

Titre: Mitigation of Hardware Interferences in Safety-Critical Multicore
Title: Systems

Auteur: Alexy Torres Aurora Dugo
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Torres Aurora Dugo, A. (2023). Mitigation of Hardware Interferences in Safety-Critical Multicore Systems [Thèse de doctorat, Polytechnique Montréal].
Citation: PolyPublie. <https://publications.polymtl.ca/55721/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/55721/>
PolyPublie URL:

Directeurs de recherche: Gabriela Nicolescu
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Mitigation of Hardware Interferences in Safety-Critical Multicore Systems

ALEXY TORRES AURORA DUGO

Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*

Génie informatique

Août 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

Mitigation of Hardware Interferences in Safety-Critical Multicore Systems

présentée par **Alexy TORRES AURORA DUGO**

en vue de l'obtention du diplôme de *Philosophiæ Doctor*

a été dûment acceptée par le jury d'examen constitué de :

Giovanni BELTRAME, président

Gabriela NICOLESCU, membre et directrice de recherche

Michel DAGENAIS, membre

Brett H. MEYER, membre externe

DEDICATION

*This dissertation is dedicated to my family and my friends,
thank you for helping me throughout this journey . . .*

ACKNOWLEDGEMENTS

I would like to thank my research director Prof. Gabriela Nicolescu. Thank you for guiding me during these past five years and helping me reach my objectives. I hope our path crosses again in the future. I am also very grateful for the help and support my laboratory partners Jean-Baptiste Lefoul and Felipe Gohring de Magalhães provided me during all this time.

I am thankful towards all the people I had the chance to work with and meet at the laboratory. Thank you all for the good memories we shared and the great friendship.

It was a pleasure to work with MANNARINO Systems and Software during these past five years. I would like to thank Nicolas Ulysse, Dahman Assal and Serge Harnois for their technical help and expertise.

To my family and friends that provided me all the support and advice that I needed, thank you for your endless support. Finally, I would like to thank all the people I met along the way and that helped me pursue my goals. I would particularly thank Professor Frédéric Pétrot for introducing me to the domain of embedded and real-time systems.

I would like to acknowledge the help of MITACS, NSERC and PROMPT for their financial support of this research.

RÉSUMÉ

Le domaine des systèmes critiques fait face à une demande plus en plus prononcée de performance et d'efficacité énergétiques. Les processeurs multicœurs sont de très bons candidats pour répondre à cette demande. Ces architectures permettent l'exécution de différentes applications en parallèle, sont plus efficaces en termes de consommation énergétique et apportent une meilleure redondance logicielle en permettant à plusieurs instances d'un même programme de s'exécuter en parallèle sur différents cœurs. Malgré ces avantages, les architectures multicœurs apportent de nombreux défis par rapport au déterminisme d'exécutions ainsi que de l'analyse temporelle du système. De part les accès simultanés des différents cœurs aux ressources, une contention sur lesdites ressources ainsi que leurs comportements potentiellement non déterministes (arbitration, éviction de données, etc.) apparaissent. Ce phénomène, appelé interférences, est difficile, voire impossible, à prédire et son impact sur les temps d'exécution dépend fortement du matériel ainsi que des applications présentes dans le système.

Les interférences empêchent l'analyse temporelles des applications, ce qui fait de l'utilisation des processeurs multicœurs dans les systèmes critiques un réel défi. En effet, l'isolation correcte des ressources, l'analyse temporelle correcte ainsi que la certification de ces systèmes sont primordiales avant leur déploiement. Ainsi, il est primordial de mieux comprendre, contrôler ou supprimer les interférences dans le système.

Les travaux existants dans le domaine proposent différents moyens de gérer les interférences, de manière indépendante. Chaque canal d'interférence (i.e. mémoire caches, bus, mémoire) est étudié de manière indépendante, sans prendre en compte l'impact des interférences entre elles. De plus, le besoin de finement configurer les méthodes de mitigations d'interférence rendent le déploiement et la configuration du système terriblement complexe, rendant ces processus impossibles pour l'intégrateur système. De ce fait, pour éviter les configurations amenant à des performances moindres, il est indispensable de fournir aux intégrateurs système des outils et méthodes permettant de les guider lors de la configuration du système.

Dans ce manuscrit, nous proposons d'étudier les interférences multicœur ainsi que de concevoir des solutions pour les contrôler. Nous apportons de nouveaux algorithmes et méthodes afin de guider l'intégrateur système dans son travail de configuration tout en lui évitant de générer des configurations qui pourraient nuire aux performances du système. Les contributions de cette thèse incluent, mais ne sont pas limitées à :

1. L'intégration d'un gestionnaire de mémoire capable de contrôler les interférences portant sur la mémoire, le cache et le bus. Ce gestionnaire propose une grande flexibilité de configuration afin d'être utilisable dans un contexte industriel;
2. L'étude et le contrôle des interférences liées aux interruptions en prenant en compte les autres types d'interférence mémoire présentés dans ce manuscript. À notre connaissance, aucuns travaux précédents ne traitent du sujet dans un contexte multicœurs en considérant les interférences de bus, cache et DRAM;
3. La définition formelle d'un jeu de contraintes devant être respectés durant l'exécution du système. Ces contraintes sont validées aux points clefs du système et permettent de veiller à l'isolation correcte des ressources, tout en notifiant le système en cas d'anomalies;
4. La conception de nouvelles méthodes et algorithmes permettant de guider l'intégrateur système lors de la configuration des méthodes de contrôle des interférences.

ABSTRACT

Safety-critical systems require more and more computational power while aiming to be more energy-efficient. Multicore processors are candidates of choice in this pursuit of efficiency. This type of architecture brings parallel computation, improves the system's energy efficiency and allows greater redundancy by having multiple instances of the same program running on different cores. However, multicore architectures suffer from major drawbacks when it comes to timing determinism and analysis. Due to multiple cores accessing the same resources, contention and delay-introducing behaviors appear at run time. These phenomena, called interferences, are difficult or impossible to predict and their impact on the applications' execution time highly depends on the hardware and the nature of the applications that concurrently run.

The presence of interferences in the system prevents the timing analysis of applications. This raises many challenges for safety-critical systems where isolation between applications, timing analysis and certification are required before the system's deployment. In this context, it is of primary importance to better understand, bound or remove interferences.

Current research proposes multiple ways to tackle interferences independently. Each interference channel (i.e., caches, bus, memory, etc.) is studied without taking into account the impact an interference channel can have on another. Moreover, the need to tune interference mitigation methods dramatically increases the system's deployment and configuration complexity, making this process nearly impossible to the system integrator. Hence, to prevent suboptimal performance provided by naive configurations, there is a need to provide tools and methods to guide the system integrator through this process.

In this thesis, we propose to study interferences and to design solutions towards their mitigation in multicore architectures. We aim to provide tools, algorithms and methods to guide the system integrator through the configuration process to prevent performance loss due to poor configuration. The contributions brought by this thesis include but are not limited to:

1. The integration of a complete memory interference mitigation framework. The proposed memory manager provides flexibility and fine-grained configuration options to be applicable in an industrial context;
2. The study and mitigation of interrupt interference alongside memory interferences. To the best of our knowledge, interrupt interferences have not been studied in multicore architecture while also considering other types of interference;

3. The formal definition of a set of constraints that must be respected during the system's execution. These constraints, validated at key points of the system, allow monitoring the correct isolation of resources and the notification of the RTOS in case of an anomaly;
4. The design of novel methods and algorithms to efficiently guide the system integrator when configuring the proposed interference mitigation modules.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
TABLE OF CONTENTS	ix
LIST OF TABLES	xiii
LIST OF FIGURES	xiv
LIST OF SYMBOLS AND ACRONYMS	xviii
CHAPTER 1 INTRODUCTION	1
1.1 Research Objectives	2
1.2 Thesis Contributions	3
1.3 List of Publications	3
1.3.1 Journal Papers	3
1.3.2 Conference Papers	5
1.4 Thesis Outline	6
CHAPTER 2 LITERATURE REVIEW	8
2.1 Definitions and Basic Concepts	8
2.1.1 Integrated Modular Avionics	8
2.1.2 Robust Partitioning	8
2.1.3 Cache Interferences	9
2.1.4 Bus Interferences	10
2.1.5 Memory Interferences	11
2.1.6 Peripherals Interferences	11
2.1.7 Interrupt Interferences	12
2.2 Cache Interference	12
2.2.1 Synthesis	15
2.3 Bus and Memory Interferences	16

2.3.1	Bus Interferences	16
2.3.2	Synthesis	18
2.3.3	Memory Interferences	19
2.3.4	Synthesis	20
2.4	Peripheral Interferences	21
2.4.1	Synthesis	21
2.5	Discussion	22
CHAPTER 3 METHODOLOGY		23
3.1	Exploration of Architectural Specification	23
3.2	Cache Interferences Mitigation	24
3.3	Bus Interferences Mitigation	25
3.4	Memory Interferences Mitigation	28
3.5	Interrupt Interferences Mitigation	29
3.6	Framework Integration	30
3.7	Test Environment and Validation Methods	32
CHAPTER 4 ARTICLE 1: CERTIFIABLE MEMORY MANAGEMENT SYSTEM FOR SAFETY CRITICAL PARTITIONED SYSTEM		33
4.1	Introduction	33
4.2	Background and Context	35
4.2.1	Multicore Safety Critical Systems Challenges	35
4.2.2	Interferences	37
4.2.3	Interferences Mitigation Means	38
4.3	Related Work	39
4.3.1	Cache Interferences	39
4.3.2	Memory Interferences	40
4.3.3	Bus Interferences	41
4.4	Global Memory Manager	43
4.4.1	Memory Management	43
4.4.2	Bus bandwidth budgeting	47
4.5	Safety Net	50
4.5.1	Core Verification	51
4.5.2	Runtime Verification	51
4.6	Results	53
4.6.1	Memory Allocation Overhead	54
4.6.2	Bandwidth Limitation Overhead	54

4.6.3	Safety Net Overhead	55
4.6.4	Scalability of the solution	57
4.6.5	Effect on Predictability	58
4.7	Conclusion	61
CHAPTER 5 ARTICLE 2: INTERRUPT RELATED INTERFERENCE BOUND- ING IN AEROSPACE MULTICORE SYSTEMS		62
5.1	Introduction	62
5.2	Related Work	64
5.3	Interrupt and Memory Configuration Study	67
5.3.1	Interrupt Path	67
5.3.2	Memory Configuration	68
5.3.3	Benchmarks and Exploration	69
5.3.4	Experiments	70
5.3.5	Measurements	73
5.3.6	Discussion	75
5.4	Multicore Interrupt Manager	76
5.4.1	Interrupt Manager Design	77
5.4.2	Safety Net	82
5.4.3	Analysis and Interference Reduction	84
5.5	Results & Discussion	87
5.5.1	Experiments Setup	88
5.5.2	Overhead	88
5.5.3	Safety Net Overhead	92
5.5.4	Interference Reduction	93
5.5.5	Combining Memory Configuration and Interrupt Limitation	97
5.6	Conclusion	98
CHAPTER 6 ARTICLE 3: SHARED CACHE ISOLATION FRAMEWORK FOR PARTITIONED SYSTEMS		100
6.1	Introduction	100
6.2	Background and Context	102
6.2.1	Partitioned RTOS	102
6.2.2	Shared Caches	103
6.2.3	Cache Partitioning	103
6.3	Related Work	104
6.4	An Evolutionary Approach to Cache Coloring	106

6.4.1	Cache Behavior Classification	107
6.4.2	Evolutionary Approach	110
6.4.3	Solutions Clustering	115
6.5	Results	115
6.5.1	Test Environment	116
6.5.2	Cache Behavior Detection	117
6.5.3	Behavior Guided Operators	117
6.5.4	Cache Partitioning Results	120
6.5.5	Overhead and limitations	124
6.6	Conclusion	124
CHAPTER 7 ARTICLE 4: EFFICIENT SCHEDULING, MAPPING AND MEM- ORY BANDWIDTH ALLOCATION FOR SAFETY-CRITICAL SYSTEMS . . .		126
7.1	Introduction	126
7.2	Related work	127
7.3	SMA heuristic	128
7.3.1	System definition	128
7.3.2	Evolutionary approach	129
7.3.3	Mapping and Scheduling	129
7.4	Experiments	131
7.4.1	Dataset generation	131
7.4.2	Comparison with the state of the art	132
7.4.3	Problem Scaling	133
7.4.4	Real-life application	134
7.5	Conclusion	135
CHAPTER 8 SUMMARY OF WORKS		136
CHAPTER 9 CONCLUSION AND FUTURE RESEARCH		139
9.1	Conclusion	139
9.2	Future Research	140
REFERENCES		141

LIST OF TABLES

Table 4.1	List of interference mitigation means available in COTS hardware . . .	39
Table 4.2	List of variable associated with safety net constraints	52
Table 4.3	Formalized safety net rules	52
Table 4.4	Test environment applications	59
Table 5.1	Comparison with the state of the art	67
Table 5.2	Benchmarks application characteristics extracted from performance monitoring executions.	72
Table 5.3	Execution time comparison with the multicore environment.	74
Table 5.4	Interrupt classes supported by the manager.	80
Table 5.5	System variables definitions	83
Table 5.6	Interrupt filter absolute execution times	88
Table 5.7	Interrupt filter overhead impact on interrupt handling	89
Table 5.8	Interrupt configuration for the context switch experiment	90
Table 5.9	Interrupt Context Switch execution times	91
Table 5.10	Memory overhead introduced by the interrupt manager with N interrupts and M partitions. All sizes are given in bytes (B).	92
Table 5.11	Overhead introduced by the <i>Safety Net</i> monitors.	93
Table 5.12	Interference mitigation using interrupt limitation mechanisms.	96
Table 5.13	Interference mitigation and memory configuration measurements against VANILLA setup.	98
Table 6.1	Cache behavior classification algorithm precision, recall and F score. .	117
Table 7.1	Test variable used to in our test bench.	131
Table 7.2	Execution times of SMA and SOTA (in seconds)	133
Table 7.3	Schedulability results between SOTA and SMA ($c = 4$).	133

LIST OF FIGURES

Figure 2.1	Federated architecture (a) composed of five tasks compared to its transition to IMA architecture (b). This figure has been taken from preliminary work we published in [1].	9
Figure 2.2	Definition of the major time frame as per the ARINC-653 standard. .	9
Figure 2.3	Typical multicore architecture where IO, cores and other memory components shared the same bus.	10
Figure 2.4	DRAM banked architecture. Each bank is represented as a matrix for which, the row of the last accessed data is temporarily stored in the row buffer, source of the bank interferences.	11
Figure 3.1	Overview of the proposed framework	24
Figure 3.2	Bus budget management for both cores and IOs.	27
Figure 3.3	Bus budget optimization and schedule decision process	28
Figure 3.4	Memory allocation between partitions in the system	30
Figure 3.5	Heatmap of the cache representing the ways and sets access rate. A red color represents a higher access rate while a green color represents a low access rate	31
Figure 4.1	Multi-core MAF with dedicated IO core to manage IO transactions and interrupts.	36
Figure 4.2	DRAM banked architecture. We represent banks as matrices for which, the row buffer, source of the bank interferences, stores the last accessed data.	38
Figure 4.3	Memory hierarchy with key points where our management framework applies.	43
Figure 4.4	Memory coloring layouts found in 40 different architectures. (a) allows separate allocation of banks and cache partitioning, (c) show overlapping bits while (b) and (d) show that bank coloring completely overlaps cache coloring and vice versa.	45
Figure 4.5	Two-level memory allocator flow. We use the same process for both private and shared memory.	46
Figure 4.6	Partitions' scheduling behavior when using full reclaiming mode (a), greedy mode (b), altruist mode (c) and strict reclaiming mode (d). .	49
Figure 4.7	Allocation time for different allocation sizes. The results show that our allocator allows faster allocation time.	54

Figure 4.8	Bandwidth limitation mechanism overhead depends on the timer's period.	56
Figure 4.9	Absolute overhead of the safety net compared to the smoothed overhead introduced at run-time.	56
Figure 4.10	WCET (a) and predictability improvements (b) provided by the memory manager. We compare the proposed metrics when using cache and bank coloring (CC + BC), bus budgeting (CC + BC + BB) and the initial reclaiming pool (CC + BC + BB GRP)	59
Figure 5.1	Major Time Frame scheduling for two cores and 5 partitions.	63
Figure 5.2	The RTOS Interrupt path is divided in two sub-paths to reduce interrupt's critical handling time.	68
Figure 5.3	Studied hardware architecture with two levels of cache and an embedded scratchpad.	69
Figure 5.4	Distribution of the MPEG2 execution times (in nanoseconds) with different interrupt types.	73
Figure 5.5	Predictability improvement (in %) for 3 partitions on the 11 memory configurations studied.	75
Figure 5.6	Execution times of BinSearch and MatMult depending on the memory configuration.	77
Figure 5.7	Interrupt management path in the RTOS. The manager is composed of two separate paths to reduce the interrupt management critical time.	79
Figure 5.8	Interrupt schedule. Core 0 has two partitions, each handling <i>interrupts 2 and 8</i> . <i>Interrupt 2</i> is non-maskable while <i>interrupt 8</i> is deferrable for partition 1 and cancellable for partition 2.	79
Figure 5.9	Interrupt Context Switch relative overhead depending on the number of configurable interrupts.	91
Figure 5.10	Execution times for the selected partitions.	94
Figure 5.11	Execution times for the QuickSort benchmark, with an increasing number of triggered interrupts.	97
Figure 6.1	Example of Major Time Frame with four partition.	102
Figure 6.2	Addresses representations in the system. (a) is the physical, or effective, addressee to which a virtual address is associated. (b) shows the physical address seen by the cache, showing in which group of sets (index field) the data will be loaded.	104
Figure 6.3	Cache behavior curves with (a) depicting the constant behavior, (b) the sensitive behavior, (c) the stepped behavior and (d) the undefined behavior.	107

Figure 6.4	Cache usage curve detection process for step behavior. (a) is the application's cache usage curve. (b) is the result of the convolution between the Gaussian first derivative and the curve (a). (c) is the result of the processing after obtaining the curve (b).	110
Figure 6.5	Example of a two-dimensional (a) solution space and three-dimensional solution space (b).	116
Figure 6.6	Global fitness function value over 100 successive mutations with balance (a), PSM (b), BGM (c), CIM (d), TWORS (e) and the operator proposed in [2] (f).	119
Figure 6.7	(a) Global fitness function value over 1000 iterations of the genetic process. Six different crossover operators were studied, showing the proposed operator (BGX) performs better both in convergence time and quality of the solution. (b-e) Intelligent cache partitioning applied to improve global worst-case cache hit/miss ratio.	120
Figure 6.8	(a) Execution time of the UCP algorithm compared with [2] and our approach. As [2] and our approach can be stopped at any time, we decided to stop the solution exploration when reaching at least 95% what UCP offers in terms of performance improvement. (b) Cache partitioning applied to improve global cache hit ratio while keeping fairness as second objective. We compare our results with the UCP algorithm solution.	122
Figure 6.9	Example of output for two (a) and three (b) different objectives. <i>cpu_cycles</i> refers to the execution time in CPU cycles, <i>an3</i> to the energy consumption of the system and <i>met2</i> to the number of cache eviction during execution.	123
Figure 6.10	Execution time of an applicative partition over 1000 executions. In gray (crosses) when caches are fully free, in red (circles) when we apply baseline coloring and in green (diamonds) when we apply our method to select colors size.	125
Figure 7.1	Dependency graph after validation. P_0 , P_1 and P_4 have a mapping and release constraints and P_3 is a conflicting partition.	130
Figure 7.2	Timeline with four slots in, two of which are free.	131
Figure 7.3	Comparison of the time elapsed to find a solution depending on the MAF length.	132
Figure 7.4	SMA solving time for different system configurations.	134

Figure 7.5	SMA solving time based on the number of times partitions can repeat in the MAF.	135
Figure 7.6	SMA solving time for the ROSACE system.	135

LIST OF SYMBOLS AND ACRONYMS

AER	Acquisition Execution Restitution
AMP	Asymmetrical Multi Processing
API	Application Programing Interface
CAST	Certification Authorities Software Team
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DAL	Design Assurance Level
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
IO	Input/Output
LRU	Line Replaceable Unit
LLC	Last Level Cache
MAF	Major Time Frame
MMU	Memory Management Unit
NoC	Network on Chip
PMC	Performance Monitoring Counters
RTOS	Real Time Operating System
SCE	Single Core Equivalence
SI	System Integrator
SWaP	Size Weight and Power
TDMA	Time-Division Multiple Access
TLB	Transaction Lookaside Buffer
TPU	Tensor Processing Unit
vCPU	virtual CPU
VM	Virtual Machine
WCET	Worst Case Execution Time

CHAPTER 1 INTRODUCTION

Since their public release, multicore architectures have been widely used in multiple domains. These processors offer many advantages compared to their single-core counterpart. Multicore processors provide the ability to execute multiple tasks at the same time while being more energy efficient [3]. The appeal for parallel execution, due to the increase of computational power required by systems, as well as the advantages brought by multicore processors (cost, energy and space efficiency) convinced processor manufacturers to reduce the design and production of single-core chips.

The aerospace domain moved from federated to Integrated Modular Avionics (IMA) architectures in the 1990s [4]. This transition allowed grouping multiple functionalities in one same computer. However, as of today, all certified avionic systems rely on single-core processors. Their reliability has been well proven through their extensive use. Unfortunately, the aerospace domains only represent a small part of the processors market. Thus, because of the declining the production of single-core processors, the transition to multicore processors for these domains is imminent.

Civilian aerospace applications and systems must follow strict rules to be certified and allowed for use. Multiple standards have been developed to ease the RTOS development process (e.g., ARINC-653). Moreover, documents such as DO-178 (A/B/C) for software, define objectives to reach certification. To allow the software to be distributed, it must conform to those documents requirements.

Multicore architectures provide better performances at the cost of a lower determinism. Due to the different optimization facilities present in Commercial Off The Shelf (COTS) processors (branch prediction, prefetch units, etc.) Worst Case Execution Time (WCET) computation became very difficult, if not impossible [5]. In multicore processors, multiple tasks can run concurrently. The parallel execution introduce contention and concurrent use on shared resources such as memory, buses, etc. When sharing resources such as a memory bus, two tasks from two different partitions can impact each other's execution time. This phenomenon is called interference [6]. Interferences reduce execution determinism, impact performance and prevent a correct WCET analysis.

To allow the use of multicore processors in critical systems two approaches are foreseen. The first is to take interferences into account during the system analysis to consider their impact on timing. The second approach is to bound or eliminate them to allow known analysis methods to be applied to the system. In this proposal, we choose the latter, to allow robust

partitioning of the shared resources. This approach will rely on the Real Time Operating System (RTOS) to manage the resources by either removing or bounding interferences. In 2016, the Certification Authorities Software Team (CAST) provided the community with a guidance paper, the CAST32-A [7]. This document proposes objectives to be met when certifying multicore critical systems. In this research project, we will closely consider the CAST32-A objectives alongside the developed methods. More recently, the AMC 20-193 [8] position paper was published to update the CAST32-A propositions and guidance.

In this thesis, our work leans towards the mitigation of interferences in critical multicore systems. This research is tightly coupled with the development of a real time operating system that will serve as a demonstrator of our work.

1.1 Research Objectives

The work we propose leans toward the removal or bounding of interferences in multicore architectures. We focus on resource segregation and management with optimization in the partitioning techniques. We study the existing methods and their potential integration to provide a final framework. This framework is used to guide the system designer during integration of the different applications.

The following objectives research objectives are proposed:

- Objective 1: Efficiently mitigate hardware memory interferences;
- Objective 2: Efficiently mitigate hardware communication interferences;
- Objective 3: Define tools and methods to guide system designers on the configuration of mitigated systems.

The first objective considers shared cache, bus and interconnect as well as main memory interferences. We decided to segregate the resources to remove interferences in the presented channels. We consider existing methods and extend them to make them applicable in our context. We also provide ways to optimize the configurations used with the different approaches.

The second objective consists in the same approach as the first objective while considering Input/Output (IO), peripherals and networks in the system.

Finally, the third objective focuses on the integration of the different mitigation solutions. We aim to provide a complete framework capable of optimizing the configurations as well as

enforcing robust partitioning in the system. This framework is presented in two modules: an offline optimization tool and a module embedded in the RTOS to manage the resources.

1.2 Thesis Contributions

This manuscript brings contribution to the real-time safety-critical embedded systems design and optimization domains. In this work, we provide novel insight and designs to mitigate multicore interferences, optimize the system’s configuration, guide the designers in their work and formalize specific aspects of the system’s run-time behavior.

The main contribution of this work are:

1. The design and implementation of a complete interference-aware memory and interrupt manager, capable of removing cache and DRAM interferences while strictly bounding bus contention interferences;
2. The formalization of run-time constraints, gathered in the form of a so-called *Safety Net*, applied to key points of a RTOS;
3. The definition of an optimization framework to guide the system’s designers to correctly allocate caches, memory and bus budgets to the different applications running in the system.

Each of these contributions are further described in the rest of this thesis.

1.3 List of Publications

This thesis is composed of five published and submitted papers listed in this section. The core of this dissertation is divided into two parts: interference mitigation and configuration optimization. During the research project, we published three additional papers briefly presented in this section.

1.3.1 Journal Papers

- Torres A., Lefoul J-B., Gohring de Magalhaes F., Nicolescu G. and Harnois S. (2023). Shared Cache Isolation Framework For Partitioned Systems. *Submitted in Springer - Design Automation for Embedded Systems*

Allocating cache partitions to applications can be a tedious process. A bad allocation can result in a drastic loss of performance and make the system impossible to schedule while

finding a good allocation scheme can reveal to be a complex problem. In this paper, we explore an evolutionary approach to provide the System Integrator (SI) with a nearly optimal allocation scheme. Our method is based on run time performance analysis and objective oriented optimization. We provide a complete framework to extract and process performance metrics and other relevant information at run time. Then, we use that information to compute a scheduling scheme that fits the needs of the SI. Finally, the framework provides a set of tools to visualize and better understand the generated configuration.

This paper tackles research objectives 1 and 3 and is part of contributions 1, 2 and 3.

- Torres A., Lefoul J-B., Gohring de Magalhaes F., Nicolescu G. and Harnois S. (2023). Interrupt Related Interference Bounding in Aerospace Multicore Systems. *Submitted in IEEE Transactions on Aerospace and Electronic Systems*

Interrupt management in safety-critical systems raises many challenges. By their sporadic nature, interrupts can break the deterministic behavior of the system. In this paper, we provide a novel interrupt manager that is capable of bounding multicore interference generated by interrupts. We further provide an extended set of interrupt-related constraints, called the Safety Net. Our approach classifies interrupt into cancelable, deferrable and non-maskable interrupt. The user can associate a class to each interrupt and allocate a certain budget to the interrupt. Once this budget is reached, an action is taken depending on the interrupt's class. This design provides strict bounds for the number of serviced interrupts and interrupt service routine execution time.

This paper was part of contributions 1, 2 and 3 and covers objectives 2 and 3.

- Torres A., Lefoul J-B., Gohring de Magalhaes F., Assal D. and Nicolescu G. (2019). Cache Locking Content Selection Algorithms for ARINC-653 Compliant RTOS. *ACM Transactions on Embedded Computing Systems (TECS)*.

This work was published from early work prior to this research. In this paper, we propose a set of algorithms to compute which data should be locked in private caches to optimize a given objective. The proposed framework relies on memory tracing and simulation to better understand the cache usage at run time. The use of cache locking allows reducing execution times and increasing determinism. Relying on traces provides a non-intrusive and accurate study of the system and its memory usage patterns. Our approach helps the system designers to correctly configure private caches to prevent suboptimal settings and loss of performance.

This paper was not part of this PhD, but tackles objective 1 and 3 and covers contributions 1 and 3.

1.3.2 Conference Papers

- Torres A., Lefoul J-B., Harnois S., Gohring de Magalhaes F. and Nicolescu G. (2022). Certifiable Memory Management System for Safety Critical Partitioned System. *Proceeding of the 11th European Congress on Embedded Real Time Software and Systems*.

In this paper, we introduce the core of our work. We present the memory manager used to mitigate interference and ensure the system’s correct behavior. Our memory manager provides robust partitioning while remaining deterministic in a certifiable safety-critical context. Cache and memory (Dynamic Random-Access Memory (DRAM)) interference are mitigated using memory coloring, a technique that relies on smart virtual-to-physical memory translation. We monitor and budget the use of the bus by the applications to provide strict bounds on the bus interferences. We further introduce the concept of Safety Net and formalize a set of constraints. Finally, provide insight on the implementation of run time monitors and their overhead to ensure that such constraints are respected at run time.

This paper addresses objectives 1 and 3 and contributions 1, and 2.

- Torres A., Lefoul J-B., Ben-Salem A., Gohring de Magalhaes F., Nicolescu G. and Harnois S. (2022). Efficient Scheduling, Mapping and Memory Bandwidth Allocation for Safety-Critical Systems. *20th IEEE Interregional NEWCAS Conference (NEW-CAS)*.

Providing a schedule that matches the requirements in terms of execution time, responsiveness and availability can become a challenge in multicore architectures. Using bus budgeting to bound bus interferences also implies that each application must be allocated a certain budget to access the bus. This budget impacts the applications’ execution time, which makes the scheduling and mapping of applications even more tedious. In this paper, we provide an algorithm to compute the mapping, scheduling and bus budget allocation of applications in partitioned systems. Our approach overcomes the previous work limitation by considering a set of constraints that can be defined by the user. Compared to previous work, our approach scales well with the size of the system (number of partitions and cores), which makes it applicable in an industrial context.

This paper enables contributions 1, 2 and 3 and covers objectives 1 and 3.

- Torres A., Lefoul J-B., Gohring de Magalhaes F., Nicolescu G. and Assal D. (2020). QEMTrace: a multi-platform memory tracer based on system emulation. *Proceedings of the 2020 Summer Simulation Conference*.

This work presents provides a method to use QEMU [9] as a non-intrusive memory tracer. We provide three different methods (in place traces, sockets and plain text traces) to collect the data generated during the execution. Using QEMU allows non-intrusive memory tracing without modifying the kernel or the application being traced. The tracer is controlled by the user relying on specifically designed assembly instructions. The source code has been made public and was ported to the x64, x86 and PowerPC platforms.

This paper enables contributions 1 and 3 and covers objectives 1 and 3.

- Lefoul J-B., Torres A., Gohring de Magalhaes F., Nicolescu G. Assal D. and Ulysse N. (2020). Simulator-based framework towards improved cache predictability for multi-core avionic systems. *2020 Spring Simulation Conference (SpringSim) Proceedings*

In this paper, we explore the use of cache simulation to replay memory access traces and gather crucial metrics such as the number of cache misses, cache hits, L1 and L2 accesses, etc. We provide a completely configurable, model-based simulation framework that can simulate any cache architecture. As a case study, we use the proposed framework to optimize cache locking configuration based on the cache usage of applications executing in the system.

This paper is part of the research project explored in this PhD, but is not directly part of it.

1.4 Thesis Outline

This thesis is divided in 9 chapters. The first and current chapter introduces the context of our work. We also present the research objectives as well as the proposed contributions. Finally, the chapter briefly presents the papers we published during this project.

In Chapter 2, we introduce the basic concepts studied in this thesis and provide a global literature review of the different research domains studied in this thesis. Chapter 3 provides the methodology followed during this research as well as the expected results.

Chapter 4 introduces the global memory manager we designed. This manager enables the system integrator to partition shared resources (caches, DRAM) to properly isolate these components. We further describe an improved bus bandwidth controller that meets the industry requirements. Finally, this chapter present a part of the Safety Net proposition that concerns the memory and the bus.

In Chapter 5, we propose a study of interrupt-related interferences. We introduce a benchmarking methodology to assess the impact of different memory hierarchy configuration on the number of interferences. In this paper, we show that the choice of memory architecture

must be considered when designing a partitioned system. Following the proposed study, we present our approach to bound interrupt-related interferences. We extend the existing interrupt manager embedded in an industrial RTOS to filter interrupts and bound the number of times interrupts can be serviced. We extend the existing Safety Net approach proposed in Chapter 4 by adding new constraints.

Chapter 6 presents an optimization framework used to improve cache partitioning schemes to provide the SI with cache configurations that meets their needs. The framework is composed of a measurement and data extraction module, a multi-objective optimization module and a visual configuration exploration tool, allowing an easy usage for the user.

Chapter 7 introduces our second optimization tool. This paper presents a bus budget allocation method to help the SI find a valid mapping, scheduling and bus budget allocation scheme. To allow the use of this tool in an industrial context, we consider constraints that the SI can provide prior to the optimization phase (mapping, scheduling and dependency).

In Chapter 8, we summarize our work and discuss the different results brought by this project.

Finally, in Chapter 9, we conclude this manuscript and provide future research perspectives related to the presented work.

CHAPTER 2 LITERATURE REVIEW

In this section, we present the concepts and definitions used in the rest of this document. We describe the architectures that are targeted by this research, the concept of robust partitioning as well as the interferences that we aim to mitigate.

Additionally, we present the existing work related to interference analysis and mitigation. We classified this literature review to separate each type of interference.

2.1 Definitions and Basic Concepts

2.1.1 Integrated Modular Avionics

Prior to the 1990s, avionic architectures relied on federated Line Replaceable Unit (LRU). As depicted in Figure 2.1(a), each unit would perform a given task and communicate with other units through communication buses. The federated LRU approach had the advantage of allowing the designers to replace units without impacting the complete system. However, with the growing number of computer-assisted tasks, the number of units required in the plane quickly became unacceptable. Indeed, the Size Weight and Power (SWaP) required by these units could not scale with the modernization of planes. Moreover, the increase of units also increased the contention on communication buses, which made them practically impossible to use.

To address these issues, the IMA architecture was proposed. This architecture groups multiple functionalities in one single computing unit. The main advantage of using such a design is the reduction of the SWaP introduced by the functionalities. Moreover, communications can now be internal to the computing units and only functionalities placed on different computing units would have to use the bus to communicate. Figure 2.1(b) shows the same architecture presented in Figure 2.1(a) transposed to the IMA design. Each task is now managed by a RTOS that dispatches the resource usage of each task.

2.1.2 Robust Partitioning

The notion of robust partitioning implies that an application cannot impact other applications in the system. This is true in a software manner, where an undefined behavior of the application cannot block or impact the execution of another application. This is also enforced

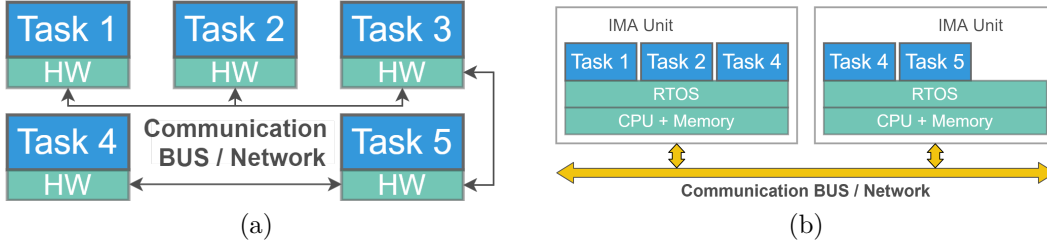


Figure 2.1 Federated architecture (a) composed of five tasks compared to its transition to IMA architecture (b). This figure has been taken from preliminary work we published in [1].

at the hardware level, where an application should not change the state of the hardware to the point that it might impact another application.

In this context, applications are called partitions. A partition may contain multiple tasks running concurrently. In this research project, we only consider interferences caused by partitions' tasks to other partitions' tasks, called *inter-partition interferences*. *Intra-partition interferences* are not considered in this work and are already well understood in our context [10].

To enforce robust partitioning, the ARINC-653 [11] standard defines the concepts of time and space partitioning. As shown in Figure 2.2, time partitioning is defined thanks to a Major Time Frame (MAF) that is repeated indefinitely. The MAF defines a static schedule and ensures strict time segregation between partitions on the same core. Space partitioning is enforced by mechanisms such as the Memory Management Unit (MMU) to allocate separate memory areas and IO to partitions.

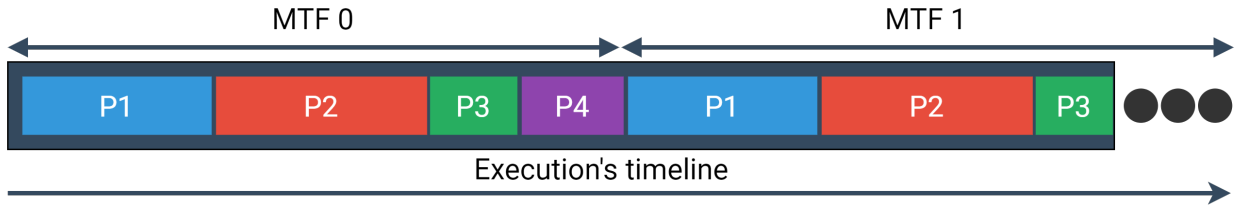


Figure 2.2 Definition of the major time frame as per the ARINC-653 standard.

2.1.3 Cache Interferences

Cache interferences in multicore systems appear in two contexts: shared caches and private caches. Shared caches interference occurs when multiple partitions share the same cache. Two types of interferences might appear in this context:

- Contention interference, when multiple cores access the cache at the same time, only one core is granted the access and the others will wait until the transaction is complete.
- Eviction interference, when a partition evicts data owned by another partition on another core. This enforces the owner partition to reload its data, which introduces delays in execution time.

The second cause of cache interference is the cache coherence feature. Cache coherence protocols are designed to ensure that data modified in the private of a core is updated in the private cache of other cores if present. Depending on the protocol, the data can be updated (which will introduce a small overhead) or invalidated in the private cache. From this point, when the core accesses the invalidated data, it has to reload it. Again, this introduces delays in execution time and goes against the robust partitioning principles.

2.1.4 Bus Interferences

Bus or interconnect interferences occur when multiple components (cores, coprocessors, etc.) access the bus at the same time. This contention results from the bus controller's arbitration. Only one component may access and process transactions on the bus at a given time. Thus, other components that requested the access will be put on hold and wait for their turn to use the bus. Different policies were introduced to reduce the average waiting time or give priority to latency critical components.

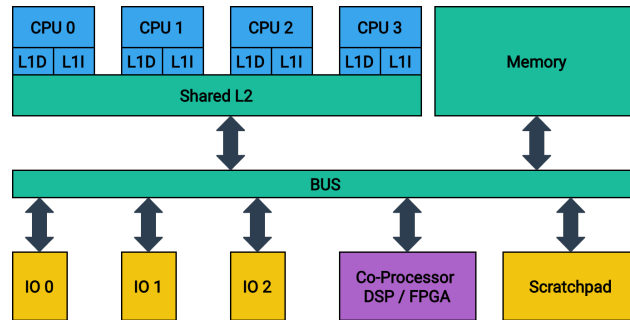


Figure 2.3 Typical multicore architecture where IO, cores and other memory components shared the same bus.

Figure 2.3 shows a typical architecture where bus interferences might occur. As depicted, those interferences can also be generated by external components such as Direct Memory Access (DMA) capable devices. Even though multiple arbitration policies have been proposed to reduce or remove bus interferences, few of them are actually available to COTS hardware.

2.1.5 Memory Interferences

Interferences on the main memory can be observed at different levels. The first memory interference is encountered at the channel level. Memory can be accessed through different channels. Each channel can be accessed independently without showing any interference. However, if multiple cores access the memory through the same channel, contention on that channel appears.

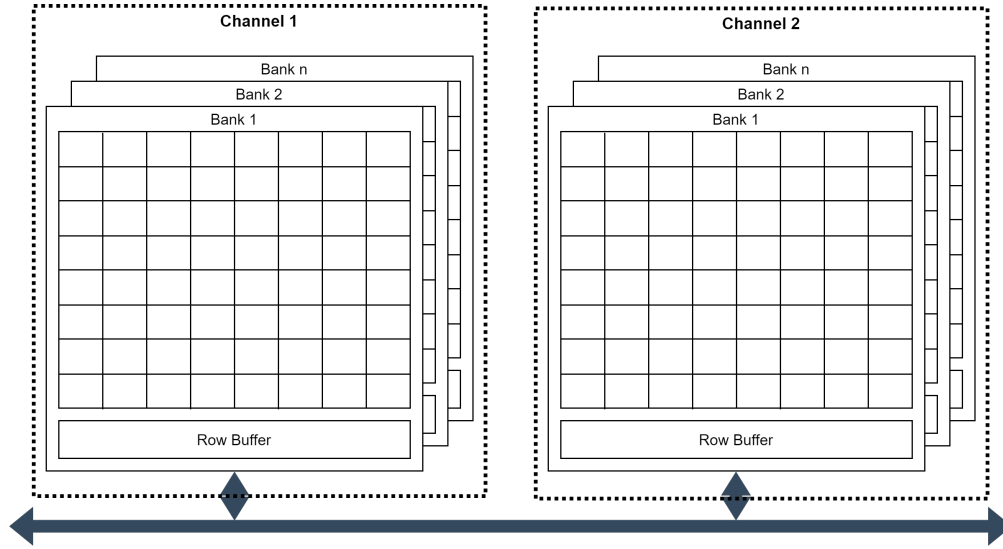


Figure 2.4 DRAM banked architecture. Each bank is represented as a matrix for which, the row of the last accessed data is temporarily stored in the row buffer, source of the bank interferences.

The second type of interference present in memories appears at the bank level. As depicted in Figure 2.4, the DRAM architecture comprises multiple banks. Each bank is composed of a row buffer, which stores the content of the last accessed row in the DRAM. The row buffer acts as a cache that allows accessing the data contained in the same row faster. When multiple core access the same memory banks, this row buffer will be unpredictably changed to accommodate for both cores access patterns. Thus, increasing the execution time of applications executing on the different cores.

2.1.6 Peripherals Interferences

When different cores access IO at the same time, arbitration on these peripherals will introduce latency for the cores that wait for their request to be serviced.

Interferences also occur when DMA capable devices are present in the system. When a DMA transaction is started, the DMA device becomes another component that has access

to the memory bus. Thus, bus interferences will be increased. Two other interferences are also present by the use of DMA: contention interferences but also interrupt interference might appear. When a DMA device finishes its transaction, the user will have the choice between polling the device to detect when the data are ready to be used. In this case, no interrupt interference can occur. However, to avoid polling, interrupt can be triggered by the DMA device to inform the system when the transaction is complete. In this case, interrupt interference appears. We discuss this type of interference in the next section.

Finally, interferences occur in extreme conditions when multiple cores can modify the state of the peripheral. In such case, legacy applications and/or peripheral might assume the state of the peripheral at the end of the last transaction will be the same at the beginning of the next transaction. This is indeed not true in the case of multicore systems, where different cores might change the state of the peripherals.

2.1.7 Interrupt Interferences

Interrupt interferences are particular in the way they can be classified as both hardware and software interferences. Indeed, the source of the interrupt might be hardware component: a peripheral that triggers an external interrupt, a timer that delivers its tick, etc. In a multicore system, interrupt routing can also be the source of interferences. When a core receives an interrupt it is not capable to service, this interrupt will be transferred to another core. This transfer time introduces latency in execution time.

Interrupts introduce delays in execution time: the RTOS need to manage the interrupt mechanism and execute the related interrupt service routine. This is analogous to interferences introduced by sporadic tasks. Moreover, interrupts need code as well as data to execute. Thus, caches, memories, buses and other hardware interferences are increased when an interrupt occurs.

Interrupt interference already existed in single-core architecture, but the indirect increase of other hardware interference it introduces makes this kind of interference critical to mitigate.

2.2 Cache Interference

Cache partitioning can be defined during the system's execution and changed over time. This method is referred as dynamic cache partitioning. It allows the system to adapt to the variation of memory usage by programs and usually achieves better results [12]. Another technique, static cache partitioning, uses a pre-defined partitioning scheme that is applied during the system's startup and will not change amid execution. Dynamic partitioning may

perform better but introduces non-deterministic cache partitioning scheme. Such a behavior prevents certification in critical systems as every configuration values should be approved and validated before the system deployment [13]. In this research project, we rely on static partitioning to comply with certification authorities' recommendations. Furthermore, we aim to provide an offline optimization tool to produce configuration outputs that respect the certification procedures.

Only shared cache interferences are presented in this document as private cache interferences have been studied in prior work. The research was conducted in our research group as a part of a master of research as an introduction to this PhD project [1, 14].

Set partitioning allows allocating cache to applicative partitions on a set basis. In [15], the authors propose an offline algorithm to define cache allocation to applications in the system. The goal of their approach is to reduce contention on the memory buses. This work relies on cache partitions for which the size is limited to a power of two. This choice allows reducing the algorithm's execution time. In this paper we propose a method that allows all cache partition size.

In [16], the Utility-Based Cache Partitioning (UCP) algorithm is proposed. This algorithm has inspired many works proposed since then. This method uses additional hardware in the system to monitor the applications at execution time. It unveiled cache partitioning as a promising method in terms of performance improvement. In [17], a semi-partitioned approach is proposed to improve execution performance. The authors rely on dynamic programming to explore the complete solution space and decide which partitioning scheme fits their need the most. This technique yields to the optimal solution but is not realizable in our context. Indeed the amount of potential solutions would require many executions on the platform and would take an unacceptable amount of time. We also aim to provide multi-objective optimization, which is not feasible with this method. In [18], another semi-partitioned caches method is proposed. The method leans towards predictability improvement and help WCET computation in real-time systems.

In [19], the authors rely on the Miss Rate Curves and Stall Rate Curves to select the size of each partition. They also propose to co-schedule applications with compatible cache behavior and provide improvements up to 17%. In this article, the authors demonstrate how to quantify the shared cache interference occurrence in a system. They advocate that the miss rate curve is not representative of the penalty of the cache interferences as misses can be serviced by different types of component. Moreover, the components can service data at different speeds. Even though their work gives a lot of information about cache set partitioning, no actual method to decide cache allocation is given.

Dynamic approaches modify the partitioning scheme at run-time, which raises determinism issues. Using dynamic partitioning will improve average performance but will lead to timing uncertainties. In hard real-time systems, this phenomenon is not acceptable. Our proposed method will rely on static partitioning, which makes it certifiable for aerospace applications. Despite this, dynamic partitioning approaches are interesting and can inspire static cache partitioning methods. In [20], the KPart tool is proposed. This method uses run-time profiling using hardware performance counters to tweak cache partition size. The granularity proposed by the tool is based on way partitioning. Similar to KPart, the work presented in [21] uses an adaptive cache partitioning technique. The algorithm tunes the partitioning scheme based on metrics gathering.

Other methods have been proposed to use cache partitioning with additional cache management methods. In [22] and [23], prefetching is used with cache partitioning to leverage cache performances. Such methods, however, rely on dynamic mechanism and cannot be exploited in our context. In [24], the authors use cache partitioning along with other processor management methods to optimize energy consumption. In [25], the principle of super pages is applied with cache partitioning to reduce execution costs implied by virtual to physical memory translation. To this end, they use additional hardware in the system. Co-scheduling is also studied to improve performance in presence of cache partitioning. In [26] the proposed approach is to use cache partition and co-scheduling of applications to select the configuration that yields to an increase of performances. In [27] and [28], virtualization techniques are used to provide cache partitioning. An hypervisor is used in [27]. This choice allows applying cache partitioning to closed source operating systems but also avoids modifying the code of each guest operating systems. More recently, [28] uses the concept of virtual Central Processing Unit (CPU)s to manage cache resources in the system.

In [29], cache partitioning is used to remove interferences in the system and reduces energy consumption. This method relies on highly configurable caches. Such caches require additional hardware to allow greater customization. Unfortunately, configurable caches are uncommon and are not available in our context. The authors rely on line partitioning to select the best partitioning scheme. Due to the size of the solution space, the authors propose to define the cache partition scheme based on a heuristic.

Machine learning has also been explored to solve the cache partitioning problem. In [30], the authors propose to overcome the hardware cost of the UCP algorithm by providing a less hardware-based approach. The approach is based on way partitioning. The authors use a trained model to predict the behavior of the cache based on the cache allocation. A reinforcement learning based approach was proposed in [31]. The authors use Q-Learning

during the system’s execution to learn how to partition the caches depending on the state of the system (applications phase for instance) and provides to the system the data to effectively apply the partitioning.

A lot of work in the literature relies on optimal solution space search. In [32], an Integer Linear Programming (ILP) algorithm is used to propose a trade-off between sharing and isolation for Last Level Cache (LLC) components. The authors advocate that even in highly critical systems, some components do not require full isolation. The approach is based on mixed criticality concepts. Their assumption is correct in some critical systems, where the level of certification is low. However in our context (highest level of criticality) regulations prevent us from using such approaches. ILP has also been studied in [33] to map applications to cores in the system as well as defining the cache partitioning scheme.

Cache partitioning can also be used to prevent attacks such as side-channel attacks. In [34–36], cache partitioning is explored to prevent such attacks and improve the system’s security. In [35], the authors discuss the possibility of using a malicious software to learn information about applications by monitoring their cache partition’s size in dynamic approaches. Other cache management techniques were proposed. In [37], the authors propose to define a cache miss threshold for each application in the system to ensure quality of service. Once the threshold reached, applications see their access to the cache disabled to ensure other applications can use it.

2.2.1 Synthesis

Many works have been proposed to leverage cache issues in multicore processors. However, a study of the current literature reveals some challenges that were not addressed. First, most approaches rely on way partitioning. This method starts to appear in multicore systems but is not available to most architectures. While other approaches rely on additional hardware to provide cache management, such hardware is either not available on critical systems or cannot be certified in such systems [38]. Most proposed work tends to optimize cache usage, using semi partitioned approaches. The notion of strict partitioning is mandatory in our context. The different certification documents, standards and guidance lean towards isolation enforcement. However, this notion of robust partitioning is usually left aside. Finally, the number of solutions for the cache partitioning problem makes it complex for systems with a large number of applications or cache partitions. Optimal algorithms will take too much time to find viable solutions for certain systems.

2.3 Bus and Memory Interferences

Different approaches have been studied to tame bus and memory interferences. While some focus on bus interferences management and others on memory interferences mitigation, other approaches study their interaction and how to treat them together. For this reason, we first provide an insight of the state-of-the-art regarding bus interference mitigation. We further complete our review by adding works focusing on memory interferences management and we finish by providing the reader with the last method mitigating both interferences at the same time.

2.3.1 Bus Interferences

Bus bandwidth management has been a widely studied approach to leverage bus interferences. In [39] the authors extend Memguard, the approach proposed in [40]. MemGuard is the basis of most current research on bus budgeting. MemGuard allows regulating bus bandwidth and propose different mechanisms to increase performances. In [41], the authors propose a scheduling framework to leverage predictability in multicore systems. The approach uses a memory bandwidth throttling mechanism to ensure bus and memory interference bounding. To prevent underutilization of the multicore resources, the authors allow co-scheduling the real-time tasks with other tasks or best-effort tasks on idle CPUs. The tasks are subject to bandwidth budgeting to bound interference they generate. The same method was applied to Linux CGroups in [42]. In this article, memory throttling is further refined to provide dynamic modification on the bus access budgets to accommodate soft real-time tasks. In [43], the authors propose to isolate bus resources when hard real-time applications execute, thus, giving them exclusive access to the bus. This is made possible thanks to the use of a specific API. This approach applies the notion of critical sections to hardware components. Recently, the work proposed in [44] describes the relation between the last level of cache usage and the bus usage. Based on the bandwidth demand, the authors optimize the budget allocated to an application.

They consider cache interference removed from LLC partitioning. The applications are clustered to reflect their use of memory (low, high). Finally, the memory throttling is applied to clusters and not applications.

In [45], the authors designed a resource management framework based on virtualization to leverage cache partitioning and memory bandwidth limitation. The approach is based on the use of virtual CPU (vCPU). Each Virtual Machine (VM) has a set of vCPU, computed by a proposed algorithm and each vCPU is allocated a certain amount of resources (cache,

memory, etc.). This implementation allows a quasi-null overhead. To decide the allocation, the authors rely on an iterative algorithm to compute the vCPU sets, allocate the resources to each set and map the vCPUs to the cores.

More recently, in [46] an evolutionary algorithm is proposed to map partitions to the available CPUs. The objective is to reduce contention on the bus by co-scheduling low bus-demanding applications with high bus-demanding ones. This approach is interesting in the way that it naturally reduces interferences without impacting performances. However, strict isolation is not ensured with this method as no throttling method is put in place. In [47], scheduling is used to reduce interferences. The authors propose a new method to dynamically manage core budgets, analyze and test schedulability of applications and provide a set of experiments to show the improvements of the method. The criticality of the applications can be modified to reduce their response time if they exceed their budget while two execution modes are provided: high and low modes. Each mode has a defined budget. In [48] this work is further refined to propose a multiframe memory bandwidth regulation to leverage the use of memory in multiframe applications. The authors propose a fitter analysis and a memory bandwidth and task-to-core assignment heuristics. The analysis is based on the memory usage and memory stall of applications. In [49], the concept of memory bandwidth server is extended with a processing server. However, only one core is used to map critical applications, leaving the other cores to best effort applications.

Execution models were studied to manage bus and memory interferences. In [50] the Acquisition Execution Restitution (AER) model was presented for critical systems. The goal of this method is to separate execution from memory access during application execution. Thus, the memory access phases can be managed by the RTOS to ensure no contention occurs. In [51] the AER model is studied alongside the Time-Division Multiple Access (TDMA) memory arbiter to provide analysis of inter-core interferences introduced by the shared memory. More recently [52] apply the AER model to the ARINC-653 compliant systems.

Analysis methods have been studied to provide tight bounds on bus contention interferences. In [49, 53, 54] the authors introduce response time analysis methods that provide precise results and a general framework to perform the analysis under bandwidth budgeting or bus contention. However, no insight on how to define the bus budgets is provided and the analysis does not take into account constraints applicable in the aerospace domain such as partitions dependencies, IO management, etc.

Other works rely on hardware mechanisms to reduce, bound and remove interferences. In [55] a memory controller is proposed to improve predictability while increasing the performance. In [56, 57] a memory arbiter achieves the same goal. Other works such as [58] propose to tame

Network on Chip (NoC) interferences by adding additional controllers to shape the traffic on the network. However, these approaches are not compatible with our decision to use COTS as they require additional hardware that is not certifiable in some cases. In [59] three techniques are studied to manage memory bandwidth: thread packing, clock modulation and Intel MBA technology. Dynamic optimization is done at run-time based on online profiling and configuration update to find better performances.

The approach of [41] is interesting for mixed criticality systems with soft real-time tasks. However, in our context, all partitions must be fully isolated and thus, no partition can be seen as “best-effort tasks”. Furthermore, the authors do not provide any means to compute the budgets to set with the throttling mechanism. Furthermore, we cannot rely on dynamic budgets as in [42]. Indeed, it would introduce non-deterministic execution time and prevent certification. The exclusive use of resources at a given time such as the method proposed in [43] cannot be applied. Indeed, all the partitions in the system are considered as highly critical. Applying this method would make the use of multicore useless as only partition could execute at a time. In [60] a software approach is designed in the same way to control accesses to the bus. The approach provided in [47] reduces the overhead and increase the performance but we cannot afford to change the criticality of partition (considered as Design Assurance Level (DAL) A) nor provide different execution modes. Unfortunately, the hardware components used in [44] cannot be used in our context. Moreover, contrary to the proposed method, isolation must be maintained at any moment in the execution window. Finally, our goal is to reduce applications WCET contrary to optimizing the average performance like what was proposed. Work presented such as [59] do not provide strict isolation of resources and thus cannot be applied.

2.3.2 Synthesis

While virtualization allows a better management of resources, this method is not applicable in our context. We aim to rely on a bare-metal RTOS that does not allow virtualizing the system’s resources. Moreover, contrary to what was presented, we cannot afford to partition shared caches on a per-CPU basis and require each partition to have its own cache section.

Using the AER model is not applicable in our context. Indeed, it needs the user’s applications to be compatible and rely on this programming model. However, we need to keep compatibility with legacy applications that do not make use of such model. This prevents the use of such model since all applications must follow it to allow the system to control memory accesses.

Moreover we cannot afford to use specific API as we aim to make interference mitigation

as transparent to the user as possible. Finally, optimization approaches lack of budget computation method scale with the size of the system.

2.3.3 Memory Interferences

In [61] a comprehensive list of memory interference in COTS platform is given. The authors propose a methodology to analyze and understand the impact of the different mechanisms in DRAM on predictability and latency. In [62] the authors propose a multi-policy resource allocation method. They group DRAM and cache partitioning to leverage interferences in the system. Their approach relies on the analysis of a dataset gathering 2000 execution workloads (over 10000 experiments). From this dataset, they extract rules to be used when partitioning cache and DRAM at the same time. They also propose a random interleaved memory allocation scheme. This allows to uniformly dispatch application addresses across all shared dram banks.

In [63] and [41] scheduling is explored to reduce interferences in the memory hierarchy. The approaches allow reducing contention on the shared resources and thus, naturally increasing the throughput of tasks. However, in [41], only one application is able to execute on multiple cores. This is not acceptable in our context, where the trend is to put one application per core. Recently, in [64], the authors propose a scheduling strategy to manage memory accesses in heterogeneous systems. They rely on dynamic channel partitioning to remove interferences and schedule memory accesses based on this approach. They further use bank partitioning to reduce inter-CPU interferences.

Even-though scheduling reduces interferences, it does not ensure isolation between the partitions. This cannot be accepted in our research domain as it does not follow the aerospace robust partitioning guidelines [7].

To sample applications' metrics, the authors rely on the page granularity to find which page is accessed the most. The *accessed* bit sets by the x86 CPUs and polling are used to count the number of times a page is accessed. Later, in [65] the authors propose a software memory coloring approach to separate applications' memory between DRAM banks and channels. The method relies on a dynamic approach that will change the number of allocated banks and channels to threads at run-time. This approach, however, cannot be applied to hard real-time systems as the dynamic behavior would introduce non-determinism and impossibility to analyze the system.

In [66], the authors coordinate the use of cache coloring and bank coloring. The duality of the approach and the conflict between cache and bank coloring is explained. Since bank bits

are not often disclosed, the authors rely on an improved method to detect the bank bit.

In [67] the authors propose a bank partitioning method to improve performance while taking into account the applications' profile. Two categories are presented: low locality applications that will share banks to improve parallelism and high locality applications for which banks will be monopolized. The method relies on application profiling and clustering to categories their memory usage. Once clustered, each thread will be assigned banks depending on the cluster's assignment policy. This process executes at run-time to adapt when needed. Applications profiling is supported by additional hardware to determine metrics (row buffer hits, etc.). In our context, such approach would need extensive redesign to allow full isolation of the tasks. Moreover, the additional hardware cannot be used to gather the metrics needed by the online algorithm. Finally, we cannot afford to share banks between applications as full isolation must be respected.

A second interference mitigation means for memory is channel partitioning. In [68] the authors propose to partition the memory, to allocate interfering applications different to memory channels. The approach gives a good insight of how to map application to memory clusters and cores. However, the use of additional hardware prevents using this method in our context.

2.3.4 Synthesis

While memory interference mitigation has been studied in the past, the current methods must be extended to be integrated in our context.

Using scheduling or execution models such as AER is not feasible due to the lack of flexibility when it comes to scheduling (for instance, intra-partition scheduling is not known). Moreover, using the AER model would break retro-compatibility with existing applications. Some scheduling approaches, limiting the number of critical partitions executing at the same time to one, are not acceptable. Indeed, in the context of the RTOS developer, all partitions must be seen as critical. Using multiple cores in this context is pointless.

Other methods propose the use of additional hardware. As we discussed previously with other interferences, we aim to rely on COTS hardware. Additional hardware is complex to use on such systems and would require a certification overhead that is not acceptable.

Finally, work such as [66] is not yet scalable with the number of cores and partitions that we target. This is why we aim to provide meta heuristics to solve the partitioning problem.

2.4 Peripheral Interferences

In [69] the authors address the IO access issues by using IO partitions. Once the IO partitions are created, accesses are serialized through partitions' scheduling. Better performance is achieved using constraint programming to decide the best scheduling scheme. To ensure that only one IO partition can execute at a time, all IO partitions are mapped to the same core. A similar approach is proposed in [70]. The authors design a dedicated IO scheduling unit. To allow better performance, two scheduling approaches are proposed. This work is said to be the first to guarantee timing predictability. However, the additional hardware required prevents the use of such a method in our context. In [71], scheduling is also used to reduce interferences on hardware resources. Based on a classification of applications, the authors proposed a scheduling scheme ensuring that each class executed exclusively. The method allows the removal of inter-class interference in the system. This work is not aimed at IO interference management only, but at all hardware resources.

Additional hardware has also been studied to manage IO interferences. In [72] a virtualization layer is added to real-time hardware bridges to control the use of the bus by IOs. The proposed method allows strict bandwidth reservations, which implies high predictability. The bridges, proposed in [73] can manage the incoming and outgoing traffic and are hardware-implemented. Two software interfaces are used to manage the traffic too, one implemented on the real-time bridges, controlling the incoming traffic and one on the system for outgoing data. The IO accesses are then scheduled through an IO scheduling module in the RTOS.

The use of virtualization also helps leverage IO interferences. In [74] multiple virtualization methods are presented. Such methods allow abstracting the use of hardware and increase the control that the user and the RTOS have on such components. Other virtualization approaches have been proposed such as in [75] where additional hardware is used to manage IOs in the system. In [76] a hardware hypervisor is presented to manage IOs. This work takes place in the many-core domain, which is quite different from the multicore architectures.

2.4.1 Synthesis

As for the previous interferences, we find the same limitations of current works on peripheral interference. Current methods rely on additional hardware, scheduling techniques and virtualization approaches such as hypervisors. Our aim is to provide purely software peripheral interference mitigation to a bare metal RTOS with no support of virtualization. This path has yet to be explored.

2.5 Discussion

With the current state of the art, it is possible to precisely understand multicore interferences. Many works have been proposed to efficiently tackle each interference in the system. However, their integration is sometimes complex to execute in hard real-time systems [66]. The Single Core Equivalence (SCE) framework [77] has been proposed to this extent. However this framework gathers different mitigation methods and integrate them together without exploring the possible optimization when putting them together.

In this project, we aim to extend existing mitigation methods to make them certifiable in a safety-critical environment. Each mitigation method should consider that all applications (or partitions) in the system are of the highest criticality. We further extend the state of the art with new data and metrics gathering tools. Such tools allow a better profiling and understanding of the system's behavior.

Finally, we focus on configuration optimization and automation. Most of the configuration process (schedule definition, resource partitioning) is left to the SI. However, with the growing number of variables to consider in the multicore architecture, the generation of the system's configuration becomes impossible. Many factors would lead the SI to produce sub-optimal configurations. This is why we propose to automatize most of the process while providing optimization methods and a tool to explore the different solutions available when configuring the system.

CHAPTER 3 METHODOLOGY

In this section we present the methodology followed in this project. The main goal is to propose new solutions for interference mitigation. These solutions are to be integrated in an ARINC-653 compliant RTOS deployed on COTS hardware.

We propose optimization tools to provide the SI with configuration for these mitigation means. Each interference is tackled with regard to other mitigation means that are already implemented or will be implemented in the system. We aim for maximum isolation of resources as requested by the CAST-32A document. If isolation is impossible, we provide solutions to analyze interferences and bound them to provide strict guarantees on execution times and resource usage.

These objectives are based on memory and communication interference mitigation and the design of tools to configure and optimize the mitigation means.

Figure 3.1 shows an overview of the research framework we propose. We show the dependency mitigation methods have on the others with the red arrows. The blue arrows are used when different mitigation methods must be considered together. Each mitigation method is further explained in the rest of this section.

3.1 Exploration of Architectural Specification

The first phase of the research project is to explore the different hardware features that can be considered as interference channels. The literature already gathers multiple surveys on the topic. We classify interferences and propose different ways to assess them, understand their impact and mitigate them. Our goal is to rank the interference channels by order of criticality.

The first direction we take is to mitigate interferences from top to bottom of the memory hierarchy. This phase requires a deep analysis of COTS hardware features such as cache locking abilities, cache way partitioning, performance monitoring counter availability, etc. For each interference channel, we discuss the constraints the commercial RTOS imposes to the research in terms of mitigation and optimization. Indeed, multiple constraints may emerge from the domain of critical applications:

- Lack of control on memory accesses (AER models cannot be applied for instance);
- Highly constrained timings with data dependencies between tasks;

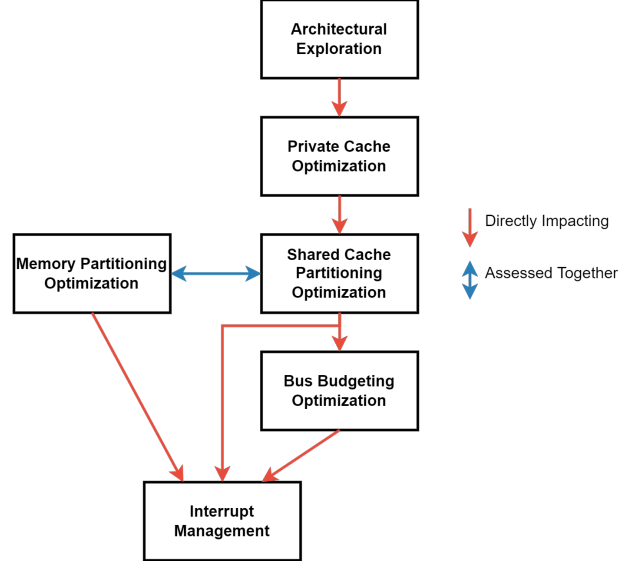


Figure 3.1 Overview of the proposed framework

- Need to support legacy applications.

Furthermore, we keep in mind that the mitigation methods should only be implemented at the RTOS level while respecting the API provided by the ARINC-653 standard. Hence, the implementation of mitigation methods must be transparent to the RTOS users.

Optimization approaches must also be easy to integrate with the design tools provided to the SI. We cannot rely on unrealistic assumptions that the literature usually relies on (all timing is known, only one core should be isolated, etc.).

3.2 Cache Interferences Mitigation

Cache interference mitigation concerns two distinct but complementary aspects: (1) shared cache interference mitigation and (2) private cache interference mitigation.

Cache partitioning is used to mitigate shared cache interferences. We first decide which type of partitioning should be used in our context. Set and way partitioning allow coarse cache partitions, while line partitioning allows finer management. The overhead of the solutions as well as the capabilities of the hardware (considering that way partitioning is a hardware-based feature) are discussed. After presenting our cache partitioning approach, we focus on configuration optimization. Such configuration should define the size of each cache partition allocated to each application in the system. This choice is crucial in terms of performance. Cache demanding applications will suffer from small cache partition allocation,

while streaming applications will trash the cache without profiting from its benefits. Finally, small applications may fit in private caches, thus not using their allocated shared cache.

In this project, we study private cache interferences and mitigate their impact. For this purpose, we rely on cache locking. Cache locking allows loading data in caches and preventing them from being evicted. This method will make cache more predictable while improving performances if the configuration is correctly done. In [1] we explored cache locking prior to this research project.

3.3 Bus Interferences Mitigation

As presented in 2, multiple mitigation methods have been proposed to control the contention on shared buses. Our approach uses bus budgeting to control cores bus usage. We base our implementation of the budgeting technique on MemGuard [39]. This technique relies on two mechanisms:

- A periodic monitor that replenishes the budget of each core;
- A sporadic handler that schedules a kernel process when a core exceeds its allocated budget.

We rely on a software approach with widely used Performance Monitoring Counters (PMC) for two reasons:

- Additional hardware would require additional certification processes, which would not only increase the cost of the hardware itself but also the complete deployment process;
- Adding hardware resources does not fit with our approach which is based on fully COTS hardware.

The sporadic handler is implemented with the use of PMC. This kind of counter is widely implemented in COTS processors which comfort us in our decision to use them as a portable system profiling tool. The PMC are used to account for the number of bus accesses made by the running partitions. Once the PMC reaches a defined value, an interrupt is generated. This interrupt is handled by the system and used to schedule the kernel process in charge of executing the bus budgeting manager. From our preliminary architecture exploration, each targeted architecture offers PMC capable of interrupts and counting the number of bus accesses. Moreover, we target intra-core PMC. Intra-core PMC are private, which allows each core to manage its own budget and reduces the implementation's overhead. This approach

also has the advantage of removing the need to synchronize cores or share resources for this task.

In our context, the periodic server is executed in a kernel process, managed by an external timer interrupt.¹ The global timer is shared among the cores to allow seamless synchronization between them. The timer starts the periodic budget replenish process. Such process, based on the definition of MemGuard, is in charge of replenishing the budget of the partition by writing specific values in the PMC. The process is also in charge of predicting the partitions' budget usage based on its previous execution. Thus, when the partition is predicted to use less than its allocated budget, the slack budget is transferred to a global pool used for further optimization explained in the next paragraph. Once all cores have replenished their budget, the process stops and the partition execution resumes.

To reduce the overhead of budgeting, MemGuard proposes two features:

- When all cores are blocked because of budget exceeded, a routine is triggered to replenish all cores' budgets and execution is resumed;
- A budget reclaiming feature. When a partition is blocked because it exceeded its budget, it can reclaim slack budget from the global budget pool.

We propose to implement those features by adding some modifications to allow full isolation between partitions (as well as certification objectives to be met). We define four modes for budget management. Each partition can define its own mode.

- Full features mode, which allows a partition to reclaim budget and provide budget to the global pool. This mode can be used for soft real-time partitions. We also propose to further constrain the budget prediction to provide bounds on the budget removal of the partition;
- Greedy mode for which a partition will not provide any budget to the global budget pool but is able to reclaim budget from this pool. This mode is suitable for any hard and soft real-time partitions, however, hard real-time partition are more prone to use this mode;
- Altruist mode, which allows a partition to provide budget to the global pool but not reclaim budget for this pool. This mode is suitable for best-effort applications that can run in degraded mode;

¹Due to the tick-less nature of our prototype RTOS, we cannot rely on the system's ticks to manage such a periodic process.

- Strict mode, for which budget reclaiming and providing budget to the global pool is deactivated.

We extend the global pool by providing an initial budget to it. For instance, in a system with partitions P1 executing on core 0 and P2 on core 1, instead of giving P1 45% of budget and P2 55%, we would provide P1 with 40%, P2 with 50% and initialize the global pool with 10% of the available budget.

Finally, the only need for shared resources used by bus budgeting is the shared timer, the shared budget pool and the global unlocking mechanism. Reducing the number of shared resources allows reducing the overhead of the solution and ensures a better isolation between cores.

We extend the concept of bus budgeting to other components such as peripherals. Indeed, the current literature uses MemGuard to provide bus control on cores. However, peripherals (and IO) also require the bus to perform tasks such as using DMA to load data to the memory. We control the peripheral access relying on global PMC offered by the system on chip. We propose the use of a virtual core to schedule IO and assign budget to IO partitions on this virtual core. The Figure 3.2 shows the implementation of our approach. The virtual core can be managed by any of the actual core. To further reduce the interference we propose two schemes:

- On architecture with more than two cores, a dedicated core is in charge of managing the virtual core (budget interrupt, PMC management, etc.).
- On architecture with two cores, the virtual core is managed by one of the two cores. Specific windows are dedicated in the major time frame of the core to manage the virtual core.

The second contribution we propose in this research is a decision tool aimed for the SI.

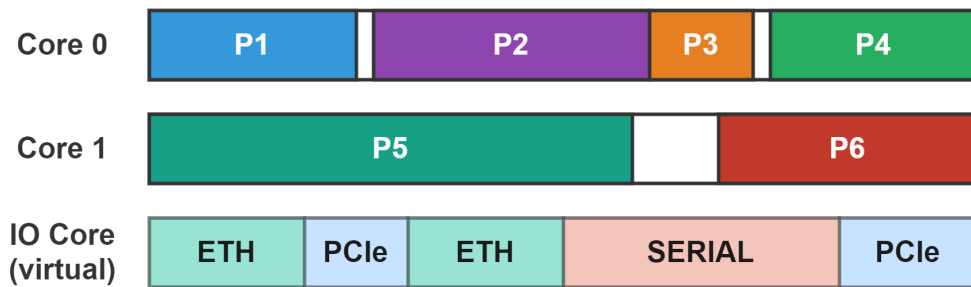


Figure 3.2 Bus budget management for both cores and IOs.

Budget must be chosen for each partition. Using a 50% / 50% approach would yield to inefficient performance, thus the need for an optimization tool is evident. To this extent, we propose a tool capable of providing the following guidance to the integrator:

- Provide the set of budgets per partition required to meet the constraints (timing, resource usage, etc.) while maximizing an objective;
- Provide a fixed schedule that satisfies the use of the proposed budgets

To perform the optimization, we rely on two exploration phases that are interleaved. First, we need to find a set of budgets to be allocated to partitions. This set is constrained by the fact that the sum of budgets of two co-running partitions should not exceed 100%. This constraint is validated during the fixed schedule research. Thus, the budget optimization algorithm has to work in coordination with the schedule search algorithm. Figure 3.3 provides a view of this process.

3.4 Memory Interferences Mitigation

Memory interferences occur at different levels of the memory. As presented in previous sections, we can separate them in two main categories: banks and channel interferences.

Our methodology to tackle interferences is the following:

1. Implement banks and channel partitioning in the RTOS with configuration means to control the partitioning;

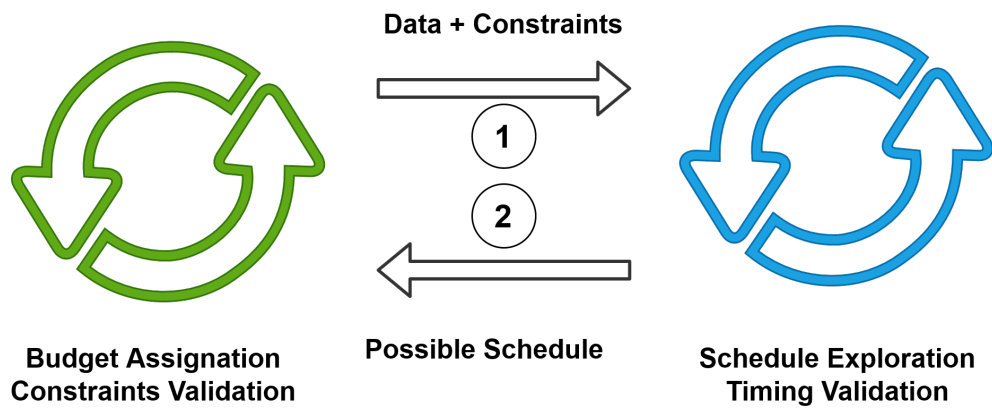


Figure 3.3 Bus budget optimization and schedule decision process

2. Provide the RTOS with analysis tools to understand and profile the interferences as well as the improvement brought by partitioning methods.

Thanks to the principle of static scheduling, we know which applications run concurrently and can allocate them different banks. However, two applications that will never run concurrently can use the same banks and channels. Figure 3.4 illustrates a potential allocation scheme we aim to deliver to the SI.

3.5 Interrupt Interferences Mitigation

Interrupt interference mitigation takes place at different levels in the RTOS. First, the interrupts that are internal to the RTOS (timers, Transaction Lookaside Buffer (TLB) miss, etc.) and happen in one core should not impact the other cores by accessing shared resources. Moreover, global interrupts should be correctly routed to the destination cores to avoid core passing interrupts between each other. Second, user interrupts, generated by either software or hardware external peripherals should be correctly routed and must not impact other cores. Finally, interrupts can occur at any moment in the execution of the system. This means a partition's interrupt routine could be triggered during the execution of another partition. This type of interference already exists in single-core systems but is amplified in multicore architectures.

In this research, we study the following steps to reduce interrupt interferences:

- Analyze the impact of interferences. In this step we will define key metrics to characterize interferences and their impact on the system. We will study the resources that are being used by interrupts and understand the different ways to mitigate them.
- Define RTOS design that reduces interferences by construction. We will provide a set of best practices to lower the interrupt interferences by applying specific design decisions to the RTOS. Such design could be the isolation of interrupt memory in caches, the locking of interrupt routines in CPUs private caches, etc.
- Research and implement an interrupt manager, integrated in the RTOS in the form of a server. This server manages the interrupt routing and interrupt passing defined in the previous steps and limits or monitors the number of accepted interrupts in the same way bus budgeting does for memory accesses.

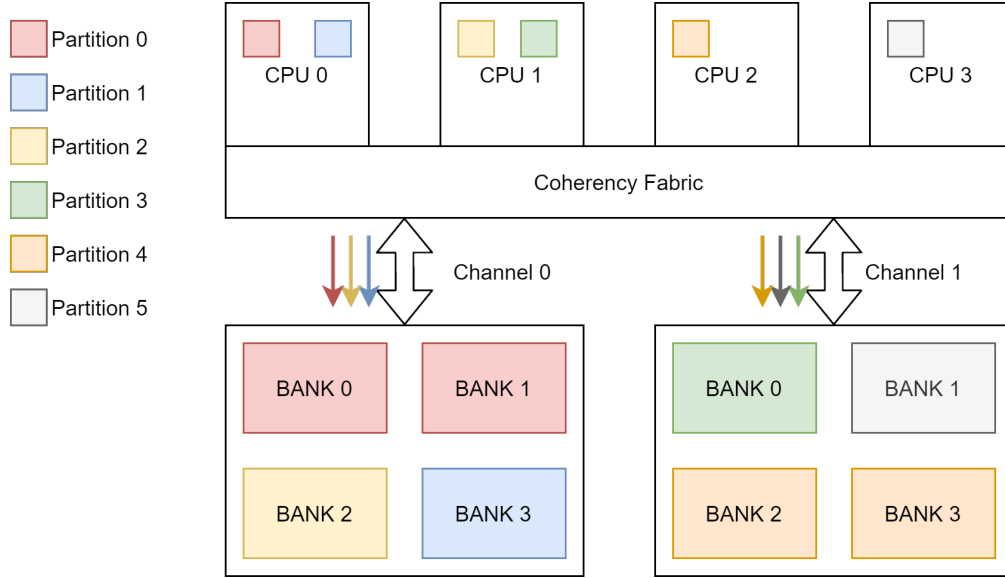


Figure 3.4 Memory allocation between partitions in the system

3.6 Framework Integration

One key point of this work is the integration of the different mitigation means and optimization tool in a global framework. This framework is aimed to help the SI when designing the system and choosing the configuration to reduce interferences. The configuration of a partitioned system requires knowledge of the complete system. This can become very challenging when interferences need to be mitigated. Indeed, the addition of new configuration constraints and data makes the process even more tedious. Moreover, current industrial processes require the SI to manually provide configuration and scheduling. This process is slow and might lead to sub-optimal configuration and resource usage. Our framework eases the design process by providing the SI with tools and profiling methods.

The goal of the framework is two-folded. One part is embedded the RTOS to enforce resources isolation (cache partitioning, bus budgeting, etc.) and profile the system for the optimization phase. The second part is composed of offline tools that consume the profiling data to either optimize resource usage or provide the SI with information to better understand and configure the system.

Embedded Modules in the RTOS are integrated from the previously mentioned contributions. The first set of modules is used to gather run-time information of the system. This comprises the following:

- Cache profiling data is gathered and synthesized to provide the following information: cache hit/miss ratio, amount of memory accesses in private and shared caches. We also provide cache locality information and a heatmap of the cache. Such heatmap is depicted in Figure 3.5. It allows understanding where the data are located in the cache to better analyze how to rearrange memory to provide better performance.
- Memory profiling data are also provided to the user. As for the cache profiling module, a heatmap is derived from the collected data to help the user and the optimization tool to define efficient resource partitioning schemes.
- A bus tracer monitors the traffic on the bus and retrieve data. This module allows analyzing the execution of a partition according to the budget it is allocated. The goal of this process is to get the execution time of a partition based on the amount of access it is allowed on the bus. This metric is used in the budget optimization tool to provide an optimized budgeting scheme. Alongside this module, we will provide the framework with a set of benchmarks to derive the practical bus bandwidth. This choice comes from the fact that platform bus bandwidth is often not disclosed to the user and might not fit the actual practical bandwidth.

The second set of modules comprises the different mitigation proposed in the previous section. A particular attention is given to the compatibility between these modules. It is important ensure that the modules are certifiable while reducing their overhead in the system. The final goal is to provide resource isolation that is totally transparent to the user, meaning that no additional code will be required in the applications.

Offline Tools The offline tools are divided in two categories: the optimization and decision tool. Such approach reduces the workload of the SI by automating multiple configuration steps. We provide the following configuration:

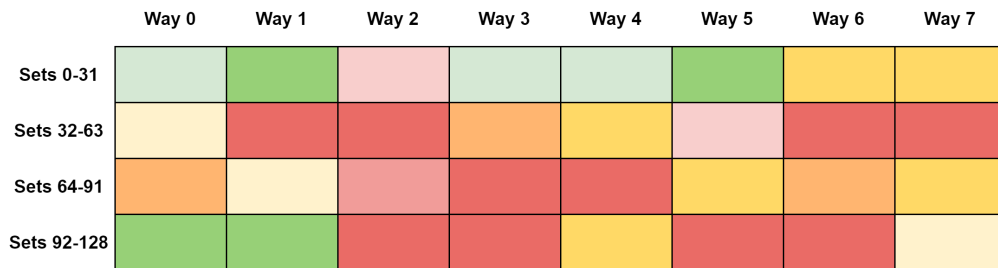


Figure 3.5 Heatmap of the cache representing the ways and sets access rate. A red color represents a higher access rate while a green color represents a low access rate

- Cache locking and partitioning schemes;
- Bus budgeting scheme;
- Memory partitioning scheme;
- Fixed schedule (MAF) definition.

The framework bases its decisions on the constraints and objectives the SI inputs as well as the run-time data gathered thanks to the embedded modules. Finally, the framework’s output consists of a set of configuration files and can be interfaced with existing tools through avionic standard such as AADL [78].

3.7 Test Environment and Validation Methods

Our test environment is based on an existing ARINC-653 compliant RTOS [79]. The software is provided by the industrial partner. This RTOS is currently available as a single-core version on different targets. Our test environment is based on the NXP T2080 RDB-PC board. This hardware provides four cores running at 1.8GHz and 4GB of internal RAM.

In order to validate our approach, the following steps are required:

- Port the industrial RTOS to the T2080RDB target;
- Enable multicore execution of the RTOS on the target. The selected multicore architecture is Asymmetrical Multi Processing (AMP), which defines that each core executes its own instance of the RTOS;
- Inter-core communication was designed and implemented to ensure a correct execution and synchronization of both cores.

To validate the mitigation methods, we rely on public benchmark suites such as Parsec [80] or TacleBench [81]. We also use a suite of ARINC-653 compliant benchmarks to specifically stress precise resources in the system. Such benchmarks allow us to ensure the isolation of the resources. This set of applications are designed in parallel to this research project in our research group. We rely on the “adversarial” approach where the stressing applications are developed to break down resource isolation.

CHAPTER 4 ARTICLE 1: CERTIFIABLE MEMORY MANAGEMENT SYSTEM FOR SAFETY CRITICAL PARTITIONED SYSTEM

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Felipe Gohring de Magalhaes, Gabriela Nicolescu and Serge Harnois

Published in IEEE - EMBEDDED REAL TIME SYSTEMS 2022, Publication date:
2022/06/01

Abstract: Aerospace systems are safety-critical systems that need to respect tight constraints in terms of execution time, resource usage and predictability. This industry is currently transitioning from predictable single-core processors to less predictable multi-core architectures. This transition reveals multiple challenges due to interferences. The contention of different cores on shared resources introduces interferences. This phenomenon prevents the required isolation between applications and the estimation of their worst-case execution time. To prevent interferences and ease the certification of robust partitioned multi-core systems, guidance documents, such as the CAST-32A, provide objectives on resource isolation and management. In this paper, we propose a memory manager to mitigate memory interferences generated in shared cache, main memory and memory bus. Our results show an increase in timing predictability by 68.1%. Aside the memory manager, based on our results, we provide a set of recommendations in order to assist system integrators' decisions and ease the certification process by conforming to the current guidance.

4.1 Introduction

The aerospace domain relies on highly critical software to ensure the reliability and safety of the system. The first generation of computer-assisted functionalities in planes is based on the federated architecture [4]. This type of architecture is designed such that each computer-assisted functionality has its controller or computer, called Line Replaceable Unit (LRU). LRUs communicate through different networks and buses. Due to the increased number of computer-assisted features in planes, the use of federated architecture is impossible to sustain. The increase in communication nodes often results in network contention and poor fuel efficiency of the plane due to the equipment weight [4].

To leverage the size, weight and power (SWaP) of federated architectures, the Integrated Modular Avionics (IMA) architecture was proposed. The IMA architecture is designed such

that multiple functionalities are gathered in the same module. This design eases the communication between applications, reduces the SWaP and facilitates the maintenance of the equipment [4]. Multiple IMA units can be scattered in the plane and communicate via networks and buses.

IMA architectures rely on Real Time Operating Systems (RTOS) to manage the different components in the system and the different applications running concurrently. Real time systems are subject to strong constraints in terms of execution time and response latency. Hence, it is imperative to ensure the Worst-Case Execution Time (WCET) of all applications. Nowadays IMA architectures are implemented relying on Commercial Off-The-Shelf (COTS) hardware [4]. This reduces the cost of the equipment but also reduces the design's flexibility.

Despite the determinism of single-core processors, more energy-efficient and powerful multi-core architectures slowly replace them [6]. Multi-core processors provide better performance by the ability to execute multiple applications at the same time. With the increasing production of multi-core architectures, single-core processors slowly disappear from the market. Aerospace system designers need to transition to multi-core architectures to keep the equipment up to date.

However, optimizations brought by multi-core processors make execution times less predictable and impact WCET. The execution of multiple parallel applications also introduces contention on the shared resources. These phenomena are called interferences [6]. Interference channels exist at different levels of the architecture. In this paper, we focus on memory interferences for which channels are the shared caches, shared memory bus and shared DRAM. We explain how interferences occur in those channels in Section 2 of this paper.

In this paper we propose a memory manager to mitigate memory interferences. For this purpose, we extend the cache, memory and bus management methods proposed by the Single-Core Equivalent framework [82] to allow better integration with certifiable RTOS and without the use of virtualization. We propose a flexible design to accommodate with different architectures and certification processes. The system integrator (SI) has the task to configure the system and integrate all applications from different providers in the same system. Our results show an increase in timing predictability by 68.1% on average while increasing the execution time of applications by 22.3% on average. We propose a set of rules and insights to the SI to improve safety and performance of the system [7]. Our work brings the following contributions:

1. We propose a novel memory manager enabling to reduce cache and memory interferences as well as the bus interferences. By integration of most efficient methods for

cache and memory partitioning and bandwidth management, we obtain a solution applicable to certifiable systems in a context where hardware-assisted virtualization is not available;

2. Improving the bus bandwidth management by proposing an extension of MemGuard [39], a memory bandwidth manager, to allow more flexibility and IO bandwidth management by extending configuration capabilities, per application budgeting, and an initial budget pool;
3. The proposition of guidelines, named safety net as per CAST-32A [7], to ensure quality of service (QoS) and safety in the RTOS and ease the certification process in avionic systems. We further formalize the notion of robust partitioning for shared memory and bus.

The rest of this paper is organized as follows: in Section 2 we present the context and the motivations of this work. In Section 3 we provide an insight of the current state of the art and the different techniques used to leverage multi-core interferences. In Section 4, we present our integrated memory manager while in Section 5, we propose a set of safety net rules to ensure safety in the RTOS and ease the certification process. Finally, in Section 6 we demonstrate the efficiency of our method and compare it to the state of the art. We also discuss the overhead of the solution and the different ways to further reduce it.

4.2 Background and Context

To allow their certification, civilian aerospace systems must follow strict rules. Multiple standards exist to ease the development process and RTOS use (e.g., ARINC-653). In this section we present those requirements and the current challenges that prevent certification in multicore systems.

4.2.1 Multicore Safety Critical Systems Challenges

The notion of robust partitioning implies that an application cannot impact any other applications in the system. This is true from a software perspective, where the undefined behavior of the application cannot block or impact the execution of another application [11]. This is also enforced in the hardware aspect, where an application should not change the state of the hardware to the point that it might impact another application. In this context, applications are called partitions [11].

ARINC-653 and CAST-32A [7] compliant systems must conform to the following constraints:

- Time isolation: a partition must execute during a given time slot without being pre-empted. As shown in Figure 4.1, the scheduling relies on a MAJOR time Frame (MAF) that is repeated indefinitely. This static schedule ensures time segregation between partitions.
- Space isolation: a partition can only access resources it has been explicitly allocated. Mechanisms such as the Memory Management Unit (MMU) enforce space partitioning.
- The CAST-32A paper also proposes the integration of a safety net. The safety net should provide means to monitor and recover from failure in space and time isolation in the advent of software or hardware flaws.

To allow the use of multi-core processors in critical systems two approaches are foreseen. The first is to take interferences into account during system analysis to consider their impact on timing. With this approach, one could account every access to the cache as a cache miss. This provides overly pessimistic results. The second approach is to bound or eliminate them to allow the use of known analysis methods. In this work, we allow robust partitioning of the shared resources. The Certification Authorities Software Team (CAST) published a guidance paper, the CAST-32A, that provides objectives to be met when certifying multi-core critical systems. However, objectives provided by the paper are abstract (for instance “Robust Time Partitioning (on an MCP) is achieved when, as a result of mitigating the time interference between partitions hosted on different cores” [7]). Thus, we propose to formalize these objectives regarding shared memories and bus for robust partitioned systems in Section 5. The design we propose in this article allows easier certification, following the CAST-32 recommendations.

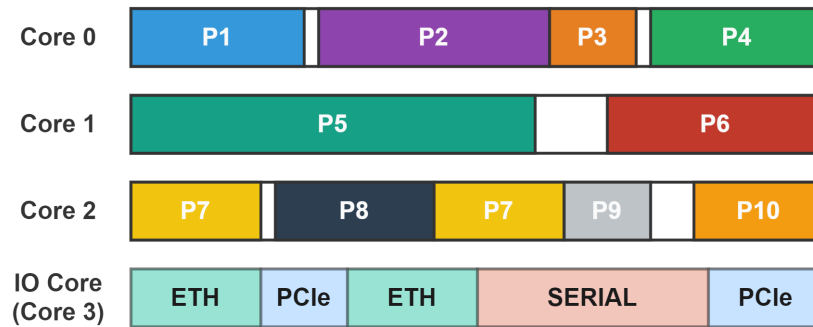


Figure 4.1 Multi-core MAF with dedicated IO core to manage IO transactions and interrupts.

4.2.2 Interferences

In this section, we present the interferences that we consider in our work. We also describe the interference channels studied and their impact on the system's execution.

Cache Interferences

Cache interferences in multi-core systems appear in two contexts: shared caches and private caches [6]. Shared caches interferences occur when multiple cores share the same cache. Two types of interferences might appear in this context:

- Contention interferences, when multiple cores try to access the cache at the same time, only one core is granted access and other core wait to access the bus.
- Eviction interferences, when an application on a core evict data owned by another application on another core. This enforces the owner application to reload its data, which introduces delays in execution time.

The second cause of cache interferences is the cache coherence feature [12]. Cache coherence protocols ensure that a data modified in the private cache of a core is updated in the private cache of other cores if it is present. Depending on the protocol, the data can be updated (which introduces a small overhead) or invalidated in the private cache. When the core accesses the invalidated data, it has to reload it. This introduces delays and goes against the robust partitioning principles. In this paper, we focus on shared cache interferences.

Bus Interferences

Bus or interconnect interferences occur when multiple components (cores, coprocessors, etc.) try to access the bus at the same time [39]. This contention results from the arbitration of the controller on the bus. Only one component may access and process transaction on the bus at a given time. Thus, the system puts other components that request the access on hold and wait for their turn to use the bus. Although multiple arbitration policies have been proposed to reduce or remove bus interferences, they are rarely available on COTS hardware.

Memory Interferences

Main memory interferences can be observed at different levels. Memory can be accessed through different independent channels without showing any interference [68]. However,

if two core access the memory through the same channel, the contention on that channel generates interferences.

The second type of interferences appears at the bank level. As depicted in Figure 4.2, the DRAM architecture comprises multiple banks [66]. Each bank comprises a row buffer, which stores the content of the last accessed row in the DRAM. The row buffer acts as a cache that allows accessing the data contained in the same row faster. When multiple core access the same memory banks, the row buffer unpredictably changes its content to accommodate cores access patterns. Thus, affecting the execution time of partitions executing on the different cores.

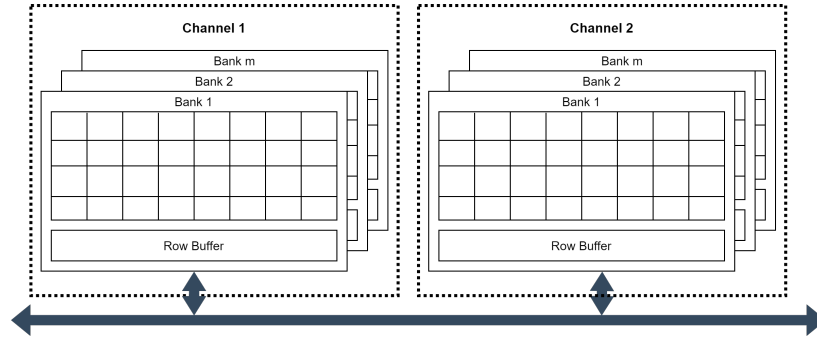


Figure 4.2 DRAM banked architecture. We represent banks as matrices for which, the row buffer, source of the bank interferences, stores the last accessed data.

4.2.3 Interferences Mitigation Means

Different mechanisms are present in COTS hardware to isolate components and make the platform more predictable. Table 4.1 provides a summary of the mitigation methods we discuss in this article. We also provide the availability of the solution, where it is said to be low when additional hardware is required and such type of hardware is usually not present in COTS processors. A high availability means that the hardware required to apply such technique is usually present in COTS processors (e.g., Performance Monitoring Counter (PMC), Memory Management Unit (MMU), etc.).

Cache way partitioning relies on hardware facilities to allocate specific ways of the cache to designated cores. Hardware specific registers must be set to allow segregation of the cache. This method does not introduce any kind of overhead but is not available on all architecture. In Section 4.1 we provide an overview of the availability of cache partitioning methods.

Cache set partitioning, also known as cache coloring, relies on the structure of cache memories and the physical address of the data to know where the data will be placed in caches. Figure 4.4 shows the physical address layout that allows to know in which set the data will be loaded.

Based on this information, the RTOS can wisely choose the virtual to physical translation to place pages in selected sets of the cache.

Finally, cache line partitioning is a combination of ways and set partitioning, where the RTOS allocate a set of cache ways and cache sets to a partition.

Memory banks and channels partitioning work the same way as cache set partitioning. By wisely choosing the physical address of a data, the RTOS is able to know in which DRAM bank and channel the data are stored. Placing data of different partitions into different banks will remove bank interferences (row buffer interferences). Channel partitioning allows assigning unique channels to cores. When a core accesses the DRAM, it can do so without contention on its channel due to another core requesting data.

Bus budgeting relies on the monitoring of memory accesses done by a core. When the number of accesses made during a given period of time exceeds a certain amount, the core stops its execution by scheduling an idle task or halting the core. This method has the effect to stop the contending core and release the bus bandwidth it uses.

All the previously mentioned methods provide the same amount of isolation and can be used for safety-critical systems. The choice of a technique over another should be based on the availability of the hardware and the overhead the technique introduces.

4.3 Related Work

In this section we present the current state of the art and position our work with regard to the interference channels we study in this article.

4.3.1 Cache Interferences

In [12], the authors present a survey of the different techniques for cache partitioning. Cache way partitioning is shown as the most used implementation. While line partitioning offers finer granularity, it also brings a higher execution and development overhead. Finally, set

Table 4.1 List of interference mitigation means available in COTS hardware

Mitigation method	Target hardware	Related interference	Availability
Cache way partitioning	Shared caches	Shared cache eviction	Low
Cache set partitioning	Shared caches	Shared cache eviction	High
Cache line partitioning	Shared caches	Shared cache eviction	Low
Memory bank partitioning	Shared DRAM	DRAM Row buffer	High
Memory channel partitioning	Shared DRAM	DRAM channel	Low
Bus budgeting	Shared bus	Bus contention	High

partitioning is known to be more complex to implement but offer better portability.

In [16], the UCP (Utility-Based Cache Partitioning) approach was introduced. It defines cache partitioning and monitoring hardware module to update the cache configuration at run-time. In the avionic domain, all configurations should be static and validated prior to system deployment as certification requires it [13]. Thus, we cannot apply dynamic approaches in this context. In [18] and [17], semi-partitioned caches are explored to improve performance while keeping the system in a more predictable state. The work of [20] extends this concept by clustering applications and partitioning caches on a per-cluster basis. In [26], the authors present a scheduling technique to account of the cache usage and reduce the performance hit introduced by cache partitioning.

In [22] and [23] cache partitioning is proposed alongside prefetching techniques. In our context, the system design often disables performance improvement facilities such as prefetching or branch prediction to reduce non-determinism produced by them. In [24], the authors use different processor management methods to reduce energy consumption. Literature also studied super pages with cache partitioning but the existing solutions use additional hardware [83].

Multiple works use virtualization in this context. In [27] and [45], hypervisors and the notion of virtual CPUs are proposed to ease cache partitioning implementation. The use of an hypervisor allows the application of cache partitioning to existing OS without modifying their code. However, it yields to an increased overhead in memory size and execution time.

4.3.2 Memory Interferences

In [61] the authors give a comprehensive list of memory interferences in COTS platform. The authors propose a methodology to analyze and understand the impact of the different mechanism in DRAM on predictability and latency. In [62] the authors propose a multi-policy resource allocation method. They use DRAM and cache partitioning together to leverage interferences in the system. Their approach relies on the analysis of a huge data set (2000 workloads) of execution (over 10000 experiments).

In [41, 57, 64] scheduling is explored to reduce interferences in the memory hierarchy. This approach allows reducing contention on the shared resources and thus, increase the throughput of tasks naturally. However, in [41], only one application is able to execute on multiple cores. Even-though scheduling reduces interferences, it does not ensure isolation between the partitions and makes certification more difficult in systems where robust partitioning is required [7].

In [65] the authors propose a software memory coloring approach to separate applications' memory between DRAM banks and channels. The method relies on a dynamic approach that changes the number of allocated banks and channels to threads at run-time. However, we cannot apply this method to hard real-time systems as the dynamic behavior would introduce non-determinism and make WCET estimation more complex due to the added factors to consider.

In [66], the authors coordinate the use of cache coloring and bank coloring. The duality of the approach and the conflict between cache and bank coloring is explained. In this paper, we review the approach proposed in [66] to allow faster integration and certification in safety-critical systems.

In [67] the authors propose a bank partitioning method to improve performance while taking into account the profile of the applications. The method relies on application profiling and clustering to categories their memory usage. However, we cannot use additional hardware to gather the metrics needed by the online algorithm. Similarly, in [84] the authors propose to isolate concurrent threads to use different memory banks for data sampling applications. Performances are further improved by balancing the load between the different memory banks.

4.3.3 Bus Interferences

Bus bandwidth management is a widely studied approach to leverage bus interferences. In [39] the authors present an approach called MemGuard. MemGuard regulates bus bandwidth and proposes different mechanisms to increase performances. Each core has an allowed amount of access to the bus during a quantum of time. If the core exceeds its budget, the RTOS puts it on hold until the next quantum of time. MemGuard uses per-core budget allocation and lacks configuration flexibility. These two main limitations that are further explained and addressed in Section 4.2.

In [41,47] and [42], scheduling frameworks to leverage predictability in multi-core systems are proposed. The approaches use memory bandwidth throttling mechanisms to ensure bus and memory interferences bounding. In [43], the authors propose to isolate bus resources when hard real-time applications execute, thus, giving them exclusive access to the bus. In [49], critical applications are mapped to a single core, leaving the other cores to use by best effort applications.

In [45], the authors design a resource management framework based on virtualization to leverage cache partitioning and memory bandwidth limitation. The approach is based on the

use of virtual CPU (vCPU).

Execution models were introduced to manage bus and memory interferences. In [50] and [52] the author use the Acquisition Execution Restitution model (AER). The objective of this method is to separate the computation (Execution phase) from memory accesses (Acquisition and Restitution) during run-time and schedule the memory phases so only one partition can be in those phases at a time. Thus, the RTOS schedules access phases to ensure no contention occurs.

Other works rely on hardware mechanism to reduce, bound and remove interferences. In [56–58] and [55] memory controllers and arbiters are proposed to improve predictability while increasing the performance. However, these approaches are not compatible with COTS hardware as it requires additional hardware that is not certifiable in some cases. In [59] the authors study three techniques to manage memory bandwidth: thread packing, clock modulation and Intel MBA technology.

We cannot apply the exclusive use of resources at a given time such as the method proposed in [43]. We consider all the partitions in the system as highly critical. Applying this method would make the use of multi-core useless as only one application could execute at a time. The approach provided in [47] reduces the overhead and improves the performance but we cannot afford to change the criticality of the applications (considered as of the highest criticality) nor provide different execution modes. Unfortunately, we cannot rely on the additional hardware components used in [44]. Moreover, contrary to the proposed method, isolation must be maintained at any moment in the execution window. Finally, our objective is to reduce applications WCET contrary to optimizing the average performance. Work such as [59] does not provide strict isolation of resources and thus, makes certification more tedious. In [82] the Single-Core Equivalence principle was proposed. We base our work on this approach and refine it to enable easier integration with non-virtualized environments. While virtualization allows a better management of resources, this method is not applicable in our context. We aim to rely on a bare-metal RTOS that does not allow virtualizing the system’s resources. We also aim to reduce the overhead introduced by these techniques by implementing our own memory manager.

Using the AER model is not applicable in our context as we need to keep compatibility with legacy applications that do not make use of such model.

In [48], the authors show that per application budgeting increases the system’s performance and scheduling feasibility. In this paper, we provide an extension to MemGuard that allows per application budgeting instead of the per core budgeting previously allowed by MemGuard.

In this paper, we choose to use set based cache partitioning alongside with DRAM bank partitioning. Both approaches are portable to any architecture with virtual to physical address translation and are applicable to COTS hardware. For the same reason, we choose to extend the MemGuard approach that relies on COTS available features such as PMC (Performance Monitoring Counters) [39]. In order to enable portability, our solution can be extended for cache way partitioning use.

In the next section, we present our approach to concurrently mitigate shared cache, bus and memory interferences.

4.4 Global Memory Manager

In this section, we present our centralized memory manager. This module integrates the state-of-the-art techniques in terms of memory interferences mitigation and extends the existing approaches to ease their certification in the context of aerospace systems. Figure 4.3 provides an overview of the system’s architecture with key points where our method applies. While the memory allocator manages how shared caches and DRAM is allocated across partitions, the bus budgeting module monitors and controls bus usage by the partitions. Finally, the safety net module is scattered across the memory hierarchy, to monitor the system’s health. In case of an error, the safety net triggers an error to the heal monitoring manager. Throughout this section, we explain in detail each module and how they interact with each other.

4.4.1 Memory Management

As presented earlier, three main approaches exist to partition the cache. Way partitioning allows segregating caches in ways, by allocating them across cores. Set partitioning, also named cache coloring, profits from the virtual to physical translation mechanism to arrange

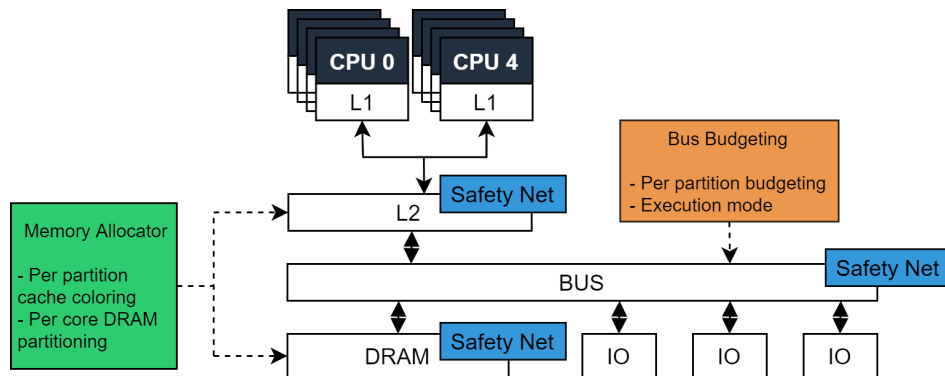


Figure 4.3 Memory hierarchy with key points where our management framework applies.

the memory layout and allocate cache sets to different applications. Finally, line partitioning allows allocating cache lines to applications. Line partitioning is available when both set and way partitioning are possible. A study on architectures' reference manuals provided the following insight on 40 different architectures (ARM, x86 and PowerPC):

- Out of 40 architectures, 14 support way partitioning and 34 support set partitioning. 3 do not support any form of cache partitioning and 11 support both set and way partitioning.
- Set partitioning is more complex to implement while way partitioning can seamlessly be integrated in existing memory managers.
- When comparing the number of available cache partitions, in 72% of the cases, set partitioning allows twice as much partitions than what way partitioning allows.
- Way partitioning only allows for core allocation in 80% of the architectures.
- Set partitioning offer a better portability across the architecture as way partitioning implementation is architecture dependent.
- Partitioning on an application basis allows avoiding flushing the cache at each major time frame [10], which drastically reduces the context-switch time and improve cache usage for applications.

In conclusion, set partitioning is privileged over way partitioning as it allows more cache partitions and is easily portable to different architectures. It is worth to be mentioned that our approach can be extended to use way or line cache partitioning with few modifications.

As discussed in [66], cache coloring (set partitioning) may interfere with memory coloring. Figure 4.4 shows the different memory layouts encountered during our study on architectures. [66] also discuss the bank bit randomization technique used in modern processors. Such process is not present in all architecture but should be taken into account by the system designer when studying the memory address layout of the system. Our allocator takes into account overlapping bits in memory coloring to ensure complete isolation of the resources.

We propose a two-level memory allocator. First the allocator selects a memory bank based on the core the application is running on. In our design, we allocate banks on a per-core basis while allocating cache partition on per-application basis. Using private cache partitions for each applicative partition allows the systems to skip cache flush and invalidate when switching between partitions as recommended by the CAST-20 [10]. This method allows us to use cache more efficiently while reducing the partition switch time.

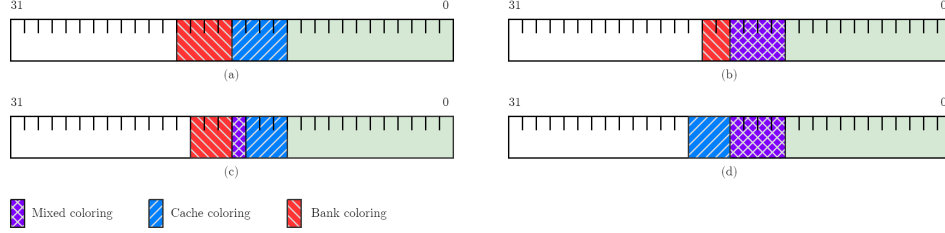


Figure 4.4 Memory coloring layouts found in 40 different architectures. (a) allows separate allocation of banks and cache partitioning, (c) show overlapping bits while (b) and (d) show that bank coloring completely overlaps cache coloring and vice versa.

We define an allocation unit by the couple $(B; C)$ where B is the bank identifier and C the cache color. An application can have one or more assigned $(B; C)$ couples. Shared memory regions (between application on the same or different cores) use the $(B; C)$ couples of all applications sharing the region. Figure 4.5 depicts the execution flow of the allocation process. Our method ensures compatibility with all layouts depicted in Figure 4.4. It is also extensible to other future layouts. The safety net is explained in the next section of this paper.

When applications' private and shared memory have multiple allowed couples, the allocator select the $(B; C)$ couple to evenly distribute the workload on all available memory.

Once the algorithm selected the memory bank, the allocator provides one or more cache colors to the application. Each memory bank contains at least one cache color. We compute the number of cache colors per bank by subtracting the number of overlapping bits between the cache color bits and the bank color bits to the number of cache color bits.

To provide a faster allocation, we do not rely on linked list to represent free memory. We represent the memory banks with the following structure:

```
struct bank_color {
    uint8_t  bank_id;
    uint32_t free_mem;
    ptr_t    cache_color_head[nb_cc];
}
```

In this structure, the *cache_color_head* array represents the next free address in memory for a given cache color. We also use *nb_cc* to represent the number of cache color associated with the current bank. Using this structure the allocation process is straightforward. When allocating a page to an application, the allocator sets the cache color head referred by $(B; C)$ to the next free page in $(B; C)$ using Equation 4.1. In this equation $\ll$ is the shift left operator and *firstBankBit* is the first bit of the bank coloring area in the address layout.

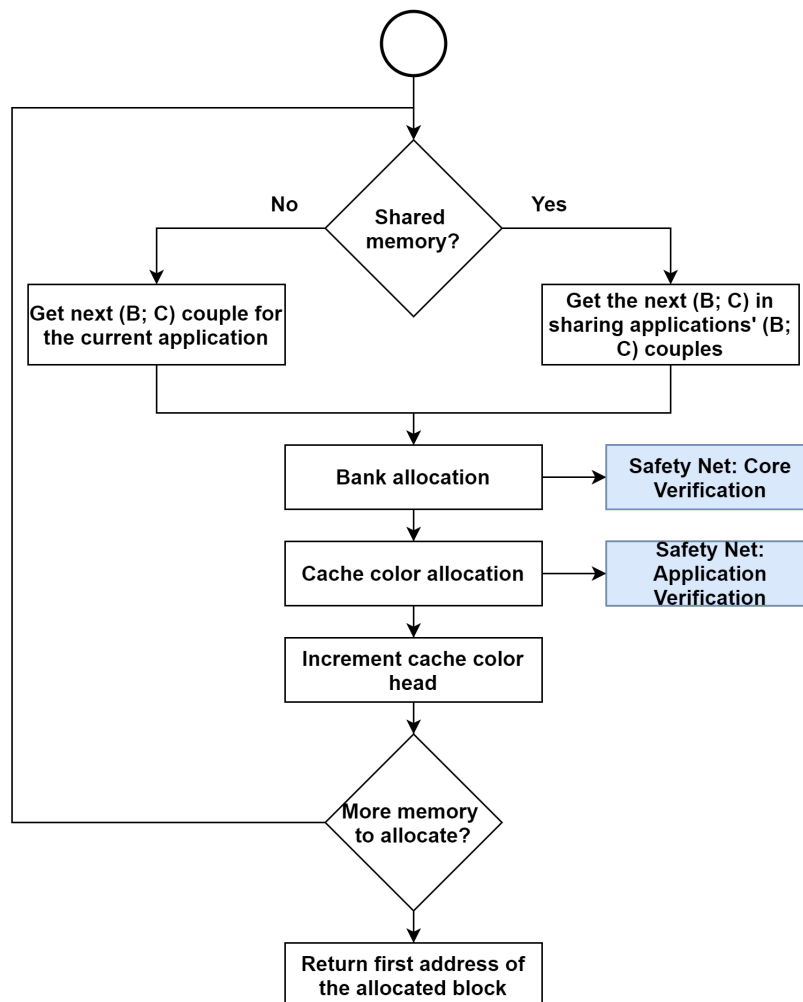


Figure 4.5 Two-level memory allocator flow. We use the same process for both private and shared memory.

$$head = head + (bankId \ll firstBankBit) \quad (4.1)$$

The memory allocator is developed to completely replace the original memory allocator of the RTOS. In this article, the allocator is written from scratch inside the RTOS [79] and takes the place of the page allocator, in charge of allocating an arbitrary number of pages when requested (in this context, at boot time). Due to its low algorithmic complexity and the use of well-known structures, the allocator can be easily analyzed, making this approach easy to certify for critical systems.

4.4.2 Bus bandwidth budgeting

Bandwidth budgeting as proposed in [39] suffered from two drawbacks in our context. First, existing approaches set budgets on a per-core basis, with the inability to change cores' budget outside of synchronization points. With this scheme, it is impossible to change the budget of a core without impacting other cores' budget, which would entail interferences and impact on performance. Second, it is possible to improve the reclaiming principle proposed to better suit critical systems by providing more flexibility and control on how applications control their budget. Reclaiming budget means that an application is capable of requesting more budget than it was allocated by the means of a shared budget pool.

We propose to extend the MemGuard approach by adding the following features: (1) Per application budget allocation: each application receives a static amount of budget for the duration of its time window in the MAF; (2) An initial reclaiming pool budget to accommodate with the scenario where all applications are critical; (3) Introduction of four application modes that we present later in this section;

The ability to define the budgets on a per-application basis allows finer control and permits to adapt to the different applications running on the same core. We allow per application budgeting by the modification of the application scheduler. At application switch, the RTOS performs a reconfiguration of the performance monitoring facility to account for the newly scheduled application's budget. Another advantage here is to monitor that the bus budgeting resource allocation is respected.

Reclaiming modes

The computation of the initial reclaiming pool size as well as the budget allocation are out of the scope of this article and can be computed using the method proposed in [48].

To accommodate with the critical nature of ARINC-653 systems and render our approach more flexible and compliant with [7], we propose to classify applications in four categories to extend the reclaiming feature:

- **Full reclaiming** mode, which allows an application to reclaim budget and provide budget to the reclaiming pool. Soft real-time partitions can use this mode. We also propose to further constrain the budget prediction to provide bounds on the budget removal of the application.
- **Greedy** mode for which an application does not provide any budget to the global budget pool but is able to reclaim budget from this same pool. This is suitable for any hard and soft real-time partitions, however, hard real-time applications are more prone to use this mode.
- **Altruist** mode, which allows an application to provide budget to the global pool but not reclaim budget from this pool. This mode is suitable for best-effort applications which can run in degraded mode.
- **Strict** mode, which forbids budget reclaiming and providing budget to the global pool.

Figure 4.6 (a) depicts the execution of a partition under full reclaiming mode, which means that it can reclaim budget from the global pool and provide budget to it. At time ① the partition starts executing. At the end of the budgeting periods ② and ③, it was predicted that the partition will use less budget than initially provided. Budget is removed from the partition to provide it to the global pool. At ④ the partition exceeded its budget and reclaimed a fixed amount of free budget from the global pool. At ⑤, the manager does not remove budget from the partition. Finally, at ⑥ the partition exceeds its budget but the global Tpool is empty, the partition is removed from execution until the next timer's period.

Figure 4.6 (b) shows the execution of a partition under greedy mode. This time the partition's budget is exceeded at ②, ③ and ④ and is replenished at ② and ④ by reclaiming budget from the global pool and at ① and ⑤ from the bus budgeting replenish server. In greedy mode, no prediction is made and the partition does not provide any budget to the global pool.

Similarly Figure 4.6 (c) shows the execution of a partition under altruist mode. The budget can only be replenished by the bus budgeting server (at times ①, ②, ④ and ⑥). The partition can provide budget to the global pool and gets removed from execution when it exceeds its budget at times ③ and ⑤.

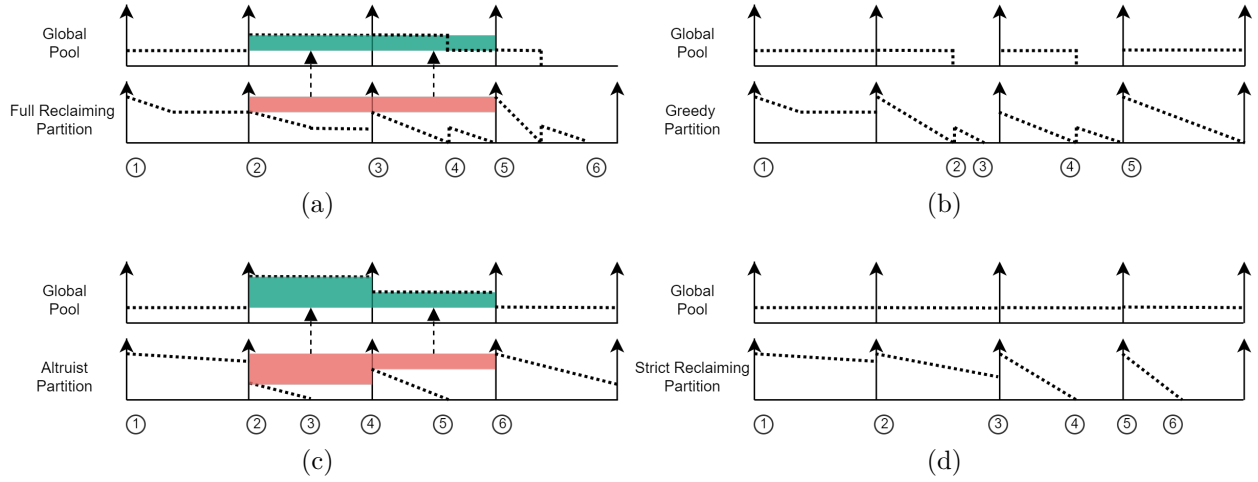


Figure 4.6 Partitions' scheduling behavior when using full reclaiming mode (a), greedy mode (b), altruist mode (c) and strict reclaiming mode (d).

Finally, Figure 4.6 (d) depicts the execution for a partition using strict mode. Here, the partition does not give any budget to the global pool but also cannot reclaim any. The only replenish events occur at ①, ②, ③ and ⑤ and the partition is removed from execution at times ④ and ⑥.

Algorithm 1 Periodic server timer handler function.

```

1: function TIMERHANDLER
2:   App = ScheduledApp()
3:   nextBud = App.initBud
4:   if App is (full reclaiming or altruist) then
5:     nextBud = PredictNextBudget()
6:     AddSlackBudgetToPool(App.initBud - nextBud)
7:   end if
8:   if HealthCheck() is Failure then
9:     RaiseHMEError()
10:    Return
11:  end if
12:  UpdatePMC(nextBud)
13:  UnlockCore()
14: end function

```

The proposed reclaiming modes allow a complete flexibility on the system bus management. The system can change the mode at any moment and for any partition during the execution of the RTOS. Changing the reclaiming more is done through an API proposed by the RTOS. This API allows reconfiguring the budgeting mode while ensuring highly critical partitions are still correctly isolated. The API can be called at any moment by the partition during its execution.

Algorithm 1 shows the execution flow of the periodic server interrupt. This process is executed

before restoring the application’s context. Algorithm 2 depicts the process executed when an application exceeds its allocated budget. As proposed in MemGuard, when all cores become idle because of budget exceeding, the mechanism will reschedule all applications. Finally, we explain the *HealthCheck* block in the next section and refers to the safety net feature proposed by the CAST-32A document.

Algorithm 2 Interrupt handler executed when an application exceeds its budget.

```

1: function PMCHANDLER
2:   App = ScheduledApp()
3:   if HealthCheck() is Failure then
4:     RaiseHMEError()
5:     Return
6:   end if
7:   if App is (full reclaiming or greedy) then
8:     nextBud = ReclaimBudget()
9:     UpdatePMC(nextBud)
10:    Schedule(App)
11:  end if
12:  if AllCoresIdle() then
13:    ForceResetTimer()
14:  else
15:    ScheduleIdle()
16:  end if
17: end function

```

In our context, we consider systems with hard real-time applications only. We cannot afford removing budget from a partition using *s*. Thus, we introduce an initial reclaiming pool containing free budget for any application. The global pool contains budget that the partitions can reclaim when their budget is exceeded. This global pool is replenished at each budget server’s tick. This approach allows improving the overall system’s performance as presented in our result section. The benefits of this method are presented in the results section.

Both the periodic replenish server and the PMC interrupt handler are integrated from scratch in the RTOS. We have chosen to implement the different modules from scratch and not use a patched version of MemGuard because the architecture of an ARINC-653 compliant RTOS greatly differs from the Linux kernel. The periodic replenish servers consists in an interrupt service routine called every time the global timer triggers its periodic interrupt. In the same manner, when the PMC detect that the budget is exceeded, an interrupt handler is called and an idle task is scheduled to replace the running application as presented in Algorithm 2.

4.5 Safety Net

To ease the certification process, we define a set of rules that should be verified at all time during the system’s execution. Apart from the CAST-32A guideline, we provide a way to

ensure synchronization between cores with regard to the MAF timings.

4.5.1 Core Verification

The RTOS architecture we use offers an Asymmetric Multi-Processing paradigm to manage the CPU. In this architecture, each CPU executes a separate instance of the RTOS. Small shared memory region enables the inter-core communications.

Each CPU has its own timer and uses a tick-less scheduler. This means no periodic tick is present in the system and timers interrupt occur only when needed (for instance, when the next application should be executed). To ensure synchronization, we add constraints to the MAF length. All CPUs can have different MAF length; however, we define a global hyper period based on the least common multiple Q between all CPUs MAF.

We define q_i the length of the MAF M_i defined for CPU i . M_i will be executed $\frac{Q}{q_i}$ times before reaching a synchronization point. When reaching such point, the CPU waits for the synchronization barrier release. When all CPUs reach the barrier, the kernel releases the CPU and immediately start a new MAF. This process corrects the drift between CPU clocks while ensuring real-time constraints are met on all cores. Our results show that the overhead introduced by the synchronization mechanism does not break any real-time constraint nor introduces deadline miss in the system. We provide the overhead considerations and experiments in Section 6.3.

Q may be further constrained to reduce the drift between CPUs clocks. The smaller Q is, the faster the synchronization module corrects the time drift.

4.5.2 Runtime Verification

To ensure safety of the proposed memory manager, we define a set of constraints that are verified at critical points during execution. Table 4.2 defines the set of variables we use to formalize the system.

We further define the following rules:

Rule 1: At any time during the system's execution, co-running applications must not share memory or cache color.

Rule 2: Shared memory regions color set must only contain colors allocated to the applications that share this region.

Rule 1 and rule 2 are verified during the system startup when allocating memory. It is further verified during a page fault, when the RTOS handles the page fault, it ensures that

Table 4.2 List of variable associated with safety net constraints

Q	System's hyperperiod, defined as the least common multiple between all CPUs MAF
q_i	The length of the MAF M_i defined for CPU i .
A_i	Application i
$\mathcal{C}_{A_i}$	The memory colors set allocated to the application i .
A_i^C	The CPU identifier where A_i is mapped.
A_i^B	The bus bandwidth budget allocated to A_i .
A_i^M	The maximum number of bus access done by A_i during a single MAF.
$Acc(A_i, t)$	Accessed number done by an A_i at time t .
$\mathcal{S}_i^C$	The memory colors set allocated to the shared memory region i .
$\mathcal{S}_i^A$	Applications set sharing the memory region i .
$G(t)$	The free budget in the global pool at t .
$R(A_i, t)$	1 when A_i executes at time t , 0 otherwise.
B_{max}	The maximal bus bandwidth.

Table 4.3 Formalized safety net rules

Rule 1	$A_i^C \neq A_j^C \Rightarrow \mathcal{C}_{A_i} \cap \mathcal{C}_{A_j} = \emptyset$
Rule 2	$\mathcal{S}_i^C \subseteq \bigcup_{A_j \in \mathcal{S}_i^A} \mathcal{C}_{A_j}$
Rule 3	$\forall t \in [0; Q], G(t) + \sum_{A_i} A_i^B \times R(A_i, t) \leq B_{max}$
Rule 4	$\forall A_i, \sum_{t=1}^{q_{A_i^C}} Acc(A_i, t) \leq A_i^M$

the address generating the fault is mapped to a physical address in the application’s colors set. This rule does not apply to explicitly shared memory.

Rule 3: At any time t during execution, the sum of the bus bandwidth budget of the executing applications and the global budget pool must not exceed the maximal bandwidth permitted by the bus.

Rule 4: For safety purpose and fault detection, we define a maximum number of bus access for each application and IO in the system. The system should never reach this number to ensure the system’s stability.

To verify rule 3, we add the *HealthCheck* function in Algorithm 1. Every time the system changes the budget configuration, the health monitor ensures that rule 3 is satisfied. If not, the health monitor handles the error. Rule 4 is verified in Algorithm 1 when handling applications budget overflow.

B_{max} can be provided by the processor’s manufacturer or can be empirically measured with benchmarks. In this paper, we rely on the work presented in [85] to compute the value of B_{max} for our architecture.

4.6 Results

In this section we present the study of our global approach and compare it to the current literature.

Our test bench comprises an NXP P2020RDB-PC board with two PowerPC e500v2 cores running at 1.2 GHz and an NXP T2080RDB-PC board with four PowerPC e6500 cores running at 1.8GHz. We rely on a proprietary ARINC-653 RTOS (M-RTOS [79]) to retrieve our experiments results. We study the system using the TACLeBench [81] benchmark suite. We do not show results for the safety net other than its overhead. This is because the safety net is a part of the environment that is evaluated. All measures and results presented here are conducted with the safety net enabled.

We compare our approach to existing work in terms of overhead only because those methods do not differ in terms of added predictability. Reducing the execution time of interference mitigation method is important because the systems in which they are implemented are usually very constrained in terms of time and resources. As a result, efforts must be done to allocate as much time as possible to running applications by reducing the time the RTOS and its services run.

4.6.1 Memory Allocation Overhead

We compare the execution time of our memory allocator for different allocation sizes to the one proposed in [66]. Our allocator initialization time is 2 us compared to 7120 us for the one proposed in [66]. This is because we do not have to create a linked list of available pages at initialization. Furthermore, we do not have to maintain this list during allocation. Figure 4.7 shows the allocation time reduction that benefits our method.

Memory allocation only occurs at the system’s startup, which implies that the allocator’s overhead only impact the system’s performance during boot time. In our test environment, memory allocation represents 9.39% (2.04ms) of the total startup time (21.74ms), which justifies the need to reduce its impact as much as possible. Our allocator reduces the startup time of the RTOS we use in our experiments by 24.8% (21.74ms) compared to the method proposed in [66] (28.91ms). After allocation, the cache partitioning technique has no overhead. Indeed, it relies on the MMU to perform a controlled translation of addresses. This process also occurs when not partitioning the cache.

4.6.2 Bandwidth Limitation Overhead

Our second experiment evaluates the overhead introduced by the bandwidth limitation module. We compare our implementation that adds reclaiming modes with the overhead introduced by MemGuard. Figure 4.8 present the overhead of our bus bandwidth management technique.

We based our test environment in the same context as what is proposed in [39]. To measure the overhead introduced by bandwidth limitation mechanisms, we use the following settings:

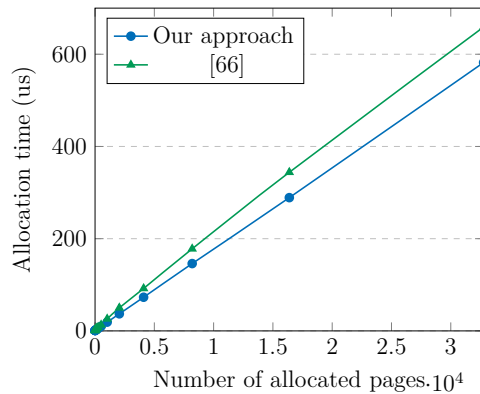


Figure 4.7 Allocation time for different allocation sizes. The results show that our allocator allows faster allocation time.

- For server’s timer overhead, we disable bandwidth limitation mechanism by disabling PMCs interrupts. The rest of the mechanism is enabled. Figure 4.8 (a) shows our results.
- For server’s timer overhead and budget exceeding mechanism, we enable bandwidth limitation mechanism by enabling PMCs interrupts. We also make sure that the application only exceeds its budget once every server’s period expires. The RTOS directly reschedule the application once its budget is exceeded to measure only the limitation mechanism’s overhead. Figure 4.8 (b) shows our results.

We compare our approach when using a predictor to add budget to the global budget pool and when not using any predictor. Results show that our extension of Memguard to support per applications budgeting as well as reclaiming modes adds a small overhead (less than 0.5%). Our experiments show that the bus usage predictors introduce the most overhead.

Finally, we studied the impact that the mode change API has on the execution time of partitions. The mode change was implemented to only impact the calling core. Thus, this API call is no different than any system call made by the user to the RTOS (e.g mutex acquisition, display print, etc.). The API performs a constant number of accesses to the memory. Based on the measurements of the execution time of common API and system calls in the used RTOS, changing mode is 4.6 times faster than a semaphore signaling, 8.5 times faster than getting the current partition’s status and takes sensibly the same time as retrieving the current process’s identifier.

4.6.3 Safety Net Overhead

We further study the impact of the safety net on the performance. We provide two metrics to express the overhead. We define the absolute overhead, which gives the absolute execution time added by the safety net. We also define the smoothed overhead, which describes the impact of the safety net on the applications execution time. It is important to note that the safety net overhead should be reduced as much as possible at its routines frequently executed in the system (e.g., TLB miss handlers, context switch, etc.).

Figure 4.9 presents the absolute and smoothed overhead of the safety net. *BL* stands for “Bandwidth Limitation”. While we manage to maintain the overhead under 1% for most safety nets, the allocation safety net (Rule 1 and Rule 2) overhead reaches 4.56%. However, this overhead only occurs during the system startup and slows down the boot process by 1.5%. This is why we do not provide any allocation smoothed overhead metric, as it is nonexistent.

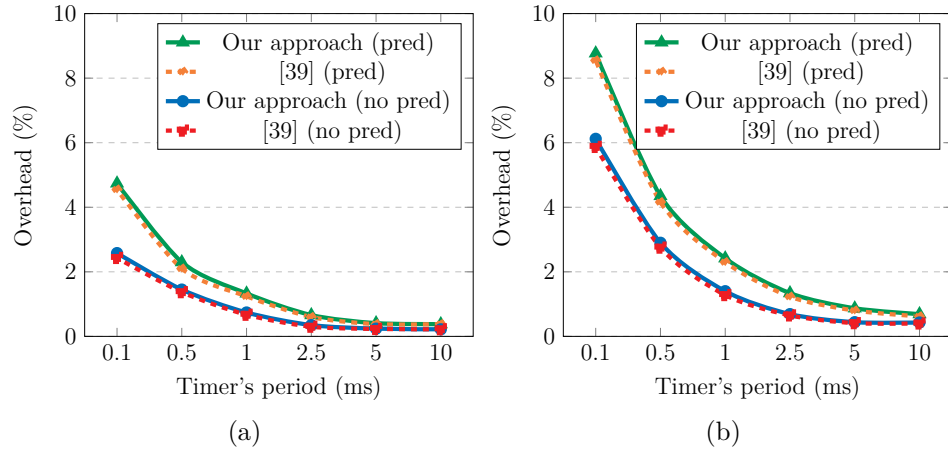


Figure 4.8 Bandwidth limitation mechanism overhead depends on the timer's period.

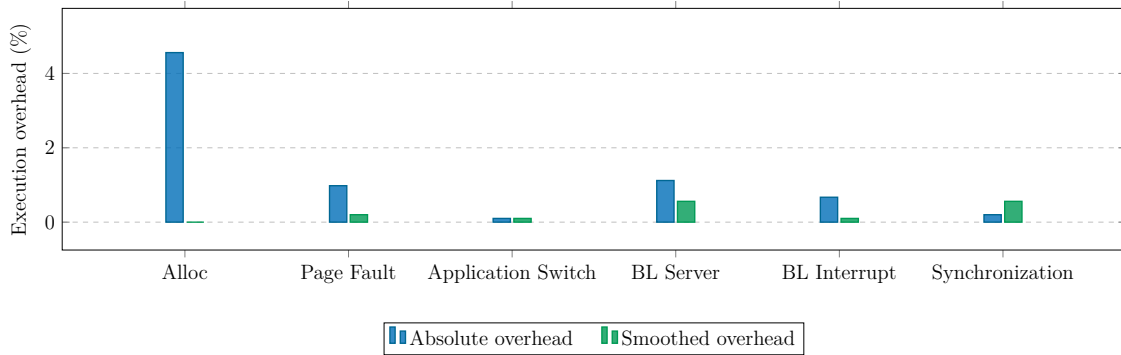


Figure 4.9 Absolute overhead of the safety net compared to the smoothed overhead introduced at run-time.

The time synchronization overhead presented in Figure 4.9 refers to the overhead added by the core MAF synchronization mechanism. Two factors cause this overhead. First, the synchronization routine itself must access shared data atomically to know the state of the synchronization barrier. The second factor comes from the potential synchronization of the cores. The more a core drifts, the more it is prone to reach the synchronization barrier late. In this test, the synchronization primitive causes the absolute overhead while the smoothed overhead accounts for the total overhead (synchronization primitive and drift of the cores clocks). The second factor is the more impacting, this is why the smoothed overhead is bigger. The results were gathered by measuring the number of cycles the cores stay in the synchronization routine at every MAF renewal and compared to the number of cycles the complete MAF renewal routine takes. The synchronization happens at most once every MAF start. Current timing analysis methods can take this overhead into account in the MAF renewal time. The overhead added by the synchronization was measured to be 0.56% of the MAF renewal time in the worst case. To gather the results, we measured 100,000 MAF renewal.

4.6.4 Scalability of the solution

The partitions-related safety net as well as the bus bandwidth limitation mechanism are designed to be independent for each core and are based on an AMP RTOS architecture. The cache coloring run time mechanism solely relies on the MMU and, thus, is independent of each core that has a private MMU. By construction, those mechanisms are not impacted by the number of cores in the system.

The startup memory allocation mechanism is impacted by the number of cores in the system. Memory allocation must keep global structures to ensure correct memory allocation between cores, thus synchronization primitives are used to ensure exclusive use of these structures. The worst case happens when all cores allocate memory at the same time. To access the shared data structures, a core might wait until all other cores have finished their allocation. Thus, the allocation time can be multiplied linearly by the number of cores in the system. To validate our assumption, we artificially created such a case by adding a barrier before each core allocates the partitions' memory. Each core has the same amount of memory to allocate. Our experimental measures show a maximal allocation time of $233\mu\text{s}$ on 1 core, $429\mu\text{s}$ on 2 cores, and $849\mu\text{s}$ on 4 cores¹.

Finally, the core synchronization mechanism itself is not impacted by the number of cores. However, the time waited for all cores to reach the barrier may differ and is likely to increase

¹The results were gathered on the NXP T2080 platform.

with the number of cores. The equation $\sum_{i=1}^N D_i$ provides a way to compute the worst synchronization time where N is the number of cores and D_i is characterized by the amount of time the clock of the core i drifts between two synchronizations.

4.6.5 Effect on Predictability

We validate our approach by studying the predictability of eight concurrent applications. We define the predictability of an application based on the standard deviation of the execution time. We compare the predictability with interference management to the predictability without interference management. A lower value means the partitions' execution time is more predictable.

To approximate the measured WCET², we analyze each application under the same input data for all executions. We further executed each application multiple times until no higher execution time is detected after 1000 executions. In a safety-critical context, a more formal approach should be used, however, timing analysis in presence of interference in multi-core system is not yet attainable and provides results that are too pessimistic [52].

This method provides a way to analyze the impact our work has on the applications' actual WCET. The WCET increase entailed by the interference mitigation mechanism is compared to the applications' WCET measured with a single-core setup, where only one application exclusively uses the system's resources. The lower the WCET increase, the better our method performs.

Finally, the bus budgets as well as the initial reclamation pool size are computed using the method proposed by [48] where an ILP formalism to the budget allocation that fits our experimental environment is defined.

Figure 4.10 (a) shows the relative increase of measured WCET compared to the WCET measured in single-core execution. Lower values are better and mean the application suffered less slow down because of the interference mitigation means. We used 8 applications from the TacleBench suite and co-run them on 2 cores. Interferences are naturally introduced by the memory usage of each application. Table 4.4 sums up our test payload. The period of each partition on the core 0 is 50 ms while periods for the core 1 are 75 ms. This ensures that with enough executions of the system, all partitions of the core 1 are co-run at least once with all partitions of the core 0. Our approach allows to reduce the impact of isolation on the

²We use a measured approximate WCET because current state of the art in multicore WCET analysis does not provide precise means of analysis in presence of interference. Our work is a step towards the use of conventional analysis methods, but other interferences exist and need to be addressed before being able to use regular analysis methods.

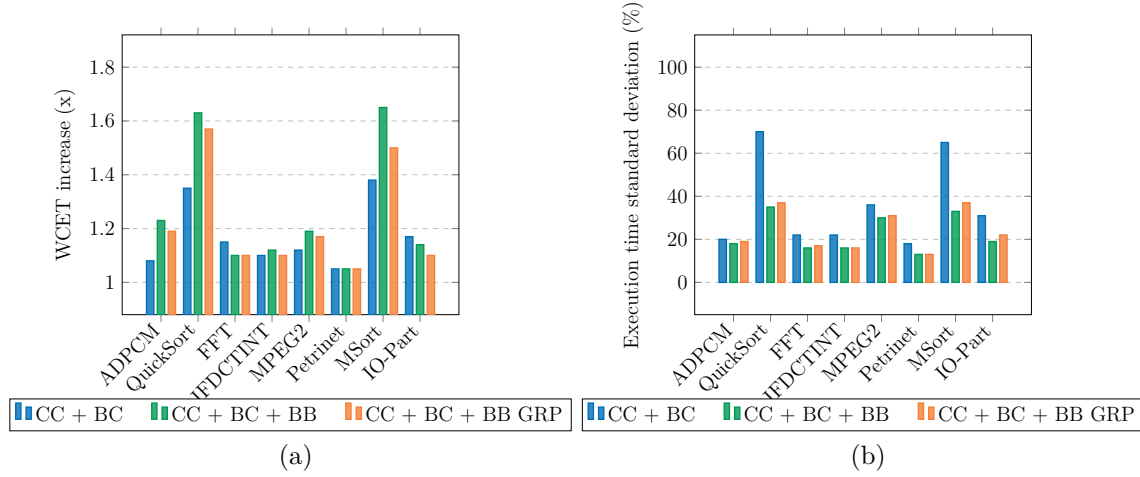


Figure 4.10 WCET (a) and predictability improvements (b) provided by the memory manager. We compare the proposed metrics when using cache and bank coloring (CC + BC), bus budgeting (CC + BC + BB) and the initial reclaiming pool (CC + BC + BB GRP)

Table 4.4 Test environment applications

Application	Description	Core
ADPCM	Analog to digital filter Memory intensity: medium	0
QuickSort	Array sorting algorithm Memory intensity: high	0
FFR	Fast Fourier Transform implementation Memory intensity: medium	0
JFDCTINT	Forward discrete cosine transform implementation Memory intensity: low	0
MEPG2	MPEG2 manipulation Memory intensity: medium	1
Petrinet	Petrinet simulation implementation Memory intensity: low	1
MSort	Array sorting algorithm Memory intensity: high	1
IO-Part	Application performing IOs on the serial port Memory intensity: high	1

WCET while drastically increasing the predictability. In Figure 4.10 (b), the predictability corresponds to the standard deviation of execution time where 100% is the standard deviation in multi-core processors with the same setup but without interference mitigation. We show that in our test bench, we reduce the standard deviation of execution times by 68.1% on average.

Isolation is ensured by design. When using cache coloring, shared caches are strictly segregated into different areas and partitions using strictly different cache colors cannot share any cache area. It is the same for memory banks. Bus budgeting will also detect when a budget is exceeded and stop the partition at the very instruction where the budget is exceeded. Thus isolation on cache and bus is ensured. However, in multicore systems, far more interference channels exist. For instance, cache coherence protocols will introduce delays when in need to update private caches [6]. Other hardware mechanisms such as the Miss Status Holding Registers (MSHR) will create interferences [86]. Thus, is it impossible to reach the same predictability in multi-core systems compared to single-core ones. Our objective is to remove the most critical one to lower the analysis pessimism and further conduct the system's analysis taking into account the remnant interferences.

It has to be noted that the increase in predictability as well as the increase of the WCET is not the same for all applications. Indeed, in the system, some applications are more prone to interferences (usually applications that use more memory). Thus, memory-intensive applications (in our experiments MSort, QuickSort) will see their execution time increased more than other applications because of the resource restriction cache coloring and bus budgeting apply to them. For the same reason, the increase of the WCET is not proportional to the decrease of the standard deviation of execution time.

We also show that for highly critical system with only hard real-time applications, the initial reclaiming pool allows improving the overall system performance. Based on experiments with the proposed benchmark, we use an initial global reclaiming pool containing 10% of the maximal bus budget. This configuration yields to the best performance by providing additional budget for hard real-time applications using full-reclaiming and greedy modes while avoiding budget waste. Our study shows that while using an initial global reclaiming pool slightly reduces predictability, it allows to reduce the WCET and reduce the impact of bus budgeting.

Finally, it is important to discuss the trade off between WCET increase and predictability. An increase of less than 2 on dual-core architecture means that the system will be able to host more partition than a single-core system. Similarly, an increase of less than 4 on a 4-core architecture would mean the same. The goal of our interference mitigation means is

to reduce the WCET impact as much as possible to ensure isolation but also allow more petitions to be hosted by the system. Based on our results, the method we propose increases applications' WCET by 22.3% on average on 2 cores, which is deemed acceptable to provide an increase of predictability of 68.1% on average. We also applied our approach on a 4-core setup, which yielded the same results, following the scalability analysis we provide in Section 6.5. For the sake of space, we do not present these results in this paper.

4.7 Conclusion

Safety-critical systems rely on partitioned system to ensure complete isolation between applications. However, with the advent of multi-core processors, such isolation is impossible when using regular resource management methods. The contention on resources shared between the different cores generates interferences. Interferences are non-deterministic delays in execution time that prevent certification of safety-critical systems. The delay interferences introduce at run-time cannot be predicted. Thus, when measuring the Worst-Case Execution Time of applications, one must assume the worst case when sharing resources with other applications. This yields to overly pessimistic results that render the system unusable.

Research in the domain proposes multiple solutions to isolate different components and improve predictability. However, such solutions are not always applicable to safety-critical systems, where the configuration and implementation of interference mitigation methods must be deterministic and known during system design to allow certification.

In this paper, we presented a statically configurable memory manager that isolates the memory hierarchy between applications. Our approach enables to mitigate shared cache, DRAM and bus interferences with deterministic and certifiable methods. We further provided safety net rules to accommodate with the CAST-32A guidance document. Finally, we studied the impact of our method in terms of performance improvement and overhead. Our approach increases predictability by 68.1% while reducing the impact of mitigation methods.

As explained in Section 3, other interferences exist. Our approach restricted interference mitigation to shared caches, memory bank and bus interferences. Future work could integrate the use of channel partitioning in our mitigation framework and study its impact.

CHAPTER 5 ARTICLE 2: INTERRUPT RELATED INTERFERENCE BOUNDING IN AEROSPACE MULTICORE SYSTEMS

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Serge Harnois, Felipe Gohring de
Magalhaes and Gabriela Nicolescu

Submitted in Springer Real-Time Systems, Submission date: 2023/07/11

Abstract: The transition to multicore architectures in safety-critical systems brings many challenges. The contention on shared components generates unpredictable timing hazards called interference. To certify safety-critical systems, interference must be mitigated by removing or bounding them. Interrupts are usually unpredictable by nature and add even more timing uncertainty to the system. Handling interrupts generates accesses to shared components (i.e., caches and bus) that impact the execution of other cores. In case of interrupt storms, the core handling the interrupts will highly impact the execution of other cores. In this paper, we study and quantify interrupt interference and propose a novel interrupt manager to bound and monitor interrupt-related interference. Our approach, implemented on a commercial RTOS, offers a low execution time overhead of 1.01% in the worst case while increasing applications' predictability by 84.1% and reducing the measured worst-case execution times by 5.9% in presence of interrupt storms. We further propose a set of constraints to monitor the system and ease its analysis.

5.1 Introduction

Safety critical systems require a high level of confidence, robustness and determinism. Many safety-critical domains rely on partitioned real-time operating systems (RTOS) that isolate applications into partitions [87]. Isolation is enforced on the timing domain by statically scheduling partitions, sharing the CPU time and ensuring scheduling determinism. The CPU time is divided into time windows during which a defined partition executes. The global schedule is called the Major Time Frame (MAF). As depicted in Figure 5.1, partitions can run in multiple time windows during the MAF. Hardware components are also considered by resource partitioning. Memories and peripherals must be tightly controlled to ensure isolation but can also be shared between partitions. For instance, the ARINC-653 standard [11] provides means to achieve the space and time partitioning concepts. Figure 5.1 gives a visual representation of the space and time partitioning where five partitions share resources and CPU time. Partitions (P1, P2, etc.) can share the same resources, which is the case for

partitions 1 and 4 in our example.

In a single-core environment, partitions have an exclusive access to all resources they are allocated. In a standard scenario, a partition is put on hold only when it reaches its deadline or relinquishes the CPU after finishing its execution. This eases the analysis of the partitions' Worst-Case Execution Time (WCET). The WCET is a key concept of real-time and safety-critical systems, which cannot be certified and deployed if the WCET of the partitions is unpredictable.

Multicore architectures are no longer a novelty and have been used for different applications for years. Multiple benefits arise from those architectures: the added computational power, the ability to have higher redundancy by executing multiple times the same algorithms on different cores, etc. Slowly, the production of multicore architectures surpassed the one of single-core and the price of Commercial Off the Shelf (COTS) hardware dropped. For those reasons, the safety-critical community began to transition to multicore architectures [87].

Multicore processors embed many optimization components to improve their average execution time performances. Examples of those components are the shared caches, scratchpads, etc. Those components were already present in single-core processors and made the computation of partitions' WCET more challenging. However, the presence of multiple cores, contending to access the shared components, introduces new challenges [6]. As shared components usually can only serve one core at a time, contention and arbitration delays occur. This contention leads to unpredictable timing hazards which prevent the precise Worst-Case Execution Time (WCET) estimation and jeopardize the certification of highly critical systems [6]. Shared components' internal mechanisms can also create unpredictable delays. For instance, when caches are full, data needs to be evicted to load new data. Thus, a core can evict another core's data, which generates execution latency when the core needs to reload the evicted data. This phenomenon, called interference, prevents the certification of multicore systems in safety-critical domains such as aerospace systems.

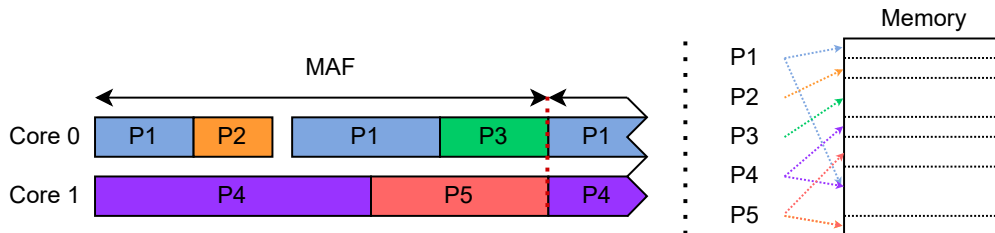


Figure 5.1 Major Time Frame scheduling for two cores and 5 partitions.

Interrupts are used to notify the CPU of an event. Upon receiving the interrupt, the CPU changes its execution flow to execute pre-defined RTOS code called interrupt handlers. The occurrence of interrupts is sporadic and can be difficult to predict in many scenarios. The interrupt handler generates traffic on shared components: shared caches are affected by the executed code as well as the main memory and shared bus. Hence, interrupts are unpredictable execution paths that generate interference. Another concern comes from interrupt storms that occur when a faulty driver or hardware component continuously delivers interrupt without control [88].

In this paper, we focus on interrupt-related interference and propose an interrupt manager to monitor and limit interrupt handling in the system. We target interference arising in the shared components used by interrupt (shared caches, DRAM, etc.). We provide an upper bound to the number of interrupts that can be handled during a given span of time and prevent interrupt storms. We further provide a comprehensive analysis of the interrupt manager and propose a run-time monitoring mechanism to ensure the correct system behavior. The contributions of this paper are the following:

1. We present a detailed study on the impact of interrupt interference in multicore systems depending on the memory configuration;
2. Based on the results of the study, we propose an interference-aware interrupt manager and implement it in an industrial RTOS. Our aim is to increase the system global predictability;
3. To ensure the correct behavior of the system, we introduce a set of run-time monitors based on formally defined constraints.

While the results presented in this paper are platform-specific, they are still general enough to provide a good representation of the impact of interrupts interference the system timing behavior.

5.2 Related Work

RTOS designers employ various approaches to remove and bound interference [82]. Shared cache partitioning removes the eviction-related interference [89]. Caches can be partitioned in *Sets*, *Ways* and *Lines* [12]. Cache way partitioning is achieved thanks to hardware configuration and Sets partitioning (also called Cache Coloring) is achieved thanks to memory translation [90]. Reducing memory interference and bus contention can also be brought by a better cache management [91, 92]. In [93] shared cache Denial of Service is tackled by

proposing an OS based regulation mechanism implemented on Linux. Cache locking is used to improve private caches usage and reduce traffic on the lower memory hierarchy [1, 94, 95]. Scratchpad memories are fast memories often implemented as a SRAM placed between the caches and the memory [96]. Event hough using scratchpads generates contention on the bus, these memories can service requests faster than the DRAM, hence reducing the time the bus is locked for the transaction [97].

Although the focus of our study on interrupt interference is on caches and scratchpads, DRAM interference [66] is also reduced. Nevertheless, a full study on DRAM is the subject of a future study. To further control the memory hierarchy, PALLOC [98] and MemGuard [43] were introduced. PALLOC is a bank-aware memory allocator. This means it knows to which DRAM bank the memory it allocates belongs to. This allows removing interbank DRAM interference. Furthermore, MemGuard allows controlling the number of accesses a core makes to the bus to bound bus contention interference. In [99] those concepts are integrated to provide a certifiable memory manager.

Different interrupt handling mechanisms are proposed to remove, limit or reduce drawbacks brought by interrupts in real-time systems. Multicore RTOS designs have been explored to reduce interference. In [100], the use of a dedicated core to handle interrupts is proposed, however, when using the Asymmetric Multi Processing (AMP) architecture, this approach suffers from severe drawbacks as each CPU is independent. In [101], a comparison between coupling or decoupling handling and servicing interrupts is proposed. The authors show that decoupling the two concepts provides a better execution predictability.

In [102], the concept of ISR-tasks (Interrupt Service Routine tasks) is introduced to prevent low-priority interrupts handling to interfere with high-priority tasks. The concept is extended to IRQs in [103] and in [104] the authors propose to delay low-priority interrupts handling to let high-priority tasks execute. In [105], the authors provide a method to satisfy low criticality latency requirements. In [106], specific interrupts are replaced by polling to reduce the overhead caused by interrupt handling, however, this approach increases the event response time. Scheduling is also explored in [107]. In [108], the notion of interrupt budgeting appears. The author restricts the amount of CPU time allocated to interrupts. In this paper, we extend this concept by formalizing those constraints and adding new budgeting approaches. More recently, in [109] the concept of IRQ coloring is proposed to disable IRQs that reduce the quality of service of critical tasks. In this paper, we not only consider IRQs but all the interrupts that occur in the system.

Hardware specific interrupt controllers and CPUs were presented in [110–112] and [113]. However, in the context of safety-critical systems, where COTS hardware is targeted, using

custom additional hardware is prevented. In this paper, we rely only on software and use hardware features present in COTS platforms. In [114], a hypervisor is used to reduce interference, in [115] virtualization was also explored with mixed purpose OSES (RTOS and General Purpose OS) and in [116] virtualization was used to control IO (Input/Output) related interrupts. In this paper, we do not rely on virtualization due to the possible increase in latency they bring to interrupt [117] and the need to support legacy systems that do not provide virtualization.

In [118] and [95] cache locking allows the reduction of RTOS-related and interrupt servicing time. However, the study presented in [1] suggests that cache locking might increase the partitions' execution times and unpredictability.

Table 5.1 provides a summary of the interrupt management techniques present in the literature with the features we target in this research. Some references we provide in this section are not present in the table, as their primary goal is not to address interrupt interference. We choose five different comparison points to position our work. First, it is important to avoid using a dedicated core for performance reasons and our context forbids the use of additional hardware (not present in COTS platforms). Our approach allows the use of all cores by the applications. We further propose an extension to existing interrupt handlers. To this end, our approach does not require an expensive redesign of the scheduler to introduce dedicated interrupt tasks and event management. We further strive to provide safe execution time and interrupt count bounds while monitoring interference and notifying the system if those limits are exceeded. This is done thanks to the proposed run-time monitoring implementation. To the best of our knowledge, we are the first to propose this formalization and the associated monitoring mechanism.

In this paper we address the interrupt-related interference issue. We first propose an exhaustive study on memory configurations and their impact on memory interference (cache, DRAM and bus interference). Based on the study's results, we propose a COTS-compatible approach that does not require to redesign the RTOS's interrupt manager. We prototype and validate our approach into a commercial RTOS [79]. We offer a fine granularity to control each type of interrupt, from system calls to external interrupts. Compared to existing work, our approach is easily applicable to any interrupt handling architecture. Since we rely on a software approach, it can be generalized to any system that provides means to account for the number of interrupts in the system, which is what we provide in this paper. To avoid redesigning the RTOS's handling mechanism, we do not rely on additional ISR-tasks. We only rely on hardware features that are available COTS platform, without the use of additional hardware. To the best of our knowledge, this work is the first to propose a practical imple-

Table 5.1 Comparison with the state of the art

References	[100]	[102]	[103]	[104]	[106]	[107]	[110]
No Dedicated Core / Hardware	✗	✓	✓	✓	✓	✓	✗ ✗
COTS Compatible	✓	✓	✓	✓	✓	✓	✗ ✗
No Dedicated Tasks	✗	✗	✗	✗	✓	✗	✓ ✓
Provides Strict Bounds	✗	✓	✓	✓	✗	✗	✓ ✓
Interference Aware	✗	✗	✗	✗	✗	✗	✗ ✗

References	[111]	[113]	[115]	[116]	[118]	[95]	Our approach
No Dedicated Core / Hardware	✗	✗	✓	✓	✓	✓	✓
COTS Compatible	✗	✗	✓	✓	✓	✓	✓
No Dedicated Tasks	✓	✓	✓	✓	✓	✓	✓
Provides Strict Bounds	✓	✓	✓	✓	✗	✗	✓
Interference Aware	✗	✗	✗	✗	✗	✗	✓

mentation of a budgeting approach to strictly limit interrupts and the *Safety Net* approach to increase the system safety.

5.3 Interrupt and Memory Configuration Study

When interrupts are raised, the interrupt handler replaces the regular partition or kernel execution path. Multiple interrupt handling mechanisms are present in the literature [119]. In this paper, we rely on a manager that registers the interrupt to deliver it once the partition is scheduled.

5.3.1 Interrupt Path

The interrupt manager we describe in this section reduces the interrupt critical handling time to its minimum. No other interrupt can be triggered and the RTOS shall finish its processing before being able to execute any other task. Hence, this span of time should be as short as possible.

The *interrupt execution path* is shown inside the top dashed area of Figure 5.2. The *Pre-Handler* is in charge of saving the current process's context and filtering the interrupt if it is not allowed to execute. Then the *dispatcher* chooses which subsystem should handle the interrupt and notifies the desired core system of the interrupt using message queues. In

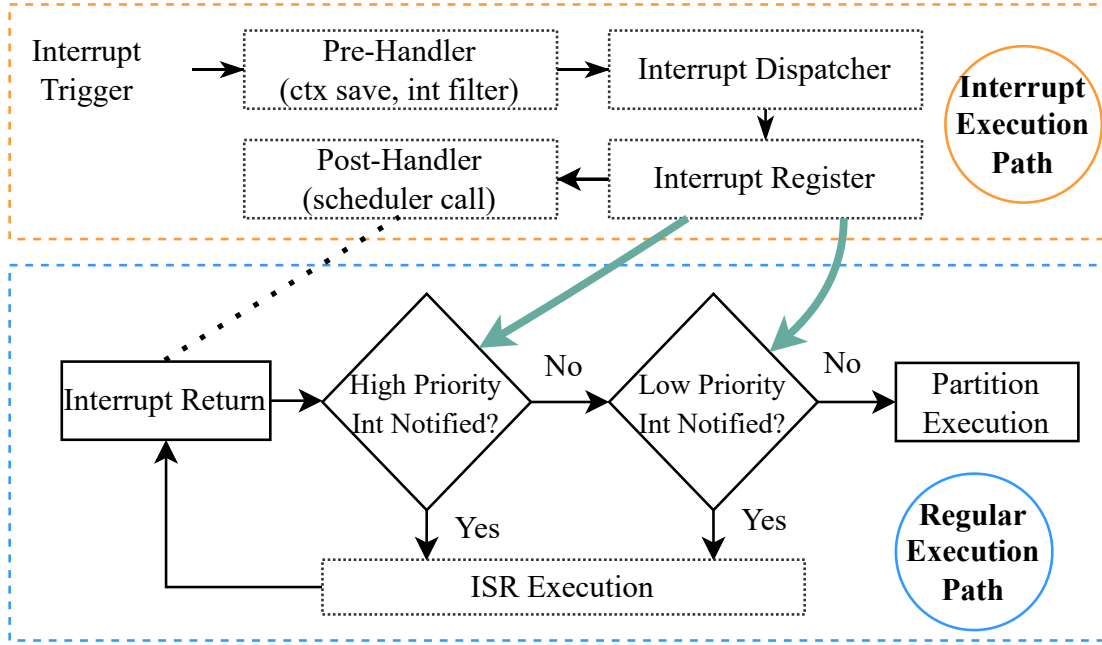


Figure 5.2 The RTOS Interrupt path is divided in two sub-paths to reduce interrupt's critical handling time.

the case of *IO interrupts* generated by peripherals, the ISR must be immediately executed to reduce IO latency. Once the interrupt is managed by the interrupt execution path, the minimal kernel context is restored in the *Post-Handler* and the scheduler is called.

The *regular execution path* is depicted in the bottom dashed area. The *scheduler* is the main entry point of the path. The *Interrupt Return Module* checks pending interrupts in the interrupts message queues. The corresponding ISR is executed and the interrupt is removed from the queue. Once all interrupts have been processed, the partition resumes its execution. The *regular execution path* can be interrupted at any time.

5.3.2 Memory Configuration

We study the different cache and scratchpad configurations available in our context and combine them to explore their advantages and drawbacks.

- *Cache Locking (L)*: interrupt data and instructions are loaded in the private caches of each core. The data and instructions cannot be evicted and accessing them does not generate any access to shared components;
- *Cache Partitioning (CP)*: shared caches are divided into areas allocated to cores. We divide the caches in N areas of defined size, where N is the number of cores. Shared

cache eviction interference is prevented by CP. We compute the size of each cache area by finding the one that yields the best performance (the best hit/miss ratio) for each core.

- *Disabled Shared Cache (D)*: shared caches are disabled. This removes eviction interference at the cost of a higher memory latency and an increased contention on the bus;
- *Scratchpad Memory: Core Platform Cache (CPC)*: core platform cache is used as a scratchpad, a small SRAM between the DRAM and the caches that services data and instruction faster than the DRAM and reduces contention time.

Figure 5.3 shows which components can be configured. Configurations can be combined when they are not exclusive. For instance, some configurations cannot be attained such as *D-CP* (partitioning the caches has no use when they are disabled). To study the impact of the configurations, we apply them to the *interrupt execution path* and the *ISR Execution* module of the *regular execution path*, as shown by components with dotted borders in Figure 5.2. The memory configuration does not apply to the rest of the system. For the RTOS and the applications, the caches, CPC and DRAM use the default SoC configuration. When not used as a scratchpad, the CPC behaves like a third level of cache, shared between all the cores in the system. If used as scratchpad, the CPC cannot be used as L3 cache for the RTOS and application and is only reserved for interrupt handling. The configuration is applied at startup and is never changed during run-time. We focus on interrupt-related interferences and applying memory configuration to applications is out of the scope of this paper.

5.3.3 Benchmarks and Exploration

We propose a benchmarking framework to measure execution times and other relevant metrics (cache miss, etc.) based on the Performance Monitoring Counters (PMC) present in COTS

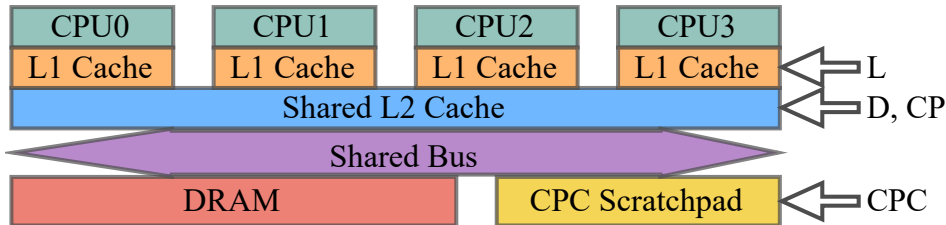


Figure 5.3 Studied hardware architecture with two levels of cache and an embedded scratchpad.

platforms. Most of the code executes in user space and the RTOS only needs to provide interfaces such as a system call handler. The benchmarking framework allows the user to generate the following interrupts:

- System calls: the framework raises a system call, the RTOS should provide a system call handler to be executed with the measurement framework;
- Internal interrupt: we rely on CPU’s free timers or inter-processor interrupts (IPI) and allow the user to add their own interrupt generation mechanism;
- External interrupt: an external timer is used to generate external (IO) interrupts. We also provide the API to integrate other interrupt generation drivers.

We provide an API to measure the partitions and ISRs behavior (execution time, cache misses, etc.) using performance monitoring counters. The API exposes the interrupt generation mechanism. Users can place API calls at any point of their applications to stimulate interrupts interference. The measurements are stored in a memory region defined by the user. Once the number of samples is reached, we dump these memory regions to a binary file using an external debugger¹. If PMCs are not available on the targeted hardware, the framework can collect data generated by external data providers such as debug and tracing probes. Some metrics, such as the execution times or the frequency of interrupts can be obtained from the RTOS itself (accounting the number of interrupts) or using internal timers.

The metrics we study can be unavailable on specific platforms. The framework is extensible to accept new metrics and means of measuring. If PMCs are not available (which rarely happens on COTS hardware), the framework’s hardware abstraction layer can be updated to take advantage of the OS’s features to gather the available metrics.

5.3.4 Experiments

Our experiment environment consists of a partitioned RTOS² running on the NXP T2080 board with four 1.8 GHz PowerPC e6500 cores, 32 KB private caches for data and instruction (Harvard architecture), 2 MB of shared L2 cache, 512 KB scratchpad (CPC) and 4 GB of DDR3 DRAM. Each core executes either applicative partitions or interrupts handling code. P0, P1, P3 and P6 are mapped to Core 0. P2, P7 are mapped to Core 1. P5 and P8 are mapped to Core 2 and P8, P9 on Core 3. The ten partitions that are allocated an execution budget of 10 ms each, Cores 1, 2 and 3 respectively generate and handle external, internal

¹We rely on the Lauterbach T32 debugger for this study: <https://www.lauterbach.com/frames.html?home.html>

²We use M-RTOS, a proprietary partitioned RTOS with paging enabled.

and system call interrupts at a rate of 1 interrupt every $0.05 \text{ ms} \pm 0.01 \text{ ms}$. This means that all three interrupt types can occur simultaneously in the system. The T0280 SoC has one interrupt controller capable of handling one interrupt at a time, queuing interrupt that happen during the handling of previous ones. Once the interrupt is generated and handled, the cores return to the partition's execution flow. Each applicative partition implements one benchmark that is part of the TACLeBench suite [81]: *Dijkstra*, *FFT*, *MatMult*, *MD5*, *MPEG2*, *QSort*, *FMRef*, *ADPCM_enc*, *BinSearch* and *LMS*. The benchmark environment is public [120]. For each evaluated interrupt, we register an ISR to measure only the RTOS execution path's interference. We measure the applications' execution times using the PMCs to get the time in CPU cycles. We further convert this number to nanoseconds.

Table 5.2 provides the memory characteristics of the applications we use as benchmarks. The information provided in this table was extracted using measurements (cache hit/miss ratio, bus accesses, etc.). The following characteristics are studied:

- **Cache Space:** Low (uses less 25% of the cache at a time); Medium (uses between 25 and 75% of the cache at a time); High (uses more than 75% of the cache at a time);
- **Cache Locality:** Low (hit/miss ratio under 1); Medium (hit/miss ratio between 1 and 2); High (hit/miss ratio over 2); Cache locality focuses on the shared unified cache (containing data and instructions);
- **Bus Contention:** Low (spends less than 33% of its execution time accessing the bus); Medium (spends between 33% and 66% of its execution time accessing the bus); High (spends more than 66% of its execution time accessing the bus);
- **Memory Access Pattern:** Constant (defined monotonic patterns); Phase (multiple phases can be detected); Random (no pattern nor phase can be defined);
- **Memory Usage:** Low (uses less than 512 KB of memory); Medium (uses between 512 KB and 4 MB of memory); High (uses more than 4 MB of memory);

The range chosen for each characteristic are the ones encountered in safety-critical avionic software. All code used in the experiments is compiled with the GCC compiler and uses an optimization level of O1. This level of optimization is chosen to facilitate certification (for requirements to binary code traceability).

We carefully set the benchmark applications' dataset size so each partition takes between 3 ms and 6 ms to execute and we allocate them a time window of 10 ms to ensure that no deadline miss occurs due to interference. We do not synchronize cores interrupt generation.

Table 5.2 Benchmarks application characteristics extracted from performance monitoring executions.

Benchmark	Cache Space	Cache Locality	Bus Contention	Memory Pattern	Memory Usage
Dijkstra	Medium	Low	High	Random	Medium
FFT	Medium	High	Low	Constant	Medium
MatMult	High	High	Low	Constant	High
MD5	Low	Low	Low	Phase	Low
MPEG2	Medium	Medium	High	Phase	High
QSort	High	Medium	High	Phase	High
FMRef	Low	Medium	Medium	Constant	Low
ADPCM	High	Medium	Medium	Constant	Medium
BinSearch	Medium	Low	High	Random	High
LMS	Medium	High	Medium	Constant	Low

This allows unpredictability of the interrupts occurrence. Once the interrupt is generated, the core returns to the partition's execution. We isolate the interrupt path to place it in specific memory regions. This allows us to load the code path in private caches (L), scratchpads (CPC) or DRAM.

We explore the following memory configuration combinations: ① CP, ② CPC, ③ CPC+CP, ④ CPC+D, ⑤ D, ⑥ L, ⑦ L+CP, ⑧ L+CPC, ⑨ L+CPC+CP, ⑩ L+CPC+D, ⑪ L+D.

The metrics we observe in this study are the minimal, maximal, average and median measured partitions' execution time over 100,000 executions. We define the predictability as the capacity of the system to produce the same results over the same dataset while maintaining the same execution time. Most of the time, using different datasets for the same partition would lead to different execution time, this is why we always study the applications under the same dataset. To assess the predictability of a partition, we rely on the standard deviation of its execution time over 100,000 executions. For space saving purpose, we provide the average of those metrics for the 10 partitions. The detailed results are provided in [120]. Each interrupt type is studied as presented in Figure 5.4 where the distribution of execution times of MPEG2 depending on the interrupt type is shown. A more detailed analysis of the other application is provided in Figure 5.10. To evaluate the amount of interference generated by a given interrupt type, we restrict the interrupts generated during the execution to the specific studied type. Interrupts of different types generate different interference as their handling may take different paths in the RTOS. The steeper the curve is, the more predictable the application is, while lower execution times are preferable. Even though the amount of interference for each interrupt type is different, the memory configurations have the same impact on each.

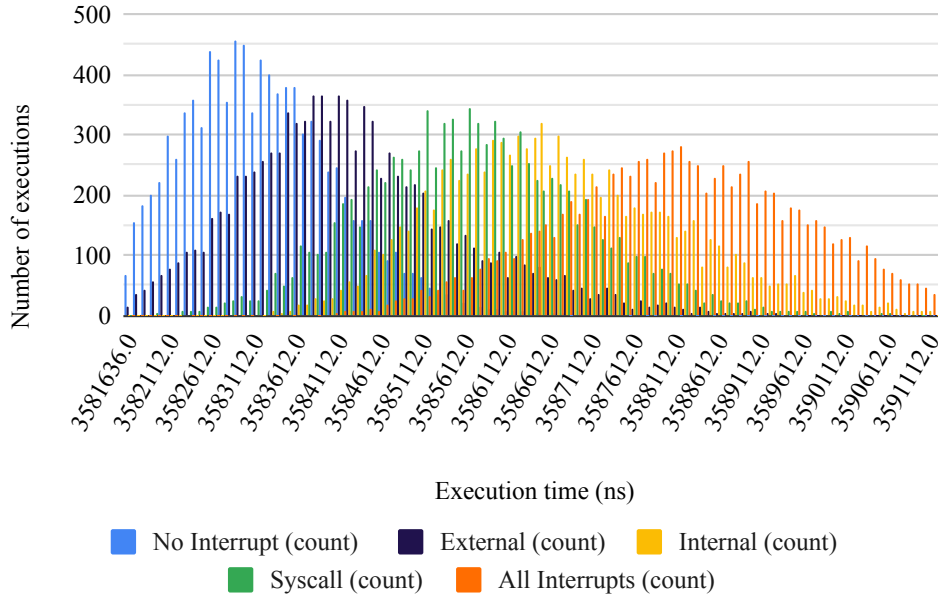


Figure 5.4 Distribution of the MPEG2 execution times (in nanoseconds) with different interrupt types.

5.3.5 Measurements

Execution Times

Table 5.3 provides the average measurements we gathered from our study. The measures are taken inside the application itself and exclude the interrupt handling execution time, which also includes the context switch required to execute the interrupt handling mechanisms. The first observation is that only CPC is capable of reducing the execution times. With the CPC configuration, the minimal execution time is reduced by 0.06% and the maximal execution time by 0.16%. The average (mean) execution time and median execution time are reduced by 0.10% and 0.12% respectively. Finally, the standard deviation of execution times (stdev), discussed in the next section, is reduced by 14.53%. The reduction of the contention on the memory as well as the faster servicing times allow partitions to access those components with less interference. The other configurations reduce the amount of resources (cache, bus bandwidth) allocated to the partitions, which impacts their execution time. L in addition of D provides the worst execution times. Disabling the shared caches increases the contention on the bus and memory. Locking the private caches for interrupts code and data also increases the contention on the bus for instructions and data that are not locked (the data must be loaded from lower memory, shared caches being disabled, the fetch is done directly to the

Table 5.3 Execution time comparison with the multicore environment.

Configuration	max %	min %	mean %	median %	stdev %
CP	100.84	100.58	100.75	100.74	80.15
CPC	99.94	99.84	99.90	99.88	85.47
CPC-CP	100.86	100.47	100.69	100.67	61.86
CPC-D	102.01	101.80	101.92	101.91	75.17
D	102.04	101.97	102.03	102.01	95.18
L	100.20	100.21	100.11	100.07	101.25
L-CP	101.10	100.86	100.93	100.90	79.29
L-CPC	100.20	100.19	100.11	100.07	101.55
L-CPC-CP	101.10	100.95	100.98	100.94	92.51
L-CPC-D	107.38	107.25	107.33	107.29	96.92
L-D	107.44	107.54	107.47	107.43	116.85

memory). Overall the impact on measured execution times for all the mitigation (except L-CPC-D and L-D) is very low (from 0.11% to 2.04%) and justifies their use to increase the partitions' predictability.

Predictability

For the same reason as presented in the last section, locking does not provide the correct mitigation to reduce the unpredictability (stdev). Disabling the shared caches slightly increases the predictability (5%). Reducing the contention on the DRAM using CPC improves the predictability by 15% and 25% when disabling the cache (D). CP (and CP + L) improves it by 20%. Finally, combining the CPC and CP provides the best results with an improvement of 38%.

Now that we looked into the global impact of memory configuration on our benchmarks, we propose to take three of the 10 partitions that have different memory usage (access pattern and intensity) and compare the obtained results. As mentioned earlier, the rest of the results are provided in [120]. We compare *MatMult* (*low intensity, low cache reuse*), *Dijkstra* (*medium intensity, medium cache space*) and *BinSearch* (*high intensity, medium cache space*).

We found three patterns when it comes to predictability improvement depending on the memory configuration. To illustrate those three patterns, we picked *BinSearch*, *MatMult* and *Dijkstra*. Figure 5.5 shows the disparity of the impact of mitigation solutions on the predictability of partitions. *BinSearch*, that highly profits from caches and suffers from memory contention, sees its predictability increased up to 84% with the use of cache partitioning and scratchpads. Figure 5.6 provides a detailed distribution of the execution times of *BinSearch*

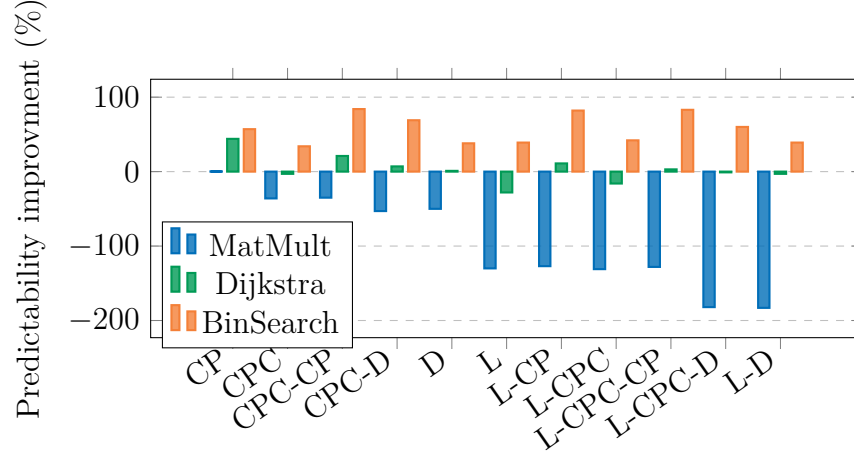


Figure 5.5 Predictability improvement (in %) for 3 partitions on the 11 memory configurations studied.

depending on the memory configuration. On the contrary, partitions for which caches are less used and that are highly memory intensive access the memory more often. *MatMult* has a low amount of access to the shared components but highly profits from private caches. It is only slightly sensitive to interference and reducing the amount of cache allocated to the partitions (CP and L), increasing the bus contention (L) or disabling the cache (D) will have a negative impact on the partition's predictability. In the case of *MatMult*, the unpredictability stays low even if it is increased as shown in Figure 5.6. Finally, for medium intensity partitions with medium cache use, CP reveals to be a good mitigation method.

5.3.6 Discussion

Based on the results, it is clear that the memory configuration to be selected depends on the partitions executing in the system. While most of the proposed configurations yield a better predictability on average, this is not the case for every partition. Thus, it is important to consider the trade-off offered for each configuration. In a system where only partitions such as *BinSearch* or *Dijkstra* execute, the use of CP or CPC-CP is suitable. However, when partitions such as *MatMult* are also embedded, it is important to quantify, by the use of measurements, the impact the configuration will have on such partitions. In our experiments, *MatMult* saw its unpredictability rise by up to 180%. However, it is still low, which allows to conclude that the impact of the applied methods is acceptable.

Following the characteristics defined in Table 5.2, we can make the following observations:

- Cache locking benefit small applications with low cache locality. Locking their data

will reduce the bus contention they generate and suffer from. However, using locking on applications with high cache locality or that have a medium to high cache space will negatively impact their predictability and execution time;

- Using CP will globally reduce cache interference, and thus increase predictability. However, for applications with a high cache space and memory usage, CP may increase the execution time as it reduces the amount of cache allocated to the application;
- Disabling the caches usually will increase the predictability of random memory pattern applications, but most of the time, will increase their execution time; Partitions with high cache locality will profit from CP and CPC, however, CL negatively impacts their predictability and execution time;
- CPC is a good ally for applications with high memory usage and high bus contention. By storing important data in the CPC, the application will reduce its contending time on the bus as the CPC services data faster than the DRAM;

We listed the following impacts of memory configurations on key points used in WCET: (1) Using the CPC as scratchpad reduces the ISR's memory accesses latency, however, it may increase applications' memory latency as the CPC is not used a L3 cache anymore; (2) Using cache partitioning reduces the amount of cache usable by applications, increasing the number of caches misses, but this does not impact WCET computation as the designer simply has to set a smaller cache size available for applications; (3) Cache locking highly reduces ISR memory access latency, which makes WCET computation for ISRs very simple. However, as for CP, it reduces the amount of cache available for the applications. (4) Disabling the cache makes all memory accesses have the same latency (when memory and bus interference are mitigated), but highly increases this latency.

We studied the impact of memory configuration on the interrupt handling mechanism, we expect such configurations to be extensible to the rest of the RTOS functionalities as our methodology is reproducible on any other components of the RTOS. However, this work is out of the scope of this paper and left as future work.

5.4 Multicore Interrupt Manager

The previous section presented a study on memory configuration to reduce memory interference generated by interrupts. However, the proposed methods cannot provide strict bounds on the amount of interference generated by interrupts at runtime. In this section, we present

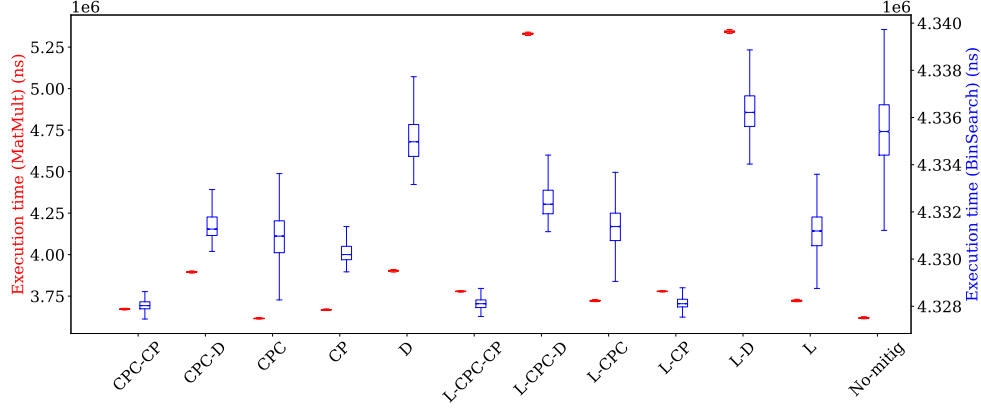


Figure 5.6 Execution times of BinSearch and MatMult depending on the memory configuration.

an interrupt manager capable of bounding the number of interrupts on a per-interrupt basis. It also relies on memory configuration to reduce its impact and provide better performance.

First, we define the system interrupt architecture and introduce the interrupt manager. Secondly, we present the monitoring system, composed of a formally defined set of constraints and run-time monitors to ensure the correct system behavior. Finally, we propose a complexity analysis of the interrupt manager and the monitoring system. The main goal of our approach is to increase the system global predictability by reducing the impact interrupt interferences have on the applications execution.

5.4.1 Interrupt Manager Design

The proposed interrupt manager is composed of two stages: the first stage executes when an interrupt is triggered. It is in charge of saving the current execution context, processing the interrupt by checking its source, validating various environmental data and finally registering the interrupt. This first stage cannot be interrupted; hence, its execution time must be the shortest possible. The second stage services the interrupt and can be interrupted by higher-priority interrupts.

Figure 5.7 provides an overview of the interrupt manager. The top rectangle shows the modules executed when handling an interrupt. The modules are used to filter interrupts and prepare the RTOS to service the interrupt. The bottom rectangle shows the RTOS servicing modules, used to react to an event notified by the interrupts. The two blocks communicate with queues present in kernel space, placed in the DRAM. We call the first interrupt stage the *Interrupt Handling Path (IHP)*. The IHP is shown in the top rectangle of the Figure

5.7. The second stage is executed when the RTOS partitions scheduler is called by the IHP. This stage is in charge of checking the interrupts that were raised, executing their ISR and updating the system state. This stage is part of the RTOS's scheduling mechanism and can be interrupted at any time. It is called the *Interrupt Execution Path (IEP)*. The IEP is shown in the bottom rectangle of the Figure 5.7.

In this paper, we use four interrupt types:

- System calls are raised by the user to execute a specific functionality of the RTOS;
- Software interrupts are generated by the RTOS to notify partitions of events, using the interrupt queues consumed by the IEP (hence, registering the interrupt without using the IHP);
- Internal interrupts come from the processor directly. For instance, exceptions (page fault) or internal timers generate internal interrupts;
- External interrupts are raised by peripherals and IOs to notify the system of various events (i.e., an IO that changed its state, etc.).

Interrupts are divided in three destinations: *User Interrupts* should be handled by the partitions, *Kernel Interrupts* should be handled by the kernel itself and finally *IO Interrupts* that should be handled by the IO System (a special type of partition in charge of handling peripheral management). All interrupts can be directed to one of the three destinations, but the general consensus is the following: system calls and software interrupts should be directed to *User* or *Kernel* interrupt handlers while internal interrupts should be handled by *Kernel* handlers only. Finally, external interrupts should be handled as *IO Interrupts*.

The IHP uses bitmaps and message queues to register interrupts. Each interrupt destination has its own bitmap and queue. Every time the scheduler is called, each destination checks its bitmap and queue and executes the ISRs related to the pending interrupts. The queues are sorted to allow priority based ISR execution. We further define the following priority order, from the highest to the lowest: *Kernel Interrupts*, *IO Interrupts*, *User Interrupts*.

To bound interrupt interference, we propose a budgeting approach to prevent handling an excessive amount of interrupts. Each interrupt is allocated an amount of times to be handled per regulation period. A regulation period is a variable span of time defined as a static schedule. In this paper, the regulation period has the same length as the current time window. A schedule example is provided in Figure 5.8. In this example, two partitions are executed on core 0. Partition 1 can handle two interrupts (2 and 8). Interrupt 2 is set as NMI,

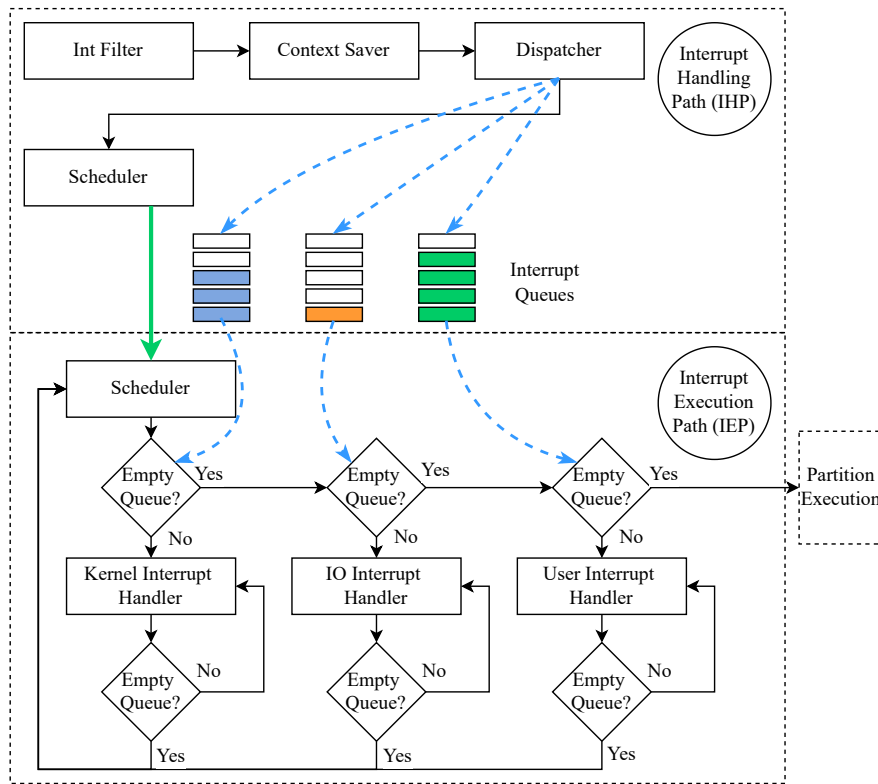


Figure 5.7 Interrupt management path in the RTOS. The manager is composed of two separate paths to reduce the interrupt management critical time.

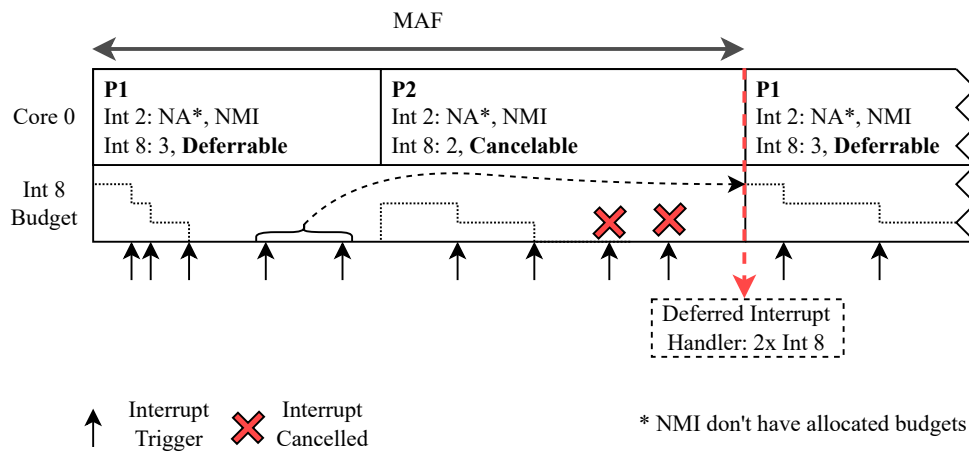


Figure 5.8 Interrupt schedule. Core 0 has two partitions, each handling *interrupts 2 and 8*. *Interrupt 2* is non-maskable while *interrupt 8* is deferrable for partition 1 and cancellable for partition 2.

thus, every time it is triggered, the RTOS needs to handle it. Interrupt 8 is set as deferrable and has a budget of 3 for P1 and of 2 for P2. In the first schedule of partition 1, the interrupt 8 is triggered 5 times. The first three times, the interrupt is handled and serviced. The next two times, the interrupt is only handled by the interrupt filter and the servicing is deferred when partition 1 is scheduled in the next MAF. The same scenario happens for partition 2. However, interrupt 8 is set as cancellable. In this case, when the interrupt 8 depleted its budget, the interrupt is simply canceled by the interrupt filter.

The regulation period keeps the following attributes for each interrupt:

- The interrupt budget indicates how many times the interrupt can be handled by the RTOS;
- The interrupt priority is used to set up the manager's priority mechanism;
- The interrupt class represents the action to execute when the budget is depleted.

Table 5.4 presents the interrupt classes supported by the manager. Depending on the interrupt class, an action is taken once the budget allocated to the interrupt is depleted.

Deferred interrupts are interrupts that were triggered after their budget was depleted and are configured as *Deferrable*. When the interrupt is triggered, it is handled by the interrupt filter (presented later in this section) and the current partition's deferred interrupt bitmap and table are updated. The table is updated to register the number of times the interrupt was deferred. Each partition owns its private deferred interrupt table and bitmap. After this step, the interrupt filter returns from the IHP directly to the partition's execution.

To prevent interrupt storms for cancellable interrupts, we the RTOS disables cancellable

Table 5.4 Interrupt classes supported by the manager.

Class	Action
Cancellable (C)	The interrupt is canceled, the filter directly returns from the interrupt to the partition's execution.
Deferrable (D)	The interrupt is registered in the deferred table and bitmap, the filter directly returns from the interrupt to the partition's execution.
Non-maskable (NMI)	The filter notifies the health manager (if required by the user) and the interrupt flow continues its execution.

interrupts that depleted their budget. The interrupt will be enabled again during the next time window which allows this interrupt index.

Figure 5.8 provides an example of the deferrable interrupt mechanism. Core 0 executes two partitions. *P1* is allowed to handle interrupts 2 and 8. While *interrupt 2* is set as non-maskable (the manager cannot cancel nor defer the interrupt), *interrupt 8* is allocated a budget of 3 and is deferrable. As *interrupt 8* is triggered five times during the first time window of *P1*, the manager only handles the first three and register as deferred the last two. The next time *P1* is scheduled, the manager raises a software interrupt to notify the partition of the deferred interrupts registered during the partition's last time window. This software interrupt gathers all the partition's deferred interrupts and is only raised once. It also has a low priority to prevent priority inversion and interrupt starvation.

Our design is composed of two modules: (1) The interrupt context switch module and (2) the interrupt filter.

The role of the Interrupt Context Switch (ICS) module is to update the interrupt configuration when a partition switch occurs. The interrupt context is defined as an array containing the configuration of interrupts that will be allowed to be handled during the next time window. Interrupt attributes presented earlier in this section are also part of this structure. The ICS implements three features:

- The ICS enables interrupts that were disabled in the previous time window and disables interrupts that were enabled in the past time window and should be disabled in the next one (and configured as cancellable) based on the interrupt configuration;
- The ICS loads the partition's deferred interrupts bitmap and, if needed, raises a software interrupt to notify the partition it has pending deferred interrupts to handle;
- Budgets for enabled interrupts are replenished and the deferred interrupts bitmap and table are reset.

The Interrupt Filter (IF) is the entry point for all interrupts. Before saving the execution context, the filter checks the interrupt budget. When budget is still available, the filter consumes one unit and the IHP continues its execution. If the budget is depleted, the filter check the interrupt class and applies the corresponding action presented in Table 5.4. If the interrupt is declared *Cancellable*, the filter disables or masks this interrupt to prevent it from being triggered again during the current time window.

5.4.2 Safety Net

The *Safety Net* concept was introduced by the CAST-32A [7] and, more recently, the AMC 20-193 [8] documents. It defines that in the advent of unmitigated interference or unwanted behavior, the system should be notified and take actions to recover. The *Safety Net* we present is a set of monitoring routines added at key points in the RTOS (context switch, interrupt handler entry point, etc.). We first propose a set of constraints that must be respected at run time. Moreover, we show the implementation of those constraints and their interface with the RTOS. This approach eases the system analysis by defining formal properties of the system (that can, for instance, be validated by model checking) and provides safer and closer bounds to execution times and system behavior.

We define the system variables used in the formalization of our constraints in Table 5.5. We refer to interrupts by their index.

In Equation 5.1, the number of interrupts handling for a given time window and a given interrupt index must be less than or equal to its defined budget. This constraint allows preventing interrupt storms. The interrupt counting is done by the RTOS and the budget is replenished periodically. Equation 5.2 further extends this concept: for a given time window, the time the RTOS takes to handle all interrupts occurring during this window must be under a given limit. Those rules are monitored in the interrupt handling entry routines and at the interrupt exit points using timers to keep track of the interrupts execution time. Each core has its own dedicated timer and keeps track of its own interrupts.

Equation 5.3 expresses that the number of accesses made by the interrupt handling mechanism and ISR to the shared bus during a given time window must not exceed a certain amount. As for other limits imposed by the *Safety Net*, this value is defined by the system designer. Equation 5.4 provides the same accesses constraint but applied to shared caches. We rely on Performance Monitoring Counters (PMCs) to keep track of the accesses made by the interrupt handling path. Thanks to the counters, the RTOS is notified when the number of accesses to the shared component exceeds the defined limit. To do so, we rely on interrupt that the PMCs can raise when they reach a defined threshold.

The constraint proposed in Equation 5.5 ensures that for each time window, only interrupts that are configured for that given time window should be triggered. If an unexpected interrupt is triggered, the RTOS is notified and relies on the health monitor to handle the fault. This constraint is monitored in the interrupt dispatcher.

Finally, we constrain the message queues used to register the interrupts show in Figure 5.7. Equation 5.6 provides the constraint used to ensure that, at any time during the execution,

Table 5.5 System variables definitions

Name	Description
Q	Length of the MAF in number of quanta
Q_{wf}	First time quanta of window w
Q_{wl}	Last time quanta of window w
$\mathbb{I}_{Types}$	Set of interrupt types indexes (0: kernel, 1: IO, 2: user)
$\mathbb{I}_A(w)$	Set of enabled interrupt indexes during the time window w
$\mathbb{I}(t)$	Set of interrupt indexes that were triggered during time quanta t
$I_C(t, i)$	Number of times interrupt i was triggered during time quanta t
$I_T(t, i)$	Execution time of interrupt handling path of interrupt i during quanta t
$I_{BA}(t, i)$	Number of times interrupt i accessed the bus during time quanta t
$I_{CA}(t, i)$	Number of times interrupt i accessed the cache during time quanta t
$I_{LC}(w, i)$	Maximum number of times the interrupt i can be triggered during the time window w
$I_{LBA}(w)$	Maximum number of bus access interrupt handling path is allowed to generate during the time window w
$I_{LCA}(w)$	Maximum number of cache access interrupt handling path is allowed to generate during the time window w
$MQ_U(t, i)$	Message queue i usage (number of messages) at time t
$MQ_L(i)$	Maximum allowed usage for queue i at any time

the queue usage should not exceed a given value. This value is provided by the system designer depending on the interrupt load of the system. This constraint is enforced by a monitor during the interrupt dispatch to the interrupt queues.

$$\forall w \mid \forall i \in \mathbb{I}_A(w), \sum_{t=Q_{wf}}^{Q_{wl}} I_C(t, i) \leq I_{LC}(w, i) \quad (5.1)$$

$$\forall w, \sum_{t=Q_{wf}}^{Q_{wl}} \sum_{\forall i \in \mathbb{I}_A(w)} I_T(t, i) \leq I_{LC}(w) \quad (5.2)$$

$$\forall w, \sum_{t=Q_{wf}}^{Q_{wl}} \sum_{\forall i \in \mathbb{I}_A(w)} I_{BA}(t, i) \leq I_{LBA}(w) \quad (5.3)$$

$$\forall w, \sum_{t=Q_{wf}}^{Q_{wl}} \sum_{\forall i \in \mathbb{I}_A(w)} I_{CA}(t, i) \leq I_{LCA}(w) \quad (5.4)$$

$$\forall w \mid \forall t \in [Q_{wf}; Q_{wl}], \mathbb{I}(t) \subseteq \mathbb{I}_A(w) \quad (5.5)$$

$$\forall i \in \mathbb{I}_{Types} \mid \forall t \in [0; Q], MQ_U(t, i) \leq MQ_L(i) \quad (5.6)$$

Whenever a monitor detects that a constraint is not respected, the said monitor send a signal through the Health Management (HM) system of the RTOS. The HM consist in a background process that handle system's errors and triggers associated actions defined by the system integrator.

5.4.3 Analysis and Interference Reduction

We propose to analyze the complexity of the two interference management modules. This section gives a theoretical view on the scalability of our interrupt management approach.

The first module to analyze is the interrupt filter. The filter must load the current interrupt budget, compare it to zero and, if equal, check the interrupt class. If not equal to zero, the budget is decremented and stored back in memory. If the class is *NMI*, the filter returns and the IHP continues. If the class is *Cancellable*, the filter simply returns from interrupt. Finally, if the interrupt is deferrable, the deferrable bitmap and table are loaded, updated and stored back. The filter has an algorithmic complexity of $\mathcal{O}(1)$ and performs in the best case 2 loads and stores, in the worst case 3 loads and 2 stores. Our implementation requires

13 assembly instructions. We provide the run-time results in Section 5.5. We further use cache locking to prevent the filter from generating any interference as proposed in [1]. In the advent cache locking is not available, we also measured the interference added by the filter when placed in a scratchpad memory. We did not observe a significant increase in interrupt-related interference in the shared caches. Thus it is a viable alternative to locking the interrupt filter in private caches.

We choose not to rely on Performance Counter interrupt to monitor the number of interrupts for the following reasons:

- We aim for fine granularity control, where each interrupt has a budget. PMCs usually account for the global number of interrupts in the system;
- On some target (it is the case for the T2080) it is not possible to know which PMC generated the interrupt;
- Not all RTOS support nested interrupts, which would be problematic since a interrupt that exceeds its budget will make the PMC generate an interrupt during the handling of the initial interrupt;
- PMCs interrupt can be reserved for specific purposes (e.g. bus budgeting) [39];

The interrupt context switch happens during every partition switch. Algorithm 3 provides its implementation where the complexity of the context switch can be studied. The first 10 lines set up the environment for the switch and are all constant operations. The *oldIntTable* and *newIntTable* contain the interrupt configuration of the previous and next time windows. Both those tables are arrays of integers terminated by a last entry of value *INT_MAX*. *INT_MAX* is the number of interrupt indexes in the system. The tables are sorted by interrupt indexes, in an ascending order. The loop from line 11 to line 37 iterates over the configuration tables of the two time windows. To prevent walking the first table and then the second table, we iterate the two at the same time, this allows better cache locality for the context table. On line 14, we check if we reached the end of both tables, meaning that we fully updated the configuration.

oldInt is the interrupt identifier pointed by the current entry in the previous window's table. *newInt* is the interrupt identifier pointed by the current entry in the next window's table. Due to the sorting of the two configuration tables, if *oldInt* is greater than *newInt*, it means that *newInt* was not in the previous window's table and should be enabled. If *oldInt* is less than *newInt* it means that *oldInt* is not in the next window's table and should be disabled. Finally, if *oldInt* equals *newInt*, it means that the interrupt was enabled for the two windows;

Algorithm 3 Interrupt context switch function

```

1: function INTCTXSWITCH
2:   hasDefInterrupts = False
3:   if NextWindow has registered deferred handler then
4:     currDefBitmap = defBitmaps[NextWindow]
5:     currDefTable = defTables[NextWindow]
6:   end if
7:   oldIntTable = intTables[PrevWindow]
8:   newIntTable = intTables[NextWindow]
9:   oldTableIdx = 0
10:  newTableIdx = 0
11:  while True do
12:    oldInt = oldIntTable[oldTableIdx]
13:    newInt = newIntTable[newTableIdx]
14:    if oldInt == INT_MAX AND newInt == INT_MAX then
15:      break
16:    end if
17:    if oldInt  $\geq$  newInt then
18:      context[newInt] = getIntConfig(newInt, NextPart)
19:      enableInt(newInt)
20:      if NextWindow has registered deferred handler then
21:        defBitIdx = newInt / 32
22:        defBitMsk = 1 << (newInt % 32)
23:        if currDefBitmap[defBitIdx] & defBitMsk  $\neq$  0 then
24:          updateBuffer(currDefBitpMap, currDefTable)
25:          hasDefInterrupts = True
26:        end if
27:      end if
28:      ++newTableIdx;
29:      if oldInt == newInt then
30:        ++oldTableIdx
31:      end if
32:    else
33:      context[oldInt] = 0
34:      disableInt(oldInt)
35:      ++oldTableIdx
36:    end if
37:  end while
38:  if hasDefInterrupts then
39:    raiseSoftInt(DEF_INT_ID)
40:  end if
41: end function

```

hence, the switch only updates the interrupt budget. On line 17, the switch perform these checks.

Finally, if an interrupt is to be enabled for the next window, we should check if this interrupt was deferred the last time the partition was executed. This is done by lines 20 to 27. The bitmap was implemented as an array of integers (32 bits), where each bit represents an interrupt. We must first compute the word in the bitmap that contains the interrupt bit, this is done by line 21. Then, we need to compute the position of the bit in the word at line 22. Finally, if the mask applied to the bitmap is not null, we update the deferred table buffer that the partition can use when handling the deferred interrupt notification that is raised at line 39. A particular attention must be given to the time the deferred interrupt handler executes. Deferred interrupts are not serviced by their respective handlers but all by the same handler defined by the user. This handler execution time must be taken into account in the schedule. The handler is given a priority which prevent it from executing before more important tasks present in the same time window.

All instructions contained in the loop can be done in constant time $\mathcal{O}(1)$. The functions *getIntConfig* (line 18) and *updateBuffer* (line 24) are only accessing a constant number of memory regions. Thus, we need to study the number of times the loop at line 11 is executed and its worst case. The worst case happens when, for each iteration $oldInt == newInt$. This means that we have to enable the interrupt, update the deferred information if needed and update both tables (old and new) indexes. In that case, the content of the loop still has a constant complexity. The maximal number of times the loop will be executed can be represented by Equation: 5.7.

$$|\{\mathcal{I}_A(n) \cup \mathcal{I}_A(p)\} \setminus \{\mathcal{I}_A(n) \cap \mathcal{I}_A(p)\}| \leq INT_MAX \quad (5.7)$$

This means that the maximum number of times the loop will execute equals the number of interrupts supported by the RTOS. Thus the final complexity of the context switch is $\mathcal{O}(INT_MAX)$.

5.5 Results & Discussion

To evaluate our approach, we propose to study two key points of our system. First, we measure the overhead added by the interrupt filter and interrupt context switch. Secondly, we propose a representative environment with benchmarks to evaluate the interference reduction brought by our method.

5.5.1 Experiments Setup

The validation environment is based on the NXP T2080RDB platform with four e6500 PowerPC processors at 1.8 GHz (indexed from 0 to 3), 32 KB instruction and data private caches, 2 MB shared caches and 4 GB of DRAM. We rely on a proprietary partitioned RTOS [79] and use the TACLEBench suite [81], picking ten applications to use as partitions. The following benchmarks were selected: *Dijkstra*, *FFT*, *MatMult*, *MD5*, *MPEG2*, *QSort*, *FMRef*, *ADPCM_enc*, *BinSearch* and *LMS*.

To study the interrupt filter and interrupt context switch time, we allow the RTOS to enable all internal and external interrupts of the platform. To avoid additional noise and only measure the execution time of the interrupt handling path, external interrupts are serviced by a simple ISR that only acknowledges the interrupt and returns to the RTOS interrupt path.

5.5.2 Overhead

We first measure the filter’s overhead based on 300 000 executions for each interrupt type (User, Kernel or IO interrupt) and filter actions: 100 000 executions for deferred, canceled and valid interrupts. Tables 5.6 and 5.7 present the execution times we measured with hardware performance counters (PMCs). In Table 5.6 we provide the filter’s absolute execution times measured with the PMCs. In Table 5.7, we compare those execution times with the measures done on the complete interrupt handling path. This last table gives the overhead of the proposed filter.

Table 5.6 Interrupt filter absolute execution times

Filter action	Mean (ns)	Median (ns)	Min (ns)	Max (ns)
Budget check / NMI	7.33	7.22	7.22	27.22
Interrupt canceled	11.14	11.11	11.11	64.44
Interrupt deferred	17.80	17.78	17.78	68.33

Table 5.6 shows that most of the time, the filter performs its checks as fast as possible. The average execution time is close to the minimal execution time while the median execution time is equal to the minimal execution time. However, for some occurrences, its execution time can be increased up to 5.8 times (when an interrupt is canceled). This slight variability is due to optimization features of the processor (e.g., branch predictor, caches, etc.). To draw this conclusion, we compared the PMCs values for cache misses, branch miss-prediction and bus accesses. worst measured execution time happens when cache misses are the highest, also

Table 5.7 Interrupt filter overhead impact on interrupt handling

Filter action	Mean (%)	Min (%)	Max (%)
Against Kernel Interrupt Handling			
Budget check / NMI	0.42	0.42	0.46
Interrupt canceled	0.70	0.63	1.00
Interrupt deferred	1.06	1.01	1.12
Against User Interrupt Handling			
Budget check / NMI	0.39	0.39	1.15
Interrupt canceled	0.60	0.59	2.71
Interrupt deferred	0.95	0.94	2.88
Against IO Interrupt Handling			
Budget check / NMI	0.24	0.09	0.24
Interrupt canceled	0.37	0.14	0.57
Interrupt deferred	0.60	0.22	0.60

increasing the bus accesses. The worst case happens when the processor has to load all data and instructions from the DRAM and that the branch predictor misses every prediction.

The filter's overhead is shown in Table 5.7 where we present the percentage of execution time the filter takes in the complete handling path's execution time. The usual scenario happens when interrupts do not deplete their budget. On average, this case represents an overhead of 0.42%, 0.39% and 0.09% for Kernel, User and IO interrupts. IO interrupts take longer to be handled due to the IO system execution, this is why the filter's overhead is much lower than the others interrupt types. User interrupts are the fastest to handle as they have less validity checks by the RTOS. Considering other scenarios, on average the highest overhead happens when deferring an interrupt. Indeed, that process needs to register which interrupt is deferred and increase its deferred count, which takes more time than just decreasing the interrupt budget or canceling the interrupt. Regarding the minimal and maximal execution times, the highest overhead occurs for user interrupts when deferring the interrupt which represents 2.88% of the interrupt handling time. Again, due to the highest amount of memory accesses in that case, if caches and other optimization factories are not in a favorable state, the filter is likely to take more time. Finally, the average overhead peaks up to 1.01%, which in our case is acceptable and was validated in the industry.

The second overhead point we study is the interrupt context switch. We compare execution time of the interrupt context switch with the partition's switch that encompasses it. During interrupt context switch, we encounter two scenarios: (1) no interrupt was deferred during the partition's last time window or (2) interrupts were deferred during the partition's last

time window. As presented in Section 5.4.3, the interrupt context switch execution time depends on the number of interrupts the previous and next partitions have configured.

Thus, we propose the following experiments: 8 partitions are set up to execute sequentially and have a distinct set of interrupt to be handled. We use the same scheduling for every measurement. We measure the execution time of the context switch without any deferred interrupt and we measure it again with all interrupt deferred. We gather the measurements over 100 000 executions. The interrupt configuration is given by Table 5.8. The maximal interrupt index for the RTOS is 290 ($\text{ISR_MAX} = 290$). Our aim is to study the scalability of our approach.

Table 5.9 provides the measurements on the context switch time. The relative value gives the percentage of time the interrupt context switch takes in the complete partition switch procedure while the absolute value represents the time that the interrupt context switch takes.

The time required to switch from 0 to 5 configured interrupts (a total of 5 interrupts to configure) is 308.12 ns or 3.18% of the total partition switch time. An important point to understand is that switching from 5 to 10 disjoint interrupts, as proposed in our experiment, is equivalent to a switch that needs to configure 15 interrupts (disable 5 and enable 10 others). Our results show a linear increase of the execution time, as expected by our previous analysis. The deferring mechanism adds more overhead and multiply the switch time by 2.67 on average. This is due to the buffer manipulation used to notify the partitions of deferred interrupts. Figure 5.9 provides a graphical interpretation of this scalability analysis.

We asked our industrial partner to provide the number of interrupts in certified critical systems. The number of interrupts in the system did not exceed 35 interrupts in total. From the previous results, we can interpolate that the overhead added by the interrupt context

Table 5.8 Interrupt configuration for the context switch experiment

Partition ID	Enabled Interrupts Range	Interrupt Count
1	0 - 4	5
2	0 - 9	10
3	0 - 24	25
4	0 - 49	50
5	0 - 99	100
6	0 - 149	150
7	0 - 199	200
8	0 - 245	250

Table 5.9 Interrupt Context Switch execution times

Environment	Relative	Absolute
Switch without deferred interrupts	(%)	(ns)
5 interrupts	3.18	308.12
10 interrupts	4.80	465.36
25 interrupts	9.82	952.57
50 interrupts	18.44	1,787.70
100 interrupts	35.29	3,422.12
150 interrupts	54.05	5,240.53
200 interrupts	73.01	7,079.41
250 interrupts	91.90	8,910.30
Switch with deferred interrupts	(%)	(ns)
5 interrupts	9.97	966.24
10 interrupts	13.59	1,317.81
25 interrupts	27.55	1,701.73
50 interrupts	50.70	4,915.59
100 interrupts	91.23	8,845.16
150 interrupts	133.59	12,952.37
200 interrupts	176.45	17,108.59
250 interrupts	219.45	21,277.50

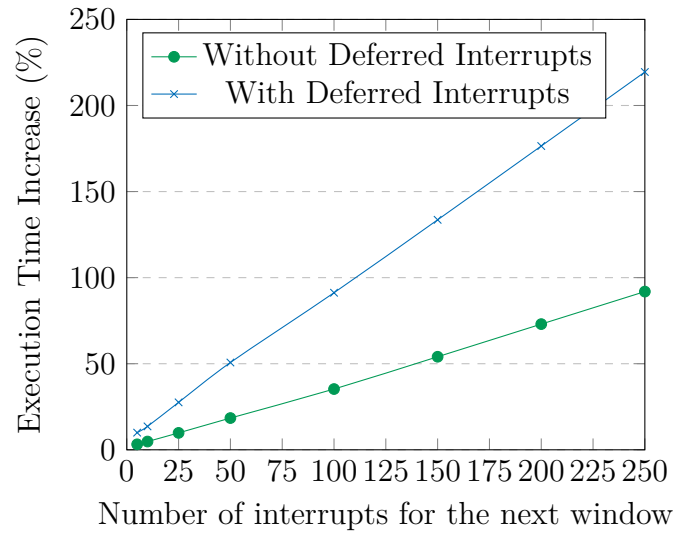


Figure 5.9 Interrupt Context Switch relative overhead depending on the number of configurable interrupts.

switch for 35 disjoint interrupts is around 13% or 1287 ns. It is important to highlight that the results we obtained for the context switch time with deferred interrupts provide an over-pessimistic scenario. In this experiment, all interrupts were deferred, which requires the interrupt context switch to update the partition's notification buffer for all interrupts. This is a pessimistic approach because in a deployed system, some interrupts simply cannot be deferred (page faults, critical events timers, etc.) and some other interrupts can be canceled (faulty hardware, debug interrupts, etc.).

Regarding the memory overhead, the data structures needed by our approach are detailed in Table 5.10. The highest amount of memory is used by deferred tables that keep one entry per interrupt per partition. We chose this trade off instead of using linked lists or other structures capable of storing sparse data because the execution time overhead introduced by these structures is orders of magnitude higher than a simple table in the memory. The interrupt filter does not have to walk a list or a tree, it just adds the interrupt offset to the table base address to increase the number of times the interrupt was deferred. Table 5.10 provides the size of a single unit of the data structures. To obtain the complete size of the structure, one must use the scale factor based on the platform's configuration. The column *Application* represents the memory overhead when applied to our environment on Core 0, using a maximal number of interrupts of 289 ($N = 289$) and 10 partitions ($M = 10$). All measures are provided in Bytes (B).

5.5.3 Safety Net Overhead

The *Safety Net* monitors are placed at given points the RTOS code. Their execution impact the global RTOS execution time as well as the functionalities the monitors are implemented in. To provide a memory and execution time overhead, we measure the RTOS as a whole and the monitored functionalities execution time and footprint in memory. We take the measurements for when the monitors are embedded in the RTOS and when they are noted.

Table 5.10 Memory overhead introduced by the interrupt manager with N interrupts and M partitions. All sizes are given in bytes (B).

Structure	Scale factor	Application
Context Table	$4 \times N$	1,156 B
Deferred Callbacks	$4 \times M$	40 B
Deferred Bitmasks	$(\frac{4 \times N}{32} + 4) \times M$	400 B
Deferred Tables	$4 \times N \times M$	11,560 B

Table 5.11 Overhead introduced by the *Safety Net* monitors.

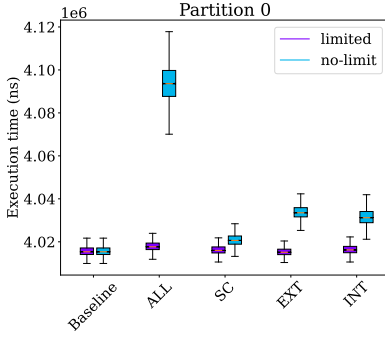
Monitor	(1)	(2)	(3)	(4)	(5)	(6)
Execution Time	0.46%	0.12%	0.20%	0.19%	0.05%	0.09%
Code Size	32 B	120 B	48 B	40 B	12 B	24 B
Data Size	4 B	12 B	28 B	28 B	8 B	16 B

Table 5.11 provides the increase of the measured worst case execution time, code size and data size observed when using the *Safety Net*. Each monitor, called (1), (2), (3), (4), (5) and (6) corresponds to the constraints defined by Equations 5.1, 5.2, 5.3, 5.4, 5.5, 5.6. Our observations show an increase of code size by 276 Bytes, which is acceptable given the increased safety that the monitors provide to the system. The increased data size of 96 Bytes is limited by the fact that most information required by the monitors are already provided by the RTOS. The execution time overhead is comprised between 0.05% and 0.42% of the functionalities execution times. This overhead is deemed acceptable in our context since it does not impact timing reliability of the system. Finally, the highest increase of 0.42% of execution time introduced by the monitor (1) is due to the fact that the studied functionality is embedded in is already very short in terms of execution time (the preliminary assembly interrupt handler).

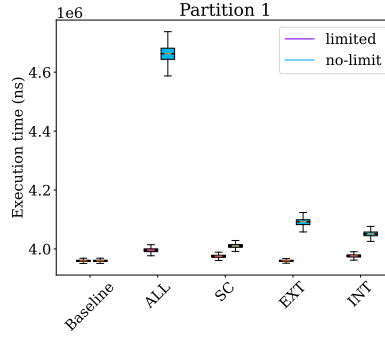
5.5.4 Interference Reduction

To assess the multicore impact of interrupt on the system, we rely on the benchmark partitions presented earlier in this section. As the execution times of the partitions varies from 3 ms to 7 ms, each partition is allocated 10 ms in the MAF to execute. P0, P1, P3 and P6 are mapped to Core 0. P2, P7 are mapped to Core 1. P5 and P8 are mapped to Core 2 and P8, P9 on Core 3. We study the impact of interference handling by all cores on the partitions execution time and predictability. Cores 1, 2 and 3 generates interrupts at random with a frequency varying between 65 kHz and 10 Hz. Interrupts types are dispatched between cores: Core 1 generates system calls, Core 2 generates internal interrupts (based on Inter Processor Interrupt, IPI) and Core 3 generates external interrupts (by programming an external timer). Each core handles the interrupt it generates. Core 0 does not generate interrupts to study the impact other Cores handling interrupt has on its partitions.

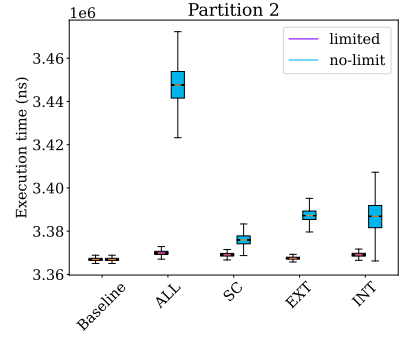
We execute each benchmark 100 000 times and measure their execution times. Each iteration is executed with the same values and configuration to keep the execution deterministic. We compare the Worst-Case Execution Time measured in the presence of interrupt with and



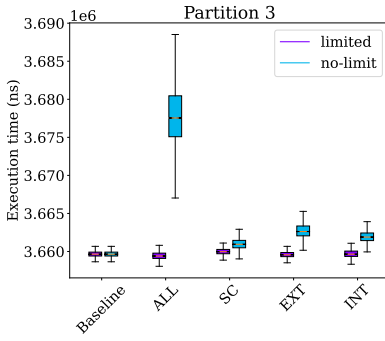
(a)



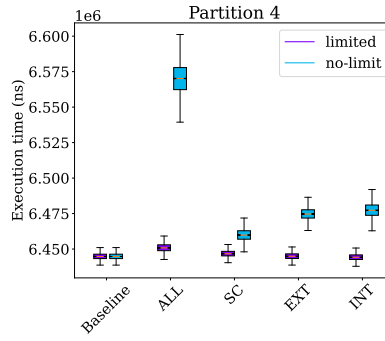
(b)



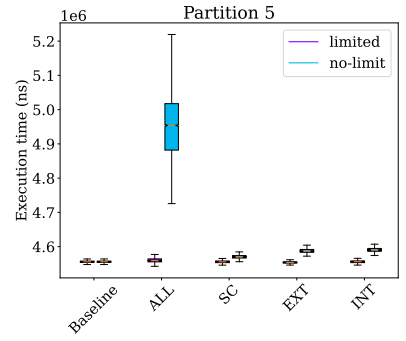
(c)



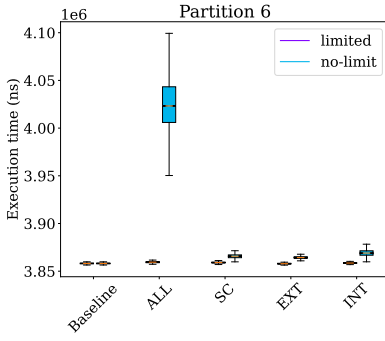
(d)



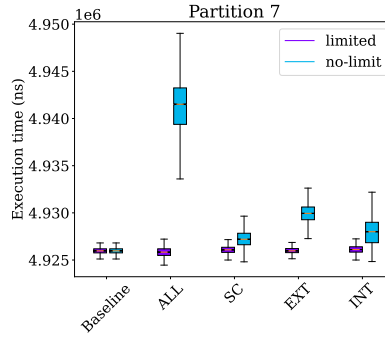
(e)



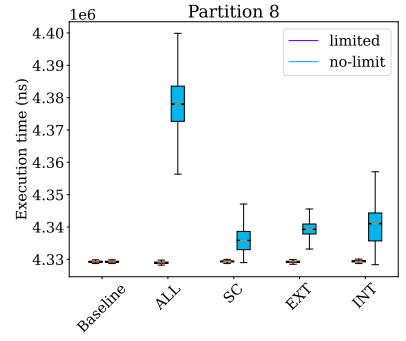
(f)



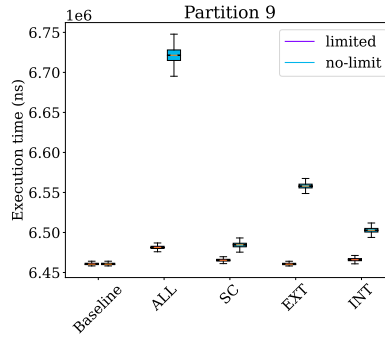
(g)



(h)



(i)



(j)

Figure 5.10 Execution times for the selected partitions.

without limitation from the interrupt manager. We also consider the standard deviation of the execution times measured as a mean to express the partitions' execution time predictability. A high standard deviation means a spread distribution of the execution times, which occurs when the partition's execution time is poorly deterministic.

The per-window budgets we provide to interrupts on Cores 1 to 3 are the following: Core 1 is limited to 2000 system calls, Core 2 is limited to 100 internal interrupts and Core 3 is limited to 200 external interrupts. When referring to interrupt types, we use the following: *ALL* refers to Core 1, 2 and 3 generating interrupts, *SC* defines that the only interrupts generated are syscalls, *INT* internal interrupts and *EXT* external interrupts. We also refer to execution without any other core generating interrupts as *Baseline*. In this setup, the memory configuration is the default one: shared caches are not partitioned, the CPC is set as L3 cache and the DRAM is not partitioned.

We first provide the detailed interference mitigation improvement brought by the interrupt manager in Table 5.12. The *Absolute Improvement* is the improvement brought by limiting interrupts compared to when no limitation is applied. Thus, for *P0*, the standard deviation of the execution times (*STDEV*) was reduced by 80.58% when limiting interrupts while the measured WCET was reduced by 3.8%. On average, limiting interrupts increases the partitions' predictability by 84.1% while the measured WCET was reduced by 5.9%. Even though the global improvement is promising, one can see that some partitions have a higher benefit from interrupt limitations. *P5* and *P8* see their predictability and measured WCET improved because they highly suffered from interrupt interference while *P7* is less impacted. Table 5.12 also provides the improvement compared to the *Baseline*. The values show how much using interrupt limitation reduced the gap (of *STDEV* and WCET) between the execution without limitation and the *Baseline* executions. This measure provides a better view on how our interrupt manager mitigate the effect of interrupts interference and bring the execution behavior closer to a single-core execution. On average, the predictability is 91.2% closer to the *Baseline* when limiting interrupt with an improvement up to 99.96% for *P6*. As for the measured WCET improvement, limiting interrupts brings WCET 87.17% closer to the *Baseline* with a reduced gap up to 99.88%.

Out of the 10 partitions studied, we selected three representative partitions for which the interrupt manager had a different impact. The rest of the results can be found in [120]. Figure 5.10 provides the distribution of the execution times under the five presented scenarios. In Figure 5.10, the *Baseline* distributions with and without limitations are the same. This means that using the interrupt manager limitation mechanism does not add any interrupt-related interference as explained in the previous section. While Figures 5.10 (a) and (c) show wider

Table 5.12 Interference mitigation using interrupt limitation mechanisms.

Environment	Absolute Improvement	Improvement to Baseline		
	STDEV	WCET	STDEV	WCET
P0 (ALL)	81.52%	3.56%	96.93%	84.90%
P1 (ALL)	82.68%	16.10%	87.30%	93.00%
P2 (ALL)	84.21%	3.87%	95.15%	89.96%
P3 (ALL)	81.82%	0.91%	89.13%	89.71%
P4 (ALL)	83.62%	2.80%	91.36%	81.11%
P5 (ALL)	92.22%	12.01%	96.26%	89.67%
P6 (ALL)	83.45%	7.83%	99.96%	99.51%
P7 (ALL)	61.67%	0.29%	70.84%	54.95%
P8 (ALL)	99.23%	6.88%	99.14%	99.88%
P9 (ALL)	81.06%	4.57%	85.96%	88.97%

dispersion when not limiting interrupts, Figure 5.10 (j) shows that dispersion can be kept low, but WCET increase is still present. Finally, an interesting observation is that not all interrupts generate the same type of interference. External interrupts generate more WCET interference (they increase the WCET) than others, while internal interrupts tend to generate more dispersion (reduce predictability) and less WCET interference than external interrupts. This is due to the way those interrupts types are handled. External interrupts usually have the same handling path, with few branching. Thus handling external interrupts adds nearly the same execution time every time it is executed. This is not the case for internal interrupts for which handling requires more branching and conditional executions.

To ensure that our approach not only reduces the impact of interrupt storms, but also mitigates the related interference, we choose to execute the *Quicksort* benchmark while increasing the number of interrupts triggered on the other cores. For each measurement, we execute the benchmark 20 000 times. We get measurements when limiting the interrupts (providing a budget of 200) and without limiting their handling. Even when we limit the interrupt handling, the interrupt is still triggered and then canceled by the interrupt manager. Figure 5.11 provides the benchmark's execution times distribution considering the number of triggered interrupts. The Figure 5.11 shows that by increasing the number of handled interrupts, the interference is becoming more prominent. When limiting the interrupts, interference is removed and no matter the amount of triggered interrupts on other cores, when the interrupts are canceled, no interference is measured. The system responsiveness to events is impacted depending on the interrupts classes. NMI interrupts will always be handled and serviced, which doesn't change the responsiveness to those events. For deferred interrupts, the handling latency will depend on the handling partition's period (as the partition will handle the

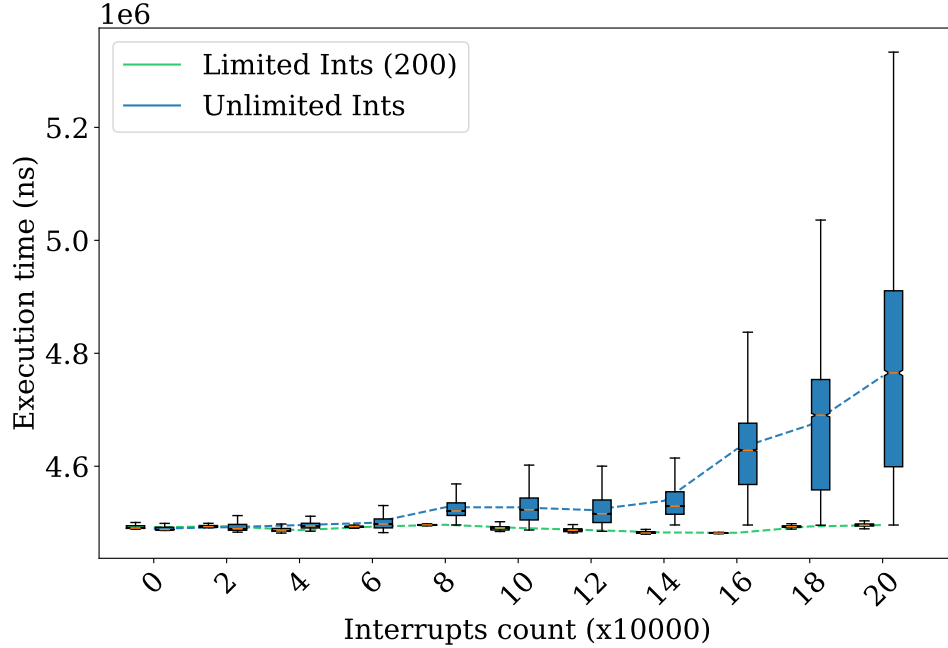


Figure 5.11 Execution times for the QuickSort benchmark, with an increasing number of triggered interrupts.

deferred software interrupt the next time it is scheduled.

5.5.5 Combining Memory Configuration and Interrupt Limitation

Finally, we apply the interrupt limitation mechanism evaluated in Table 5.12 and combine its effect to the different memory configurations studied in Section 3. We compare our results against the basic setup, where cache partitioning and cache locking are disabled, the scratchpad is used as last level cache and all caches are enabled. When only the interrupt manager is enabled, we refer to this setup as *VANILLA* and its results are the ones provided in Table 5.12 (compared to a setup where no interrupt management is done). A negative value depicts a gain (increase) of performance (WCET) or predictability (STD) while a positive value depicts a loss (decrease) of performance (WCET) or predictability (STD). The rest of the experimental setup is the same as the one used in the previous section. We provide the measurements in Table 5.13. For the sake of clarity and without loss of generality, we only provide results for P0 to P4, which provides a global representation of the system behavior. We highlighted the best and worst measurement for each partition and each metric observed.

The results confirm the observations we made in Section 3.5 where we compared the different memory configurations. Most interference are present in the shared caches. Table 5.13 shows

Table 5.13 Interference mitigation and memory configuration measurements against VANILLA setup.

Memory	P0		P1		P2		P3		P4	
	STD	WCET	STD	WCET	STD	WCET	STD	WCET	STD	WCET
CP	-20.56%	-1.3%	-45.87%	-0.39%	-25.83%	2.47%	-2.07%	-0.11%	-10.83%	-3.41%
CPC	-15.37%	-2.04%	-35.54%	-1.27%	-18.57%	-5.83%	-0.98%	-0.07%	-9.02%	-6.59%
CPC-CP	-28.38%	-2.65%	-46.96%	-0.85%	-53.61%	-1.20%	-5.30%	-0.05%	-12.43%	-4.22%
CPC-D	-21.94%	4.94%	-51.05%	6.72%	-38.53%	5.92%	-5.25%	2.73%	-10.93%	5.39%
D	-11.37%	7.6%	-20.60%	10.40%	-11.71%	6.18%	0.07%	3.60%	-4.31%	11.04%
L	-4.11%	2.83%	1.54%	-1.08%	-1.39%	-1.07%	4.31%	0.01%	3.97%	-0.98%
L-CP	-21.92%	-1.18%	-45.33%	1.02%	-27.48%	-0.61%	-1.02%	-0.03%	-10.95%	-2.77%
L-CPC	-4.13%	-1.81%	1.45%	-1.06%	-1.00%	-2.94%	4.59%	-0.98%	3.18%	-3.11%
L-CPC-CP	-13.31%	0.65%	-28.06%	4.14%	-15.00%	2.45%	-0.48%	0.99%	-7.79%	0.43%
L-CPC-D	-9.62%	8.54%	-15.44%	12.73%	-10.42%	30.02%	0.09%	5.43%	1.32%	21.01%
L-D	7.42%	8.76%	5.92%	12.15%	2.22%	31.28%	12.03%	6.01%	4.51%	21.01%

that applying CP and D greatly reduce these interference. CPC provides a better WCET but at the cost of less determinism. The configurations have a smaller impact on applications that uses private caches and few other resources like P3 (MD5). Combining CPC and CP reduces the WCET and increase predictability. While CP reduces cache interferences, CPC makes up for the loss of cache space by servicing cache misses faster than the lower memory levels. This can be seen with CPC-D too, but since caches are disabled, a bigger performance impact is present. Finally using cache locking does not prove to be effective in most cases. We conclude that the best memory configuration in our case is to use CPC-CP.

5.6 Conclusion

Interrupt handling can introduce unpredictable timing delays due to their sporadic nature but also due to the generation of interference in multicore environments. In safety-critical systems, controlling those behaviors is essential to ensure robustness and isolation between the components.

In this paper we presented the impact of memory configurations on interrupt-related interference in multi-core systems. We found that while some configurations yield to better global results by increasing the predictability of the system, it is important to take into account the impact of such configuration on all partitions in the system.

We proposed a novel interrupt manager that pursue a two-fold objective. First, we provide a flexible and fine-grained configuration for all interrupts in the system. We allow the system designers to provide budgets to interrupts and, when an interrupt budget is depleted, a set of actions to handle this scenario. This way, we prevent interrupt storms, reduce interference and increase robustness of the system. The second objective of our work is to ease the analysis

of the system. By providing bounds to the number of times an interrupt can happen and designing a simple and interference-free interrupt filter, we ease the system designer’s work. We also provide a *Safety Net* solution, formalizing the constraints the system is required to meet and monitoring those constraints during run time.

We implemented our approach in a real-life RTOS environment and focused on two main points: the overhead brought by our method and the reduction of multicore interference. Our experiments showed that on average, the interrupts filter overhead is between 0.09% and 0.42% of interrupt handling time increase for interrupt monitoring. When an action needs to be taken (deferring, canceling), the average overhead is between 0.14% and 1.01%. Those numbers are acceptable in our targeted domain. We also showed that multicore interference was reduced by measuring a decrease of the partitions’ execution time by 5.9% on average while the execution time’s predictability was improved by 84.1%.

CHAPTER 6 ARTICLE 3: SHARED CACHE ISOLATION FRAMEWORK FOR PARTITIONED SYSTEMS

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Felipe Gohring de Magalhaes, Gabriela Nicolescu and Serge Harnois

Submitted in Springer - Design Automation for Embedded System. Submission date:
2022/08/30

Abstract: The transition from single-core to multi-core architectures in safety-critical systems presents many challenges in terms of resource isolation. Due to the concurrence on resources, phenomena called interferences introduce unpredictable delays in execution time. This prevents a correct estimation of the Worst Case Execution Time of applications executing on the platform. Shared caches being a key component of the memory hierarchy, are one of the most prominent sources of interference and cache partitioning is the method of choice to mitigate shared cache interferences. In this article, we propose a novel framework to wisely choose the size of each cache partition based on a multi-objective optimization approach. We compare our work with the state of the art solutions in the literature. Our results show that we provide nearly-optimal solutions while ensuring fairness in terms of resource sharing.

6.1 Introduction

Real Time Systems (RTS) offer insurance in terms of execution deadline, quality of service and overall robustness. Real-Time Operating Systems (RTOS) provide services and ease the software design process focusing on determinism of execution using strict resource management and tasks scheduling. RTOSes highly control resource usage, by enforcing their access through standardized interfaces (ports, API, etc.) [121].

To ease their analysis and bring higher confidence in critical systems, partitioned RTOSes are employed. This category of RTOS adds new constraints to tasks and resources management to ensure a more deterministic and controlled execution. Each application is seen as a completely isolated partition and executes as if no other application was in the system [4].

Most certified partitioned systems rely on a single core for execution, even on multi-core processors. This choice of architecture ensures that only one task executes at a given time to allow tasks to have the exclusivity on the resources. However, single-core processors are

getting replaced by the more efficient multi-core architecture [6]. They provide better energy management, parallel computing and execution optimizations. For these reasons, aerospace actors want to transition to Commercial Off-The-shelf (COTS) multi-core processors to support their new systems [87].

Due to the different optimization facilities present in COTS processors (branch prediction, prefetch units, etc.) Worst-Case Execution Time (WCET) computation became very difficult if not impossible [5]. In multi-core processor, multiple tasks can create contention and concurrent use of shared resources. Thus, two tasks from two different partitions can impact each other's execution time. Interferences reduce execution timing predictability, impact performance and prevent correct WCET analysis [6].

CPU caches are very fast memory units placed between the CPU and the rest of the slower memory hierarchy. Caches temporarily store data that were recently accessed to serve the CPUs faster. Due to their high cost, caches are small memories and the data they store are constantly replaced with new data when no space is left. This process is called cache eviction. When different tasks execute on different cores, they may evict each other's shared cache data. Thus, a task that saw its data evicted needs to reload the evicted data. This introduces a delay in execution and voids the isolation principle defined for partitioned systems. As partitions are usually developed and analyzed separately, it is impossible to detect nor predict such delays. Cache interferences are one of the most impacting interferences in multi-core systems [122]. However, guidance documents begin to emerge to address and point out issues brought by interferences [7].

In this paper, we propose the following contributions:

Contribution 1 A novel method to classify applications based on their cache usage and behavior;

Contribution 2 Cache classification with a knowledge enabled evolutionary algorithm. The knowledge allows to explore the solution space more efficiently and provides better solutions in fewer iterations;

Contribution 3 We integrate the classification method and evolutionary algorithm in a framework to help the system's designer choose the desired configuration. This framework relies on solution clustering to ease the designers' decisions.

The rest of this article is organized as follows. In Section 6.2, we present the basic concepts and the background of our research. In Section 6.3, we describe the state of the art of cache partitioning and the different cache management methods proposed to reduce interferences.

In Section 6.4, we detail our approach to optimize cache coloring schemes. In Section 6.5, we present the results of our approach in an actual avionic environment. Finally, in Section 6.6, we conclude our work and present the future research points.

6.2 Background and Context

In this section, we present the context related to our research. We introduce partitioned RTOSes and their concepts and describe the mechanism known as cache coloring to manage shared cache.

6.2.1 Partitioned RTOS

In partitioned RTOSes, applications, known as applicative partitions, are scheduled in a cyclic and fixed manner, repeated all through the system's execution [11]. The System Integrator (SI) is in charge of gathering the different partitions and define their schedule based on multiple constraints (execution period, deadlines, etc.). The SI defines a *Major Time Frame* (MTF) that enforces tasks scheduling in a fixed way such as the one presented in Figure 6.1. The MTF is defined offline and cannot change amid the system's execution. Each partition has a period, a start time and an execution time. We formalize the system's model in Section 6.4. An applicative partition contains multiple processes (tasks) that are scheduled thanks to real-time scheduling algorithms (Rate Monotonic, Earliest Deadline First, etc.). Each process in an applicative partition shares the same resources but have different stacks.

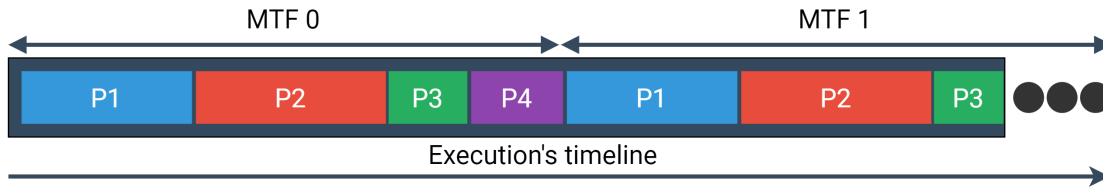


Figure 6.1 Example of Major Time Frame with four partition.

Partitioned RTOSes enforce an explicit resource partitioning between system's partitions. This concept defines how resources (IOs, memory, peripherals, etc.) are dispatched between partitions and can be accessed through the RTOS's primitives. Memory partitioning is usually enforced by Memory Management Units (MMU) to define memory areas accessible by a partition. The ARINC-653 standard [11] is an example of guidance for partitioned RTOS designers that proposes different services and implementation regulation for aerospace systems.

6.2.2 Shared Caches

Main memories cannot deliver requests as fast as the CPU is able to process them. This is why architecture designers opted for a hierarchical design with usually two or three levels of cache before the main memory. Caches are usually organized as a matrix. Rows are called sets and columns are called ways. This is the case for way-associative caches (N-way associative, where N is the number of ways). Other types of caches exist such as full-associative caches (only one way), hash-based caches, etc. Even though our method is applicable to any type of cache, we focus on way associative caches.

The following vocabulary is used when referring to caches. *Cache miss* occurs when a data requested by the CPU is not present in the cache. The cache must load it from lower memory levels (caches, scratchpad, memory, etc.). *Cache hit* occurs when a data requested by the CPU is present in the cache and serviced by the cache. *Line eviction* occurs when a data is replaced by another in the cache.

6.2.3 Cache Partitioning

To prevent unwanted cache evictions system designers can rely on two methods: disabling shared caches or partitioning the shared caches in multiple private areas.

Disabling caches implies a huge performance penalty, hence in this article we explore set partitioning and more specifically rely on a technique called cache coloring [123]. Cache coloring relies on the capacity of the CPU to use virtual addressing and the caches to use physical addressing. The page number is chosen by the RTOS when allocating memory for the partition. In Figure 6.2.(a) we show the representation of the physical address. This physical address is selected by the RTOS and associated with the virtual address. Even-though contiguity on the virtual addresses is required, contiguity in physical addresses is not mandatory. The way physical addresses are seen by the cache is presented in Figure 6.2.(b). The index field allows to define in which group of sets the data will be loaded. We use this mechanism to color (partition) the cache in different areas that we allocate to the application partitions.

Other cache partitioning methods can be relied on, but are not always suitable. Way partitioning makes use of additional hardware features which are rarely present on COTS processors. Line partitioning combines set partitioning and way partitioning. This technique provides a finer granularity at the expense of a higher overhead. Our implementation and experiments rely on set partitioning to be as portable as possible. However, our frameworks can be easily applied to the other cache partitioning methods.

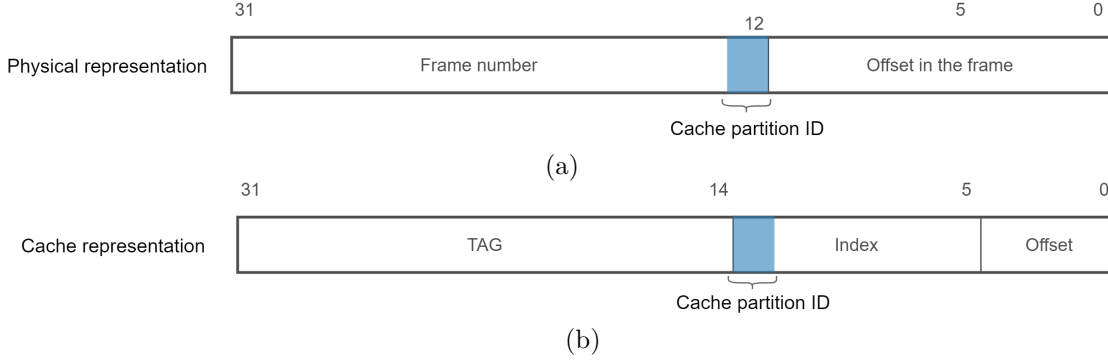


Figure 6.2 Addresses representations in the system. (a) is the physical, or effective, addressee to which a virtual address is associated. (b) shows the physical address seen by the cache, showing in which group of sets (index field) the data will be loaded.

6.3 Related Work

Cache partitioning has been studied in different contexts, ranging from resource isolation to performance improvement using semi-partitioned approaches. Dynamic partitioning, where the partitioning scheme can be modified during execution, may perform better. However, it introduces non-deterministic behaviors which prevent certification in critical systems as every configuration value should be validated before system deployment [13]. In this article we focus on static partitioning, where the partitioning scheme does not change, to allow system compliance with certification authorities recommendations. Furthermore, we provide an offline optimization tool that produces configuration outputs that respect the certification procedures.

In [15], bus contention is reduced thanks to the use of an offline cache allocation algorithm. To speed up the execution time, the authors rely on fixed power-of-2 memory block size. In [16], the Utility-Based Cache Partitioning (UCP) algorithm is proposed. This method uses additional hardware in the system to monitor the applications at execution time. In [19], the authors rely on the Miss Rate Curves and Stall Rate Curves to select the size of each partition. They also propose to co-schedule applications with compatible cache behavior and provide improvements up to 17%. In [17], a semi-partitioned approach is proposed to improve execution performance and relies on dynamic programming to explore the solution space. In [18], the authors propose semi-partitioned caches that lean towards predictability improvement and help WCET computation in RTS.

In [20], the KPart approach uses run-time profiling using hardware performance counters to tweak cache partition size. Similar to KPart, the work presented in [21] uses an adaptive cache partitioning technique. In [22] and [23], prefetching is used with cache partitioning

to leverage cache performances. In [25], the principle of super pages is applied with cache partitioning to reduce execution costs implied by virtual to physical memory translation. To this end, additional hardware and co-scheduling are employed. In [26] the proposed approach is to use cache partition and co-scheduling of applications to increase performances. In [27] and [28], virtualization techniques are used to provide cache partitioning. An hypervisor is used in [27]. This choice allows to apply cache partitioning to closed source operating systems but also avoids modifying the code of each guest operating systems. More recently, [28] uses the concept of virtual CPUs to manage cache resources.

In [30], the authors overcome the hardware cost off the UCP algorithm by using a trained model to predict the behavior of the cache based on the cache allocation. A run-time reinforcement based approach was proposed in [31]. The agent learns how to partition the caches depending on the state of the system. In [2] a genetic algorithm is proposed to reduce the worst case utilization of shared caches. The authors provide a mutation and a crossover operator. In this article, we extend this concept by proposing new operators using injected knowledge to provide faster search in the solution space. Using knowledge inject operators improves the research efficiency by avoiding exploring inherently bad solutions. This method allows to prune the search space and provide better results faster. Our results showed that our extension reduced the exploration time by up to 29%.

In [32], an integer linear programming (ILP) algorithm is used to propose a trade-off between sharing and isolation for last level cache (LLC) components. However in our context (highest level of criticality) regulations prevent us from using such approaches. ILP has also been studied in [33] to map applications to cores in the system as well as defining the cache partitioning scheme. Other cache management techniques were proposed. In [37], the authors propose to define a cache miss threshold for each application in the system to ensure quality of service. Once the threshold reached, applications see their access to the cache disabled to ensure other applications can use it.

Different works have been proposed to leverage cache issues in multi-core processors. However, a study of the current literature reveals some challenges that were not addressed. First, most approaches rely on way partitioning because of its ease of implementation. This method starts to appear in multi-core systems but is not available to most architectures. While other approaches rely on additional hardware to provide cache management, such hardware is either not available on critical systems or cannot be certified in such systems as discussed in [38]. Most proposed work tend to optimize cache usage, using semi partitioned approaches while strict partitioning is mandatory in our context. Finally, the amount of solutions for the cache partitioning problem makes it complex for systems with a large number of applications

or cache partitions. Contrary to previous work, we allow any size of cache to provide a better optimization. Optimal algorithms may take too much time to find viable solutions for certain systems. We solve this issue by proposing a multi-objective heuristic to solve the cache partitioning problem. We provide a way to find a trade-off solution that both minimizes execution time and energy consumption in the system.

6.4 An Evolutionary Approach to Cache Coloring

In this article, we propose an offline meta-heuristic based on an evolutionary algorithm to find the cache partitioning scheme. This scheme is used to apply static cache partitioning at the system's boot time. For clarity purposes, we refer to cache partition as cache colors in the rest of this article.

Our approach is based on three main axes: cache behavior classification, solution space exploration and solution selection. We rely on metrics gathering to quantify the impact of the cache configuration. This process is done offline, using static and dynamic analysis methods. It provides information such as the cache use curve according to a certain metric as a function of the cache size. The curve can be based on the execution time, the energy consumption, the cache hit/miss ratio, etc. This process is not in the scope of this article and has already been studied in the literature [16]. Once the applicative partitions are classified, we use a genetic algorithm to find a set of potential solutions (cache configurations) that maximizes or minimizes an objective function. In this article, we propose two domain-specific core components: a new mutation and a new crossover operator. Finally, the third step is used to filter the solution proposed by the genetic algorithm. This step eases the system integrator process to identify and select the solution to be used in the system. The objectives are described thanks to the objective function defined by the user. This function allows to quantify the quality of a given solution. The value provided by this objective function is called the *fitness* of the solution.

The partial fitness functions f and h are expressed in terms of different metrics that are gathered in the system. We used equations 6.1 where the execution time and the energy consumption are two objectives to minimize. Note that we can add weight to objectives. Execution time ($eT(i, c)$) and energy consumption ($eC(i, c)$) are the metrics gathered for the partition i when it is allocated c cache colors. Finally, n is the number of applicative partitions in the system. We compute the global fitness function fit as shown in Equation 6.2, using a weighted sum of both objective functions where α and β are used to weight between the two objectives.

$$f(c) = \frac{1}{n} \sum_{i=1}^n [eT(i, c)] \quad h(c) = \frac{1}{n} \sum_{i=1}^n [\log(eC(i, c))] \quad (6.1)$$

$$fit(c) = \sum_{i=0}^M \alpha f(c) + \beta h(c) \quad (6.2)$$

Equations 6.1 and 6.2 are examples of fitness function the user may chose. The framework we propose accept a wide range of fitness function with multiple objective to minimize or maximize.

6.4.1 Cache Behavior Classification

In this section we propose a novel classification of applicative partition based on their cache usage and behavior. The first step of the proposed approach is to characterize each applicative partition's behavior depending on the cache sizes it is allocated. To perform this process we first gather, for all possible cache size, the value of the applicative partition's partial fitness function.

We propose four categories of cache behavior. In the rest of this section, we define each category and describe how to classify the applicative partitions.

Constant Behavior

Constant behavior is defined for a partial fitness function that is not impacted by the size of the cache (Figure 6.3.(a)). Applicative partitions showing constant cache behavior should be allocated as less cache as possible. We define two thresholds based on the mean of the partial fitness function for all possible cache allocation size. N is the number of available

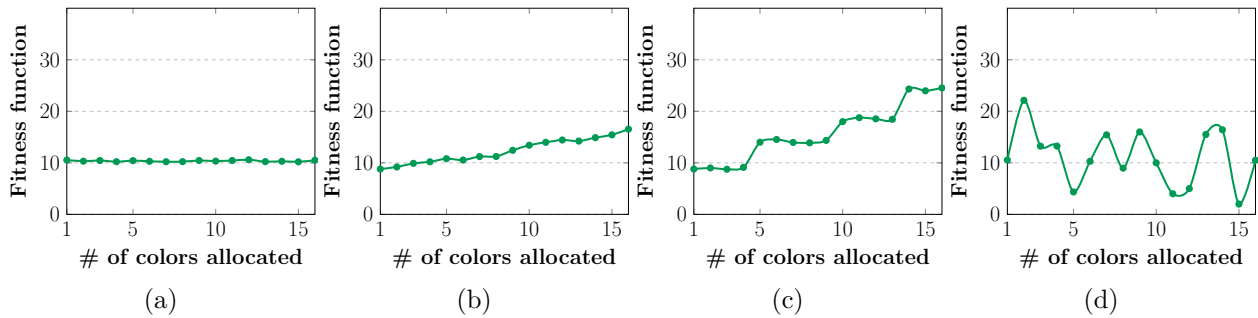


Figure 6.3 Cache behavior curves with (a) depicting the constant behavior, (b) the sensitive behavior, (c) the stepped behavior and (d) the undefined behavior.

colors in the cache.

$$T_{min}(i) = \frac{0.99}{N} \sum_{c=0}^N f(c, i) \quad T_{max}(i) = \frac{1.01}{N} \sum_{c=0}^N f(c, i) \quad (6.3)$$

Once both thresholds are computed, we check all partial fitness values of the applicative partition. If all the points are contained in the interval $[T_{min}, T_{max}]$, then the applicative partition is said to have a constant cache behavior.

Sensitive Behavior

We define a cache sensitive applicative partition as its ability to profit from the cache it is allocated. Such applications usually show data locality and reuse. When allocating cache to the applications, the algorithm needs to make sure they receive as much cache space as possible. Figure 6.3.(b) shows a typical sensitive application cache usage curve. Many other curves are possible. However, we decide to restrict sensitive applications' curves to specific features: (1) The curve must be either increasing or decreasing but cannot be both; (2) Variation from a cache capacity i to a cache capacity $i + 1$ is acceptable; (3) Curves should be correlated to a polynomial approximation of degree at most 2.

To detect such behavior, we first approximate the mathematical function represented by the points on the curve. We use the least squares fitting method to this end. We rely on an iterative process depending on the polynomial degree of the approximate function. We first approximate the curve with a polynomial function of degree n . We then compute the Pearson correlation coefficient (PCC) between the approximated curve and the actual one. When the PCC is greater than 0.9, the curve is correctly fit and we stop the iterative process. If the polynomial order of the fitting curve is lower than 3, the cache usage function is defined as sensitive.

Step Behavior

The third cache usage curve shape we observed occurs in rare cases. Step usage is defined by a sudden gap in the partial fitness function between two cache sizes. This gap is followed by a constant phase until the next gap. Figure 6.3.(c) depicts this behavior.

Applicative partitions that have stepped cache usage curves usually work with chunks of data that change in size and space over time. When allocating more cache to these applications, we allow multiple data chunks to be fully or partially kept in caches. This increases the data reuse and reduces the execution time. We rely on Gaussian filtering to detect the presence of steps

and their location on the curve. This method is similar to an edge detection procedure [124]. A convolution between the Gaussian first derivative function and the cache usage curve allows to detect both increasing and decreasing steps. This convolution gives us spikes where the steps happen. Figure 6.4 explains the result of the convolution. The convoluted function is then normalized and only the spikes over a certain threshold are kept. We further process the convoluted function to transform the Gaussian bell into a single spike at the center of the Gaussian bell. The resulting spikes are the locations of the steps in the initial cache usage curve.

Undefined Behavior

Undefined behavior cache usage can be encountered in a few applicative partitions. Figure 6.3.(d) shows the erratic cache usage of such function. This can be achieved by randomly accessing memory, having dependencies on other applications impacted by their cache size, random path execution or other factors. Moreover, we also classify all applications that do not show constant, sensitive of stepped behavior in this category.

The detection must be executed in a specific order due to the nature of each behavior detection. We provide the following order:

- First, we check for constant behavior. Indeed, in the case of a constant application, if the sensitivity test is done prior to the constant test, it will succeed as we can approximate a constant function with a polynomial function of order 0.
- The stepped behavior detection is then run. In the case of a stepped application, if the steps are not too high (or low), we can find a polynomial function of order 2 or less that correlates with the application's cache usage curve.
- Next, we run the cache sensitive behavior test. If it succeeds, the application is categorized in the cache sensitive behavior category.
- If none of the previous steps were successful, the application behavior is categorized as undefined.

In the next section, we propose a genetic algorithm that relies on the cache behavior presented in this section. The cache behavior detection adds knowledge to domain specific mutation and crossover operators.

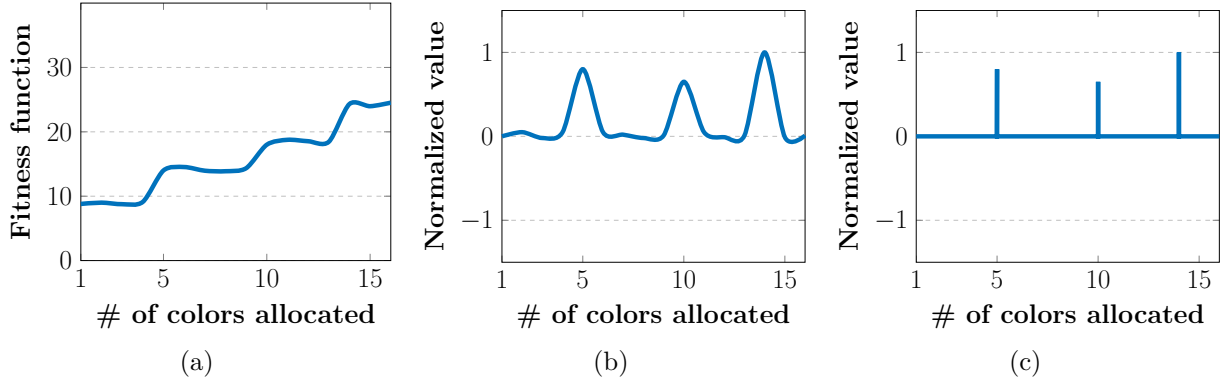


Figure 6.4 Cache usage curve detection process for step behavior. (a) is the application's cache usage curve. (b) is the result of the convolution between the Gaussian first derivative and the curve (a). (c) is the result of the processing after obtaining the curve (b).

6.4.2 Evolutionary Approach

In this section, we present our evolutionary approach to the cache partitioning problem. Our goal is to provide complete cache isolation between applicative partitions. Bad partitioning scheme may lead to performance reduction [125], and this is why we guide the user by providing them a set of solutions that maximizes (or minimizes) an objective function. We choose to rely on an evolutionary approach for two reasons:

- It allows convergence to a near-optimal solution in a reasonable time while an optimal algorithm would take too long to find the solution (sometimes weeks or months depending on the system), and;
- Evolutionary algorithms usually allow multi-objective research and hence the problem can be formalized to find a trade-off between different metrics.

We propose genetic algorithms (GA) that relies on a population of different solutions and make it evolve [126]. Other approaches usually maintain one solution that is evolved during the optimization process. The population allows the algorithm to have a set of solutions in which one can find solutions that will maximize one objective while other solutions maximize other objectives and contains trade-off solutions.

A genetic algorithm is based on an iterative process where each iteration is in charge of evolving the population by generating new offspring from the current generation. At the end of each iteration, a selective process based on the fitness function is used to maintain the population size under a certain threshold. The population will be evolved until a certain point called convergence, where new offspring does not bring sensibly better solutions. An

individual is a solution in the population and is represented by string of integers. The integer at index i represents the number of colors allocated to the partition i . We call this integer a *gene*.

The population initialization is done when we gather the partial fitness functions for each applicative partition. Each configuration represents an individual of the initial population. One may want to add more diversity to the initial population. To this end we allow the injection of randomly generated individuals. Fitness computation occurs every time new individuals are added to the population or modified. Based on the fitness of each individual, 60% of the population is kept based on its fitness and the other 40% are randomly selected (based on tuning experiments).

We rely on stochastic universal sampling to select individuals to be mated. The fittest the individual, the more chances it has to join the selection. We successively take two individuals from the selection and apply the crossover operation between them according the *crossover probability*. Contrary to usual crossover operators that are problem agnostic, we propose a metric guided crossover operator to allow faster convergence by using domain knowledge it is provided. We present this operator later in this section.

The last two steps, mutation and injection are used to keep diversity in the population and explore the solution space more efficiently. We rely on the pre-processing done during the cache behavior classification phase to inject knowledge in our genetic processing and speed up the solution space exploration. To allow faster convergence while improving the diversity of our population, we propose a novel mutation operator later in this section.

Behavior Guided Crossover Operator (BGX)

Crossover operators presented in the literature can be used to solve the cache coloring problem. However, our study shows that we can improve the crossover operation if we add domain knowledge to it. The goal of our crossover operator is to allow faster convergence to the solution. To achieve this goal we try to anticipate crossover decisions that will lead to creating a child which quality is lower than its parents' quality.

To avoid this phenomenon, we rely on the cache behavior information. For each gene of an individual, we compare it to the same gene of the other individual. We call the individuals *Indiv1* and *Indiv2*. As explained earlier, a gene i corresponds to the number of colors allocated to an applicative partition i . The following process is applied to all genes in the individual:

- If the applicative partition i has constant cache behavior, select for the new child the

smallest gene between *Invid1* and *Indiv2*.

- If the applicative partition i has sensitive cache behavior, select for the new child the greatest gene between *Invid1* and *Indiv2*.
- If the applicative partition i has stepped cache behavior, if the two genes are not located on the same step, select for the new child the greatest gene between *Invid1* and *Indiv2*. Otherwise, select the smallest gene between *Invid1* and *Indiv2*.
- If the applicative partition i has undefined cache behavior, randomly select the gene for the new child.

These decisions might lead to cases where the number of colors allocated to all partitions is either smaller or greater than the total number of colors in the cache. To solve this issue, we introduce the notion of cache equalizer later in this section. This function reduces or increases the allocated cache of partitions according to their cache behavior.

This operator only generates one child, that is supposed to be a better combination of its two parents. However, one can produce a second child, that takes all the genes not used by the first child. This would lead to a badly fitted child, but would help maintain population diversity other than by using mutation and injection processes.

Behavior Guided Mutation Operator (BGM)

Following the same reasoning, we create a new domain specific mutation operator. The objective of this new operator is to use the knowledge it is provided to improve solution space research. This is achieved by preventing inefficient mutation. One can find a multitude of mutations that will not change the quality of the solution or impact it negatively (for instance, increasing the number of colors allocated to a constant behavior applicative partition).

Thus, we prevent such occurrences to happen and allow the mutation process to find more interesting solutions. Mutation should impact the quality of the solution while trying to impact it in a good way (increasing the quality of the solution). However, one might say that this kind of method will decrease the diversity in the population and prevent exploring potentially good allocation scheme. To avoid this, we still let mutation of less efficient individuals happen with a probability computed when tuning the genetic algorithm hyper parameters.

To apply mutation to an individual, we randomly select one of its genes and apply the following modifications:

- If the applicative partition corresponding to the gene has *constant cache* behavior, the operator reduces the number of allocated colors by a random number. The cache equalizer will be in charge to redistribute the free colors to other partitions.
- If the applicative partition corresponding to the gene has *sensitive cache* behavior, the operators should check the surrounding applicative partition cache colors. If one of the surrounding partitions has a constant cache behavior, then the operator increases the number of cache colors allocated to the partition corresponding to the gene and reduces the number of colors allocated to the constant applicative partition. If none of the surrounding applicative partition is categorized as *constant*, then randomly increase or decrease the number of allocated colors.
- If the applicative partition corresponding to the gene has *stepped cache* behavior, the operator will randomly change the number of cache colors allocated to the partition. However, the number of colors will depend on the steps the partial fitness function has. If the applicative partition partial fitness function has two steps, then the operator will select the smallest number of colors contained in the other step. If the function has three steps, then the operator will randomly choose a step and allocate the minimal number of colors of this step, etc.
- If the applicative partition corresponding to the gene has *undefined cache* behavior, its number of allocated colors is randomly increased or decreased.

After the mutation process, the cache equalizer will be in charge of regulating the number of allocated colors if needed.

Cache Equalizer

During mutation or crossover operations, the total number of colors allocated to the applicative partitions can be either less or more than the available number of colors the cache actually offers. The cache equalizer has an important role in the generation of new individuals and can be seen as a part of a mutation process. When the individual has allocated fewer colors than available, the equalizer is capable of allocating the free colors in a meaningful way. To the contrary, if the total number of allocated cache colors exceeds the number of available colors, the equalizer should deallocate colors while minimizing the impact on the individual quality.

To achieve this goal, we rely again on the cache behavior applicative partitions have. Algorithm 4 shows the process defined for the equalizer. In Algorithm 4, S is a set that contains

all the applicative partitions information. This set is shuffled so when accessing it sequentially, applicative partitions are uniformly accessed. The first while loop (from line 4 to line 15) is used to reduce the number of allocated colors. Our approach is based on the fact that applicative partitions with constant cache behavior should be allocated as fewer cache colors as possible. In this first loop, we decrease the number of allocated colors to all cache constant applicative partition.

Algorithm 4 Cache equalizer procedure.

```

1: function EQUALIZER(individual, availableColorsCount)
2:   S := shuffle(individual.partitions)
3:   lastAllocatedCount := individual.allocatedColors
4:   while individual.allocatedColors > availableColorsCount do
5:     for all p ∈ S do
6:       if p has constant cache behavior then
7:         p.allocatedColors := p.allocatedColors − 1
8:         individual.allocatedColors := individual.allocatedColors − 1
9:       end if
10:    end for
11:    if lastAllocatedCount == individual.allocatedColors then
12:      break
13:    end if
14:    lastAllocatedCount := individual.allocatedColors
15:  end while
16:  x := −1
17:  if individual.allocatedColors < availableColorsCount then
18:    x := 1
19:  end if
20:  while individual.allocatedColors != availableColorsCount do
21:    for all p ∈ S do
22:      if p has not constant cache behavior then
23:        p.allocatedColors := p.allocatedColors + x
24:        individual.allocatedColors := individual.allocatedColors + x
25:      end if
26:      if individual.allocatedColors == availableColorsCount then
27:        break
28:      end if
29:    end for
30:  end while
31: end function

```

However, it is sometimes impossible to reduce the number of allocated colors enough. It happens when all cache constant applicative partition allocation reached the minimal number of allocated colors. We detect this thanks to the condition at line 11. Lines 16 to 19 allow to detect if we still need to reduce the number of allocated colors or if we should increase it (not all cache colors were allocated in the individual).

Finally, the last loop, from lines 20 to 27 is used to either increase or decrease the number of colors allocated to any applicative partition that does not have a constant cache behavior.

This choice is made since we do not want to allocate more colors to constant applicative partitions in the first case (they do not need it) and we already decreased their number of allocated colors to the minimum in the second case.

6.4.3 Solutions Clustering

The final process of our approach consists in selecting the solution to use in the final system. The output of the previous step is a set of solutions that solves the cache partitioning problem. In the case of a single-objective optimization, the solution is straight forward. The user will take the solution that minimizes or maximizes the global fitness function.

However, when following a multi-objective approach, multiple solutions can be selected. Figure 6.5.(a) shows an example of two-objective solution space outputted by the algorithm. One can clearly see what we call a Pareto front. The red line shows the solutions that maximize, minimize or offer a trade-off to the problem the user is trying to solve. In our approach, we provide this graph to the user and let them choose the solution that fits their needs.

In the presence of more than two objectives, the solution selection is more difficult for the user. Figure 6.5.(b) depicts this case, where three objectives are studied. In this case, multiple clusters of solutions are present and the user should select between these clusters.

However, how to choose between the different solutions of the same cluster? To ease the decision process, we provide the user with the clustered solutions. We first cluster the solutions, using the K-Mean clustering algorithm [127] and the average silhouette method to detect the number of clusters. Then, for each cluster, we detect which solution yields the best results regarding the objectives to optimize.

We choose to use K-Mean because it adapts well with clusters of different size and shapes. The solutions space we compute at the end of the evolutionary process might contains spread data points. Hence, we aim for a coarse clustering with a fairly low amount of parameters to ease the designers' task. Our goal is to provide a general trend of the different solutions provided by the framework.

6.5 Results

In this section we study the impact of our method on the cache partitioning problem. We first discuss our test environment and the metrics we studied to validate the method. Next, we present the accuracy of our cache behavior detection algorithm and show the improvement brought by the cache behavior knowledge added to our mutation and crossover operators. We

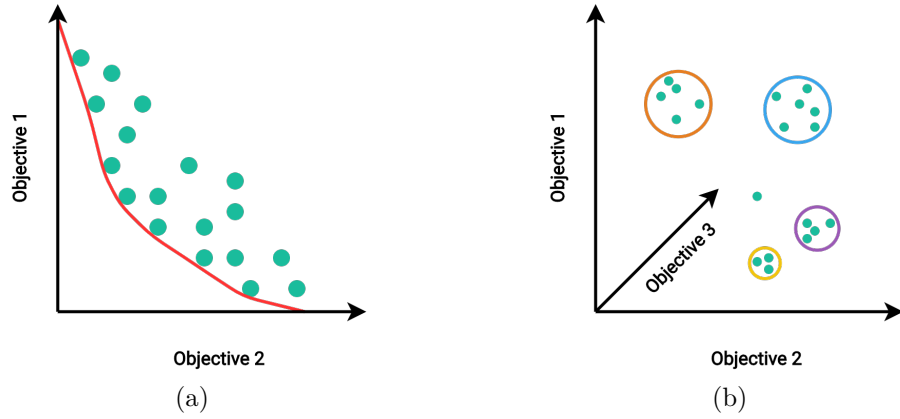


Figure 6.5 Example of a two-dimensional (a) solution space and three-dimensional solution space (b).

further proceed to explain the result obtained when coloring the cache in a real-life system. We finally bring the overhead advantages and limitations of our approach.

6.5.1 Test Environment

Our hardware system is a PowerPC architecture (P2020) composed of a Dual-core PowerPC e500v2 at 1.2GHz where each core has a private data and instruction L1 private cache of 32KB and the board embeds a shared L2 cache of 512KB, 8 ways associative. To compute the number of available colors in L2 cache, we use the formula $C = \frac{Z}{(W \times P)}$, Where C is the number of available colors, Z the total cache size, W the number of ways in the cache and P the size of a memory page in the RTOS. In our context, $Z = 512Kb$, $W = 8$, $P = 4KB$ so we conclude that the number of available colors C is 16. We rely on a partitioned ARINC-653 RTOS running on two cores using the Asymmetric Multi-Processing architecture. The RTOS manages the memory by coloring the cache through its Memory Management Unit (MMU).

The applications (applicative partitions) we use are selected from the TACLe benchmark suite [81]. We created five groups of applications by randomly picking applications from the benchmark pool. Each group is mapped to a CPU and runs concurrently with another group. We compare the number of cache misses generated when we apply equal coloring and when we rely on our method to decide the coloring scheme.

Our experiments consider three metrics. *an3* is the energy consumption of the system during the applicative partition's execution; *met2* is the number of cache evictions accounted during the applicative partition's execution; *cpu_cycles* is the execution time of the applicative partition accounted in cpu cycles. For the ease of our experiments, those metrics where

gathered experimentally, running the applicative partitions 1000 times each for *met2* and *cpu_cycles* and by simulation for the *an3*. For each applicative partition, the worst case was kept for our experiment.

6.5.2 Cache Behavior Detection

To measure the precision of the cache behavior detection algorithm, we generated and manually labeled 10,000 cache usage curves. Each curve is categorized as one of the four possible cache usage categories: constant, sensitive, stepped, undefined. After classification by the algorithm, we obtain the results gathered in Table 6.1. It has to be noted that a miss classification does not prevent our approach to succeed. The classification allows our algorithm to be more efficient and converge to the solution in a fewer number of iterations. The randomness and non-elitist processes used by the genetic algorithm allow finding solutions even in the case of miss classified cache usage. The reduced precision to detect undefined behavior

Table 6.1 Cache behavior classification algorithm precision, recall and F score.

Class	Precision	Recall	F score
Constant	0.99	0.88	0.94
Sensitive	0.97	0.99	0.98
Stepped	0.99	0.99	0.99
Undefined	0.89	0.97	0.92

is due to the proximity of certain undefined cache curves with other potential curves. Some undefined curves may be close to stepped functions.

The overhead of the cache behavior detection is negligible. This process only takes place during the algorithm's initialization and uses fast and well known methods to perform the detection. Those methods are presented in Section 4.

6.5.3 Behavior Guided Operators

We chose to compare the proposed mutation operator (BGM) shown in Figure 6.6.(c) with well-known operators from the literature:

- Balance mutation in Figure 6.6.(a) which takes the sum of the size allocated to two adjacent cache partitions and randomly redistributes it between the two
- Partial Shuffle Mutation (PSM) in Figure 6.6.(b) which shuffles the gene values in the individual

- Centre Inverse Mutation (CIM) in Figure 6.6.(d) which cuts the individual in two parts. Each part is then inverted ([1,3,4] becomes [4,3,1])
- TWORS mutation in Figure 6.6.(e) takes two genes of this individual and exchanges their position
- The mutation operator proposed in [2]. We modified the operator to forbid assignation of partitions to shared cache colors.

Results show the impact of the choice of a mutation operator on the solution space exploration of an individual over 100 successive mutations. Two interesting conclusions emerge from this experiment. First, gene swapping mutation operators are not effective to solve the cache partitioning problem as we defined it. This comes from the fact that swapping two genes between application without changing the global cache allocation layout might not impact much the solution quality in our context. In the case of multiple constant behavior applicative partitions, exchanging two genes between two constant applicative partitions will not impact the solution's quality.

Second, we show that our mutation operator (BGM) allows a wider exploration of the solution space. As explained in the previous section, this is achieved thanks to the knowledge we add to the operation. For instance, the operator will avoid allocating more cache to the constant application (we try to keep the solution's quality as high as possible). We further provide a comparison between popular crossover operators and our proposed operator. We choose to implement the following operators:

- The average crossover takes two individuals and, for each gene, computes the average of their size (allele) and creates the corresponding gene for the child
- The random crossover randomly selects for each gene, the allele of either the parent 1 or parent 2 and gives it to the child 1. Child 2 will be set with the other allele that have not been selected during the random selection process
- The one point crossover operator selects a random gene's index and transmits all the genes before the index from parent 1 to the child 1 and the genes after the index from parent 1 to the child 2. The inverse is done from parent 2 to the two children
- The two points crossover operator applies the same process as the one point operator but two indexes are defined. The child 1 will receive genes before the first index and after the second index from the parent 1 and genes between the two indices from parent 2. The inverse is done for the second child

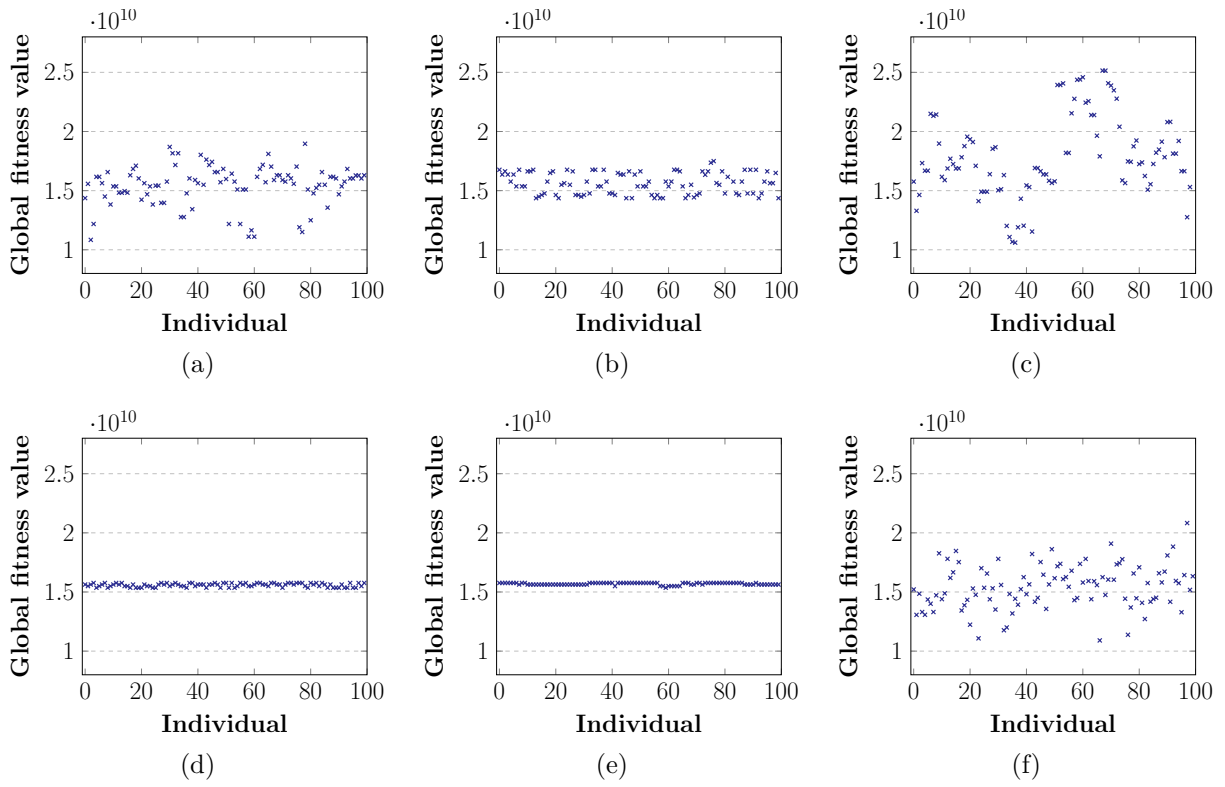


Figure 6.6 Global fitness function value over 100 successive mutations with balance (a), PSM (b), BGM (c), CIM (d), TWORS (e) and the operator proposed in [2] (f).

- The crossover operator proposed in [2]. We modified the operator to forbid assignment of partitions to shared cache colors.

Results provided in Figure 6.7.(a) show the global fitness function's evolution over 1000 iterations of the algorithm. The curves show the ability of the BGX operator to find better solutions in less time. This is the improvement attained by injecting knowledge in the operator process. The crossover allows the generation of better solution by avoiding worse solutions to be created. It allows faster convergence by preventing the generation of individuals which quality is similar to the current ones.

6.5.4 Cache Partitioning Results

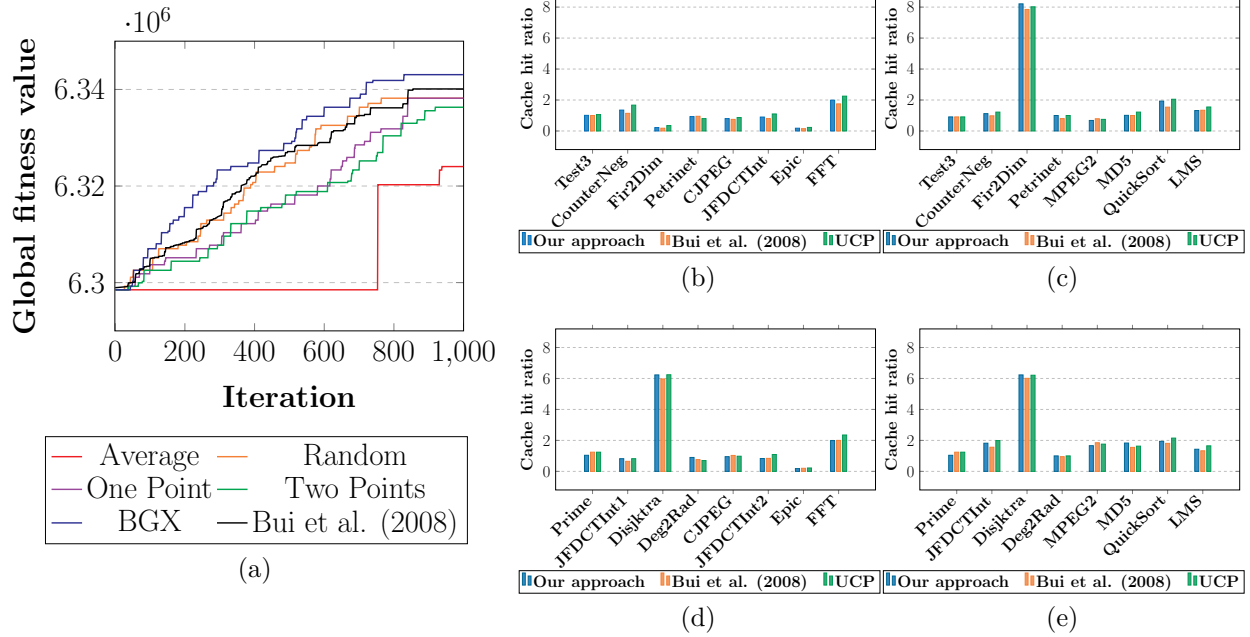


Figure 6.7 (a) Global fitness function value over 1000 iterations of the genetic process. Six different crossover operators were studied, showing the proposed operator (BGX) performs better both in convergence time and quality of the solution. (b-e) Intelligent cache partitioning applied to improve global worst-case cache hit/miss ratio.

Our experiment involved multiple groups of applications to have a more realistic view of what our method could achieve. Indeed, real-life case study should contain memory intensive and non memory intensive applications. Figure 6.7.(b-e) shows that depending on the co-scheduling groups, the optimization adapts to provide the best results. Our method provides an increase of the global worst-case cache hit/miss ratio of up to 3.25 times compared to

equal coloring. It also improves the worst-case hit/miss ratio of single applications of about 1.8 times on average.

We compared our approach to the state of the art algorithm: the UCP algorithm proposed in [16]. We also compare our result with the genetic algorithm approach proposed in [2]. We compared both the performance resulting from the decision of the algorithms in Figure 6.8.(b) and the execution time to find the matching solution in Figure 6.8.(a). Figure 6.8 shows the result obtained by UCP after completing its execution, while results for the approach proposed in [2] and our approach correspond to the results obtained after 1000 iterations of the algorithm. After this number of iteration, the genetic algorithm stabilizes.

Results show that our approach performs better than [2] thanks to the better guidance in the genetic algorithm. This guidance is provided by the operator we introduced in Section 4. We show that our approach performs nearly as good as the UCP algorithm that will find the optimal solution.

The advantage of our algorithm over the UCP and the approach of [2] is the execution time. Indeed, the UCP algorithm does not scale well with the increasing number of applications in the system. Figure 6.8 shows the execution time of the UCP algorithm compared with [2] and our approach. As [2] and our approach can be stopped at any time, we decide to stop the solution exploration when reaching at least 95% of performance improvement compared to UCP.

For this experiment, we only directed the algorithm towards a global reduction of caches misses. This led to excellent result system-wise, but highly impacted some application's cache hit/miss ratio. Indeed, in Figure 6.7.(c), Epic sees its ratio reduced by 5 times. In Figure 6.7.(a) Fir2Dim is impacted the same way and has its ratio reduced by 4 times. This comes from the fact that these applications do not access the cache a lot. However, FFT in Figure 6.7.(a) accesses a lot more compared to these two applications. Hence, it is the application that affects the most the system in term of global hit/miss ratio. That is why it is allocated the most memory.

Multi-objective approach

In Figure 6.8.(b), we show the use of the multi-objective approach to leverage the unfairness in the cache coloring scheme. The system's global worst-case hit/miss ratio has a lower improvement but applications see their worst-case cache use increased on average.

To achieve these results we base our objective function on two objectives: the sum of the applicative partitions execution time and the maximal gap between the execution time of

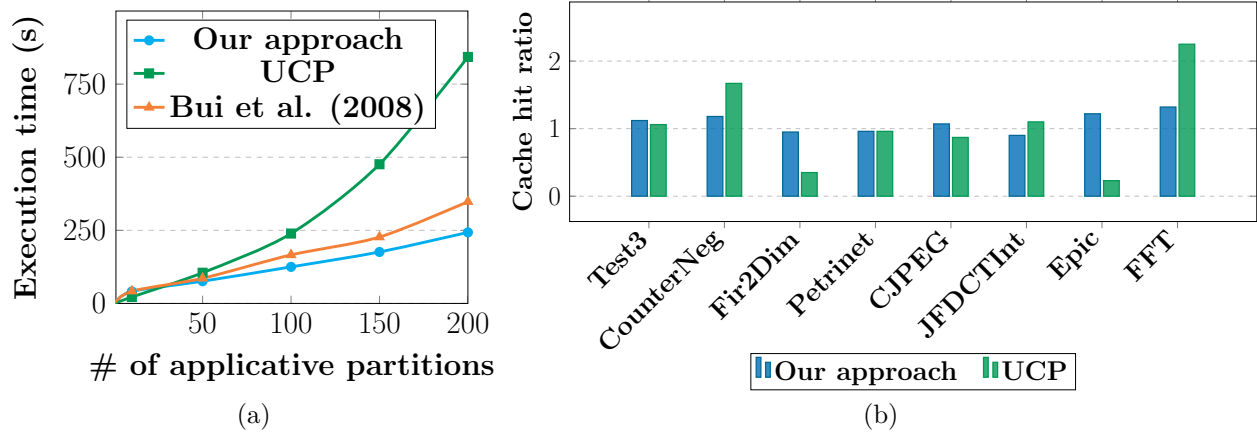


Figure 6.8 (a) Execution time of the UCP algorithm compared with [2] and our approach. As [2] and our approach can be stopped at any time, we decided to stop the solution exploration when reaching at least 95% what UCP offers in terms of performance improvement. (b) Cache partitioning applied to improve global cache hit ratio while keeping fairness as second objective. We compare our results with the UCP algorithm solution.

an applicative partition before and after cache partitioning. Both objectives should be minimized. Equation 6.4 provides the equation we used. In our experiments, both objectives have the same weight.

$$\min_{\forall c} \left\{ \sum_{i=0}^N eT(i, c) + \max_{i=0}^N \{ \|eT(i, INIT) - eT(i, c)\| \} \right\} \quad (6.4)$$

Where $eT(i, c)$ is the normalized execution time of applicative partition i with the cache configuration c . $INIT$ is the initial cache configuration, where $1/N$ area of the cache is allocated to each of the N applicative partitions.

We also consider two multi-objective functions. Equation 6.5 and Equation 6.6 to conduct our experiments on multi-objective.

$$\min_{\forall c} \left\{ \sum_{i=0}^N eT(i, c) + \sum_{i=0}^N eC(i, c) \right\} \quad (6.5)$$

$$\min_{\forall c} \left\{ \sum_{i=0}^N eT(i, c) + \sum_{i=0}^N eC(i, c) + \sum_{i=0}^N eM(i, c) \right\} \quad (6.6)$$

Where $eT(i, c)$, $eC(i, c)$ and $eM(i, c)$ are the execution time, the energy consumption and the number of cache eviction of the applicative partition i with the cache configuration c and the energy consumption with the same settings.

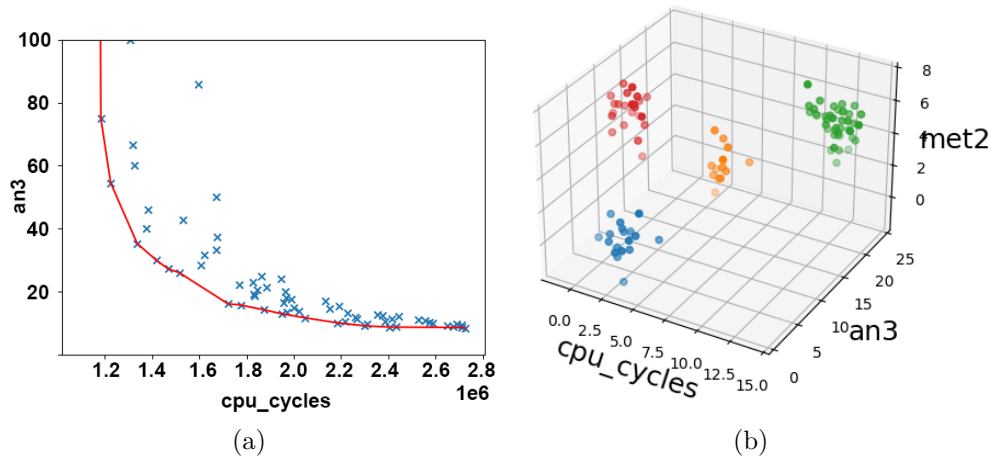


Figure 6.9 Example of output for two (a) and three (b) different objectives. *cpu_cycles* refers to the execution time in CPU cycles, *an3* to the energy consumption of the system and *met2* to the number of cache eviction during execution.

To obtain the correct configuration, we rely on the framework and the data clustering module proposed in this article. The rational behind the integration of this module is to simplify the use of this tool for the system integrator. Thanks to the visualization and the clustering tool, the SI only needs to know which criteria is optimized and the tool allows to guide the SI when selecting the required trade off. Figure 6.9 shows the interface used to cluster solutions. In this example, two and three objectives are considered.

This is why we introduced a multi-objective approach, where cache fairness can be coupled with global cache hit/miss ratio to reduce the occurrence of the solutions.

Comparison against adversarial application

Adversarial applications are developed to expose interferences. They allow to stress shared components and provide a mean of asserting the impact of interference mitigation solutions. We compared three cache configurations in a system with two applicative partitions. The first core runs the applicative partition (Dijkstra) and the second core runs an adversarial application. The adversarial application has for goal to trash the cache by evicting any data present in it. Our selected application creates an array of the size of the cache plus one way. This ensures to evict others applications' data. Once created, the array is indefinitely accessed in a random manner. To increase the amount of interferences, we also introduced random sleeping times in the application.

Figure 6.10 show the results obtained with this test environment. For each cache configuration

the test is run 1000 times. In gray, the cache is not partitioned, in red, each applicative partition is allocated the same amount of cache and in green, we apply our cache coloring optimization method to allocate cache to both applicative partitions. The results show the presence of interferences when caches are free. The execution times are impossible to predict. Baseline coloring provides cache interference mitigation but increases the average execution time. Our method also mitigates cache interferences but clearly reduces the average execution time under the average execution time provided by free caches.

Figure 6.10 also depicts an interesting phenomenon. When caches are partitioned, a clear pattern appears. Four distinct values for the execution time can be seen. This pattern is due to additional underlying interferences the adversarial application itself creates. To generate interferences, the adversarial application has four different memory intensity modes. Those modes will contend more or less on the bus, DRAM and additional shared component, revealing other types of interferences and generating the patterns depicted in Figure 6.10.

6.5.5 Overhead and limitations

In terms of overhead, our solution can be studied on two axes. First, the implementation overhead happens at boot time. Indeed, most OSes already use paging mechanism, which is used to color caches. Thus, to make cache coloring possible in the system, one has to modify the page and frame allocator. At boot time, the page tables are created, which increased the boot time by 10%.

The second type of overhead is execution overhead. In critical systems, memory is allocated at the beginning of the system's execution and cannot be freed. Thus, there is no recoloring overhead at run-time. The execution time is not impacted because of memory management since memory translation is already something happening in systems without cache coloring. Finally, cache coloring, as we implemented it, constraints the use of fixed page size of 4Kb. This constraint increases the number of TLB misses that will occur in the system compared to the use of bigger pages (32KB, 4MB, etc.). However, this overhead increases memory management time by only 0.5%. To measure the increase of execution time, we executed the previously presented tests and gathered the number of memory related events (TLB miss, page fault, etc.) and compared it to the same test bench without cache coloring feature.

6.6 Conclusion

Partitioned systems have been used to provide safety and quality of service in highly critical systems. However, most of these systems needing certification still rely on single-core

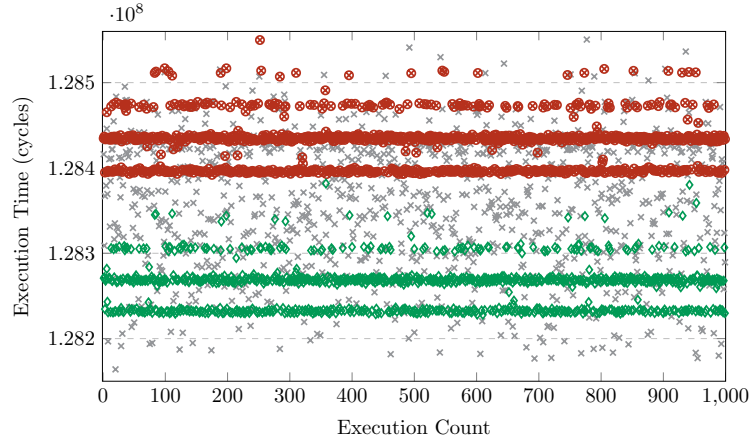


Figure 6.10 Execution time of an applicative partition over 1000 executions. In gray (crosses) when caches are fully free, in red (circles) when we apply baseline coloring and in green (diamonds) when we apply our method to select colors size.

processor. The popularization and improvements of multi-core processors force those critical domains to update their architectures to these new processors. Multi-core platforms bring an increase of computational power at the cost of unpredictable timing behavior called interferences. Such interferences occur when a resource is shared between multiple cores. When different partitions run on different cores, the partition principle cannot be respected as two partitions sharing the same resources are able to impact each other's execution.

In this article we proposed a way to mitigate interferences in shared caches. The current literature mainly relies on cache partitioning to improve cache performance. In our context, we ensure isolation of shared caches between partitions in the system while we improve execution performance. To do so, we rely on a multi-objective solution approach to optimize different aspects of the system. Our evolutionary method converges while it yields better solutions after a fixed number of iterations.

The proposed approach has been implemented on real-life systems to validate its feasibility in the industrial domain. The resulting method is compatible with certification processes in the aerospace industry and can be applied in many other domains that require software qualification and interference mitigation.

CHAPTER 7 ARTICLE 4: EFFICIENT SCHEDULING, MAPPING AND MEMORY BANDWIDTH ALLOCATION FOR SAFETY-CRITICAL SYSTEMS

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Aymen Ben-Salem, Serge Harnois, Felipe
Gohring de Magalhaes and Gabriela Nicolescu

Published in 20th IEEE Interregional NEWCAS Conference. Publication date: 2022/08/22

Abstract: In safety-critical multi-core architectures, isolation becomes more challenging as multiple components are shared between the cores, which generates delays in execution time called interferences. To reduce buses contention, it is important to limit the bandwidth allocated to each application. Current state-of-the-art solutions for bus budgeting allocation are not scalable and do not provide the correct level of flexibility to model complex systems. To cope with these challenges, we propose a novel heuristic to generate the schedule, mapping and bus budget allocation. We achieve scalable execution times up to 45 times faster compared to the state of the art. We further introduce real-life constraints such as application dependencies, forced mapping and release time. Compared to the state of the art, our approach is applicable in complex systems.

7.1 Introduction

Safety critical systems usually rely on partitioned Real-Time Operating Systems (RTOS) [6] to ensure that applications of different criticality do not impact with each other. Applications, called partitions, are scheduled in a static scheme called the MAJor Time Frame (MAF) defined at design time [11]. An example of MAF could be the following pattern $P1 \rightarrow P3 \rightarrow P2 \rightarrow IDLE \rightarrow P1$. In a multi-core architecture, each core has a specific MAF and is synchronized.

Despite the advantages brought by multi-core architectures (increase in performance, better energy efficiency, etc.), having multiple execution threads concurring for resources creates contention [38] (called interference). This contention, also happening in shared buses, requires arbitration which introduces delays in partitions' execution time which prevent the correct timing analysis and breaks isolation principles.

To bound this contention, we rely on the memory bandwidth management proposed in [39] to monitor the number of transactions made by partitions on the bus. For a certain regulation

period (different than the MAF), a partition is allocated a budget in number of accesses. If the budget is exceeded, the partition is stopped and is released at the beginning of the next regulation period. If the sum of the budgets allocated to partitions at a certain time is at most 100%, we ensure that all partitions can perform the transactions they are provided.

The state-of-the-art solutions in terms of bus budget allocation lack of real-life constraints modelization and do not scale well with complex systems. In this paper, we present a model that is applicable to real-life systems with real-time constraints such as applications dependencies.

Our contributions can be summarized as follows: **(1)** a novel method capable of mapping, scheduling and allocating bus budgets to partitions while scaling with the size of the system; **(2)** We introduce constraints on partitions required by real-life applications (dependencies, mapping and release time); **(3)** We overcome previous work limitations by allowing partitions to have different periods and cores to have different scheduling window lengths.

7.2 Related work

In [41, 42, 47], co-scheduling partitions to reduce bus contention is explored. In [50, 52], the Acquisition, Execution, Restitution (AER) model is used to ensure that only one core uses the bus at a time by limiting the number of tasks in the acquisition and restitution phases. In [43], a scheduling approach enforces the system to run only one core when a critical partition executes.

Virtualization is also explored in [45] where shared resources are distributed across virtual CPUs. The method allocates shared caches and bus bandwidth to ensure isolation of the memory hierarchy. Works that rely on additional hardware were proposed in [55–58]. Using special bus arbiter and memory controllers allows to precisely manage interferences.

In [39], the authors propose MemGuard, a bus bandwidth management method to ensure bus reservations for a given core. Their work relies on Performance Monitoring Counters (PMC) to monitor partitions' bus usage and throttle the core when the partition's budget is exceeded. In [48], an Integer Linear Programming (ILP) approach is proposed to compute the budget allocated to partitions. ILP models are used to optimize objective function based on mathematical constraints. In this paper, we address the scalability of the ILP-based approach by proposing an optimization heuristic. We further extend the model to fit real-life systems by adding dependency, release time and mapping constraints. Finally, we remove the aforementioned work's limitations with different partitions' periods and the use of uneven MAF's length among cores.

7.3 SMA heuristic

In this section, we present SMA: our approach for Scheduling, Mapping and Allocating bus bandwidth in the system. For the sake of clarity, we use the same notations as in [48].

7.3.1 System definition

The MAF are expressed by the variable Q_i where i is the core identifier. Each MAF is divided in time quanta q of length R that designates the regulation period. The number of quanta for each MAF is given by $\frac{Q_i}{R}$. The length of R is discussed in [39] and is out of the scope of this work.

The system is composed of M partitions P_k mapped on N cores. Each partition has a set of tuples (K, X) where K is a bus bandwidth budget and X is the time allocated to the partition in the MAF. X is defined in function of K since the allocated time of a partition should vary depending on the budget it is allocated. Each partition is characterized by a tuple $\langle T_k, A_k, N_k, (K_k; X_k) \rangle$. T_k is the partition's period. A_k is the release time, a phase applied to the first execution of P_k in the MAF. N_k is the set of core identifiers on which P_k can mapped. Due to the criticality of some partitioned systems, partitions can only be mapped to one core and cannot migrate. Finally, $(K_k; X_k)$ is the couple of bus budget and allocated time for the partition.

We define the hyper-period H that is the least common multiple of all the cores' MAF. It represents the global schedule that constitutes the schedule pattern repeated during the system's execution. If no MAF's length is specified, we compute the hyper-period as the least common multiple of all partitions' periods. In that case, for each core, $Q_i = H$.

We further introduce the three following scheduling and mapping constraints:

Dependency constraint: $A_j \geq A_i + X_i$. If P_j depends on P_i , then P_i must execute at least once before P_j . Hence, A_j must be greater than or equal to the sum of A_i and X_i . Dependency between partitions is written as $i \in Dep(j)$.

Release time constraint: $A_k = cst$. A partition's release time can be constrained to a specific quantum cst . This is useful if a partition handles a specific event that occurs at a given time in the MAF or hyper-period.

Mapping constraint: $N_k = \{1, 3, \dots\}$. Mapping can be constrained to a set of cores (for instance, when partitions of the same type should be grouped). If N_k is empty, our method can assign P_k to any core during the optimization step.

7.3.2 Evolutionary approach

We define the solution space as the set of possible combinations of bus bandwidth budget (K_k) assigned to each partition. A solution is an array of budgets, where the k^{th} element is the bus budget allocated to P_k . Hence, our heuristic needs to find the combination of (K_k, X_k), N_k and A_k for each partition P_k that make the system schedulable. To guide the exploration, we define a score which depends on the feasibility of the solution:

- If the solution is not schedulable, the score is defined as

$$-\sum_{k=0}^M X_k \times (1 - SCHED(k))$$

where $SCHED(k)$ is 1 if P_k was scheduled and 0 otherwise.

- If the solution is schedulable, the score is defined as

$$\sum_{i=0}^N Q_i - \sum_{k=0}^M X_k$$

The first score is the amount of time represented by partitions that were not scheduled while the second score represents the amount of free time remaining on all cores. With this objective, we optimize the amount of free time the system has. We use hill-climbing to explore the search space. For each iteration, we compute the neighbors of the current solution. At the end of each iteration, we keep the solution that yields the best score. If no improvement was made during the current iteration and no solution has been found, the current solution is randomized and the process continues.

The neighbors of a solution are obtained by adding and subtracting a certain value *Inc* (increment) to the bus budgets. To obtain the i^{th} neighbor, if i is even, then we add *Inc* to the budget allocated to P_i . If i is odd, then we subtract *Inc* to the budget allocated to P_{i-1} . The neighbor is discarded if the budget allocated to a partition is greater than 100 or lesser than the minimal budget required by a partition (defined by the user). *Inc* is set by the user depending on the granularity they want to explore the solution space. A smaller *Inc* entails a slower search but increases the chance of finding a solution while a bigger *Inc* will have the contrary effect.

7.3.3 Mapping and Scheduling

To avoid the computation time entailed by the scheduling and mapping heuristic, we remove solutions that do not meet preliminary conditions to be scheduled (Equation 7.3). We create a DAG (Directed Acyclic Graph) where dependent partitions are linked. Partitions with no dependency will be seen as non-connected nodes in the graph. For each sub-graph we

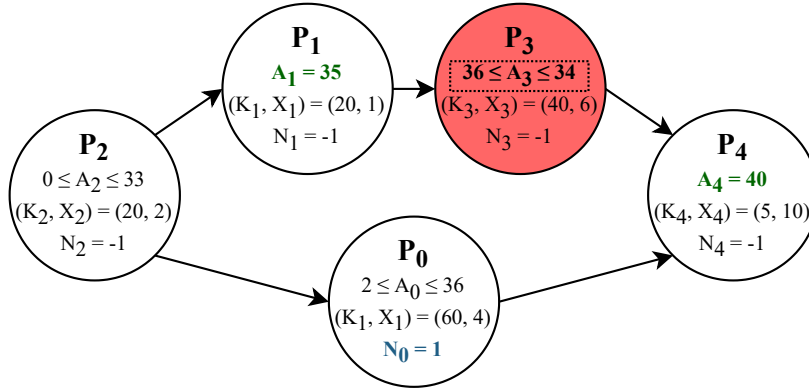


Figure 7.1 Dependency graph after validation. P_0 , P_1 and P_4 have a mapping and release constraints and P_3 is a conflicting partition.

compute the minimal ($\min(A_k)$) and maximal ($\max(A_k)$) release time of each partition. If A_k is constrained then $\min(A_k) = \max(A_k) = A_k$. Iteratively, starting from the leaves, we compute $\max(A_k)$ using Equation 7.1. For the leaves, no partition is dependent on P_k , $\max(A_k) = T_k - X_k$. Then, starting from the roots of the graph, we compute $\min(A_k)$ using Equation 7.2. Since roots do not depend on any partition, $\min(A_k) = 0$.

$$\max(A_k) = \min_{\forall j \text{ if } k \in \text{Dep}(j)} \{\max(A_j)\} - X_k \quad (7.1)$$

$$\min(A_k) = \max_{\forall j \in \text{Dep}(k)} \{\min(A_j) + X_j\} \quad (7.2)$$

Equation 7.3 is applied on each node of the sub-graph. It validates that P_k is scheduled, does not overlap any other partitions and has enough time to execute.

$$0 \leq \min(A_k) \leq \max(A_k) \leq T_k - X_k \quad (7.3)$$

Fig. 7.1 shows a dependency graph. In red are the conflicting partitions. This step allows propagating partitions' maximal and minimal release time. After propagation, an overlap occurs for a node if $\min(A_k) > \max(A_k)$ (P_3 in our example).

We find a scheduling and mapping scheme by relying on a first first bin packing algorithm that verifies that the sum of all partition's budgets less or equal to 100% at any time. We use the notion of timeline and time slots shown in Fig. 7.2 to reduce the computation time. A time slot is a contiguous span of time that is either used by a partition or free. A slot is defined as $s(t_s, t_e)$ where t_s is the first quantum and t_e the last quantum. A timeline is a sequence of multiple time slots. Times slots are coarser than quantum for faster validation of the scheme.

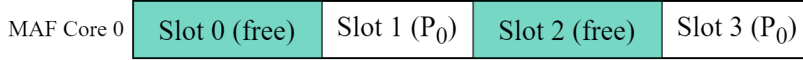


Figure 7.2 Timeline with four slots in, two of which are free.

7.4 Experiments

To compare our approach, we need to extend the ILP model proposed in [48] (we refer to the state-of-the-art method proposed as *SOTA*). For ease of reading, we reuse the notation proposed in the article:

- c_k is the core identifier on which P_k is mapped.
- f_{ki} equals 1 if i is the first quanta of P_k in the MAF.
- l_{ki} equals 1 if i is the last quanta of P_k in the MAF.
- F_k is the first quanta when P_k is scheduled (Equation 7.4).
- L_k is the last quanta when P_k is scheduled (Equation 7.4).

$$F_k = \sum_{i=0}^Q f_{ki} * i \quad L_k = \sum_{i=0}^Q l_{ki} * i \quad (7.4)$$

Dependency	$\forall P_i, P_j \in M \mid i \in Dep(j) = 1 \Rightarrow L_i < F_j$
Release Time	$\forall P_k \in M \mid A_k \neq -1 \Rightarrow F_k = A_k$
Mapping	$\forall P_k \in M \mid N_k \neq \emptyset \Rightarrow c_i \in N_k$

7.4.1 Dataset generation

Our test variables are presented in Table 7.1. The *average system's memory intensity* represents the percentage of execution time partitions use for memory transactions. We rely on a real-world scenario in the avionic domain where the number of partitions per core is

Table 7.1 Test variable used to in our test bench.

Name	Description	Values
n	Number of partitions in the system	2, 4, 8, 16, 32, 64
c	Number of cores in the system	2, 4, 8, 16
μ	Average system's memory intensity	10%, 20%, 30%, 50%
Q	MAF length (in number of quantum)	[25; 300] by step of 25
Inc	Evolutionary step size	5

usually between 1 and 4. The memory intensities chosen in our setup are usual values we encountered while studying avionic systems.

For each experiment, we execute our algorithm on 100 partitions sets generated by SimSo [128]. Partitions have random allocated execution time (X_k) when 100% of the bus budget (K_k) is allocated. The following process is applied: (1) Based on μ , randomly choose a memory intensity value for each partition. The average of each partition's memory intensity is equal to $\mu(\pm 1\%)$; (2) Based on a stride of 5% we compute each partition's (K_k, X_k) couple for $K_k \in [5, 100]$. (3) A set of constraints is generated with dependency constraints (between 2 and 40 partitions), mapping constraints (between 1 and 20 partitions) and release time constraints (between 1 and 10 partitions). Tests are executed on Intel Core I7-9700K 3.6GHz CPU and 32GB of DDR4 DRAM at 3200MHz using the Gurobi Optimizer [129] to implement the ILP model.

7.4.2 Comparison with the state of the art

As SOTA only allows same core's MAF length and partitions' period restricted to be equal to the unique MAF's length, we compare our approach with the same limitation. Table 7.2 shows the execution time (in seconds) of both SOTA and SMA. SMA provides faster results on average and is 9.64 times faster for 2-cores systems and 42.26 times faster for 4-cores systems. SOTA approach scales nonlinearly with the number of cores and partitions while SMA experimentally shows a quasi-linear increase in execution time.

Depending on the system the regulation period can be as small as 1 ms or even smaller. In such system it is also possible to have a MAF that is around 200 ms. Which makes a MAF containing at least 200 regulation periods. Hence, it is important to consider scalability regarding the MAF length.

Fig. 7.3 shows the comparison between the execution time of SMA and SOTA. The experi-

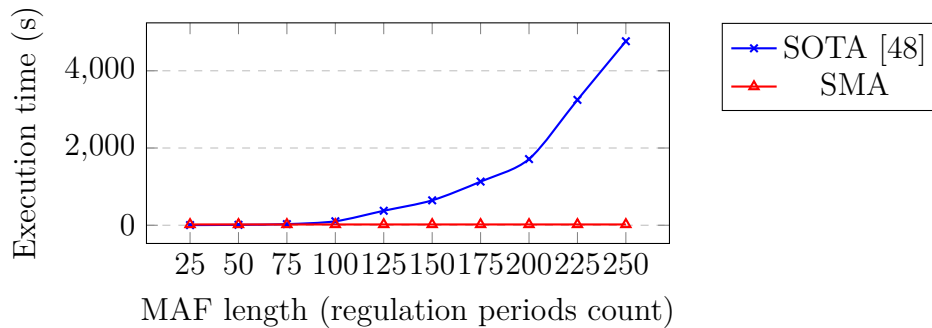


Figure 7.3 Comparison of the time elapsed to find a solution depending on the MAF length.

Table 7.2 Execution times of SMA and SOTA (in seconds)

$\mu(\%)$	2 Cores ($c = 2$)				4 Cores ($c = 4$)			
	SOTA		SMA		SOTA		SMA	
	4P	8P	4P	8P	8P	16P	8P	16P
10	1.3	51.2	2.5	5.7	1645.4	1611.0	61.7	155.1
20	17.5	179.3	2.6	6.53	1727.5	1721.9	70.4	357.3
30	20.9	1182.3	33.8	68.3	2047.8	2367.9	78.1	452.0
50	137.5	2409.5	40.5	314.1	4962.7	5409.5	85.7	528.5

ment was conducted on 100 datasets containing 16 partitions each and with a 4 cores system. Results show that our approach is slightly impacted by the MAF length, contrary to SOTA. This is because we consider the MAF as a set of time slots.

We provide the schedulability analysis in Table 7.3. SMA achieves nearly the same performances as SOTA, with a slight decrease in the schedulability rate as memory intensity increases. Contrary to SOTA, in the advent of an unschedulable system, our method proposes a partial schedule. Table 7.3 shows that in this case, our approach allows to schedule between 69% and 88% of the partitions. We also provide the system integrator with the reasons (deadline miss, budget overflow, etc.) of the non-schedulability.

7.4.3 Problem Scaling

The main goal of the SMA approach was to offer scalability regarding the number of cores and partitions in the system. Fig. 7.4 shows the results of our experiments on 2, 4, 8, 16 and 32 cores systems. For each system, 50 datasets containing from 4 to 128 partitions were studied. Results show that SMA can schedule those systems and scales linearly with the number of cores and partitions. In this section, we do not compare SMA with SOTA as SOTA took more than 1 hour on average to find solutions to the proposed dataset for more than 4 cores.

Table 7.3 Schedulability results between SOTA and SMA ($c = 4$).

μ (%)	SOTA		SMA		Partial solutions	
	8P	16P	8P	16P	8P	16P
10	91%	91.3%	91%	91.1%	7	14
20	86.3%	86.8%	84.3%	84.5%	7	13
30	79%	78.8%	75.3%	75.1%	6	13
50	73.1%	73.2%	70%	69.7%	6	11

We also study the impact of uneven partitions' period distribution on the optimization time. We reuse the same environment presented for the scalability experiment. We introduce a new variable that represents the number of times a partition will be executed in the MAF. We base our study on a system with 4 cores, an average system's memory intensity of 20%, a MAF length for all the cores of 200 quanta, a number of partitions varying between 4, 16 and 32 and a number of times partitions can repeat in the MAF between 1 and 10 times.

Fig. 7.5 shows that increasing the average number of times partitions can repeat from 1 to 10 multiply the execution time by 2.35. Increasing the number of times partitions can execute increases the number of slots to validate in the partition placement algorithm. Hence, we linearly increase the number of quanta to check in the hyper-period and the cores' MAF.

7.4.4 Real-life application

Our method has been validated against a real-life scenario presented in [130] where a flight controller is proposed. We consider each task as partitions and use the same WCET, periods and constraints (dependency and mapping). Execution measurements (as used by the authors) allowed us to compute the set of (K_k, X_k) for each partition. Our final system is composed of 11 partitions (tasks described in [130]). We vary the number of cores from 2 to 8. We further experiment two mapping policies proposed: ① partitions can be placed on any cores; ② we group partitions with the same period on the same core (when using 2 cores, we group 50Hz and 200Hz partitions together). As presented in Fig. 7.6, our method finds a solution with any of the proposed setup and in an acceptable time. We ran the optimization 1000 times and report the average execution time. Since the number of partitions does not increase, when the number of cores increases, it is easier to find a scheduling solution when the mapping is not constrained. Intuitively, when the mapping is constrained, adding cores does not ease the optimization. This case study proved to allow an easy modeling of the ROSACE system and successfully applied our approach on a real-life scenario.

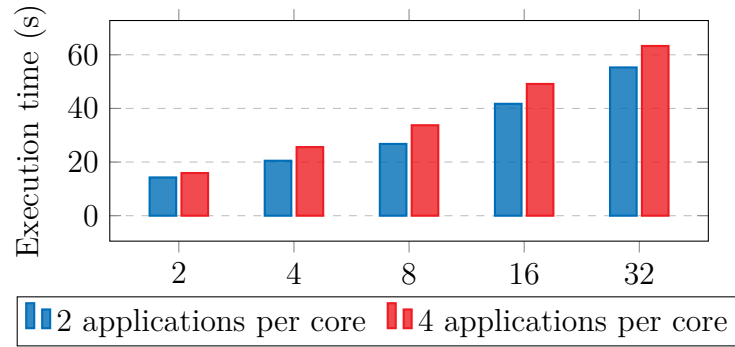


Figure 7.4 SMA solving time for different system configurations.

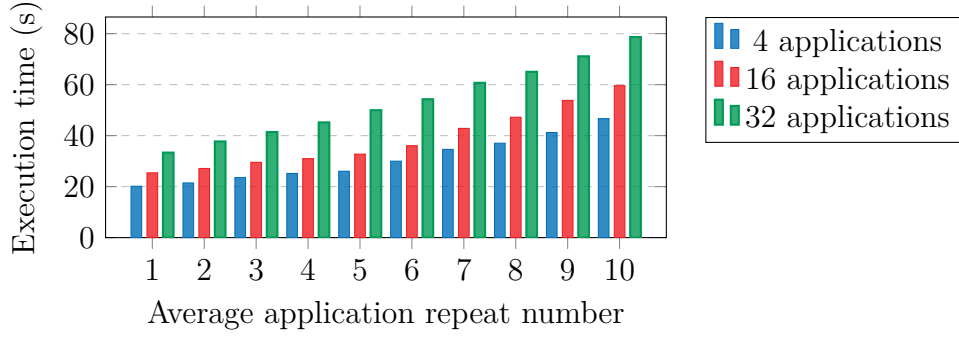


Figure 7.5 SMA solving time based on the number of times partitions can repeat in the MAF.

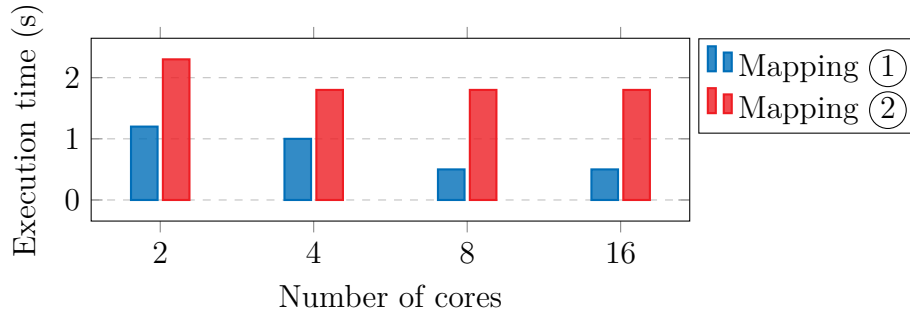


Figure 7.6 SMA solving time for the ROSACE system.

7.5 Conclusion

In this paper, we presented the Scheduling Mapping and Allocation Heuristic. This method helps the system integrator by providing a bandwidth allocation scheme as well as a scheduling and mapping configuration for partitioning systems. Our SMA heuristic is up to 42 times faster than the method proposed in the state of the art. Our method extends the current state of the art by adding indispensable configuration (constraints) and scales well with the number of partitions and cores present in the system. We applied our approach to the aerospace domain where the transition from single-core to multi-core architecture requires mitigation methods to ensure robust partitioning.

CHAPTER 8 SUMMARY OF WORKS

The transition to multicore architectures in safety-critical systems unveils many challenges due to the contention on shared resources. In the previous chapters, we presented the notions of interferences and the different interference channels present in multicore systems. This manuscript focuses on few specific interference channels, however, many more exists. For instance, peripherals themselves, DMA engines and other IO components also suffer from interferences.

Mitigating interference is a complex process as each interference channel can impact the others. This dependency between the channels might prevent the use of common mitigation that were studied by isolating one interference channel and then mitigating it. The integration of the different methods to lessen, bound or remove interferences is then essential. The configuration of the mitigation methods (i.e cache allocation, bus budgets breakdown, etc.) is a major challenge and has a significant impact on the system's performances as well as the interferences impact. A wrongly configured cache allocation can reduce the caches' performance, increase the execution time and reduce the determinism of the system. Moreover, it can increase the bus contention and interference.

The work we proposed was motivated by the previously described challenges. We divided our approach in three axis. The first one was to provide a configurable set of interference mitigation methods and safety net constraints to ensure the correct behavior of the system. To this end, we provided novel modules to an industrial RTOS that allow to control memory accesses, cache usage and interrupt handling. The second axis was designed alongside the mitigation methods and ensures the well-behaving of the system at run time. We defined the Safety Net, a set of constraints that are monitored at key points of the RTOS. The thirds axis of research leaned towards the optimization of the mitigation modules. We brought new methods to compute a nearly-optimal cache partitioning scheme and bus budgeting in a system that was ready to be deployed.

Based on these three categories, the results we gathered during this research are the following:

1. Interference mitigation by either removing or bounding them, making them quantifiable and analyzable.
 - We proposed an interference-aware memory manager capable of removing shared cache interference by strictly partitioning the cache. The memory manager is portable on COTS hardware and was successfully applied to a commercial RTOS.

Compared to the state of the art, the memory allocator provides faster execution times by 24.8%. The memory manager monitors the bus usage and regulates it based on a per-application budgeting approach. Its overhead is comparable to the state of the art while providing finer grain budget allocation and execution modes, allowing a more flexible execution and configuration. The integration of the two interference mitigation methods allows us to increase predictability of the applications' execution time by 68.1% on average and scales with the number of application and cores present in the system.

- An interrupt manager was proposed to bound the interrupt-related interferences. The manager adds an interrupt filter capable of canceling or deferring interrupt, which provides a strict bound on the number of handled and serviced interrupts. We further rely on cache configuration to prevent the filter from generating interference. On average, the predictability of the system is 84.1% closer to a single-core execution than when interrupts are not bounded and the measure WCET of applications reduced by 5.9%. The overhead introduced by our method ranges from 0.14% and 1.01% of execution time increase.

2. Run-time interference monitoring and RTOS health notification.

- For each interference we studied in this work, we provided a set of formally defined constraints name Safety Net. Those constraints represent properties that should be met at any time during the system's execution. We implemented the Safety Net relying on monitors executing at different point of the RTOS such as the page fault handler, the partition switch, etc. The Safety Net detects when a constraint is not respected and notifies the system to take actions. Based on our experiments, the addition of the monitors added as few as 0.5% of execution time on average.

3. System integrator guidance and configuration optimization.

- Cache partitioning configuration is a tedious process. Each application or core must be allocated a certain amount of cache, which will impact their performance. In this project, we proposed an evolutionary-based framework to compute the cache allocation scheme that yields to nearly-optimal, if not optimal performance. We base our approach on a multi-objective optimization and provide a visual tool to guide the SI in the configuration process. Our results show that using the optimization tool compared to an even allocation of the cache to each core drastically increases the performances.

- We provide the SI with a tool capable of providing a schedule, the mapping and bus budget allocation of applications in the system. Scheduling each application on a fixed schedule is already a tedious process in single-core environment. Adding the mapping makes it even more difficult. But adding bus budgeting to the equation makes the process nearly impossible. In this work, we proposed a tool that provides a complete configuration scheme. When the system is not schedulable, our method provides the SI with the reasons of this non-schedulability. Compared to optimal methods (i.e. ILP), our approach scales well with the number of application and cores and allow the SI to provide constraints to the configuration, which makes it applicable to the industrial world.

CHAPTER 9 CONCLUSION AND FUTURE RESEARCH

The research work presented in this manuscript overcomes part of the challenges brought by multicore interferences in safety-critical systems. This last chapter presents a summary of this thesis and proposes potential future work to be explored.

9.1 Conclusion

To allow the use of multicore architectures in safety-critical systems, interferences must be better understood, bounded and, when possible, removed. In this thesis, we selected the most impacting interference channels present in COTS processors and proposed to mitigate their impact to allow predictable and more deterministic execution times and behaviors. We explored novel ways to precisely control how shared caches behave and prevent unwanted inter-core data evictions. We extended the state-of-the-art bus control mechanism to allow a more fine-grained control of how applications can access the bus. An interrupt management mechanism was also designed to bound interrupt-related interferences and prevent interrupt storms. Alongside interference mitigation, for each interference channel, we formalized a set of constraints that are validated during run-time to ensure the correct behavior of the system. We gathered those constraints in the form of a so-called Safety Net that can be applied to the RTOS. Finally, we provided tools and methods to compute the system's configuration and reduce the impact interference mitigation methods have on the performance.

This thesis provides the following contributions to the literature:

1. The design and implementation of multiple RTOS modules, capable of bounding or removing cache, DRAM, bus and interrupts related interference.
2. The formalization of run-time constraints, gathered in the form of a so-called *Safety Net*, applied to key points of the RTOS.
3. The definition of an optimization framework to guide the system's designers to correctly allocated caches, memory and bus budgets to the different applications running in the system.

9.2 Future Research

The work presented in this thesis only explore a small area of a very large domain. Interferences are present in many more channels than the ones presented in this work. Moreover, the growing use of dedicated hardware and accelerators (i.e., Graphical Processing Unit (GPU)s, Tensor Processing Unit (TPU)s, etc.) bring completely new interference channels that were not explored before. As a perspective of future work, we propose the following directions:

1. DMA engines bring their lot of challenges when it comes to managing the memory and data transfers implied by the DMA. How to mitigate these interference channels represents a great research challenge.
2. As for Simultaneous Multithreading, heterogeneous systems, which contains different processors can be tedious to work with. While the approaches we presented in this thesis are applicable to such systems, it might not always be the case when some processors embedded in the system do not meet the correct hardware requirements.
3. Manycore processors, such as the *MPPA* architecture from Kalray¹ are slowly introduced in embedded systems. The use of such processor, with Network On Chip communication means presents many new interference channels that need to be addressed.
4. GPUs are being used for Artificial Intelligence and Machine Learning, two domains that emerge in safety-critical systems. The use of a GPU implies the generation of new types of interferences that must be assessed and mitigated. This axis of research represents a very novel and important challenge.
5. The growing use of artificial intelligence brought new architectures with Field Programmable Gate Array (FPGA) embedded in the hardware loop. As for GPUs, controlling FPGA accesses to the resources represent a new challenge in safety-critical systems.

¹<https://www.kalrayinc.com/products/mppa-technology/>

REFERENCES

- [1] A. T. A. Dugo, J.-B. Lefoul, F. G. De Magalhaes, D. Assal, and G. Nicolescu, “Cache locking content selection algorithms for arinc-653 compliant rtos,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: <https://doi.org/10.1145/3358196>
- [2] B. D. Bui *et al.*, “Impact of cache partitioning on multi-tasking real time embedded systems,” in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.
- [3] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.
- [4] C. B. Watkins and R. Walter, “Transitioning from federated avionics architectures to integrated modular avionics,” in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, 2007, pp. 2.A.1–1–2.A.1–10.
- [5] R. Fuchsen, “How to address certification for multi-core based ima platforms: Current status and potential solutions,” Published in 29th Digital Avionics Systems Conference, Salt Lake City, UT, USA, Oct 2010, pp. 5.E.3–1–5.E.3–11.
- [6] A. Löfwenmark and S. Nadjm-Tehrani, “Challenges in future avionic systems on multi-core platforms,” in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014, pp. 115–119.
- [7] C. A. S. Team, “Cast-32, multi-core processors,” USA, November 2016.
- [8] E. U. A. S. Agency, “General acceptable means of compliance for airworthiness of products, parts and appliances amc 20-193,” Europe, January 2022.
- [9] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. USA: USENIX Association, 2005, p. 41.
- [10] C. A. S. Team, “Cast-20, addressing cache in airborne systems and equipment,” USA, July 2003.
- [11] ARINC, “Arinc specification 653: Avionics application software standard interface,” Maryland, USA, 2015.

- [12] S. Mittal, “A survey of techniques for cache partitioning in multicore processors,” *ACM Comput. Surv.*, vol. 50, no. 2, May 2017. [Online]. Available: <https://doi.org/10.1145/3062394>
- [13] R. SC-205, “Do-178c, software considerations in airborne systems and equipment certification,” 2011.
- [14] J. B. Lefoul, A. T. Aurora Dugo, F. Magalhaes, G. Nicolescu, D. Assal, and N. Ulysse, “Simulator-based framework towards improved cache predictability for multi-core avionic systems,” in *2020 Spring Simulation Conference (SpringSim)*, 2020, pp. 1–12.
- [15] C. Yu and P. Petrov, “Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms,” in *Proceedings of the 47th Design Automation Conference*, ser. DAC ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 132–137. [Online]. Available: <https://doi.org/10.1145/1837274.1837309>
- [16] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*, 2006, pp. 423–432.
- [17] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo, “Optimal cache partition-sharing,” in *2015 44th International Conference on Parallel Processing*, 2015, pp. 749–758.
- [18] G. Kedar, A. Mendelson, and I. Cidon, “Space: Semi-partitioned cache for energy efficient, hard real-time systems,” *IEEE Transactions on Computers*, vol. 66, no. 4, pp. 717–730, 2017.
- [19] D. Tam, R. Azimi, L. Soares, and M. Stumm, “Managing shared l2 caches on multicore systems in software,” in *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007, pp. 26–33.
- [20] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez, “Kpart: A hybrid cache partitioning-sharing technique for commodity multicores,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.
- [21] A. Garcia-Garcia, J. C. Saez, F. Castro, and M. Prieto-Matias, “Lfoc: A lightweight fairness-oriented cache clustering policy for commodity multicores,” in *Proceedings*

- of the 48th International Conference on Parallel Processing, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3337821.3337925>
- [22] G. Sun, J. Shen, and A. V. Veidenbaum, “Combining prefetch control and cache partitioning to improve multicore performance,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 953–962.
 - [23] J. Xiao, A. Pimentel, and X. Liu, “Cppf: a prefetch aware llc partitioning approach,” 08 2019, pp. 1–10.
 - [24] M. Nejat, M. Manivannan, M. Pericàs, and P. Stenström, “Coordinated management of processor configuration and cache partitioning to optimize energy under qos constraints,” *ArXiv*, vol. abs/1911.05114, 2019.
 - [25] Z. Cui, L. Chen, Y. Bao, and M. Chen, “A swap-based cache set index scheme to leverage both superpage and page coloring optimizations,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.
 - [26] G. Aupy, A. Benoit, S. Dai, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam, “Co-scheduling amdahl applications on cache-partitioned systems,” *The International Journal of High Performance Computing Applications*, vol. 32, p. 109434201771080, 06 2017.
 - [27] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, “A simple cache partitioning approach in a virtualized environment,” in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009, pp. 519–524.
 - [28] M. Xu, R. Gifford, and L. T. X. Phan, “Holistic multi-resource allocation for multi-core real-time virtualization,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.
 - [29] R. Reddy and P. Petrov, “Cache partitioning for energy-efficient and interference-free embedded multitasking,” *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 3, Mar. 2010. [Online]. Available: <https://doi.org/10.1145/1698772.1698774>
 - [30] I. A. Guney, A. Yildiz, I. U. Bayindir, K. C. Serdaroglu, U. Bayik, and G. Küçük, “A machine learning approach for a scalable, energy-efficient utility-based cache partitioning,” in *ISC*, 2015.

- [31] R. Jain, P. Panda, and S. Subramoney, “A coordinated multi-agent reinforcement learning approach to multi-level cache co-partitioning,” 03 2017, pp. 800–805.
- [32] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson, “Cache sharing and isolation tradeoffs in multicore mixed-criticality systems,” in *2015 IEEE Real-Time Systems Symposium*, 2015, pp. 305–316.
- [33] G. Zhi-Hua and G. Zhi-Min, “Wcet-aware task assignment and cache partitioning for wcert minimization on multi-core systems,” in *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2015, pp. 143–148.
- [34] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, “Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies,” in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 121–130.
- [35] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, “Secdcp: Secure dynamic cache partitioning for efficient timing channel protection,” in *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.
- [36] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell, “Scheduling, isolation, and cache allocation: A side-channel defense,” in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 34–40.
- [37] K. Sivakumaran and A. Siromoney, “Priority based yield of shared cache to provide cache qos in multicore systems,” *International Journal of Parallel Programming*, vol. 45, pp. 634–656, 2016.
- [38] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, “The shift to multicores in real-time and safety-critical systems,” in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2015, pp. 220–229.
- [39] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memory bandwidth management for efficient performance isolation in multi-core platforms,” *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, feb 2016.
- [40] —, “Memory access control in multiprocessor for real-time systems with mixed criticality,” in *Proceedings - Euromicro Conference on Real-Time Systems*, 2012, pp. 299–308.

- [41] W. Ali and H. Yun, “RT-Gang: Real-time gang scheduling framework for safety-critical systems,” in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2019-April. Institute of Electrical and Electronics Engineers Inc., apr 2019, pp. 143–155.
- [42] J. Kim, P. Shin, S. Noh, D. Ham, and S. Hong, “Reducing Memory Interference Latency of Safety-Critical Applications via Memory Request Throttling and Linux Cgroup,” in *International System on Chip Conference*, vol. 2018-September. IEEE Computer Society, jan 2019, pp. 37–42.
- [43] H. Yun, W. Ali, S. Gondi, and S. Biswas, “BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms,” *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1247–1252, jul 2017.
- [44] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, “EMBA: Efficient memory bandwidth allocation to improve performance on intel commodity processor,” in *ACM International Conference Proceeding Series*. Association for Computing Machinery, aug 2019.
- [45] M. Xu, R. Gifford, and L. T. X. Phan, “Holistic multi-resource allocation for multicore real-time virtualization,” in *Proceedings - Design Automation Conference*. Institute of Electrical and Electronics Engineers Inc., jun 2019.
- [46] S. A. Rashid, “Server based task allocation to reduce inter-task memory interference in multicore systems,” in *Proceedings - 2019 International Conference on Frontiers of Information Technology, FIT 2019*. Institute of Electrical and Electronics Engineers Inc., dec 2019, pp. 322–327.
- [47] M. A. Awan, K. Bletsas, P. F. Souto, B. Akesson, and E. Tovar, “Mixed-criticality scheduling with dynamic memory bandwidth regulation,” in *Proceedings - 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018*. Institute of Electrical and Electronics Engineers Inc., jan 2019, pp. 111–117.
- [48] M. A. Awan, P. F. Souto, K. Bletsas, B. Akesson, and E. Tovar, “Memory bandwidth regulation for multiframe task sets,” in *Proceedings - 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2019*. Institute of Electrical and Electronics Engineers Inc., aug 2019.

- [49] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, “Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study,” in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 2:1—2:22. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2017/7174>
- [50] S. Girbal, G. Durrieu, M. Faugere, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, “Predictable flight management system implementation on a multicore processor,” 02 2014.
- [51] S. Park, D. Song, H. Jang, M. Y. Kwon, S. H. Lee, H. K. Kim, and H. Kim, “Interference Analysis of Multicore Shared Resources with a Commercial Avionics RTOS,” in *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 2019-Sept. Institute of Electrical and Electronics Engineers Inc., sep 2019.
- [52] S. Park, M.-Y. Kwon, H.-K. Kim, and H. Kim, “Execution model to reduce the interference of shared memory in arinc 653 compliant multicore rtos,” *Applied Sciences*, vol. 10, p. 2464, 04 2020.
- [53] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler, “Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems,” in *Proceedings - Real-Time Systems Symposium*, vol. 2018-December. Institute of Electrical and Electronics Engineers Inc., jan 2019, pp. 230–241.
- [54] J. Rosen, A. Andrei, P. Eles, and Z. Peng, “Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, dec 2007, pp. 49–60.
- [55] A. Dabaghi and H. Farbeh, “High performance and predictable memory controller for multicore mixed-criticality real-time systems,” *IET Computers & Digital Techniques*, jun 2019.
- [56] M. Hassan and H. Patel, “Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., apr 2016.

- [57] M. Hassan, H. Patel, and R. Pellizzoni, “A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems,” in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2015-May. Institute of Electrical and Electronics Engineers Inc., may 2015, pp. 307–316.
- [58] A. Kostrzewa, S. Tobuschat, R. Ernst, and S. Saidi, “Safe and dynamic traffic rate control for networks-on-chips,” in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2016, pp. 1–8.
- [59] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, “HyPart: A hybrid technique for practical memory bandwidth partitioning on commodity servers,” in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. New York, NY, USA: Institute of Electrical and Electronics Engineers Inc., nov 2018, pp. 1–14. [Online]. Available: <https://dl.acm.org/doi/10.1145/3243176.3243211>
- [60] M. Ziccardi, A. Cornaglia, E. Mezzetti, and T. Vardanega, “Software-enforced Interconnect Arbitration for COTS Multicores,” in *OpenAccess Series in Informatics*, vol. 47. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, jul 2015, pp. 11–20.
- [61] M. Hassan, “Managing dram interference in mixed criticality embedded systems,” in *2019 31st International Conference on Microelectronics (ICM)*, 2019, pp. 253–257.
- [62] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu, “Going vertical in memory management: Handling multiplicity by multi-policy,” in *Proceedings - International Symposium on Computer Architecture*. Institute of Electrical and Electronics Engineers Inc., 2014, pp. 169–180.
- [63] M. Hassan, H. Patel, and R. Pellizzoni, “A framework for scheduling dram memory accesses for multi-core mixed-time critical systems,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 307–316.
- [64] J. Fang, M. Wang, and Z. Wei, “A memory scheduling strategy for eliminating memory access interference in heterogeneous system,” *Journal of Supercomputing*, vol. 76, no. 4, pp. 3129–3154, apr 2020.
- [65] L. Liu, Z. Cui, Y. Li, Y. Bao, M. Chen, and C. Wu, “BPM/BPM+: Software-based dynamic memory partitioning mechanisms for mitigating dram bank-/channel-level interferences in multicore systems,” in *Transactions on Architecture and Code Optimization*, vol. 11, no. 1. Association for Computing Machinery, 2014.

- [66] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, “Coordinated bank and cache coloring for temporal protection of memory accesses,” in *Proceedings - 16th IEEE International Conference on Computational Science and Engineering, CSE 2013*, 2013, pp. 685–692.
- [67] T. Ikeda and K. Kise, “Application aware DRAM bank partitioning in CMP,” in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. IEEE Computer Society, 2013, pp. 349–356.
- [68] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, *Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning*, 2011.
- [69] J. E. Kim, M. K. Yoon, R. Bradford, and L. Sha, “Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems,” in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 321–331.
- [70] S. Zhao, Z. Jiang, X. Dai, I. Bate, I. Habli, and W. Chang, “Timing-accurate general-purpose i/o for multi- and many-core systems: Scheduling and hardware support,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [71] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele, “An isolation scheduling model for multicores,” in *2015 IEEE Real-Time Systems Symposium*, 2015, pp. 141–152.
- [72] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha, “Real-time i/o management system with cots peripherals,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 45–58, Jan 2013.
- [73] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, “Coscheduling of cpu and i/o transactions in cots-based embedded systems,” in *2008 Real-Time Systems Symposium*, 2008, pp. 221–231.
- [74] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang, “A survey on i/o virtualization and optimization,” in *2010 Fifth Annual ChinaGrid Conference*, July 2010, pp. 117–123.
- [75] D. Muench, O. Isfort, K. Mueller, M. Paulitsch, and A. Herkersdorf, “Hardware-based i/o virtualization for mixed criticality real-time systems using pcie sr-ioV,” in *2013 IEEE 16th International Conference on Computational Science and Engineering*, 2013, pp. 706–713.

- [76] Z. Jiang, N. C. Audsley, and P. Dong, “Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 75–84.
- [77] R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, and R. Bradford, “Single core equivalent virtual machines for hard real-time computing on multicore processors,” 2014.
- [78] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.
- [79] M. S. . S. I. MSS, “M-rtos,” 2020. [Online]. Available: <https://www.mss.ca/m-rtos/>
- [80] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.
- [81] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, “TACLeBench: A benchmark collection to support worst-case execution time research,” in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASICS), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.
- [82] L. Sha *et al.*, “Single core equivalent virtual machines for hard real-time computing on multicore processors,” 2014.
- [83] Z. Cui *et al.*, “A swap-based cache set index scheme to leverage both superpage and page coloring optimizations,” in *Proceedings of the 51st Annual Design Automation Conference*, 2014.
- [84] J. Zhou and J. Wang, “Archsampler: Architecture-aware memory sampling library for in-memory applications,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.
- [85] J. D. McCalpin, “Stream: Sustainable memory bandwidth in high performance computers,” University of Virginia, Tech. Rep., 1991-2007.
- [86] P. K. Valsan *et al.*, “Taming non-blocking caches to improve isolation in multicore real-time systems,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

- [87] P. Parkinson, “The challenges and advances in cots software for avionics systems,” in *2007 Institution of Engineering and Technology Seminar on The Pros and Cons of Using Commercial 'Off the Shelf' Components in Aviation Applications*, 2007, pp. 51–61.
- [88] J. Martins *et al.*, “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems,” in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, 2020. [Online]. Available: <https://drops.dagstuhl.de/opus/volltexte/2020/11779>
- [89] G. E. Suh *et al.*, “Dynamic partitioning of shared cache memory,” *J. Supercomput.*, apr 2004. [Online]. Available: <https://doi.org/10.1023/B:SUPE.0000014800.27383.8f>
- [90] B. Sun, T. Kloda, S. Arribas Garcia, G. Gracioli, and M. Caccamo, “Minimizing cache usage for real-time systems,” in *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, ser. RTNS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 200–211. [Online]. Available: <https://doi.org/10.1145/3575757.3593651>
- [91] B. Forsberg *et al.*, “Taming data caches for predictable execution on gpu-based socs,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.
- [92] F. N. et al., “Combining instruction prefetching with partial cache locking to improve wcet in real-time systems,” *PloS one*, vol. 8, 2013.
- [93] M. Bechtel and H. Yun, “Denial-of-service attacks on shared cache in multicore: Analysis and prevention,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2019, pp. 357–367. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/RTAS.2019.00037>
- [94] T. Zhang *et al.*, “A dynamic instruction cache locking approach for minimizing worst case execution time of a single task,” *IEEE Access*, 2020.
- [95] K. K. et al., “Functional-level energy characterization of uc/os-ii and cache locking for energy saving,” *Bell Labs Tech Journal*, June 2012.
- [96] R. Tabish *et al.*, “A real-time scratchpad-centric os for multi-core embedded systems,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

- [97] Y. Liu and W. Zhang, “Exploiting multi-level scratchpad memories for time-predictable multicore computing,” in *IEEE 30th International Conference on Computer Design (ICCD)*, 2012.
- [98] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, “Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.
- [99] A. T. A. Dugo, J.-B. Lefoul, S. Harnois, F. G. de Magalhaes, and G. Nicolescu, “Certifiable memory management system for safety critical partitioned system,” in *ERTS2022*, 2022.
- [100] J. Stankovic and K. Ramamritham, “The spring kernel: a new paradigm for real-time systems,” *IEEE Software*, vol. 8, no. 3, pp. 62–72, 1991.
- [101] M. K. Ludwich and A. A. Frohlich, “Proper handling of interrupts in cyber-physical systems,” in *2015 International Symposium on Rapid System Prototyping (RSP)*, 2015, pp. 83–89.
- [102] J. E. Amiri and M. Kargahi, “A predictable interrupt management policy for real-time operating systems,” *CSI Symposium on Real-Time and Embedded Systems and Technologies, RTEST 2015*, dec 2015. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=cpx_17ec175f15496696b42M589910178163171&database=cpx&view=detailed
- [103] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, “Integrated task and interrupt management for real-time systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 2, jul 2012. [Online]. Available: <https://doi.org/10.1145/2220336.2220344>
- [104] P. Patel, M. Vanga, and B. B. Brandenburg, “TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference,” *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 3–12, jun 2017. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec_5a347cd015d196fc6d9M4d7510178163176&database=ins&view=detailed
- [105] M. Vanga, A. Bastoni, H. Theiling, and B. B. Brandenburg, “Supporting low-latency, low-criticality tasks in a certified mixed-criticality os,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS ’17. New

- York, NY, USA: Association for Computing Machinery, 2017, p. 227–236. [Online]. Available: <https://doi.org/10.1145/3139258.3139274>
- [106] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day, “Chaos: a System for Criticality-Aware, Multi-Core Coordination; Chaos: a System for Criticality-Aware, Multi-Core Coordination,” *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019. [Online]. Available: <https://lkml.org/lkml/2012/12/11/172>.
- [107] S. Sáez and A. Crespo, “Integrated schedulers for a predictable interrupt management on real-time kernels,” in *Ada-Europe*, 2014.
- [108] S. H. Vanderleest, “Taming interrupts: Deterministic asynchronicity in an ARINC 653 environment,” *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pp. 8A31–8A311, dec 2014. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec_5c9dc45d14aeb65b892M49691017816338&database=ins&view=detailed
- [109] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, F. Tronci, A. Biasci, and S. Pinto, “Irq coloring: Mitigating interrupt-generated interference on arm multicore platforms,” vol. 108, Toulouse, France, 2023, arm;Bao;Hypervisors;Interrupt interference;IRQ coloring;Mixed-criticality systems;Multi-core platforms;Runtimes;Spatial isolation;Stringents;. [Online]. Available: <http://dx.doi.org/10.4230/OASIScs.NG-RES.2023.2>
- [110] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, “Task-aware interrupt controller: Priority space unification in real-time systems,” *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 27–30, 2015.
- [111] J. Strnadel, “Load-adaptive monitor-driven hardware for preventing embedded real-time systems from overloads caused by excessive interrupt rates, in: Architecture of computing systems - arcs 2013,” vol. 77, 02 2013, pp. 98–109.
- [112] M. Bargholz, C. Dietrich, and D. Lohmann, “Psic: Priority-strict multi-core irq processing,” Piscataway, NJ, USA, 2022//, pp. 9 pp. –, priority-strict execution;minimal interruption-induced overheads;minimal IRQ software subsystem;inter-core synchronization;PSIC;priority-strict ISR execution;priority-strict multicore IRQ processing;interrupt requests;digital control systems;high-priority interrupt-service routines;ISRs;priority inversions;single-core machines;priority-strict

- IRQ handling;multicore systems;interrupt controllers;. [Online]. Available: <http://dx.doi.org/10.1109/ISORC52572.2022.9812796>
- [113] I. Zagan and V. G. Gaitan, “Real-Time Event Handling and Preemptive Hardware RTOS Scheduling on a Custom CPU Implementation,” *Canadian Journal of Electrical and Computer Engineering*, vol. v 43, n 4, no. 4, pp. 364–373, sep 2020. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec_M9d7bdd9175fbc6fM73c710178163190&database=ins&view=detailed
- [114] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, “BAO: A lightweight static partitioning hypervisor for modern multi-core embedded systems,” *OpenAccess Series in Informatics*, vol. v 77, jan 2020. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=cpx_M2039faa81705a5b5168M6a0810178163146&database=cpx&view=detailed
- [115] I. Pavic and H. Dzapo, “Virtualization in multicore real-time embedded systems for improvement of interrupt latency,” in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018, pp. 1405–1410.
- [116] D. Münch, M. Paulitsch, O. Hanka, and A. Herkersdorf, “Sgint: Safeguarding interrupts for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non transparent bridges.” in *ARCS*, ser. Lecture Notes in Computer Science, L. M. Pinho, W. Karl, A. Cohen, and U. Brinkschulte, Eds., vol. 9017. Springer, 2015, pp. 15–27. [Online]. Available: <http://dblp.uni-trier.de/db/conf/arcs/arcs2015.html#MunchPHH15>
- [117] S. Alonso, J. Lazaro, J. Jimenez, L. Muguira, and U. Bidarte, “The influence of virtualization on real-time systems’ interrupts in embedded soc platforms,” Piscataway, NJ, USA, 2022//, pp. 01 – 6, different isolated environments;different operating systems;embedded applications;embedded SoC platforms;general-purpose;heterogeneous multicore systems;homogeneous multicore systems;interrupts;OpenAMP framework;real-time operating systems;real-time systems;Xen hypervisor;. [Online]. Available: <http://dx.doi.org/10.1109/DCIS55711.2022.9970041>
- [118] B. Blackham, Y. Shi, and G. Heiser, “Improving interrupt response time in a verifiable protected microkernel,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12. New York, NY, USA:

- Association for Computing Machinery, 2012, p. 323–336. [Online]. Available: <https://doi.org/10.1145/2168836.2168869>
- [119] R. Kamal, *Embedded systems: Architecture, programming and design*. Tata McGraw-Hill, 2011.
 - [120] “Interrupt-related interference study | benchmarks, measurements and scripts repository,” <https://github.com/HESL-polymtl/IIntStudy>, 2022.
 - [121] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer: Springer Publishing Company, Incorporated, 2011.
 - [122] L. Bai, J. Wang, and B. Huang, “The effect of shared l2 cache on determinism in airborne embedded system,” 12 2016, pp. 697–700.
 - [123] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multi-core cache management,” in *EuroSys ’09*, 2009.
 - [124] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.
 - [125] H. Qin, “When partitioning works and when it doesn’t: An empirical study on cache way partitioning,” in *Human Centered Computing*, Q. Zu and B. Hu, Eds. Cham: Springer International Publishing, 2018, pp. 595–607.
 - [126] D. E. Goldberg and J. H. Holland, “Genetic algorithms and machine learning,” *Machine Learning*, vol. 3, no. 2, pp. 95–99, Oct 1988. [Online]. Available: <https://doi.org/10.1023/A:1022602019183>
 - [127] S. Lloyd, “Least squares quantization in pcm,” *IEEE Transactions on Information Theory*, vol. 28, no. 2, 1982.
 - [128] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, “SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms,” in *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Madrid, Spain, Jul. 2014, p. 6 p. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01052651>
 - [129] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2021. [Online]. Available: <https://www.gurobi.com>

- [130] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, “The rosace case study: From simulink specification to multi/many-core execution,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 309–318.