handle potential software bugs). The error detection mechanism is assumed to be present in the platform already.

A similar method is proposed by Huang et al. [36] for handling transient hardware-faults, but instead of dropping less critical tasks when switching to HI-criticality mode the service can be degraded (e.g., by changing the inter-arrival time of these tasks). Safety requirements are introduced for each criticality level based on the probability-of-failure-per-hour metric (such as those in Table 2.1) and a re-execution profile is defined for each task to ensure it meets the safety requirement. For the HI-criticality tasks a killing profile is defined specifying the number of re-executions that are allowed before LO-criticality tasks are dropped. The impact of task re-execution, task dropping and service degradation are thus bounded.

The above methods present scheduling algorithms that consider fault tolerance and WCET in a single-core setting. Kang et al. [44] apply the concepts of mixed-criticality and reliability to a multi-processor system-on-chip (MPSoC) using the standard model with two criticality levels (normal and critical state). Tasks are mapped to processing elements (PEs) and then locally scheduled. Re-execution and replication is used for reliability and droppable tasks are dropped when switching to the critical state. The dropped tasks are allowed to execute again at the new hyperperiod once the system is switched back to normal mode. So, in this case the critical state is just a temporary state to cope with the faults without risking non-droppable tasks missing their deadline. The focus here is on bounding the worst-case response time (WCRT) by using static mapping and optimizations.

By integrating fault tolerance in the methods above, the transition to the critical state is performed for both detected faults and deadline misses even though different handling of the two may be more suitable. To overcome this and to improve the quality of service for the LO-criticality tasks, a four-mode model is proposed by Al-bayati et al. [3]. In addition to the standard two modes (LO, HI) the *transient fault* (TF) and the *overrun* (OV) modes are introduced. The TF mode will be transitioned to when a transient fault is detected and task re-execution is needed, and the OV is transitioned to when a task misses its deadline. The HI mode is used to cover the cases where both overrun and a transient fault are detected. LO-criticality tasks are dropped when the system enters OV or TF. Similar to Huang et al. [36] not all LO-criticality tasks are dropped, but they try to maximize the number of LO-criticality tasks that can run in each mode without affecting the schedulability of the HI-criticality tasks.

Pathan also presents a global scheduling algorithm (FTM) for real-time sporadic tasks on multi-core platforms [72], but with no focus on mixed-criticality (this is left as future work). The algorithm considers a combination of active and passive redundancy, where active backups are executed in parallel on a different core from the primary task and passive backups are executed in sequence. Both permanent and transient hardware faults are considered. The application-level model considers errors to be detectable in both the primary task and in the backups. The stochastic behavior of the actual fault model is inserted later, which results in a probabilistic analysis capable of assessing an application's ability to tolerate faults during run-time. A heuristic to help the designers configure the number of active backups is also presented.

WCET estimation is a difficult task and most of the WCET analysis methods assume a fault-free execution, but this may no longer be an acceptable assumption as mentioned before. As a result, the probability of failures increases for circuits and assuming that components are fault-free may lead to an underestimation of WCET. Höfig [35] propose a static probabilistic timing analysis (SPTA) approach considering faulty sensors. The Failure-Dependent Timing Analysis (FDTA) is based on tools such as *Enterprise Architect* (EA), *Simulink* and *aiT*. Simulink is used to model the system and the model is imported into EA where a safety analysis is performed to generate a fault tree. The failure modes resulting from this analysis is used to generate a new Simulink model for each mode in which a specific fault has been inserted. Code is generated and compiled for each of the fault-injected models and aiT is used to statically estimate the WCET. A probability is calculated for each of the resulting WCET estimates providing a probabilistic guarantee that the deadline miss ratio of a task is below a given threshold.

With the shrinking sizes of components, the number of permanent faults increases and caches taking up a large area in the processor will be a non-negligible source of performance degradation. Slijepcevic et al. [88]

present a measurement-based probabilistic timing analysis (MBPTA) approach for faulty caches. The method requires a hardware platform that is compliant with a probabilistic timing analysis (e.g., caches with random (re)placement). It also requires a mechanism to disable a cache line once a permanent fault is detected in that line. Execution times are measured on hardware with maximum degradation. Therefore, they introduce Degraded Test Mode (DTM) that allows a number of cache lines to be disabled. Using DTM, it is possible to get measurements for a faulty cache on a fault-free processor so that pWCET estimates are safe. Slijepcevic et al. [89] continue by also handling transient faults and the timing impact of error detection, correction, diagnosis and reconfiguration. A lot of requirements are placed on the hardware, but they claim the result is a tight pWCET.

Chen et al. [16] perform static probabilistic timing analysis (SPTA) on instruction caches with random replacement on single-cores considering both permanent and transient hardware faults. Memory traces for single-path programs are used as input and the probability of exceeding a certain execution time is computed using state space techniques based on a non-homogeneous Markov chain model. The result of the Markov model is a timing analysis taking all cache states into account. However, the number of states grows polynomially with high exponent values as more and more memory addresses are requested. To overcome this, they introduce a method that limits the number of states by using a lower number of memory addresses for the states.

The method used by Hardy and Puaut [31] is quite similar to what Slijepcevic et al. [88] present, but not as many requirements are posed on the hardware. Hardy and Puaut [31] present an SPTA-based approach for instruction caches with least recently used (LRU) replacement. They perform a low-level static analysis of the cache using abstract interpretation and a high-level WCET analysis using an integer linear programming (ILP) technique (Implicit Path Enumeration Technique (IPET)). Cache sets are independent, which means it is not necessary to explore all faulty cache configurations to obtain the penalty probability distribution, but rather compute the convolution of the set's probability distributions. They compute fault-free WCET and then derive an upper bound of the time penalties caused by fault-induced misses. The evaluation performed shows that for a given probability of a static random-access memory (SRAM) cell failure the pWCET estimates are significantly larger than the fault-free WCETs.

Hardy et al. [32] introduce two hardware based mitigation mechanisms, *Reliable Way (RW)* and *Shared Reliable Buffer (SRB)*. RW introduces a permanent fault resilient way for each cache set to capture the spatial locality of memory requests, which otherwise would be missed as all those requests would end up in an entirely faulty cache set. RW ensures that the cache performance is not worse than a direct-mapped cache with a size equal to the number of sets. SRB is also introduced to mitigate the increase in cache misses when a whole set is faulty, but at a lower hardware cost. The price

for the lower cost is a performance that may be worse than RW when there are faults. The WCET estimation is adapted to using RW or SRB and their experimental evaluation shows a gain in pWCET of 48 percent and 40 percent for RW and SRB respectively, compared to their previous work [31].

4.4 Fault Injection

To verify fault tolerance mechanisms such as those reviewed in Section 4.3, one can use automated fault-injection tools to speed up the otherwise often time-consuming verification and validation. Most fault-injection tools focus on a particular type of fault (hardware or software fault) and on a particular component such as cache or memory, which occupy large parts of the chip area and are thereby extra vulnerable to faults. To complement the survey on software fault injection for single-core platforms by Natella et al. [68], we select representative papers addressing fault injection on multi-core platforms. We are not only interested in verifying the fault tolerance mechanism, but also timeliness (Question 2).

4.4.1 Hardware Fault Injection

Most modern processors include capabilities for detecting and reporting errors in most processor units. Lanzaro et al. [50] use this mechanism in Core i7 from Intel to emulate machine check errors (e.g., cache, memory controller and interconnection errors) by writing to registers associated with the error-reporting capability. This is an easy and lightweight fault injection method, but requires support for writing to these error-reporting registers.

Wulf et al. [98] present a software-based fault-injection tool, SPFI-TILE, which emulates single-bit hardware faults in registers or memory on the many-core TILE64 using a debugger (GDB). They also present a data cache fault-injection extension called Smooth Cache Injection Per Skipping (SCIPS), which distributes fault-injection probabilities evenly over all cache locations. As the cache is not directly accessible from software, they emulate faulty cache data by using the debugger to halt the selected tile (core) at a fault-injection point, step to the next load instruction, inject a fault in the correct memory address, and then continue the execution to let the load instruction finish with the faulty data. SCIPS is used to balance the fault-injection probabilities by randomly skipping several load instructions instead of injecting into the first load instruction after the location where the debugger halts the processor.

By using the *fork()* and *ptrace()* system calls and operating system signals (e.g., SIGSTOP and SIGCONT) Vargas et al. [91] develop a fault-injection tool that is hardware-independent (but not operating system-independent) and can inject faults into general purpose registers, some selected special purpose registers and in memory regions. The parent process (after the *fork()* call) is used as the fault injector and the child process executes the

application under analysis. This is actually similar to what GDB does under the hood (used by Wulf et al. [98]). Multi-threaded (pthreads or OpenMP) applications are supported as the fault injector queries the operating system for the number of threads in the child process and their IDs.

Software-based fault injection is easy to use and portable, but cannot be used to inject faults into parts that are not accessible from software. A full-system simulator can expose the internal state of the processor, which simplifies fault injection and no modifications to the device under test is required. Carlisle et al. [13] use the Simics full-system simulator to develop DrSEUs (Dynamic robust Single-Event Upset simulator), which is used to simulate single-event upsets (SEU) and single-event functional interrupts (SEFI). They use the checkpointing capability of Simics to inject bit-flips into any of the components in the processor (e.g., general-purpose registers, program counter, Ethernet controller registers and translation-lookaside buffer entries). Caches and main memory are not targeted by DrSEUs. The modified checkpoints can be loaded in Simics and the simulation can continue with faults injected. Checkpoints taken after the fault-injection can be compared to the fault-free checkpoints (gold checkpoints) created before the injection campaign starts.

Another simulation-based fault-injection framework called OVPSim-FIM is presented by Rosa et al. [78]. It also uses the concept of golden execution and checkpoints (like DrSEUs [13]), which are created by executing the application on the original OVPSim. Faults (single bit-flips) are injected at a random time, in a random component (registers or memory) and then the simulation continues. Checkpoints can be used to reduce the amount of re-executed code and speed up the simulation.

4.4.2 Software Fault Injection

SFI is often used to validate fault tolerance mechanisms in systems. Natella and Cotroneo [67] investigate whether SFI really does emulate mandelbugs (transient software faults) to a satisfactory degree. They perform a case study and analyze the SFI tool G-SWFIT, finding that the injected faults do not represent mandelbugs that well. This is because the injected faults are activated early in the execution phase and all process replicas are affected in a deterministic way.

Natella et al. [68] have also performed a thorough survey of software fault injection (SFI) for systems executing on single-core processors. They analyze a number of methods and tools and discuss several important aspects of SFI, such as how well the injected faults represent real faults, how efficiently the experiments can be performed and how usable the methods and tools are.

4.5 Discussion

In our pursuit of answers to Question 1 and 2 we have reviewed several state-of-the-art methods for dealing with fault tolerance in presence (or fault injection) of hardware faults, and in some cases software faults. In this section, we compare and evaluate these methods with regard to our posed questions.

Table 4.1 summarizes the surveyed papers and their respective focuses. Column two clarifies whether the method has a pronounced focus on critical systems (e.g., avionic, automotive, space, train or medical systems) as these may be a better starting point. We assume the generic application domain does not have any strict timing requirements (i.e., the main goal is performing the computations correctly). In the third column, the main focus of the paper is presented, whether the method is applicable for designing or verifying (validating) a system. The fourth column specifies the type of faults that are considered and in the last column we indicate whether any impact on timing analysis is considered.

The following aspects will be considered for the evaluation.

Portability to hard real-time systems We look at the facilities used from the underlying hardware and operating system and how (if) they can be ported to a constrained environment (RTOS and design paradigms). As an example, safety-critical systems rarely allow dynamic memory allocation. (Question 3)

Inter-core interference When running mixed-criticality systems the temporal partitioning is fundamental. No function should be able to affect the execution of another. In a multi-core platform with shared resources multiple cores can access the same shared resource in parallel, which could affect the execution time of the applications on the cores. (Question 1 and 2)

WCET Safety-critical systems have to perform the intended function in a timely fashion even under faulty conditions so fault tolerance should consider WCET impacts (or WCET estimates should consider faults). (Question 1 and 2)

We start by looking at the execution environment for the methods and whether they depend on specific features of either the hardware or the operating system the applications execute on.

Several of the proposed fault tolerance methods are implemented on Linux and rely on its features to alter the running process by preloading (*LD_PRELOAD*) dynamic shared libraries to extend system call functionality and override other shared library functions. Another example is *fork()*, which is used to dynamically create an identical copy of a process [65,66]. Process-level redundancy (PLR) as proposed by Shye et al. [86] uses Intel

Table 4.1: Classification of Surveyed Papers.

Paper	Application Domain		Fault Tolerance Injection		Hardware Faults		Software Faults		Addresses Timing Analysis (WCET)
	Critical ¹	Generic			Transient	Permanent			
Shye [86]		✓	✓		✓				
Mushtaq [65]		✓	✓		✓				
Mushtaq [66]		✓	✓		✓				
Meisner [61]		✓	✓		✓	✓			
Peshave [74]	✓		✓		✓	✓			
Hari [33]		✓	✓		✓	✓	✓		
Kang [44]	✓		✓		✓				✓
Al-bayati [3]	✓		✓		✓				✓
Sljepcevic [89]	✓		✓		✓	✓			✓
Pathan [72]	✓		✓		✓	✓	✓		✓
Lanzaro [50]		✓		✓	✓				
Wulf [98]		✓		✓	✓				
Vargas [91]		✓		✓	✓				
Carlisle [13]		✓		✓	✓				
Rosa [78]		✓		✓	✓				
Mushtaq [64] ²		✓	✓		✓	✓	✓		
Manoochehri [60]		✓	✓		✓				
Pathan [71]	✓		✓		✓		✓		✓
Huang [36]	✓		✓		✓				✓
Höfig [35]	✓		✓		✓	✓			✓
Sljepcevic [88]	✓		✓		✓	✓			✓
Chen [16]		✓	✓		✓	✓			
Hardy [31]	✓		✓		✓	✓			✓
Hardy [32]	✓		✓		✓	✓			✓
Yuce [100]		✓	✓		✓				
Natella [67]		✓		✓			✓		
Natella [68] ²		✓		✓			✓		

¹ Avionics, Automotive, Space, Trains or Medical Systems² Survey

Pin for dynamic code patching on Linux. The methods are easily implemented using the support of the OS, but may be hard to port to an RTOS designed for safety-critical systems, where dynamic features are often disallowed. Linux is recognized as not being suitable for RTCA/DO-178C [81] DAL C and higher in safety-critical avionics systems, where the OS needs to be certified to the same DAL as the most critical application running in the system.

Simulators are often used to demonstrate the fault tolerance methods [16, 33,88,89] because the existing COTS platforms do not support the controllability or observability required for the method to work. For some methods, where the required hardware support does not exist, additional hardware modifications are required for the method to be implementable. This makes it very difficult to deploy the methods in a real-world system as it may be hard to get the chip manufacturers to implement the needed features into their processors unless there is a big demand for these features. At least the avionic industry is quite small compared to the consumer market with general-purpose computers, mobile phones and tablets. The methods using additional hardware outside of the system-on-chip (SoC) have bigger potential of actually making it to the industry. FPGA implementations together with existing COTS components may be a way forward.

The most common surveyed method for detecting and recovering from faults is process redundancy or re-execution [44,61,64–66,86]. Redundancy is particularly relevant when it comes to multi-core platforms, since plenty of resources exist for execution. However, re-execution can be problematic on multi-cores due to inter-core interference when accessing shared resources, which will lead to a variability of the request time as the cores contend for the shared resource.

WCET estimation for a task in a multi-core system is increasingly difficult due to the dependence of what is executing on the other cores. Methods (e.g., by us as presented in Chapter 6 and in works discussed in Chapter 3) have been proposed that will allow WCET to be estimated for each task in isolation and then determining the maximum interference the task can be subjected to. In the integration phase where all tasks on all cores are brought together the final WCET (or WCRT) can be determined and schedulability tests can be performed. The works considering schedulability [3,36,44,71,72] all assume that WCET estimates have been derived and are ready to be used. The response times estimated in these works have to be interference-aware as the fault tolerance method used is re-execution of a task, which may introduce additional memory requests that interfere with or are interfered with by tasks on other cores resulting in a variability of WCRT.

Most of the works focus on functional fault tolerance with no regard to execution (or response) time. The fault tolerance methods surveyed will result in additional execution time and memory requests, which could lead to unwanted (and unaccounted for) interference between tasks on different cores. Delays for memory requests are modeled with a constant value in the

works considering WCET estimates [16,31,32,35,88,89], which is not accurate considering the inter-core interference and the resulting timing variability for memory requests on a shared-memory multi-core system.

The methodology we present in Chapter 6 does consider this timing variability of shared memory requests. To ensure an upper bound on the interference processes may experience, the memory requests issued from processes are monitored and the process is suspended when the per-process allotted number of requests is exceeded. However, we also assume a fault-free execution and leave the fault tolerance aspect as future work.

Related to the inter-core interference is the recovery method used for some caches, where the cache line in a write-through cache is automatically invalidated when a fault is detected. This also applies to instruction caches as no modified data exist. This may result in additional memory requests and thus increase the inter-core interference. This has not been considered in any work. Chen et al. [16] identify that faults in the instruction cache result in new memory requests to fetch the instructions again, but as the method is developed for a single-core processor the application to multi-cores is not well-understood.

Triple Modular Redundancy has no effect on WCET as the three units are executing independently and the voter logic has a well-defined delay, but it is expensive in terms of hardware as complete hardware components have to be tripled. For some systems, this may still be the only viable option. Using redundant processes on the other hand does affect the WCET since the processes will execute concurrently and will access the same shared resources as mentioned before. Memory-intensive applications suffer from more overhead than CPU-intensive applications due to the larger cache miss rate and the subsequent larger number of memory requests [86].

No fault tolerance method targets all features (e.g., register, cache, interconnect) of a processor. When determining, which features to protect, the safety assessment process has to estimate the probabilities of faults and compare them to the probabilistic safety guarantees required by the criticality level (DAL in RTCA/DO-178C) of the software. The interconnect that is present in many multi-core platforms is often part of the manufacturer's intellectual property and details are seldom disclosed, making it difficult to assess its fault tolerance capabilities.

The use of probabilistic methods [16,31,32,35,88,89] does sound interesting and is worth a more in-depth study in the domain of safety-critical avionics systems. The hardware requirement with random caches and buses may be an obstacle. The avionic industry is currently focused on platforms with PowerPC processors and as far as we know they do not support this. If the gain is big enough a change of processor family may be possible, but the avionic industry is slow moving with long-lived projects.

Simulators can also be used for fault injection [13,78], making it possible to inject faults in basically every component of a processor. Simulations of complex processors can be very slow, offering a best-case simulation perfor-

mance of 2-3 MIPS (33 injections per second) [78].

Much of the research efforts in the software fault injection area are looking at the problem of producing representative software faults, that is, focusing on what to inject, where to inject it, and how to inject it. Software-based fault injection [50,91,98] affects the execution time and thus it may be hard to verify WCET estimates in presence of faults.

No work has been found that investigates software fault injection in multi-core systems, which is a bit surprising. Additional software faults when single-core tested software is migrated to multi-core can arise since new race conditions or lack of locks can show up in interference scenarios. For instance, in a single-core priority-based preemptive system, the software developer can often assume that a high-priority and a low-priority task will not access the memory simultaneously, since the high-priority task will preempt the low-priority task. This may lead to applications failing to use a lock to properly synchronize access to the memory. In a multi-core system, both of these tasks can run in parallel, resulting in simultaneous access to the memory with unpredictable consequences. Other problems may exist due to synchronization mechanisms that work well on a single-core system, but lead to problems that surface only in a multi-core system.

Software faults that do not show in every execution, sometimes referred to as mandelbugs, where the effect is transient in nature are not well represented by existing SFI tools [67]. This type of bugs is hard to find during design time and pre-production testing. Thus, most of the bugs found in an operational system are classified as mandelbugs.

We have found several interesting aspects in need of more research. In order to understand how fault tolerance methods affect the timing predictability (e.g., the inter-core interference), we will first investigate fault-free timing predictability issues.

Chapter 5

Memory Request Monitoring

In this chapter, we look at resource monitoring systems, and shared memory in particular, to understand the impact of the RTOS on the number of memory requests issued from a core. So, in addition to the memory requests issued by the applications running on the different cores, we also consider memory requests originating from the RTOS.

5.1 Motivation

None of the resource monitoring systems discussed in Chapter 3 consider the effects of memory requests issued by the RTOS(es) running on the cores and many of them use Linux as the evaluation platform together with Linux-specific features.

However, Linux is recognized as not being suitable for RTCA/DO-178C [81] DAL C and higher in safety-critical avionics systems, where the OS needs to be certified to the same DAL as the most critical application running in the system. This means that any effects the OS may have on the response time of applications cannot be captured accurately on such a platform.

For instance, system calls, memory management for space partitioning and task switching could all introduce an overhead in the number of memory requests issued from a particular core. These extra memory requests could interfere with memory requests issued by other cores, and thus affect the WCET of those applications.

In order to illustrate the impact of memory requests from the RTOS on the execution of an application on another core, consider a scenario where two applications execute on one core each. A memory request monitoring system is in place and each application has been assigned a memory request budget (M) and a CPU usage budget (D, since it is also the deadline). Both

budgets are replenished periodically and when either budget is depleted the application is suspended until the start of the next period. The derivation of the memory request budget is based on the memory access pattern of each application in isolation and can be performed with static analysis or based on measurements.

The effects of RTOS memory requests are shown in Figure 5.1, where the run-time monitoring counts all requests issued from the core (i.e., including requests from the RTOS); and Figure 5.2, where the run-time monitoring only counts the requests issued by the application. The dashed line represents the (off-line) estimated number of memory requests, while the solid line is the actual number of memory requests issued by the core.

First, we consider the case where all requests are counted (Figure 5.1). On core 1, the run-time monitoring counts all requests and the budget (M) is actually depleted before the task has issued all of its intended requests. The task will be suspended and may miss its deadline before the intended function has been performed. The task on core 2, on the other hand, will thus experience less interference and may finish earlier (not a problem).

Next, we instead consider the case where only the application requests are counted (Figure 5.2). Here, the total number of requests from core 1 will exceed the budget (M) without being detected by the run-time monitoring. This will introduce additional interference with requests issued by core 2, where the task may miss its deadline (D) due to this interference and the impact it has on the memory access time.

Figure 5.1: Effects of RTOS memory requests when counted by the run-time monitoring system.

We aim to evaluate the hypothesis that RTOS memory requests on one core could interfere with applications executing on other cores when memory budgets have been determined only using knowledge about the application behaviors.

Figure 5.2: Effects of RTOS memory requests when not counted by the run-time monitoring system.

5.2 Methodology

We use four applications, described in Section 5.2.2, with different memory request characteristics to evaluate the relation between the number of memory requests issued by the applications and the number of memory requests issued by the RTOS. In addition, we also measure the number of memory requests issued by the RTOS during partition (re)scheduling.

The applications are run for approximately 30 minutes, the length of a typical mission, after which the applications output the collected data.

5.2.1 Hardware Platform

The T4240¹ belongs to the state-of-the-art NXP QorIQ T series. The T4240 consists of 12 Power Architecture®e6500 cores. Each core supports two hardware threads, which software views as two virtual CPUs, and thus rendering a total of 24 (virtual) cores. The e6500 cores are clustered in banks of four cores sharing 2 MB level 2 (L2) cache, but with private 32 KB L1 data and instruction caches.

The e6500 core provides three privilege levels to provide different levels of protection for software to operate under. This allows for building systems that provide partitioning and virtualization. Hypervisor mode is the highest privilege level; it allows access to all instructions and resources, the supervisor mode is the next level; it has access to several privileged instructions, and user mode is the unprivileged and lowest level; this is where most normal applications run.

To implement the space and time partitioning of ARINC 653, the RTOS can execute in hypervisor or supervisor mode and the applications execute in user mode. In the rest of this chapter we will use the term supervisor

¹<http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-platforms/t-series/qoriq-t4240-t4160-t4080-multicore-communications-processors:T4240>

mode for the mode in which the RTOS is executing, regardless of whether it executes in supervisor or hypervisor mode.

5.2.2 Software Platform

We use a proprietary ARINC 653 compliant research RTOS, extended by us as described later in this section. The system is run in an asymmetric multiprocessing (AMP) configuration, meaning separate RTOS instances on each core.

Each application is implemented in a partition separate from the other applications, and the partitions are scheduled to execute on separate cores. Thus, we have one partition running on each of the four cores.

A current single-core partition schedule would normally consist of more than one partition and one partition window, typically 6-10 partitions, in a major frame. In our case, we would schedule all four partitions within a major frame. In this chapter, the partition schedule for each core consists of one single partition window, since we only have one partition on each core. The partitions execute in 60 Hz, which is a commonly used frequency in avionics. The partition window duration, and also the major frame time, is thus 16.67 ms. Even though there is only one partition window in each schedule, the partition scheduling is still activated after the partition window duration when starting a new major frame.

Applications

The applications have been selected as they exhibit characteristics representative for avionic applications. From Figure 5.3 in Section 5.3, we can also see that the applications exhibit quite different memory request characteristics. The applications are contained within one ARINC 653 partition each.

Nav This partition consists of two periodic processes. The first, runs in 60 Hz and implements a navigation algorithm for an unmanned aerial vehicle (UAV). The second process, runs in 1 Hz and simulates a Global Positioning System (GPS) device (by keeping a static array of coordinates). Every second a new GPS coordinate is made available from the GPS process to the navigation process, which in turn processes the position and takes the appropriate action in navigating to the destination.

The GPS path consists of 76 points, going from point A to point B and back again. There are four waypoints along the path, two going from A to B and two going from B to A. It can be seen as a surveillance mission for a UAV, going back and forth between the two end points.

Mult This partition consists of one periodic process, which performs a multiplication of two 1000x1000 matrices. One complete matrix multiplication is performed each period.

Cubic This partition consists of one periodic process, which is based on the basic math test from the Automotive and Industrial Control category of MiBench [30]. It performs simple mathematical calculations such as cubic function solving, integer square root and angle conversions from degrees to radians, which are all calculations needed for calculating for instance airspeed or other avionics related values. The input data for these calculations is a fixed set of constants.

It takes several partition windows to complete the entire set of calculations.

Image This partition consists of one periodic process, which is based on the Susan test from the Automotive and Industrial Control category of MiBench [30]. It is an image recognition package including edge and corner detection, smoothing and adjustments for threshold, brightness, and spatial control. The input data is a black and white image of a rectangle.

For each period the process alternates between the available image recognition operations, and it takes several periods to complete all of them.

RTOS Extensions

The RTOS is proprietary software, available for the PowerPC e500, e500mc and e6500 architectures, and also for the ARMv7-A architecture.

The monitoring is implemented using the Performance Monitor Counters (PMCs) that exist in the e6500 core. The cores provide the ability to count L2 instruction misses and L2 data misses per core. Six performance counters are set up, two for counting data and instruction misses in supervisor mode, two for counting data and instruction misses in user mode, and the last two for counting data and instruction misses in supervisor mode when the Performance Monitor Mark (PMM) flag is set in the Machine Status Register. The RTOS is also extended to set the PMM flag under the partition switch interval while scheduling. By setting the PMM flag during partition scheduling and setting the last two counters to only count when PMM is set, we can separate the memory requests issued in supervisor mode during a partition window and the memory requests issued during partition scheduling, which are also performed in supervisor mode.

On each core, the PMCs will count each instruction and data miss to the L2-cache in each partition window. During partition scheduling the RTOS will store the values to a buffer and the counters are set to zero. The buffers are available to the partitions in user mode.

5.3 Measurement Results

In this section, we evaluate the memory requests issued from the described applications and also from the RTOS.

Figure 5.3 shows that the average number of memory requests from the four selected applications, each running in its own partition, differ by orders of magnitude. Also, the standard deviation, shown as error bars, varies greatly between the partitions. Mult, for instance, has a very low standard deviation, meaning its memory request pattern is deterministic from one partition window to another. Cubic on the other hand has a more diverse memory request pattern between partition windows. This indicates that the impact of the RTOS memory requests will depend on the memory request characteristics of the partitions. A high number of requests from the partition will mask the requests from the RTOS as they will constitute a smaller percentage of all memory requests issued from that core.

Figure 5.3: Memory request characteristics for included applications.

Figure 5.4a and Figure 5.4c show the memory requests per partition window during the experiment for Nav and Cubic respectively. These two are illustrated here as they represent two distinct characteristics. For Nav, the supervisor mode requests are of the same magnitude as the user mode requests, while for Cubic the user mode requests are orders of magnitude larger. The underlying reason for the large standard deviation for Cubic in Figure 5.3 shows up in Figure 5.4c through the order of magnitude difference in the number of requests between partition windows. The compact band of request numbers in the range just below 10^4 requests represents a kind of

dominant behavior in some partition windows, and the thinner bands just below 10^5 requests represent another type of behavior exhibited in fewer number of partition windows.

Mult and Image show a request pattern similar to Cubic, but the user mode memory requests are magnitudes larger than the supervisor mode requests as can be seen in Figure 5.4b and Figure 5.4d.

Figure 5.4: Memory Requests per partition window.

Both supervisor mode and partition scheduling (Reschedule in the figures) requests are fairly similar for all partitions, as shown in Figure 5.5. This figure displays the total number of memory requests monitored in all partition windows during the experiment. The magnitude of the supervisor mode memory requests depends on the behavior of the application running in that partition (e.g., the usage of system calls and is thereby different for different applications). The partition scheduling requests, on the other hand, are independent of application behavior, and these memory requests are a constant overhead regardless of whether the partitions on the core execute or not. However, the fact that stable numbers are exhibited displays a degree of determinism in this function of the RTOS.

For Nav, the supervisor mode actually contributes more memory requests

than the application itself. There are more than twice as many memory requests compared to only using application behavior. This clearly illustrates that had a budget been set on the memory requests solely dependent on the application behavior, the budget would have been far too tight, and the application operation disrupted at run-time far too often.

Figure 5.5: Total number of memory requests.

Table 5.1 summarizes these observations. For the memory-intensive Mult and Image, the supervisor mode requests account for less than 1 percent. For Nav, on the other hand, being more computationally intensive and consisting of two processes (resulting in more supervisor mode requests due to process scheduling overhead), the supervisor mode requests account for almost 50 percent of the total number of memory requests. Note that even though the percentage is low for three out of the four applications, the extra memory requests in supervisor and reschedule modes could still have an actual impact on the worst-case response time of the applications on other cores.

Table 5.1: Distribution of total number of memory requests.

Partition	Supervisor Mode	User Mode	Reschedule
Nav	48.3%	30.9%	20.8%
Mult	0.07%	99.87%	0.06%
Cubic	5.6%	92.0%	2.4%
Image	0.8%	98.7%	0.5%

Looking at individual partition windows, Table 5.2 shows that the mem-

ory requests from the RTOS can be quite significant. Even for Image where the supervisor mode requests account for less than 1 percent of the total number of requests, we can see that for an individual partition window supervisor mode requests can account for 66 percent of the memory requests.

Table 5.2: Proportion of supervisor mode requests per partition window.

Partition	Supervisor mode requests	
	Min	Max
Nav	3.2%	72.0%
Mult	0.1%	7.2%
Cubic	0.3%	8.3%
Image	0.3%	66.0%

Using the box plot in Figure 5.6 to visualize the diversity of the number of memory requests over different partition windows, we see the difference in the nature of the four applications in another light.

Figure 5.6: Distribution of user mode memory requests per partition window (outliers have been grouped together into one if there are many with the same value).

A relatively narrow box in this context, as illustrated by the Nav and Mult request patterns, demonstrates a more deterministic behavior. Here Mult is the most deterministic with fewer outliers. Nav is typically deterministic, but there are partition windows in which outliers exhibit an order

of magnitude larger number of requests, thus clearly questionable if the narrow band would be a better estimate for a budget, or the pessimistic count based on outliers. A further accentuated version of this behavior is shown by Cubic, where a larger number of outliers are shown. This is an example of a process where using the maximum number of requests in some partition windows, if used for setting the budget, would be too pessimistic. However, barring those outliers, the process shows a relatively deterministic memory request pattern (a narrow band). Finally, image processing is depicted in a wide box with an average that is far less than the other potential number of requests exhibited. This would be an example of a process for which making an educated guess to a relevant budget for memory requests would be quite difficult. Reconsidering the structure of the code and potentially dividing it into more deterministic subfunctions would be a recommended approach.

5.4 Discussion

If we look closely at Figure 5.4a, we can see that there is one sample in the very beginning (just below 10^4) where the user mode requests are a magnitude larger than anywhere else. This has to do with the initialization of the partition before entering normal mode. This can also be seen from the outliers in Figure 5.6, where we can see a similar behavior for Cubic (although we cannot see if the outliers belong to the initialization phase). If we were to use this maximum for doing inter-core interference analysis, the result would be very pessimistic. The outliers should be further analyzed for relevance, and if only occurring during initialization or fatal error handling the outliers can safely be ignored.

The large differences in number of requests between partition windows exhibited by Cubic and Image may be a result of the fact that the calculations take several partition windows to complete and the memory request balancing could perhaps be improved to yield a narrower band of memory requests. Without paying attention to memory request characteristics during application design, it may be difficult to avoid overly pessimistic budgets and this can affect the possibility to efficiently integrate legacy applications on a multi-core platform. We have a simplified system for our experiments and because of this there are additional sources of supervisor mode memory requests that have not been accounted for. The fact that there is only one partition on each core probably results in less space partitioning overhead as the memory management unit of the core is less utilized and this will lead to fewer misses. As stated previously, the partition scheduling of an ARINC 653 system will incur a number of memory requests due to the static cyclic schedule. Since partition scheduling occurs after each partition window, the overhead will increase as new partition windows are added to the schedule. In our case with only one partition there is also only one partition window, which will result in less partition-scheduling overhead than would occur in a real avionics system. None of these simplifications make the implications

less valid. On the contrary, there would be additional memory requests to support our hypothesis. Additionally, some system calls may be handled by the RTOS in user mode and memory requests for these will be hidden in the user mode numbers and depending on the analysis of the application memory request behavior, they may be missed and not accounted for.

The results are also applicable to a symmetric multiprocessing (SMP) ARINC 653 system, but the partition scheduling would not cause interference in the same way as in AMP systems. The partition scheduling would always occur at the same time on all cores and the interference would be internally in the RTOS, but the delay until the next partition starts executing would probably be longer than if running on a single-core system.

Our measurements on the four diverse types of applications reveal some challenges related to determining memory request budgets.

- Nav poses no challenges if the initialization outlier can be ignored.
- Mult is deterministic in its memory request behavior and poses no challenges.
- Cubic has too many outliers that nevertheless represent a smaller fraction of number of requests, resulting in large pessimism.
- Image is an example where it is difficult to avoid pessimism.

The results show that the impact of RTOS memory requests are dependent on application behavior. For the computationally intensive Nav the impact is quite significant with the RTOS contributing more memory requests than the application itself (48 percent vs. 30 percent of the total number of memory requests). On the other hand, for the very memory-intensive Mult the impact is marginal with less than 0.1 percent contributed by the RTOS.

Even though the percentage is low for three out of the four applications, there may be a substantial amount of memory requests from the RTOS adding extra interference on other cores and impacting the worst-case response time of the applications. The partition scheduling in an ARINC 653 RTOS also introduces memory requests not accounted for due to the static cyclic nature of the schedule.

While this study is limited and based on only one RTOS, our approach and methodology is equally applicable to other RTOSes. The experimental setup could be applied to different hardware platforms and different RTOSes to build more confidence. However, our results clearly show that RTOS memory requests need to be incorporated to correctly perform the inter-core interference analysis.

We have so far only considered the number of memory requests issued by a core. Next, we will study the timing behavior of memory requests experiencing inter-core interference, and how the requests issued by the RTOS will be incorporated into the analysis. We will use a more detailed model containing the banks of a DRAM module and the interference this may result in.

Chapter 6

Shared Memory Bank Interference

In Chapter 5, we were only concerned with the *number* of memory requests issued by the RTOS and applications. In this chapter, we will study *timings* of these memory requests and the impact of the inter-core interference that is made apparent when a more detailed model of the memory hierarchy is used.

For some systems, it may be possible to locate the data in such a way that each core access its own private DRAM bank(s), but in the general case there will be some sharing of data between applications and these applications may reside on different cores resulting in a use-case where shared banks is a necessity. It may also be the case that we have more cores than banks, which also will result in the necessity of sharing banks. Currently, the number of cores in a multi-core chip is growing faster than the number of banks in the DRAM [46].

In this chapter, we consider integrating the SCE concepts (Colored Lock-down [58], MemGuard [105] and PALLOC [102]) discussed in Chapter 3.2 in an ARINC 653 [1] real-time operating system (RTOS) designed for avionics systems. Specifically, we study the general case of bounding the interference delay when using shared DRAM banks.

6.1 DRAM Model

We use the basic concepts of DRAM and memory controllers as described in Chapter 2. In Section 2.2.3 we described the commands (PRE, ACT, RD/WR) sent by a DRAM controller to the DRAM devices. These commands take time to finish and the DRAM controller must thereby satisfy timing constraints between the commands to ensure a proper function. The JEDEC standard [42] specifies a number of requirements for JEDEC-

compliant SDRAM devices as shown in Table 6.1. These timing constraints

Table 6.1: DRAM timing parameters.

Parameter	Value	Description
BL	8 columns	Burst Length
CL	13 cycles	Column Access Strobe Latency
WL	9 cycles	Write Latency
t_{RCD}	13 cycles	Activate to read/write latency
t_{RRD}	5 cycles	Activate to activate interval
t_{RP}	13 cycles	Precharge to activate interval
t_{FAW}	26 cycles	Window for four activates
t_{WTR}	7 cycles	Write to read interval
t_{WR}	14 cycles	Data to precharge min interval
t_{CK}	1 ns	DRAM clock cycle time

affect the timing variability of the access time to the DRAM and must be considered to fully understand the interference delays that may arise due to requests from multiple cores.

In modern memory controllers, the memory requests are not always sent to the DRAM in the order they are sent by the core. Instead, they are buffered in request buffers and issued to the DRAM in the order specified by a memory scheduler. The policy often used today, is the First-Ready First-Come First-Served (FR-FCFS) policy. This policy prioritizes requests to already open rows before closed rows in order to minimize row conflicts, but this may introduce additional interference delays.

6.1.1 Inter- and Intra-Bank Delay

In the general case tasks share data and need to share banks. To more accurately model the worst-case memory interference delay, we build on the work by Kim et al. [46] that includes the interference delay resulting from shared banks.

We use the request-driven approach presented by Kim et al. [46] since it does not make any assumptions on the memory requests of applications running on other cores. The job-driven approach can in some situations reduce the pessimism of the delays, but it requires us to know the number of interfering memory requests from other cores and goes against the reconfiguration ideas of Integrated Modular Avionics (IMA) [80].

The interference delay experienced by a core p for a memory request is given by $RD_p = RD_p^{inter} + RD_p^{intra}$, where RD_p^{intra} is the inter-bank interference delay for core p and RD_p^{inter} is the inter-bank interference delay. RD_p^{inter} is the delay due to a memory request generated by a core p is being delayed by requests from other cores due to timing effects of accessing the common command and data bus. RD_p^{intra} is a result of multiple cores

(a) Inter-bank interference. Core p being delayed because of other cores accessing other banks (RD_p^{inter}).

(b) Intra-bank interference. Core p being delayed because core $q1$ is accessing the same bank (RD_p^{intra}).

Figure 6.1: Components of RD_p .

accessing (different rows in) the *same* bank. Figure 6.1a and 6.1b illustrate these scenarios. RD_p^{inter} is given by:

$$RD_p^{inter} = \sum_{\substack{\forall q: q \neq p \wedge \\ shared(q,p) \neq \emptyset}} (L_{inter}^{PRE} + L_{inter}^{ACT} + L_{inter}^{RW}) \quad (6.1)$$

where $shared(q, p)$ is the set of DRAM banks shared between core q and core p , L_{inter}^{PRE} reflects timings of the address/command bus scheduling. L_{inter}^{ACT} is related to the minimum separation time between two activate commands sent to two different banks, and L_{inter}^{RW} is related to the data bus contention and the bus turn-around delay as a result of the data flow direction change on the bus if a read is issued after a write or vice versa. No more than four ACTs are allowed during the period of time denoted t_{FAW} . This gives the following:

$$\begin{aligned} L_{inter}^{PRE} &= t_{CK} \\ L_{inter}^{ACT} &= \max(t_{RRD}, t_{FAW} - 3 \cdot t_{RRD}) \cdot t_{CK} \\ L_{inter}^{RW} &= \max(WL + BL/2 + t_{WTR}, CL + BL/2 + 2 - WL) \cdot t_{CK} \end{aligned}$$

where WL , CL , BL and t_{WTR} are part of the DRAM timing parameters in Table 6.1.

RD_p^{intra} is then given by:

$$RD_p^{intra} = reorder(p) + \sum_{\substack{\forall q: q \neq p \wedge \\ shared(q,p) \neq \emptyset}} (L_{conf} + RD_q^{inter}) \quad (6.2)$$

where $reorder(p)$ calculates the delay from the reordering based on the number of queued row hits ($N_{reorder}$) that may be scheduled before the request under analysis and L_{conf} is a constant that represents a row-conflict in the same bank, which requires both a PRE and an ACT command to

close the current row and open a new row.

$$\begin{aligned}
 L_{conf} &= (t_{RP} + t_{RCD}) \cdot t_{CK} + L_{hit} \\
 L_{hit} &= \max(CL + BL/2 + 2, WL + BL/2 + \max(t_{WTR}, t_{WR})) \cdot t_{CK} \\
 reorder(p) &= \begin{cases} 0 & \text{if } \nexists q : q \neq p \wedge shared(q, p) \neq 0 \\ L_{conhit}(N_{reorder}) + \sum_{\substack{\forall q: q=p \wedge \\ shared(q, p) \neq \emptyset}} L_{inter}^{RW} \cdot N_{reorder} & \end{cases} \quad (6.3)
 \end{aligned}$$

$L_{conhit}(N_{reorder})$ is the time it takes to service $N_{reorder}$ consecutive row hits:

$$L_{conhit}(m) = \{\lceil m/2 \rceil \cdot (WL + BL/2 + t_{WTR}) + \lfloor m/2 \rfloor \cdot CL + (t_{WR} - t_{WTR})\} \cdot t_{CK}$$

Based on this, Kim et al. extend the classical response time test [43] to include the memory interference delay:

$$R_i^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot C_j + H_i \cdot RD_p + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot H_j \cdot RD_p \quad (6.4)$$

where H_i denotes the maximum number of memory requests generated by task i , C_i denotes the WCET of task i when run in isolation, R_i denotes the response time of task i , and T_i denotes the minimum inter-arrival time of task i . $R_i^0 = C_i$ and the test terminates when $R_i^{k+1} = R_i^k$.

6.2 SCE Adaptation and Extension

In this section, we outline the steps needed to implement the SCE concepts on our evaluation platform and RTOS from Chapter 5. The applications tasks are organized in ARINC 653 partitions scheduled in a static cyclic schedule unique for each core.

6.2.1 Memory Request Monitoring

The memory request monitoring within the RTOS has been modified to also suspend the partition if the counted number of memory requests exceeds a specified limit during its partition window, in effect regulating the number of memory requests that can be issued from a partition. This is accomplished by using the Performance Monitor Counters (PMCs) to generate an interrupt at overflow. The budget is replenished at the start of each period. The PMC is set up to count the requests issued by both the partition and by the RTOS itself. MemGuard [105] implements an even bandwidth distribution, each core is allowed the same number of requests during each regulation period. The regulation period is a system-wide parameter. Our implementation is more flexible as the regulation period is

equal to the partition window duration of each partition, also the allowed number of memory requests can be specified per partition. This flexibility complicates the analysis, but it is quite simple to achieve the same behavior as MemGuard by using the same partition window duration and the same allowed number of memory requests for all partitions on all cores. The sum of the memory budgets for all partitions must not exceed the total number of requests possible during a regulation period as this would saturate the DRAM controller and introduce additional delays.

6.2.2 Memory Bank Partitioning

Earlier Linux based memory bank partitioning schemas assume (such as PALLOC [102]) that the page size of the memory management unit (MMU) is smaller than the row size of the DRAM (e.g., 4KiB versus 8KiB). This makes it possible to always allocate contiguous virtual memory that will map to a set of physical memory pages belonging to the same bank. Our chosen RTOS uses a different approach, where all memory is allocated using variable page sizes during initialization. This will minimize page misses in the MMU to be handled by the RTOS, but it will also make it very difficult to implement bank partitioning as the MMU page sizes used could possibly span multiple DRAM banks. Therefore, we aim to use the SCE concepts extended with the shared bank interference delay estimations described in Section 6.1.1.

6.2.3 Cache Partitioning

Instead of implementing the cache-coloring concept, used in SCE by Mancuso et al. [59], we utilize the ability to allocate cache ways for exclusive use by a specific core. The cache ways can be evenly distributed; giving each core the same number of ways ($\frac{Cachesize}{Totalnumberofways}$) or the tasks running on each core could be analyzed/profiled to determine the proper amount of cache ways to reserve for that particular core (similar to page coloring). In our system (described later) the L2 cache is set up to allocate four ways to each of the four cores. This effectively limits the available cache for each core to 512 KiB and will ensure that there is no inter-core cache interference.

6.3 Validation of the SCE Extension

In this section, we describe the methodology for validating the extended SCE model (as described in Section 6.2) on our platform, and show the validation results. We use the four avionics related applications from Chapter 5: Nav, Mult, Cubic and Image. In this setup all partitions consist of only one process (Nav has two, but we are only interested in the highest priority one), which simplifies the response time calculations. Without loss of generality,

Equation 6.4 is therefore reduced to:

$$R_i = C_i + H_i \cdot RD_p \quad (6.5)$$

6.3.1 Methodology

We use the following method for validating the WCRT estimations:

- We estimate the WCET and count the number of memory requests for each partition in isolation.
- The worst-case response time for each partition when executing in parallel is calculated based on the (extended) SCE formulas.
- We measure the (worst case) response time for each partition and compare with the calculated estimates.
- For critical partitions where calculations indicate a small margin to the relative deadline, we perform additional interference studies with a memory-intensive synthetic application to ensure that maximum memory bank interference is properly accounted for.

Each application is run inside one partition and deployed to different cores and they all execute in 60 Hz. We run the system in an asymmetric multiprocessing (AMP) configuration (i.e., each core has its own instance of the RTOS).

The experiments are performed on an NXP (Freescale) T4240 using only one cluster with four cores sharing the 2048 KiB L2 cache, which is partitioned to allocate four ways for each core resulting in each core having 512 KiB of L2 cache each. Without loss of generality, we have in this chapter disabled the reordering of requests in the DRAM controller (i.e., $N_{reorder} = 0$).

6.3.2 WCET Estimation

Single-core WCET can be estimated either using static analysis, by measuring the execution time of the application when running on the target hardware or by some hybrid method. In this chapter, we use a measurement-based approach where we (manually) insert instrumentation points (IPOINTs) that store an identifier and a timestamp. Post-processing the IPOINTs gives us the execution times for different parts of the task and these measurements can be combined to find a high-water mark of the execution time, which can be used for deriving an estimate of the WCET.

The four applications are run in isolation on core 0 with the rest of the cores disabled. The histograms in Figure 6.2 show the distribution of execution times for the applications when run in isolation and Table 6.2 summarizes the measured WCET and the number of memory requests per period for each application and also for the RTOS.

Figure 6.2: Execution time measurements in isolation.

To ensure that the IPOINTs do not introduce any unintentional probe effects, we measure the execution overhead of an IPOINT and also the number of memory requests with and without IPOINTs in the applications. The maximum execution time of one IPOINT is 23 ns, which gives a maximum overhead of 0.5 percent per partition window. No significant increase of memory requests is observed when using IPOINTs (see Figure 6.3).

6.3.3 Response Time

To calculate the response time (according to Equation 6.5) we need the total interference delay, RD_p . Using the equations in Section 6.1.1, we calculate RD_p^{inter} and RD_p^{intra} to get RD_p . Table 6.3 summarizes the worst-case service times calculated using the DRAM timing parameters in Table 6.1. This will result in the following inter- and intra-bank interference delays for

Table 6.2: Characterization of partitions in isolation.

Partition	Period (μs)	WCET (C) (μs)	Memory Requests (H) (Partition)	(RTOS)
Nav	16667	14	93	54
Mult	16667	16615	21740	160
Cubic	16667	9345	45	38
Image	16667	4391	560	40

Figure 6.3: Comparison of memory requests.

our evaluation system:

$$RD_p^{inter} = \sum_{\substack{\forall q: q \neq p \wedge \\ shared(q,p) = \emptyset}} (32 \cdot t_{CK}) \quad (6.6)$$

$$RD_p^{intra} = reorder(p) + \sum_{\substack{\forall q: q \neq p \wedge \\ shared(q,p) \neq \emptyset}} (53 \cdot t_{CK} + RD_q^{inter}) \quad (6.7)$$

Table 6.3: Worst-case service times.

Parameter	Value
L^{PRE}	1 ns
L^{ACT}	11 ns
L^{RW}	20 ns
L_{conf}	53 ns

Figure 6.4 shows the calculated intra- and inter-bank delays as well as the combined delay (RD_p). Intuitively, one would imagine that a larger number of cores would give more interference and that maximum would occur when

all cores share the same bank, as illustrated in Figure 6.5b. However, as we can see the maximum delay does not occur when all four cores share the same bank. Instead, the maximum occurs when three cores share the same bank, corresponding to Figure 6.5a.

This is a result of the intra-bank delay depending on the inter-bank delay for other cores (Equation 6.2). When all cores access the same bank the inter-bank delay is zero, which will result in the drop of delay time seen in Figure 6.4, given an L_{conf} smaller than the $\sum RD_q^{inter}$ contributing in the case with two cores sharing bank with the core under analysis. In the following estimates, we use the maximum total interference corresponding to the highest point on the curve (209 ns).

Figure 6.4: Theoretical interference delay when using four cores.

Figure 6.5: Cores sharing banks causing interference delay.

The estimated WCRTs listed in Table 6.4 show that the calculated response time of Nav, Cubic and Image is safely below their relative deadline, but for Mult the estimated WCRT exceeds the relative deadline. Mult performs several orders of magnitude more memory requests compared to Nav, Cubic and Image. For every partition window, we use the earlier mentioned IPOINTs to record response times over a 30 second interval (1800 measurements) with partition placement on cores according to column 2. The maximum measured response time is then disclosed in column 4.

This shows that when the partitions execute in parallel no partition misses their deadlines though the critical application Mult has a tight margin (see periods in Table 6.2). We can also see that the WCRT measurements do not differ in any significant way from the WCET measurements in that table. The memory controller can service all the memory requests without being saturated. The memory request patterns of the partitions are such that they do not interfere in many instances. The estimation model assumes that all cores issue memory requests simultaneously. To ensure the measurements are not overly optimistic we perform additional measurements on Mult, whose estimated WCRT exceeds its relative deadline, in a scenario with maximal memory interference.

Table 6.4: Maximum response time of partitions.

Partition	Core	Response time (R) (μs)	
		Estimated	Measured
Nav	0	45	14
Mult	1	21192	16620
Cubic	2	9362	9345
Image	3	4516	4391

6.3.4 Critical Processes

When we run Mult on core 0 in parallel with a memory-intensive task deployed on core 1–3 with disabled memory request regulation, Mult misses its deadline. If we turn on the memory regulation, mentioned in Section 6.2.1, for the memory-intensive tasks on core 1–3 with a suitable budget, we notice that Mult no longer misses its deadline. This shows that *given* a suitable restriction of the memory requests issued by partitions running on other cores we are able to run Mult within its time constraints. So, for a critical task, the correct estimation of regulation budget of *other* tasks is essential.

To measure Mult’s response time in this scenario we disable the overrun detection function and perform repeated experiments where the memory budgets of the memory-intensive tasks were reduced until a WCRT value below the relative deadline was found for Mult. The resulting response times (with and without regulation) can be found in Table 6.5.

Table 6.5: Measured maximum response time of Mult with memory-intensive tasks in parallel.

Partition	Core	Response time (R) (μs)	
		No regulation	Regulation
Mult	0	17075	16654

6.3.5 Applying the Method to an Earlier Benchmark

To further assess the validity of the approach we also use the *Latency* and *Bandwidth* benchmarks from Yun et al. [105], adapted to our environment and RTOS, to measure the worst-case memory interference. The benchmarks are modified to enable us to direct the requests from *Bandwidth* to a specified DRAM bank. We run *Latency* on core 0 and *Bandwidth* on core 1–3 several times with different number of *Bandwidth* instances targeting the same DRAM bank as *Latency*. These measurements compared to the estimations are shown in Figure 6.6. As we can see, the estimations are a conservative (and possibly somewhat pessimistic) approximation of the measurements.

Figure 6.6: Comparison of measured and estimated request time.

6.4 Discussion

One assumption affecting the pessimism of the method is the sequential issuing of subcommands (PRE, ACT, RD/WR) from the memory controller to the DRAM device, causing a memory request to always suffer inter-bank delay. Modern memory controllers can interleave commands for different banks and in a way, create a pipelining effect resulting in a non-additive delay. A more accurate modeling of the subcommand timings would result in less pessimistic response times. Another assumption that should be relaxed in the future is the assumption of only one outstanding memory request per core, this could potentially have a large impact on the delay times if several requests from other cores are prioritized before the request on the core under analysis.

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions based on the work performed in this thesis and discusses potential future research directions.

7.1 Conclusions

Multi-core platforms have been introduced in many different settings, but so far they have not been utilized in the domain of safety-critical avionic real-time systems. A substantial amount of research has been put into the area and many promising methods have been proposed.

In this thesis, we address the areas of timing predictability and fault tolerance for avionics systems. While addressing these areas, we also (informally) assess the maturity and applicability of existing methods.

Regarding timing predictability, several sources that present challenges exist as discussed in Chapter 2.2. The inter-core interference due to parallel accesses to the DRAM memory has been addressed by several proposed resource-monitoring systems, none of which consider the memory requests issued by the RTOS and the impact these requests have on the DRAM access timings for requests issued by other cores.

Our results show that the memory requests issued by the RTOS can have a significant impact on the inter-core interference if not accounted for in the run-time monitoring and during derivation of memory request budgets for resource-monitoring systems.

Another aspect of the inter-core interference is related to the allocation of instructions and data to core-private banks in the DRAM. The use of only private DRAM banks is not scalable when the applications grow and the number of cores increase faster than the number of banks in the DRAM. Also, applications sharing data in shared memory cannot use this scheme.

To address this, we present an adaptation and extension of the SCE framework for an avionics ARINC 653 RTOS targeting the T4240 multi-core SoC from NXP. We relax the constraints requiring private memory banks for each core and our extension provides an analytical upper bound on the interference delay.

Our work also highlights an interesting aspect of the calculated response times as a function of the number of cores, namely that, in theory, the maximum core deployment need not give maximum memory bank interference. This has not yet been investigated further and is left as future work.

Another important aspect needing attention in the use of multi-core platforms is the impact of faults. The faults themselves and the fault tolerance mechanisms could affect both timing predictability and temporal isolation. We have only begun the work regarding the impact of fault tolerance on safety-critical multi-core systems.

In this thesis, we present a comprehensive analysis of the state-of-the-art fault tolerance and fault injection methods aiming at validated WCET estimations within multi-core systems and find several shortcomings in the currently available research. We posed three questions in Chapter 4 that we set out to answer. So far, none of the existing literature have been able to help us answer these questions in a satisfactory way. While many of the approaches are promising, several challenges remain.

For instance, we identify a lack of research on WCET/WCRT estimates under faulty conditions on safety-critical COTS multi-core platforms that require temporal partitioning. This research is needed to safely deploy multi-core platforms in the avionics domain and for certification authorities to accept and approve their usage.

There is also a lack of research on fault tolerance induced memory requests and the effects on resource-monitoring systems with deterministic timing (this work is related to the memory requests of the RTOS as discussed in Chapter 5).

None of the fault injection methods consider verification of timeliness, but a cycle-accurate simulator could perhaps be used for WCET estimation in presence of hardware faults provided the hardware model in the simulator is detailed enough.

To summarize, we find that no work combining timing predictability and hardware reliability in presence of inter-core interference on multi-core systems has been identified. This makes the topic an interesting area for future research.

Once the research directions presented in Section 7.2 bring the maturity needed in analyses for assurance of timing predictability of applications running on multiple cores, we will be closer to the adoption of multi-core platforms in safety-critical avionics systems.

7.2 Future Work

Timing predictability and fault tolerance for safety-critical systems are active research areas. While our work only touched parts of the areas, there are many interesting research directions:

- **Timing effects due to fault tolerance:** There are more or less isolated islands between work on fault tolerance and timing assurance in the multi-core setting. We believe a method integrating fault tolerance and timing predictability is needed. Integrating the impact of fault tolerance methods on shared resources (e.g., memory requests), which affects the WCET and WCRT estimates is needed.
- **Fault injection on multi-core:** Together with an integrated method, we also need fault injection platforms aiming at multi-core fault tolerance. Furthermore, combining outcomes of fault injection experiments with models for analysis of WCET and WCRT in a systematic way is a future direction of research.
- **Formal methods:** Another interesting path explores the area of formal verification instead of empirical measurement-based methods. Modeling the relevant parts of the system, including fault impacts, and deriving worst-case response times by using UPPAAL and model checking is a research direction we will consider.
- **Optimal budgets:** We showed that the RTOS memory requests need to be considered when deriving the budgets for the applications on a core in a resource monitoring system. However, how to derive optimal CPU usage and memory request budgets given a number of tasks on a number cores is an open problem. Properly assigned budgets could increase the schedulability of a task system and also increase the utilization of the cores.
- **Improved DRAM model:** The DRAM model used in this thesis does not consider request reordering and it also assumes each command (PRE, ACT, RD/WR) suffers from inter-bank interference. Many modern memory controllers perform request reordering to improve latency and they can also pipeline commands resulting in non-additive delays. Improving the DRAM model is also an interesting direction of work, which could result in tighter interference delays.
- **Predictable hardware:** In this thesis, we have only considered readily available COTS hardware platforms, but these are developed for a different market and optimized for maximal throughput. We believe that work on timing predictability issues in several parts of a system on chip is an interesting direction. Timing predictable soft cores and real-time memory controllers implemented on FPGAs may be a way forward.

Bibliography

- [1] Aeronautical Radio Inc (ARINC). ARINC 653: Avionics Application Software Standard Interface Part 1 - Required Services, 2010.
- [2] H. Agrou, P. Sainrat, M. Gatti, and P. Toillon. Mastering the behavior of multi-core systems to match avionics requirements. In *Proceedings of the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 6E5–1–6E5–12, Oct 2012.
- [3] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 97–102, March 2016.
- [4] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Syst.*, 51(1):77–123, Jan. 2015.
- [5] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. Multicore Operating-System Support for Mixed Criticality. In *Proc. of Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, April 2009.
- [6] ASHLEY: Avionics Systems Hosted on a distributed modular electronics Large scale dEmonstrator for multiple tYpes of aircraft. <http://www.ashleyproject.eu/>, March 2013.
- [7] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.
- [8] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

- [9] S. K. Baruah, A. Burns, and R. I. Davis. Response-Time Analysis for Mixed Criticality Systems. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 34–43, Washington, DC, USA, 2011. IEEE Computer Society.
- [10] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core Composability in the Face of Memory-bus Contention. *SIGBED Rev.*, 10(3):35–42, Oct. 2013.
- [11] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, pages 125–126, Sept 2006.
- [12] A. Biondi, G. Buttazzo, and M. Bertogna. Supporting Component-Based Development in Partitioned Multiprocessor Real-Time Systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 269–280, July 2015.
- [13] E. Carlisle, N. Wulf, J. MacKinnon, and A. George. DrSEUs: A dynamic robust single-event upset simulator. In *2016 IEEE Aerospace Conference*, pages 1–11, March 2016.
- [14] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):94:1–94:26, May 2013.
- [15] Certification Authorities Software Team. CAST 32A Multi-core Processors. https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf, 2016.
- [16] C. Chen, L. Santinelli, J. Hugues, and G. Beltrame. Static probabilistic timing analysis in presence of faults. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.
- [17] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multipath Programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, July 2012.
- [18] D. Dasari, B. Andersson, V. Nelis, S. Petters, A. Easwaran, and J. Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proceedings of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068–1075, Nov 2011.

- [19] D. Dasari, V. Nelis, and B. Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, 2016.
- [20] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.
- [21] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, Feb 1990.
- [22] L. Ecco, A. Kostrzewa, and R. Ernst. Minimizing dram rank switching overhead for improved timing bounds and performance. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, July 2016.
- [23] European Aviation Safety Agency (EASA). MCP CRI Issue 3.0, The Use of Multi-Core Processors in Safety-Critical Applications, March 2016.
- [24] G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla. Resource Usage Templates and Signatures for COTS Multicore Processors. In *Proc. of 52nd Annual Design Automation Conference, DAC '15*, pages 155:1–155:6, New York, NY, USA, 2015. ACM.
- [25] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.
- [26] R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *Proceedings of the 29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 5.E.3–1–5.E.3–11, Oct 2010.
- [27] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–15, Sept 2013.
- [28] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, March 2014.
- [29] J. Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, Tandem Computers, 1985.
- [30] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

- [31] D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51(2):128–152, 2015.
- [32] D. Hardy, I. Puaut, and Y. Sazeides. Probabilistic WCET estimation in presence of hardware for mitigating the impact of permanent faults. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 91–96, March 2016.
- [33] S. K. S. Hari, M. L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-cost hardware fault detection and diagnosis for multi-core systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 122–132, Dec 2009.
- [34] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS Support for Multicore Mixed-Criticality Systems. In *Proc. of 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.
- [35] K. Höfig. Failure-Dependent Timing Analysis - A New Methodology for Probabilistic Worst-Case Execution Time Analysis. In J. B. Schmitt, editor, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance: 16th International GI/ITG Conference, MMB & DFT 2012, Kaiserslautern, Germany, March 19-21, 2012. Proceedings*, pages 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [36] P. Huang, H. Yang, and L. Thiele. On the Scheduling of Fault-Tolerant Mixed-Criticality Systems. In *Proceedings of the 51st Annual Design Automation Conference, DAC '14*, pages 131:1–131:6, New York, NY, USA, 2014. ACM.
- [37] Y. Huang, P. Jalote, and C. Kintala. Two techniques for transient software error recovery. In M. Banâtre and P. A. Lee, editors, *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, pages 159–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.
- [38] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for Predictable Execution on Multi-core Platforms. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2014.
- [39] R. Inam, M. Sjödin, and M. Jagemar. Bandwidth measurement using performance counters for predictable multicore software. In *Proceedings of the 17th IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, Sept 2012.

- [40] R. Inam, J. Slatman, M. Behnam, M. Sjödin, and T. Nolte. Towards implementing multi-resource server on multi-core Linux platform. In *Proceedings of the 18th IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, Sept 2013.
- [41] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert. Ensuring robust partitioning in multicore platforms for IMA systems. In *Proceedings of the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 7A4–1–7A4–9, Oct 2012.
- [42] JEDEC. DDR3 SDRAM Standard, 2012.
- [43] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.
- [44] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele. Static Mapping of Mixed-Critical Applications for Fault-Tolerant MP-SoCs. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 31:1–31:6, New York, NY, USA, 2014. ACM.
- [45] T. Kelter, T. Harde, P. Marwedel, and H. Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *13th International Workshop on Worst-Case Execution Time Analysis*, pages 1–10, 2013.
- [46] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 145–154, April 2014.
- [47] Y. Kim, D. Broman, J. Cai, and A. Shrivastava. WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, April 2014.
- [48] L. Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *Proceedings of the 28th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 1.E.4–1–1.E.4–6, Oct 2009.
- [49] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov. A formal approach to the wcert analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Syst.*, 50(5-6):736–773, Nov. 2014.
- [50] A. Lanzaro, A. Pecchia, M. Cinque, D. Cotroneo, R. Barbosa, and N. Silva. A Preliminary Fault Injection Framework for Evaluating Multicore Systems. In F. Ortmeier and P. Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 106–116. Springer Berlin Heidelberg, 2012.

- [51] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.
- [52] J. Li, J. Xue, X. Xie, Q. Wan, Q. Tan, and L. Tan. Epipe: A low-cost fault-tolerance technique considering WCET constraints. *Journal of Systems Architecture*, 59(10, Part D):1383 – 1393, 2013.
- [53] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, 52(5):675–729, 2016.
- [54] Y. Li, B. Akesson, K. Lampka, and K. Goossens. Modeling and verification of dynamic command scheduling for real-time memory controllers. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.
- [55] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, Jan. 1973.
- [56] J. Lu, K. Bai, and A. Shrivastava. SSDM: Smart Stack Data Management for software managed multicores (SMMs). In *Proceedings of the 50th ACM / EDAC / IEEE Design Automation Conference (DAC)*, pages 1–8, May 2013.
- [57] G. Macher, A. Höller, E. Armengaud, and C. Kreiner. Automotive embedded software: Migration challenges to multi-core computing platforms. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1386–1393, July 2015.
- [58] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. of 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, April 2013.
- [59] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) Estimation in Multi-core Systems Using Single Core Equivalence. In *Proc. of 27th Euromicro Conference on Real-Time Systems*, pages 174–183, July 2015.
- [60] M. Manoochehri and M. Dubois. Accurate Model for Application Failure Due to Transient Faults in Caches. *IEEE Transactions on Computers*, 65(8):2397–2410, Aug 2016.

- [61] S. Meisner and M. Platzner. Thread Shadowing: Using Dynamic Redundancy on Hybrid Multi-cores for Error Detection. In D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, editors, *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, pages 283–290. Springer International Publishing, Cham, 2014.
- [62] U. M. Mirza, F. Gruian, and K. Kuchcinski. Mapping streaming applications on multiprocessors with time-division-multiplexed network-on-chip. *Computers & Electrical Engineering*, 40(8):276 – 291, 2014.
- [63] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Proc. of 10th International Conference on Computer and Information Technology*, pages 1864–1871, June 2010.
- [64] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12–17, Dec 2011.
- [65] H. Mushtaq, Z. Al-Ars, and K. Bertels. A user-level library for fault tolerance on shared memory multicore systems. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pages 266–269, April 2012.
- [66] H. Mushtaq, Z. Al-Ars, and K. Bertels. Fault tolerance on multi-core processors using deterministic multithreading. In *2013 8th IEEE Design and Test Symposium*, pages 1–6, Dec 2013.
- [67] R. Natella and D. Cotroneo. Emulation of Transient Software Faults for Dependability Assessment: A Case Study. In *Dependable Computing Conference (EDCC), 2010 European*, pages 23–32, April 2010.
- [68] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing Dependability with Software Fault Injection: A Survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, Feb. 2016.
- [69] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Proc. of 26th Euromicro Conference on Real-Time Systems*, pages 109–118, July 2014.
- [70] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero. An Analyzable Memory Controller for Hard Real-Time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90, Dec 2009.

- [71] R. M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, 2014.
- [72] R. M. Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems*, 53(1):45–81, 2017.
- [73] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 269–279, April 2011.
- [74] M. Peshave, F. B. Bastani, and I. L. Yen. High-Assurance Reconfigurable Multicore Processor Based Systems. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 220–226, Nov 2011.
- [75] D. K. Pradhan, editor. *Fault-tolerant Computer System Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [76] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaud, and E. To-var. Cache-Persistence-Aware Response-Time Analysis for Fixed-Priority Preemptive Systems. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262–272, July 2016.
- [77] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *Proc. of the 7th International Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.
- [78] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. In *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 211–214, Oct 2015.
- [79] RTCA, Inc. RTCA/DO-254, Design Assurance Guidelines for Airborne Electronic Hardware, 2000.
- [80] RTCA, Inc. RTCA/DO-297, Integrated Modular Avionics (IMA) Development, Guidance and Certification Considerations, 2005.
- [81] RTCA, Inc. RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification, 2012.
- [82] J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance, 1999.

- [83] SAE International. ARP 4754A, Guidelines For Development Of Civil Aircraft and Systems, 2010.
- [84] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–224, April 2010.
- [85] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-Time Computing on Multicore Processors. *Computer*, 49(9):69–77, Sept 2016.
- [86] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, April 2009.
- [87] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 1:1–1:10, New York, NY, USA, 2013. ACM.
- [88] M. Slijepcevic, L. Kosmidis, J. Abella, E. Q. nones, and F. J. Cazorla. DTM: Degraded Test Mode for Fault-Aware Probabilistic Timing Analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 237–248, July 2013.
- [89] M. Slijepcevic, L. Kosmidis, J. Abella, E. Q. nones, and F. J. Cazorla. Timing Verification of Fault-Tolerant Chips for Safety-Critical Applications in Harsh Environments. *IEEE Micro*, 34(6):8–19, Nov 2014.
- [90] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems. In *Proceedings of the 51st Annual Design Automation Conference, DAC '14*, pages 198:1–198:6, New York, NY, USA, 2014. ACM.
- [91] V. Vargas, P. Ramos, R. Velazco, J. F. Mehaut, and N. E. Zergainoh. Evaluating SEU fault-injection on parallel applications implemented on multicore processors. In *Circuits Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*, pages 1–4, Feb 2015.
- [92] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243, Dec 2007.

- [93] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Proc. of 25th Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013.
- [94] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10, Oct 2007.
- [95] J. Whitham, N. C. Audsley, and R. I. Davis. Explicit Reservation of Cache Memory in a Predictable, Preemptive Multitasking Real-time System. *ACM Trans. Embed. Comput. Syst.*, 13(4s):120:1–120:25, Apr. 2014.
- [96] M. M. Wilding, D. S. Hardin, D. A. Greve, and R. C. Inc. Invariant Performance: A Statement of Task Isolation Useful for Embedded Application Integration. In *In Dependable Computing for Critical Applications, DCCA-7*, pages 287–300, 1999.
- [97] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *Proc. of 34th Real-Time Systems Symposium*, pages 372–383, Dec 2013.
- [98] N. Wulf, G. Cieslewski, A. Gordon-Ross, and A. D. George. SCIPS: An emulation methodology for fault injection in processor caches. In *Aerospace Conference, 2011 IEEE*, pages 1–9, March 2011.
- [99] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *Computers Digital Techniques, IET*, 5(2):123–135, March 2011.
- [100] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont. FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 8:1–8:8, New York, NY, USA, 2016. ACM.
- [101] H. Yun, S. Gondi, and S. Biswas. Protecting Memory-Performance Critical Sections in Soft Real-Time Applications. *CoRR*, abs/1502.02287, 2015.
- [102] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proc. of 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166, April 2014.

- [103] H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *Proc. of 27th Euromicro Conference on Real-Time Systems*, pages 184–195, July 2015.
- [104] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308, July 2012.
- [105] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. of 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64, April 2013.

- No 17 **Vojin Plavsic**: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
- No 28 **Arne Jönsson, Mikael Patel**: An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
- No 29 **Johnny Eckerland**: Retargeting of an Incremental Code Generator, 1984.
- No 48 **Henrik Nordin**: On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
- No 52 **Zebo Peng**: Steps Towards the Formalization of Designing VLSI Systems, 1985.
- No 60 **Johan Fagerström**: Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
- No 71 **Jalal Maleki**: ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
- No 72 **Tony Larsson**: On the Specification and Verification of VLSI Systems, 1986.
- No 73 **Ola Strömfors**: A Structure Editor for Documents and Programs, 1986.
- No 74 **Christos Levcopoulos**: New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
- No 104 **Shamsul I. Chowdhury**: Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
- No 108 **Rober Bilos**: Incremental Scanning and Token-Based Editing, 1987.
- No 111 **Hans Block**: SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
- No 113 **Ralph Rönnquist**: Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
- No 118 **Mariam Kamkar, Nahid Shahmehri**: Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
- No 126 **Dan Strömberg**: Transfer and Distribution of Application Programs, 1987.
- No 127 **Kristian Sandahl**: Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
- No 139 **Christer Bäckström**: Reasoning about Interdependent Actions, 1988.
- No 140 **Mats Wirén**: On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
- No 146 **Johan Hultman**: A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
- No 150 **Tim Hansen**: Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
- No 165 **Jonas Löwgren**: Supporting Design and Management of Expert System User Interfaces, 1989.
- No 166 **Ola Petersson**: On Adaptive Sorting in Sequential and Parallel Models, 1989.
- No 174 **Yngve Larsson**: Dynamic Configuration in a Distributed Environment, 1989.
- No 177 **Peter Åberg**: Design of a Multiple View Presentation and Interaction Manager, 1989.
- No 181 **Henrik Eriksson**: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
- No 184 **Ivan Rankin**: The Deep Generation of Text in Expert Critiquing Systems, 1989.
- No 187 **Simin Nadjim-Tehrani**: Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
- No 189 **Magnus Merkel**: Temporal Information in Natural Language, 1989.
- No 196 **Ulf Nilsson**: A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
- No 197 **Staffan Bonnier**: Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
- No 203 **Christer Hansson**: A Prototype System for Logical Reasoning about Time and Action, 1990.
- No 212 **Björn Fjellborg**: An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
- No 230 **Patrick Doherty**: A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
- No 237 **Tomas Sokolnicki**: Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
- No 250 **Lars Strömberg**: Postmortem Debugging of Distributed Systems, 1990.
- No 253 **Torbjörn Näslund**: SLDFA-Resolution - Computing Answers for Negative Queries, 1990.
- No 260 **Peter D. Holmes**: Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
- No 283 **Olof Johansson**: Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.
- No 298 **Rolf G Larsson**: Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
- No 318 **Lena Srömbäck**: Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
- No 319 **Mikael Pettersson**: DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
- No 326 **Andreas Kågedal**: Logic Programming with External Procedures: an Implementation, 1992.
- No 328 **Patrick Lambrix**: Aspects of Version Management of Composite Objects, 1992.
- No 333 **Xinli Gu**: Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
- No 335 **Torbjörn Näslund**: On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.
- No 348 **Ulf Cederling**: Industrial Software Development - a Case Study, 1992.
- No 352 **Magnus Morin**: Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
- No 371 **Mehran Noghabai**: Evaluation of Strategic Investments in Information Technology, 1993.
- No 378 **Mats Larsson**: A Transformational Approach to Formal Digital System Design, 1993.

- No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
- No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
- No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
- No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
- No 398 **Johan Boye:** Dependency-based Groudnness Analysis of Functional Logic Programs, 1993.
- No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
- No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
- No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
- No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
- No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
- No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
- No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
- FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
- FHS 4/94 **Karin Petersson:** Informationssystemstrukturer, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
- No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
- No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
- No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
- No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
- No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
- No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
- FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
- No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
- No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
- No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
- No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
- No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
- No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
- No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
- No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
- FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
- No 482 **Eva L. Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
- No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
- No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
- No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
- No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
- No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
- FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
- FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
- No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
- No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
- No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
- No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
- No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
- No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
- No 546 **Magnus Werner:** Multidatabase Integration using Polymorphic Queries and Views, 1996.
- FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
- No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
- No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.
- No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
- No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
- No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
- No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

- No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
- No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
- No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
- No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
- No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
- No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
- No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
- No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
- No 598 **Rego Granlund:** C³Fire - A Microworld Supporting Emergency Management Training, 1997.
- No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
- No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
- No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
- FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
- FiF-a 6 **Tommy Wedlund:** Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
- No 615 **Silvia Coradeschi:** A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
- No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
- No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
- No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
- No 629 **Gunilla Ivelors:** Krigsspel och Informationsteknik inför en oförutsägbart framtid, 1997.
- No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
- No 639 **Jukka Mäki-Turja:** Smalltalk - a suitable Real-Time Language, 1997.
- No 640 **Juha Takkinen:** CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
- No 643 **Man Lin:** Formal Analysis of Reactive Rule-based Programs, 1997.
- No 653 **Mats Gustafsson:** Bringing Role-Based Access Control to Distributed Systems, 1997.
- FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
- No 674 **Marcus Bjärelund:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
- No 676 **Jan Håkegård:** Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
- No 668 **Per-Ove Zetterlund:** Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
- No 675 **Jimmy Tjäder:** Projektleddaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
- FiF-a 14 **Ulf Melin:** Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
- No 695 **Tim Heyer:** COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
- No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
- FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
- No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
- No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
- No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
- No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
- No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
- No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
- No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
- No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
- FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
- FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
- No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
- No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
- FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
- No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
- No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
- No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
- No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
- No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

- No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
- No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
- No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
- No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
- FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
- No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
- No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
- No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
- No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
- No 800 **Anders Subotic:** Software Quality Inspection, 1999.
- No 807 **Svein Bergum:** Managerial communication in telework, 2000.
- No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
- FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
- No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
- No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
- No 823 **Lars Hult:** Publikas Gränssytor - ett designexempel, 2000.
- No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
- FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
- No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
- No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
- FiF-a 37 **Ewa Braf:** Organisationers kunskapsverksamheter - en kritisk studie av "knowledge management", 2000.
- FiF-a 40 **Henrik Lindberg:** Webaserade affärsprocesser - Möjligheter och begränsningar, 2000.
- FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
- No 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
- No 863 **Dan Lawesson:** Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
- No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
- No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
- No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
- FiF-a 47 **Per-Arne Segerkvist:** Webaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
- No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
- No 906 **Lin Han:** Secure and Scalable E-Service Software Delivery, 2001.
- No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
- No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
- FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
- FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
- No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
- No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter, 2001.
- No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
- No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
- No 938 **Bourhane Kadmiry:** Fuzzy Control of Unmanned Helicopter, 2002.
- No 942 **Patrik Haslum:** Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
- No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
- FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
- No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
- No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
- No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
- FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
- No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
- No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
- No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
- No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
- No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

- No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
- No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
- No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
- No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
- FiF-a 62 **Lennart Ljung:** Utveckling av en produktivitetmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
- No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
- No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
- No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
- No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
- No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
- No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
- No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
- FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
- No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
- No 1034 **Arja Vainio-Larsson:** Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
- No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
- FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
- No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
- No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
- No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
- FiF-a 71 **Emma Eliason:** Effekthanlys av IT-systems handlingsutrymme, 2003.
- No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
- No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
- FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
- No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
- No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
- FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
- No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
- No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
- No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
- No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
- No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
- FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
- No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
- No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
- No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
- No 1130 **Ioan Chisalitã:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
- No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
- No 1149 **Vaida Jakonienė:** A Study in Integrating Multiple Biological Data Sources, 2005.
- No 1156 **Abdil Rashid Mohamed:** High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
- No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
- No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
- FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
- No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
- No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
- No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
- FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
- No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
- FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
- No 1172 **Petter Ahlström:** Affärsstrategier för seniorbostadsmarknaden, 2005.
- No 1183 **Mathias Cöster:** Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
- No 1184 **Åsa Horzella:** Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
- No 1185 **Maria Kollberg:** Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
- No 1190 **David Dinka:** Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.

No 1192 **Nicklas Bergfeldt**: Towards Detached Communication for Robot Cooperation, 2005.

No 1194 **Dennis Maciuszek**: Towards Dependable Virtual Companions for Later Life, 2005.

No 1204 **Beatrice Alenljung**: Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.

No 1206 **Anders Larsson**: System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.

No 1207 **John Wilander**: Policy and Implementation Assurance for Software Security, 2005.

No 1209 **Andreas Käll**: Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.

No 1225 **He Tan**: Aligning and Merging Biomedical Ontologies, 2006.

No 1228 **Artur Wilk**: Descriptive Types for XML Query Language Xcerpt, 2006.

No 1229 **Per Olof Pettersson**: Sampling-based Path Planning for an Autonomous Helicopter, 2006.

No 1231 **Kalle Burbeck**: Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.

No 1233 **Daniela Mihailescu**: Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.

No 1244 **Jörgen Skågeby**: Public and Non-public gifting on the Internet, 2006.

No 1248 **Karolina Eliasson**: The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.

No 1263 **Misook Park-Westman**: Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.

FiF-a 90 **Amra Halilovic**: Ett praktikerspektiv på hantering av mjukvarukomponenter, 2006.

No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.

No 1277 **Viacheslav Izosimov**: Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.

No 1283 **Håkan Hasewinkel**: A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.

FiF-a 91 **Hanna Broberg**: Verksamhetsanpassade IT-stöd - Design teori och metod, 2006.

No 1286 **Robert Kaminski**: Towards an XML Document Restructuring Framework, 2006.

No 1293 **Jiri Trnka**: Prerequisites for data sharing in emergency management, 2007.

No 1302 **Björn Hägglund**: A Framework for Designing Constraint Stores, 2007.

No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.

No 1305 **Magnus Ingmarsson**: Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.

No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.

No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.

No 1309 **Ola Leifer**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.

No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.

No 1313 **Zhiyuan He**: System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.

No 1317 **Jonas Elmqvist**: Components, Safety Interfaces and Compositional Analysis, 2007.

No 1320 **Håkan Sundblad**: Question Classification in Question Answering Systems, 2007.

No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.

No 1329 **Martin Magnusson**: Deductive Planning and Composite Actions in Temporal Action Logic, 2007.

No 1331 **Mikael Asplund**: Restoring Consistency after Network Partitions, 2007.

No 1332 **Martin Fransson**: Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.

No 1333 **Karin Camara**: A Visual Query Language Served by a Multi-sensor Environment, 2007.

No 1337 **David Broman**: Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.

No 1339 **Mikhail Chalabine**: Invasive Interactive Parallelization, 2007.

No 1351 **Susanna Nilsson**: A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.

No 1353 **Shanai Ardi**: A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.

No 1356 **Erik Kuiper**: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.

No 1359 **Jana Rambusch**: Situated Play, 2008.

No 1361 **Martin Karresand**: Completing the Picture - Fragments and Back Again, 2008.

No 1363 **Per Nyblom**: Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.

No 1371 **Fredrik Lantz**: Terrain Object Recognition and Context Fusion for Decision Support, 2008.

No 1373 **Martin Östlund**: Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.

No 1381 **Håkan Lundvall**: Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.

No 1386 **Mirko Thorstensson**: Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.

No 1387 **Bahlol Rahimi**: Implementation of Health Information Systems, 2008.

No 1392 **Maria Holmqvist**: Word Alignment by Re-using Parallel Phrases, 2008.

No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.

No 1401 **Annika Öhgren**: Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.

No 1410 **Rickard Holmsmark**: Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.

No 1421 **Sara Stymne**: Compound Processing for Phrase-Based Statistical Machine Translation, 2009.

No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.

No 1450 **Fabian Segelström**: Visualisations in Service Design, 2010.

No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.

No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.

No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.

No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.

No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.

No 1485 **Loove Broms**: Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.

FiF-a 101 **Johan Blomkvist**: Conceptualising Prototypes in Service Design, 2011.

No 1490 **Håkan Warnqvist**: Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.

No 1503 **Jakob Rosén**: Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.

No 1504 **Usman Dastgeer**: Skeleton Programming for Heterogeneous GPU-based Systems, 2011.

No 1506 **David Landén**: Complex Task Allocation for Delegation: From Theory to Practice, 2011.

No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.

No 1509 **Mariusz Wzorek**: Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.

No 1510 **Piotr Rudol**: Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.

No 1513 **Anders Carstensen**: The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.

No 1523 **Jody Foo**: Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.

No 1550 **Anders Fröberg**: Models and Tools for Distributed User Interface Development, 2012.

No 1558 **Dimitar Nikolov**: Optimizing Fault Tolerance for Real-Time Systems, 2012.

No 1582 **Dennis Andersson**: Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.

No 1586 **Massimiliano Raciti**: Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.

No 1588 **Banafsheh Khademhosseini**: Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.

No 1589 **Amy Rankin**: Resilience in High Risk Work: Analysing Adaptive Performance, 2013.

No 1592 **Martin Sjölund**: Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.

No 1606 **Karl Hammar**: Towards an Ontology Design Pattern Quality Model, 2013.

No 1624 **Maria Vasilevskaya**: Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.

No 1627 **Ekhiotz Vergara**: Exploiting Energy Awareness in Mobile Communication, 2013.

No 1644 **Valentina Ivanova**: Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.

No 1647 **Dag Sonntag**: A Study of Chain Graph Interpretations, 2014.

No 1657 **Kiril Kiryazov**: Grounding Emotion Appraisal in Autonomous Humanoids, 2014.

No 1683 **Zlatan Dragisic**: Completing the Is-a Structure in Description Logics Ontologies, 2014.

No 1688 **Erik Hansson**: Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.

No 1715 **Nicolas Melot**: Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015.

No 1716 **Mahder Gebremedhin**: Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015.

No 1722 **Mikael Nilsson**: Efficient Temporal Reasoning with Uncertainty, 2015.

No 1732 **Vladislavs Jahundovics**: Automatic Verification of Parameterized Systems by Over-Approximation, 2015.

FiF 118 **Camilla Kirkegaard**: Adding Challenge to a Teachable Agent in a Virtual Learning Environment, 2016.

No 1758 **Vengatanathan Krishnamoorthi**: Efficient and Scalable Content Delivery of Linear and Interactive Branched Videos, 2016.

No 1771 **Andreas Löfwenmark**: Timing Predictability in Future Multi-Core Avionics Systems, 2017.