Universitat Politècnica de Catalunya

PhD Thesis

A time-predictable many-core processor design for critical real-time embedded systems

Author:
Miloš Panić

Supervisors:
Dr. Eduardo Quiñones
Dr. Francisco J. Cazorla

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

Departament dArquitectura de Computadors
Facultat d'Informàtica de Barcelona

February 2018

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Declaration of Authorship

I, Miloš Panić, declare that this thesis titled, 'A time-predictable many-core processor design for critical real-time embedded systems' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Universitat Politècnica de Catalunya.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at the Universitat Politècnica de Catalunya or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"I myself know nothing,
except just a little,
enough to extract an argument
from another man who is wise
and to receive it fairly."
Socrates

Abstract

Critical Real-Time Embedded Systems (CRTES) are in charge of controlling fundamental parts of embedded system, e.g. energy harvesting solar panels in satellites, steering and breaking in cars, or flight management systems in airplanes. To do so, CRTES require strong evidence of correct functional and timing behavior. The former guarantees that the system operates correctly in response of its inputs; the latter ensures that its operations are performed within a predefined time budget.

CRTES aim at increasing the number and complexity of functions. Examples include the incorporation of "smarter" Advanced Driver Assistance System (ADAS) functionality in modern cars or advanced collision avoidance systems in Unmanned Aerial Vehicles (UAVs). All these new features, implemented in software, lead to an exponential growth in both performance requirements and software development complexity. Furthermore, there is a strong need to integrate multiple functions into the same computing platform to reduce the number of processing units, mass and space requirements, etc. Overall, there is a clear need to increase the computing power of current CRTES in order to support new sophisticated and complex functionality, and integrate multiple systems into a single platform.

The use of multi- and many-core processor architectures is increasingly seen in the CRTES industry as the solution to cope with the performance demand and cost constraints of future CRTES. Many-cores supply higher performance by exploiting the parallelism of applications while providing a better performance per watt as cores are maintained simpler with respect to complex single-core processors. Moreover, the parallelization capabilities allow scheduling multiple functions into the same processor, maximizing the hardware utilization.

However, the use of multi- and many-cores in CRTES also brings a number of challenges related to provide evidence about the correct operation of the system, especially in the timing domain. Hence, despite the advantages of many-cores and the fact that they are nowadays a reality in the embedded domain (e.g. Kalray MPPA, Freescale/NXP P4080, TI Keystone II), their use in CRTES still requires finding efficient ways of providing reliable evidence about the correct operation of the system.

This thesis investigates the use of many-core processors in CRTES as a means to satisfy performance demands of future complex applications while providing the necessary timing guarantees. To do so, this thesis contributes to advance the state-of-the-art towards the exploitation of parallel capabilities of many-cores in CRTES contributing in two different computing domains. From the hardware domain, this thesis proposes new many-core designs that enable deriving reliable and tight timing guarantees. From the software domain, we present efficient scheduling and timing analysis techniques to exploit the parallelization capabilities of many-core architectures and to derive tight and trustworthy Worst-Case Execution Time (WCET) estimates of CRTES.

Acknowledgements

I would like to express my deep gratitude to Dr. Eduardo Quiñones, Dr. Jaume Abella, Dr. Carles Hernández, and Dr. Francisco Cazorla, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank Dr. Theo Ungerer, Dr. Bert Böddeker, Dr. Sebastian Kehr, Dr. Pavel Zaykov and Dr. Antoni Roca for their valuable and constructive suggestions during the planning and development of this research work.

My grateful thanks are also extended to the researchers and engineers of the Computer Architecture / Operating System Interface research group of the Barcelona Supercomputing center: Dr. Leonidas Kosmidis, Dr. Javier Jalle and Mr. Mikel Fernández, for their help and contribution in the development of the simulation framework.

Moreover, I wish to thank my family for their support and encouragement throughout my study.

This work has been funded by the European Commission through the project PARMERASA (FP7-287519) and by the Spanish Ministry of Science and Innovation (TIN2012-34557, TIN2015-65316-P). Miloš Panić held the FPU grant FPU12/05966 (Programa Nacional de Formacin de Profesorado Universitario) of the Ministry of Education, Culture and Sports of Spain.

Contents

List of Figures

List of Tables

Abbreviations

ADAS	Advanced Driver Assistance System
AUTOSAR	AUTomotive Open System ARchitecture
AR-OS	AUTOSAR Operating System
CD	Contention Delay
CFG	Control Flow Graph
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
CRTES	Critical Real-Time Embedded Systems
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
DTA	Deterministic Timing Analysis
ECU	Electronic Control Unit
EMS	Engine Management System
FPGA	Field-Programmable Gate Array
GRP	Guaranteed Resource Partition
GS	Guaranteed Service
HPC	High Performance Computing
IARA	Interference Aware Resource Arbiter
IMA	Integrated Modular Avionics
ILP	Integer Linear Programming
ISA	Instruction Set Architecture
IV	incremental verification
MBPTA	Measurement-Based Probabilistic Timing Analysis
NIC	Network Interface Controller
NoC	Network on Chip
OET	Observed Execution Time
PME	Processor/Memory Element
pSWP	Parallel Software Partition
PTA	Probabilistic Timing Analysis

pWCET	Probabilistic Worst Case Execution Time
QoS	Quality of Service
RAM	Random Access Memory
RTOS	Real-Time Operating System
SoCLib	SoCLib
SMART	Specific, Measurable, Achievable, Realistic, Timely
SWaP	Size, Weight, and Power
SWC	Software Components
SWP	Software Partition
TDMA	Time Division Multiple Access
UAV	Unmanned Aerial Vehicle
UBD	Upper-Bound Delay
UoS	Unit of Scheduling
VC	Virtual Channel
VCI	Virtual Component Interface
WaP	WCTT-aware Packetization
WaW	WCTT-aware Weighted
wNoC	Wormhole-based Network on Chip
WCD	Worst-Contention Delay
WCET	Worst-Case Execution Time
WCB	Worst-Case Execution Time Bound
WCRT	Worst-Case Response Time
WCTT	Worst-Case Traversal Time
zll	zero load latency

To Filip and Tijana.

Part I

Introduction

Chapter 1

Introduction

Embedded systems are ubiquitous nowadays, ranging from mobile phones and medical devices to airplanes and satellites, with a tendency of growth [1, 5]. According to the report by Transparency Market Research [1], global spending on embedded systems will reach US$233.13 bn by 2021, growing from US$152.94 bn in 2014. This growth comes with a significant increment in the performance requirements for future embedded devices to cope with the newest "smart" software functionality. Overall, this tendency is changing the landscape of computer market driving a true convergence of high-performance and embedded computing systems [6–8].This trend can be observed as most chip manufacturers are nowadays targeting embedded systems, diversifying their product portfolio [9, 10]. Furthermore, chip manufacturers are also introducing hardware techniques commonly used in general purpose and high-performance computing into the next generation embedded chips to cope the performance requirements of the modern systems. This thesis advances the current state-of-the-art in this field, boosting the convergence of high-performance and embedded domain, with the emphasis on embedded systems with real-time requirements.

1.1 Real-time Systems

Real-time systems represent a significant part of embedded market [1] with further tendencies of growth [9]. They cover a wide range of applications: from cellphones and routers to medical devices, automobiles, airplanes, satellites, etc. In those systems, time has an essential role in their correct execution, as most of the tasks that form the system have to finish before specific time boundaries – called deadlines – in order to have the system functioning correctly. In order to ensure that tasks deadlines are met, and by doing so guarantee the system correctness from the timing perspective, a process called *timing analysis* is performed. Timing analysis produces upper bounds on the maximum execution times of a task, called Worst-Case Execution Time (WCET) , derived for execution of the task in a specific hardware platform.

Based on the severity of consequences of a task not meeting its associated deadline, we categorize tasks into four groups:

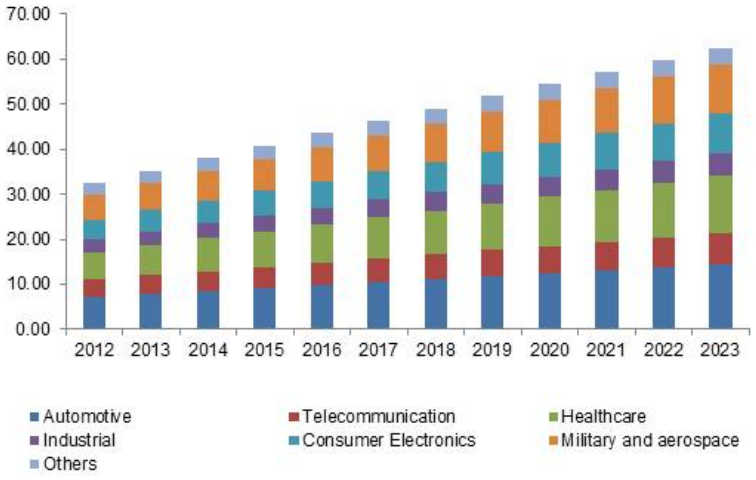


FIGURE 1.1: Europe embedded system market size, by application (US$ billions). Source: [1]

- *Hard real-time tasks* are controlling the most critical functions of the system, e.g. energy harvesting solar panels in satellites, steering and breaking in cars, flight management systems in airplanes, etc. Unexpected deadline misses for these tasks can cause a system malfunctioning that could lead to a loss of human lives or irreversible damage to the equipment and environment. Thus, system construction must guarantee that no deadline is missed, or countermeasures must be provided to ensure system correctness in the case of a missed deadline. Systems containing hard real-time tasks are also known as Critical Real-Time Embedded Systems (CRTES) (either safety-critical or mission-critical). CRTES are the focus of this thesis.

- *Firm real-time tasks* can tolerate very rare deadline misses as they don't have catastrophic consequences. In these systems, the results of tasks are useless after the deadline, leading to controlled degradation of Quality of Service (QoS), e.g. in software defined radio[11].

- *Soft real-time tasks* have more relaxed deadline constraints compared to the hard and firm real-time tasks. Even though the task output can still be useful to the system after the deadline, missing it can lead to uncontrolled QoS degradation. However, in some cases, a missed deadline might not be even noticed by the end user, e.g. skipping a frame in video decoding.

- *Non real-time tasks* do not associate system correctness to their timing behavior and they are rare in real-time systems, e.g. a telematics unit of a car sending usage data to a server of an insurance company.

The High-Performance Embedded Architecture and Compilation (HiPEAC) network [12] recognizes the importance of CRTES industry in Europe and necessity of its further growth and development. Figure 1.1 shows compound annual growth rate of embedded systems market in Europe (courtesy of Global Market Insights [5]), highlighting the importance of automotive and aerospace CRTES industries, that are in the focus of this thesis.

1.2 Trends in Critical Real-Time Embedded Systems (CRTES)

Similar to the embedded systems in general, CRTES industry aims at increasing the number and complexity of system functions to keep the competitive edge, e.g. in avionics [13] and automotive [14] domains. Covering the performance needs of the new software functionalities in CRTES will lead to an increase in safety and comfort of passengers, a better assistance to pilots and drivers, a reduction of fuel demands and carbon emissions, etc. For instance, modern cars already incorporate complex Advanced Driver Assistance System (ADAS) functionality, state-of-the-art Unmanned Aerial Vehicles (UAVs) feature advanced collision avoidance systems, etc. All these new features are implemented in software, thus leading to an exponential growth in both performance requirements and software complexity [13, 15].

Along with a growth in complexity of software functions in CRTES, there is a trend towards integration of multiple systems into the same computing platform. For instance, a modern luxury car can have up to 100 microprocessor based Electronic Control Units (ECUs), ranging from 8-bit, 16-bit, 32-bit microcontrollers, up to multi-core ECUs [16], each executing different functions. There is a severe limitation of space and mass in the car for adding new ECUs. Thus, integration of these multiple functions spread across various ECUs into a single, more powerful computing platform is a must, since it reduces the number of ECUs, cabling and cooling provision, mass and space requirements, etc. This leads to reduction in Size, Weight, and Power (SWaP) costs and keeps the costs of automobiles under constrain imposed by a competitive market.

Overall, in order to support new sophisticated and complex functionality, and integrate multiple systems into a single platform, CRTES require levels of computing power higher than what currently used processors can supply.

Until recently, CRTES have featured simple embedded processors, with single core, short pipelines and in-order execution, suitable for current timing analysis techniques. Even with transistors technology scaling they cannot sustain projected guaranteed performance demands of future CRTES. The computational power deficit cannot be overcome by using more complex single-core processors, i.e. ones with longer pipelines, out-of-order speculative execution of instructions and higher frequencies, due to two major drawbacks: First, such processors can suffer from timing anomalies [17], i.e. events that happen when faster execution of a portion of the code leads to higher execution time in total. This inherently complicates timing analysis that has to consider much higher number of possible states and scenarios of execution in order to derive trustworthy WCET estimates. And second, speculative execution and high-frequencies have high power demands that goes against limited power and cooling budgets in CRTES.

The use of multi- and many-core processor architectures [1] introduced in High Performance Computing industry more than ten years ago, is seen by the CRTES industry as a solution to cope with the performance demand and cost constraints of future CRTES. They provide better performance per watt and maintain simple core design w.r.t. powerful though complex single-core processors, leading to a better thermal and energy efficiency. Moreover, many-cores allow developers to use parallelization as a means to improve applications performance. Finally,

[1] In this thesis, we consider that multi-core processors have up to 16 cores, while many-core processors have 16 or more cores.

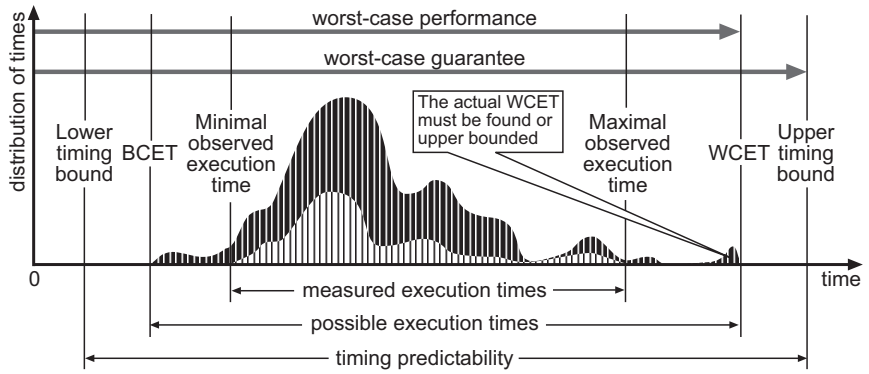


FIGURE 1.2: Time predictability. Source [2]

their use in CRTES allows scheduling multiple applications into the same processor, maximizing the hardware utilization while meeting SWaP constraints.

However, the use of many-cores in CRTES brings significant challenges. Providing evidence of the functional and timing correctness of system components is not trivial in the case of multi-core processors, especially in the timing domain. This effect is exacerbated in many-cores, due to increment in the number of cores [2]. Hence, despite the advantages of many-cores and the fact that they are nowadays a reality in the embedded system domain (e.g. Tilera [18], Kalray MPPA [4], Freescale/NXP P4080 [19], TI Keystone II [20]), their use in CRTES environment still requires finding efficient ways of providing tight and trustworthy WCET estimates.

This thesis investigates the use of many-core processors in CRTES as a means to provide a level of guaranteed performance required for future complex applications and integration of several systems into a single platform.

1.3 Applying multi/many-core technology to CRTES

Timing correctness is a mandatory property of CRTES. It is assessed by system designer and it relies on providing evidence that tasks meet their respective deadlines. However, quantifying the execution time of a task is not trivial, as it depends on many factors (input data, programming language, compiler, hardware architecture, etc.). For example, a simple addition of two variables stored in memory can take between few and dozens of cycles, even when executed on a rudimentary hardware. Thus, the execution time of a task cannot be represented by a single value, but with a distribution of execution times (Figure 1.2). As determining exact WCET is mostly infeasible, the goal of timing analysis is to provide WCET estimates, that satisfy the following constraints:

- **Trustworthiness** – WCET estimate is always higher than actual WCET
- **Tightness** – The difference between WCET estimate and actual WCET is finite and as low as possible, in order to maximize guaranteed system performance.

[2] The solutions proposed in this thesis can be applied for both multi- and many-cores orthogonally.

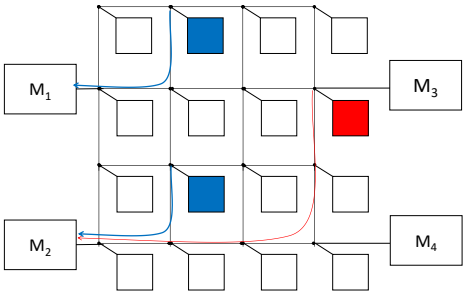


FIGURE 1.3: Memory accesses in many-cores

1.3.1 The timing behavior of CRTES in multi/many-core platforms

Proving timing correctness in multi/many-core processor architectures is challenging due to the impact that interferences have on the timing behavior of the system when accessing the shared hardware resource (shared caches, interconnection networks, memory controllers, I/O devices, chip pins, etc.). Interference occurs when several requests coming from different cores (and possibly from various tasks of the system) try to access a shared hardware resource at the same time, requiring some arbitration mechanism in order to handle contention.

As a result, the latency of accessing a shared resource from one core becomes dependent on the contention created (by a given frequency and an access pattern) from other cores on the same shared resource. This makes derivation of tight and precise WCET estimates more difficult as the WCET estimates dependent on the workload. This effect, which occurs in both multi- and many-core architectures, is exacerbated as the number of cores increases. Additionally, in most of the many-core architectures, latency in accessing memory controllers is not uniform, as the requests traverse various distance in the Network on Chip (NoC), making the analysis more complex (see Figure 1.3).

There are two main ways to account for contention among accesses to shared hardware resources, when analyzing the timing behavior:

- The contention can be accounted as a part of the WCET estimation process. In order to do so, for each shared resource we derive an upper-bound on maximum access time to that shared resource when being affected by interferences. This upper bound is called Upper-Bound Delay (UBD) [21]. Then, at the analysis time (performed in isolation), each access to a shared resource is delayed by its UBD, trustworthy upper-bounding the impact of potential interference.

- The contention a task suffers can be alternatively handled factoring in the schedulability analysis, which is performed at system integration. At this point, the workload is known, as well as the individual characteristics of each task forming the system, i.e. their demand for shared resources. As an input to this process, we use WCET estimates computed in isolation and contention impact is accounted by adding to the tasks WCET estimate in isolation the maximum contention that could be generated due to interference by its co-running tasks.

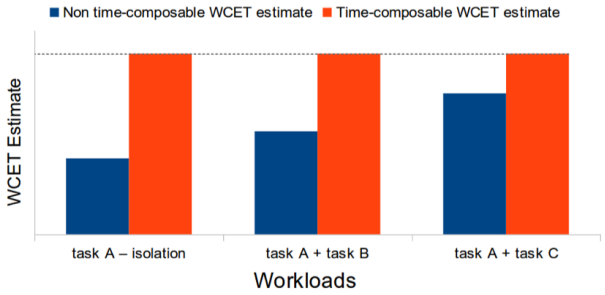


FIGURE 1.4: Time composability

Each approach has its pros and cons. The former allows making WCET estimates time-composable, i.e. independent on the workload, at the cost of WCET over-estimation. The latter enables deriving tighter estimates since it builds upon the knowledge of interference generated by the tasks in the observed task set. However, obtaining time-composability property with the latter approach is extremely challenging [22].

Figure 1.4 highlights the differences among two approaches, when obtaining WCET estimates for the *task A*. *Task A* is run together with *tasks B* and *C* on an example system. Time-composable WCET estimates of *task A* (in orange) do not change, regardless of the workload, while the non time-composable ones change depending whether *task A* is run in isolation or co-running with *task B* or *task C*.

1.3.2 The design of CRTES: Time composability

Software complexity in CRTES is rising with each new generations of systems, e.g. $F-22$ Raptor fighter jet's software has 2.2 millions of lines of code compared to around 1 million in $F-16C$. In order to reduce the complexity, facilitate software development, and improve code portability, maintainability, and interoperability, latest CRTES software is built on top of the standardized software architectures like ARINC 653 (Avionics Application Standard Software Interface), used in avionics domain [23], and AUTomotive Open System ARchitecture (AUTOSAR) used in automotive domain [24].

Along with the growth of software complexity, in CRTES there is a trend towards the integration of various functions into the same devices, further increasing systems complexity. In Integrated Modular Avionics (IMA) [23, 25], as well as in AUTOSAR, engineers rely on robust functional and temporal partitioning, to provide "freedom from interference" in integrated systems [26]. This thesis focuses on providing temporal partitioning in integrated CRTES, achieved through a means of time composability.

Time composability is a design principle that requires WCET estimates hold regardless of the co-running tasks running on the same chip and accessing the shared resources. In CRTES design, time composability is a pillar that enables incremental development and incremental verification (IV) of integrated systems [27]. During system development, it allows independent application/system development, across several vendors. Furthermore, it enables determining

whether application fits its timing budget during the development, while it's easier and cheaper to make the necessary changes, compared to discovering that during system integration. During the system integration, the ability to incrementally integrate applications without the need of regression tests to validate the timing properties of already-integrated applications heavily reduces integration costs. At system deployment, the ability to update functions and their associated software, without the need for re-analyzing and re-certifying the system, is vital in domains like space where systems operate during dozens of years and whose functionality is usually updated once deployed.

Overall, an increase in functions and systems complexity, combined with the integration of multiple systems into the same platform, comes at the cost of more complex and expensive verification and certification processes. This thesis will consider above-mentioned industrial standards without impacting on system development and integration complexity.

1.4 Thesis Goals and Objectives

This thesis aims at boosting the use of many-core processors in CRTES industry, finding ways of efficiently exploiting their strengths and mitigating their negative impact on timing analysis. In order to do so, this thesis defines the following goals:

Goal 1 *Investigate hardware and software solutions to boost guaranteed performance of CRTES applications from avionics and automotive domains through the use of parallel computing.* More guaranteed performance in CRTES domain will not only bring benefits in safety of land- and air-borne vehicles, but will also reduce air pollution and energy consumption, with clear positive impact on quality of life and the environment. Furthermore, it would create a competitive advantage for some of the key European industries.

Goal 2 *Compliance of proposed solutions to current industrial practices and standards.* As stated in Section 1.3.2, avionics and automotive software is built and executed on top of standardized software architectures, namely Integrated Modular Avionics (IMA) [23] and AUTomotive Open System ARchitecture (AUTOSAR) [24]. On top of that, when building CRTES applications, developers comply with the domain-specific safety standards, e.g. DO-178B [28] in the avionics and ISO26262 [29] in the automotive domain. Compliance with these standards is a requirement for all solutions proposed in the thesis to be relevant and useful to these CRTES industries. This thesis is focused on ARINC 653 and AUTOSAR standards.

Goal 3 *Re-usability of legacy code of industrial applications and facilitating migration towards many-core platforms.* In CRTES, many applications/systems have been developed, tested, improved and fine-tuned for several years. Consequently, the CRTES companies already made significant investments in their legacy applications, ranging from requirements, design, development and verification processes. Thus, proposals made in this thesis cannot require from developers to significantly change their well-tested applications, and ideally, the application should behave the same on the many-core platform as when executed on a

Table 1.1: Goals and objectives

	O1	O2	O3	O4	O5
Goal 1	✓	✓	✓	✓	
Goal 2				✓	✓
Goal 3		✓		✓	

single-core. Therefore, we must facilitate the reuse of the existing software designs to reduce the cost of migration to many-core platforms, as well as the reuse of test-cases in order to minimize verification and certification costs

This thesis addresses its goals by defining the following objectives (Table 1.1 shows the relation among them):

O1 *Providing hardware support for improving guaranteed performance of CRTES applications on many-core processors.* Without novel many-core processors designs, tailored to reduce WCET estimates of complex applications, CRTES industry might fall short on reaching its long-term goals. Thus, development of scalable and time-analyzable many-core processor is a must. Furthermore, our envisioned many-core designs, should be as close as possible to Commercial-Off-The-Shelf (COTS) processors, to ease their adoption in CRTES and so that even some low-volume CRTES markets can also exploit their benefits.

O2 *Deriving software solutions for enabling parallel execution on many-core platforms.* This thesis investigates providing software mechanisms in order to enable efficient parallel/concurrent execution of CRTES applications in many-core platforms. Furthermore, it devises new scheduling algorithms for both parallel/concurrent execution of applications and parallel/concurrent execution of functions inside applications, aiming to facilitate parallelization of complex legacy code.

O3 *Improvement of timing analysis methods and techniques.* Improvement of guaranteed performance of CRTES applications cannot only come from hardware/software solutions. In order to satisfy the guaranteed performance needs of future CRTES, it is necessary to reduce the potential overestimation due to multi- and many-core execution, when computing WCET estimates as much as possible as well as evaluate novel approaches in timing analysis.

O4 *Validation of proposals with real industrial applications.* This thesis evaluates and validates the proposed solutions with three industrial case-studies: collision avoidance and stereo navigation applications from avionics (provided by Honeywell Int.) and Engine Management System (EMS) from automotive domain (provided by Denso Deutschland GmbH) to assess their applicability to industry.

O5 *Recommendation to standardization authorities for extension of standards to support parallel execution of applications on many-core platforms.* Current CRTES standards are built and written with single-core processors in mind as the target platform. In order to enable adoption of many-core platforms by CRTES industries, the standards have to be revised. This thesis aims at making same recommendations for updating IMA and AUTOSAR standards, for avionics and automotive respectively.

1.5 Thesis Contribution

This thesis contributes to advancing the state-of-the-art toward the convergence of high-performance and embedded domain. It proposes new many-core hardware designs and efficient software techniques that exploit their parallelization capabilities, targeting CRTES. These contributions are fully inline with the thesis objectives defined in Section 1.4.

1.5.1 Hardware

We propose time-predictable many-core designs suitable for the CRTES of the future. They aim at enabling time-predictable parallelization of CRTES applications as well as the time-predictable parallel execution of them, in line with objective **O1**. To that end, this thesis proposes two novel concepts as an extension to ARINC 653 standard (objective **O5**).

First, aligned with the objective **O2**, we propose Parallel Software Partitions (pSWPs) which improves the concept of ARINC Software Partitions (SWPs), providing a means for parallel/concurrent execution of CRTES applications in many-core platforms.

Second, it proposes physical (hardware) counterparts of the pSWPs – called Guaranteed Resource Partitions (GRPs). For the design of GRPs, we focus on two of the most critical hardware shared resources: *(i)* Network on Chip (NoC) and *(ii)* memory controller, and provide two many-core architectures that implement GRPs: one implementing hierarchical NoC (tree+bus) and another featuring mesh-based NoC, and evaluate the proposals with industrial avionics applications provided by Honeywell International (**O4**). This work, named *"Parallel many-core avionics systems"*, was presented at the ACM International Conference on Embedded Software, EMSOFT 2014, in New Delhi, India [30].

Third, in line with objectives **O1** and **O3**, we introduce a new metric called Contention Delay (CD) that captures the impact of interference in NoC on WCET estimates more accurately compared to currently used metrics. We provide a taxonomy of NoC parameters and an analytical model for computing our proposed metric. This work was presented in an article named *"Modeling High-Performance Wormhole NoCs for Critical Real-Time Embedded Systems"*, at IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016, in Vienna, Austria [22].

Finally, we present two mechanisms that improve utilization and guaranteed performance of a mesh-based many-core processor, by providing fair bandwidth distribution across the chip. This work was published in the Proceedings of IEEE Design, Automation & Test in Europe, DATE 2016, Dresden, Germany, as an article named *"Improving performance guarantees in wormhole mesh NoC designs"* [31].

1.5.2 Scheduling techniques

We propose a set of scheduling techniques targeting proposed many-core designs, along with objective **O2**. First, we present an allocation algorithm *RunPar* that uses functions (in AUTOSAR

called runnable entity or short runnable) as a Unit of Schedulings (UoSs) and maps them into GRPs. It exploits runnable-level parallelism, while reusing legacy code and maintaining the application configuration (**Goal 3**). We apply and evaluate *RunPar* with a real automotive application: Engine Management System (EMS) provided by DENSO Deutschland (objective **O4**). This work appeared in the ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, New Delhi, India, as an article name *RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores* [32].

Second, aligned with the objectives **O2** and **O3**, the thesis builds upon the pSWP and GRP mechanisms and the compositional analysis presented in [30] and provides an algorithm for allocation of parallel applications (wrapped inside pSWPs) onto a many-core platform featuring GRPs. We present Communication-aware Allocation algorithm for real-time Parallel applications on many-cores (CAP) that takes into account communication among applications and tries to reduce its impact on WCET estimates and overall system throughput. This work named *CAP: Communication-aware Allocation algorithm for real-time Parallel applications on many-cores* was presented in the IEEE Euromicro Conference on Digital System Design, DSD 2015, in Madeira, Portugal [33].

1.5.3 Probabilistic timing analysis

In this thesis, we show the applicability of novel Probabilistic Timing Analysis (PTA) techniques to deterministic many-core architectures (objectives **O1** and **O3**). We show that is possible to use and analyze Time Division Multiple Access (TDMA) arbitration policy in the context of Measurement-Based Probabilistic Timing Analysis (MBPTA), leading to trustworthy and tight WCET estimates without introducing any hardware changes, just with padding of observed execution times. This work was also presented at the IEEE Euromicro Conference on Digital System Design, DSD 2015, in Madeira, Portugal, as an article named *Enabling TDMA Arbitration in the Context of MBPTA* [34]. Extended version of this work appeared as a journal article in Microprocessors and Microsystems - Embedded Hardware Design, volume 52, 2017 [35].

1.5.4 Standards

This thesis makes recommendations for extension of ARINC 653 with pSWPs and GRPs concepts to support parallel execution of IMA applications, sent to Honeywell Int. in order to influence ARINC653 standardization committee (objective **O5**).

1.5.5 Open source software

The work done in this thesis is a part of European FP7 project named Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability – parMERASA [36]. Thus, the final contribution of the thesis is the open source software that was developed in the scope of the parMERASA project, and that is publicly available at `http://www.parmerasa.eu/index.php?menu=deliverables`.

1.6 Thesis Organization

This thesis comprises four parts, which are further broken down into chapters:

- *Part I* introduces the reader to the topic and sets the environment for the rest of the thesis. After introducing the problem in Chapter 1, Chapter 2 presents our evaluation framework, including simulation tools SoCLib [37] and gNoCSim [38], WCET analysis tools RapiTime [39] and OTAWA [40], and industrial applications and benchmarks used to evaluate the proposals.

- In *Part II* , we propose a time-predictable many-core hardware design, together with timing models that improve the analysis. In Chapter 3, we show mechanisms for enabling concurrent execution of parallel applications on many-core processors, implemented in two architectures, together with the suitable analysis. Chapter 4 focuses on improving the modeling of latencies of one of the most important shared resources: Network on Chip (NoC). Further, in Chapter 5, we propose two mechanisms for improving guaranteed performance in wormhole NoCs.

- *Part III* presents software scheduling algorithms designed to exploit the performance capabilities of the hardware techniques presented in *Part II*. Chapter 6 presents scheduling techniques for parallelization of complex automotive applications, while in Chapter 7 we investigate the ways of scheduling parallel CRTES applications on many-core platforms supporting GRPs.

- *Part IV* gives a glimpse into the future of the CRTES. In Chapter 8, we investigate the use of promising probabilistic timing analysis techniques with deterministic hardware architectures. Chapter 9 concludes the thesis and considers next barriers and challenges for bringing many-core processors into CRTES domain.

Chapter 2

Experimental Setup

This chapter covers the tools used for the experiments performed in the scope of this thesis. It describes simulation framework used to develop hardware platforms and implement the techniques proposed. Furthermore, it presents case studies and benchmarks for evaluation as well as timing analysis tools used to obtain Worst-Case Execution Time (WCET) estimates.

2.1 Simulation framework

Simulation is an established technique in both industry and academia for evaluation of novel techniques and designs. It is even more crucial in the field like computer architecture, where the cost of building a proposal in a silicon chip is extremely high, in both time and money.

Since the focus of the thesis is the timing behavior of Critical Real-Time Embedded Systems (CRTES) applications, we chose an execution-driven, cycle-accurate simulation framework named SoCLib [37, 41] as the basis for our platforms development. As the thesis investigates novel Network on Chip (NoC) models and designs for CRTES, we integrated a powerful NoC simulator named gNoCSim, developed in scope of the NaNoC project [38], into the SoCLib framework. We also implemented a no-overhead tracing mechanism, in order to support a commercial measurement-based timing analysis tool used in parMERASA project.

It is important to remark that development of the simulation framework is a result of a group effort from current and past members of the Computer Architecture/Operating System interface (CAOS) group at Barcelona Supercomputing Center (BSC) and is used in several research projects: parMERASA, PROARTIS, PROXIMA, etc.

Figure 2.1 gives a high-level overview of the simulation framework and how it integrates with applications, system software and timing analysis tools used in scope of the thesis.

SoCLib is a framework that enables creation of execution-driven cycle accurate simulators. It features its own build system and comes with a set of common components (cores, caches, terminals, memory controllers, buses, NoCs, etc.). Communication among components is carried

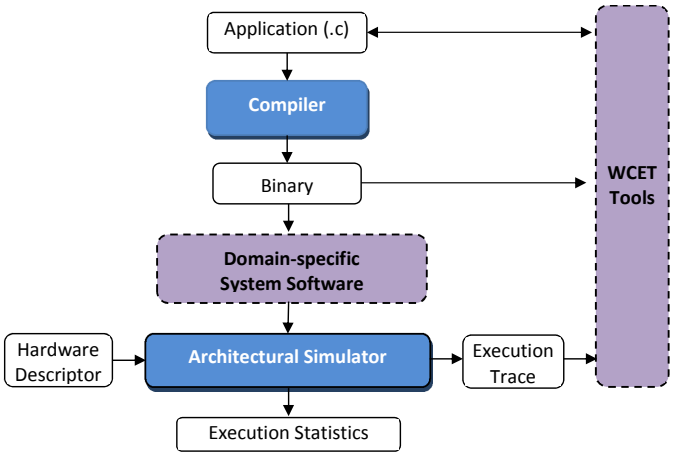


FIGURE 2.1: High-level overview of simulation framework together with system software and WCET tools

out via Virtual Component Interface (VCI) interfaces. It provides a set of emulators for various Instruction Set Architecture (ISA), which enables application development for the simulated platforms, as well as the tools for debugging those applications. Furthermore, it also provides a significant portion of standard low level system libraries, abstracting the hardware from the developer and facilitating the development of applications.

In order to accomplish the goals set of this thesis, we made the following improvements to the SoCLib (SoCLib) framework:

- We added support for PowerPC750 ISA, by extending PowerPC405 emulator provided by SoCLib. We also added atomic fetch-and-add instructions, to help system software implement time-predictable synchronization mechanisms.
- In the original SoCLib, emulators were fully integrated into the platform, and there was no way of providing application developers only functional emulator without the timing simulator. Thus, we created a layer of separation (called RPIBuffer) between functional emulator and timing simulator, allowing our platforms to execute without timing simulator attached.
- The introduction of RPIBuffer allowed us to decouple functionality of the processor core from the emulator and introduce a means of creating different pipelines. In all of the experiments done in scope of this thesis, we use 3-stage pipeline (Fetch, Execute/Memory-Operations, Commit).
- We implement and use VCI interfaces that support multiple VCI requests. That allowed us to have multiple requests in flight coming from a core, memory controllers handling multiple requests, split transactions on the buses, etc.
- We also implement new component for caches in order to support multi-level caches, novel placement and replacement strategies, etc. First level caches are integrated with the processor core module, in order to speed up simulation (they can be disabled through configuration). Second and higher level caches can be integrated into the core, or connected

to several cores via VCI interface, enabling us to build complex cache hierarchies. In scope of this thesis, we use only first level caches, due to the limitations of the analysis methods and to speed up the execution of the simulations.

- We also added support for software controlled cache memories, i.e. scratchpads, as they are widely used in automotive industry. Platform used in Chapter 6 uses scratchpads instead of first level instruction caches.

- We created new memory controller component that supports multiple requests in flight. It allows execution of the applications in the Upper-Bound Delay (UBD) mode and implements prioritization of certain types of requests, as explained in Chapter 3). It provides support for atomic fetch-and-add instructions and can be attached to more complex DRAM simulators.

- SoCLib support for NoCs is lackluster. We created a wrapper component with VCI interfaces around full-blown cycle-accurate flit-level NoC simulator, called gNoCSim [38]. That allowed us to create complex NoC topologies (Chapter 3), implement virtual channels prioritization (Chapter 3). Further, it enabled us to implement weighted round-robin arbitration and packetization (Chapter 5) as well as to experiment with synthetic traffic in gNoCSim standalone mode (Chapter 4 and Chapter 5).

- Alongside gNoCSim, we also added support for buses, both inside the core (between different cache levels in hierarchy) and outside the core with VCI interfaces. We implemented several arbitration policies (Time Division Multiple Access (TDMA), round-robin, etc.) and added support for split-cycle transactions.

- In order to support measurement-based timing analysis (see Section 2.3), we implemented no-overhead tracing mechanism. Inside of RPIBuffer, we detect execution of certain instructions (list of labels provided by the tool) and output the traces of application execution with no calls to instrumentation functions inside the analysed code.

Using this enhanced SoCLib framework, we built a highly-configurable many-core simulation platform, used in the FP7 project parMERASA, that implements designs presented in Chapter 3, Chapter 4 and Chapter 5. Along with objective **O5**, we published this platform as an open-source software project (see Section 1.5.5).

Note that, even though the author contributed in all enhancements listed above, he was the main contributor to the development of the configurable platform, no-overhead tracing mechanism, memory controller, synchronization mechanisms and virtual channel prioritization.

2.2 Case studies

This section presents industrial case studies and benchmarks used in scope of this thesis to evaluate its proposals. In general, there is a lack of standard multi-threaded benchmarks for CRTES applications. Thus, aligned with the objective **O4**, we use 3 industrial applications: 2 from avionics and 1 from automotive domain. We also use a standard suite of single-threaded automotive benchmarks.

2.2.1 Avionics domain

From the avionics domain we use two parallel avionics applications: 3D Path Planning (*3DPP*) and Stereo Navigation (*StereoNav*), used for the navigation of Unmanned Aerial Vehicles (UAVs), provided by Honeywell Int.

The *3DPP* application computes the path between the current UAV position (obtained from a satellite) and the target position (defined by a user), while avoiding obstacles in a 3D environment. It employs Laplace's equation for airborne collision avoidance. It is parallelized based on the split of the 3D obstacle grid into compartments, i.e., sub-grids. Consequent parallel processing depends on data dependencies and thus varies on individual operations. The use of the Gauss-Seidel method [42] in the calculation of the potential matrix creates data dependencies between compartments. Coarse-grained synchronization is required for establishing a proper sequence to process the individual compartments, which follows a well-defined software pipeline pattern.

The *StereoNav* is intended for determining the direction and speed of UAV movement in case satellite signal is unavailable. The *StereoNav* application receives as an input two independent images derived from two cameras pointing at approximately the same direction, and extracts features common for both images in the 3D-space, i.e., dominant entities in the image invariant to rotation and translation. Based on the changes in the features position in different pairs of adjacent images, the absolute translation and rotation of the UAV can be computed.

The parallelization of the *StereoNav* application is based on the ability to process each of the images in a pair in parallel. Steps up to and including the matching of features in each of the images in a pair are currently executed in parallel. The maximum theoretical speed-up achievable in each of the steps of the application varies significantly.

2.2.2 Automotive domain

From the automotive domain, we use an Engine Management System (EMS) application, provided by Denso Deutschland to evaluate proposals developed in scope of this thesis. An EMS is a typical contorl automotive embedded real-time system application. It controls that the amount of fuel and the fuel injection times which is fundamental in ensuring smooth revolutions of the engine. The injection time and fuel amount depend on the state and the rotation speed of the engine, which changes continuously during operation. The EMS requires updates based on tasks that are time-triggered, with task periods ranging from one millisecond to one second, and a task that is triggered based on position of the crankshaft.

The EMS comprises around 1 thousand functions, grouped by the trigger to eleven time-triggered tasks, with periods of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms, and a crank-angle interrupt-triggered task, with a minimum period of 1.25 ms corresponding to the maximum engine rotation speed (4000 rpm, in our case).

Name	Short Description
a2time	Angle to Time Conversion
basefp	Basic Integer and Floating Point
bitmnp	Bit Manipulation
cacheb	Cache "Buster"
canrdr	CAN Remote Data Request
aifft	Fast Fourier Transform (FFT)
aifirf	Finite Impulse Response (FIR) Filter
aiifft	Inverse Fast Fourier Transform (iFFT)
aiirflt	Infinite Impulse Response (IIR) Filter
matrix	Matrix Arithmetic
pntrch	Pointer Chasing
puwmod	Pulse Width Modulation (PWM)
rspeed	Road Speed Calculation
tblook	Table Lookup and Interpolation
ttsprk	Tooth to Spark

TABLE 2.1: EEMBC Automotive Suite

2.2.3 Benchmarks

In order to asses the impact of our designs on single-threaded applications, we use EEMBC Automotive Benchmark suite [43], developed by the Embedded Microprocessor Benchmark Consortium. It captures frequent operations of automotive systems. For instance, a2time simulates an embedded automotive application where the Central Processing Unit (CPU) tries to measures the real-time delay in movement of a crankshaft of an engine.

Table 2.1 shows the list of benchmarks contained in this suite. We execute all of the benchmark on bare-metal system, without any Real-Time Operating System (RTOS) support. Thus, we don't have to account for impact of RTOS, as benchmarks run end-to-end and without preemption.

2.3 Timing analysis

In order to verify the timing correctness of CRTES, it is necessary to perform WCET analysis, i.e. to compute a WCET estimation. To do so, today industries and academias follow two main approaches for WCET analysis [44]: static analysis and measurements based analysis. The former, e.g. aiT [45], OTAWA [40], relies on the construction of a specific cycle accurate model of the computational unit in which the code will run (e.g. the complete processor or a unique core in a many-core architecture), and the construction of a mathematical representation of the timing behavior of the application under analysis running on that processor. The mathematical representation is then processed with Integer Linear Programming (ILP) techniques to determine a safe upper-bound on the execution time. The latter, e.g. RapiTime [39], relies instead on thorough testing of the application under analysis on the real processor or a cycle accurate timing simulator of that processor, with high-coverage stressful input data, and recording the longest observed execution time.

This thesis has considered two WCET analysis tools, one of each type: OTAWA and RapiTime, static and measurement-based timing analysis tools respectively. Next we provide a very short introduction to each them.

2.3.1 Static timing analysis – OTAWA

OTAWA is an open-source static timing analysis [40]. In other to determine the WCET estimate of a program, OTAWA does the following steps:

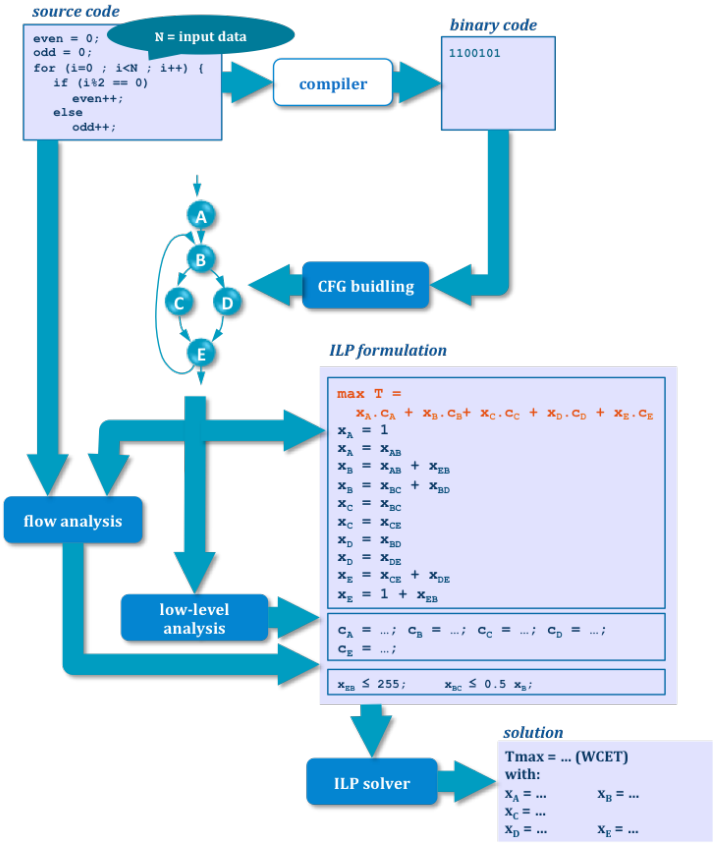


FIGURE 2.2: Workflow of the static timing analysis tool OTAWA.

1. The Control Flow Graph (CFG) is derived from the executable code.
2. Based on the CFG, the tool performs 3 types of analyses: (i) value analysis, (ii) loop bound analysis and (iii) control flow analysis, in order to construct annotated CFG. Value analysis consists of over-approximating set of values and addresses of memory locations. Loop bound analysis determines the bounds on number of iterations for all of the loops in the code, needed to bound the WCET. Control flow analysis eliminates infeasible paths as well as the target of indirect branches. We can further improve the precision of all of these steps by adding annotations to the source code.
3. After constructing annotated CFG, low-levell analysis is perform (comprising pipeline and cache analysis). It is used to determine bound on the execution time for each basic blocks[1]

[1]Basis block is a straight sequence of instructions with single point entry (all jumps go to the first instruction of the block) and only branches out are at the last instruction.

in the annotated CFG are computed. This step, derives the execution time of basic blocks based on the abstract interpretation of each assembly instruction in the corresponding core. As stated in Section 2.1, we consider a PPC750, in which one instruction is executed per cycle, except for memory accesses. In case of memory operations, a *cache analysis* in which for each memory access (either instruction or data) is classified as *AlwaysHit* (for any execution of the program, the instruction or data is in the cache), *AlwaysMiss* (the instruction or data misses and must be retrieved from the higher levels of the memory hierarchy), and *NotClassified* or *Conflict* (the analysis is not able to determine a constant behavior for the access). In this latter case, the analysis will consider the latency of miss.

4. With the information about timing behavior of basic blocks and annotated CFG, OTAWA determines the longest possible execution time using an ILP solver.

Figure 2.2 gives a high level overview of the four steps presented above.

2.3.2 Measurement-based timing analysis – RapiTime

In this thesis, we use the measurement-based WCET analysis tool RapiTime [39], though all our solutions can also be used with static-based WCET analysis tools. RapiTime computes the WCET estimation of a program as a whole probability distribution of the execution time of the longest path, from which the absolute lower and upper bound (i.e. the WCET estimates) are obtained. RapiTime derives an upper bound of the Maximum Observed Execution Time (MOET) for a particular section of code (generally a basic block). It combines MOET with the CFG to determine an overall estimation for the longest control-flow path through the program.

Control-flow graph is generated through a compiler wrapper, provided by RapiTime (see Figure 2.3.). Compiler wrapper inserts instrumentation through annotations at the granularity defined by the user. We analyzed everything at the basic block level, though this can be set at the level of function as well. In our case, with the support from simulation platform and its no overhead tracing mechanism, the code was annotated by just adding labels to the code (no additional instructions).

When the application is run on the target system, our simulation platform in scope of this thesis, a execution trace is produced. A trace basically comprises a sequence of time-stamped values that show when the instrumentation code is executed. From this trace, RapiTime shows performance metrics for each part of executed code and provides boundaries to WCET.

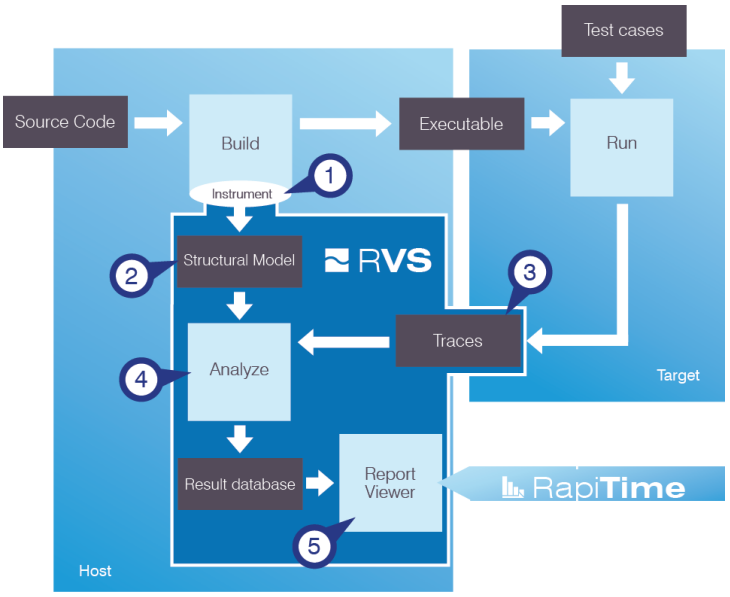


FIGURE 2.3: Workflow of the measurement-based timing analysis tool RapiTime. Source https://www.rapitasystems.com/products/rvs/how-does-rvs-work

Part II

Manycore Hardware Design and Analysis

Chapter 3

A Time Predictable Architecture

Critical Real-Time Embedded Systems (CRTES) rely upon incremental software development and incremental verification to develop and verify each system component in isolation and independently from others. As a means to facilitate incremental development and incremental verification, standardized system software architectures such as Integrated Modular Avionics (IMA) in the avionics domain and AUTomotive Open System ARchitecture (AUTOSAR) in the automotive domain, provide *robust space and time partitioning* mechanisms as a means to isolate timing behavior of components when executing in single-core processors. These mechanisms implement the concept of Software Partitions (SWPs) (as defined in ARINC 653 and ISO26262 standards) that enables incremental verification of applications executed on IMA and AUTOSAR frameworks. The transition towards parallel execution in multi- and many-core processors however, invalidates the space and time partitioning mechanisms as different system components can access simultaneously to shared hardware resources. This can influence their timing behavior and violate the isolation of SWPs.

In this chapter, we introduce two new concepts that enable the parallel execution of multiple system components in parallel architectures, while maintaining the time and space isolation. First, we define Parallel Software Partition (pSWP) which extends ARINC 653 and ISO26262 SWP specifications, maintaining the isolation properties of SWPs when running in a many-core. Second, we introduce Guaranteed Resource Partition (GRP), a new hardware feature that defines an execution environment in which pSWPs run so that interferences in the accesses to shared hardware resources among pSWPs can be controlled. Use of these two new concepts allows incremental verification of many-core avionics systems. increasing the performance while meeting size, weight and power constraints of future CRTES.

3.1 Introduction

Critical Real-Time Embedded Systems (CRTES) industry aims at increasing the number and complexity of system functions to keep the competitive edge, e.g. in avionics [13] and automotive [14] domains. In order to support new sophisticated functionality, CRTES require levels

of computing power higher than what currently used processors can supply. In this context, many-core [1] processors stand as the solution to cope with the performance demand and cost constraints of future CRTES. The use of many-cores in CRTES allows scheduling multiple applications into the same processor, maximizing the hardware utilization while meeting size, weight and power constraints. Furthermore, many-cores allow developers to exploit *task level parallelism* and improve performance of applications.

However, the use of many-cores in CRTES brings significant challenges. CRTES require evidence of the functional and timing correctness for all of their system components, which in case of many-core processors in not trivial, especially in the timing domain, which is the focus of this thesis. Hence, despite the advantages of many-cores and the fact that they are nowadays a reality in the embedded system domain (e.g. Tilera [18], Kalray MPPA [4]), their use in CRTES environment relies on finding efficient ways to deal with timing correctness issues.

A fundamental property of CRTES is incremental verification, that allows each system component to be subject to formal verification in isolation and independently from other components, with obvious benefits for cost, time and effort, reducing the products time to market. Current CRTES enable incremental verification by using standardized system software architectures, such as the Integrated Modular Avionics (IMA) [23, 27] in the avionics domain and AUTOSAR [46] in the automotive domain.

Both software architectures enable incremental verification by guaranteeing *robust space and time partitioning* that make the functional and timing behavior of each application unaffected by other applications. To do so, applications are encapsulated into Software Partitions (SWPs) as defined in the ARINC 653 avionics [23] and ISO 26262 automotive [29] standards. This thesis focuses on time partitioning and facilitating derivation of time-composable Worst-Case Execution Time (WCET) estimates of IMA and AUTOSAR applications (i.e. WCET estimates independent of the co-runners).

SWPs are devised for running in single-core platforms. Each SWP has a dedicated time window in which it enjoys exclusive access to processor resources (e.g., bus and memory). Unfortunately, when moving towards parallel execution on many-cores, *SWPs do not provide the desired time isolation properties*. The fact that several SWPs can simultaneously access shared processor resources creates interferences among them. Thus the use of a dedicated time window per SWP fails in guaranteeing time isolation. This directly impacts certification cost, since when a new SWP is added or changed the entire system needs to be validated. Therefore, providing isolation among applications is key to exploit the performance opportunities of many-cores into CRTES while containing verification and certification costs.

Without loss of generality, this chapter will focus on the avionics domain, considering the communication and isolation mechanisms of IMA and ARINC653, with the objective to facilitate the explanation of the proposed time predictable architecture. However, the same principles presented in this chapter apply to ISO26262 and AUTOSAR. Section 3.2.3 describes the similarities among avionics and automotive software frameworks.

[1] We use the term many-core for processors with at least 16 cores . The problems that this chapter addresses, also arise, to a lesser extent, in multi-core processors.

This chapter comprises the conference paper [30]. It adheres to the thesis goals and objectives defined in Section 1.4 and makes the following contribution:

- We extend the concept of ARINC 653 SWPs and introduce Parallel Software Partition (pSWP) in Section 3.4. We specify how interference among pSWPs in the accesses to hardware resources is controlled to enable incremental verification. pSWPs guarantee time and space partitioning, enable deriving time-composable WCET estimates and reduce integration-time effort (objectives **O2** and **O5**).
- We propose the novel concept of Guaranteed Resource Partition (GRP) in Section 3.5. GRP defines an execution environment comprising a set of processor resources (cores, memory, etc.) in which a SWP runs, avoiding or bounding interferences among applications(objectives **O1** and **O5**).
- We evaluate and compare two many-core architectures supporting GRPs: one using hierarchical (tree+bus) Network on Chip (NoC) in Section 3.5.2.1 and another featuring mesh-based NoC (Section 3.5.2.2); as well as implementation aspects of a required memory controller (Section 3.5.2.3).
- We propose the compositional timing analysis that benefits from pSWPs and GRPs (Section 3.4.5 and Section 3.5.3) and reduces the pessimism in WCET estimates, while maintaining required property of time-composability (objective **O3**).

Overall, the combined use of pSWPs and GRPs enables incremental verification in the time domain for IMA systems running on many-cores and allows the use of compositional timing analysis that reduces the pessimism in WCET estimates. By doing so, it better exploits performance benefits of many-cores in avionics systems.

We evaluate our proposal with a system comprising two *real* ARINC 653-compliant parallel avionics applications provided by Honeywell International (Section 2.2.1): 3D Path Planning (*3DPP*) and Stereo Navigation (*SteroNav*) in Section 3.6 (objective **O4**). We demonstrate that pSWP and GRP fully isolate intra-SWP activities among different SWPs, while inter-SWP effect is reduced to less than 1%. Furthermore, in Section 3.6.4 we show benefits of folding of several pSWPs into a single GRP and flexibility of mesh-based GRPs implementation in improving overall system performance up to 4.9x.

3.2 Integrated Modular Avionics

Integrated Modular Avionics (IMA) enables incremental verification by providing *robust space and time partitioning* to avionics applications [23]. The functional and the timing behavior of each application is isolated from the other applications. This makes applications' behavior *composable* i.e. not affected when the other applications of the system are added or updated. *Functional isolation* prevents any unauthorized service to access and corrupt the private data of other applications. *Time isolation* guarantees that the timing behavior, and so the Worst-Case Execution Time (WCET) estimate of an application, is not affected by the presence of

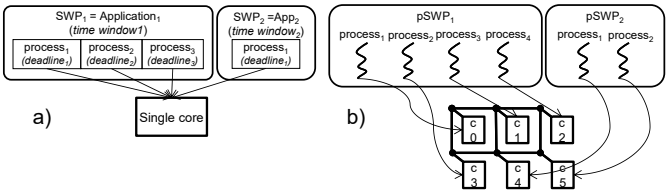


FIGURE 3.1: (a) Time partitioning as defined in ARINC653. (b) 2 pSWP comprising 4 and 2 processes respectively and their mapping to a 6-core multi-core deploying a mesh NoC.

other applications. Time isolation in IMA is mainly driven by the ARINC 653 standard, which encapsulates each avionics application into a SWP as shown in Figure 3.1(a). A SWP comprises one or several processes that share the same memory address space. The interference among processes of the same and different SWPs, in a single core system, is subject to the several ARINC 653 principles, as presented in Section 3.2.2 and Section 3.2.1, to preserve time isolation.

3.2.1 Interference among processes of a SWP

The processes belonging to the same SWP require no time partitioning, hence it is possible that, within a SWP, processes interfere with each other.

Communication. Intra-SWP communication, i.e. communication among the processes belonging to the same SWP, uses buffers, commonly implemented with global variables. ARINC653 provides mutual exclusion and synchronization mechanisms for accessing to those communication buffers.

Computation. Each of the processes comprising the SWP has an associated *deadline*. Scheduling a process in a given SWP occurs exclusively during its time window. ARINC does not specify how processes are scheduled in SWP.

3.2.2 Interference among SWPs

ARINC 653 imposes time and space isolation among SWPs, i.e. among processes associated with different SWPs. Isolation covers both communication and computation activities.

Communication. For communication among SWPs ARINC 653 defines inter-SWP communication means that use queues and messages to exchange data among SWPs. The destination of an inter-SWP communication is a SWP, not a process within it. The source, destination, size and deadline for inter-SWP communications is contained in the configuration tables developed and maintained by the system integrator. Inter-SWP communication among SWPs imposes the order in which SWPs are executed. This makes the *scheduling of SWPs fixed and pre-defined at system integration time*. As guaranteed by ARINC 653, the data coming from other SWPs is available before the execution of their SWP. This is fundamental to ensure that the timing behavior of the destination SWP is independent of the source SWP producing its input data.

Computation. For each SWP, ARINC 653 assigns a *CPU capacity*, implemented in the form of a time window. Each SWP is allocated one time window during which the system *exclusively* executes processes belonging to that SWP, with no interferences from processes of other SWPs.

3.2.3 Similarities between IMA and AUTOSAR frameworks

The AUTOSAR software framework implements a similar communication and computation mechanisms to guarantee space and time partition.

The structural elements of an AUTOSAR application are Software Componentss (SWCs), each containing a set of *runnable entities* (which we call *runnables* for short) that implement the functionality of the SWC, similar to IMA partitions and processes in case of avionics. SWC are forced to be executed in the same single-core processor and runnables are executed as predefined in the AUTOSAR scheduling tables. The tables include a release time and a deadline for each runnable.

Regarding communication, AUTOSAR provides two communication methods among runnables: *sender-receiver* and *client-server* ports. The former uses a global shared memory for communication and it is used to communicate runnables belonging to the same SWC. The latter allows runnables to invoke services from other runnables belonging to different SWC. In case of sender-receiver, runnables read all input data before starting the execution and results are written back after finishing the execution, and so synchronization mechanisms are not required. No limitations on the number of ports or complexity of components are imposed by the model.

All SWC, ports and runnables are known at application configuration time.

3.3 ARINC 653 and many-cores

In single-core execution, the use of time partitioning mechanisms (named *time capacity* in ARINC 653 nomenclature) provide time isolation to SWPs. This is so because shared hardware resources such as the communication bus, memory controller or peripherals are accessed by only one given SWP during its assigned time window. Serialization also simplifies the access to shared software resources like buffers.

Unfortunately, this is not the case in many-core execution models in which the simultaneous execution of SWPs makes software and hardware resources to be shared at the same time by multiple processes belonging to different SWPs (see Figure 3.1(b)). This makes that the timing behavior of one SWP can be affected by other SWPs due to interferences accessing shared resources named inter-SWP interferences, hence breaking time partitioning provided by ARINC 653.

Providing full timing isolation among SWPs in many-core systems is complex if at all possible. Although, some hardware resources can be replicated so that interferences among SWPs do not arise, this principle cannot be extended to other resources such as chip pins – one of the most expensive resources in a processor–, hence inevitably other mechanisms are required to deal with inter-SWP interferences.

In order to contain verification and certification costs in many-core environments, our approach provides time composability (1) to handle the local activities among the processes of a given SWP;

and time compositionality (2) to account for the effect among global activities from different SWPs.

As already introduced in Chapter 1, time composability is a fundamental design pillar for CRTES as it enables incremental development and incremental verification of integrated systems. Moreover, time composability enables to update functions and their associated software without the need for re-analyzing and re-certifying the system.

Time compositionality is a concept orthogonal to time composability. It enables the decoupled analysis of the timing contributions for selected sources of interference [47]. Furthermore, it allows combining the results of individual analysis to a trustworthy upperbound. Thus, leveraging time compositionality, we can separate the analysis of the impact of the global activities of a SWP on the local activities of others SWPs, and account for it when integrating the system.

To that end we introduce two novel concepts: Parallel Software Partition (pSWP) and Guaranteed Resource Partition (GRP) that are described in the following sections.

3.4 Parallel Software Partitions

pSWP is the extension of ARINC 653 Software Partition, designed for use in many-core systems, in which applications are encapsulated to provide the desirable time isolation properties imposed by the standards. Concretely, pSWP specifies the properties required in the timing behavior of SWPs and their processes when executed in many-core architectures. pSWP ensures that a WCET estimate for each process can be derived disregarding the time impact of *inter-SWP interferences*, i.e., interferences among SWP. pSWP covers the impact of accessing to shared software (e.g. buffers and queues) and hardware resources (e.g. bus).

Next we illustrate how SWP interference in shared hardware and software is controlled under pSWP specification.

3.4.1 Shared Software Resources

We distinguish two scenarios: shared software resources within SWP and among different SWP:

Regarding shared software resources within SWP, ARINC653 allows processes of the same SWP to interfere with each other. In that respect, pSWP add no extra constraints. Hence, if the processes of a given pSWP share a software resource, their accesses have to be controlled by using ARINC 653 synchronization mechanisms, e.g. semaphores. The implementation of these synchronization mechanisms must be predictable, like in [48, 49] so timing bounds of the application execution can be derived. The use of synchronization mechanisms in parallel execution must be taken into account by the WCET estimation analysis [50].

Regarding shared software resources among SWP, the communication across SWPs occurs through shared queues. Compliance with ARINC 653 standard ensures that by the time the consumer SWP starts, its producer SWP has ended, preventing any conflict in the accesses to

the software resources. It may be the case that a given SWP_i receives input messages from two SWPs running in parallel, SWP_j and SWP_k. The fact that there is a separate queue for each pair of communicating SWPs, prevents all conflicts in accesses to shared software resources among SWPs.

3.4.2 Shared Hardware Resources

For shared hardware resources, pSWP specify that inter-SWP impact must have an *additive* nature such that the application WCET in isolation can be easily augmented at integration time with such inter-SWP effect, Δ_{inter}.

The interference among processes from the same and different SWPs can occur on computation (P) resources or communication (M) resources. To that end, pSWP specify the impact of intra-SWP (I) and inter-SWP (E) communication and computation activities have on processes, i.e., how processes of the same and different SWP affect each other when running in a manycore.

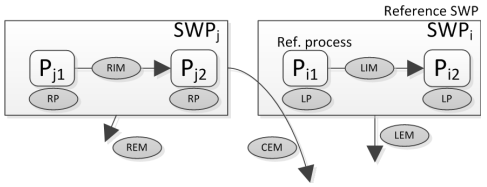
Figure 3.2(a) shows two SWPs (SWP_i and SWP_j) that are executed in parallel, from which SWP_i is taken as reference SWP. Both SWPs comprise two processes: P_{i1} and P_{i2} from SWP_i, and P_{j1} and P_{j2} from SWP_j. Process P_{i1} is taken as reference process. In Figure 3.2(b) columns show the activity carried out in the reference process P_{i1}, which includes local intra-SWP communication (LIM) and local processing (LP); and locally generated inter-SWP communication (LEM). The first two rows show the activities carried out by the other processes in the reference SWP (P_{i2} in the example). The remaining rows show the activity carried out in the other SWP (SWP_j), which includes remote processing (RP), remote intra-SWP communication (RIM) and remote inter-SWP communication, which includes two scenarios: the first scenario may need some resources of SWP_i to be carried out and it is called crossing inter-SWP communication (CEM); the second scenario that does not require any resources assigned to SWP_i is called remote inter-SWP communication (REM).

Next section considers Figure 3.2 to illustrate the impact of hardware intra-SWP and inter-SWP interferemces.

3.4.3 Methods to control Intra-SWP interferences

The ARINC 653 standard does not impose any constraint on the interference in the access to hardware resources among processes of the same SWP. Since only one process can execute at time in single-core processors, the interference among processes is reduced to run-after effects i.e a process P_{i1} depends on the state left by preceding task P_{i2} in the stateful resources. In a many-core processor, the processes of a given SWP execute in parallel sharing hardware resources and hence causing more interference on each other.

pSWP extends ARINC 653 by controlling the interference in terms of communication and computation of processes of the SWPs.



(a)

Reference proc. (P_{i1}) → / offending proc./SWP ↓	Computation (LP)	Intra-SWP Comm (LIM)	Inter-SWP Comm (LEM)
Computation (LP)	Bound or carry out parallel WCET estimation		Remove
Local Intra-SWP Comm. (LIM)			
Remote Intra-SW Comp. (RP)	Remove		
Remote Intra-SWP Comm. (RIM)			
Remote Inter-SWP Comm (REM)			Bound
Crossing Inter-SWP Comm (CEM)	Make additive		

(b)

FIGURE 3.2: Different conflicts among two SWPs and how they are handled by pSWP specification.

3.4.3.1 Computation

On the one hand, the computation that processes P_{j1} and P_{j2} carry out is considered remote computation (RP) for SWP_i. pSWPs specify that RP must not introduce any impact (i.e. must be removed) on local processing (LP) and (local) intra-SWP communication (LIM) of P_{i1} and P_{i2}.

On the other hand, pSWP specifies how the interference among processes in a given SWP is controlled (LP effect on LP). The problem arises in the access to shared hardware resources (e.g. a shared bus) since the slowdown that a process suffers due to contention on that resource depends on the load the other processes put on that resource. As a result, the WCET estimate that may be derived for a process becomes dependent (non time composable) of the behavior of the other processes.

A set of hardware techniques [21, 51, 52] already exist to bound the maximum delay each request of a task (process in our case) may suffer from other tasks in the access to each shared resource. When deriving the WCET estimate for a task this delay is assumed for each request, hence making the WCET of the process independent of the load the others put on each resource.

Alternatively, combined, a.k.a. multi-process, WCET analysis [53, 54] can be carried out. This requires analysing all processes to be run in parallel in the different cores tracking their accesses to the different shared resources to determine whether the accesses from each process would interfere with others. In theory, this analysis leads to tighter WCET estimates but it is more complex and WCET estimates for a process depend on the other processes, so if a process in the SWP changes, all of its processes have to be re-analyzed.

It is important to remark that pSWP enables that WCET bounds are derived taking into account only intra-SWP interferences (Figure 3.2(b)), without requiring any mechanism to control interference among processes of the same pSWP, making them time composable.

3.4.3.2 Communication

Communication among two processes P_{j1} and P_{j2} in a different SWP_j is considered remote intra-SWP communication (RIM) for SWP_i. pSWPs specify that RIM must not introduce any impact (i.e. it must be removed) on local processes (i.e. P_{i1} and P_{i2}) communication and computation. That is, RIM is restricted to SWP boundaries so interference with other SWPs is avoided and time isolation is guaranteed at SWP level. Restricting the effect of intra-SWP

communication on other SWP enables scalability and facilitates incremental development and incremental verification.

Communication among processes belonging to the same SWP (i.e. local communication) is performed through shared memory. The accesses to communication resources are controlled similarly to computation resources, i.e. by upper-bounding the effect of inter-process interference (Figure 3.2(b)) or carrying out a multi-process WCET analysis.

3.4.4 Methods to Control Inter-SWP interferences

pSWP specify that inter-SWP impact must have an *additive* nature such that the application WCET in isolation can be easily augmented at integration time with such inter-SWP effect, Δ_{inter}, and so providing *time compositionality*.

Communication among SWPs is performed through inter-SWP communication methods such as message passing. While intra-SWP activities can be kept local, inter-SWP activities involve at least 2 SWPs: the sender and the receiver. Moreover, the communication among them may require using communication resources assigned to other SWPs.

For instance, let us assume 4 SWP (SWP_1, SWP_2, SWP_3 and SWP_4), with SWP_1 communicating with SWP_4 and SWP_2 executing after SWP_1, and SWP_4 executing after SWP_3 (see Figure 3.3). Under this scenario, the inter-SWP communication between SWP_1 and SWP_4 is a CEM for SWP_3 since it uses resources assigned to SWP_3, e.g. NoC and memory. Next we present three interference scenarios.

3.4.4.1 Impact of intra-SWP activities on Inter-SWP communication ($RIM/LIM \rightarrow REM/CEM$)

Inter-SWP communication uses both, the resources assigned to other processes in its own SWP and in other SWPs. In other words, the communication request must traverse the SWP in which it was generated, as well as others SWPs to reach its destination.

pSWP specifies that inter-SWP communication does not suffer interference from the processing and intra-SWP communication along traversal of its path to reach the destination memory. This is achieved at hardware level by a new concept called *transparent execution* provided by GRPs. One way to achieve transparent execution is by giving priority to crossing inter-SWP communication over intra-SWP communication (Figure 3.2(b)). Section 3.5 explains in detail transparent execution as implemented in GRPs.

3.4.4.2 Impact of Inter-SWP communication on intra-SWP activities ($CEM/REM \rightarrow RIM/LIM$)

Enabling incremental verification in the timing domain requires the ability to derive WCET estimates for the processes of one SWP such that those estimates: (i) do not depend on

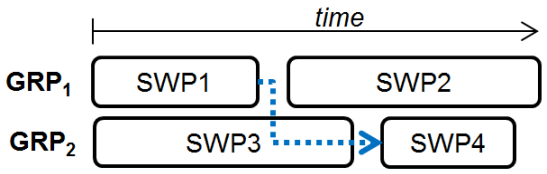


FIGURE 3.3: Example of execution of SWP over time

remote (REM) inter-SWP communication; and (ii) the dependence on crossing inter-SWP communication CEM is limited. Failing to do so would imply that WCET estimates would depend on the particular processes running in other SWPs, thus breaking timing composability.

The main idea to achieve this is letting CEM proceed with higher priority without changing the state of shared hardware resources so that intra-SWP activities are simply delayed by the duration of inter-SWP communication and then resumed. The *additive* delay intra-SWP activities suffer can be easily accounted at integration time when **inter-SWP communication characteristics are known** (Figure 3.2(b)) as required by IMA systems.

3.4.4.3 Interferences among inter-SWP communication ($CEM/REM \rightarrow CEM/REM$)

Eventually, several inter-SWP communications can occur simultaneously and compete for shared resources. pSWPs impose that the timing effect that one inter-SWP communication may have on other inter-SWP communications is bounded by a means of time-predictable hardware arbitration policies so that interferences can be bounded (Figure 3.2(b)). For instance, by using round robin and accounting for the maximum arbitration delay at analysis time[21]. Similarly, Time Division Multiple Access (TDMA) arbitration policies serve the purpose of covering this constraint of pSWP[55].

3.4.5 WCET and Time composability under pSWP

To sum up, there are four pillars of pSWP specification that make intra-SWP effect on execution time composable and inter-SWP effect on execution time compositonal:

- The effect of intra-SWP activities of different SWPs are isolated from each other.
- Inter-SWP activities are prioritized over intra-SWP activities making the former to suffer no impact due to the latter (transparent execution).
- The effect that intra-SWP activities suffer from inter-SWP communications is additive and can accounted for at integration time. This is the consequence of the transparent execution to be provided by GRPs.
- The effect of the inter-SWP communication of one SWP over the inter-SWP communication of another SWP is bounded.

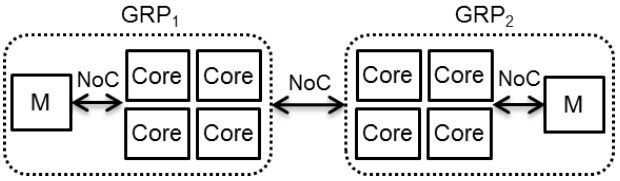


FIGURE 3.4: Processor architecture comprising 2 GRPs.

Overall, transparent execution makes that the computed WCET estimate of an application in isolation can be simply augmented by an *additive factor* that bounds the increment on the WCET estimate due to system integration, i.e. due to inter-SWP communication when the application is integrated into the system. This is shown in Equation 3.1.

$$WCET_{integration} = WCET_{isolation} + \Delta_{inter} \tag{3.1}$$

$WCET_{integration}$ is the final WCET estimate of the application after system integration, $WCET_{isolation}$ the WCET estimate of the application computed in isolation, while Δ_{inter} bounds inter-SWP interference. The benefits at system integration are that pSWP specification allows computing the WCET estimation for each application in isolation. Then, each process allocates an interval Δ_{inter_i} to enable crossing inter-SWP communications. The fact that the information of the inter-SWP communications is known at integration time (as requested by IMA) and the fact that pSWP make the effect of inter-SWP communication additive on the WCET computed in isolation, simplifies validating the timing behavior at integration time.

At deployment time, when a given SWP_j is updated leading to SWP'_j, if the effect of SWP'_j' on any other SWP SWP_i, $\Delta_{inter(j' \rightarrow i)}$, is smaller than the effect generated by SWP_j, $\Delta_{inter(j \rightarrow i)}$, then SWP'_j can be integrated (composed) in the system without requiring reanalyzing any existing SWP. Analogously, SWP'_j should also be able to allocate at least the same time as SWP_j for crossing communications.

Interestingly, transparent execution makes SWP timing behavior independent of the *particular pattern of inter-SWP communication* of other crossing SWPs. Instead, once a SWP has a Δ_{inter} computed, it is time composable with any other SWP that incurs on the former less than that Δ_{inter}.

Next section describes the hardware support required by pSWP to guarantee the timing properties presented in this section and summarized in Figure 3.2(b).

3.5 Guaranteed Resource Partitions: GRP

In this chapter, we propose that many-core processor architectures tailored for use in CRTES introduce a new hardware feature called Guaranteed Resource Partition (GRP). GRP, which is the hardware counterpart of the pSWPs, defines an execution environment composed of a set of processor resources, including cores, NoC resources, memory, etc., that provides to pSWP

the desirable time composability properties as defined in Section 3.4. In other words, GRPs provide *islands of execution* at hardware level to execute pSWPs and so provide the required time isolation guarantees at the hardware level.

Concretely, GRPs guarantee that interferences among intra-SWP requests are not allowed, while the interference among inter-SWP requests in different islands is limited, to facilitate the estimation of their WCET. Hence, GRPs remove the need to control interferences among all running SWPs when computing the WCET of each SWP. In the following sections, we show that the concept of GRP can be implemented in two common many-core designs: one based on hierarchical organization and another featuring mesh-based NoC.

3.5.1 Main timing aspects of GRPs

GRPs maintain the timing characteristics imposed by pSWPs. To that end, GRPs rest on the following main principles: *time predictability* in the access to shared hardware resources, *transparent execution* between intra-SWP and inter-SWP communication and *isolation of intra-SWP communication* requests among different GRPs. At core level, we assume a design free of timing anomalies [56]. Extending GRPs for cores that exhibit timing anomalies is outside of the scope of thesis, and remains future work.

3.5.1.1 Time Predictability

In order to be able to derive WCET estimates, all the shared hardware resources have to be time predictable. A shared hardware resource is said to be time predictable if (i) the time a request has to wait to have the access granted is bounded; and (ii) the time a request takes to be serviced by that resource, once it has been granted access to it, is also bounded. For instance, consider a shared bus deploying round-robin access policy. The service time of the bus is fixed by design. Further, the longest time a request has to wait to get access to the bus can be derived [21], making the bus time predictable resource.

3.5.1.2 Transparent execution

One way to consider the impact of inter-SWP communications on intra-SWP ones (and vice-versa) is assuming that they interfere with each other. This would require assuming that, at analysis time, for every intra-SWP communication a potential conflict with an inter-SWP communication may occur (and vice-versa), which would lead to pessimistic WCET estimates (quantitative figures about pessimism are provided in Section 3.6). Instead, we propose transparent execution of inter- and intra-SWP communications where intra-SWP communications are assumed not to compete with any inter-SWP one for WCET estimation. Inter-SWP communication effect is later accounted for at integration time. *This can be done because IMA systems impose that inter-SWP communication is statically known and so the impact of inter-SWP requests is known at system integration time, when the different SWPs are mapped into the many-core.*

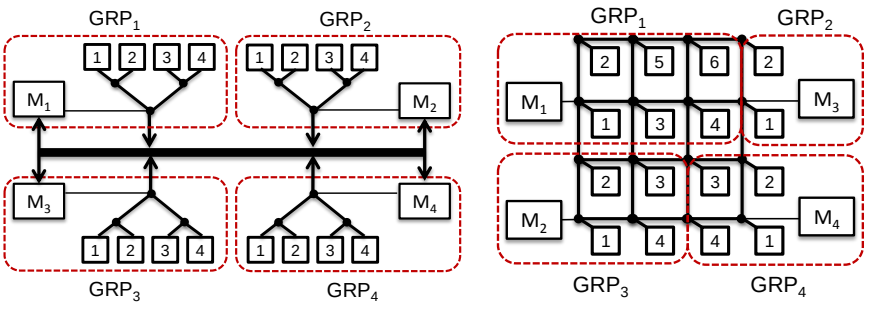


FIGURE 3.5: (a) Clustered design with four clusters, each with four cores. (b) Regular design comprised by four GRPs with different number of cores (6, 2, 4 and 4 cores)

In order to implement transparent execution we propose the memory device and the NoC to provide mechanisms to *freeze* local GRP communications, i.e. intra-SWP requests, instruction requests and process' private data accesses. On the event of an inter-SWP communication, GRP resources are 'frozen' for local requests letting the inter-SWP communication requests to proceed. That is, inter-SWP communications are prioritized over intra-SWP activities. This, on the one hand, makes that inter-SWP communications suffer no slowdown due to interferences in the use of resources. On the other hand, since inter-SWP communication is known statically at system integration and GRP components are time predictable, the impact of inter-SWP communication requests when traversing the NoC and the memory device can be easily determined [21, 51, 52, 57].

Those shared processor resources implementing the freeze mechanism to provide transparent execution must guarantee that the resource state is not affected by the execution of inter-SWP communication requests, so the contribution of intra-SWP communication requests to the WCET estimate remains the same when running the application in isolation and in conjunction with others applications. This is not the case, for instance, for shared caches, in which the access of inter-SWP communication requests may change the cache state, making intra-SWP communication requests vary its timing behavior with respect to running the application in isolation. In this case, the resource would require implementing cache partitioning techniques [21] to isolate inter-SWP communication from intra-SWP.

3.5.1.3 Isolation of intra-SWP communication requests among different GRPs

The requests generated among processes belonging to the same SWP, i.e. intra-SWP communication requests, as well as instruction fetch memory requests or process private data accesses, are not allowed to exceed GRP boundaries. This effectively avoids remote intra-SWP activities to interfere with local intra-SWP activities, and thus simplifies deriving time-composable WCET estimates. To that end, each GRP has a private memory region that is accessed by intra-SWP requests without interference from/to other GRPs. Moreover, the NoC design must guarantee that there is no path from cores to memory that exits GRP boundaries.

3.5.2 Implementation aspects of GRPs

The timing properties of GRPs can be attained by deploying clustered architectures [4, 58], which organize processor resources into islands such that accesses to local resources are faster than to remote resources, see Figure 3.5(a). Alternatively, certain physically monolithic (regular) architectures can be also deployed by creating virtual islands of execution (virtual clusters), see Figure 3.5(b) if certain properties are fulfilled. That is, communication between cores and memory has to be performed in such a way that the interference among virtual clusters is controlled.

The most critical hardware shared resources of a many-core are the NoC and the memory controller. This section analyzes NoC design in the context of physically clustered architectures(Section 3.5.2.1) and following the same principles we extend the analysis to virtually clustered architecture (Section 3.5.2.2). Furthermore, we analyze the memory design in Section 3.5.2.3.

3.5.2.1 NoC Design: Physical GRPs

Clustered architectures usually deploy hierarchical NoCs: a first-level NoC connects cores that compose a cluster and one or several NoC levels connect clusters. Figure 3.5(a) shows an example of a clustered design considered in this chapter. It features a two-level NoC: a first-level composed of a tree, and a second-level composed of a bus.

A hierarchical NoC design provides *isolated communication islands* in which different communication requests, i.e. intra- and inter-SWP, use different NoC levels that do not interfere among them. In our hierarchical design shown in Figure 3.5(a) the memory address space of the SWPs executed in GRP_i resides in memory M_i and so intra-SWP requests will only use its corresponding first-level NoC, i.e. a tree, without interfering with intra-SWP requests of other GRPs. When a SWP wants to communicate with another SWP, the requests traverse the second-level NoC, i.e. a bus, and the message is directly stored in the memory resource corresponding to the GRP in which the destination SWP will run in the future without affecting other clusters. Note, however, that both communication requests (i.e. intra-SWP and inter-SWP) may conflict in the memory device. Section 3.5.2.3 discusses the memory design in the context of GRPs.

The different NoC levels must also provide time predictability. In particular, the latency of each request to cross the NoC must be have a Worst-Case Traversal Time (WCTT) bound. The WCTT is the sum of two factors: The *zero load latency* (*zll*) [59] and the *NoC Request Interference Delay* (NoC_{RID}). The former provides the traversal time of a NoC request assuming zero interferences. The latter provides the maximum time a packet can be delayed due to contending flows in the network when accessing the main memory. Moreover, we consider that requests and responses use different networks (as in [18, 60]) so an independent analysis can be applied to requests and responses.

In NoCs each flow comprises a set of packets, which are further split into flits whose size depends on the NoC implementation. A flit is the minimum flow control unit in the NoC load request to memory is a packet of 64 bytes that can be divided into four 16-byte flits. We define L as the

number of flits of the packet. The maximum number of flows contending for resources in a given router at a given time instant is called Zc [3]. A given flow Z_i has L_{Z_i} flits.

Our first-level NoC design, in which both intra-SWP and inter-SWP communications occur, considers a wormhole-based tree implementing $N-1$ simple pipelined 2-to-1 routers, with a traversal time of D_{router} cycles, to connect N cores [61], so each core requires $log_2(N)$ hops to reach the memory or the second-level NoC. In a such NoC design Zc equals 1, so the maximum time a message is blocked at each hop is determined by the number of flits of the contending message (L_{Z_i}). Equation 3.5.2.1 provides a means to calculate the $WCTT$ of a tree:

$$\begin{aligned} WCTT_{tree} &= NoC_{RID}^{tree} + zll_{tree} \\ NoC_{RID}^{tree} &= \sum_{i=1}^{log_2(N)} L_{Z_i} \\ zll_{tree} &= (log_2(N) \times D_{router}) + (L-1) \end{aligned} \tag{3.2}$$

For our 4-core cluster assuming $D_{router} = 2$ and $L = 4$ we have $zll_{tree} = (2 \times 1) + (4-1) = 5$ and $NoC_{RID}^{tree} = \sum_{x=1}^{log_2(4)} 4 = 8$.

The second-level NoC is exclusively used for inter-SWP communications, since intra-SWP communications are not allowed to leave each cluster. For the second-level NoC we use a non-pipelined bus with a latency D_{bus} implementing a round robin arbitration policy [21]. Inter-SWP communications coming from a different cluster are serialized by their access to the bus. In a bus, the maximum time a message is blocked is set by the latency of the bus (D_{bus}), the number of flits of each contending message (L_{Z_i}) and Zc that equals the number of GRPs minus one ($Zc = N_{cl} - 1$). Equation 3.5.2.1 shows how to derive the $WCTT$ for a bus. In [55] authors show how round-robin arbitration achieves comparable results to TDMA in terms of average and guaranteed performance. In our setup from Figure 3.5(a), we have $zll_{bus} = 8$ and $NoC_{RID}^{bus} = 24$, assuming $D_{bus} = 2$ and $L = 4$.

$$\begin{aligned} WCTT_{bus} &= NoC_{RID}^{bus} + zll_{bus} \\ NoC_{RID}^{bus} &= \sum_{i=1}^{Zc} D_{bus} \times L_{Z_i} \\ zll_{bus} &= D_{bus} \times L \end{aligned} \tag{3.3}$$

3.5.2.2 NoC Design: Virtual GRPs

In *regular NoC designs*, e.g. mesh networks, cores are organized in a regular 2-dimensional grid. These networks are very common in current processor designs due to their regular physical arrangement and short wire connection allowing high-speed operations among neighbor nodes.

Meshes also allow defining GRPs such that local intra-SWPties does not affect remote intra-SWP activities. To that end we define *virtual clusters* by grouping adjacent cores in rectangular shapes (i.e. organizing cores in groups of 2, 4, 6, 8, 9) with a memory device connecting to one of the cores. By using XY (or YX routing) [2] policy [59], we create isolated communication islands with properties similar to those of clustered architectures.

Figure 3.5(b) shows a processor implementing a mesh in which four GRPs are defined: GRP_1 composed of 6 cores, GRP_2 composed of 2 cores and GRP_3 and GRP_4 composed of 4 cores each. By using XY (or YX) routing we can guarantee that requests to nodes within the GRPexceed its boundaries if the memory device resides within the virtual cluster as shown in Figure 3.5(b). In that Figure, packets from $node_6$ in GRP_1 to memory are routed through nodes 5-2 (X dimension) and then through node 1 (Y) dimension.

Inter-SWP communication can affect other SWPs when it accesses to the memory devices of other GRPs. In order to account for this interference in the WCET estimate of the SWP, we propose the use of Virtual Channels (VCs), one per each communication type, providing higher priority to the inter-SWP communication. Such a design forces local GRP communication, i.e. intra-SWP requests, instruction requests and process' local data accesses, to be stalled until the inter-SWP communication finishes as imposed by the GRP definition.

Note that the use of virtual channels accomplishes the property that the state of the NoC does not change after inter-SWP communication requests are executed. However, in general case, virtual channel prioritization could lead to starvation of intra-SWP requests. In order to avoid it, we can leverage the fact that ARINC 653 imposes that the amount of inter-SWP communication and its impact are known at the system integration. To do so, in Chapter 7, we present a scheduler that takes into account inter-SWP communication and prevents starvation of intra-SWP requests.

Meshes also allow deriving the WCTT as the sum of zll and NoC_{RID} in the same way we did for the hierarchical NoCs. We consider a mesh network design where requests and responses use different virtual networks as in [18, 60] and routers are pipelined. Additionally, the proposed mesh implements 2 VCs: one for requests of intra-SWP communication and one for requests of inter-SWP communication. Equation 3.4 computes the WCTT factors of a mesh.

$$NoC_{RID}^{mesh} = \sum_{j=1}^{hops} \sum_{i=1}^{Zc} U_{Z_i}^{j}$$
$$zll_{mesh} = (hops \times D_{router}) + (L - 1) \tag{3.4}$$

Where $U_{Z_i}^{j}$ is the time required for a message of contending flow Z_i to go from the output buffer of the router in hop j to the input buffer of router in the next hop $(j+1)$, and $hops$ is the total number of hops to the target node. Similarly to the tree, $U_{Z_i}^{j}$ is a function of the number of flits of the contending message (L_{Z_i}). For instance, in Figure 3.5(b) up to 6 communication

[2] With XY routing packets are forced to use the X dimension first. In the X dimension the position of the target node with respect to the source node determines whether to go right or left for the X dimension or up or down for Y dimension.

TABLE 3.1: WCTT factors (zll + NoC_{RID}) for regular (Mesh) NoC designs assuming pipelined routers with $D_{router} = 2$ and $L_{Z_i} = 4$. The *Core Id* refers to the location of cores shown in Figure 3.5(b).

	6-Mesh GRP		4-Mesh GRP		2-Mesh GRP	
Core	RID	zll	RID	zll	RID	zll
1	8	5	8	5	4	5
2	20	7	8	7	4	7
3	20	7	20	9	-	-
4	20	9	20	7	-	-
5	36	9	-	-	-	-
6	36	11	-	-	-	-

flows of GRP_1 can reach the memory controller (the ones originated at nodes 2, 3, 4, 5, and 6) crossing $node_1$'s router. However, only $Zc = 2$ flows contend in node1 of GRP_1 at the same time. Therefore, we have $zll_{mesh} = (2 \times 1) + (4 - 1) = 5$ and $NoC^{mesh}_{RID} = \sum_{j=1}^{1} \sum_{i=1}^{2} 4 = 8$. Note that in the last hop the time a contending flow requires to leave the router is equal to $L_{Z_i} = 4$. We refer the reader to [3] for a detailed explanation of the Equation 3.4. It is important to remark that, because the number of hops required to reach the memory depends on the core in which the request is issued, different cores have different $WCTT$. Alternatively, the worst $WCTT$ could be considered, being a pessimistic solution though.

Table 3.1 shows the zll and NoC_{RID} of each core that form the four GRPs shown in Figure 3.5(b) (labeled as *2-Mesh GRP*, *4-Mesh GRP* and *6-Mesh GRP* respectively), assuming $D_{router} = 2$ and $L_{Z_i} = 4$. The *Core Id* refers to the location of cores shown in Figure 3.5(b).

Overall, despite physically clustered architectures lead to tighter WCTT, in a clustered architecture the number of cores per GRP is fixed, and this determines the maximum parallelization level that the SWP can exploit. Increasing the number of cores would require SWPs to use multiple GRPs, which would force intra-SWP communication requests to traverse the second level of NoC, and so conflicting with inter-SWP communication requests. Instead, meshes allow defining virtual clusters in which the number of cores is not fixed by the hardware, providing higher flexibility than hierarchical NoCs, at the price of increasing the WCTT and complicating the estimation of the WCET as the WCTT depends on the core in which processes run.

3.5.2.3 Memory Design

ARINC 653 communication methods are implemented through memory, so that memory requests generated by inter-SWP communication and intra-SWP (local-GRP) memory accesses, may potentially conflict in the access to memory affecting each other behavior.

In order to support transparent execution, we propose a memory controller in which intra-SWP and inter-SWP requests are put in different queues, giving higher priority to inter-SWP requests and freezing intra-SWP requests until pending inter-SWP requests are serviced.

Our memory controller enables bounding the impact that inter-SWP requests have on other inter-SWP requests, as well as the impact that intra-SWP requests generated by a process may also have on the requests of another process of the same SWP. In the case of intra-SWP requests,

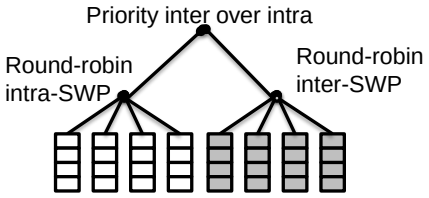


FIGURE 3.6: Structure of memory request queues.

this is achieved by having one request queue per core – which in fact are the intra-SWP requesters – and using a time-predictable arbitration policy, such as round robin [21]. Similarly, by having one request queue by each potential generator of inter-SWP requests, i.e. clusters, together with the use of a predictable arbitration it is enough to bound the effect of inter-SWP memory requests on other inter-SWP memory requests.

As a result, we propose a structure of requests as shown in Figure 3.6. Requests coming from intra-SWPs are organized per core and arbitrated using round-robin. Similarly, inter-SWP coming from other clusters are split per cluster and arbitrated using round robin. Finally, to implement transparent execution, inter-SWP requests are prioritized over intra-SWP ones, which are frozen if needed to favor the former. Using separate queues among inter- and intra-SWP requests ensures that, inter-SWP memory requests do not change hardware state visible to intra-SWP requests when they are frozen and resumed. This allows bounding effect from one inter-SWP request to another and from intra-SWP requests generated by one process on the request of others.

These multiqueue structures can be implemented with a *single* physical queue and the proper use of pointers [62]. In fact, current processors already implement multiqueues with a single queue operated by multiple pointers. For instance, this is the case of the Global Completion Table in each core of the IBM POWER7 [63], that is a queue shared by all 4 running threads in the core.

Similarly to the NoC design, the memory controller must be time predictable, i.e. the WCET estimate accounts for the *memory worst case response time* (Mem_{WCRT}), which expresses the maximum time a memory request can take due to interferences, in our case interference among intra-SWP communication requests from the same SWP and among inter-SWP communication request from different SWP.

The Mem_{WCRT} is composed of the sum of two factors [51]: (1) The *request execution time* (Mem_{RET}) and (2) the *Memory request interference delay* (Mem_{RID}). The former provides the amount of time a request takes to be completed assuming no interferences. The latter provides the maximum time a request may be delayed due to other memory requests. In this thesis, we use a notation similar to the one used in [51]. The memory request interference delay is given by $Mem_{RID} = \sum_{x=1}^{NumQ} t_{LID}$ where t_{LID} is the longest issue delay that a memory request may suffer considering the generic timing constraints described in the JEDEC standard [64] and $NumQ$ the number of request queues. In case of intra-SWP communication requests, $NumQ$ will be determined by the number of processes in a GRP that can simultaneously issue a request, which is bounded by the number of cores in the GRP; in case of inter-SWP communication requests, $NumQ$ will be determined by the number of SWPs that can simultaneously issue a request.

Mem_{RET} depends on timing constraints of memory device operations (e.g. row buffer activation, read, write, precharge). We refer the reader to [51] for a detailed explanation of Mem_{RET}.

3.5.3 From $WCTT$ and MEM_{WCRT} to WCET Computation

3.5.3.1 Computing the WCET Estimation in Isolation

GRPs enable deriving an Upper-Bound Delay (UBD) bounds for every intra- and inter-SWP communication request accessing the NoC and memory. These bounds cam be incorporated in the timing analysis in order to consider the contention in the hardware shared resources with requiring changes in the timing analysis tools. contention that the processes can suffer during deployment time [21].

For communication requests UBD represents the maximum delay a request to NoC and memory resources can suffer due to interferences. The UBD of intra-SWP requests (UBD_{intra}) depends on the $WCTT$ internal to the GRP and Mem_{WCRT}. Equation 3.5 and Equation 3.6 shows the UBD_{intra} for clustered and regular designs. Note that in the case of the mesh, WCTT varies depending on the core, thus UBD also depends on the core.

$$UBD_{intra}^{cluster} = WCTT_{tree} + Mem_{WCRT} \tag{3.5}$$

$$UBD_{intra_{core\_id}}^{regular} = WCTT_{mesh_{core\_id}} + Mem_{WCRT} \tag{3.6}$$

Inter-SWP requests instead, are not only affected by intra- and inter-SWP requests belonging to the same application, but also by inter-SWP requests belonging to other applications. Thus, the UBD of inter-SWP requests (UBD_{inter}) depends on both, the WCTT internal and external of the GRP and Mem_{WCRT}. Equation 3.7 shows the UBD_{inter} for the clustered NoC design, while Equation 3.8 shows it for regular NoC design. It is important to remark that in case of a mesh, inter-SWP requests target the destination SWP whose mapping is unknown and UBD has to take a conservative assumption: the worst-case destination (farthest possible).

$$UBD_{inter}^{cluster} = WCTT_{tree} + WCTT_{bus} + Mem_{WCRT} \tag{3.7}$$

$$UBD_{inter_{core\_id}}^{regular} = WCTT_{mesh_{core\_id}}^{farthest\_dest} + Mem_{WCRT} \tag{3.8}$$

For static timing analysis, the access latency of each request to the NoC and memory is augmented by UBD. If measurement-based timing analysis tools are used, during the testing phase the process under study is run in isolation. Every time an intra-SWP request to memory is ready it is artificially delayed by the architecture so it suffers UBD_{intra}, which can be carried out with a technique called worst-case mode [21]. From the traces obtained during this execution in isolation, a WCET estimate for the task is obtained. At deployment time, the hardware is

instructed not to introduce any artificial delay. The key point of this solution is that the artificial delay introduced during testing upper-bounds the delay a request might suffer.

3.5.3.2 Computing Δ_{inter}: NoC and Memory Impact

At system integration time, the WCET estimate of one application computed in isolation ($WCET_{isolation}$) can be affected by inter-SWP communication as expressed in Equation 3.1. Concretely, transparent execution mechanism makes inter-SWP requests to delay intra-SWP requests because of their higher priority in NoC and memory. Note that the impact that inter-SWP requests may have on other inter-SWP requests coming from other SWP is already considered in the WCTT used to compute the $WCET_{isolation}$.

As described in Section 3.2, IMA systems impose that the amount of data transferred in an inter-SWP communication from the source to the destination SWP is known at system integration time, so the application development becomes independent from the system integration. This allows computing the WCET increment (Δ_{inter}) of an application due to interferences that intra-SWP requests may suffer in NoC and memory due to inter-SWP communication requests.

Δ_{inter} is computed using Equation 3.9 and Equation 3.10, for the clustered and regular architecture designs respectively. In these equations, P is the set of SWPs that can simultaneously send inter-SWP communication requests to the GRP in which the destination SWP runs and N_{inter_i} is the number of inter-SWP communication requests of the source SWP_i. Note that in the clustered architecture intra-SWP requests do not use the bus (see Figure 3.5(a)) so we address only the interference in the memory ($\Delta_{inter}^{clustered}$).

$$\Delta_{inter}^{clustered} = \sum_{i \in P} N_{inter_i} \cdot Mem_{WCRT} \tag{3.9}$$

$$\Delta_{inter}^{regular} = \sum_{i \in P} N_{inter_i} \cdot (WCTT_{mesh} + Mem_{WCRT}) \tag{3.10}$$

3.6 Experimental Results

3.6.1 Experimental Setup

3.6.1.1 Hardware Setup

All experiments presented in this section are executed on a cycle-accurate simulator compatible with PowerPC ISA binaries and based on the *SoCLib* simulation infrastructure [37] and the *gNoCSim* cycle-accurate flit-level NoC simulator [38], as described in Section 2.1. In our simulation framework we model the two 16-core processor architectures presented in Figure 3.5, a clustered and a regular architecture, both implementing 4 GRPs with 4 cores each.

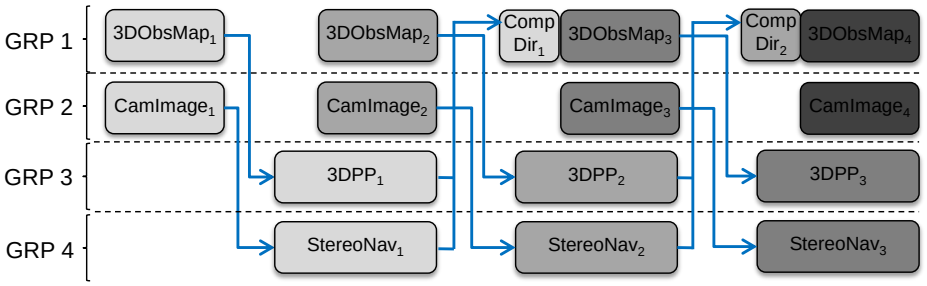


FIGURE 3.7: 4 SWPs executed following a software pipelining approach.

The clustered architecture (see Figure 3.5(a)) comprises a hierarchical NoC, with a tree and a bus as first and second level NoCs. We assume $D_{router} = 1$ and $D_{bus} = 2$. The NoC design fulfills the WCTT shown in Section 3.5.2.1 ($zll_{tree} = 5$; $NoC^{tree}_{RID} = 8$; $zll_{bus} = 8$; $NoC^{bus}_{RID} = 24$). The regular architecture models a 4x4 mesh NoC with two virtual channels (one for each communication type) giving higher priority to the virtual channel used by inter-SWP communication requests. In our experiments, virtual clusters comprise 4 cores each rather than 6, 4, 4, and 2 as presented in Figure 3.5(b) for sake of fair comparison versus regular architecture. Moreover, the mesh implements a wormhole switching policy and stop-and-go flow control. Overall, the two NoC designs fulfill the WCTT computed in Table 3.1.

Finally, the our experimental platform also models separated instruction cache and data write-through cache of 64 KB each in each core and four 256MBx16 DDR2 SDRAM 400B memory controllers, one per GRP, implementing two queues each (one per communication type). Higher priority is given to the queue used by inter-SWP requests. We assume that CPU frequency doubles memory frequency. This configuration provides a $Mem_{WCRT} = 42$ processor cycles [51].

3.6.1.2 Parallel Avionic Applications

pSWPs and GRPs are evaluated using a *real A653-compliant avionic system* provided by Honeywell International. It comprises two parallel avionics applications: 3D Path Planning (*3DPP*) and Stereo Navigation (*StereoNav*), used for the navigation of Unmanned Aerial Vehicles (UAVs). These applications are described in Section 2.2.1.

Moreover, the system also includes two applications for data generation: *3DObsMap* and *CamImage*. The former provides the 3D grid obstacle map required by 3DPP; the latter provides the two images (maps) required by StereoNav. Finally, 3DPP and StereoNav outcomes are compared in *CompDir* application. The communication among applications is performed using inter-SWP communication requests. If there is a mismatch in the computed direction and velocity, the StereoNav application output is the one trusted.

Figure 3.7 shows how all five applications are executed in parallel in a software pipelined manner. The data generated by *3DObsMap* and *CamImage* applications at stage n is stored in the memory of the GRP in which 3DPP and StereoNav execute at the next stage $n + 1$. Then, the output of 3DPP and StereoNav is compared by *CompDir* at stage $n + 2$ (*CompDir* is executed within the same GRP of *3DObsMap* due to its short execution time; the application simply compares

the outcome of 3DPP and StereoNav). Under this scenario, 3DPP is only affected by inter-SWP requests sent by 3DObsMap and StereoNav is only affected by inter-SWP requests sent by CamImage. 3DObsMap and CamImage transmit the data that 3DPP and StereoNav will require in the next pipeline iteration.

The implementation of the five avionic applications fulfills the ARINC 653 APEX API specification [23]. Concretely, parallel tasks are managed as *A653 processes* (i.e. CREATE_PROCESS, START_API); intra-SWP communication uses *A653 buffers* (i.e. CREATE_BUFFER, SEND_BUFFER, RECEIVE_BUFFER); inter-SWP communication uses *A653 queuing ports* (i.e. CREATE_QUEUING_PORT, SEND_QUEUING_MESSAGE,
RECEIVE_QUEUING_MESSAGE); and finally, synchronization mechanisms to share data among processes within a SWP use *A653 semaphores* (i.e. CREATE_SEMAPHORE, WAIT_SEMAPHORE, SIGNAL_SEMAPHORE).

3.6.1.3 Compuitation of the WCET estimation of Parallel Avionic Application

WCET estimates of applications are derived with measurement-based techniques. Concretely, the architecture introduces the *WCET computation mode* [21], in which at analysis time intra-SWP and inter-SWP requests are artificially delayed by an UBD as defined in Equation 3.5, Equation 3.6, Equation 3.7 and Equation 3.8 respectively. By doing so, the resultant execution time can be considered as an Worst-Case Execution Time Bound (WCB). At deployment time, the WCET computation mode is deactivated so a request suffers only actual delays which are bounded by UBD.

3.6.2 Impact of intra-SWP Communication on Execution Time

GRPs prevent intra-SWP activities from being affected by the intra-SWP activities generated by other SWP executed on different GRPs. To illustrate this, we run each application in isolation, i.e. no other application is executed in other GRPs, and we measure its execution time. In a second experiment we run all applications simultaneously as shown in Figure 3.7 with each application mapped into a different GRP. In this second experiment we collect execution times and discount the effect of inter-SWP communications. In both cases the execution times *were exactly the same*, evidencing that the integration of several SWPs can be done without any impact of their intra-SWP activities on other SWPs.

3.6.3 Impact of Inter-SWP Communication on Execution Time

One of the central elements of pSWP specification is transparent execution, which allows considering as an additive factor (Δ_{inter}) the impact that inter-SWP requests have on intra-SWP requests at system integration (see Equation 3.1). This section illustrates that this considerably reduces the WCB on execution times of applications. To that end, we compute the WCB of 3DPP and StereoNav assuming two different strategies:

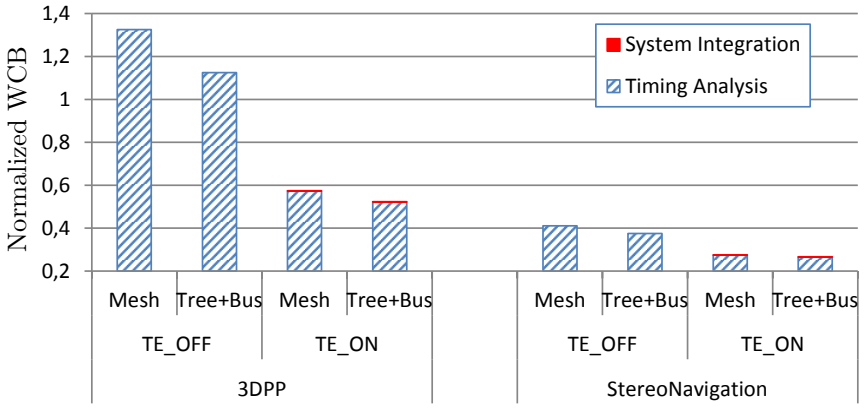


FIGURE 3.8: WCB of 3DPP and StereoNav. We assume a regular (*Mesh*) and a clustered (*Tree+Bus*) architectures. We analyze the impact of activating and deactivating the transparent execution mechanism.

- Assuming that no transparent execution mechanism is implemented and so the WCB includes the effect that inter-SWP communications have on intra-SWP ones and vice-versa [65]. That is, at analysis time, that for every intra-SWP communication it is considered that a potential interference may occur with an inter-SWP communication, and vice-versa.
- Assuming transparent execution in which intra-SWP requests do not compete with inter-SWP requests when computing the WCB. The impact of inter-SWP interference is accounted later at integration time using Equation 3.1.

Figure 3.8 shows the WCB of 3DPP and StereoNav assuming the two strategies presented above, i.e. with and without transparent execution (labeled as TE_ON and TE_OFF respectively), for the two modeled processor architectures shown in Figure 3.5, i.e. clustered (*Tree+Bus*) and regular (*Mesh*). All values are normalized with respect to the sequential execution time of 3DPP and StereoNav in which no interference among intra-SWP and inter-SWP requests occurs. In this case, the complete system executes sequentially in a single core. Figure 3.8 also shows the WCB portion of each application coming from the timing analysis, i.e. $WCET_{isolation}$, (in stripped blue) and the portion of the additive factor (Δ_{inter}) coming from the system integration (in red) when the transparent execution mechanism is used.

We observe that the use of the transparent execution mechanisms reduces considerably the WCB of both applications being executed in both processor architectures. Assuming at analysis time that every intra-SWP request is affected by an inter-SWP request leads to a pessimistic WCET estimate. It is of special interest the 3DPP case, in which not using the mechanism makes the WCB of the parallel version being worse than the sequential version, increasing the WCB by 33% and 13% when executing on a regular and a clustered architecture respectively. This is not the case of StereoNav, in which the WCB of the parallel version is better than the sequential one, reducing it by 59% and 62% when executing on a regular and a clustered architecture respectively.

When applying the transparent execution mechanism, the WCB of both applications is reduced with respect to the sequential execution. In case of 3DPP, the WCB is reduced by 43% and 48%

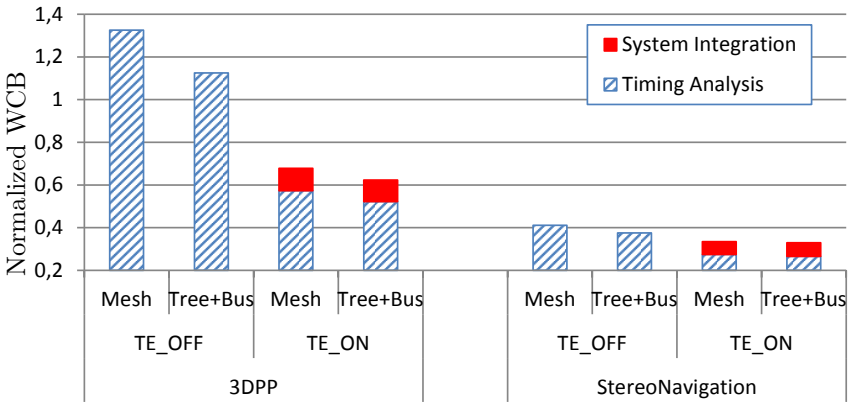


FIGURE 3.9: WCB of 3DPP and StereoNav, assuming regular and hierarchical architectures (*Mesh* and *Tree+Bus* respectively) and activating and deactivating the transparent execution mechanism.

considering a regular and a clustered architecture respectively. In the case of StereoNav the WCB is further reduced by 73% and 74% on a regular and a clustered architecture respectively.

We also observe that the resultant WCB of both applications is tighter for the clustered architecture than for the regular one. The reason is because the WCTT of the hierarchical NoC in the clustered architecture (a tree and a bus) is lower than the WCTT of the regular one (a mesh) as shown in Table 3.1.

The portion of the WCB coming from the additive factor Δ_{inter} represents less than 1% in both applications. This is due to the fact that the number inter-SWP requests that affects 3DPP and StereoNav (coming from 3DObsMap and CamImage respectively) at system integration is relatively small with respect to intra-SWP requests.

With the aim of showing the impact of inter-SWP requests, we repeat the same experiment presented in Figure 3.8 but we artificially increase the number of inter-SWP requests suffered by 3DPP and StereoNav by 100x. The results are shown in Figure 3.9. We observe that the WCB with no transparent execution mechanism (TE_OFF) remains exactly the same. This is so because WCB already accounts for the impact of inter-SWP interferences on intra-SWP requests. In case of using the transparent execution mechanism (TE_ON), the portion of the additive factor in the WCB of both applications increases as well, 10% in case of the 3DPP and 6% in case of StereoNav. However, despite the spectacular increment of inter-SWP requests, the WCB is still significantly lower than when not using the mechanism.

Therefore, we can conclude that accounting for inter-SWP request impact at system integration reduces considerably the WCB estimates of applications.

3.6.4 Executing several SWP into a single GRP

As presented in Section 3.5, GRPs enforce the pSWP specification by, among others, isolating intra-SWP activities from different SWPs. This is obtained by executing one SWP per GRP. However, it may be the case that the application encapsulated in the GRP does not have enough

task level parallelism to exploit all cores in the GRP. This can lead to under-utilization of resources.

In order to improve GRP occupancy in clustered architectures, it is possible to simultaneously run several SWPs in one GRP with certain considerations. In order to maintain pSWP principles, it is required to bound the impact that intra-SWP activities from different SWP may suffer when sharing GRP shared processor resources. Some hardware techniques naturally control this [21, 51, 52] since the WCET for a task is made independent of the load that other tasks (processes) put on the shared resources. That is, the resultant WCET estimate of each SWP is not affected by intra-SWP activities from other SWPs, despite being executed within the same GRP. This of course comes at the cost of more pessimistic WCET estimates. On the other hand, the number of queues for inter-SWP requests in the memory controller has to be increased to the maximum number of SWPs that may be active at any point in time in the architecture. It is worth noting that inter-SWP impact still has an additive nature that can be factored in at analysis time due to the highest priority of inter-SWP requests over intra-SWP requests, despite being within the same GRP.

The SWPs shown in Figure 3.7 benefit from this SWP folding into GRPs. Figure 3.10 shows WCB of four applications under three different execution scenarios in regular architecture.

In Scenario 1, SWPs use all 4 GRP, and the performance bottleneck is *StereoNav*. It takes longer to execute than the other applications, which leads to underutilization of the GRPs in which the other applications run as well as suboptimal performance of our software-pipelined system. As shown in Figure 3.8, *StereoNav* reduces its WCB approximately by 75% (from 1.53s to 0.4s) when it executes in parallel mode enjoying 4 cores in its GRP, with respect to its single-core execution, representing a speed-up of 3.85x. Given that it is the bottleneck application, the speed up of the whole software-pipelined system is 3.85x as well. This does not seems a convenient speed-up for a 16-core architecture.

In Scenario 2, we run *StereoNav* into one GRP and fold all the other SWP into a second GRP, which hence time share the GRP. In this case, the same WCB speed-up of 3.85x is observed. Note that in this case, only 8 cores are used, leaving 2 extra GRPs for executing other parallel applications.

Virtual GRPs naturally provide capability to adapt to the task level parallelism of the SWP. To illustrate this point, in Scenario 3, on the regular (mesh-based) architecture, we increase the number of cores assigned to a GRP from 4 to 6 to *StereoNav*, and so leaving only 2 cores for the virtual GRP executing the other 3 applications. In this case, we observed a WCB speed-up of 4.9x of the*StereoNav* w.r.t its single-core version.

3.7 Related Work

For current Commercial-Off-The-Shelf (COTS) multicores, it is difficult to provide time composable WCET by default. In this regard, authors in [66][67] quantify the delay suffered due to interferences in shared processor resources of a COTS multicore. Fuchsen [68] performed a

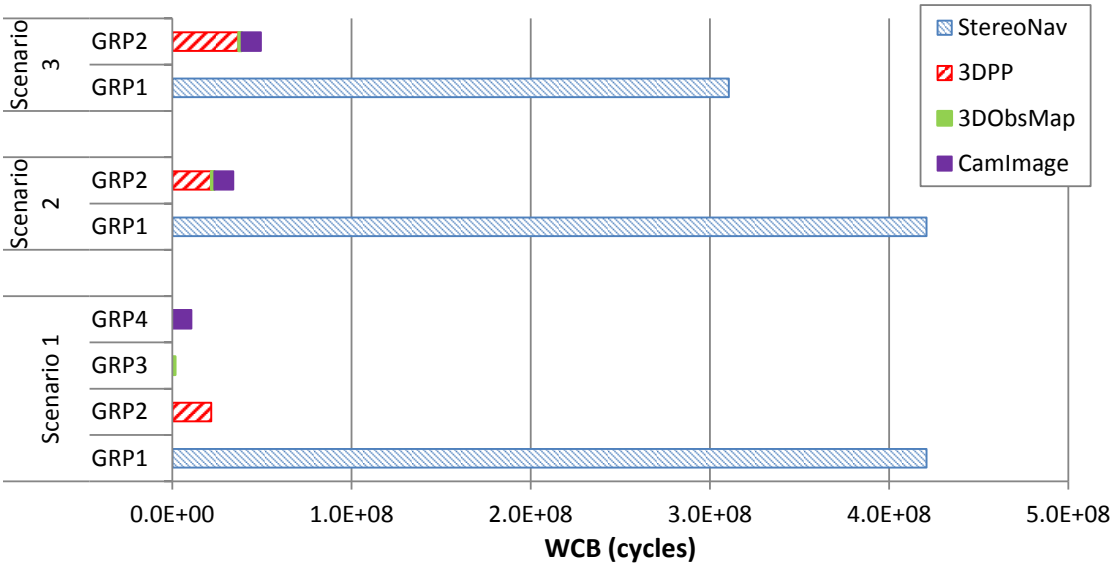


FIGURE 3.10: Combining several pSWPs into a single GRP

similar analysis, but focusing on the hardware and software related interference channels between SWPs in multi-core based IMA platforms.

The literature in the area of analysis of hardware shared resource contention in multicore is vast. At system level, several analysis frameworks have been developed to compute worst-case access time bounds [69–72]. These frameworks model one off-chip shared resource that can process only one request at a time and in which requests cannot be split. It is assumed that on-chip shared resources (e.g. core-to-cache bus, caches, ...) are replicated or partitioned across tasks. This makes that tasks suffer no contention accessing on-chip resources. Further it is assumed that the accesses to the off-chip shared resource are synchronous (i.e. the accessing task is stalled while the access is performed). The focus is on a specific task model in which tasks are divided into superblocks for which maximum and minimum access bounds and execution time bounds can be derived.

Under this scenario, the access to the shared resource is assumed to be arbitrated by either a TDMA bus [69], a dynamic arbitration bus [70] or an adaptive bus arbiter [71]. For those cases in which the arbiter is dynamic, the load that a task puts on the shared resource affects other tasks access time. Other authors [70, 72] propose different approaches to derive per-task bounds to the number of accesses in a period of time. While the number of accesses that a task generates to the resource can be considered intrinsic to the task (i.e independent of the co-runners) as long as caches are partitioned, its frequency of access depends on how often the co-runners delay the task requests, dependence that is captured by the presented models. With dynamic arbiters time-composability cannot be guaranteed since the WCET derived for a task depends on its co-runner tasks. Time-composability is of paramount importance to enable incremental verification as required by ARINC 653, and hence it is the objective of our designs, preventing us from using dynamic arbiters. Further, we focus on shared on-chip resources and the main memory, which handle multiple requests and naturally split cache misses (the requests) into several memory commands that are parallelized across requests, preventing us from using the

analysis frameworks presented above. Finally, we use real unmodified avionics applications, which do not follow the superblock model.

Several efforts coming from the WCET community have focused on providing combined (i.e. multitask) WCET estimates for tasks sharing a bus and a cache [53, 54]. This can be used at the process level (i.e. among the processes of a given application) as shown in Figure 3.2(b). Applying it across applications would break time composability and hence incremental verification.

At hardware level, we identified two main approaches to deal with contention [73]. The first approach relies on designing a custom platform targeting a specific application with timing constraints, as it is the case for the time-triggered [74], the PRET [75], and the CompSOC [76] architectures. The second approach, although it still requires hardware changes, focuses on adapting general-purpose platforms to allow the execution of those applications with timing constraints. This is the case of the MERASA [65] approach. We note that some special features included in the application-specific architectures like scratchpads require modifications in the application's code, challenging portability of legacy applications. Further, the growing cost of developing and manufacturing chips makes the use of application-specific architectures only relevant for high volume products [73], which is not typically the case for the avionics domain. Therefore, we have used as baseline the MERASA architecture, on top of which we implemented the novel concept of transparent execution.

3.8 Conclusions

Current ARINC 653 time partitioning techniques are not suitable for many-cores as they fail in isolating IMA Software Partitions (SWPs): the execution of multiple traditional SWPs in the many-core affects the timing behavior of each other due to uncontrolled simultaneous access to shared hardware resources.

In this chapter, we have introduced the concept of Parallel Software Partitions (pSWPs) as an extension to ARINC 653 standard(objectives **O2** and **O5**). pSWPs specify how intra-SWP and inter-SWP interferences are controlled to isolate the timing behavior of SWPs. In particular, pSWPs prevent local intra-SWP activities from affecting (or being affected by) remote intra-SWP activity. pSWPs require hardware support so that the impact of inter-SWP activities is made *additive*. By doing so at integration time different SWPs can be independently developed, time analyzed and brought together to form a system with minimum effort. pSWPs rely on a new hardware feature called Guaranteed Resource Partitions (GRPs) (objective **O1**). GRP defines an execution environment composed of a cluster of processor resources in which SWPs run, providing the desirable timing isolation and time predictability properties among intra-SWP activities and making inter-SWP activities to have an additive nature. This is done by implementing transparent execution mechanism that freezes intra-SWP and local GRP requests to let inter-SWP requests proceed, allowing to consider the impact of inter-SWP communication at system integration time as required by pSWP (Δ_{inter} addend).

We evaluate two many-core processor architectures that implement GRPs with a *hierarchical* and a *regular* NoC designs with a real avionic system composed of two parallel applications provided

by Honeywell, 3D path planning and stereo navigation (objective **O4**). We show that transparent execution enables compositional timing analysis improving application performance by 43% and 48% in case of 3DPP for hierarchical and regular architectures respectively (objective **O3**). In case of StereoNav, the performance improvements go up to 74%. The time overhead due to inter-SWP communications is reduced to less than 1% for both applications. Furthermore, we show the benefit of flexibility in defining GRPs in mesh-based architectures, and speed-up the entire system by 4.9x.

Chapter 4

Modeling High-Performance Wormhole NoCs for Critical Real-Time Embedded Systems

Manycore chips are a promising computing platform to cope with the increasing performance needs of Critical Real-Time Embedded Systems (CRTES). As highlighted in the previous chapter, manycores adoption by CRTES industry requires understanding task's timing behavior when their requests use manycore's Network on Chip (NoC) to access hardware shared resources.

As thesis focuses on closing the gap among high-performance and CRTES domain, this chapter analyzes the contention in Wormhole-based Network on Chip (wNoC) designs – widely implemented in the high-performance domain – for which we introduce a new metric: Worst-Contention Delay (WCD) that captures wNoC impact on Worst-Case Execution Time (WCET) in a tighter manner than the existing metric, Worst-Case Traversal Time (WCTT). Moreover, we provide an analytical model of the WCD that requests can suffer in a wNoC and we validate it against wNoC designs resembling those in the Tilera-Gx36 and the Intel-SCC 48-core processors (objectives **O1** and **O3**). Building on top of our WCD analytical model, we analyze the impact on WCD that different design parameters such as the number of virtual channels, and we make a set of recommendations on which wNoC setups to use in the context of CRTES (objective **O1**).

4.1 Introduction

Manycore chips can accommodate high task counts in a single hardware device which helps reducing size, weight and power costs in CRTES. The deployment of manycores as baseline computing platform in CRTES requires a means for the safe consolidation of multiple CRTES applications on the same chip. In that respect, one of the stumbling blocks in the manycore adoption in CRTES is understanding how manycore internal complexity affects tasks' timing behavior. The interconnection network is, arguably, one of the manycore shared resources with

highest impact on timing. Unlike multicores, which use a centralized interconnect (e.g. a bus) to access hardware shared resources, manycores implement Network on Chip (NoC). In NoCs, requests are arbitrated in a distributed manner at various routers severely complicating timing analysis.

In the high-performance domain, Wormhole-based Network on Chips (wNoCs) are well understood [77][78] and used in several Commercial-Off-The-Shelf (COTS) products [60][18][79]. The high-throughput and low-hardware cost features of wNoCs make them attractive for CRTES as an alternative to real-time customized networks whose adoption in commercial products is harder to achieve. In this respect, we address the problem of whether high-performance wNoC designs can be used to consolidate, in a trustworthy manner, multiple CRTES applications into a single manycore. This requires providing high-performance and isolation among tasks so that time composable WCET estimates [17] (independent of the load that co-running tasks put on the NoC) can be derived.

We take time composability as a premise in CRTES design since it enables two fundamental properties to reduce system development and deployment costs: incremental development and incremental verification of integrated systems (e.g. Integrated Modular Avionics (IMA) [25, 27], AUTomotive Open System ARchitecture (AUTOSAR) [24]). During system development, the ability to incrementally integrate applications without the need of regression tests to validate the timing properties of already-integrated applications heavily reduces integration costs. At system deployment, the ability to update functions and their associated software, without the need for re-analyzing and re-certifying the system, is vital in domains like space where systems operate during dozens of years and whose functionality is usually updated once deployed.

In this chapter we propose an analytical model that captures the impact of the different wNoC design choices and parameters on WCET estimates. Our goal is to adhere to existing COTS wNoC designs without the need of adding extra hardware support. In particular, we make the following contributions:

- We introduce Worst-Contention Delay (WCD) as a new metric to accurately capture the impact of wNoC inter-task interferences on WCET estimates (Section 4.3). WCD takes into account the pipelined behavior of wNoCs, leading to tighter WCET estimates compared to the ones obtained with the Worst-Case Traversal Time (WCTT) [3][80][81][82] (objective **O3**).
- We provide a taxonomy of wNoC design parameters (Section 4.4), identifying those that have to be fixed in order to provide trustworthy and composable WCD bounds; and those where some flexibility is allowed. We show that the default values for some of the latter set of parameters are configured to improve average performance, increasing WCD bounds (objective **O1**).
- We derive an analytical model for time-composable WCD bounds in a wNoC (Section 4.5), covering a vast set of parameters including flits-per-packet, number of virtual channels

and queue size in the router[1]. Our model achieves high coverage of existing COTS high-performance wNoC designs. We discuss static virtual channel allocation and show that it has to be applied smartly to reduce WCD bounds. Otherwise, it can result in an increase in WCD bounds, e.g, using virtual channels to separate the traffic generated by applications under different criticality levels increases WCD bounds and hence WCET estimates. Further, in Section 4.6, we discuss the impact of various wNoC parameters on system design.

- We assess the accuracy of our analytical model on two wNoC setups resembling the ones deployed in real processors (objective **O1**): the Tilera-Gx36 [18] and the 48-core Intel SCC [60] (Section 4.7). In all cases, our WCD estimates tightly upperbound the measured contention delay values with up to 5% over-estimation on average. Further, we show that on average, WCD bounds are 2.7x and 2.94x lower than WCTT bounds for the Tilera-Gx36 and the Intel SCC setup respectively.

Overall, our analysis shows that simple but effective design and configuration choices make efficient use of wNoCs in CRTES possible.

4.2 Background

In CRTES, there are two main ways to handle contention among accesses to shared hardware resources, including NoCs, as explained in Section 1.3.1. First, the NoC contention is accounted as part of the WCET estimation process by deriving a time composable bound of the Worst-Case Traversal Time (WCTT). WCTT defines the longest time a request could take since the moment it is injected in the NoC by a source node until it is delivered to the destination node. Alternatively, NoC contention delay that a task suffers can be handled as part of the Worst-Case Response Time (WCRT) analysis by adding to the task's execution time in isolation the contention generated by the flows of its co-running tasks – which are assumed known at this stage.

Each approach has its own pros and cons: while the latter enables deriving tighter estimates, since it builds upon the knowledge of interference generated by the tasks in the observed task set, it violates time composability. The former, which is the focus of this chapter, maintains time composability at the expense of higher WCET estimates.

In Section 4.3 we propose the use of Worst-Contention Delay (WCD) instead of WCTT as a means to provide tighter WCET estimates for the tasks running in the wNoC based manycore processor.

[1]We consider arbitration, routing and virtual-channel allocation policies, to be configurable from software similarly to the way cache replacement is currently adjustable in high-performance architectures. This is in contrast to hardware proposals that require global changes like, new signals among routers and nodes, different flow-control, global clocks or the like.

4.3 Contention Delay: A New Metric to account for the impact of NoC on WCET

Given a task under analysis, we call Contention Delay (CD) the delay caused by the *other* co-running tasks in the access to the shared NoC. As an alternative to WCTT we introduce a new metric, called Worst-Contention Delay (WCD), that captures in a tight manner the impact that accesses to the NoC have on programs' execution time and WCET. WCD stands for the highest impact that a request may have on WCET due to contention in the NoC. It stems from the appreciation that requests can suffer two types of delays: intra-task delay (*atd*) that is caused among requests coming from the same core; and inter-task delay (*etd*) that covers the delay that one request from a core causes on the request of a different core.

We illustrate the difference between WCD and WCTT with the example in Figure 4.1. Figure 4.1 (a) shows a simple NoC connecting three cores, out of which one is the destination core (c_2). Our focus is determining the delays suffered by the requests from core c_0 when accessing the NoC. An arbiter, which implements round-robin policy, handles requests coming from c_0 and c_1, with separate buffers to handle the requests of c_0 and c_1.

The time diagram in Figure 4.1 (b) shows the actual traversal time and the actual contention delay suffered by subsequent requests r_0^0 - r_0^4 sent from c_0 (upper time diagram) and r_1^0 - r_1^4 sent from c_1 (lower time diagram with grey background) when cores inject packets at the maximum rate. In absence of interference, we assume that a request takes 1 cycle to traverse the router, and that buffers can store up to 2 requests. In the time diagram, I stands for the cycle when the request is transferred from c_0 to the buffer and C_X the cycle when the request is sent from the router (eXits) to the target node c_2. Ba corresponds to cycles when the request is in the buffer but not at the top, hence suffering *atd*. Likewise, Be corresponds to cycles when the request is at the top of the buffer and hence suffering *etd* since the arbiter grants access to c_1.

We assume that first request of c_0 lost the arbitration in Cycle 1 so that it is delayed by a request of c_1 in traversing the router. We observe that request r_0^0 only suffers *etd*. r_0^1 enters in the second entry of the buffer in cycle 1 (cyc_1), in cyc_2 it suffers *atd* and reaches the top of the buffer in cyc_3 where it suffers a cycle of *etd*. We observe a similar behavior for other requests, with the difference that when there are two requests in the buffer, c_0 has to wait until one is released before sending another request.

The two columns on the right of the time diagram show the interference cycles for traversal time (cTT) and for contention delay (cCD) metrics. For the traversal time the interference cycles suffered by r_0^0 - r_0^4 are $2, 3, 4, 4, 4$ respectively. Meanwhile for the contention delay they are $1, 1, 1, 1, 1$. The key appreciation is that with traversal time the interference accounted for each request covers both *atd* (Ba) and *etd* (Be). *However, the impact of Ba for one request is already accounted as part of the Be of another request.* For instance, the *atd* suffered by r_0^2 in cyc_3 is the *etd* suffered by r_0^1. Overall, the main problem with traversal time is that it doesn't capture well the pipelined behavior of the NoC and that the same cycle can be accounted several times as contention, either intra- or inter-task, in different requests. Contention Delay instead, only focuses on inter-task contention and does not over-account *atd* and *etd* cycles.

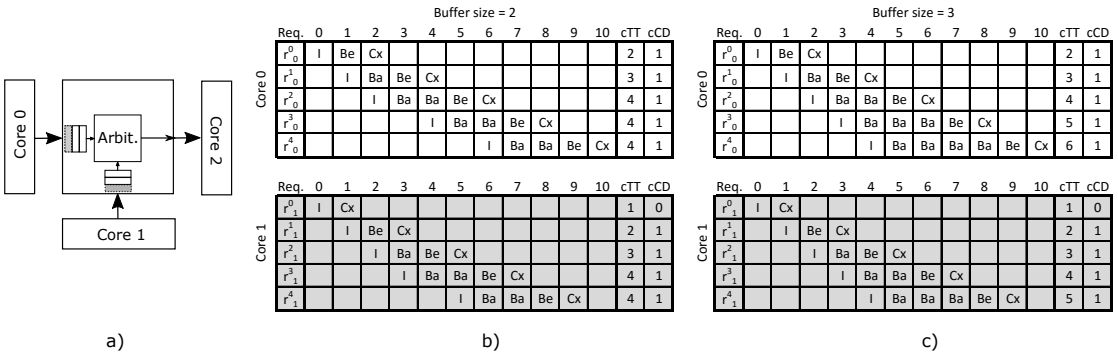


FIGURE 4.1: Simple router and the impact of *atd* and *etd* on traversal time and contention delay.

Therefore, considering WCTT as the extra delay that each request suffers due to contention in the NoC can be (very) pessimistic. Instead, WCD provides tighter estimates since it prevents requests to account for multiple *atd* interference delays as shown in Figure 4.1. Our results in Section 4.7 show that for a wide range of NoCs WCD tightly and trustfully upperbounds the impact of inter-task interferences in wNoCs.

4.3.1 WCD Properties

WCD shows a few interesting properties:

- Larger buffers increase the *atd* since a request is potentially delayed by higher number of requests coming from the same core. This translates on the fact that larger buffers lead to higher WCTT. This is counterintuitive since the more the resources of the wNoC, in the form of larger buffers, the worse the WCTT is. Since WCD is not affected by *atd* the impact of buffer size on WCD is reduced, which enables use of wNoCs with larger buffers in real-time domain with benefits for average performance of the wNoC.

  Time diagrams in Figure 4.1 (c) show the behavior or previously discussed sequences of requests from cores c_0 and c_1, if we increase the sizes of input buffers in the router to 3. This change doesn't affect execution time of the sequence nor contention delay. However, it shows an increase in traversal time for some requests, e.g. for r_0^4 it grows from 4 to 6.

- Solutions based on limiting the injection rate at hardware [80] or software-level [83] effectively reduce *atd* since requests are injected into the wNoC at a lower rate and so fewer conflicts occurs. In the extreme case, these technique can prevent flows from having more than one packet in any single router, completely removing the *atd* but jeopardizing the wNoC utilization. While this reduces the problem of *atd* accounting for WCTT, WCD completely removes *atd* providing tighter bounds, without any impact on average performance.

- WCD leads to tighter WCET estimates than WCTT since the *atd* a request suffers occurs in parallel to the *etd* for other requests, which is already captured by WCD.

TABLE 4.1: Summary of main symbols used

Symbol	Description
VC	Virtual Channel
$\overline{WCD}$	Time-composable upper bound to contention delay
F_i	Packet stream traversing the same source-destination route and requiring the same grade of service along the path.
H_i	Number of hops in a flow F_i
R_i^j	Router (hop) j in a flow F_i (see Figure 4.2 (a))
r_i^k	Packet (request) k in a flow F_i
L^{flits}	Number of flits of a packet
P_i^j	Number of ports in router R_i^j
NR_i^j	Number of queues that can potentially contend for an output port that F_i is targeting at R_i^j
$\omega(i,j)$	Function that returns the index x of the worst-case destination flow F_x that starts at the hop R_i^{j+1} and reaches the worst-case destination in terms of indirect blocking of packets of flow F_i

4.3.2 WCD Assumptions

WCD applies to processors free of timing anomalies such that increasing the local delay suffered by any request leads to an increase in execution time by at most the magnitude of the local delay. In particular, by increasing contention delay by d cycles, execution time grows by up to d cycles.

Further, WCD applies to network designs implementing back pressure flow-control policies i.e. NoC designs with no packet loss such as wormhole and virtual cut-through [77]. WCD works for work-conserving policies such as round-robin so that links are never left idle if there are pending requests.

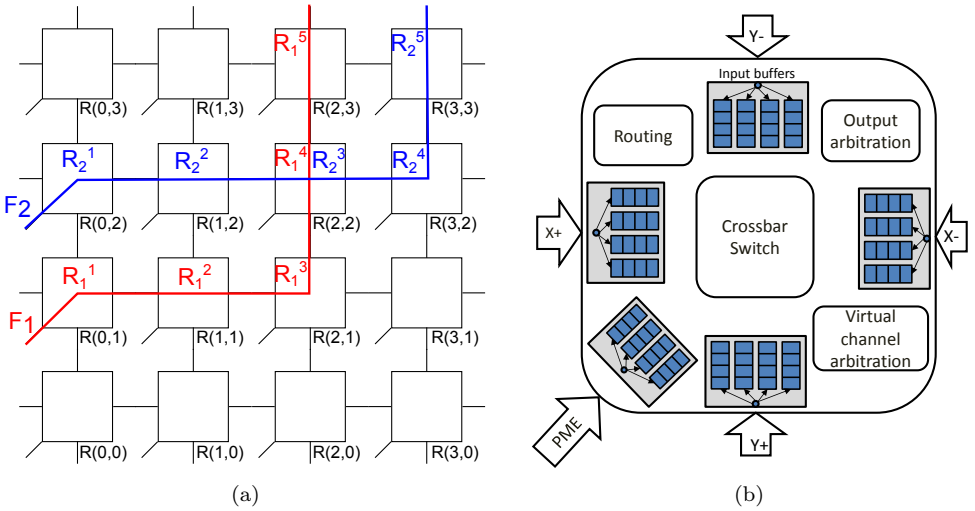
4.4 NoC Parameters Taxonomy

This section presents a taxonomy of wNoC parameters. We consider a mesh network topology as it is the most common topology used in wNoCs, though the analytical model presented in this chapter also applies to other network topologies (e.g. torus) by simply varying some parameters such as the number of ports per node. Table 4.1 lists the symbols and the corresponding description used in the rest of the chapter. We distinguish between the WCD, which corresponds to the actual worst contention delay a NoC request may suffer due to interferences with other requests, and the $\overline{WCD}$, which corresponds to an upper-bound of the actual WCD derived by our analytical model.

4.4.1 Wormhole mesh NoC fundamentals

In our reference *NxM* mesh wNoC configuration, depicted in Figure 4.2 (a), each node comprises a Processor/Memory Element (PME) and a router that communicates with the rest of nodes. The PME can be either a processor core, a cache memory, main memory, I/O, etc. In the network several traffic flows (F_i) may be active at the same time. Each node can be identified using (x, y) coordinates. The router located at coordinates (x, y) is referred to as $R(x, y)$.

Routing decides the path that a packet follows within the network, and consequently, the number of routers or *hops* (h), a given flow requires to move from a source to a destination node. Hence,



(a) (b)

FIGURE 4.2: Mesh basics. (a) Router coordinates in a 4x4 part of a mesh. (b) Canonical 2D-mesh router.

a router can also be identified as R_i^j, in which j represents the hop j of flow F_i, when moving to its destination.

Communication flows comprise multiple NoC packets or requests. We refer to the k-th packet generated by flow F_i as r_i^k. A packet is the minimum arbitration unit in the network and it can be split into one or several *flits* (short of control flow units). Flits can be further decomposed into smaller units called phits when available link wires cannot accommodate an entire flit. For the sake of clarity in the equations we consider equal phit and flit sizes. The first flit of a packet is called header flit and contains the information required to forward the packet to the destination.

We consider a canonical 5-port[2] 2D mesh router architecture in which input ports have a queue to store packet flits (see Figure 4.2(b)). In order to alleviate the contention caused by flows going to different destinations, high performance NoCs multiplex physical channels using Virtual Channel (VC). To do so, an input queue resource per port is assigned to a Virtual Channel. In a canonical wormhole router with Virtual Channels, two rounds of arbitration are performed. The first selects the input port that is granted access to a given output port and the second one selects the Virtual Channel (queue) that is selected in a given input port. In the case of a router without Virtual Channels the latter arbitration is not required.

Routers are usually pipelined in multiple stages, e.g. the Intel SCC [60] comprises routers with an input buffer, routing of the header flit, switch allocation and link traversal stages. The header flit is the only one arbitrated from a packet and once it is granted access to a given output port this connection remains established until the entire packet leaves the router.

When a header flit arrives at an input port of R_i^j, this flit is stored in the corresponding queue. Next, the routing determines the next hop router, R_i^{j+1}, and the router allocates an entry queue in R_i^{j+1}. Once the router in the next hop can accept the header flit, it competes for an output

[2]In order to simplify our formulation we assume that all routers, including those at the edges, have the same number of ports, which in our case is 5. We could consider that some routers (e.g. those at the edges) may have fewer ports, which would decrease the WCD. However, for the sake of clarity and due to space limitations to extend equations we stick to 5-port routers for meshes as the ones used in Tilera chip [18].

TABLE 4.2: List of wNoC main features analyzed

ID	Feature	Comment
-	Routing	Fixed to achieve time analyzability. Static (e.g. XY routing)
-	Flow control	No impact. Credit-based or stall-and-go.
-	Arbitration	Fixed to achieve time analyzability
-	Switching	Fixed to wormhole (widely implemented).
cF	Number of flows	Limited by static routing.
nVC	No. of queues per input port	$nVC = 1$ or $(1 < nVC < cF)$
E	Entries per queue	< 1 packet, $= 1$ packet or > 1 packet
S	Packet size	Single or Multiple
FT	Flits per Packet	$= 1$ or > 1

port and traverses the router crossbar. Once a header flit is granted access to a given output port, the remaining flits of the packet are forwarded to this port without any further arbitration.

However, contention may cause the header flit to be stalled. When this happens, the remaining flits of the packet are also stalled. One of the causes of stalls is the finite size of queues in input ports. In wNoCs, the minimum allowable queue size is one flit. In any case, queues are typically sized with enough space to avoid bubbles in the packet transmission. For instance, if the time required to know if there is enough space in the next router queue is equal to one cycle, the queue needs to have a minimum size of two flits to avoid bubbles. The latency experienced by a packet to traverse the network from source to destination in the absence of contention is usually referred to as zero load latency (zll).

4.4.2 Proposed Taxonomy

The main properties the wNoC needs to provide in order to be used in real-time systems are (i) time analyzability, i.e. enabling the derivation of as tight as possible contention delay bounds, and (ii) time composability, i.e. making contention delay bounds independent of the load that co-runners put on the wNoC. This translates into deriving the trustworthy upper-bound to the highest possible contention delay ($\overline{WCD_i}$) a communication flow F_i of a given task can suffer due to conflicts with other task's flows. In the following we show how different wNoC parameters impact $\overline{WCD_i}$. Table 4.2 summarizes the wNoC features we analyze.

4.4.2.1 Fixed parameters

Some wNoC parameters are usually constrained to enable time analyzability and composability:

- *Routing* determines the flows that potentially contend with F_i at a given router. Deterministic routing is shown to provide time analyzability [3]. Hence, for our mesh analysis we use XY as it is the preferred solution for routing in regular NoCs due to its low implementation cost. With XY routing packets are forced to use the X dimension first: In the X dimension the position of the target node with respect to the source node determines whether to go right (X+) or left (X-) direction. The same approach is used for the Y dimension. Once a packet is routed using the Y dimension, it cannot be forwarded back to the X dimension. These routing restrictions determine the maximum number of flows contending with F_i at a

given router for an output port. Note that the opposite port of a given input/output port is represented as $\bar{Y}$ and $\bar{X}$.

Note that our analysis can be extended for any other deterministic routing policy. In order to do so, one should recompute the maximum number of flows contending with F_i at a given router for an output port according to that particular routing policy.

- *Flow control* determines how packets traverse the routers. In the context of wormhole switching, back pressure flow control can be based either on the use of credits or stall-&-go signals. In this thesis, we provide expressions assuming the most common case that the flow control mechanism is designed in such a way that no bubbles occur in the packet transmission. However, the impact of bubbles on contention delay can be easily accounted for by considering that a bubble in the transmission is equivalent to increasing packet size by one flit.

- *Active nodes.* In order to achieve time composable contention delay bounds, no assumptions can be made on the particular active flows in the wNoC. That is, it is assumed that any node in the network is entitled to send and receive packets from any other node.

- *Active flows.* Similarly, when computing the contention delay for a packet, we assume that, by the time it is injected in the network, any other potential contending flow is also active at that moment, transmitting its packets in a way that it produces the worst-case contention scenario. In order to reproduce the worst-case contention scenario we need to consider the *worst direct contention* and the *worst indirect contention* [84]. The former can be easily reproduced by considering that for a packet r_i^k of F_i at every hop, all possible contenders (i.e. all queues that can forward a packet to the requested output port) are also requesting the same output port. The latter is caused by packets of flows not sharing the path with F_i but blocking at least one flow that does share at least one link in the path with F_i. In the following sections we derive expressions that account for worst-case contention considering the impact of both indirect and direct contention.

- *Output port arbitration.* Likewise, packets contending in a router for a given output port are arbitrated using a time-analyzable policy. Regular wormhole-based mesh designs like the ones in [18][60] use round-robin arbitration. The use of round-robin arbitration enables the computation of timing guarantees[85][86]. In the case of wNoCs with virtual channels an additional round-robin arbitration is also implemented to select the channel that contends for the output port resources.

4.4.2.2 Parameters to adjust

Other wNoC parameters have some flexibility in the values they can take, though each set of parameters (network parameter configuration) leads to different contention delay. We study the following set of parameters: buffer capabilities and number of flits per packet.

The *buffering capabilities* of the wNoC are further shaped by two parameters:

Table 4.3: Setups

	Setup	Description
Impact of VC	$(FT = 1, nVC = 1, E = 1)$	1 queue per input port, 1-flit packets and input queue holds 1 packet
	$(FT = 1, 1 < nVC < cF, E = 1)$	nVC queues per input port, 1-flit packets and input queue holds 1 packet
Impact of FT and E	$(FT > 1, 1 < nVC < cF, E = 1)$	Input queue holds 1 packet that is multi flit
	$(FT > 1, 1 < nVC < cF, E > 1)$	Input queue holds more more than 1 packet that is multi flit
	$(FT > 1, 1 < nVC < cF, E < 1)$	Input queues cannot store entire an entire packet that is multi flit

- The number of queues per input port – which matches the number of virtual channels – (nVC). nVC leads to two scenarios: First, when there is a single queue per input port, i.e. no virtual channel is implemented, which is referred to as $nVC = 1$; second, there are several queues each of which can – statically or dynamically – hold different flows $(1 < nVC < cF)$.
- The number of entries per queue (E) that can get three values: it can have the exact size of a packet, given by its number of flits $(E = 1)$, be smaller than packet size $(E < 1)$ and be larger than packet size $(E > 1)$.

For the number of *flits per packet* we consider two cases: Each packet comprises a single flit $(FT = 1)$ and each packet comprises several flits $(FT > 1)$.

4.5 Time-Composable WCD bounds

This section provides an analytical model for time-composable bounds to $\overline{WCD}$ with the ultimate goal of deriving time-composable bounds to tasks execution time. Table 4.3 presents the different scenarios we analyze in coming sections. In doing so, we proceed incrementally analyzing the contention delay affecting packets in the NoC, going from simple scenarios to more complex and realistic ones.

4.5.1 Single-Flit, One Virtual-Channel, Single-entry Queue ($FT = 1, nVC = 1, E = 1$)

In this scenario, packets are composed of one single flit and every router input port has a single-entry queue (no virtual channel is implemented). The queue stores the requests coming from all flows sharing it.

4.5.1.1 Single-router traversal

Let us assume a request r_i^1 from a flow F_i going from a source node R_i^1 to a destination node R_i^2 that are adjacent in the direction of F_i. In this wNoC setup, traversing one router is similar to traversing a bus with round robin arbitration policy [85]. The worst contention delay that r_i^1 can experience is:

$$\overline{WCD_i} = (NR_i^1 - 1) \times L_{router} \tag{4.1}$$

NR_i^j is the number of queues contending for an output port that F_i is targeting at router R_i^j. With XY routing if the destination port is $X+$ or $X-$, the number of contending queues is 2 (PME and $\bar{X}$). If the destination port is $Y+$ or $Y-$ (or the PME) the contending queues are 4: $X+, X-, \bar{Y}$ and PME (or $X+, X-, Y+$ and $Y-$). L_{router} represents the time a packet requires to cross a non-pipelined router. In the case of pipelined routers, the pipeline mitigates the impact of L_{router} and it can be safely assumed $L_{router} = 1$. In the rest of the chapter, we make this assumption for the sake of clarity and readability.

4.5.1.2 Worst Contention

Contention is caused by packets of any flow partially sharing the path with F_i. Let's assume F_i traverses R_i^1-R_i^2-R_i^3. When r_i^1 is issued from the PME it enters the arbitration in R_i^1. In the highest contention scenario $NR_i^1 - 1$ requests with higher priority than r_i^1 are ready in R_i^1 per-input-port queue targeting the same output port. For any of these requests to go through R_i^1 the corresponding input queue in R_i^2 input port has to be free. In the worst-case situation, the target input port already contains a packet (r_j^k) from a different flow, F_j, and r_j^k shares the same path as r_i^1. Further, all packets in different input ports in R_i^2 can target the same output port as r_j^k and have higher priority than r_j^k. Overall, we observe that r_j^k causes contention on r_i^1 despite not contending within the router R_i^1 as they share at least one link in the path (direct contention). Additionally, requests not sharing the path with r_i^1 can be blocking r_j^k which in turns causes contention in r_1^1. This contention is usually regarded as indirect contention [84].

4.5.1.3 worst-case Destination

From the previous discussion it follows that the route followed by r_j^k determines the contention it suffers, so the more hops r_j^k traverses the more the contention it may suffer, which in turns affects the contention on r_i^1. In order to account for the worst contention, considering both indirect and direct contention that any F_i can suffer at hop R_i^j, we introduce the concept of *worst-case destination flow*, $F_{\omega(i,j)}$. $F_{\omega(i,j)}$ considers the next hop's input port that a packet of flow F_i targets from current hop R_i^j. The destination of flow $F_{\omega(i,j)}$ is chosen in such a way that causes worst indirect contention to the packets of flow F_i, i.e. it prevents packets of F_i cross the hop R_i^j for the longest time possible.

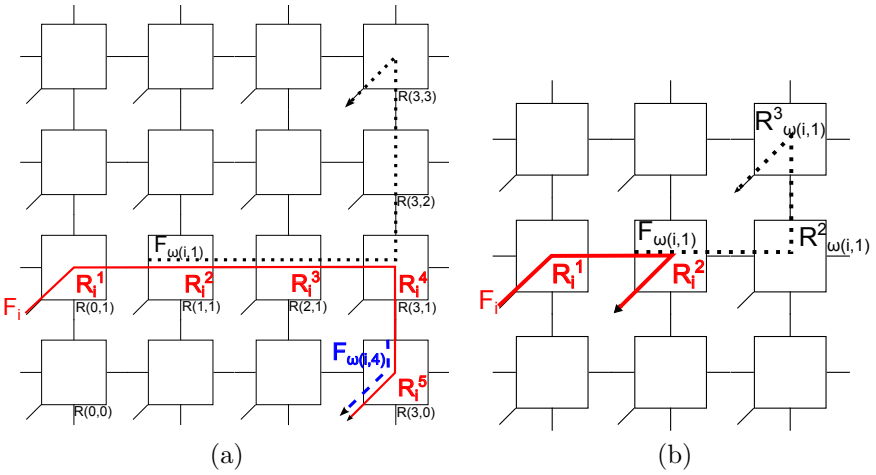


FIGURE 4.3: (a) Worst destination; and (b) A flow crossing 2 routers

The choice of $F_{\omega(i,j)}$ depends on the routing algorithm used. In the wNoC mesh considered in this thesis, with XY routing, the worst destination of flow $F_{\omega(i,j)}$ corresponds to the farthest node that can be reached from the next F_i hop's input port[3], depending on the traversing direction:

- When the packets of flow F_i traverse the Y dimension, the farthest reachable node is the one with highest hop count in the same direction, since once a packet starts using the Y direction, it cannot be routed on the X direction again.
- When the packets of flow F_i traverse the X dimension, the farthest node is the one with highest hop count in X and then in Y dimension.

Let's consider the case from Figure 4.3(a) in which a packet r_i of flow F_i is transmitted from router $R(0,1)$ to router $R(3,0)$ (represented with a solid arrow in the figure). When the packet enters in R_i^1, it waits for an input port to become ready in R_i^2. r_i^1 suffers the longest contention when the packets in the west input port of R_i^2 target their worst-case destination, i.e router $R(3,3)$ (represented with a flow $F_{\omega(i,1)}$, dotted arrow in the figure). The same worst-case destination is maintained as the packet traverses $R(1,1)$ and $R(2,1)$. However, when r_i reaches router $R(3,1)$ as the packet requests the north input port in router $R(3,0)$, the worst-case destination changes. The reason is because $F_{\omega(i,1)}$ considers Y+ but r_i goes Y-. As a result, a new worst-case destination flow is computed, i.e. $F_{\omega(i,4)}$, marked with a dashed arrow in Figure 4.3(a).

4.5.1.4 Computing the time-composable upper bound Worst-Contention Delay ($\overline{WCD_i}$)

In order to derive the general $\overline{WCD_i}$ expression, we first focus on the case of a flow crossing only two routers as shown in Figure 4.3(b). Flow F_i targets next hop's (R_i^2) PME. In order to cross router R_i^1, it has to win the arbitration for the east output port of R_i^1. To keep time composability we assume that it has the lowest priority in that arbitration, as shown in Equation 4.1. We further assume that each of its contenders in R_i^1 suffers the worst contention from $F_{\omega(i,1)}$ (marked

[3]This assumption is only valid in the case all routers have the same number of ports. In other cases the worst-case destination is computed iterating contending flows to the possible destination and selecting the one causing the highest contention.

with a dotted arrow in the figure), which determines the delay suffered by each contender in the arbitration.

In order to compute the impact that $F_{\omega(i,1)}$ has on contenders of F_i, we follow an iterative process, assuming that $F_{\omega(i,1)}$ also suffers the worst contention at each hop until reaching its destination $R^3_{\omega(i,1)}$. Thus, the worst contention for every request going from $R^2_{\omega(i,1)}$ to $R^3_{\omega(i,1)}$ is $NR^3_{\omega(i,1)}$. Likewise, the contention when going from $R^1_{\omega(i,1)}$ to $R^2_{\omega(i,1)}$, is $NR^3_{\omega(i,1)} \times NR^2_{\omega(i,1)}$ as it includes the contention of $R^2_{\omega(i,1)}$ when going to $R^3_{\omega(i,1)}$.

Overall, the worst contention that $F_{\omega(i,1)}$ causes on F_i contenders at router R^1_i, includes the arbitration of $F_{\omega(i,1)}$ at router $R^1_{\omega(i,1)}$ and is equal to $NR^3_{\omega(i,1)} \times NR^2_{\omega(i,1)} \times NR^1_{\omega(i,1)}$. As in the previous case, it includes the contention of $R^1_{\omega(i,1)}$ when going to $R^2_{\omega(i,1)}$ and $R^3_{\omega(i,1)}$. Once the impact of $F_{\omega(i,1)}$ is computed, we derive from Figure 4.1 the contention that F_i suffers for crossing from R^1_i to R^2_i. As a result, the $\overline{WCD_i}$ expression when crossing two routers is:

$$\begin{aligned} \overline{WCD_i} &= NR^3_{\omega(i,1)} \times NR^2_{\omega(i,1)} \times NR^1_{\omega(i,1)} \times (NR^1_i - 1) \\ &\quad +(NR^2_i - 1) \end{aligned} \tag{4.2}$$

In the general case, for a packet of an arbitrary flow F_i injected in an arbitrary node R^1_i and that it has to cross H_i other routers (R_1, R_2, ... R_{H_i}) to get to its destination, the general expression for computing $\overline{WCD_i}$ of F_i is given by:

$$\overline{WCD_i} = \sum_{j=1}^{H_i} \left((NR^j_i - 1) \times \prod_{m=1}^{H_{\omega(i,j)}} NR^m_{\omega(i,j)} \right) \tag{4.3}$$

where $H_{\omega(i,j)}$ is the number of hops in the worst-case destination flow $F_{\omega(i,j)}$. The first multiplicand $(NR^j_i - 1)$ corresponds to the *contention introduced by the round robin arbitration* in each of the routers that the flow F_i traverses. The second multiplicand $\prod_{m=1}^{H_{\omega(i,j)}} NR^m_{\omega(i,j)}$ corresponds to the *indirect contention delay* in each hop due to the worst-case destination flow $F_{\omega(i,j)}$. In the rest of the chapter, we refer to the first multiplicand as rr_{cont}, and to the second as ind_{cont},

Interestingly, whether or not a node has several requests in flight has no impact on $\overline{WCD}$, since this only affects *atd* and WCD metric is insensitive to *atd*.

4.5.2 Single-Flit, Virtual-Channels, Single-entry Queue ($FT = 1, 1 < nVC < cF, E = 1$)

Virtual channels are allocated to flows in a wNoC either statically or dynamically. With dynamic allocation, virtual channels are assigned to flows at run-time based on their availability with the overall goal of maximizing buffer occupation and consequently, average network throughput. With static allocation instead, virtual channels are statically assigned to a given set of flows, such that any of these flows uses the same virtual channel until reaching the destination, in order to

reduce head-of-line blocking [77]. The way in which each of the two allocation schemes impacts wNoC contention is as follows:

For dynamic allocation, at analysis time no assumption can be made about the particular flows contending at a given router with F_i. The only safe assumption that can be made is that all flows can be potentially contending for virtual channel resources at every router. Hence, dynamic virtual channel allocation does not help reducing contention delay.

The impact of static virtual channel allocation is more complex to ascertain. Hence, if we consider terms rr_{cont} and ind_{cont} from Equation 4.3:

- An increase in nVC translates into a increase of rr_{cont}. This occurs because every arbitration round covers the selection of a contending port (e.g. for the Y+ output port the contending input ports are $X+$, $X-$, $\overline{Y}$ and PME) and a specific virtual channel of that port. Hence, the number of contenders within a router, NR_i^j, in the presence of nVC virtual channels per input port, is defined depending on the destination port as follows:

$$NR_i^j = \begin{cases} nVC \times 2 & if\ destination\ is\ X+\ or\ X- \\ nVC \times 4 & if\ destination\ is\ Y+,\ Y-\ or\ PME \end{cases}$$

  Note that the expression above generalizes the definition of NR_i^j presented in Equation 4.3, which considers no virtual channels are implemented ($nVC = 1$).

- Having more than one virtual channel, if they are smartly allocated, offers a solution to reduce ind_{cont}. The achieved reduction factor, referred to as Δ_{ind} (with $\Delta_{ind} \leq 1$), depends on the particular static allocation used. For instance, let us assume a wNoC with two virtual channels, one of which is assigned to packets sent from a given node $R(x, y)$ while the other nodes share the second virtual channel. In this scenario the packets from $R(x, y)$ suffer no indirect contention, i.e. $rr_{cont} \times \Delta_{ind} = 1$, hence reducing $\overline{WCD}$ since the reduction is expected to be higher than the increase caused on rr_{cont}. It is noted that the requests sent from the other nodes suffer a higher $\overline{WCD}$ since their ind_{cont} is not affected while rr_{cont} increases. Investigating smart static allocation virtual channel policies is out of the scope of this thesis and remains a part of our future work, in terms of the $\overline{WCD}$ model presented in this chapter we use the terms NR_i^j and Δ_{ind} to factor in these effects into $\overline{WCD}$. Overall, the general expression for $\overline{WCD_i}$ is as follows:

$$\overline{vcWCD_i} = \overline{WCD_i} \times nVC \times \Delta_{ind} \tag{4.4}$$

4.5.3 Multiple-Flit, Virtual-Channels, Single-entry Queue ($FT > 1, 1 < nVC < cF, E = 1$)

In this section we model wNoC contention when packets can have more than one flit, which are transmitted in a pipelined fashion. In order to account for the contention delay introduced by multi-flit packets, we consider L_i^{flit} as the maximum number of flits a packet of flow F_i can have.

In a pipelined router, the time that a packet r_i^k is blocked in R_i^j is given by the number of queues that can potentially contend with r_i^k ($NR_i^j - 1$) and the maximum number of flits (L_i^{flit}) of the contending requests: $(NR_i^j - 1) \times L_i^{flit}$. As a result, in order to provide a contention delay bound ($\overline{WCD_i}$), it is required to know the maximum packet length of every flow in the wNoC as in [87]. However, the latter breaks time-composability. In order to retain time composable behavior while supporting any bounded length packets, we define L_{MAX}^{flit} as the maximum length that packets in the wNoC can have. In this context, the general expression for $\overline{ftvcWCD_i}$ can be formulated as follows, based on Equation 4.3:

$$\overline{ftvcWCD_i} = L_{MAX}^{flit} \times \overline{vcWCD_i} \tag{4.5}$$

Note that typically L_{MAX}^{flit} is determined by the communication protocol (e.g. [88]) on top of the wNoC. Also, it can be limited at network interfaces by performing packetization [89].

4.5.4 Multiple-Flit, Virtual-Channels, Multiple-entry Queue ($FT > 1, 1 < nVC < cF, E > 1$ or $E < 1$)

For $\overline{WCD}$ equations so far, we have considered that queue size is equal to packet size. In this section, we consider the impact of having queues not matching the packet size.

4.5.4.1 Queue size larger than packet size ($E > 1$)

When queues have enough space to store more than one packet, the number of in-flight packets in the network increases. However, this affects the worst-case latency experienced by packets in the wNoC but not necessarily its contention delay. This is because packets in a given virtual channel queue are served using a *first-in first out* policy and therefore fairly arbitrated by round-robin arbiters across router ports. In fact, the maximum number of packets that r_i^k is contending with at a given router R_i^j is given by NR_i^j and is the same regardless the number of contending packets that can be stalled in a given router port. In this case Equation 4.5 is a valid expression for this case ($\overline{ftvcWCD_i}$).

4.5.4.2 Queue size smaller than packet size ($E < 1$)

When a packet cannot be completely buffered in a router, its flits are spread across several of the router queues the packet traverses until reaching its destination node. The number of effective contenders in a network with buffers smaller than packet size is reduced since a given packet cannot be requesting an output port at two routers at the same time [3]. With this in mind it can be concluded that the resultant $\overline{WCD}$ is equal or smaller than the one derived in Equation 4.5 ($\overline{ftvcWCD_i}$). However, composability requirements make almost impossible determining the number of effective contenders as this would require knowing the exact length for all packets of all flows in the network. Therefore, in this scenario a safe upper bound is $\overline{ftvcWCD_i}$.

4.5.5 Impact of variable size packets

The effect of message length in the maximum contention a request can suffer is huge and if the NoCs are not carefully designed this could lead to an unbounded WCTT. For wormhole NoCs the arbitration slot duration is exactly the message size, so the larger the message is the longer the time slot will be (from the arbitration point of view). In a regular wormhole NoC there is no mechanism that prevents a given source to inject messages of an undefined length in the network but is the protocol on top of the NoC the one that should set the sizes of the different messages in the NoC.

Typically, networks have messages of different lengths. Messages of different sizes are used because the payload is different based on the message type. The higher message length variability comes when different components like cores, sensors, or Direct Memory Accesss (DMAs) are interconnected using the same network. Having large and small messages is common practice in regular NoCs as this has no significant impact in the average performance of the network. However, in the context of time composable bounds mixing messages in the same network severely penalizes time composable network bounds as this implies that a given request is contending always with messages of the maximum possible length (this should be limited in the communication protocol if no special hardware mechanism is used).

4.6 System design considerations

Incremental verification calls for composable WCET estimates that are not affected by other applications. At the NoC level this translates into each request to account for a time-composable upperbound to the contention delay ($\overline{WCD}$) it can suffer. From Equation 4.5 it follows that Worst-Contention Delay (WCD) depends on three main factors: (1) the highest packet size a flow may have (L_{MAX}^{flit}), (2) the number of VC (nVC), and (3) the network size. This section discusses about these three effects when designing a CRTES and reviews some existing techniques that can be employed to minimize their impact.

4.6.1 Packet Size

Interestingly some NoC designs limit the maximum packet size by hardware and on others this responsibility is left to the software layer. In the former case, the hardware enables L_{MAX}^{flit} to be factored in WCD, which in turn enables bounding the impact that other flows on the WCD.

However, in the latter case, a low priority task may send long (or even unbounded) packets over the network, thus increasing – potentially in a unbounded manner – the WCD of high-priority tasks. In this context, it is required a suitable mechanism allowing high-priority tasks to be isolated from flows having unbounded packet size.

4.6.2 Virtual Channels

The use of virtual channels, which need to be allocated in a static manner, helps reducing $\overline{WCD}$ under the conditions presented in Section 4.5.2. Interestingly, in a mixed-criticality system, allocating each criticality level an independent VC does not help reducing WCD. First, VCs increase rr_{cont} since it is multiplied by nVC. Further, if no constraint is put on the number of cores that in a given point in time can send requests under a criticality level, ind_{cont} is not reduced, i.e. $\Delta_{ind} = 1$. Moreover, L^{flit}_{MAX}, which captures the impact that the longest packet transmitted by any flow causes on WCD, is independent of the particular VC in which that packet is allocated (see Section 4.5.3).

The dual-criticality systems, for instance, in the space domain, it is well accepted that on-board systems comprise two criticality levels [90]: one for *control* applications requiring real-time execution, and another for *payload* applications that are high-performance driven and have some (soft) real-time requirements. In such dual-criticality systems having one virtual-channel per criticality level may help reducing the WCD suffered by requests of high-criticality tasks due to low-criticality ones. This requires prioritizing high-criticality requests over low-criticality ones. Flit-level preemption can also be used to further reduce this impact. However, this comes at the cost of providing no WCD guarantees to the low-criticality tasks', since their requests' WCD depends on the load high-criticality tasks put on the NoC. In other domains, such as avionics or automotive, comprising more than two criticality levels, with several of them requiring time guarantees, the dual-criticality approach does not apply. Investigating static VC allocation policies for these domains is a fertile area of research and part of our future work.

4.6.3 Network Size

WCD directly depends on the mesh size. Clustered designs like those proposed in Chapter 3 allow creating independent islands of communication (i.e. clusters) within the wNoC by properly routing packets, which in turn reduces the WCD. However, in this case a mechanism is needed to allow inter-cluster communication without jeopardizing the WCD of the affected clusters if they are intended to run real-time tasks. In [30] inter-cluster communication is allowed by using multiple VCs. This approach proposes the use of two VC: one for intra-cluster (inside a Guaranteed Resource Partition (GRP)) and another one for inter-cluster communication (among the GRPs) assuming the amount of inter-cluster communication is a known parameter at system integration. While this holds for the case of communication requirements defined in avionics applications [25], this would jeopardize time-composability in other domain applications.

4.7 Modeling existing NoC designs

In this section we first assess the accuracy of the WCD model presented in Section 4.4 with special emphasis on the accuracy of $\overline{WCD}$ in the case of the most complex wNoC designs ($\overline{ftvcWCD_i}$). We also compare WCET estimates obtained when both WCD and WCTT are used to capture

TABLE 4.4: Technical details of the mesh NoC in high-performance chips: 48-core Intel SCC and 36 core Tilera-Gx36

	size	routing	L_{router}	nVC	wNoCs	Link width	L_{flits}^{max}
Intel SCC	6x4	XY	4 cyc	8	1	128 bit	4
Tilera-Gx36	6x6	XY	1 cyc	1	5	32 bit	16

the impact of the wNoC contention on application's WCET. Finally, we evaluate the impact that different network parameters have on the contention.

Our experimental setup comprises *gNoCsim* [38], a powerful cycle-accurate simulator of wormhole networks developed in the context of the NaNoC project, which we connect to an enhanced version of the *SoClib* simulator [37] to model a complete manycore processor (see Section 2.1). With this framework, we model two network setups resembling the ones deployed in real processors: the TileraGx36 [18] and the 48-core Intel SCC (ISCC) [60] manycore designs, based on publicly available data. Table 4.4 shows the relevant parameters of the two different wNoC setups. The main difference between these two network setups is on the usage of virtual channels. The ISCC implements a wNoC with eight virtual channels and the Tilera-Gx36 chip uses five independent networks that are used to completely isolate different types of traffic.

4.7.1 $\overline{WCD}$ accuracy and comparison with WCTT

We assess the accuracy of our $\overline{WCD}$ model in upperbounding the actual contention caused in the wNoC by creating a high-congestion scenario. To that end, we simulate the traffic generated by a memory-intensive scenario in which all cores in the network send packets to memory continuously. Note that such traffic can be produced in real scenarios by programs writing to memory continuously. In fact, we have noticed that similar congestion scenarios can be reproduced even when each node only sends packets sporadically if they share the same destination node.

In this experiment, for all the network setups we analytically compute bounds to WCTT [3] and $\overline{WCD}$ (Equation 4.5) and compare them with the measured contention delay and worst-case traversal time obtained with the simulator under highest congestion scenario. It is noteworthy to mention that the model in [3] computes bounds provided the existing flows in the system at deployment time are known and thus, precluding incremental verification. To enable a fair comparison of our metric WCD) with the WCTT metric we have adapted the expressions in [3] to achieve composable WCTT bounds by considering an all-to-all communication scenario. Note that it would also be possible to adapt WCD expressions to compute bounds for a particular application. However, in this chapter we study the benefits of WCD over WCTT in a time-composable scenario. In the modeled high-congestion scenario, the measured contention delay and traversal time closely match actual WCD and WCTT respectively. To ensure a steady congestion state is reached, measurements do not start until at least 1,000 packets per node have been injected and are performed until all nodes have sent at least 2 million requests.

Figure 4.4 shows a comparison of the measured and computed metrics in both Tilera and ISCC-like networks. We assume that buffers have capacity to store two packets to avoid bubbles in the

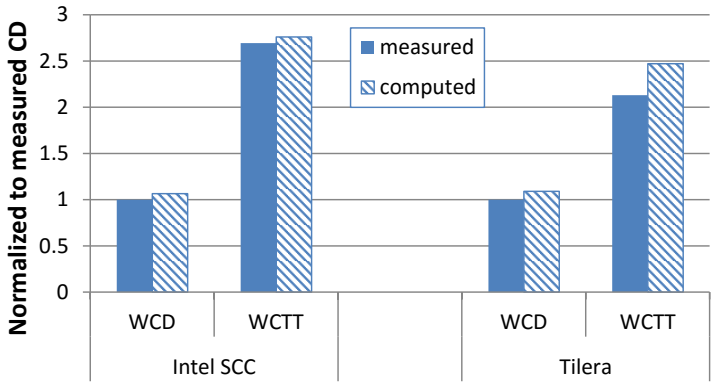


FIGURE 4.4: WCD bounds derived in this thesis and adapted WCTT from [3]

network. The results in each bar show the geometric mean (gmean) of WCD (and WCTT) for all the communication flows in the wNoC (i.e. for the packets of all nodes). This provides a measure of the WCD (WCTT) each packet suffers on average. All values are normalized to the measured (observed) contention delay.

We make the following observations.

- The derived $\overline{WCD}$ bounds are always higher than the measured contention delays confirming they upperbound the contention in the wNoC. Moreover, the difference between measured and predicted contention with our model is very small: 5% on average and 7% in the worst case.
- Likewise, measured WCTT values are close to the predicted ones [3]. The difference between WCD and WCTT (which is roughly the same for measured and predicted values) is significant, evidencing that WCTT can be a pessimistic metric to account for the interference of co-running tasks in the network. In particular, for the ISCC WCTT is 2.94x higher than the $\overline{WCD}$ and 2.7x higher for the Tilera.

It is worth mentioning, although it is not presented in any chart, that we have observed that, unlike $\overline{WCD}$, WCTT grows with buffer size which in turns makes this metric even more pessimistic when using wNoCs with larger buffer sizes.

4.7.2 Reducing $\overline{WCD}$ values

Another parameter with high impact in wNoC contention is the number of VCs and how they are allocated. In Figure 4.5 (in logarithmic scale) we show the effect of reducing in the ISCC-like wNoC the number of VCs from 8 to 1. We observe a reduction in terms of WCD (both observed and predicted) of more than 7 times. Further, a smart deployment of the wNoC by, for example, using regions of execution (clusters) as presented in Chapter 3 and Section 4.6 and properly mapping applications to cores [30] produces reduced contention delays. If we further create clusters of size $3x4$ or $3x2$ contention delay is additionally reduced. Note that all those adjustments can be done from the software without any change at hardware level in the wNoC

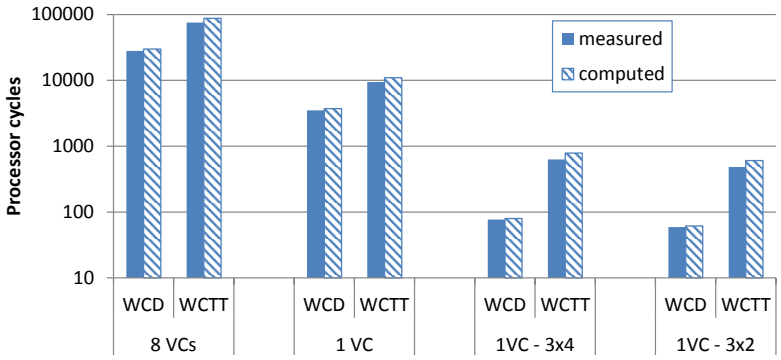


FIGURE 4.5: Effect of disabling VC and clustering on WCD for the SCC setup

design. For instance, islands can be implemented by mapping application so that communication doesn't not exceed defined island [30]. Likewise, the number of VCs is a parameter that can be easily changed from software. Researching on the convenient wNoC configurations (regions, static VC allocation) is part of our future work, building on the contention delay model developed in this thesis. Finally, it is worth noting that in all scenarios in Figure 4.4, our $\overline{WCD}$ model tightly captures the impact of these parameter variations.

4.7.3 Impact of wNoC interference on WCET

For the experiments in this section we use EEMBC Autobench benchmarks [43]. We execute benchmarks in 3 different scenarios: *2-hop* where the memory is 2 hops away (1 in X and 1 in Y dimension) from the core where the benchmark runs; *Y-only* where the benchmark is executed on the node farthest away from the memory in the Y-axis and *farthest* in which the benchmark is placed most hops away from the memory.

We compare Observed Execution Time (OET) against WCET estimates. The former is computed, by running the application under a high-congestion scenario (as explained in previous section) in order to provide fair comparison to time-composable WCET estimates. For the latter, we first compute the worst case execution time of the application using zero-load latency ($WCET_{zll}$) and increment it with the predicted impact of the wNoC (Δ_{wNoC}). As a result, the WCET estimate for manycore ($WCET_{mc}$) is computed as follows:

$$\begin{aligned} WCET_{mc} &= WCET_{zll} + \Delta_{wNoC} \\ \Delta_{wNoC} &= n_{req} \times \begin{cases} \overline{ftvcWCD_i} \\ WCTT - zll \end{cases} \end{aligned} \tag{4.6}$$

where n_{req} is the number of requests the application makes to the wNoC along the worst-case path. Note that WCTT already includes zero-load latency and in order to provide a fair comparison, we have to deduct zll from WCTT, as it is already included in $WCET_{zll}$.

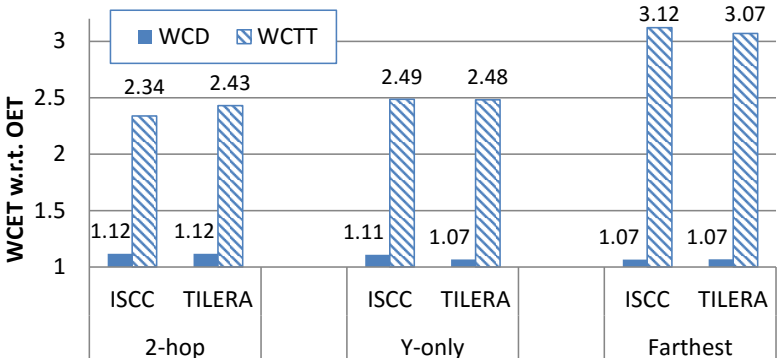


FIGURE 4.6: $WCET_{mc}$ estimates derived with WCD and WCTT w.r.t. OET

Figure 4.6 shows the gmean of $WCET_{mc}$ estimates of EEMBC benchmarks obtained with both $\overline{WCD}$ and WCTT. We observe that $WCET_{mc}$ estimates obtained with $\overline{WCD}$ are between 7% and 12% higher than OET. The maximum difference for any benchmark is around 16% in the 2-hop scenario. Meanwhile, in the case of $WCET_{mc}$ estimates obtained with WCTT there is a significant difference w.r.t. OET. They are between 2.3x and 3.1x higher, with a maximum difference of 3.2x across benchmarks.

4.8 Related Work

Several network designs targeting soft and hard real-time systems have been proposed. This section makes a short summary of them.

4.8.1 Quality of Service (QoS)

In the high-performance and high-end embedded domain, Quality of Service (QoS) is used as a metric to measure time predictability. Several proposals exist to improve predictability on wNoCs, e.g. [91]. These QoS techniques are specially suitable, for multimedia applications since QoS can be offered under severe network load conditions. Unfortunately, these techniques make difficult deriving tight contention-delay bounds to each request to the wNoC, which challenges deriving guarantees on that tasks running in a wormhole-based manycore will meet their deadlines. Authors in [92] proposed the QNoC architecture, which offers several degrees of guarantee at low hardware cost. However, despite that achieving real-time traffic guarantees is one of the targeted services in QNoC, latency bounds provided in this study do not actually bound contention delay experienced in the wNoC, preventing the derivation of time-composable WCET bounds.

4.8.2 Real-time Specific NoCs

While there are several proposal of real-time aware NoC designs – some of which have implemented in Field-Programmable Gate Array (FPGA) or implemented in real products (e.g. in the multimedia domain) –, exploring to which extend high-performance (COTS) NoC designs can be

used in the real-time domain is of paramount importance. It is well accepted that the hard real-time domain is a relative small market in comparison with other domains such as mobile. Hence, customized NoCs tailored for real-time such as Time Division Multiple Access (TDMA)-based or time-triggered ones will naturally find difficulties in being adopted by real-time industry [73] since their implementation incur high non-recurrent costs. On the other hand, the big majority of the proposed manycore designs across all computing domains use high-performance wNoCs to perform the interconnection of cores and shared resources within the chip. This makes wNoC accessible (at low cost) by the CRTES as they are implemented in a vast set of chips. This thesis makes an effort in the direction of understanding the limitations and challenges in adoption of wNoCs in real time systems.

Although it is not the topic of this chapter, in the field of real-time specific NoCs we highlight TDMA-based NoCs [93–95] approaches that satisfactorily provide time composable behavior. While this TDMA-based NoCs that deal with contention at transaction level (e.g. read and write memory operations), time-triggered architectures [96] increase the abstraction level by introducing the notion of a micro-component, which is a self-contained computational unit. In time-triggered architectures micro-components exchange messages in contention-free slots. However, event-triggered transactions, such as cache misses that access main memory through the NoC, may suffer contention delay which also must be upper-bounded.

Several hard real-time wormhole-based designs use of flit-level virtual-channel preemption mechanisms for dual-criticality systems [97, 98] to control contention in the network in order to reduce network latency. In these approaches high-priority virtual-channels can preempt packets from low-priority virtual channels so that contention delay high-priority channels is reduced at the cost of removing time-composable contention delay guarantees to low-priority channels. They require a virtual channel per communication flow, which limits their scalability. This limitation is addressed in [99] by proposing a priority share policy where contending flows can share a given priority. Along this line, a recent work [87] proposes an enhanced priority-shared flit-level virtual-channel preemption NoC to support two criticality operation modes. This design fulfills isolation requirements across criticality levels without incurring in hardware resource wasting. Further, in [100], authors build on top of [97, 99] and provide response time analysis which considers impact of pipelining in the NoC. However, this analysis considers communication among tasks and do not consider memory accesses inside a task.

In general, flit-level virtual-channel preemption mechanisms offer tight contention delay estimates. However, these approaches consider the impact of contention delay at the schedulability and response time analysis and require knowing the exact set of tasks using the wNoC and their communication flows, which negatively affects time composability and incremental verification, as discussed in Section 1.3.1 and in Section 4.2.

4.8.3 WCTT in wNoCs

Several works focus on deriving an upperbound of WCTT adapting network calculus [101] to model wNoCs [81]. Network calculus relies on the determination of arrival curves of the applications running in the system to determine an actual upperbound of WCTT. While these

approaches allow providing tight WCTT estimates, as WCTT is adapted to the exact network load conditions, using per-application arrival curves reduces time composability, since WCTT estimates depend on the load corrunners put on the NoC. Another set of works focuses on determining wNoC packets WCTT by considering worst-case conditions, first with assuming limitations on the packet-injection rate [80]. However, the bounds provided in [80] required assuming packet injection is limited. In later works [3, 102] this limitation is removed.

In this line of work, Dasari et. al [82] achieve tighter WCTT bounds – than those derived with [3, 102] – based on the following two observations: (1) flows injection rate is inherently limited by the speed at which the processor pipeline can process request-generating instructions (e.g load or stores); and (2) packets of a given flow do not always contend with the flow under analysis due to the existing release time of their request of a flow/core. Regarding observation (1) we have shown that while limiting the injection effectively reduces WCTT values the actual contention that packets in the NoC suffer remains unaltered, hence producing no effect on WCD. Regarding observation (2) knowing what is the actual interval between consecutive requests in every flow in the network breaks the time composability requirement.

4.9 Conclusions

This chapter analyzes the suitability of applying wNoCs in the real-time embedded domain. To do so, we propose a new metric to account for the impact that NoC interferences coming from different requesters have on the WCET estimates: the Worst-Contention Delay (WCD), which replaces the traditional metric, the Worst-Case Traversal Time (WCTT). Moreover, we derive an analytical model that computes time-composable WCD bounds ($\overline{WCD}$) based on common wNoC design parameters including flits-per-packet, number of virtual channels and queue size in the router. $\overline{WCD}$ is computed based on a wNoC parameter taxonomy that identifies those parameters that must be fixed in order to provide trustworthy and composable WCD bounds; and those allowing certain flexibility (objectives **O1** and **O3**).

Our $\overline{WCD}$ model allows evaluating a wide range of existing COTS high-performance wNoCs. To that end, we apply the model considering the design parameters of two wNoCs deployed in real processors: the Tilera-Gx36 and the 48-core Intel SCC (objective **O1**). Our analysis shows that considering WCD rather than WCTT reduces WCET estimates by around 2.5x for Tilera and ISCC on average (objective **O3**).

Chapter 5

Improving Performance Guarantees in Wormhole Mesh NoC Designs

Wormhole-based Network on Chips (wNoCs) are deployed in high-performance many-core processors due to their physical scalability and low-cost. Delivering tight and time composable Worst-Case Execution Time (WCET) estimates for applications as needed in safety-critical real-time embedded systems is challenged by wNoCs due to their distributed nature. In this chapter we propose a bandwidth control mechanism for wNoCs that enables the computation of tight time-composable WCET estimates with low average performance degradation and high scalability. Our evaluation with the EEMBC automotive suite and an industrial real-time parallel avionics application confirms so.

5.1 Introduction

Critical Real-Time Embedded Systems (CRTES) industry is gradually shifting towards multi- and manycore processors to satisfy the performance needs of complex safety-related functions. This transition challenges the derivation of time-composable WCET estimates, i.e. tasks' execution time bounds that are independent of the load that co-running tasks put on shared resources. Time-composable WCET estimates enable incremental verification [103] by allowing each system component to be subject to formal timing verification in isolation and independently from other components.

From an end-user perspective, the deployment of manycores in CRTES, as stated in Section 1.3 requires following properties:

- *UserReq1*: Manycores should facilitate deriving tight WCET estimate so that high (guaranteed) performance is provided (objectives **O1** and **O3**);

- *UserReq2*: Manycores must facilitate deriving time composable WCET estimates (objective **O1**);
- *UserReq3*: Manycores should also provide high average performance for some applications (objective **O1**);
- *UserReq4*: Manycores for real-time should use technology as close as possible to Commercial-Off-The-Shelf (COTS) (high-performance) technology to ease their adoption (objective **O1**). The low manycore demand of safety-critical real-time systems, w.r.t. the mainstream market, calls for reducing the need for customized real-time technology.

This chapter tackles the fulfillment of the above requirements on Network on Chip (NoC) designs, as it is one of the manycore shared resources with the highest impact on average performance and WCET. Concretely, we consider wNoC mesh as a candidate NoC solution as it is widely accepted in the high-performance market due to its physical scalability and low cost [18][60].

The high-performance requirements (*UserReq3*) are already fulfilled by wNoCs as they are designed for high-performance systems. *UserReq2* for real-time applications requires time-composable Worst-Case Traversal Time (WCTT), i.e. WCTT not affected by the load contender tasks put on the wNoC. Typically, latency bounds for wNoCs are reffered as WCTT. wNoCs can also meet this by using time-analyzable arbitration policies [86][85] and applying the model in [3].

This chapter makes the following contribution:

- We show that current wNoCs fail to achieve tight WCTT (*UserReq1*), which negates their benefits. In particular we show that (i) WCTT values derived for current wNoCs poorly scale with network size – even for small networks; and (ii) the WCTT derived for a task depends on the maximum allowed packet size and poorly scales with it. Further, current wNoCs do not necessarily impose a limit on the packet size and leave that to the protocol on top of the network (e.g. AMBA [88]).
- We propose a new time-composable wNoC design relying on concepts developed for high-performance wNoCs, hence achieving *UserReq4* and objective **O1**. Our design focuses on controlling the network bandwidth (the main factor affecting *WCTT*) to provide a fair guaranteed bandwidth distribution across the different communication flows. Bandwidth control is exercised at two levels. At local level, we ensure fairness by providing a WCTT-aware Packetization (WaP) that makes real-time guarantees independent of contenders packet size. At global level, we provide fairness across contenders by performing a WCTT-aware Weighted (WaW) round-robin arbitration.
- We evaluate WaW+WaP on a 64-core manycore architecture with cores accessing memory controllers through a wNoC. We use EEMBC [43] autobench and an avionics real-time parallel application provided by Honeywell [30] (objective **O4**). We show that our design significantly decreases WCET estimates for the parallel application by a factor of 4.8× to 9.5× depending on the number of flits per packet. For single-threaded applications WCET decreases by 230× on average across all cores and by 1.4× w.r.t 25% of the best cores of the baseline NoC.

Note that proposals made in this chapter can be orthogonally applied to Worst-Contention Delay (WCD) metric proposed in Chapter 4. We opt for comparison to the WCTT and state-of-the art wNoC proposals[58], as the main benefit of the proposed WaW+WaP is the fairness and it is affecting both approaches in the same manner, but comparison using WCTT better highlights the effectiveness of the solution.

5.2 Reference mesh network

We model a canonical 2D wormhole mesh router comprising five input ports that have queues to store packet flits. The router arbiter grants an output port to a given input flow. To be able to have time-composable WCTT estimates, no prioritization mechanism is used in the router, and arbitration decisions to select the flow accessing the requested output port are taken using a time-analyzable arbitration policy, e.g. round-robin.

We consider a NxM mesh NoC configuration as depicted in Figure 5.1(a), in which each node can be identified using (x,y) coordinates. The router located at coordinates (x,y) is referred to as $R(x, y)$. Each node comprises the router that communicates the node to the mesh and a Processor/Memory Element (PME) The PME can be either a processor core, a cache memory, main memory, I/O, etc. In the network several traffic flows may exist. A traffic-flow (F_i) is a packet stream that traverses the same H-node route from a source to a destination node and requires the same grade of service along the path.

We use deterministic XY routing, which is time analyzable [3] and has low implementation costs. It enables identifying routers in a given path as R^j where j is the hop number of the path (e.g. R^1 is the source node). With XY routing packets are forced to use the X dimension first. In the X dimension the position of the target node with respect to the source node determines whether to go right (X+) or left (X-) direction. The same approach is used for the Y dimension. Once packets are routed using the Y dimension they cannot be forwarded to the X dimension. Note that the opposite port is represented as $\bar{Y}$ and $\bar{X}$. For instance the opposite port of $Y+$ is $Y-$. Routing restrictions help determining the number of requests (P_i^j) that might contend at router R^j for the same output port as $F_i$in the worst-case. P_i^j values can be determined as follows:

$$P_i^j = \begin{cases} 2 & if\ destination\ is\ X+\ or\ X- \\ 4 & if\ destination\ is\ Y+,\ Y-\ or\ PME \end{cases}$$

Wormhole switching is the most adopted approach in NoCs due to its low buffering requirements. In a wormhole NoC every core request translates into a packet, which is the minimum arbitration unit and that can be split into one or several flits. The header flit of a packet contains the destination information required to forward the packet to the corresponding router output port. Once the header flit is granted access to a given output port, the remaining packet flits are forwarded to this port without any further arbitration.

5.3 Wormhole-based mesh NoCs

Deriving WCET estimates in manycores relies on bounding access times to shared hardware resources [36][104]. In the case of NoCs this traditionally translates into i) bounded WCTT such that every request sent to the NoC has a service time, i.e. traversal time, boundable at analysis; and ii) time-composable WCTT such that the bound to the traversal time derived for the request of a task does not depend on the load put by other co-running tasks on the NoC. Low WCTT translates into tighter WCET estimates, which allows increasing the guaranteed performance that the manycore chip can provide.

5.3.1 Assumptions

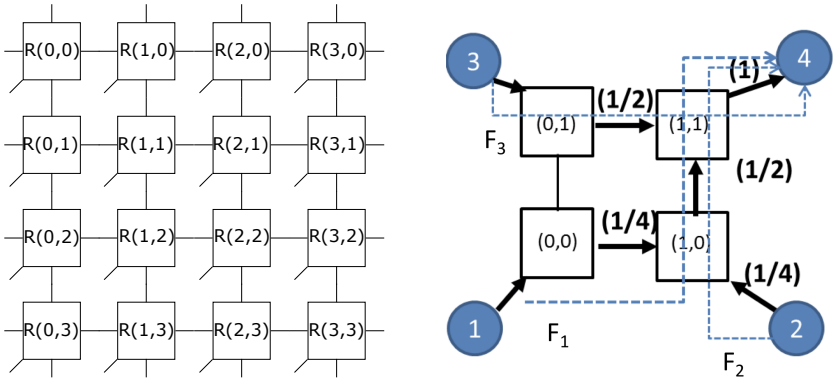
We assume a canonical 2D-mesh [77] with wormhole switching and XY routing policies (Figure 5.1(a)). The need for time-composable WCTT prevents making assumptions about the number and load of crossing flows. Time-composable WCET estimates provide a drastic reduction of development costs as each subsystem can be independently developed, verified and incrementally integrated. These benefits pay off the increase in WCET caused by achieving time composability. Instead, we assume the worst-case of the wNoC state and load:

1. Every node in the network is able to send and receive packets to/from any other node in the network.
2. Every time we inject a packet in the NoC, any possible contending flow is also sending packets creating a worst-case contention scenario, i.e. for a packet of a given flow at every hop all possible contenders (i.e. all possible flows partially sharing the path) are also requesting the same output port (see Section 4.4).
3. Packets contending for an output port are arbitrated using a time-analyzable policy – round-robin in our case [85], which is already used in some existing mesh wNoCs [18][60].
4. Maximum allowed packet size in the network is known. We assume that packets of contending requests have maximum size when deriving WCTT bounds.
5. Finally, it is also required assuming that the network is congested by the time packets are injected in the network.

5.3.2 Factors impacting WCTT estimates

There are two main aspects affecting real-time guarantees that we address with our design:

- Message size impact on arbitration slot duration: In wNoCs only the header flit of a packet is arbitrated. This implies that *the time that requests in a given router wait to be arbitrated depends on the size of the particular requests contending for the same output port.* Hence, deriving time-composable WCTT values requires considering that all possible requests (i.e.



(a) (b)

FIGURE 5.1: (a) Router coordinates in a 4x4-Mesh. (b) Unfair bandwidth allocation in wormhole.

the number of router ports minus one) can contend for the same output port and the size of all requests is the maximum allowed size. However, some wNoCs do not impose any limit on packet size, enabling undefined-length requests [88]. Even with the maximum packet size limited, different lengths of packets penalize real-time guarantees, since we have to consider that contending requests have the maximum size.

- Unfair bandwidth allocation: In a network where all flows may contend for the same resource, WCTT mainly depends on the flow's allocated bandwidth. The latency in a congested system can be approximated as $1/(bandwidth)$ [77]. Despite round-robin arbitration ensures a fair distribution of resources when it is used in a centralized way, *round-robin fails to fairly share the bandwidth in distributed networks*. For example, when round-robin is used in an on-chip bus, it distributes the bandwidth amongst the cores accessing the bus fairly. However, when a request passes through several chained routers to reach a given node, the bandwidth allocated to this request is not the same as the one allocated to a closer or farther request. In Section 5.6 we show how the unfair bandwidth allocation translates into bad (high) WCTT values.

5.4 Computing Worst-case Traversal Time

Worst-Case Traversal Time (WCTT) values can be derived for regular wNoC designs following the expressions given in [3]. However, for those bounds to be time-composable the assumptions described in Section 5.3.1 need to be enforced when computing NoC latency bounds. Moreover, expressions given in [3] are only suitable for wormhole NoC designs that consider a regular round-robin arbitration policy.

In this section, we provide novel expressions to compute time-composable WCTT bounds that are also suitable for NoCs using weighted round-robin arbitration. Expressions given in this section are based on the concept of worst-case ejection rate (ER_i^j). We define ER_i^j as the rate at which flits of flow F_i can be ejected from router R^j to the corresponding port when the next router (R^{j+1}) is accepting incoming packets (i.e. it is not stalling R^j packet transmission). We

also extend the concept of worst-case network ejection rate to model the rate at which flits can be ejected from a given router port when the network is fully congested. To do so, we define propagated worst-case ejection rate PER^{wc} as the minimum rate at which flits of F_i can be ejected from R^j in the worst-case situation. ER_i^j values can be computed by considering the maximum number of flows P_i^j contending at R_i^j for the same output port as F_i : $ER_i^j = 1/P_i^j$. $PER_i(R^j)$ is computed by multiplying ER_i^j factors from the current router R_i^j to the target router R_i^H as presented next:

$$PER_i^j = \prod_{k=j}^{H} \frac{1}{P_i^k} \tag{5.1}$$

Let D_i^j be the time that a packet of flow F_i requires to go from the input port of R^j to its destination node. D_i^j can be computed recursively by considering the time required to reach R^{j+1} $(1/PER_{fx\{i\}}^j)$ plus the time required to reach its destination once at R^{j+1}. $fx\{i\}$ represents the index of the flow that causes the worst-case blocking in F_i. Note that a $F_{fx\{i\}}$ packet stalled in a subsequent router of the path followed by F_i might cause F_i to suffer worst contention than one following exactly the same path. In the same way $PER_{fx\{i\}}^j$ represents the worst ejection rate for F_i packets. To determine the flow causing the worst contention, PER values for all routers and all flows have to be computed in advance, and for any particular flow and router we choose the worst $PER_{fx\{i\}}^j$. Equation 5.2 shows the recursive definition of D_i^j.

$$D_i^j = \frac{1}{PER_{fx\{i\}}^j} + D_i^{j+1} \tag{5.2}$$

The WCTT for flow F_i, given by D_i^0, is the time required to reach its destination $(j = h)$ from the source node $(j = 0)$.

We illustrate how to compute WCTT using Equations above with the example presented in Figure 5.1(b). We aim at computing F_1 WCTT, i.e. WCTT of packets with source node 1 and destination node 4. First, we compute PER_i^j as the product of the ER_i^j coefficients (shown in brackets in Figure 5.1(b)) of all the routers that F_1 traverses. Later, we start from the last hop $(j = 3)$ and compute all D_i^j values shown below.

$$\begin{aligned} D_i^3 &= \frac{1}{1/2} = 2 & \qquad D_i^1 &= \frac{1}{1/4} + D_i^2 = 10 \\ D_i^2 &= \frac{1}{1/4} + D_i^3 = 6 & \qquad D_i^0 &= \frac{1}{1/4} + D_i^1 = 14 \end{aligned} \tag{5.3}$$

Note that the expressions above can also be employed to compute WCD in the context of a weighted arbitration. To do so, in the context of the modeled mesh wNoC, we just have to remove the interference coming from packets of the same flow (intra-task interference) from the D_i^j expression. Since internal interference is only exclusively accounted for at the source node. This translates into iterating D_i^j from j=1 to j=h instead of from j=0 to j=h.

5.5 Flit-Homogeneous Guarantees in Meshes

We present a new wNoC design that performs a flit-level fair distribution of guaranteed bandwidth to achieve time-composable and tight WCTT. Our proposal requires minimum modifications to regular mesh designs in the packet generation (*local fairness*) and in the packet arbitration (*global fairness*).

5.5.1 WCTT-aware Packetization (WaP)

Packet length has high impact on the maximum contention a request can suffer. If the wNoC is not carefully designed this could lead to an unbounded WCTT. For wNoCs the arbitration slot duration is equal to the packet size, the larger the packet is, the longer the time slot is. To avoid arbitration slots of different duration we use WaP that forces all packets to have the size of the smallest packet in the system. This is achieved by slicing a request into one or more minimum size packets at the Network Interface Controller (NIC).

When a request (Req_i) arrives at the NIC a regular packetization scheme creates a single packet that is injected in the network. With WaP the request payload is sliced in minimum sized packets and header info is replicated. WaP improves NoC WCTT as the size of contending packets is bounded to the minimum size packet. For instance, with a regular packetization scheme, the worst-case latency for a S-flit packet for reaching an output port to which 4 different input ports are contending is $3 * L + S$ where L is the maximum allowed size of packets in the network. Instead, with WaP, for a minimum packet size of m, the worst-case latency is $3 * m + m$. Note that maximum allowed packet size in the network (L) is much larger than minimum size packets (m) that commonly consist of one-flit.

WaP penalizes the effective bandwidth due to the overhead of the required routing and control information (that can be significant in a manycore). In Section 5.6 we evaluate WaP in terms of both average and worst-case performance.

5.5.2 WCTT-aware Weighted (WaW)

WaW relies on a weighted round-robin arbitration scheme [105] to enable a globally fair link bandwidth distribution that balances the WCTT off all nodes in the NoC. Weighted round-robin uses weights to assign the rotating priorities to contending input ports. WaW uses arbitration weights per router input port that balance WCTT in all nodes of the router. It set weights by accounting for the number of contending flows coming through a given input port and the total number of flows traversing the requested output port. These numbers are determined by the routing algorithm [77].

Let $I_{dir_i}(i,j)$ be the number of communication flows traversing the dir_i input port of router $R(i,j)$ – dir_i can be any of the possible mesh router port directions $X+$,$X-$, $Y+$, $Y-$, or PME (signs + or − refer to the direction of travel within a dimension). Let $O_{dir_o}(i,j)$ be the number of flows traversing the dir_o output port of $R(i,j)$. WaW per-input/output port pair arbitration

TABLE 5.1: Arbitration weights for a 2x2-mesh router R(1,1) in a regular mesh and with WaW

	Regular Mesh	Weighted Mesh
W(PME,X-)	1	1
W(PME,Y-)	0.5	0.5
W(X-,PME)	0.5	0.33
W(X-,Y-)	0.5	0.5
W(Y-,PME)	0.5	0.66

weights $W(I_{dir_i}, O_{dir_o})$ can be computed for any of the possible input/output port combinations at $R(i,j)$ using the following equations:

$$\begin{aligned} I_{X+} &= x & O_{X+} &= x+1 \\ I_{X-} &= N-x & O_{X-} &= N-x+1 \\ I_{Y+} &= N*y & O_{Y+} &= N*(y+1) \\ I_{Y-} &= N*(M-y-1) & O_{Y-} &= N*(M-y) \\ I_{PME} &= 1 & O_{PME} &= N*M-1 \end{aligned} \tag{5.4}$$

N and M are the horizontal and vertical dimensions of the network, respectively, while x and y stand for the horizontal and vertical coordinates of the node under analysis. For the $X+$ input port the number of flows coming through it corresponds to the x coordinate i.e. the number of nodes that precede the actual node in the same row. Note that with XY routing, packets in the Y direction cannot be forwarded to the X direction. Therefore, the flows accessing an X port are only the ones in the same row. On the contrary, flows crossing Y-direction ports may be originated at any of the preceding nodes in any row. Per direction router weights are derived using Equation 5.5.

$$W(I_{dir_i}, O_{dir_o}) = I_{dir_i}/O_{dir_o} \tag{5.5}$$

Let us illustrate how WaW works with the example from Figure 5.1(b). Let us consider all flows with destination node 4. At $R(1,1)$ only $X+$ and $Y+$ input ports can access the PME output port. $O_{PME} = 3$ as the flows originated at the 3 remaining nodes access node 4 using O_{PME}. For the input ports we have $I_{X+} = 1$ and $I_{Y+} = 2$. We consider that in this example $N = 2$ and $M = 2$ so $I_{Y+} = |2*(1)| = 2$ and $I_{X+} = x = 1$. Table 5.1 shows $R(1,1)$ weights required to perform the weighted arbitration in the 2x2 mesh NoC and compares them with the default weights of the round-robin arbitration. The weight values range from 0 to 1 and represent the bandwidth that is allocated to a given input/output pair. For example, for the input ports requesting the PME output port the weighted arbitration assigns 1/3 of the bandwidth to the flows coming from X- and 2/3 of the bandwidth to the flows from Y-. Note that X- only serves one flow from node 3 to node 4 while Y- serves 2 flows (from nodes 1 and 2 to node 4). Instead, round-robin arbitration assigns always the same bandwidth (0.5) to any of the 2 input ports requesting a given output port, regardless of the number of potential flows using these input ports.

TABLE 5.2: WCTT values for different Mesh sizes for 1-flit packets.

	Regular Mesh			WaW + WaP		
NxM	max	mean	min	max	mean	min
2x2	14	10	6	11	9	8
3x3	123	39.16	9	32	24	17
4x4	1071	145.68	9	64	45	31
5x5	8895	568.14	9	108	72	49
6x6	72447	2375.85	9	163	105	71
7x7	584703	10632.53	9	230	144	97
8x8	4698111	50516.79	9	310	189	127

5.5.3 WaW implementation

XY routing allows precomputing the weights and assigning them to input ports statically, as needed for WCET estimation. In our implementation, input port weight is measured as the number of flits it can transmit to an output port. When several input ports contend for an output port, the input port with the largest flit count wins, and decrements its flit count by one. If more than one contender has the largest flit count, a conventional round robin policy is used to arbitrate amongst them. Instead, when no input ports demand an output port, each input port flit count is incremented (if it is not larger than its weight). When an input port is the unique candidate to access an output port, its flit count is unaltered.

5.5.4 Hardware modifications

In order to increase compliance with COTS wNoC designs (objective **O1**), WaW and WaP incur minimum local changes. Those changes can be implemented in regular wNoCs which could provide a feature to enable/disable them depending on the average and guaranteed requirements of the wNoC. This departs from other designs that might require changes in buffering, switches architecture, synchronization, etc., that would decrease the chance of adoption of our proposal.

NICs are already equipped with the logic to perform packetization of processor requests. Hence, WaP only requires the size of packets to be parametrizable from the software. Meanwhile WaW requires per-input port counters (no more complex than the ones required for regular round-robin arbitration) and an additional arbitration policy. Our results – obtained from the NoC area decomposition given in [106] – show that the area increase incurred in the NoC is below 5%.

5.6 Evaluation

We use a cycle-accurate simulator based on SoCLib [37] with gNoCSim [38] integrated (see Section 2.1). We model a 64-core mesh-based processor (routers range from $R(0,0)$ to $R(7,7)$). In our manycore, load (and write-miss) requests comprise a one-flit message from the core to memory. Given that cache line size is 64-bytes and we need 16-bits for control data (512+16 bits), memory answers with 4-flit messages over 132-bit wide links. Evicted line requests require a 4-flit

TABLE 5.3: Normalized WCET per core of EEMBC with WaW+WaP

Y-axis position \ X-axis position	0	1	2	3	4	5	6	7
0	1.4841	1.4841	1.4920	1.4387	1.3046	1.0850	0.8131	0.7292
1	1.3609	1.3806	1.2843	1.0899	0.8262	0.5575	0.3427	0.3260
2	1.2454	1.0856	0.8441	0.5777	0.3553	0.2027	0.1112	0.1226
3	0.9855	0.6078	0.3739	0.2123	0.1150	0.0609	0.0321	0.0428
4	0.6024	0.2304	0.1219	0.0634	0.0328	0.0169	0.0088	0.0145
5	0.2779	0.0692	0.0345	0.0174	0.0089	0.0046	0.0024	0.0049
6	0.1063	0.0189	0.0093	0.0046	0.0024	0.0012	0.0004	0.0016
7	0.0528	0.0067	0.0033	0.0016	0.0008	0.0004	0.0002	0.0008

message and a one-flit answer. WaW+WaP adds control data to each flit, therefore requiring an extra flit, so 5 instead of 4 (512+5*16 bits over a 132-bit wide channel), leading to 25% overhead.

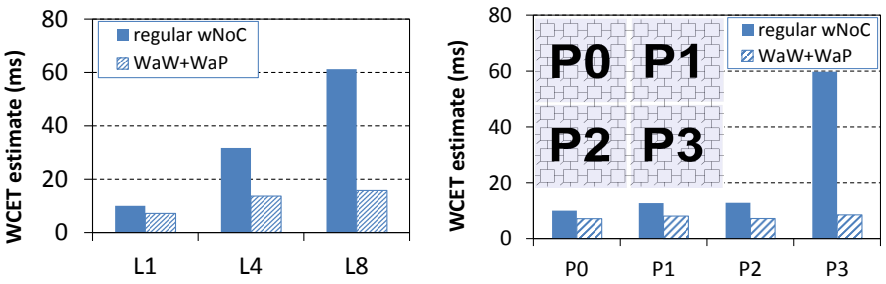
5.6.1 Reducing WCTT with WaW+WaP

Table 5.2 shows average, max, and min WCTT values for the regular wNoC and WaW+WaP across several network sizes. While regular mesh designs obtain always the lowest WCTT values (for the nodes that are directly attached to destination) our proposal achieves significantly better WCTT values for the majority of the network flows (as shown by the average WCTT results). For instance, for the 64-node NoC the minimum WCTT with regular meshes is 9 and with WaW+WaP is 127 cycles, while the maximum value decreases from above 4 million cycles to 310 (a decrease of 4 orders of magnitude). On average the WCTT for the original NoC is above 50,000 cycles (largely above our design, 189).

5.6.2 Improving WCET estimates for single threaded applications

Our simulation architecture supports the *WCET computation mode* [86], in which at analysis time, requests accessing the NoC are artificially delayed by an Upper-Bound Delay (UBD). During operation, WCET computation mode is disabled and NoC requests suffer only actual delays, which are safely upper-bounded by UBD.

In Table 5.3 each cell represents a node of a $8x8$ wNoC. All nodes communicate to the memory connected to the top-left node $R(0,0)$. Each cell shows the WCETs of WaW+WaP normalized w.r.t. a regular wNoC. In particular we show the average reduction across all (single-threaded) EEMBC Automotive benchmarks. Values above 1 show that WaW+WaP provides higher WCET estimates than a regular wNoC and vice versa. We observe that WCET values for nodes close to $R(0,0)$ are slightly higher than for the regular wNoC. In particular 11 nodes present WCET values worse than the ones provided by a regular wNoC with a maximum slowdown of up to 1.5× for the best situated node. However, on the other 53 nodes, average WCET estimates are significantly higher (worse) with the regular wNoC than with WaW+WaP. In some cases, as shown in Table 5.3, the difference is 3-4 orders of magnitude, i.e. the WCET obtained with WaW+WaP is only 0.002 of that with the regular wNoC.



(a) Regular wNoC vs. WaW+WaP (b) Impact of Allocation

FIGURE 5.2: WCET estimates for the 16-core parallel avionics application

5.6.3 Improving WCET estimates for Parallel Applications

We also evaluate WaW+WaP using 3D path planning (3DPP), an industrial avionics parallel application provided by Honeywell [30] (see Section 2.2.1). 3DPP uses 16 cores to guide an aircraft through the obstacle map represented as a 3D matrix. In the $8x8$ wNoC we run 3DPP under four different placements (see Figure 5.2(b)).

With focus on P0, Figure 5.2(a) shows the WCET estimates for the regular and WaW+WaP wNoC considering that the maximum packet size in the network is 1, 4 and 8 flits (labeled L1, L4 and L8 respectively). We observe the significant impact of WaW+WaP. Overall, it outperforms the regular wNoC for all packet sizes considered, with improvements ranging from 1.4X for L1 to 3.9x for L8.

For the L1 setup Figure 5.2(b) shows the impact of placement of the application. WaW+WaP benefits are two-fold. It achieves lower WCET estimates (from 1.4x to 7x) than the regular wNoC and leads to smaller variability across placements (around 20% in our setup compared to over 6x with the regular NoC). This is of paramount importance in real-time systems to control the impact of placement, which has been shown as a first-order factor in the WCET [107].

5.6.4 Average performance

We have as well evaluated WaW+WaP and regular wNoC in terms of average performance. Results show that WaW+WaP incurs negligible average performance degradation (less than 1%) for both single-threaded and parallel applications. The origin of the degradation resides in the overhead introduced by packetization that is minimized as it only affects those packets having more than one flit.

5.7 Related Work

Customized NoCs for real-time such as Time Division Multiple Access (TDMA)-based or time-triggered ones will find difficulties in being adopted by the real-time industry [73] since their implementation incurs high non-recurrent costs (see Section 4.8) This is the case for [93, 95, 96, 108].

In best-effort wNoCs the use of virtual channel prioritization has been proposed as an effective way to provide tight latency bounds [97] and [109]. The same logic applies to [110], where authors provide bandwidth guarantees for Guaranteed Service (GS) connections per port. However provided guarantees require a detailed knowledge of the applications/tasks that will run in the final system and thus, fail to satisfy incremental verification requirements.

In [3, 80] authors provide realistic bounds for wNoCs without using flit-level virtual channel preemption. The model in [3] requires knowing all communication flows integrated in the system to derive safe upper-bounds, making those bounds not time-composable. Interference-free NoC designs using wormhole-based NoC designs have been recently proposed in [111] and [112]. While [111] shows lower best-effort traffic degradation than [112] by smartly multiplexing virtual channels, the degradation of best-effort traffic performance is significant.

We follow a different approach to fulfill hard-real time requirements by deriving time-composable WCTT bounds in wNoCs without sacrificing average performance. Further, we address the scalability problems of latency bounds in wNoC by proposing a mesh design that significantly improves default mesh WCTT values with low hardware complexity.

5.8 Conclusions

The use of wormhole-based NoCs in the context of CRTES applications complicates the timing analysis of applications, making the WCET estimates of those applications rapidly increase with the network size. The latency bounds achieved by our design are scalable. Our proposal enables a fair sharing of the available bandwidth across the different flows in the network. This makes time-composable WCET estimates less affected by the core count in manycore (objective **O3**).

Our results with benchmarks and a real application (objective **O4**) confirm that the proposed mesh achieves tight and uniform scalable WCET values with negligible average performance degradation. Furthermore, hardware modifications required for the proposed design w.r.t. regular mesh designs are few, easing its adoption (objective **O1**).

Part III

Software Support for Exploiting Manycore Potential – Scheduling

Chapter 6

Intra-GRP Scheduling Strategy for Parallelization of Complex Automotive Applications

This chapter tackles improvement of guaranteed performance for complex legacy applications by parallelizing and allocating them to a many-core processor designs described in Chapter 3. We focus on control applications from automotive domain, as they were built with single-core architectures in mind and they are good candidates for parallelization due to their complexity and minimizing efforts in parallelization and avoiding re-validation of the applications stands as an imperative.

6.1 Introduction

Modern road vehicles carry up to 100 single-core Electronic Control Units (ECUs) performing various functions, from opening a window to controlling the engine. This makes automotive industry to pay special attention to minimize Size, Weight, and Power (SWaP) costs, while increasing the services delivered per ECU. Multi- and many-core processor architectures, which are nowadays a reality in other embedded domains [4, 19, 113], are considered as a promising solution to cope with such performance and cost constraints.

Many-core ECUs aim at providing the performance required to run a high number of complex functions by:

- Integrating distributed applications into a single ECU;
- Parallelizing the computation of complex systems, such as the Engine Management System (EMS) or Advanced Driver Assistance System (ADAS);
- Combination of both.

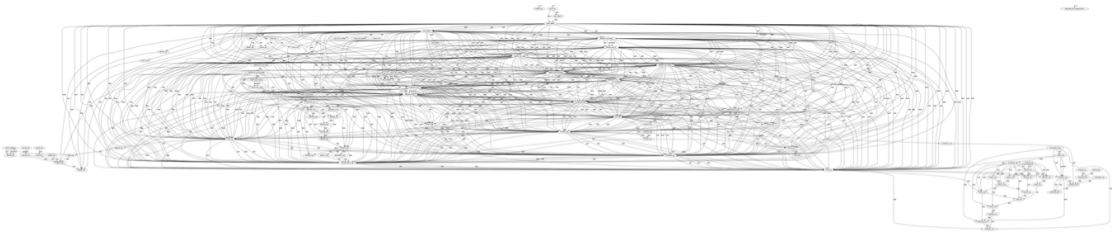


FIGURE 6.1: Inter-runnable dependencies existing among three of the twelve tasks that compose the EMS (tasks 1, 4 and 8 ms). Nodes represent runnables and lines the dependencies among them

We focus on the parallelization of complex applications, i.e. improving the performance of an automotive legacy application by effectively parallelizing it over several cores (Objective **O2**) of a single Guaranteed Resource Partition (GRP), considering EMS as a case study (Objective **O4**). In this respect, it is important to remark the relevance of this problem given that a significant part of automotive software is composed of legacy code (Goal 3).

Automotive applications increasingly rely on the AUTomotive Open System ARchitecture (AUTOSAR) [24], a standardized system software architecture upon which applications are built and executed. In AUTOSAR, applications comprise a set of functions, named *runnables*, that are either executed periodically or triggered by an interrupt. When developing an AUTOSAR application, runnables are grouped into *AUTOSAR tasks*[1], which are the Unit of Scheduling (UoS) of the AUTOSAR Operating System (AR-OS). The runnable-to-task mapping and the single-core task scheduling of an application is known as *application configuration* and it is static and known at system integration time. Development of application configuration has high cost of validation its functional and timing correctness [29], and it is done infrequently (only once for most of the applications, exceptions are e.g. Formula 1 engine control applications, where you have several application configurations).

The current strategy of using tasks as UoS works well on applications running on single-core ECUs, because it facilitates scheduling runnables with the same timing properties by grouping runnables with the same release period or interrupt into the same task.

A single GRP is an equivalent a multi-core ECU in terms of scheduling. Current approaches targeting multi-core ECUs also consider tasks as UoS [114][115], allocating them to different cores. To do so, all dependent runnables are grouped into a single task, minimizing or even removing all inter-task communications, and so scheduling independently the different tasks to the processor cores. This approach, which is in-line with the latest AUTOSAR guide for developing and configuring AUTOSAR-compliant software for multi-core systems [24], works well for integrating multiple applications into a single ECU or for parallelizing applications with little inter-runnable communication. However, the use of tasks as UoS on many-core processors to extract parallelism of applications with highly-connected runnables is inefficient, as most runnables are allocated to a single task and thus executed sequentially in one core. This is the case for the EMS application, in which almost all runnables depend on each other. Figure 6.1 provides an intuition on the level of communication existing in the EMS, showing the inter-runnable dependencies of three of the

[1]In this chapter, we refer to an AUTOSAR task simply as task.

twelve tasks that compose the EMS (concretely time-triggered tasks with periods of 1, 4 and 8 ms; see Section 6.4.1 for further details).

Moreover, current approaches require changing the runnable-to-task mapping and/or the single-core task scheduling to execute tasks in parallel as a means to improve application performance. This, in turn, implies changing the application configuration, resulting in extra effort to verify and validate the new configuration [116]. This is due to the fact that the sequential execution model of tasks abstracts and may hide mutual exclusion constraints when accessing shared resources, critical sections, etc. The parallel execution of tasks can then break this mutual exclusion relations present in applications configured for execution in single-core processors [116].

In this chapter, we propose exploiting the performance opportunities of multi-core ECUs by proposing a new allocation strategy in which *legacy automotive applications* (objective **O2**) with runnables highly connected are parallelized while maintaining the single-core application configuration (**Goal 3**). We present *RunPar*, a new allocation algorithm that considers runnables, and not tasks, as the UoS. *RunPar* assigns runnables of the same task to different cores respecting inter-runnable dependencies and forces tasks to execute sequentially following the task ordering of the application's single-core task scheduling. To do so, *RunPar* does not allow runnables from different tasks to be executed in parallel. This approach significantly improves the state-of-the-art techniques under which runnables cannot be executed in parallel.

This runnable scheduling strategy, i.e. the allocation of tasks, guarantees that the composition of tasks and the order in which they are executed in the single-core and in the multi-core ECU remains the same. Therefore the same functional behavior is guaranteed in both platforms.

We evaluate the benefits of *RunPar* on an EMS, a real automotive application (Objective **O4**) that controls the injection time and amount of fuel in a diesel engine and composed of more than one thousand highly connected runnables grouped into twelve tasks (see Section 2.2.2). Our results confirm that *RunPar* effectively increases the performance of EMS tasks by providing an increment of the Central Processing Unit (CPU) capacity of 31% and 42% for the two-core and four-core ECU respectively. This extra capacity can be then exploited for executing new application functionality or other automotive applications, which ultimately contributes allocating more functionality per ECU, reducing size, weight and power costs.

We consider *RunPar* a necessary step towards porting current legacy software to many-cores, for exploiting the many-core performance potential while containing verification effort (Objective **O5**). The use of runnables as UoS implies minimum modifications at AR-OS level: The scheduling tables used in the AR-OS to execute tasks are extended to incorporate the core, the order and the time in which runnables are executed, so inter-runnable dependencies are respected.

How to better exploit multi-core ECU and GRP capabilities for new AUTOSAR applications to minimize inter-runnable communications, and so increase parallelism, is a challenging problem that is out of the scope of this thesis and part of our future work.

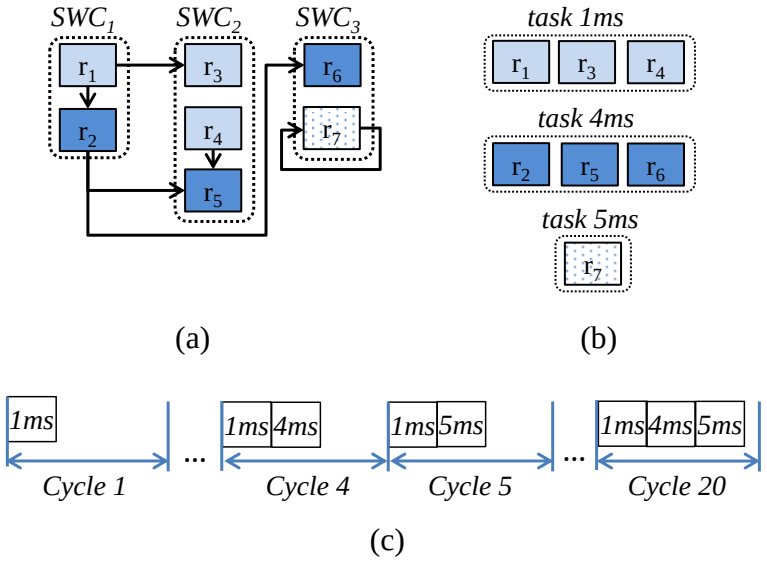


FIGURE 6.2: Part of the runnable flow-graph of an automotive application composed of 3 SWC, 7 runnables and 3 tasks executed in a single-core processor. (a) Structure of the application; (b) application configuration from an AR-OS point of view; (c) a possible single-core task scheduling of the three tasks.

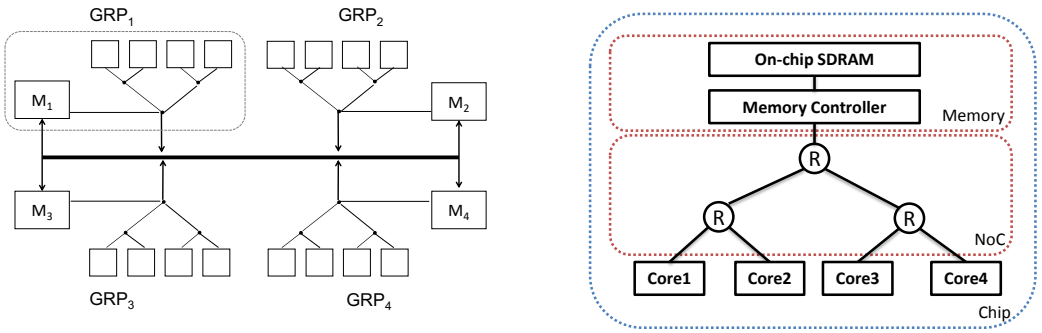
6.2 Background

6.2.1 AUTOSAR Applications

The structural elements of an AUTOSAR application are Software Componentss (SWCs), each containing a set of *runnable entities* (which we call *runnables* for short) that implement the functionality of the SWC. AUTOSAR provides two inter-runnable communication methods: *sender-receiver* and *client-server* ports. The former uses a global shared memory for communication while the latter allows runnables to invoke services from other runnables. In case of sender-receiver, runnables read all input data before starting the execution and results are written back after finishing the execution. No limitations on the number of ports or complexity of components are imposed by the model. All SWC, ports and runnables are known at application configuration time.

During the application configuration phase, runnables are assigned to *tasks*, which are the UoSs of the AR-OS [117]. The execution of runnables follows a model in which they are periodically executed in a recurring *cycle* or triggered by an interrupt. To that end, in legacy applications running on single-core systems, runnables with the same period or interrupt invocation are grouped into the same task, so each task is executed based on either a fixed period or an interrupt. As a result, from an AR-OS point of view, an automotive application can also be defined as a set of tasks with a period or interrupt invocation equal to the period or interrupt invocation of the runnables that compose those tasks.

Figure 6.2 shows the relationship between SWC, runnables and time-triggered tasks for an automotive application composed of three SWC and seven runnables with a period associated to each runnable: r_1, r_3 and r_4 have a 1 ms period, r_2, r_5 and r_6 a 4 ms period and r_7 a 5 ms



(a) Many-core with GRPs (b) Multi-core ECU

FIGURE 6.3: Block diagram of our target architectures.

period. The arrows represent the communication ports, and so *run-after dependencies*, among them. Figure 6.2(b) shows the structure of the application from the AR-OS point of view after the configuration phase. The runnables are grouped into three tasks based on their periods: Tasks *1ms*, *4ms* and *5ms* contain runnables whose period is 1, 4 and 5 ms respectively. Moreover, the order in which runnables are executed within tasks must respect run-after dependencies. In this case runnable r_1 executes before r_3 in task *1ms*. Finally, Figure 6.2(c) shows a possible single-core scheduling of the three tasks. Again, the order in which runnables and tasks are executed must respect the run-after dependencies. Hence, task *1ms* must execute before task *4ms* to respect the dependencies between runnables r_4 and r_5 executed in tasks *1ms* and *4ms* respectively. When a runnable consumes data produced by the runnables from other tasks or by past instances of the same task it belongs to, the sequential execution of tasks guarantees that, when a new task instance starts, all previous tasks instances (including itself) have already finished, and so the data dependence is not violated. This is the case of runnable r_7 in Figure 6.2.

It is important to remark that a different sequence of execution of runnables and tasks, defined in the application configuration, could be defined as well, and still respecting dependencies among runnables. In fact, a different application configuration could result in a more efficient execution in multi-core platforms. However this would imply re-validating completely the new application configuration, which would go against one of the main goals of our proposal, i.e. to contain the cost of validation when migrating from single-core to a multi-core ECU by maintaining same the application configuration. Generating more a efficient application configuration for multi-core ECUs remains as a future work.

Although the structure of the application in Figure 6.2(a) is relatively simple, real automotive applications are composed of a high number of highly connected runnables. This is the case for the our EMS case study, which is composed of **more than one thousand runnables highly connected among them**. Figure 6.1 shows a *part of* the inter-runnable dependencies existing in three of the twelve tasks that compose the EMS (concretely tasks 1, 4 and 8 ms; see Section 6.4.1 for further details).

Schedule tables are used in AUTOSAR to implement statically defined task activation. A schedule table comprises a set of *expiry points*, which are characterized by one or more actions that must occur (activate a task) and an offset in ticks from the start of the table. The AR-OS iterates

through the schedule table and processes an expiry point at the specified tick. The schedule can either be executed *repeated* or only once as a *single-shot*.

6.2.2 Multi-cores and Worst-Case Execution Time (WCET) estimation

Timing analysis of the runnables in a multi-core has to handle contention of accesses to shared resources. Given a runnable under analysis, the contention it suffers can be handled as part of the worst-case response time analysis by factoring in the contention of its co-running runnables which are known at this stage. Alternatively, the contention can be accounted as part of the WCET estimation process by deriving time composable WCET estimates which are made independent of the particular load of co-running runnables. Each approach has its own pros and cons. The former enables deriving tighter estimates, since it builds upon the knowledge of interference, it defies time composability. The latter maintains time composability [17] and the advantages it brings in reducing overall incremental development and verification cost at the expense of higher WCET estimates. We focus on the latter approach for its ability to enable incremental verification.

We consider multi-core architecture resembling single GRP of the processor architecture described in Chapter 3 (see Figure 6.3(a)). It can also be applied to multi-core architecture described in [21, 51, 118] as the target processor in which runnables are allocated. In these architectures the maximum delay a request can suffer when accessing hardware shared resources is bounded by a pre-computed Upper-Bound Delay (UBD) [21]. Architectures based on UBD have been shown to provide competitive results in terms of average and guaranteed-performance (i.e. WCET estimates) with respect to other time-predictable approaches such as Time Division Multiple Access (TDMA) [55](see Section 2.3).

We consider multi-core ECU with 2 or 4 cores, in which each core has a private instruction scratchpad and a data cache. The core is assumed to exhibit no timing anomalies [56] and it is connected to an on-chip SDRAM memory device through a tree Network on Chip (NoC), see Figure 6.3(b). The tree is a wormhole-based topology implementing 3 simple pipelined 2-to-1 routers, so each core requires 2 hops to reach the memory [61]. Such an architecture is similar to the one used in current multi-core ECUs [16].

Under this architecture, there are two sources of interferences that can increase the WCET estimate of runnables, and so tasks: NoC and *memory interferences*. The maximum delay a request to both resources can suffer due to interferences is shown in Equation 6.1. L_{tree} is the tree traversal time. L_{mem} is the memory latency and it is high enough to hide the delay of a round-robin arbitration policy in our tree router, making L_{tree} independent of the number of cores, $ncores$. More details on how L_{tree} and L_{mem} are computed for the setups considered in this chapter are provided in Section 6.4.1.2. Note that, in the worst-case, a memory request is stalled by the rest of cores when accessing the memory.

$$UBD = L_{tree} + (ncores - 1) * L_{mem} \tag{6.1}$$

In order to estimate the WCET for runnables, we use OTAWA static timing analysis tool set [40]. OTAWA is supports analyzing multi-cores and parallel execution [119]. To do so, it considers that the maximum delay a request to both NoC and memory resources can suffer due to interferences is bounded by UBD defined above, so runnables are subject to the worst-case delay they can suffer due to interferences. As a result, the WCET estimates of runnables computed considering UBD are time-composable, i.e. their timing behavior is independent of the runnables executed simultaneously on other cores and insensitive to allocation to the core and independent of the sharing the state of caches with other runnables.

It is important to remark that the approach presented in this chapter is independent of the processor architecture and timing analysis tool considered. Therefore, any particular core count, NoC topology and timing analysis tool, as long as it is possible to derive time composable WCET estimates for runnables.

6.3 RunPar Allocation Algorithm

This section covers the main contribution of this chapter: the *RunPar* allocation algorithm that allows the parallelization of AUTOSAR legacy applications, while maintaining the single-core application configuration, and so allowing reducing the effort of validating applications when migrating from single-core to multi-core ECUs.We consider a partitioned multi-core scheduling approach as it better fits current AUTOSAR standard prescriptions. The use of static cyclic scheduling of runnables is common in AUTOSAR, making the static partition approach very likely to be adopted in a first step when moving towards multi-core ECUs [120].

6.3.1 Problem Definition

AUTOSAR allows describing a wide range of applications to cover most of the functionality required within a car. We focus on applications with the following properties:

- Runnables exchange data through sender-receiver ports using a shared global memory.
- Client-Server communication is always synchronous, i.e. the execution of the server blocks the client until the server finishes.
- Each runnable is assigned to exactly one task.
- Tasks are triggered based on either a fixed period or an interrupt. In case of interrupt-driven tasks, the period for schedulability purposes is defined as the minimum distance between 2 consecutive interrupts. By doing so, it is guaranteed that interrupt-driven tasks can be scheduled in the worst case.
- Time-triggered tasks are not preempted by other time-triggered tasks, and follow the same ordering like in a single-core platform. Interrupt-driven tasks may preempt time-triggered tasks in order to serve the event that generated the interrupt.

These application properties cover a significant range of AUTOSAR applications, including the one considered in this thesis, the EMS. It remains as future work to extend *RunPar* to support applications with other properties, specially task preemption among time-triggered tasks.

Overall, we focus on the *allocation problem* of runnables of an AUTOSAR application in *homogeneous multi-core processors*, considering a *partitioned* scheduling approach in which once a runnable has been assigned to one core it is not allowed to migrate.

We define an AUTOSAR automotive application as a set of tasks $\mathcal{T} = \{\tau_1, \cdots, \tau_k\}$ executed sequentially as defined by the single-core task scheduling. Each task τ_p is represented by a direct acyclic graph $\delta_p = (\mathcal{R}_p, \mathcal{E}_p)$. The nodes in $\mathcal{R}_p = \{r_1, \cdots, r_n\}$ represent the runnables that comprise the task. Each runnable r_i is characterized by a WCET estimate C_i, a period P (the same for all runnables in a task) and a deadline D, assuming *implicit deadlines*, i.e. $D = P$. The utilization u_i of runnable r_i is defined as $\frac{C_i}{P}$, where $0 \leq u_i \leq 1$. The edges in $\mathcal{E}_p$ represent communications between runnables: An edge $e_{i \to j} \in \mathcal{E}_p$ represents any communication method implemented from runnable r_i to runnable r_j, so r_j cannot start until r_i finishes. The period of τ_p is equal to the period of all of its runnables in $\mathcal{R}_p$.

An *application configuration* Ψ is defined as the single-core task scheduling $\mathcal{T}$ and the runnable-to-task mapping per each task $\tau_p = (\mathcal{R}_p, \mathcal{E}_p)$. *RunPar* assigns the n runnables in $\mathcal{R}_p$ from a task to a set of m identical cores $sc = (c_1, \cdots, c_m)$ respecting the run-after dependencies defined in $\mathcal{E}_p$ and the application configuration Ψ. Concretely, the allocation algorithm generates a static partition $\Phi_p = (\varphi_1, \cdots, \varphi_m)$ in which a subset of $\mathcal{R}_p$ is assigned to each core. Thus, $\varphi_k = \mathcal{R}_p^k \subseteq \mathcal{R}_p$ assigns $\mathcal{R}_p^k$ to core c_k with a *cumulative utilization* defined as $u_{sum}^k = \sum_{r_i \in \mathcal{R}_p^k} u_i \leq 1$. Each runnable can be assigned to only one core. Moreover, given any two runnables $r_i \in \mathcal{R}_p$ and $r_j \in \mathcal{R}_p$, *RunPar* guarantees that r_j is allocated after r_i finishes if exists $e_{i \to j} \in \mathcal{E}_p$. Furthermore, in case a runnable consumes data from two or more runnables, it must be allocated after all of them are guaranteed to complete.

The utilization of a task τ_p, given by the longest dependent runnable chain, is the maximum cumulative utilization of all cores in Φ_p, and it is defined as $u_{\tau_p} = max(\forall \varphi_k \in \Phi_p \mid u_{sum}^k)$. Similarly, the WCET estimate of a task τ_p is the maximum sum of C_i allocated to a core in Φ_p, and it is defined as $WCET_{\tau_p} = max(\forall \varphi_k \in \Phi_p \mid \sum_{r_i \in \mathcal{R}_p^k} C_i)$.

In a partitioned scheduling scheme, once runnables are allocated to cores, an on-line *uniprocessor* scheduling algorithm is used. In our case, the on-line scheduler must guarantee that, when a runnable starts executing all its predecessor runnables upon which it depends have already finished.

RunPar is compatible with static time-triggered schedulers, like in AR-OS. AR-OS implements *scheduling tables* [117] that define the starting point and the order in which tasks are activated. Section 6.3.3 describes the changes required at AR-OS level to schedule, not only tasks but runnables as defined by *RunPar* in Φ so the starting point of runnables guarantees the fulfillment of inter-runnable dependencies. Extension of RunPar to support dynamic priority-based schedulers (e.g. rate monotonic) remains as future work.

6.3.2 Mapping Runnables to Cores

RunPar considers the set of tasks $\mathcal{T} = (\tau_1, \cdots, \tau_k)$ that form the application, and allocates the runnables of each task τ_p into m cores. Runnables from different tasks are not allowed to be executed in parallel, forcing tasks to be executed sequentially and so respecting the single-core task scheduling. This section presents our runnable-to-core allocation algorithm, which is called for each of the tasks defined by the application developer in $\mathcal{T}$.

Figure 6.4 shows the pseudo-code implementation. The algorithm takes as input the task $\tau_p = (\mathcal{R}_p, \mathcal{E}_p)$ (i.e. Figure 6.1) and a set of m cores $sc = (c_1, \cdots, c_m)$. As output it provides a valid allocation Φ_p.

6.3.2.1 Runnable classification

RunPar starts classifying runnables into two different types: *dependent runnables* (dR) and *independent runnables* (iR), lines 2 and 3 in the algorithm. A runnable r_i in $\mathcal{R}_p$ is dependent if there exists a runnable r_j in $\mathcal{R}_p$ and the edge $e_{i \to j}$ or $e_{j \to i}$ in $\mathcal{E}_p$. In other words, the runnable r_i produces (or consumes) data that is consumed (or produced) by r_j, creating a run-after dependence among them. Similarly, runnable r_i is independent if for all runnables in $\mathcal{R}_p$, neither exist edges $e_{i \to j}$ nor $e_{j \to i}$ in $\mathcal{E}_p$. In Figure 6.2, runnables r_4 and r_7 from tasks $1ms$ and $5ms$ respectively, are classified as independent runnables. The remaining ones are classified as dependent.

It is important to recall that runnables can also consume data produced by other tasks or by past instances of the same task they belong to. Such a dependence is not taken into account because the sequential execution of tasks guarantees that when a new task instance starts, all previous tasks instances (including itself) have already finished, and so the data dependence is not violated. This is the case of runnable r_7 in Figure 6.2.

6.3.2.2 Sorting criteria

After the classification of runnables, dR and iR are sorted. There are several criteria that can be used for sorting: deadline, period, utilization, density. Since all runnables of a task share the same deadline and the same period, utilization (u) remains the criterion to use. Moreover, we introduce a new sorting criterion: the combined utilization (cU) of a runnable, which is computed as the highest sum of utilization across the chains of dependencies starting from the observed runnable. For instance, in case of runnable r_2 from task *4ms* of Figure 6.2, the combined utilization cU_2 is equal to $max(u_2 + u_5, u_2 + u_6)$, while the cU_1 of runnable r_1 from task *1ms*, is $u_1 + u_3$. The use of cU guarantees that the longest chain of dependent runnables is allocated first.

The function in line 5, sorts dR and iR runnables based on the sorting criteria, i.e. u or cU. Note that for independent runnables cU equals to u. In Section 6.4, we discuss the impact of sorting criterion on the effectiveness of the algorithm.

6.3.2.3 Bin packing heuristics

The search for optimal allocation of runnables to cores is an NP-hard problem [121] which introduces the need for using non-optimal heuristics in order to do the allocation. We evaluate worst-fit and first-fit decreasing heuristics[2] [122–124]. In case of the worst-fit decreasing heuristic, runnables are allocated to the least-occupied cores, i.e. cores in which u^k_{sum} is smallest; first-fit heuristic allocates runnables in the first core they fit, i.e. u^k_{sum} remains smaller or equal than 1. Moreover, runnables are sorted in decreasing order by their utilization/combined utilization, which prioritizes runnables with higher utilization to be allocated first. Section 6.4 provides a quantitative comparison of applying different heuristics. Our proposed algorithm is compatible with any bin-packing heuristic as long as run-after runnable dependencies are fulfilled.

After creating Sorted dependent Runnable (SdR) and Sorted independent Runnable (SiR) sets, *RunPar* spawns into two different subphases: allocation of dependent and independent runnables.

6.3.2.4 Dependent Runnables

The algorithm works as follows. The first runnables to be allocated are those that produce data, but do not consume data from any other runnable from the same task (line 9). Runnables are allocated to cores using the bin-packing heuristic (lines 10 and 11). In Figure 6.2, the first allocated runnables when processing tasks *1ms* and *2ms* are r_1 and r_2 respectively.

In the subsequent step, the rest of consumer/producer runnables are allocated respecting run-after dependencies, i.e. the runnable is not allocated until all its precedence dependent runnables are allocated (lines 13-14). Moreover, the time at which a runnable starts executing must be after all runnables producing its input data finish. Therefore, given a runnable r_i and its producer runnable finishing the latest r_j, r_i is allocated after r_j finishes its execution, considering its WCET estimate (C_j) to guarantee that the data will be available. To do so, the algorithm searches for the largest cumulative utilization among the cores in which the dependent runnables are allocated (lines 13-21). Function *allocated_cUtil* returns the values of u^k_{sum}, after allocating runnable r_j to that core. Then, with function *binpack_startdef* (line 22) we select the core in which r_i fits best according to the chosen heuristics, starting its execution after $r_{largest}$ finishes, i.e. at *maxdep* (lines 18 and 19).

In case the cumulative utilization of the core c in which r_i will be allocated (u^c_{sum}) is smaller than *maxdep* – i.e. the cumulative utilization of the core where is $r_{largest}$ allocated is higher than u^c_{sum} after allocation of $r_{largest}$ – an *idle runnable* (r_{idle}) is inserted (lines 23-26) with its corresponding utilization (u_{idle})(line 24). In other words, idle runnables delay the start of runnables until all their input data are guaranteed to be available. Idle regions will then be used to allocate independent runnables as explained in the next section.

[2]Other bin-packing heuristics such as best-fit and next-fit are a variant of the two heuristics considered, producing similar results.

***RunPar* Allocation Algorithm**
Input $\tau_p = (\mathcal{R}_p, \mathcal{E}_p)$: A task of the application,
$sc = (c_1, \cdots, c_m)$: total number of cores
Output $\Phi_p = (\varphi_1, \cdots, \varphi_m)$: A valid allocation of $\mathcal{R}_p$ into sc

```
1 Φ_p = ∅
2 dR = ∀r_i ∈ R_p | ∃r_j ∈ R_p, ∃e_{i→j} ∨ e_{j→i} ∈ E_p
3 iR = ∀r_i ∈ R_p | ∀r_j ∈ R_p,  ∄e_{i→j} ∧ e_{j→i} ∈ E_p
4
5 <sdR, siR>=sort_runnables(dR, iR, criterion)
6
7 // First-phase:Allocation of dependent
8 // runnables
9 forall (r_i ∈ sdR | r_i ∉ Φ_p and ∄r_j ∈ sdR, e_{j→i} ∈ E_p) do
10   c = binpack(r_i, heur_1)
11   Φ_p += allocate(c,r_i)
12 endfor
13 forall (r_i ∈ sdR | r_i ∉ Φ_p and ∃r_j ∈ Φ_p | e_{j→i} ∈ E_p
14               and ∄r_k ∈ Φ_p | e_{k→i} ∈ E_p) do
15   maxdep = 0
16   forall r_j ∈ φ_k  do
17     if(maxdep < allocated_cUtil(r_j)) then
18       maxdep = allocated_cUtil(r_j)
19       r_largest = r_j
20     endif
21   endfor
22   c = binpack_startdef(r_i,maxdep, heur_1)
23   if(maxdep − u^c_sum > 0) then
24     u_idle = maxdep − u^c_sum
25     Φ_p += allocate (c,r_idle)
26   endif
27   Φ_p += allocate (c,r_i)
28 endfor
29
30 // Second-phase:Allocation of independent
31 // runnables
32 forall(r_i ∈ siR | r_i ∉ Φ_p) do
33   if(∃r_idle ∈ Φ_p and u_i ≤ u_idle) then
34     c = core(r_idle)
35     Φ_p += allocate(c,r_i)
36     if(u_idle − u_i = 0) then
37       remove(r_idle, Φ_p)
38     else
39       u_idle = u_idle − u_i
40     endif
41   else
42     c = binpack (r_i)
43     Φ_p += allocate(c,r_i)
44   endif
45 endfor
46
47 return Φ_p;
```

FIGURE 6.4: Pseudo-code implementation of the allocation algorithm.

6.3.2.5 Independent Runnables

The allocation of independent runnables occurs once all dependent runnables have been allocated (lines 30 to 45). The reason is that independent runnables have the freedom to be allocated at any point in time within its corresponding period. As already pointed above, it is important to remark that independent runnables are in fact dependent runnables that consume data produced by other tasks or by past instances of the same task they belong to. The EMS does not contain any purely independent runnable.

RunPar uses the set of independent runnables sorted by their utilization (which equals to combined utilization), so independent runnables with higher utilization are allocated first. The algorithm first checks if a runnable fits within any idle region (line 33). If it fits, the runnable is allocated within the same core and time slot assigned to the idle runnable (lines 34 and 35), and

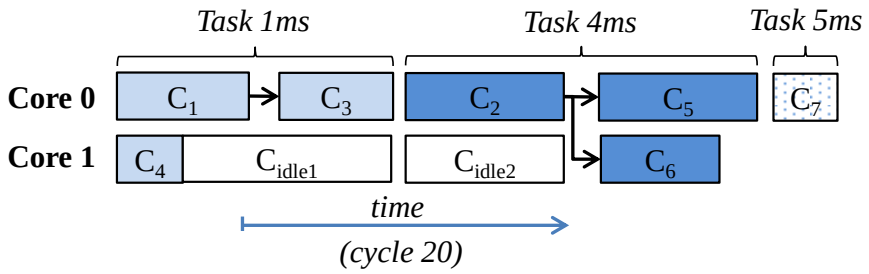


FIGURE 6.5: Valid allocation (Φ) of the automotive application presented in Figure 6.2 in a two-core processor, executing cycle 20. C_i is the WCET estimate of runnable r_i.

the utilization of the idle runnable is reduced (line 39) or even eliminated (line 37). If it does not fit, the runnable is allocated using a bin-packing heuristic (lines 42 and 43).

6.3.3 Allocation Algorithm Solution: Φ

If a valid allocation Φ_p of task τ_p is found, each runnable (including idle regions) is assigned to a core (line 47) and executes within its corresponding period. Figure 6.5 shows the resultant Φ_{1ms}, Φ_{4ms} and Φ_{5ms} of *1ms, 4ms* and *5ms* tasks respectively, when applying *RunPar* to the application presented in Figure 6.2, executing in a two-core processor. C_i is the WCET estimate of runnable r_i; C_{idle} is the time slot in which the core executes an idle runnable.

In order to support *RunPar*, AR-OS scheduling tables have to be extended to schedule the runnables from task τ_p as defined by Φ_p. To that end, each task entry in the scheduling table is extended with a new *runnable scheduling table* that defines the starting point, the order and the core of each of the runnables that form the task. It is important to remark that runnables are scheduled based on their WCET estimates (C), guaranteeing that no race conditions can occur, i.e. consumer runnables do not start executing until all their dependent producer runnables finish, even if they execute for their WCET estimates. In order to ensure that runnables do not start until the WCET estimate of runnable allocated before it expired, an idling function as proposed in PharOS [125] can be used.

6.3.4 Validating the Single-core Task Scheduling

Our allocation strategy reduces the WCET estimate of tasks, by exploiting runnable-level parallelism of application's task, but does not necessarily reduce the response time of the overall application. The reason is that our strategy maintains the single-core task scheduling, and so the starting point and order in which tasks are executed remains the same. However, the overall time the processor is used by the application is reduced because tasks execute faster, providing extra computational space in the task scheduling to allocate new application functionality, allocate other tasks from different applications or even to reduce the clock frequency and so reduce the energy consumption of the ECU [126].

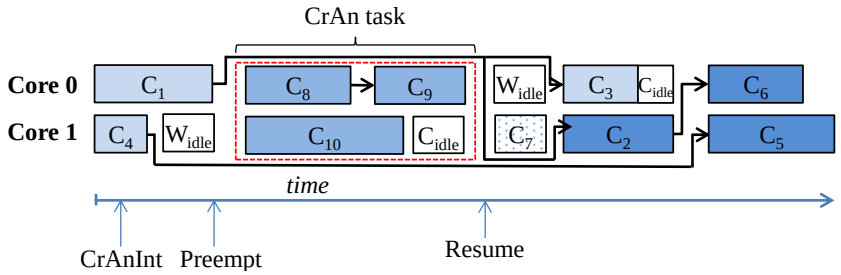


FIGURE 6.6: The CrAn interrupt-triggered task preempting time-triggered one.

The single-core task scheduling remains valid in the multi-core ECU if the resultant task utilization after parallelizing runnables (u_{τ_p}), is smaller than or equal to the task utilization when executing runnables sequentially ($useq_{\tau_p}$).

It could be the case, however, that the WCET estimate reduction obtained by executing runnables of a task in parallel is not enough to hide the overhead introduced due to interferences when accessing the hardware shared resources [21, 51], making u_{τ_p} be higher than $useq_{\tau_p}$. In this case, it is required to validate that the single-core task scheduling, and the increment of the utilization of a task can be compensated by the utilization reduction of the other tasks.

If u_{τ_p} is higher than $useq_{\tau_p}$, the system can emulate the single core execution. All runnables of the tasks are executed sequentially in one core while other cores are idle and do not execute anything. Since we know that there will be no interference from other cores, we can safely use WCET estimates obtained for single core execution (assuming $UBD = 0$) and by doing so decrease the WCET estimate of the task.

6.3.5 Execution of interrupt-driven tasks (*CrAn* task)

It is common in AUTOSAR applications that interrupt-triggered tasks, i.e. triggered by an external event, are served as soon as possible. An example is the *crank-angle* task (*CrAn*) in the EMS, which is activated based on the position (angle) of the camshaft of the engine. Since the occurrence of this task depends on the engine rotation speed, which is not constant but has a maximum value (in our case 4000 rpm), we derive that the minimum time between 2 arrivals of *CrAn* task is 1.25ms, making this task sporadic [127].

Sporadic tasks in table driven schedulers are supported with use of acceptance tests, checking whether there is sufficient slack time in the frames to follow before tasks deadline. In our case, since we cannot allow acceptance test to fail and our time-frames are of 1ms, in each of the frames needs to be enough slack to allow the execution of *CrAn*.

Moreover, in order to guarantee that the CrAn task is served as soon as possible, it can preempt other time-triggered tasks being executed at the time the interrupt arrives. In order to provide support to time-triggered task preemption, we need to do the following:

- Preempt at the end of execution of runnables, preserving time composability of runnable WCET estimates and so the allocation provided by *RunPar*, as the timing analysis is performed at runnable level.
- Wait for all currently executed runnables to finish, before starting *CrAn* and prevent subsequent runnables of interrupted tasks to start. This guarantees that the sequentially execution of time-triggered tasks is maintained.
- When *CrAn* starts executing all cores must be available, as RunPar allocation assumes that all cores are available to schedule runnables from a new task. We guarantee this is by applying the mechanism presented in [128].
- For the same reason of previous point, the preempted task is not resumed until all cores have finished executing runnables of *CrAn* task. At this point it is guaranteed that producers of remaining runnables have completed due to the second point.
- When the preempted task is resumed, the runnables blocked after the interrupt arrived must be resumed. In order to maintain the sequence of execution as defined by *RunPar* idle slots may be required to guarantee the same runnable scheduling. The size of those idle slots is equal to the difference between the ending time of the last executed runnable before the preemption, and the ending time of the first runnable that got stopped due to the interrupt.

Figure 6.6 shows how *CrAn* task is handled if it arrives at the same time-frame as a task executes. The interrupt that triggers *CrAn* task arrives at *CrAnInt*. In this case, our system lets runnables being executed to complete, i.e. runnables C_1 and C_4, preventing further execution of runnables from current task, i.e. runnable C_7. When runnables finish their execution, it starts executing runnables from *CrAn* as defined by *RunPar*. When *CrAn* task finishes, it restores the previous context, adding necessary idle slots (labeled as W_{idle}), in order to make execution of *CrAn* invisible to the preempted task and respect dependencies of the preempted task. The runnable C_7 is resumed first in core 1, introducing an idle slot of C_1 ending time minus C_4 ending time cycles in core 0. By doing so the execution of the task can be resumed at runnables C_3 and C_4 transparently to the execution of *CrAn*. The introduction of the idle slot is controlled by the AR-OS, by measuring the ending points of the different runnables at the point the interrupt arrives.

Overall, the overhead introduced due to preemption equals to the WCET estimate of 2 task context switches (i.e. the starting of the interrupt-triggered task and the resume of the preeempted time-triggered task) plus WCET estimate of the longest runnable scheduled to that frame (due to waiting for all cores to finish currently executed runnables).

6.4 Results

6.4.1 Experimental setup

6.4.1.1 EMS application

Our allocation algorithm has been evaluated with a real automotive application, an EMS (see Section 2.2.2). An EMS is a typical automotive embedded real-time system in which the amount of fuel and the injection time are fundamental for smooth revolutions of the engine. The injection time and fuel amount depend on the state and the rotation speed of the engine, which changes continuously during operation. EMS is composed of eleven time-triggered tasks, with periods of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms, and a crank-angle interrupt-triggered task, with a minimum period of 1.25 ms corresponding to the maximum engine rotation speed (i.e. 4000 rpm, see Section 6.3.5).

6.4.1.2 WCET analysis tool and Processor Setup

In order to compute time composable WCET estimates (C) of runnables, we use the static timing analysis tool OTAWA (see Section 2.3). OTAWA models the multi-core processor architecture presented in Chapter 3 and Section 6.2.2 in which every request that accesses the NoC and the on-chip memory is delayed by UBD, so the runnable is subject to the worst-case delay that it can suffer due to interferences, and so WCET estimates are time composable. We consider 2-core and 4-core processor configurations, with private per-core scratchpads for instructions and write-through data caches. For both processor configurations cores are connected through a tree NoC to the on-chip Random Access Memory (RAM) memory. We consider $L_{tree} = 1$ cycles for a 2-core processor, $L_{tree} = 2$ for a 4-core processor (i.e. each core has to traverse 1 and 2 routers for 2- and 4-core processors respectively) and 1-cycle routers. The memory latency is set to $L_{mem} = 10$ cycles. This configuration provides a $UBD = 11$ cycles for the 2-core architecture and $UBD = 32$ cycles for the 4-core architecture (see Section 6.2.2 for further details).

The approach presented is independent of the processor architecture and the timing analysis method, so other architectures and tools can be used to compute time composable WCET estimates of runnables.

6.4.1.3 Metrics

In order to evaluate our allocation algorithm, we consider the $WCET_\tau$ *speed-up* metric, defined as $\frac{seqWCET_\tau}{parWCET_\tau}$, where $seqWCET_\tau$ is the WCET estimate of task τ executing runnables sequentially, and $parWCET_\tau$ is the WCET estimate of task τ executing runnables in parallel as defined in Φ by *RunPar*. $seqWCET_\tau$ is given by the sum of C of all runnables that compose τ, assuming $UBD = 0$ in the computation of C of each runnable, , i.e. runnables do not suffer any delay due to interferences. $parWCET_\tau$ is given by the sum of C of the longest chain of dependent

runnables as defined by *RunPar*, assuming $UBD = 11$ and $UBD = 32$ for the 2-core and 4-core processor configurations respectively in the computation of C.

Moreover, in order to evaluate the maximum runnable-level parallelism that our algorithm is able to exploit, we use the *no interference* $WCET_\tau$ *speed-up* metric, in which the effect of interferences is assumed 0. To do so, the $parWCET_\tau$ of the 2-core and the 4-core processor architectures are computed assuming no interferences, i.e. $UBD = 0$. Considering WCET estimates in which interferences are not accounted, allows discounting the pessimism introduced by the timing analysis tool because of sharing processor resources such as the NoC and the memory as well as the actual contention in shared resources.

6.4.2 Choosing the appropriate heuristics

Table 6.1 shows average WCET speedup of EMS tasks when applying worst-fit and first-fit bin-packing decreasing heuristic (labeled as *WF* and *FF* respectively), and applying utilization and combined utilization sorting criteria (labeled as U and cU respectively). The same heuristics are applied for allocating both dependent and independent runnables. The results show that worst-fit guarantees a better runnable-to-core load balance, which in turn provides more parallelism among runnables, outperforming first-fit. This is so because data dependencies among runnables reduce considerably the possibilities in which runnables can be allocated, and so worst-fit allows selecting the less loaded core, increasing the scheduling opportunities. We can also notice that when using combined utilization as the sorting criterion, we can obtain an extra bit of improvement in almost all of the cases apart from the case of worst-fit heuristics with 4-core processors, where the results are roughly the same for both criteria.

Table 6.1: Average WCET speed-up of EMS' tasks

	FF+U	WF+U	FF+cU	WF+cU
2-core	1,04	1,32	1,06	1,35
4-core	1,04	1,42	1,06	1,43

Table 6.2 shows the average WCET speedup of EMS tasks when we use different bin-packing heuristics for allocating dependent and independent runnables. Concretely, the table denotes *(W+F)* as the combination of applying worst-fit for allocating dependent runnables and first-fit for allocating independent; *(F+W)* denotes the opposite. As expected, applying worst-fit to dependent runnables results in a better performance than applying to independent. Overall, the combination of heuristics cannot match worst-fit in terms of WCET speed-up of EMS tasks. Similarly, the use of combined utilization allows obtaining an extra bit of improvement in almost all of the cases apart from the 4-core processor *W+F* strategy.

Table 6.2: Average WCET speed-up of EMS tasks when combining heuristics

	(F+W)+U	(W+F)+U	(F+W)+cU	(W+F)+cU
2-core	1,12	1,25	1,14	1,26
4-core	1,13	1,38	1,16	1,35

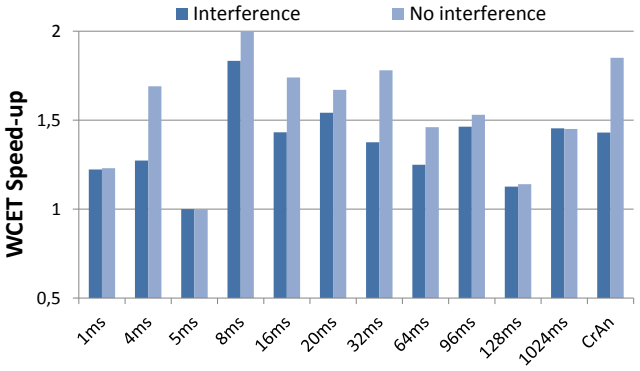


FIGURE 6.7: WCET speed-ups of EMS tasks in a 2-core processor architecture, in which the WCET estimation accounts and discards the impact of *interferences* (labeled as *interferences* and *no interferences* respectively).

In the rest of this section, we present detailed results when using *RunPar* algorithm with worst-fit heuristics and using combined utilization as the sorting criterion.

6.4.3 WCET Speed-up of EMS tasks

Figure 6.7 and Figure 6.8 show the WCET speed-ups of EMS tasks obtained with *RunPar* in 2-core and 4-core processor architectures respectively (the higher the better). In both cases, *RunPar* is used with (1) WCET estimates that consider the effect of interferences (labeled as *interferences*) and (2) WCET estimates assuming no interferences (labeled as *no interferences*).

In the 2-core processor architecture (Figure 6.7), and assuming the impact of interferences, *RunPar* is capable of exploiting the performance opportunities of the multi-core ECU, improving the WCET performance of almost all tasks with respect to a single-core ECU. The speed-up of the 8 ms task achieves a significant 1.8x, being close to the ideal speed-up in a 2-core processor architecture, i.e. 2x. Tasks 16, 20, 96 and 1024 ms, and the *CrAn* task also exhibit high speed-ups, around 1.5x. Such a WCET speed-up represents a WCET reduction of 45% in case of 8ms task, and around 33% for tasks 16, 20, 96 and 1024 ms.

When we compare the speed-ups obtained without accounting for the impact of interferences on the computation of the WCET estimates, we observe the performance degradation of multi-core execution due to sharing the NoC and the memory processor resources. Such a degradation is further augmented by the pessimism introduced by the timing analysis tools. In our case, OTAWA assumes that each memory request is delayed by UBD cycles. Hence, if no interferences are assumed, the 8 ms task achieves the maximum speed-up, i.e. 2x, and *CrAn* and 4, 16 and 32 ms tasks increase the speed-up close to 1.8X for the 2-core processor.

In any case, achieving maximum levels of parallelism not only depends on the effect of interferences and quality of the allocation algorithm, but also on the characteristics of the task, e.g. number of runnables, inter-runnable dependencies. This is the case of tasks 1, 20, 64, 96, 128 and 1024 ms tasks, in which the WCET estimates are not affected much by interferences. The 5 ms task is composed of a single runnable which is executed in a single core, and so does not benefit at all of parallel execution.

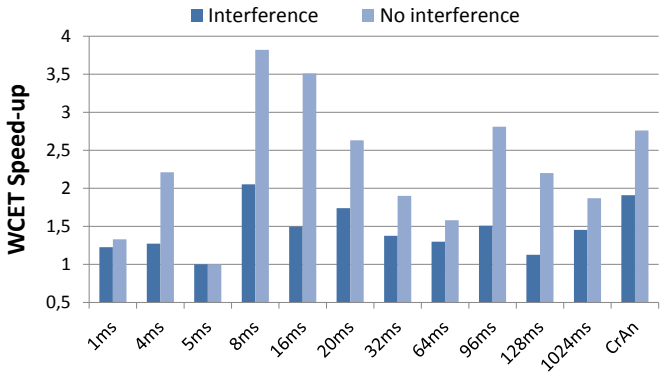


FIGURE 6.8: WCET speed-ups of EMS tasks in a 4-core processor architecture, in which the WCET estimation accounts and discards the impact of *interferences* (labeled as *interferences* and *no interferences* respectively).

In the 4-core processor architecture (Figure 6.8), the impact of interferences on the WCET estimates increases significantly (OTAWA assumes UBD equal to 32 cycles), which makes the performance benefits brought by multi-core execution being higher than those for the 2-core processor but their scalability is poorer with respect to the number of cores, as the WCET degradation due to interferences grows noticeably. As a result, the speed-up of EMS tasks slightly increases when moving from a 2-core to a 4-core processor architecture, achieving speed-ups of 2.1x in case of 8 ms and *CrAn* tasks, and 1.7x in case of 20 ms task. Such speed-up represents a WCET reduction of 53% in case of 8 ms and *CrAn* tasks, and 41% for 20 ms task.

When discarding the effect of interferences in WCET estimates, the speed-up of 8, 16, 96 ms and *CrAn* tasks increases significantly, with speed-ups of 3.8x, 3.5x, 2.8x and 2.7x respectively. In the case of 1, 64 and 1024 ms tasks the WCET speed-up is not affected much by interferences. In fact, if we compare the WCET speed-up of these three tasks obtained in the 2-core and the 4-core processor architectures, we observe the exact same performance.

We conclude that *RunPar* effectively exploits the maximum runnable-level parallelism exposed in EMS tasks in a multi-core ECU. Unfortunately, the impact of interferences due to shared resources reduces a bit the benefits brought by multi-core execution. Such negative effect is increased as the number of cores in the ECU increases. Results could be improved by using a timing analysis tool delivering less pessimistic WCET estimates or a processor architecture where the maximum effect of interferences (UBD) is lower. Investigating new timing analysis techniques and processor architectures to reduce the impact of interferences is part of the future work.

6.4.4 Increasing Overall Available CPU Capacity

The parallel execution of runnables reduces the WCET estimates of tasks, reducing the task utilization (u_τ) and so reduces the CPU capacity required by the EMS application with respect to the single-core execution. This extra capacity can be then exploited for executing new application functionality, other automotive applications or even reducing the CPU frequency to reduce the energy consumption.

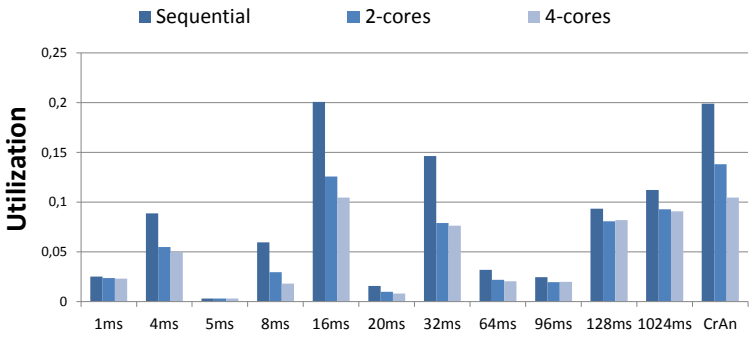


FIGURE 6.9: Utilization of EMS tasks being allocated on a single-core, 2-core and 4-core ECU labeled as *sequential*, *2-core* and *4-core* respectively).

Figure 6.9 shows the utilization of EMS tasks when being allocated on a single core ECU (labeled as *sequential*) and on a 2-core and 4-core ECU using the *RunPar*. Note that in all processor configurations, the task scheduling is the same as our allocation strategy maintains the single-core task scheduling.

The utilization (u_τ) of all EMS tasks (except *5ms* task because it contains one single runnable) is reduced. It is of special interest the tasks with highest utilization i.e. those whose WCET estimate is higher with respect to their period. This is the case of tasks 16 and 32 ms and *CrAn* with utilization reductions of 8, 6 and 6 percentage points (pp) respectively in case of the 2-core ECU (a reduction of the utilization of, for instance, 8 pp in case of 16 ms task means that the utilization goes from 0.20 to 0.12) . In the case of 4-core ECU, these 3 tasks have utilization reduced by 10, 7 and 9 pp respectively. These three tasks are, in fact, the longest tasks of the EMS application with respect to their period, and so are the tasks that benefits the most of a reduction to their WCET estimates. This is not the case of 8 ms task, which despite being the task that achieves the highest WCET speed-up (see Figure 6.7 and Figure 6.8), this is translated in an utilization reduction of only 3 pp.

Overall, the reduction on EMS tasks utilization represents an increment of the CPU capacity of 31% and 42% for the two-core and four-core ECUs respectively. Such an extra capacity can be then re-used for executing new application functionality or scheduling other applications using AUTOSAR recommendations for executing in a multi-core ECU environment [24]. Alternatively, it can be used for applying dynamic voltage and frequency scaling techniques, so the energy consumed by ECU can be reduced [129].

6.5 Related Work

There are two main strands of research in multi-core scheduling [130], reflecting the ways in which processes are allocated to cores. Partitioned approaches allocate each process to a single core, dividing the problem into one of process allocation (bin-packing) followed by single processor scheduling. In contrast, global approaches allow processes to migrate from one core to another at run-time. Following AUTOSAR standard prescriptions, we have considered partitioning approach.

In partitioned approaches, finding an optimal allocation is an NP-hard problem in the strong sense [121] and so non-optimal solutions derived from the use of bin-packing heuristics are typically used [122–124]. In [131], authors evaluate the impact of different bin-packing heuristics on mixed-criticality systems. They show that first-fit heuristics achieve better results when allocating low-criticality tasks [131] after the allocation of high-criticality tasks with worst-fit heuristics. However, in the case of EMS application, the use of worst-fit heuristics outperforms both first-fit and the combination of the two heuristics.

Stochastic approaches such as genetic algorithms [132] and simulated annealing [133] have been used with different degrees of success in different domains. These search algorithms are general enough to be adapted to many different problems to look for the best solution. However, adapting them is not a trivial task. For instance, in the case of genetic algorithms one needs to define the fitness function to quantify how good each solution is, the alphabet used to encode solutions, population size, crossover function across individuals, mutation probability and convergence criteria. To the best of our knowledge stochastic solutions have not been devised yet for our problem.

Most scheduling works proposed in the Critical Real-Time Embedded Systems (CRTES) domain consider independent processes that do not communicate among them. Along this line, Monot et al. [114] presented recently a scheduling algorithm for multi-source AUTOSAR applications on multi-core ECUs. Their approach groups all runnables with inter-runnable dependencies into a single task, allowing tasks to be scheduled independently. In the next step, a runnable scheduling is build on each core independently of other cores. Overall, one of the main objectives of the approach is obtaining uniform utilization of the cores during the application execution. The scheduling algorithm presented in [114] has two main differences with our allocation strategy. First, it requires changing the application configuration, i.e. it reassigns runnables to tasks, creating a new runnable and task scheduling, which causes the need for re-validating legacy applications. Second, it assumes little dependencies among runnables so a sufficient number of tasks can be created to exploit parallelism in multi-core ECUs. This is not the case of the EMS in which almost all runnables have inter-runnable dependencies (see Figure 6.1). In fact, applying this approach to the EMS would lead to schedule all runnables into a single core.

Faragardi et al. [115] presented a scheduler for AUTOSAR applications on multi-core ECUs having as scheduling criterion the minimization of the worst-case communication delays among runnables scheduled in different cores. This solution assumes that the runnable-to-task mapping that minimizes the communication among cores is already given, and so it focuses only on the task scheduling. This approach requires changing the application configuration, causing the need for re-validating legacy applications.

Wieder and Brandenburg [134] target a real-time partitioned scheduling of independent tasks (not AUTOSAR related) in which accesses to the shared resources are protected with spin locks. They provide an ILP formulation for computing optimal partitioning w.r.t. to schedulability analysis of the MSRP protocol [135] and resource-aware partitioning heuristics called GreedySlacker. GreedySlacker tries to allocate a task in such a way that it maximizes the minimum slack left in the cores after the allocation of that task, and by doing so evenly spreads the workload across the cores. Such an approach cannot be applied to allocate runnables, as they have run-after data

dependencies. Applying it to the task scheduler would imply an effort in changing the application to implement the supported synchronization mechanism (spin-locks), resulting in the need for re-validation.

An interesting work to mention is the one presented by Sinnen et al. [136] targeting general purpose processor architectures. In [136] a task scheduling approach is proposed to make the scheduler aware of the cost of inter-processor communication in a general purpose architecture. The scheduler is a variant of the list scheduling in which tasks are allocated in two phases following a similar approach to the one used by *RunPar*, i.e. allocating first dependent tasks and then independent tasks. The main difference with *RunPar* is that the purpose of [136] is to improve the execution time of a parallel application in a general purpose architecture by minimizing the communication delay.

To the best of our knowledge, this is the first allocation algorithm that exploits runnable-level parallelism instead of task-level parallelism and maintains the application configuration. Schneider et al. [116] pointed that the parallel execution of tasks can break the mutual exclusion relations between the critical sections present in applications configured to be executed in single-core processors, leading to race conditions. Therefore, considering runnables as the UoS is the best approach to avoid race conditions.

6.6 Conclusions

This chapter presents a new allocation strategy in which the single-core configuration of AUTOSAR applications (objective **O2**), i.e. its runnable-to-task mapping and single-core task scheduling, is maintained when migrating from single-core to multi-core ECUs, so the effort of re-validating the applications is minimized (**Goal 3**).

To do so, we present *RunPar*, a new allocation algorithm for AUTOSAR automotive applications with runnables highly connected among them, which considers runnables, and not tasks, as the UoS. *RunPar* maintains the single-core task scheduling applying the following allocation strategy: Only runnables from the same task are allowed to execute in parallel,.This allocation strategy requires minimum modifications at AR-OS level.

RunPar is independent on the heuristics used for allocation of runnables to cores, though in the case of EMS application, it shows the best results when using the worst-fit decreasing heuristic which prioritizes runnables with higher *combined utilization*, so the longest chains of dependent runnables are allocated first.

The allocation has been evaluated with a real automotive application, an Engine Management System (EMS) (objective **O4**), which controls the injection time and amount of fuel in a diesel engine and is composed of more than one thousand highly connected runnables, grouped into twelve tasks.

Results show that *RunPar* algorithm reduces on average, WCET estimates of the EMS tasks by approximately 26% and 30% in the case of 2-core and 4-core architectures respectively,

representing a WCET speed-up of 1.35x an 1.43x. Such a WCET speed-up translates into an increment of the CPU capacity of 31% and 42% for the two-core and four-core ECUs respectively.

Therefore, *RunPar* stands as a necessary step for porting AUTOSAR automotive applications from single-core ECUs and exploiting the potential performance of multi-core ECUs (objective **O5**).

Chapter 7

Inter-GRP Scheduling Strategy for Real-time Applications on Many-cores

This chapter devises a scheduling strategy for allocation of Parallel Software Partitions (pSWPs) to an architecture implementing Guaranteed Resource Partitions (GRPs). It builds upon the fact that account communication among pSWPs is known at system integration time, and tries to reduce its impact on Worst-Case Execution Time (WCET) estimates of applications encapsulated in pSWPs. We show that our algorithm increases the number of applications that can be scheduled on the many-core platform thus facilitating system integration (Objective **O3**).

7.1 Introduction

Many-core processors stand out as a potential means for Critical Real-Time Embedded Systems (CRTES) industry to satisfy growing performance demands. However, deriving time-composable WCET estimates in many-cores is challenging and it can lead to pessimism in WCET, slowing adoption of many-cores in CRTES.

One way to reduce the pessimism of WCET analysis of CRTES applications running on many-cores is performing compositional timing analysis [47], i.e. analyzing the impact of certain components independently and combining them to obtain WCET estimates. This thesis provides two many-core architectures based on GRPs together with time-compositional analysis (details in Chapter 3) that exploits the fact that the amount of data sent from one application to another is known at system integration time [23] and thus, its impact on WCET estimates can be accounted for at system integration time.

However, determining the most convenient scheduling of applications onto a many-core platform so that the impact of communication among applications on WCET estimates at integration

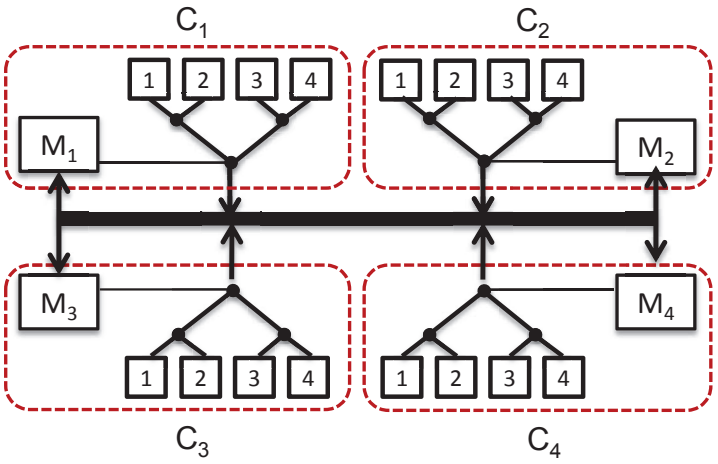


FIGURE 7.1: Time-predictable many-core architecture with GRPs resembling [4]

time is kept low is still a challenge. Therefore, it is of prominent importance devising algorithms tackling this challenge so that an efficient use of the hardware resources is obtained, thus allowing more applications to be integrated onto the same many-core platform, thus reducing procurement, maintenance and power costs as well as size and weight of the CRTES. To the best of our knowledge no specific scheduling algorithm has been proposed for this problem.

We propose *CAP*: *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on Many-cores implementing GRPs. *CAP* reduces the impact of communication on guaranteed performance of parallel CRTES applications (encapsulated within pSWPs while facilitating system integration. *CAP* is based on *worst-fit heuristics* used in combination with a non-preemptive time-triggered online scheduler (Objective **O2**).).

CAP constructs the schedule starting from the consumer applications, selecting the ones with highest *consumer weight* metric first. The consumer weight metric sums up computational requirements as well as the amount of communication among applications across the producer-consumer chains of dependencies that reach this application. *CAP* schedules applications considering the impact of their communications on already scheduled applications iteratively until all applications are scheduled.

We illustrate the concept of *CAP* by applying it to many-core processor architectures with GRPs (e.g. one in Figure 7.1), and evaluate *CAP* with a set of randomly generated workloads, which is the common practice in the area of scheduling, emulating future CRTES with more than 10 parallel applications. Overall, we show that the use of *CAP* allows us to schedule up to 29% more workloads on average compared to basic worst-fit heuristic allocation algorithms while facilitating system integration.

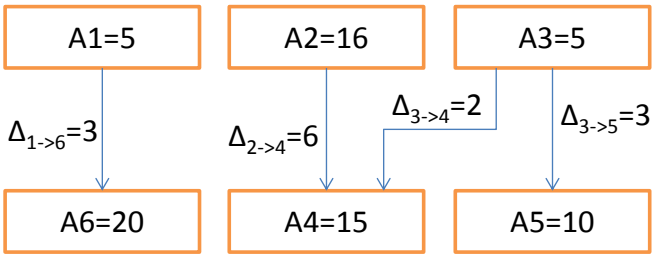


FIGURE 7.2: Example of the directed acyclic graph for a CRTES comprising 6 applications

7.2 Background

7.2.1 CRTES applications

CRTES consist of several applications exchanging data. We focus on parallel CRTES applications [137], encapsulated within pSWPs, that comprise several processes that communicate among them as well as with other applications, similar to the ones described in Section 2.2.1. Understanding the impact of communication on the timing behavior of the system is one of the main obstacles for the use of many-core processors in CRTES.

In order to facilitate system integration, AUTomotive Open System ARchitecture (AUTOSAR) [24] and ARINC653 [23] standards differentiate 2 types of communication: (i) communication that occurs among the processes of the same Software Partition (SWP) (*intra-SWP communication*) and (ii) communication that occurs among the processes of different SWP (*inter-SWP communication*). Each type of communication is implemented through a specific API, e.g. in the avionics domain intra-application communication is done through software structures called buffers and inter-SWP communication is done through software structures called queues. Those standards also require that the amount of inter-SWP communication is known at system integration time. This requirement allows us to perform a compositional timing analysis and account for the impact of inter–SWP communication during system integration.

We can exploit the asymmetry among these 2 types of communication in order to obtain tighter WCET estimates and analyze their impact separately. During timing analysis we consider only the impact of intra-application communication and the impact of sending inter–SWP communication. We defer the analysis of the impact of inter–SWP communication on the other applications executed concurrently in other clusters until the system integration phase. More details can be found in Chapter 3.

Figure 7.2 shows an example system comprising 6 applications encapsulated in pSWPs. Each application has a WCET estimate and few of them communicate using inter-SWP communication mechanisms. (Section 7.3.1 provides details on $\Delta_{i \to j}$, which represents the impact that each inter–SWP communication has on the timing behavior of the system).

7.3 Allocation Algorithm

This section presents the main contribution of this chapter: *CAP*, a *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on Many-cores, whose purpose is reducing the impact of communication on WCET estimates of applications while facilitating system integration. We focus on the allocation of parallel CRTES applications in many-core processor architectures like the ones presented in Section 3.5.

7.3.1 Problem Definition

We focus on those CRTES with the following properties:

- Applications are encapsulated within pSWPs
- Applications are periodic with period P_i and have implicit deadline ($D_i = P_i$).
- The amount of inter-SWP communication is known at system integration time, when the scheduling tables are created, and the maximum impact of communication on applications running concurrently in the destination GRP – $\Delta_{i \to j}$ – can be determined.
- Applications do not assume any specific communication pattern (e.g. point-to-point, broadcast, etc.).
- Data coming from other applications must be available before application starts.
- pSWPs are not preempted and they run until completion.
- Intra-SWP communication and inter-SWP communication are explicitly separated in line with AUTOSAR [24] and ARINC653 [23] standards.

Our system can be represented as a Directed Acyclic Graph (DAG) $\sigma = (\mathcal{A}, \mathcal{D})$. The nodes in $\mathcal{A} = \{A_1, ..., A_n\}$ represent the applications that compose the system. Each application A_i is characterized with a WCET estimate C_i obtained in isolation, running in a GRP. The utilization u_i of application A_i is defined as $\frac{C_i}{P_i}$, where $0 \leq u_i \leq 1$. The edges in $\mathcal{D}$ represent inter-SWP communication, in which $\Delta_{i \to j} \in \mathcal{D}$ represents the precedence constrains among the nodes in $\mathcal{A}$, such that A_j cannot start executing until A_i finishes. It is important to remark that the use of DAGs allows representing a wide range of communication patterns, including point-to-point, broadcasting, etc.

The weight of an edge $\Delta_{i \to j} \in \mathcal{D}$ represents the maximum impact of inter-SWP communication between A_i and A_j on the WCET of any arbitrary affected application A_m running concurrently with A_i and allocated to the same GRP as A_j. An inter-SWP communication request (from A_i to A_j) delays intra-SWP requests increasing the WCET estimate of the affected application A_m by a value I_k. I_k is upper-bounded by inter-SWP communication request Upper-Bound Delay (UBD) (see Section 3.5).

Therefore, the maximum impact that inter-SWP communication among applications A_i and A_j can have on application A_m running in the destination GRP, marked as $\Delta_{i\to j}$, is the addition of the impact of each I_k, as shown in Equation 7.1:

$$\Delta^m{}_{i\to j} = \sum_{k=1}^{N_{req}} I_k \tag{7.1}$$

where N_{req} is the number of requests of inter-SWP communication among applications i and j. Then, during system integration, when the allocation of applications is known, the WCET estimate of the affected application A_m must be increased by the corresponding $\Delta_{i\to j}$ values.

For the sake of clarity and to ease the explanation, we assume that all applications have the same deadline. If this was not the case, the DAG of our system would comprise all instances of the applications during one hyper-period of the system (least common multiple of application periods) and we would have to slice it into time slots and apply *CAP* to each time slot independently, following a similar approach to the ones shown in [32, 114].

CAP assigns the n parallel applications in $\mathcal{A}$ to a set of m identical GRPs $sc = \{c_1, \cdots, c_m\}$, respecting precedence constraints in $\mathcal{D}$. GRPs isolate intra-SWP communication and bound inter-SWP communication (see Section 3.5). *CAP* generates a static partition $\Phi = (\varphi_1, \cdots, \varphi_m)$ in which a subset of $\mathcal{A}$ is assigned to a GRP c_i. Each application can be assigned to only 1 GRP.

In order to guarantee that precedence constraints are respected, *CAP* has to allocate application A_j after every application $A_i \in \mathcal{A}$ finishes if there is inter-SWP communication from A_i to A_j ($\exists \Delta_{i\to j} \in \mathcal{D}$). Application A_i is called producer application if there is an application A_j that uses the results produced by A_i. A_j is then called consumer application.

7.3.2 Mapping Applications to GRPs

Figure 7.3 shows a pseudo-code implementation of *CAP*. The algorithm takes a DAG $\sigma = (\mathcal{A}, \mathcal{D})$ and a set of m GRPs $sc = \{c_1, ..., c_m\}$ as the input and provides a valid allocation Φ as the output.

CAP allocates a set of application inside a given time slot (in this case equal to the deadline of applications). It constructs the schedule by assigning offsets from the end of the given time slot to application. It starts the allocation with consumer application first. This facilitates accounting for inter-SWP communication impact $\Delta_{i\to j}$. At the moment when producer application is allocated, its consumer is already allocated and *CAP* can detect whether $\Delta_{i\to j}$ impacts other applications. Note that once *CAP* produces the schedule, the whole schedule can be shifted to the the beginning of the time slot, if the system integrator prefers having slack at the end of the time slot, instead of at the beginning.

CAP starts by assigning *consumer weights* to all applications in $\mathcal{A}$ (line 3). Consumer weight of application A_l (cw_l) is initially computed as the highest sum of all C_k and $\Delta_{i\to j}$ across all chains of dependencies that have application A_l as the consumer. For example in Figure 7.2, initial

CAP allocation algorithm
Input $\sigma = (\mathcal{A}, \mathcal{D})$: A DAG of the system ,
$sc = (c_1, \cdots, c_m)$: a set of GRPs

Output $\Phi = (\varphi_1, \cdots, \varphi_m)$: A valid allocation of σ into sc

```
1  Φ = ∅
2  deltas = ∅
3  set_consumer_weights(σ)
4  forall (A_i ∈ A | A_i ∉ Φ; ∄A_j ∈ A; Δ_{i→j} ∈ D; A_j ∉ Φ)
5
6    select A_i with highest cw_i from A
7    update_predecessors_consumer_weight(A_i)
8    A_i.start_time = A_i.end_time = 0
9    A_i.end_time = lat_start_time (A_i.successors())
10
11   if (∃ idle_slot ∈ φ_j | idle_slot.size > C_i + deltas(idle_slot) and idle_slot.
     end_time ≥ A_i.end_time)
12     A_i.end_time = idle_slot.end_time
13     A_i.start_time = A_i.end_time + C_i + deltas(idle_slot)
14     Φ += allocate(c_j, A_i)
15   else
16     c_j = worst_fit(A_i)
17     apply_deltas(c_j, A_i)
18     Φ += allocate(c_j, A_i)
19   endif
20   forall (A_k ∈ Φ and A_k ∉ φ_j)
21     deltas += Δ_{i→k}
22 endfor
23 return Φ;
```

FIGURE 7.3: Pseudo-code implementation of the allocation algorithm.

consumer weight of A_4 is computed as $cw_4 = max(16 + 6 + 15, 5 + 2 + 15) = 37$, while the rest of them are $cw_1 = 5, cw_2 = 16, cw_3 = 5, cw_5 = 18, cw_6 = 28$. We use the consumer weight metric to identify consumers that belong to the longest chains of dependencies and create highest pressure on the allocation algorithm possibly containing $\Delta_{i \to j}$ with high impact.

CAP does all allocations in a single loop. Before each iteration of the main loop (line 4), *CAP* updates the list of applications that are ready for allocation. It puts the following types of applications into the list: (i) independent applications, (ii) consumer only applications and (iii) producers with all of their consumers already allocated. Among them it selects the one with highest consumer weight metric (line 6) for allocation. In the example in Figure 7.2, applications A_4, A_5 and A_6 are ready for allocation and *CAP* selects A_4 since it has the highest consumer weight.

Once the application is allocated, *CAP* updates consumer weight metric for all of its predecessors (line 7). In this step, we add the $\Delta_{j \to i}$ to consumers weight of A_i predecessor for each communication in which A_i is consumer. It helps us keeping track of high $\Delta_{j \to i}$ values when choosing the next application to allocate. In the example, selecting A_4 leads to updating consumer weights of A_2 and A_3: $cw_2 = cw_2 + \Delta_{2 \to 4} = 22; cw_3 = cw_3 + \Delta_{3 \to 4} = 7$.

In order to allocate the application, we compute its starting and ending time in the schedule (expressed as the offset from the end of the time slot). Since we start from the end of the time slot, we try to allocate applications as late as possible in the schedule. We determine the latest ending time of the application A_i by examining the earliest starting time of its successors in σ (line 9) to guarantee that all producer applications will finish before any consumer starts, e.g. A_3 has to finish before A_4 and A_5 start.

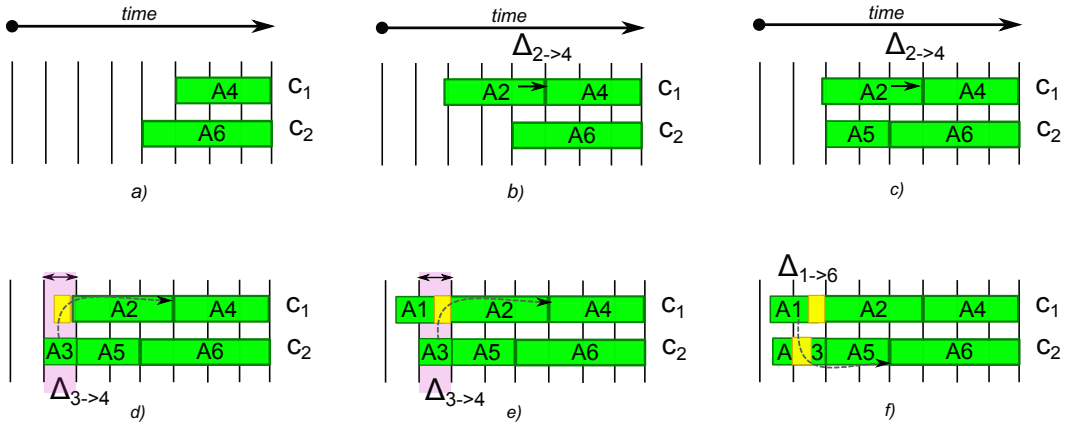


FIGURE 7.4: Allocation of the example applications from Figure 7.2

CAP uses starting and ending times of an application A_i in order to delimit the interval of the time slot when the applications affected by $\Delta_{j \to i}$ have to be given additional resources to compensate for the impact of $\Delta_{j \to i}$ (see Section 7.3.3 and Figure 7.4(e)). Also those applications that could be potentially affected by inter-SWP communication will have their starting time shifted and therefore the difference between its start_time and end_time will be greater than their respective WCET estimates.

After computing the latest ending time of an application, *CAP* assigns the application to a GRP. In order to do this allocation in a single loop and maintain low complexity, *CAP* looks for idle bubbles in the schedule and tries to fit the current application there (lines 11-14) respecting the dependencies and WCET constraints. Idle bubbles in the schedule exist if the application cannot start as late as it could in the assigned GRP but has to start earlier due to dependencies. This step is designed for small chains of dependencies and independent applications that are allocated late in the algorithm and tries to maximize utilization of the processor.

If *CAP* cannot find a suitable idle bubble in the schedule, it uses worst-fit heuristics to choose the GRP where A_i is allocated (line 16). We check if there are existing $\Delta_{k \to j}$ in the interval between start and ending time of A_i in the GRP c_j. If there is any inter-SWP communication targeting the GRP c_j during that interval, we update the starting time of A_i to accommodate a $\Delta_{k \to j}$ that could affect the application (line 17). For example, when allocating A_1 to c_1, we have to shift A_1 start time to accommodate for $\Delta_{3 \to 4}$ (see Section 7.3.3 and Figure 7.4(e)). Then we allocate the application to the GRP c_j (line 18).

The last step of the loop consists of adding all $\Delta_{j \to k}$ to the list of inter-SWP communications that cross GRP boundaries and creating *zones of communication impact* in the schedule as well as updating existing ones (lines 20-21).

7.3.3 Example

Figure 7.4 illustrates how *CAP* allocates the example in Figure 7.2 into a 2-GRP many-core. In the first 2 steps, *CAP* selects applications with highest consumer weights $A_4 = 37$ and $A_6 = 28$, allocates them to GRPs c_1 and c_2 respectively (Figure 7.4(a)). It updates consumer weights of

their predecessors by adding $\Delta_{2->4} = 6$ to consumer weight of A_2 making it $cw_2 = 22$, as well as consumer weights of A_1 and A_3.

After this, A_2 has the highest consumer weight, and *CAP* allocates it to c_1 (Figure 7.4(b)). There is communication between applications A_2 and A_4, but they are allocated to the same GRP c_1. Thus, this communication has no impact on any other application (its impact is included in the WCET estimate of A_2 and it does not use resources of any other application). Since there are no predecessors of A_2 in the DAG, *CAP* does not update any consumer weights and it selects the application with highest consumer weight, i.e. A_5 and allocates it to the GRP c_2 (Figure 7.4(c)).

The next application for allocation is A_3. Based on worst-fit heuristics, *CAP* allocates it to GRP c_2. Since $\Delta_{3\to4}$ exists and applications A_3 and A_4 are allocated to different GRPs, *CAP* creates an interval of communication impact in GRP c_1. This interval (marked purple in Figure 7.4(d)) makes *CAP* shift the start of the applications affected by the amount of interference that can be created inside it ($\Delta_{3\to4}$ - marked yellow). A_2 was already allocated inside this interval and its start has to be shifted by $\Delta_{3\to4} = 2$ time units to compensate for the impact inter-SWP communication between A_3 and A_4.

Finally, *CAP* allocates A_1 to c_1 (see Figure 7.4(e)). A_1 communicates with A_6 allocated to c_2. Again, *CAP*, same as before, treats $\Delta_{1\to6}$ creating another zone of communication impact in GRP c_2. This zone causes the shift of the start time of A_3 and *CAP* must ensure that the zone of communication impact created by A_3 is also updated accordingly. Figure 7.4(f) represents the final allocation of the example.

In this example $\Delta_{3\to4}$ affects multiple applications, A_1 and A_2. In order to avoid "double accounting" of communication impact (once per each application) as well as to prevent starting A_2 application while A_1 has not finished, *CAP* requires simple support from the operating system (online scheduler). *CAP* has to detect cases where 2 (or more) applications are affected by 1 zone of communication impact, e.g. A_1 and A_2 in f7.4(e). Starting times of applications have to be shifted by $\Delta_{3\to4}$ and the operating system must start A_2 only if: (i) the start time of A_2 has passed and (ii) application A_1 has finished.

7.4 Evaluation methodology

CAP targets future CRTES comprising several parallel applications. To evaluate its effectiveness we use randomly generated application-sets and allocate them to the many-core processors supporting GRPs presented in Section 3.5.

In order to better resemble the communication requirements of real systems, the randomly generated application-sets are based on the avionics system presented in Chapter 2 and Chapter 3 comprising 3D obstacle and stereo camera image generators, that create input for 2 collision avoidance parallel applications. An additional application checks the results of collision avoidance applications and compares them.

	WCET(cycles)	$\Delta_{i->j}$
Compositional	500,000-3,000,000	100,000-3,000,000
Composable (1.5-2.15)	750,000-6,450,000	N/A
Composable (1.3-1.6)	650,000-4,800,000	N/A

TABLE 7.1: WCET intervals in cycles, assuming 1GHz processor

In order to emulate a more complex system with higher workload, we consider two different scenarios: (i) randomly-generated application-sets comprising between 11 and 16 parallel applications allocated onto a 16-core, 4-GRP many-core; and (ii) randomly-generated application-sets comprising between 43 and 64 parallel applications allocated to a 256-core, 16-GRP many-core processor. In both scenarios, the targeted many-core architecture is similar to the one presented in Figure 7.1.

Our random application-set generator creates DAG representing application-sets with a given utilization level assuming that applications fully utilize GRPs resources assigned to them. This means that a parallel application utilizes all the cores available in the GRP where it is assigned and that GRPs execute only one application at a time. WCET estimates of applications are random values from an interval. Impact of communication among applications is also a random value with the following constraint: randomly created communication edges cannot form loops in the graph.

CAP allows the use of compositional timing analysis and much tighter WCET estimates w.r.t. traditional time-composable WCET. In order to fairly evaluate *CAP*, we create a time-composable copy of randomly created DAGs.

Table 7.1 shows the intervals used by our random generator to choose WCET estimates of the applications and $\Delta_{i->j}$. As shown in Table 7.1, we assume a significant amount of inter-SWP communication, in order to create more pressure on *CAP*. In the case of the time-composable approach, we consider 2 scenarios: row 2 of Table 7.1 represents the case from Chapter 3, where time-composable WCET estimates of two industrial parallel avionics applications presented in [30] are 1.5x and 2.15x higher w.r.t. to compositional ones, while row 3 represents an optimistic case where it is possible to conduct an improved composable timing analysis so that time-composable WCET estimates are only between 1.3x and 1.6x higher than compositional ones. In case of the time-composable approach, the weight of the communication impact edges is set to 0, since the WCET estimates already account for all possible program interactions, but the edges are maintained to keep the precedence constraints.

Then we allocate the initial DAG with *CAP* and its time-composable counterpart with a *b*asic *a*llocation algorithm based on *w*orst-*f*it heuristics (*BAWF*).

BAWF works similarly to the algorithm presented in Chapter 6. It allocates applications to GRPs based on their time-composable WCET estimates (rows 2 and 3 of Table 7.1). For each application it selects the GRP with lowest scheduled utilization respecting preceding constraints.

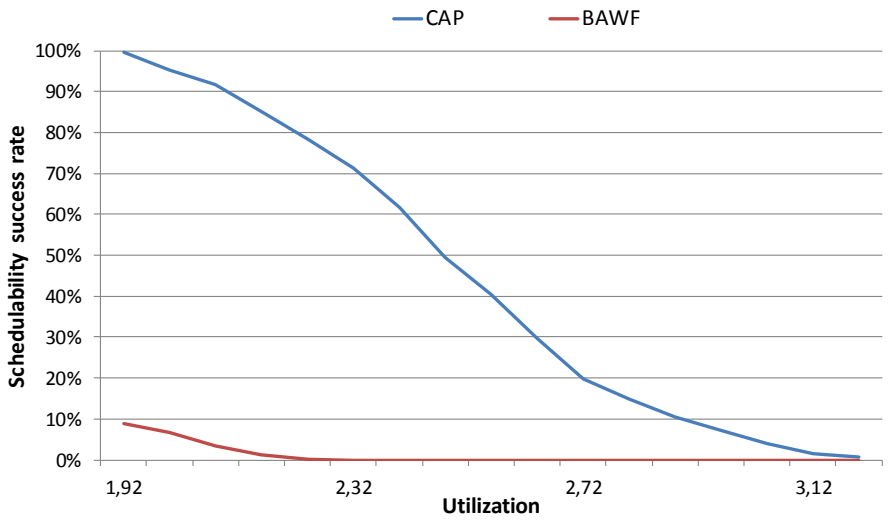


FIGURE 7.5: Schedulability success rate - CAP vs. BAWF (Composable 1.5-2.15) on a 4-GRP many-core

7.5 Results

We evaluate *CAP* by applying it to randomly generated application-sets allocating them to a many-core processor with 4 GRPs as presented in Section 3.5. We also apply it to a larger version of this processor, comprising 16-GRPs with 16 cores each. A currently used processor with these number of cores and GRPs is Kalray MPPA [4].

7.5.1 4-GRP many-core

In the case of 4-GRP many-core, *CAP* is used with a series of randomly generated application-sets. For each utilization value in [1.92, 4), with an utilization increment of 0.08, we create 1,000 application-sets, 30,000 in total. Each application-set comprises between 11 and 16 applications and between 7 and 12 inter-application communication dependencies. Utilization of 1.92 means that 48% of the GRPs are used by an application-set, and utilization of 4 means that all 4 GRPs are fully utilized during the time-slot.

We compare *CAP* against a time-composable approach using worst-fit heuristics - *BAWF*.

Figure 7.5 compares the schedulability success rates[1] of application-sets using *CAP* (labeled as *CAP*) and the basic worst fit algorithm (labeled as *BAWF*) considering the scenario in row 2 of Table 7.1 when allocating tasks to a 4-GRP many-core. Utilization increases from 1.92 up to 3.2. We observe that *CAP* is superior to *BAWF* in this scenario, being able to allocate most of the application sets (95.7%) at utilization 2, while *BAWF* is able to allocate only 6.7% of the application sets. *CAP* schedulability success rate decreases as we increase the utilization, but it is still able to allocate around 50% of the application sets at utilization 2.56.

[1]Schedulability success rate represents the percentage of the application sets that an algorithm is able to allocate at a given utilization level.

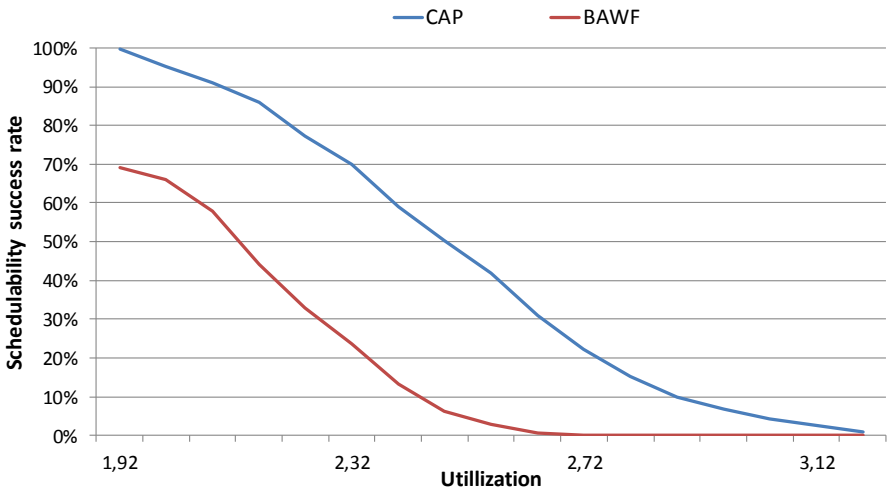


FIGURE 7.6: Schedulability success rate - CAP vs. BAWF (Composable 1.3-1.6) on a 4-GRP many-core

Figure 7.6 compares *CAP* and *BAWF* in an optimistic scenario (row 3 of Table 7.1). *CAP* still outperforms *BAWF* by allocating 28.9% additional application-sets on average. In the case of an utilization around 2.64, *CAP* is able to allocate around 40% of the application-sets, whereas *BAWF* can hardly allocate few of them.

7.5.2 16-GRP many-core

In the case of 16-GRP many-core, *CAP* is used with a series of randomly generated application-sets. For each utilization value in [4.8, 12.8), with an utilization increment of 0.32, we create 1,000 application-sets. Each application-set comprises between 43 and 64 applications and between 40 and 60 inter-application communication dependencies. Utilization of 4.8 means that 30% of the CPU capacity is used by an application-set, and utilization of 12.8 represents 70% of the CPU capacity.

Figure 7.7 shows the schedulability success rates of application-sets using *CAP* and *BAWF* considering the scenario in row 2 of Table 7.1 when allocating applications to a 16-GRP many-core. Utilization increases from 4.8 up to 12.8. We observe that *CAP* outperforms *BAWF* in this scenario, being able to allocate most of the application sets (97.8%) at utilization 4.8, while *BAWF* is able to allocate only 46.7% of the application sets. Even though, both algorithms under-utilize many-core resources in this case, *CAP* can still allocate around 50% of the application sets at utilization of 7.04.

Figure 7.8 shows the more optimistic scenario (row 3 of Table 7.1) for *BAWF* algorithm, when it works with tighter time-composable WCET estimates. *CAP* again has higher schedulability success rates w.r.t. *BAWF*, being able to allocate 26.8% more application sets on average.

In the case of 16-GRP many-cores, the difference between time-composable and time-compositional WCET estimates could be higher than the values we extracted from [30] due to the higher core count and higher interference. If that was the case, the advantage of *CAP* would become even more obvious.

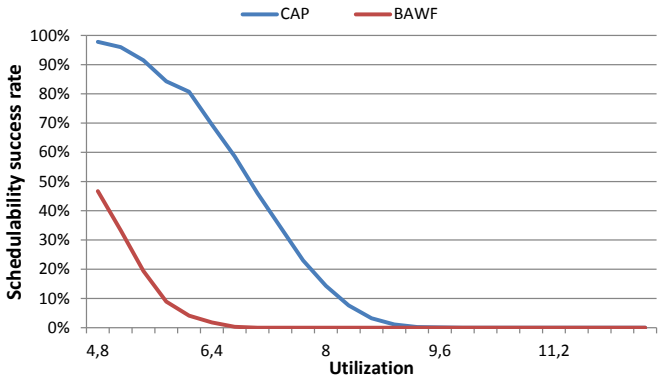


FIGURE 7.7: Schedulability success rate - CAP vs. BAWF (Composable 1.5-2.15) on a 16-GRP many-core

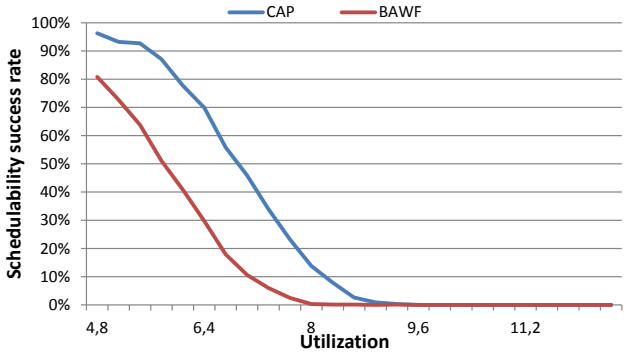


FIGURE 7.8: Schedulability success rate - CAP vs. BAWF (Composable 1.3-1.6) on a 16-GRP many-core

We can observe that utilization of a 16-GRP many-core is low (Figure 7.7 and Figure 7.8). This is an inherited limitation of partitioned time-triggered non-preemptive scheduling, required by the time-compositional analysis from Chapter 3. Extending the analysis and *CAP* to support preemption as well as other techniques for improving utilization [138] remains as future work and one of the obstacles for using high core-/GRP- number many-cores in CRTES.

7.5.3 Algorithm complexity

Even though *CAP* is an offline allocation algorithm, and its performance is not crucial for the performance of the system, it is a light-weight allocation algorithm. Allocating a 60+ application set from previous section to a 16-GRP many core takes less than 10 seconds in a typical laptop.

All allocations are done in a single loop (line 4 in the algorithm Figure 7.4). At the beginning of the loop, there is a search through a list of applications for the one with highest consumer weight metrics (line 6). These 2 operations define algorithm's complexity as $O(N^2)$, where N is the number of applications in the application set. Note that there is another loop (lines 20-21), but since $k < N$, it does not affect algorithm complexity.

7.6 Related Work

Scheduling of parallel CRTES applications to clustered many-core processors relates to scheduling of processes to multi-core processors. We can split research on multi-core scheduling [130], into 2 categories w.r.t. allocation of processes to cores: (i) partitioned approaches that allocate each process to a single core and later use single processor scheduling for each core and (ii) global approaches that allow processes to migrate from one core to another at run-time.

We opted for a partitioned approach to benefit from predictability of inter-application communication and to improve WCET estimates of applications and throughput of the system. Finding an optimal allocation using partitioned approaches is an NP-hard problem [121] and so sub-optimal solutions are derived using, for instance, bin-packing heuristics [122–124].

Most scheduling works proposed in the CRTES domain consider independent processes that do not communicate among them. Lakshaman et al. [138] presented a preemptive fixed-priority partitioned scheduling for multi-cores that relies on task-splitting to improve utilization bounds. However, this approach requires using time-composable WCET estimates and has additional costs of preemption and task migration, mitigating the benefits of the compositional analysis from [30].

Paolieri et al. [139] present an interference-aware allocation algorithm that uses multiple WCET estimates per application when constructing the schedule. The WCET estimate value is chosen from a structure called WCET-matrix that contains a set of WCET estimates obtained under different execution scenarios. *CAP* requires only 1 WCET estimate per application and takes into account only the interference of inter-application communication when creating the schedule.

Wieder and Brandenburg [134] propose a real-time partitioned scheduling of independent tasks in which accesses to the shared resources are protected with spin locks. They provide an ILP formulation for finding optimal partitioning w.r.t. schedulability analysis of the MSRP protocol [135]. Their approach cannot be applied to these application-sets, as they have run-after data dependencies and would require use of synchronization mechanisms across applications, with which the analysis presented in [30] is incompatible.

Along this line, in [140, 141] authors propose scheduling and mapping of mixed-criticality applications on multi-core platforms. They present the global time-triggered scheduling algorithm that uses barriers for synchronization. However, they allow concurrent execution of applications from only 1 criticality level, in order to derive timing guarantees. *CAP* does not impose such a restriction, since it is only required to know the amount of communication among applications, regardless of the criticality level.

In the automotive domain, there are few recent multi-core scheduling proposals [32, 114, 115]. Monot et al. [114] present a scheduling algorithm for multi-source AUTOSAR applications that allows communication only for functions executing on the same core, so each core can be scheduled independently. Faragardi et al. [115] present a scheduler that reduces communication delays for AUTOSAR applications on multi-cores and Panic et al. [32] present an allocation algorithm that parallelizes legacy single-core AUTOSAR applications. All those solutions consider

time-composable WCET estimates, and do not consider the impact of communication on the affected applications, limiting the performance potential of many-core processors. Sinnen et al. [136] propose a task scheduling mechanism similar to [32], that targets general purpose processor architectures where the task scheduler is aware of the cost of inter-processor communication, addressing the system throughput instead of the WCET of applications.

In the avionics domain, Kim et al. [142] propose a scheduler for multiple Integrated Modular Avionics (IMA) applications on a multi-core. This proposal considers inter-application communication, but imposes a severe constraint: while inter-application communication executes nothing else is executed in other cores.

7.7 Conclusions

In this chapter, we present *CAP*, a *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on Many-cores, that significantly reduces the impact of communication on guaranteed performance of parallel CRTES applications while facilitating system integration (Objective **O2**).

CAP exploits the fact that inter-application communication is known at system integration time enabling the use of tight WCET estimates. It constructs the schedule starting from consumer applications first giving higher priority to those belonging to the chains of dependencies with longest utilization. For each inter-application communication that crosses the cluster boundaries, *CAP* creates a zone of communication impact inside which applications allocated to the destination cluster are given additional computational resources.

We evaluate *CAP* with sets of randomly-generated application-sets, based on the case studies presented in [30] that represent future complex CRTES targeting many-core processors implementing GRPs (Chapter 3). We compare *CAP* with a baseline state-of-the-art allocation algorithm that uses time-composable WCET estimates. *CAP* is able to allocate up to 29% more workloads on average to many-core processors with 4-clusters compared to the baseline algorithm.

Therefore, *CAP* stands as an important step towards the efficient use of many-core processors in CRTES so that their performance potential can be fully exploited.

Part IV

The Thesis and Beyond – Conclusions and Future Work

Chapter 8

Enabling TDMA Arbitration in the Context of MBPTA

Current timing analysis techniques can be broadly classified into two families: Deterministic Timing Analysis (DTA) and Probabilistic Timing Analysis (PTA). Each family defines a set of properties to be provided (enforced) by the hardware and software platform so that valid Worst-Case Execution Time (Worst-Case Execution Time (WCET)) estimates can be derived for programs running on that platform. However, the fact that each family relies on each own set of hardware designs limits their applicability and reduces the chances of those designs being adopted by hardware vendors.

In this chapter, we show that deterministic architectures that are the focus of this thesis, can be analyzed in the context of PTA. We focus on Time Division Multiple Access (TDMA) arbitration policy, one of the most common arbitration policies in Critical Real-Time Embedded Systems (CRTES). We show that even though TDMA is suits DTA well, it also can be made PTA-compliant with little effort. To that end, we analyze TDMA in the context of Measurement-Based Probabilistic Timing Analysis (MBPTA) and show that padding execution time observations conveniently leads to trustworthy and tight WCET estimates with MBPTA without introducing any hardware change. In fact, TDMA outperforms round-robin and time-randomized policies in terms of WCET in the context of MBPTA.

8.1 Introduction

Developing CRTES requires validating its timing behavior. This can be done by deriving WCET estimates to the execution time of each task, which are passed as input to the scheduler that combines them with other task information such as deadline, period and priority to validate that the budgets provided to each task are sufficient to satisfy the tasks' execution time needs. DTA techniques [143], both static and measurement-based (SDTA and MBDTA), advocate for time-deterministic architectures. The goal is that the access time to each resource can be

upper-bounded so that (1) with SDTA, bounds can be *incorporated* in the analysis and (2) with MBDTA, bounds can be *enforced* in the measurements taken during the analysis phase. PTA [144–148] supports architectures in which some resources are time-deterministic whereas others are time-randomized [149]. The goal is that resources' impact on execution time can be bounded either with a fixed value (deterministic upper-bounding) or a distribution function (probabilistic upper-bounding) [149].

Mixed-criticality applications running in multi- and many-cores challenge both timing analysis families, DTA and PTA, because the time it takes a request from a given task to be granted access to a resource depends on the load other co-running tasks put in that resource. Under DTA, specially SDTA, this dependence is in general controlled by advocating for hardware support that isolates tasks against each other, e.g. using TDMA arbitration [54], or allows upper-bounding the maximum impact of contention, e.g. round robin arbitration. Such isolation is a key enabler for mixed-criticality systems by preventing interferences across criticality levels. Under MBPTA it is required that the impact of contention captured in the measurements taken during the analysis phase of the system upper-bounds, deterministically or probabilistically [149], the impact of contention that can occur during the deployment of the system. While round-robin arbitrated shared resources used in the context of DTA have also been shown analyzable with MBPTA [150], this is not the case for TDMA arbitrated shared resources.

This chapter analyzes in detail TDMA in the context of MBPTA (Objective **O3**) and provides means to allow TDMA resources to be used together with MBPTA. Furthermore, we show that TDMA allows obtaining tighter WCET estimates than round-robin by padding execution time once instead of padding the latency of each request. To reach these objectives:

- We analyze the timing characteristics of TDMA in the context of MBPTA from a theoretical perspective. We show that TDMA cannot be directly analyzed with MBPTA. The difficulty lies in the variable (i.e. jittery) nature of the delay that a request incurs to get access to the arbitrated resource and that a probability cannot be assigned to each specific delay value, thus failing to attain the properties required by PTA [149].
- We show that the effect of TDMA on execution time is limited to the duration of a single TDMA window when there is a single TDMA-arbitrated resource for asynchronous requests, as already proven for synchronous ones in [54]. Also, we show that the effect of TDMA for several chained arbitrations is limited to the least-common-multiple of the TDMA windows.
- We apply a simple modification to the application of MBPTA as a means to enable the analysis of TDMA. In particular, we *augment* the execution time observations collected when running the task of interest in the target system, which are used as input to MBPTA.

Our analysis not only advances the limits on the arbitration policies that can be analyzed with MBPTA without requiring MBPTA-customized designs [150], but also helps promoting *one-design-fits-all* for arbitration policies (Objective **O1**). The latter makes that different timing analysis techniques are enabled on the same hardware. This increases the impact that the research on time-analyzable hardware may have on chip vendors to adopt such hardware in actual processor designs, hence, reaching the goal of having time-analyzable multicores. Our solution based on

padding produces 9% lower WCET estimates on average than round-robin and MBPTA-specific arbitration policies.

8.2 Contention analysis for DTA and MBPTA

The access latency to a hardware shared resource includes the arbitration delay and the service latency. The former is the time a request spends to get access to the resource. The latter is the time that the request takes to be processed once it is granted access. Both of them may be impacted by contention, specially the arbitration delay. Several proposals have shown how to handle contention in the access to hardware shared resources so that trustworthy WCET estimates can be provided. For on-chip resources, the goal is providing time composability in the access latency for WCET estimation. This means that access latency can be upper-bounded such that the load that other tasks put on that resource does not exceed the access latency used for WCET estimation purposes for the task under analysis, thus avoiding interferences across tasks with mixed criticalities.

8.2.1 SDTA and MBDTA

SDTA [143, 151] abstracts a model of the hardware which is fed by a representation of the application code to derive a single WCET estimate. On the contrary, MBDTA makes (extensive) testing on the target system with stressful, high-coverage input data. From all tests it is recorded the longest observed execution time and an engineering margin is added to make safety allowances for the unknown. This margin is extremely difficult to determine in the general case. Under SDTA trustworthy WCET estimates are attained in the presence of contention by different means:

- At analysis time requests are assumed to experience always the worst-case latency in the access to the shared resource [152]. For instance, with round-robin, SDTA assumes that whenever the request becomes ready, it has the lowest arbitration priority so it is delayed by all other cores before getting access. As analysis-time latencies upper-bound deployment ones, the execution time derived at analysis time for the program upper-bounds the impact of the shared resource. Note that with MBDTA it is not assumed that requests suffer an upper-bound contention latency but, instead, this is enforced by a specific hardware mechanism [86] making each request be delayed as if it was experiencing the highest contention possible[1].
- Alternatively at analysis time each request is assumed to suffer a fixed impact on its duration. This approach is used by SDTA when applied to TDMA-arbitrated resources, by determining the alignment of each request w.r.t. the TDMA window and hence, the delay it suffers until its next available slot.
- It is also possible to carry out a combined timing analysis of all the tasks simultaneously running in the multicore [153]. While this may reduce the impact of contention on WCET

[1] If no hardware support is in place measurements need to capture high contention scenarios, but trustworthiness of the WCET estimates is hard to support with evidence.

TABLE 8.1: Random arbitration bus example.

Number of rounds	contenders 4	3	2
1	0,2500	0,3333	0,5000
2	0,2344	0,2963	0,3750
3	0,2031	0,2222	0,1250
4	0,1563	0,1111	0,0000
5	0,0938	0,0370	0,0000
6	0,0469	0,0000	0,0000
7	0,0156	0,0000	0,0000
8	0,0000	0,0000	0,0000

(a) Probability of getting the bus in a given round X

Number of rounds	contenders 4	3	2
1	0,2500	0,3333	0,5000
2	0,4844	0,6296	0,8750
3	0,6875	0,8519	1,0000
4	0,8438	0,9630	1,0000
5	0,9375	1,0000	1,0000
6	0,9844	1,0000	1,0000
7	1,0000	1,0000	1,0000
8	1,0000	1,0000	1,0000

(b) Accumulated prob. of getting the bus in the first X rounds

estimates, since only the actual contention generated by the co-running tasks is considered, it comes at the cost of losing time composability, since any change in the tasks in the workload requires reanalyzing all the tasks in it.

8.2.2 MBPTA

MBPTA derives a distribution, called Probabilistic Worst Case Execution Time (pWCET), that associates a probability of exceedance to each WCET value. The exceedance probability, which upper-bounds the probability that a single run of the task exceeds its WCET budget, can be set arbitrarily low in accordance with the requirements of the corresponding safety standard. For instance DO-178B/C [28] for avionics sets the maximum allowed failure rate of a system component to 10^{-9} per hour of operation for its highest integrity level. This translates into 10^{-15} exceedance probability for tasks triggered every 10ms [154].

MBPTA, builds on end-to-end measurements taken on the platform to derive a WCET distribution, rather than a single WCET estimate per task, as it is the case for SDTA. MBPTA requires understanding and controlling the nature of the different *contributors* to the execution time of a program [155]. These contributors, also known as sources of execution time variability (*setv*), include (i) the initial conditions of hardware and software (e.g., cache state), (ii) those functional units with input-dependent latency (e.g., integer divider), (iii) the particular addresses where memory objects are placed, (iv) the number of contenders in the access to shared resources, and (v) the execution paths of the program. MBPTA requires that the jitter, i.e. execution time variability, of all *setv* captured in the end-to-end execution times collected at analysis time upper-bound the jitter of each *setv* when the system is deployed (*deployment phase*). In [149] it is explained how upper-bounding these *setv* enables collecting execution time observations that can be regarded as independent and identically distributed, as required by MBPTA [146].

Jitter can be upper-bounded *deterministically* [149] by forcing *setv* to experience a single latency at analysis time lat_{det}^{an} that upper-bounds any latency that the *setv* may take at deployment, $lat_{det}^{dep,i}$. That is, $\forall i : lat_{det}^{an} \geq lat_{det}^{dep,i}$. For instance, enforcing functional units with input-dependent latencies to operate at their highest latency during the analysis phase leads to deterministic

upper-bounding as their latency at analysis time is constant. At deployment, real latencies will be equal or lower than those at analysis time.

Jitter can also be upper-bounded *probabilistically* [149] by forcing the latencies of a *setv* to have a probabilistic distribution at analysis time such that for any exceedance probability (e.g., 10^{-3}), the latency at analysis time is equal or higher than that of the distribution at deployment. For instance, let us assume random-permutations arbitrated bus [150] shared by N_c cores. Further assume that at deployment the bus is arbitrated only across all cores with pending requests, which are a subset of all N_c cores. In this scenario, the analysis-time delay distribution experienced due to contention upper-bounds that at deployment if at analysis time arbitration always occurs across N_c cores. This upper-bounding is probabilistic since such delay is not a fixed value but a distribution. Table 8.1(a) shows the probability of getting the bus in a given round under different contender (core) counts, while Table 8.1(b) shows the accumulated probability, that is the probability of getting the bus in any of the first X rounds[2]. We observe that when all $N_c = 4$ cores are assumed active, as it is the case at analysis time, the accumulated probability of getting the bus is smaller than when the number of cores is 3 or 2. Hence, given that at deployment time the number of active cores is at most 4, the analysis time contention distribution upper-bounds that obtained at deployment time rendering this arbitration policy as MBPTA analyzable.

8.3 TDMA impact on execution time

TDMA ensures that the load a task puts on a shared resources does not affect the WCET of its co-runners [152], thus isolating tasks with different criticality levels. In this section we make a detailed analysis of TDMA impact on the timing behavior of the application. Without loss of generality we focus on a bus as the resource arbitrated with TDMA.

We assume canonical TDMA so that it splits time into windows of size w cycles, each of which is further divided into slots of size s. Each bus contender (cores in our case) is assigned one such slot in a cyclic fashion. During a given slot only its owner can send requests. When a contender has no pending requests, the bus remains idle for that slot even if there are pending requests from other contenders (non-work-conserving approach). We call *tdma-relative cycle* or simply *relative cycle* (cyc_i^{rel}) to the cycle in which a request, r_i, becomes ready within the TDMA window. It can be computed as shown in Equation 8.1, where cyc_i^{abs} stands for the absolute execution cycle.

$$cyc_i^{rel} = cyc_i^{abs} \mod w \tag{8.1}$$

8.3.1 Request Types

We consider a timing-anomaly free architecture [156–159]. A number of definitions have been devised for timing anomalies. In our case, a processor architecture free of timing anomalies refers

[2]Note that random permutations works similarly to TDMA but sorting slots randomly within each window. Thus, the maximum arbitration delay is always below two TDMA windows.

to an architecture where an increase in the access latency of a request to any resource (e.g., due to contention) can only lead to an equal or higher execution time.

We consider both synchronous and asynchronous requests. Synchronous requests are blocking. This means that they stall the corresponding pipeline stage until served. In our reference architecture this is the case of load operations that miss in first level (L1) caches and access the second level cache (L2).

Asynchronous requests, instead, are kept in a buffer until served not stalling any pipeline stage unless the buffer is full. This is the case, for instance, of those processors that do not stall the pipeline on a store (write) operation. Since no instruction in the core has to wait for the results of such write operation, the store operation is put in a store-buffer, which sends the request to the data cache afterwards. The store operation is considered as committed (serviced) when it is sent to the store-buffer. However, the write request may take a variable number of cycles to access the bus. This creates asynchronous accesses to the bus.

Split transactions are used when the target resource for the request, L2 in our case, takes long to answer (e.g. ARM AMBA bus [88] implements them). Instead of holding the bus for tens of cycles, the L2 answers the request with a 'split transaction' command allowing the other requester use the bus while L2 processes the request in background.

8.3.2 TDMA impact on execution time for synchronous request

The slot alignment delay (sad) for each request defines the time the request has to wait for its slot in a TDMA window so it can be granted access. In the worst case a request becomes ready one cycle after its slot expires making it wait sad_{tdma} cycles that is defined in Equation 8.2.

$$sad_{tdma} = (Nc - 1) \times s \tag{8.2}$$

Note that, without loss of generality and for the sake of simplifying formulation, we have assumed that the access time of a request is one cycle. In the general case, assuming a request latency lat_r, the worst scenario occurs when it becomes ready during its slot $lat_r - 1$ cycles before it elapses, making the request wait $sad_{tdma-gen}$ cycles as defined in 8.3.

$$sad_{tdma-gen} = (Nc - 1) \times s + lat_r - 1 \tag{8.3}$$

As shown in Equation 8.2, the particular sad of a request may make it be served right away (so 0 delay) or delayed by up to $(Nc - 1) \times s$ cycles, or in other words, $w - 1$ cycles. Therefore, given a program with a single synchronous request r_i, the execution time of the program can vary up to $w - 1$ cycles depending on how r_i aligns with the TDMA window as already shown in [54]. Further, if multiple synchronous requests exist in the program, the execution time variation that the TDMA resource can introduce is still up to $w - 1$ cycles as proven in [54]. The intuition behind this effect lies on the fact that a particular sad achieves the fastest execution time across the w different sad (w different alignments w.r.t. the TDMA window). Under any other sad the

program only needs to be stalled by up to $w-1$ cycles to align with the TDMA window as the fastest *sad*, and execute identically from that point onwards. We refer the interested reader to the work by Kelter et al. [54] for a formal proof.

8.3.3 *sad* for Multiple Asynchronous Requests

In the case of synchronous requests the time between requests accessing the bus is fixed, regardless of the particular *sad* each of them suffers. However, this is not the case for asynchronous requests (e.g., stores). Let δ_i^{inj} be the injection delay between a preceding instruction generating request r_{i-1} and the instruction generating request r_i. The injection delay can be measured as the time elapsed since r_{i-1} is fetched into the processor until r_i is fetched.

Hence, a program P with $n+1$ requests can be represented as $\Delta_P^{inj} = \{-, \delta_1^{inj}, ..., \delta_n^{inj}\}$. If the injection delay is fixed Δ_P^{inj} for the store operations in P, the access time of those requests to the bus, and hence the time among them Δ_P^{bus} may vary depending on the *sad* scenario.

In order to illustrate this scenario we assume a program with $\Delta_P^{inj} = \{-, 4, 1\}$ in which all operations are stores. Stores are sent to a 2-entry store buffer from where they access a TDMA-arbitrated bus. Figure 8.1 shows the timing of the different requests depending on the relative ready cycle of r_0. Note that requests are considered as completed once they are sent to the store buffer. For instance, in the first scenario ($cyc_0^{rel} = 1$, so the first shaded row) r_0 becomes ready in cycle 0 in which it is buffered (b_0) and it is served in cycle 1 (s_0). r_1 becomes ready 4 cycles after that, and it is put in the buffer b_1 until the next slot for the core starts in cycle 8. Once r_1 is in the buffer in cycle 4, it is considered completed, so in cycle 5 r_2 is processed, i.e. also sent to the buffer b_2. Once the slot for this core starts in the second TDMA window, r_1 and r_2 are served consecutively in cycles 8 and 9. Thus, it takes 10 cycles to send all requests (from cycle 0 till 9). In the second scenario ($cyc_0^{rel} = 2$) r_0 enters the store buffer in cycle 1 and cannot be sent to the bus in cycle 2 because the $S0$ slot has elapsed. r_1 is queued in cycle 5 and the store buffer is full. Thus, although r_2 gets ready in cycle 6, it cannot enter the buffer until an entry is released, which occurs in cycle 8 when r_0 is sent to the bus. Then r_1 is sent in cycle 9 and r_2 has to wait until cycle 16 to be granted access to the bus. Thus, it takes 16 cycles to send all requests (from cycle 1 till 16). Overall, each different *sad* takes 10, 16, 15, 14, 13, 13, 12 and 11 cycles respectively.

In Figure 8.1 we observe that the number of different *sad* scenarios, impacting both the *sad* of the different requests and the program execution time, is limited to $w-1$. This leads to the following observation.

Observation 1: *The impact of* sad *on a program with different asynchronous requests is determined by* cyc_0^{rel}. *Under each different* cyc_0^{rel} *scenario the* sad *for each request – and hence the impact on the program's execution time – may vary.*

As with synchronous requests, the execution time difference between different *sad* can only be up to $w-1$ cycles. To illustrate it, let us take as a reference the scenario executing the fastest (e.g., first scenario in Figure 8.1) and any other arbitrary *sad scenario*. Let $shift^{sad}$ be the cycle count difference between both scenarios – $sad_{fastest}$ and sad_{slow} – such that sad_{slow} synchronizes with $sad_{fastest}$ as described in Section 8.3.2. By construction, $shift^{sad} < w$ given

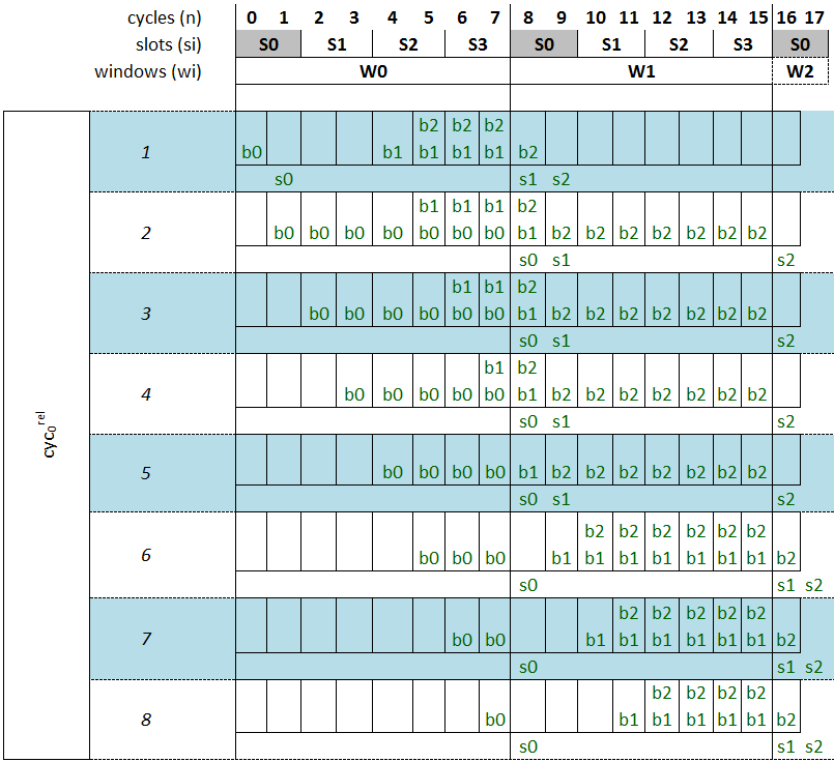


FIGURE 8.1: Example of 3 requests with $\Delta^{inj} = \{-, 4, 1\}$ and their *sad*. b_i are the cycles in which the request is ready but waiting in the buffer due to *sad*; s_i represents cycle in which the request gets access to the bus. Finally blanks represent the cycles with no requests on the bus.

that there are w different *sad* where the $shift^{sad}$ for the $w-1$ slowest ones w.r.t. the fastest one is 1, 2,... $w-1$ cycles respectively. Eventually, in sad_{slow} requests can wait $shift^{sad}$ cycles and execute identically as in $sad_{fastest}$, or it may be the case that they execute faster because during those $shift^{sad}$ cycles some requests find an available slot. This reasoning applies to each request individually given that, although they are injected synchronously (e.g., instructions are fetched synchronously), they access the bus asynchronously due to some buffering mechanism (e.g., requests are buffered in the store buffer without stalling the fetch stage). Thus, given that all requests can be served as in the fastest case if they get delayed by $shift^{sad}$ cycles, the execution time would be increased by $shift^{sad}$ at most, where $shift^{sad} < w$. If any request is served earlier, this cannot increase the execution time further because we rely on a processor free of timing anomalies. Hence, sad_{slow} can only take up to $shift^{sad} < w$ more cycles than $sad_{fastest}$.

Observation 2: *The maximum execution time impact between the different* sad *that a program with asynchronous requests may suffer is smaller than a TDMA window (w cycles). The execution time difference among two particular* r_0 *TDMA alignments, i.e.* cyc_0^{rel}*, is up to* $w-1$ *cycles.*

8.3.4 Multiple TDMA resources

When several TDMA-arbitrated resources are used in the system (e.g., k TDMA resources), at most $lcm(w_1, w_2, ..., w_k)$ different *sad* scenarios across all k TDMA resources exist, where *lcm* stands for the *least common multiple*.

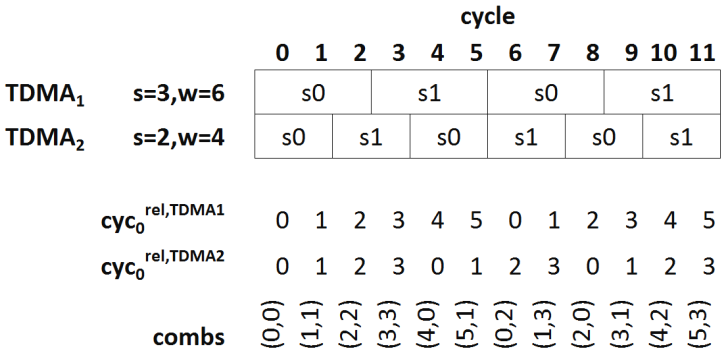


FIGURE 8.2: Different combinations – in a two TDMA-window case – for $cyc_0^{rel,TDMA1}$ and $cyc_0^{rel,TDMA2}$.

The key factor in determining the impact of crossing k TDMA resources is the relative cycle in which the first request, r_0, becomes ready across all TDMA windows. Hence, for the case of two TDMA windows there are a total of $lcm(w_1, w_2)$ combinations of $cyc_0^{rel,tdma1}$ and $cyc_0^{rel,tdma2}$. For instance, in Figure 8.2 we have an example with two TDMA resources each one with two slots. Slots in the first and second TDMA resource have 3 and 2 cycles respectively. Thus, $w_{tdma1} = 6$ and $w_{tdma2} = 4$. This leads to a total of $lcm(w_{tdma1}, w_{tdma2}) = 12$ different *sad*, shown in the last row. In this case, we consider all those 12 *sad* scenarios. Based on the arguments given before, the execution time in the worst *sad* scenario is at most 11 cycles worse (slower) than in the best *sad* as this is the longest time needed to align the slots across both TDMA resources. Thus, the same rationale used for a single TDMA resource can be applied in this case.

Observation 3: *When multiple TDMA resources are used, those TDMA resources can create execution time variations of up to* $lcm(w_1, w_2, ..., w_k) - 1$ *cycles due to* sad.

8.3.5 Other considerations

Split Requests. As explained before, some requests to the bus are split. For example a L1 cache miss may require a split request to access first the L2 cache, get a response indicating it misses in L2, and some time later get the data back with the second part of the request that has been split. In any case these two requests originated by the split mechanism are either synchronous or asynchronous and the same observations presented in previous sections for independent synchronous and asynchronous requests apply in this case.

Variable injection rate. Let $fc(I_{ri})$ be the cycle in which the instruction generating a request to the bus is fetched. So far in our discussion we have assumed a fixed injection rate across *sad* scenarios. That is, $\delta_i^{inj} = fc(I_{ri}) - fc(I_{ri-1})$ is the same for any consecutive pair of instructions under any two *sad* scenarios. In reality, however, if under some scenarios the instructions between I_{r_i} and $I_{r_{i-1}}$, after the execution of $I_{r_{i-1}}$ block the pipeline such that I_{r_i} cannot be fetched then δ_i^{inj} varies across *sad* scenarios. However, δ_i^{inj} is determined by a combination of synchronous and asynchronous events: pipeline stalls bring the synchronous component whereas buffering capabilities of the pipeline bring the asynchronous component. Hence, delaying timing events by at most the cycles between the current *sad* and the one leading to the fastest execution is enough

to have the same execution behavior from that point onwards. Anything occurring with a delay shorter than that cannot lead to a longer execution time in a processor free of timing anomalies. Overall, the maximum impact on execution time of TDMA is limited to $lcm(w_1, w_2, ..., w_k) - 1$ cycles.

8.4 TDMA in the context of MBPTA

In this section we show how TDMA affects WCET estimation under MBPTA. We start by introducing the particular timing characteristics of MBPTA-compliant processors.

8.4.1 Timing of MBPTA-Compliant Processors

DTA-compliant processors experience deterministic latencies in the different resources and hence, execution time can be regarded as deterministic given a set of initial conditions. This occurs because each event leads to a single (deterministic) outcome and so, a single processor state can be reached. This is not the case for MBPTA-compliant processors, in which a number of random events may alter the execution time, thus leading to a different number of states, each of which is reached with a given probability as shown in [160]. We refer to those states as *probabilistic processor states*.

We illustrate through a synthetic example how those different states influence the latency between different bus requests. We consider a processor in which instructions take a fixed latency and where memory operations are all loads. Load operations access a time-randomized data cache [161], which is the only source of execution time variability (the instruction cache is assumed perfect)[3]. The total latency of a load that misses in cache, in the absence of any contention, is 100 cycles: 1 cycle to access cache, 1 cycle to traverse the bus and 98 cycles to fetch data. Note that in this simple example we assume no contention to send data from memory to the core. In this first experiment we also consider that, whenever a load misses in cache, main memory is reached through a bus that *creates no contention*. Let us assume that the program under analysis has the following sequence of instructions $I = \{ld_0, i_0, i_1, ld_1, i_2, ld_2, i_3, i_4, i_5, i_6, i_7, ld_3\}$. Further assume that ld_0 always misses in cache and the other three load operations — ld_1, ld_2 and ld_3 — have an associated hit probability of 75%, although the actual value of those probabilities is irrelevant for the example. Other core instructions — i_0, i_1,... i_7 — do not access the data cache and have a fixed 1-cycle latency.

In this architecture, load operations generate a new probabilistic state in the execution as shown in Figure 8.3. Every access leads to two possible probabilistic states (hit or miss), each with an associated probability. In that respect, there is a probability for each of the 8 possible combinations of hit-miss outcomes of the 3 load instructions ($hhh, hhm, hmm, ..., mmm$), which can be easily derived (e.g., $0.75 \cdot 0.25 \cdot 0.75 = 0.140625$ for the hmh case). Interestingly, any execution of the program can only lead to one of those 8 probabilistic processor states, and for

[3]These assumptions simplify the discussion in this section. In Section 8.5 we consider a multicore processor with time-randomized data and instruction caches.

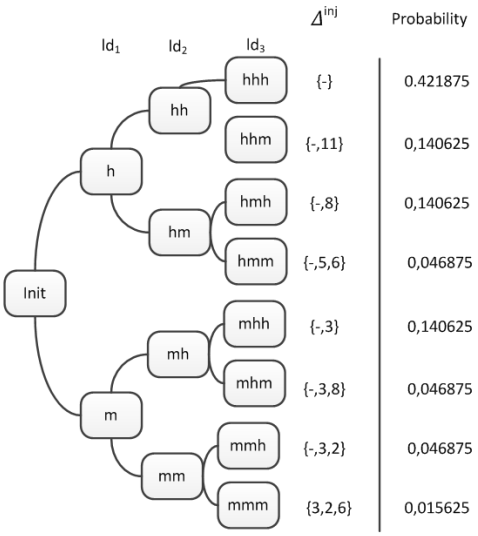


FIGURE 8.3: Different probabilistic states in which the processor may be after the execution of each of the 3 loads in the example.

each of them the delay among requests is fixed. Moreover, each such state (and set of delays among requests) occurs with a given probability. For instance, for the sequence mhm, which occurs with a probability of 0.046875, $\Delta^{inj} = \{-, 3, 8\}$ since 3 cycles elapse between ld_0 and ld_1, in which i_0 and i_1 are executed and ld_1 requires an extra cycle to access cache. Analogously, 8 cycles elapse between ld_1 and ld_3 to execute 7 1-cycle instructions before ld_3 accesses cache.

In a second experiment, instead of assuming a no-contention bus, we use assume TDMA arbitration for the bus that is shared among 4 cores. For TDMA the slot for each core is $s = 2$ cycles with windows of $w = 8$ cycles. The execution time of the program under each probabilistic state is affected by the bus contention. Hence, the observations made in Section 8.3 for the impact of TDMA on execution time are to be considered for *each probabilistic state* in a MBPTA-compliant processor.

8.4.2 TDMA analysis with MBPTA

As explained in Section 8.3.2, a shared resource implementing a TDMA arbitration policy may introduce execution time variations of up to $w - 1$ cycles, where w is the window size. From the point of view of MBPTA, the *sad* suffered by each request is indeed a *setv*. Hence, *sad* for TDMA is ruled by the same principles as other *setv*: its jitter has to be upper-bounded deterministically or probabilistically.

Observation 4: *In the absence of MBPTA-specific support, TDMA is not by default analyzable with MBPTA because one cannot prove that the delay experienced by each request (and hence the whole program) at analysis time due to the alignment with the TDMA slots upper-bounds the impact of TDMA at deployment.*

In the case of MBPTA, we have shown that each probabilistic state leads to a different Δ^{inj}, thus making the impact of the TDMA slot alignment different for each such states. Intuitively,

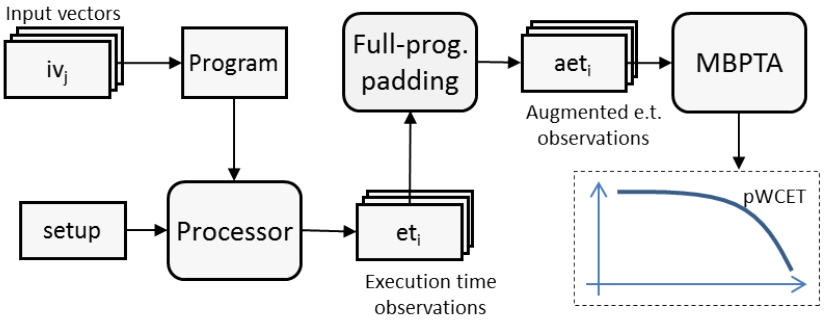


FIGURE 8.4: Full-program padding in the context of MBPTA.

one should consider the TDMA *sad* alignment individually for each probabilistic state to account for TDMA impact in execution time. However, this may be overly expensive since the number of probabilistic states grows exponentially with the number of probabilistic events [148, 160]. A different approach is needed to account MBPTA impact on execution time and pWCET estimates.

8.4.3 Full-program padding

We rely on the knowledge acquired in Section 8.3 on the maximum impact that TDMA can incur in the execution time of a program to propose a solution that has minimum impact on pWCET estimates. In particular, we show that the maximum impact that the alignment with respect to the TDMA window that a program can suffer is limited to w, so the maximum difference in execution time (i.e. jitter) between two runs of the same program due to TDMA is limited to $w-1$ cycles when one TDMA-arbitrated resource is used and $lcm(w_1, w_2, ..., w_k) - 1$ when $k > 1$ TDMA resources are used.

Hence, we could increase the execution time observations obtained at analysis time by $w-1$ cycles without breaking MBPTA compliance and trustworthily upper-bound the effect of TDMA alignment in the execution time. The process is as depicted in Figure 8.4.

MBPTA [146, 147] performs several runs of the program under analysis on the target platform for a set of input vectors, labeled as iv_j in Figure 8.4. These runs are done under a setup in which the seeds for the hardware random generators as well as other setup parameters are properly initialized by the system software. As a result of this step, several execution time observations (et_i) are obtained. With the full-program padding approach, there is no need to control the *sad* for each run. Each of et_i is augmented leading to a set of augmented execution time observations as shown in Equation 8.4, where k is the number of TDMA-arbitrated resources.

$$aet_i = \begin{cases} et_i + w - 1 & \text{if } \ k = 1 \\ et_i + lcm(w_1, w_2, ..., w_k) - 1 & \text{if } k > 1 \end{cases} \tag{8.4}$$

The augmented observations, which deterministically upper-bound the maximum impact of TDMA *sad* alignment, are passed as input to MBPTA that obtains a pWCET estimate trustworthily upper-bounding the impact of TDMA *sad*. Note that augmenting all observations may be

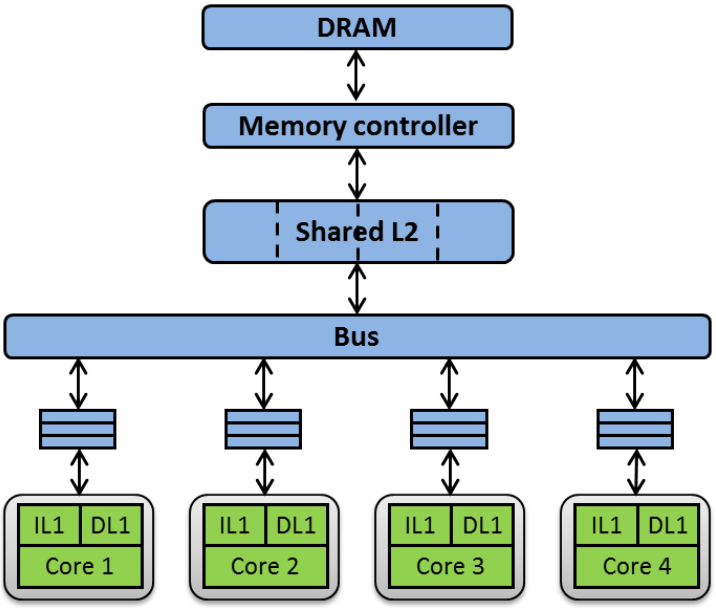


FIGURE 8.5: Schematic of the multicore processor considered.

pessimistic since the actual *sad* experienced might not be the fastest one. However, as shown later in our evaluation, such pessimism is irrelevant in practice.

8.5 Results

In this section we first introduce the evaluation framework. Next, we examine how TDMA *sad* impacts execution time. Finally, we compare 3 arbitration policies: TDMA, Interference Aware Resource Arbiter (IARA) based on round-robin [86] and a MBPTA-specific randomized arbitration policy called random permutations [150].

8.5.1 Evaluation Framework

Processor setup. We use a cycle-accurate modified version of the SoCLib [37] framework modeling a multicore processor as the one shown in Figure 8.5 (see Section 2.1). We use 3-stage in-order execution cores. Caches implement random placement and random replacement[4]. First level data (DL1) and instruction (IL1) caches are 8KB, 4-way with 32-byte lines. DL1 is write-through. The L2 is 128KB, 8-way with 32-byte lines. The L2 deploys cache partitioning, in particular way-partitioning as implemented in real processor like ARM A9 or Aeroflex NGMP, so that each core has exclusive access to 2 ways. This prevents contention in the cache as it is hard to model. These cache designs have been shown to be MBPTA compliant [154, 161, 162]. Cache latencies are 1 cycle for DL1/IL1 and 2 cycles for L2. Note that L2 turnaround time can be typically around 10 cycles due to 2 bus traversals to send the request and receive its corresponding answer.

[4]Time-randomized caches have been shown effective in conjunction with MBPTA [161, 162]. It is worth mentioning that their use in the context of MBPTA has been regarded as risky in [163]. However, authors in [164, 165] provide detailed arguments about those concerns and why time-randomized caches can be used safely.

TABLE 8.2: Maximum exec. time variations due to TDMA *sad*.

Bench.	TDMA bus only	TDMA bus and mem.ctrl.	Bench.	TDMA bus only	TDMA bus and mem.ctrl.
a2time	7	215	**idctrn**	7	215
aifftr	7	215	**iirflt**	7	111
aifirf	7	111	**matrix**	7	215
aiifft	7	215	**pntrch**	7	111
basefp	7	215	**puwmod**	7	111
bitmnp	7	215	**rspeed**	7	111
cacheb	7	111	**tblook**	7	111
canrdr	7	111	**ttsprk**	7	111

There are two independent buses to send requests from cores to L2 and to send answers from L2 back to the cores. Both buses have a 2-cycle latency once access is granted.

We use a time-analyzable memory controller [166] with per-request queues. We assume a Central Processing Unit (CPU) frequency of 800MHz and DDR2-800E SDRAM with the memory controller implementing close-page and interleaved-bank policies, which delivers 16-cycles access latency and 27-cycles inter-access latency [64]. Thus, an access completes in 16 cycles once it is granted access to memory, but the next access has to wait 11 extra cycles to start to allow the page accessed to be closed. This typically leads to memory latencies around 100 cycles due to contention and access delay.

In our experiments, to control the access to both the bus and memory controller, we deploy three different arbitration policies: random permutations [150], IARA based on round-robin [86, 152] and TDMA. The particular policy used in each experiment is indicated conveniently.

Benchmarks. We consider the EEMBC Autobench benchmarks [43], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems (see Section 2.2.3).

When computing pWCET estimates, we collected 1,000 execution times for each benchmark, which proved to be enough for MBPTA [146]. The observations collected in all the experiments passed the independence and identical distribution tests as required by MBPTA [146].

8.5.2 Impact of TDMA *sad* on Execution Time

In this section we empirically confirm that the impact of TDMA resources is at most w cycles when a single TDMA resource is used and $lcm(w_1, w_2, ..., w_k)$ cycles for k TDMA resources.

Single TDMA resource. For this experiment we use a TDMA-arbitrated bus to access L2. Bus latency is 2 cycles and $w_{bus} = 8$ (4 slots for the 4 cores, each slot of $s = 2$ cycles). The responses from the L2, which is assumed perfect (i.e. all accesses hit) arrive in a fixed latency of 2 cycles. DL1/IL1 cache memories are always initialized with the same seeds so that the random events produced are exactly the same across all experiments. This way the only *setv* is the *sad* for the bus. We run 8 experiments with the 8 different *sad* for each benchmark. The *"TDMA bus only"* columns in Table 8.2 show the maximum execution time variation observed for each benchmark. As shown, *all* benchmarks observe exactly a maximum difference of $w_{bus} - 1 = 7$ cycles. In fact, we have corroborated that execution times for the 7 slowest *sad* of each benchmark are exactly 1, 2, 3, 4, 5, 6 and 7 cycles higher than that of the fastest *sad*. This means that in

all runs at some point requests get delayed until they align (synchronize) with TDMA as in the fastest case, and then execution continues identically.

Multiple TDMA resources. For this experiment we use the original processor setup. We have 3 TDMA resources: the buses to reach L2 and get answers from it, and the memory controller. Both buses have $w_{bus} = 8$, 2-cycle slots. The memory controller has 27-cycle slots, so $w_{memctrl} = 108$ cycles due to the 4 contender cores. Thus, $lcm(8, 8, 108) = 216$. Experiments are run as before fixing seeds for caches so that execution time variations are produced only due to the alignment with TDMA resources. We have run 216 experiments for each benchmark with the 216 different *sad*. The *"TDMA bus and mem. ctrl."* columns in Table 8.2 show the maximum execution time variation observed for each benchmark. As shown, such difference is at most $lcm(8, 8, 108) - 1 = 215$ cycles, thus further corroborating our hypothesis. In fact, in 7 out of the 16 benchmarks such difference is exactly 215 cycles. In the other 9 cases it is 111 cycles. Those 111 cycles come from the fact that the memory controller window is much larger than the bus one, and in some cases it is enough to align with such window to get identical or near identical timing behavior as in the fastest case. This explains $w_{memctrl} - 1 = 107$ cycles. The other 4 cycles correspond to the misalignment of the TDMA bus windows after $w_{memctrl} = 108$ cycles.

8.5.3 Performance Comparison

We evaluate arbitration policies in terms of worst-case performance, which is measured with the probabilistic WCET estimates provided by MBPTA. In all experiments, we use the same arbitration policy in the buses and in the memory controller. Seeds for the caches are initialized randomly on each run. We use the following setup for each policy:

- **Time-randomized**. We use random permutations arbitration, with which on every arbitration window a random permutation of the slots is created so that in every window the contenders access the bus in a random fashion [150].
- **IARA**. Bus latency is always 8 cycles (4 cores x 2-cycle latency). Memory latency is always 97 cycles due to the 3 slots for the other cores (3 x 27) and the 16-cycle access of the current request.
- **TDMA**. With TDMA experiments are run assuming always an arbitrary *sad*. We use full-program padding increasing the observations passed to MBPTA by 215 cycles.

Since we use non-work-conserving versions of all arbitration policies, the task under analysis can only access the resources in its slot (time-randomized and TDMA). Those slots in which it is not granted access, the task cannot access the shared resource even if it is idle. This makes irrelevant what is run in the other cores or whether they are idle. Thus, pWCET estimates are time-composable (do not depend on the co-runners as needed to isolate across different criticalities), and results can be obtained keeping the other cores idle.

pWCET estimates. Figure 8.6 shows the pWCET estimates for each benchmark. We use a cutoff probability of 10^{-15} per activation as it has been shown appropriate to use in some domains

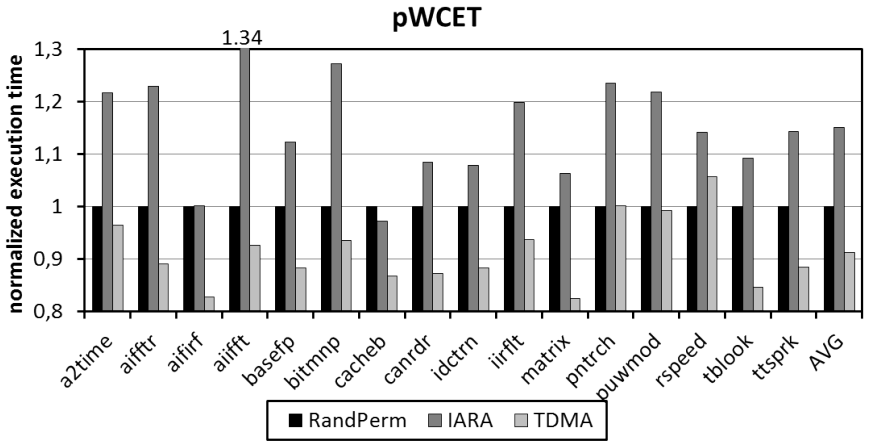


FIGURE 8.6: pWCET estimates for a cutoff probability of 10^{-15} normalized w.r.t. time-randomized arbitration.

as avionics [154]. Results have been normalized with respect to the time-randomized bus. IARA is 15% worse than random permutations on average. IARA is the worst policy since it assumes each request to experience its worst-case latency. TDMA is 9% better on average than random permutations because TDMA slots for a given core are homogeneously distributed in time, thus leaving some time between consecutive slots. Conversely, random permutations may lead with relatively high probability to consecutive slots assigned for a given core in the memory controller because it is granted last in one permutation and first in the next one. However, some cycles elapse since the data reach the core for a load request until the next request (either a load or a store) from this core reaches the memory controller. This is enough to miss its opportunity and wait for a later slot that will not show up until the next permutation. Overall, although the average time between slots for random permutations and TDMA is the same, under random permutations some slots cannot be used and performance (and so WCET estimates) is affected.

Differences for individual benchmarks w.r.t. the average case occur due to the random variations that affect measurements, which may lead to higher or lower tightness in some cases [167]. Still, results are quite consistent across benchmarks.

8.6 Related work

Several works analyze, from a SDTA point of view, the impact of on-chip bus arbitration policies, specially TDMA [54, 168] and round-robin [86], on WCET. In [54] an analysis and evaluation of a TDMA arbitrated bus under the context of SDTA, considering both architectures with and without timing anomalies is performed. In [86] an analysis of the delay that every request can suffer when accessing a round-robin arbitrated resource is carried out.

More complex inter-connection architectures such as meshes [169] or rings [170] based on the use of TDMA and round-robin have also been shown to be analyzable with SDTA techniques. For the TDMA case the Time-Triggered Architecture [171] (TTA) implements timing-predictable communication by means of customized TDMA schedules. Other approaches like T-CREST [169]

deliver low complexity TDMA-based Network on Chips (NoCs) with global schedule that enable straightforward WCET analysis. For round-robin several studies [172, 173] propose offering several levels of round-robin arbitration for asymmetric upper-bound delay (*ubd*) so that high priority tasks may enjoy lower *ubd*. In [152, 153] authors present a comparison of TDMA and round-robin for SDTA and MBDTA considering different metrics.

For MBPTA several specific arbitration policies have been proposed which includes random lottery access [174] and random permutations [150], both based on the idea of introducing some type of randomization when granting access to the different contenders. With the lottery bus on every (slot) round the grant is given randomly to one of the resource contenders. With random permutations, on every window a random permutation of the slots is assigned so that in every window the contenders access the bus in a random fashion. To the best of our knowledge this is the first attempt to analyze the benefits of TDMA, a DTA-amenable arbitration policy, in the context of MBPTA.

8.7 Conclusions

Different types of timing analyses impose heterogeneous constraints on hardware designs, so chip vendors have to face the challenge of deciding which timing analysis to support (if any). Hence, proving that the same hardware design can be used to obtain trustworthy and tight WCET estimates with different families of timing analyses is of prominent importance to increase the chance of those hardware designs being realized (Objective **O1**).

We have shown that shared resources implementing TDMA arbitration, which meet mixed-criticality systems requirements, can be analyzed in the context of MBPTA (Objective **O3**). We introduce small changes to the application of MBPTA with which WCET estimates obtained are 9% lower on average than those obtained with MBPTA-friendly designs.

Chapter 9

Conclusions, Impact and Future Work

There is a strong requirement to increase the computation capabilities of CRTES to cope with the necessity of increasing the functional value of systems and so stay competitive in the market. Many-core processor architectures are increasingly seen as the solution to safisty this performance requirement. On one side, many-core architectures significantly increase system's performance by exploiting the parallelisms exhibited by applications while providing a better performance per watt ratio due to its simpler core design compared to single-core architectures. On the other side, many-cores allow scheduling multiple functionalities within the same computing unit, leading to a reduction in SWaP costs.

However, the use of many-core architectures complicate significantly the extraction of timing guarantees, a fundamental requirement of CRTES. Concretely, the interferences generated when multiple applications access simultaneously shared hardware resource (e.g., interconnection networks, memory controllers) impacts on the timing behavior of the system, as it becomes dependent on application's workload, breaking a fundamental design pillar in CRTES: time composability. Time composability enables incremental development and incremental verification of integrated systems, and so allows independent application/system development, across several vendors.

This thesis has advanced one step towards the adoption of many-core processor architectures in current and future CRTES designs from the hardware and software perspective. From the hardware perspective:

- This thesis has proposed a new hardware feature called *Guaranteed Resource Partition* (GRP) that defines an execution environment composed of a cluster of processor resources in which parallel applications run, providing the required timing isolation properties among systems' components. This is done by implementing a transparent execution mechanism that freezes internal GRP requests in favour to communications among GRPs, allowing to account for the impact of communications at system integration time. GRPs are the

basis to provide the timing isolation properties of *Parallel Software Partitions* (pSWPs), an extension of the software partition concept defined in the ARINC 653 and the AUTOSAR standards.

- This thesis has analyzed the suitability of applying wormhole NoC (wNoC) designs in many-core processor design targeting CRTES. To do so, this thesis has proposed a new metric, named Worst-Contention Delay (WCD), that enables accounting for the impact of NoC interferences coming from different requestors on the WCET estimates. The WCD allows to derive an analytical model that computes time-composable WCD bounds based on common wNoC design parameters, covering a wide range of existing wNoCs. To that end, we apply the model considering the design parameters of two wNoCs deployed in real processors: the Tilera-Gx36 and the 48-core Intel SCC. Moreover, this thesis has proposed a minimal set of hardware modifications on the wNoC design that enables a fair sharing of the available bandwidth across the different flows in the network, making time-composable WCET estimates less affected by the core count in many-core designs.

From the software perspective, this thesis has proposed new allocation strategies that assigns processor resources within GRPs and across them, that takes full benefit of the proposed many-core hardware architecture:

- The allocation strategy within GRPs (named *RunPar*), suitable for the AUTOSAR framework (and similar ones), enables to maintain the same configuration of AUTOSAR applications, i.e. its runnable-to-task mapping and single-core task scheduling, when migrating from single-core to many-core processor designs, so the effort of re-validating the applications is minimized. To do so, RunPar maintains the single-core task scheduler by exploiting only the parallelism of the units of scheduling (named *runnables* in case of AUTOSAR) belonging to the same task.

- The allocation strategy across GRPs (named *CAP*) minimizes the impact that communication across GRPs has on the WCET of applications, while facilitating system integration. To do so, it constructs the schedule starting from consumer applications first giving higher priority to those belonging to the chains of dependencies with longest utilization. Then, for each inter-application communication that crosses the cluster boundaries, CAP creates a zone of communication impact inside which applications allocated to the destination cluster are given additional computational resources.

Finally, this thesis also explores the use of probabilistic timing analyses techniques when considering TDMA arbitration policy to have access to shared hardware resources, proving that the same hardware design can be used to obtain trustworthy WCET estimates with different families of timing analyses.

9.1 Impact and Future Work

The work done in this thesis has opened several research lines targeting new challenges in CRTES some of which are already covered by other PhD students in the Universitat Politecnica de Catalunya (UPC).

Based on the proposals described in this thesis, which in our humble opinion has been one of the first attempts to design a time composable many-core processor architecture, the European FP7 project PROXIMA [175] has explored extensions to the NoC designs proposed in this thesis to support probabilistic timing analysis techniques, extending the work introduced in Chapter 8. It is also worth mentioning the collaboration with DENSO, a Japanese automotive company interested in exploiting the RunPar scheduling strategy presented in Chapter 6 into the AUTOSAR execution framework. Finally, the many-core designs and the static scheduling strategies have been the baseline for the PhD thesis entitled "Parallelization of Automotive Control Software" [176].

A research line initiated upon the results of this thesis is the use of parallel programming models to facilitate the programmability of many-core architectures. In that regard, several PhD theses at the UPC have started tackling this challenge. It is worth mentioning the research conducted within the European FP7 project P-SOCRATES [177], that explored the time predictability properties of the OpenMP parallel programming model. The project investigated the use of OpenMP to develop CRTES on top of clustered-based many-core processor architectures supporting guaranteed resource particitions (GRPs) presented in Chapter 3. Concretely, the project considered the MPPA 256 processor from Kalray [4] featuring a many-core fabric composed of 256-cores, organized in 16 clusters with 16-cores each. Similarly, a project supported by the European Space Agency (ESA) [178] investigated the use of OpenMP in the space domain. Finally, the OpenMP Advisory Review Board (ARB), the organization in charge of the evolution of this parallel programming model, is currently considering introducing changes on the standard to enable the development of CRTES with OpenMP.

There are still a number of challenges to be investigated with respect to the allocation strategies implemented at operating system level to properly schedule the units of parallelism defined by the parallel programming model, and its impact on time predictability. Here, the NoC plays a fundamental role, as the communication characteristics among the differtent units of parallelism impacts of the timing behavior of the system. In that regard, the proper definition of the weights used in the wNoC design presented in Chapter 4 by means of new allocation scheduling strategies can minimize the impact of NoC interferences on the different units of parallelism. Some work on this line has been initiated recently at UPC as part of a Master thesis where those weighted wNoCs are used together with parallel applications in CRTES to optimize their WCET by combining weight assignment and thread-to-core mapping.

Clearly, as already identified in Chapters 3 to 5, the NoC and the memory are hardware components that dramatically impacts on the time predictability of the system. The increasing adoption of accelerators in CRTES, such as GPUs in automotive systems for autonomous driving, require the proper design and verification of NoCs and GPU-to-memory interconnects. While the organization of those architectures and the data traffic may differ from those in the many-cores proposed

and studied in this thesis, concepts such as complex NoCs, contention, time-composability and time-predictability need to be accounted for similarly. Hence, part of the future work is leveraging the know-how developed in this thesis to design and optimize those newer architectures so that reliable and tight timing guarantees can be obtained while making an efficient use of their resources.

List of Publications

Publication included in this thesis (in chronological order):

- Milos Panic, Eduardo Quiñones, Pavel G. Zaykov, Carles Hernández, Jaume Abella, Francisco J. Cazorla: **Parallel many-core avionics systems**. EMSOFT 2014
- Milos Panic, Sebastian Kehr, Eduardo Quiñones, Bert Boddeker, Jaume Abella, Francisco J. Cazorla: **RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores**. CODES+ISSS 2014
- Milos Panic, Eduardo Quiñones, Carles Hernández, Jaume Abella, Francisco J. Cazorla: **CAP: Communication-Aware Allocation Algorithm for Real-Time Parallel Applications on Many-Cores**. DSD 2015
- Milos Panic, Jaume Abella, Carles Hernández, Eduardo Quiñones, Theo Ungerer, Francisco J. Cazorla: **Enabling TDMA Arbitration in the Context of MBPTA**. DSD 2015
- Milos Panic, Carles Hernández, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla: **Modeling High-Performance Wormhole NoCs for Critical Real-Time Embedded Systems**. RTAS 2016
- Milos Panic, Carles Hernández, Jaume Abella, Antoni Roca, Eduardo Quiñones, Francisco J. Cazorla: **Improving performance guarantees in wormhole mesh NoC designs**. DATE 2016

Other publications:

- Milos Panic, Germàn Rodrìguez, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla: **On-chip ring network designs for hard-real time systems**. RTNS 2013
- Theo Ungerer, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, J. Fernandes, Pavel G. Zaykov, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Ian Broster, Nick Lay, David George, Eduardo Quiñones, Milos Panic, Jaume Abella, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde, Arthur Pyka: **parMERASA - Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability**. DSD 2013
- Sebastian Kehr, Milos Panic, Eduardo Quiñones, Bert Böddeker, Jorge Becerril Sandoval, Jaume Abella, Francisco J. Cazorla, Günter Schäfer: **Supertask: Maximizing runnable-level parallelism in AUTOSAR applications**. DATE 2016
- Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel G. Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernández, Francisco

J. Cazorla, Sascha Uhrig, Mathias Rohde, Arthur Pyka: **Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore**. ACM Trans. Embedded Comput. Syst. 15(3): 53:1-53:27

- Milos Panic, Jaume Abella, Eduardo Quiñones, Carles Hernández, Theo Ungerer, Francisco J. Cazorla: **Adapting TDMA arbitration for measurement-based probabilistic timing analysis**. Microprocessors and Microsystems - Embedded Hardware Design 52

Bibliography

[1] Transparency Market Research. *Embedded System Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2015 - 2021*, 2015. URL http://www.transparencymarketresearch.com/embedded-system.html.

[2] Reinhard et al Wilhelm. The Worst-Case Execution-Time Problem Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3): 36, 2008.

[3] D. Rahmati, et al. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 62(3), 2013. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2011.240.

[4] *Kalray MPPA 256 Many-Core Processor, http://www.kalray.eu/ products/mppa-manycore,*.

[5] Global Market Insight. *Embedded System Market Size By Application (Automotive, Industrial, Consumer Electronics, Telecommunication, Healthcare, Military & Aerospace), By Product (Software, Hardware) Industry Outlook Report, Regional Analysis, Application Development Potential, Price Trends, Competitive Market Share & Forecast, 2016 2023*, 2016. URL https://www.gminsights.com/industry-analysis/embedded-system-market.

[6] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. P-SOCRATES: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems - Embedded Hardware Design*, 39(8): 1190–1203, 2015. doi: 10.1016/j.micpro.2015.06.004. URL http://dx.doi.org/10.1016/j.micpro.2015.06.004.

[7] Forbes. *Weak Desktop Sales Impact Intel's Q1'15 Earnings, Data Center, IoT; NAND See Double Digit Growth*, 2015. URL http://onforb.es/1mXq5yW.

[8] Financial Times. *Internet of things drives Intel revenues*, 2015. URL http://on.ft.com/1oH1QXI.

[9] Financial Times. *Arm profits and sales up as shift away from mobile gains pace*, 2016. URL http://on.ft.com/1T6I8Bi.

[10] Intel. *Next-Generation Transportation*, 2017. URL http://www.intel.com/content/www/us/en/automotive/automotive-overview.html.

[11] Markus Dillinger, Kambiz Madani, and Nancy Alonistioti. *Software Defined Radio: Architectures, Systems and Functions*. 2003. ISBN 9780470851647.

[12] Marc Duranton, Koen De Bosschere, Albert Cohen, Jonas Maebe, and Harm Munk. The hipeac vision 2015, 2015.

[13] G. Edelin. Embedded systems at thales: the artemis challenges for an industrial group. In *ARTIST*, 2009.

[14] Silabs. http://www.silabs.com/Marcom%20Documents/Resources/automotive-applications-guide.pdf, 2013.

[15] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.

[16] Infineon. AURIX Safety joins Performance. http://www.infineon.com/cms/en/product/promopages/32-bit-microcontroller-for-automotive/index.html?intc=0140013.

[17] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time software. In *1st International Workshop on Software Technologies for Future Dependable Distributed Systems. 2009.*

[18] Tilera. *TILE-Gx Processors Family http://www.tilera.com/products/TILE-Gx.php.*

[19] *P4080 QorIQ Integrated Processor Hardware Specifications, Rev. 0, 02/2011, Doc. Num. P4080EC.* Freescale, 2011.

[20] Texas Instruments. *C6000 Multicore DSP + ARM SoC*, 2016. URL http://www.ti.com/lsds/ti/processors/dsp/c6000_dsp-arm/overview.page?paramCriteria=no.

[21] M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[22] Milos Panic, Carles Hernández, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. Modeling high-performance wormhole nocs for critical real-time embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 267–278, 2016. doi: 10.1109/RTAS.2016.7461342. URL http://dx.doi.org/10.1109/RTAS.2016.7461342.

[23] *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4, Subset Services.* ARINC Inc., June 2012.

[24] AUTOSAR consortium. AUTomotive Open System ARchitecture (AUTOSAR). Standard v4.1, 2014. URL www.autosar.org.

[25] *ARINC Report 651-1: Design Guidance for Integrated Modular Avionics.* ARINC Inc., November 1997.

[26] David Haworth, Tobias Jordan, Alexander Mattausch, and Alexander Much. Freedom from interference for autosar-based ecus: a partitioned AUTOSAR stack. In *Automotive - Safety & Security 2012, Sicherheit und Zuverlässigkeit für automobile Informationstechnik, 14.-15. November 2012, Karlsruhe, Proceedings*, pages 85–98, 2012. URL http://subs.emis.de/LNI/Proceedings/Proceedings210/article6842.html.

[27] A. Wilson and T. Preyssler. Incremental certification and Integrated Modular Avionics. In *DACS*, 2008.

[28] RTCA and EUROCAE. *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[29] *Road vehicles – Functional safety – Part 6: Product development at the software level, Ref. num. ISO 26262-6:2011(E)*. ISO, 2011.

[30] M. Panic, et. al. Parallel many-core avionics systems. *EMSOFT*, 2014.

[31] Milos Panic, Carles Hernández, Jaume Abella, Antoni Roca, Eduardo Quiñones, and Francisco J. Cazorla. Improving performance guarantees in wormhole mesh noc designs. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1485–1488, 2016. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7459546.

[32] M. Panic, et. al. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. *CODES+ISSS*, 2014. doi: 10.1145/2656075.2656096.

[33] Milos Panic, Eduardo Quiñones, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. CAP: communication-aware allocation algorithm for real-time parallel applications on many-cores. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 685–692, 2015. doi: 10.1109/DSD.2015.71. URL http://dx.doi.org/10.1109/DSD.2015.71.

[34] Milos Panic, Jaume Abella, Carles Hernández, Eduardo Quiñones, Theo Ungerer, and Francisco J. Cazorla. Enabling TDMA arbitration in the context of MBPTA. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 462–469, 2015. doi: 10.1109/DSD.2015.68. URL http://dx.doi.org/10.1109/DSD.2015.68.

[35] Milos Panic, Jaume Abella, Eduardo Quiñones, Carles Hernández, Theo Ungerer, and Francisco J. Cazorla. Adapting TDMA arbitration for measurement-based probabilistic timing analysis. *Microprocessors and Microsystems - Embedded Hardware Design*, 52: 188–201, 2017. doi: 10.1016/j.micpro.2017.06.006. URL https://doi.org/10.1016/j.micpro.2017.06.006.

[36] parMERASA. *EU-FP7 Project: http://www.parmerasa.eu/*.

[37] Soclib, http://www.soclib.fr/trac/dev, 2012.

[38] *NanoC: http://www.nanoc-project.eu*.

[39] RapiTime. *www.rapitasystems.com*, 2008.

[40] Clement Ballabriga, Hugues Casse, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *SEUS 2010*.

[41] Nicolas Pouillon, Alexandre Bécoulet, Aline Vieira de Mello, François Pêcheux, and Alain Greiner. A generic instruction set simulator API for timed and untimed simulation and debug of mp2-socs. In *Proceedings of the Twentieth IEEE/IFIP International Symposium on Rapid System Prototyping, Shortening the Path from Specification to Prototype, RSP 2009, Paris, France, 23-26 June 2009*, pages 116–122, 2009. doi: 10.1109/RSP.2009.11. URL https://doi.org/10.1109/RSP.2009.11.

[42] H. Jeffreys and B. S. Jeffreys. *Methods of Mathematical Physics, 3rd ed.* Cambridge University Press, 1988.

[43] Jason Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.

[44] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[45] R. Heckmann and R. Ferdinand. Worst-case execution time prediction by static program analysis. In *AbsInt White paper, 2009.*

[46] AUTOSAR. AUTomotive Open System ARchitecture, 2012. http://www.autosar.org.

[47] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

[48] M. Gerdes, et. al. Time analysable synchronisation techniques for parallelised hard real-time applications. In *DATE*, 2012.

[49] M. Gerdes, et. al. The split-phase synchronisation technique: Reducing the pessimism in the WCET analysis of parallelised hard real-time programs. In *RTCSA*, 2012.

[50] C. Rochange, et. al. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In *WCET workshop*, 2010.

[51] M. Paolieri, E. Quinones, and F. J. Cazorla. Timing effects of the memory system in real-time multicore integrated architectures: Problems and solutions. In *Transactions on Embedded Computing Systems*, 2012.

[52] B. Akesson, et. al. Predator: a predictable sdram memory controller. In *CODES+ISSS*, 2007.

[53] Yan Li, et. al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[54] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014.

[55] Javier Jalle, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Deconstructing bus access control policies for real-time multicores. In *SIES*, pages 31–38, 2013.

[56] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[57] Zheng Shi, et. al. Schedulability analysis for real time on-chip communication with wormhole switching. In *IJERTCS*, 2010.

[58] L. Benini, et. al. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *DATE*, 2012.

[59] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Elsevier, May 2004.

[60] J. Rattner. Single-chip cloud computer: An experimental many-core processor from Intel Labs. URL http://download.intel.com/pressroom/pdf/rockcreek/SCCAnnouncement.

[61] A. Roca, C. Hernandez, J. Flich, F. Silla, and J. Duato. Enabling high-performance crossbars through a floorplan-aware design. In *Intl. Conf.Parallel Processing (ICPP)*, pages 269–278, 2012.

[62] Y. Tamir and G. L. Frazier. High-performance multiqueue buffers for VLSI communication switches. In *ISCA*, 1988.

[63] B. Sinharoy, et. al. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3), May 2011. doi: 10.1147/JRD.2011.2127330.

[64] JEDEC. *DDR2 SDRAM Specification JEDEC Standard No. JESD79-2E*, April 2008.

[65] MERASA. *EU-FP7 Project: www.merasa.org*.

[66] P. Radojkovic, et. al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. In *HiPEAC*, 2012.

[67] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.

[68] R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *DACS*, 2010.

[69] A. Schranzhofer, et. al. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[70] S. Schliecker, et. al. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, 2010.

[71] A. Schranzhofer, et. al. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.

[72] D. Dasari and V. Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *HPCC-ICESS*, 2012.

[73] J. Sparsoe. Design of networks-on-chip for real-time multi-processor systems-on-chip. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 1–5, 2012. doi: 10.1109/ACSD.2012.27.

[74] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805821.

[75] Precision Timed Machines. http://chess.eecs.berkeley.edu/pret.

[76] K. Goossens, et. al. Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. *SIGBED Rev.*, 10(3):23–34, October 2013. ISSN 1551-3688. doi: 10.1145/2544350.2544353. URL http://doi.acm.org/10.1145/2544350.2544353.

[77] J. Duato, et al. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, 2002. ISBN 1558608524.

[78] Jos Flich and Davide Bertozzi, editors. *Designing network on-chip architectures in the nanoscale era*. Chapman & Hall/CRC computational science series. Chapman and Hall/CRC, 2011.

[79] S. Ramos and T. Hoefler. Capability models for manycore memory systems: A case-study with xeon phi knl. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, May 2017. doi: 10.1109/IPDPS.2017.30.

[80] Sunggu Lee. Real-time wormhole channels. *Journal Of Parallel And Distributed Computing*, 63:299–311, 2003.

[81] Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *IEEE/ACM NoCS*, 2009.

[82] D. Dasari, et al. Noc contention analysis using a branch-and-prune algorithm. *ACM Trans. Embed. Comput. Syst.*, 13(3s), March 2014. ISSN 1539-9087. doi: 10.1145/2567937. URL http://doi.acm.org/10.1145/2567937.

[83] P. Munk, et al. Dynamic guaranteed service communication on best-effort networks-on-chip. In *PDP*, 2015. URL http://dx.doi.org/10.1109/PDP.2015.47.

[84] B. Kim et al. A real-time communication method for wormhole switching networks. In *ICPP*, 1998.

[85] J. Jalle, et al. Deconstructing bus access control policies for real-time multicores. In *SIES 2013*, 2013.

[86] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[87] A. Burns, et al. A wormhole noc protocol for mixed criticality systems. In *RTSS*, 2014.

[88] *AMBA Bus Specification. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.*

[89] A. Roca, et al. VCTlite: Towards an efficient implementation of virtual cut-through switching in on-chip networks. In *HiPC 2010*, 2010.

[90] Mathieu Patte and Vincent Lefftz. System impact of distributed multi core systems. Technical Report ESTEC Contract 4200023100, ESA, 2011.

[91] José Duato, Sudhakar Yalamanchili, Blanca Caminero, Damon S. Love, and Francisco J. Quiles. Mmr: A high-performance multimedia router - architecture and design trade-offs. In *HPCA*, pages 300–309, 1999.

[92] E. Bolotin, et al. Qnoc: Qos architecture and design process for network on chip. *JOURNAL OF SYSTEMS ARCHITECTURE*, 2004.

[93] M. Schoeberl, et. al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *IEEE/ACM NoCS*, 2012.

[94] M. D. Gomony, et al. A generic, scalable and globally arbitrated memory tree for shared DRAM access in real-time systems. In *DATE*, 2015. URL http://dl.acm.org/citation.cfm?id=2755795.

[95] K. Goossens, et al. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

[96] R. Obermaisser, et al. The time-triggered system-on-a-chip architecture. In *ISIE*, 2008.

[97] Zheng Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *NoCS*, 2008. doi: 10.1109/NOCS.2008.4492735.

[98] B. Nikolic, et al. Worst-case communication delay analysis for many-cores using a limited migrative model. In *RTCSA*, 2014.

[99] Zheng Shi and Alan Burns. Real-time communication analysis with a priority share policy in on-chip networks. *ECRTS*, 2009.

[100] H. Kashif, et al. ORTAP: an offset-based response time analysis for a pipelined communication resource model. In *RTAS*, 2013. URL http://dx.doi.org/10.1109/RTAS.2013.6531097.

[101] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001. ISBN 3-540-42184-X.

[102] T. Ferrandiz, et al. A sensitivity analysis of two worst-case delay computation methods for spacewire networks. In *ECRTS*, 2012. doi: 10.1109/ECRTS.2012.35.

[103] RTCA Inc. RTCA DO-297 integrated modular avionics (IMA) development guidance and certification considerations. 2005.

[104] GENESYS. GENeric Embedded SYStem Platform. http://www.genesys-platform.eu.

[105] Hanmin Park and Kiyoung Choi. Position-based weighted round-robin arbitration for equality of service in many-core network-on-chips. NoCArc '12. ACM, 2012. ISBN 978-1-4503-1540-1. doi: 10.1145/2401716.2401728. URL http://doi.acm.org/10.1145/2401716.2401728.

[106] Antoni Roca. *Floorplan-Aware High Performance NoC Design*. PhD thesis, Universitat Politecnica de Valencia, 2012.

[107] Jörg Mische and Theo Ungerer. Guaranteed service independent of the task placement in nocs with torus topology. In *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 151, 2014. doi: 10.1145/2659787.2659804. URL http://doi.acm.org/10.1145/2659787.2659804.

[108] M. Millberg, et al. The nostrum backbone-a communication protocol stack for networks on chip. In *IEEE VLSI Design*, 2004. doi: 10.1109/ICVD.2004.1261005.

[109] E. A. Rambo and R. Ernst. Worst-case communication time analysis of networks-on-chip with shared virtual channels. DATE, 2015.

[110] T. Kranich and M. Berekovic. Noc switch with credit based guaranteed service support qualified for GALS systems. In *DSD*, 2010. URL http://dx.doi.org/10.1109/DSD.2010.30.

[111] A.Psarras, et al. Phase-noc: Tdm scheduling at the virtual-channel level for efficient network traffic isolation. DATE 2015.

[112] H. M. G. Wassel, et al. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. *SIGARCH Comput. Archit. News*, 41(3):583–594, June 2013. ISSN 0163-5964. doi: 10.1145/2508148.2485972. URL http://doi.acm.org/10.1145/2508148.2485972.

[113] Aeroflex Gaisler. *Quad Core (LEON4 SPARC V8) Processor - (LEON4-NGMP-DRAFT) - Data Sheet and Users Manual*, 2011.

[114] Aurelien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multisource software on multicore automotive ecus - combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, 2012.

[115] Hamid Reza Faragardi, Björn Lisper, and Thomas Nolte. Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores. In *Emerging Technologies and Factory Automation (ETFA)f*, pages 1–5, 2013.

[116] Jorn Schneider, Michael Bohn, and Robert Robger. Migration of automotive real-time software to multicore systems: First steps towards an automated solution. In *ECRTS, WIP Session*, 2010.

[117] AUTOSAR consortium. Specification of Operating System. Standard 4.1, 2014. URL www.autosar.org.

[118] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. ISSN 0272-1732.

[119] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET analysis of real-time parallel applications. In *WCET*, 2013.

[120] Licong Zhang, Reinhard Schneider, Alejandro Masrur, Martin Becker, Martin Geier, and Samarjit Chakraborty. Timing challenges in automotive software architectures. In *ICSE Companion*, pages 606–607, 2014.

[121] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990. ISBN 0716710455.

[122] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44 (12):1429–1442, 1995. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.477248.

[123] S.K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operation Research*, pages 127–140, 1978.

[124] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, 1995. ISSN 0922-6443. doi: http://dx.doi.org/10.1007/BF01088806.

[125] C Aussagues, D Chabrol, V David, D Roux, N Willey, A Tournadre, and M Graniou. Pharos, a multicore OS ready for safety-related automotive systems: results and future prospects. *Proc. of The Embedded Real-Time Software and Systems (ERTS2)*, 2010.

[126] T. Sakurai, H. Kawaguchi, and T. Kuroda. Low-power CMOS design through Vth control and low-swing circuits. In *ISLPED '97*, 1997.

[127] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems; Predictable Scheduling Algorithms and Applications*. 2011. ISBN 971-1-4614-0675-4.

[128] Christian Bradatsch, Florian Kluge, and Theo Ungerer. Synchronous Execution of a Parallelised Interrupt Handler. In *RTAS, WiP session*, apr 2014.

[129] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP*, pages 89–102, 2001.

[130] Rob Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys, available from http://www-users.cs.york.ac.uk/ robdavis/*, 2010.

[131] Irina Iulia Lupu, Pierre Courbin, Laurent George, and Joël Goossens. Multi-criteria evaluation of partitioning schemes for real-time systems. In *ETFA*, 2010. doi: 10.1109/ETFA.2010.5641218. URL http://doi.ieeecomputersociety.org/10.1109/ETFA.2010.5641218.

[132] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

[133] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[134] Alexander Wieder and Björn B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *SIES*, 2013. doi: 10.1109/SIES.2013.6601470.

[135] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS*, pages 73–83, 2001. doi: 10.1109/REAL.2001.990598. URL http://dx.doi.org/10.1109/REAL.2001.990598.

[136] O. Sinnen, L.A. Sousa, and F.-E. Sandnes. Toward a realistic task scheduling model. *Parallel and Distributed Systems, IEEE Transactions on*, 17(3):263–275, 2006.

[137] H. Ozaktas, et. al. Automatic WCET analysis of real-time parallel applications. In *WCET*, 2013. doi: 10.4230/OASIcs.WCET.2013.11.

[138] K. Lakshmanan, et. al. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*, 2009. doi: 10.1109/ECRTS.2009.33.

[139] M. Paolieri, E. Quinones, F. J. Cazorla, R. I. Davis, and M. Valero. Ia3: An interference aware allocation algorithm for multicore hard real-time systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 280–290, 2011.

[140] G. Giannopoulou, et. al. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT*, 2013. doi: 10.1109/EMSOFT.2013.6658595.

[141] G. Giannopoulou, et. al. Mapping mixed-criticality applications on multi-core architectures. In *DATE*, 2014. doi: 10.7873/DATE.2014.111.

[142] J.-E. Kim, et. al. Optimized scheduling of multi-ima partitions with exclusive region for synchronized real-time multi-core systems. In *DATE*, 2013.

[143] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[144] G. Bernat, A. Colin, and S.M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.

[145] F.J. Cazorla et al. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2012.

[146] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

[147] L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.

[148] S. Altmeyer and R.I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*, 2014.

[149] L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *Euromicro DSD*, 2014.

[150] J. Jalle et al. Bus designs for time-probabilistic multicore processors. In *DATE*, 2014.

[151] Enrico Mezzetti and Tullio Vardanega. On the industrial fitness of wcet analysis. In *WCET Workshop*, 2011.

[152] J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.

[153] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. *ECRTS*, 2011.

[154] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[155] F.J. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET workshop*, 2013.

[156] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

[157] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *ICQS*, 2005.

[158] J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

[159] Peter P. Puschner. Albrecht Kadlec, Raimund Kirner. Avoiding timing anomalies using code transformations. In *ISORC*, 2010.

[160] L. Kosmidis et al. Applying measurement-based probabilistic timing analysis to buffer resources. In *WCET workshop*, 2013.

[161] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

[162] L. Kosmidis et al. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.

[163] J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1): 03:1–03:13, 2014.

[164] J. Abella et al. Heart of gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.

[165] E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.

[166] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, and Mateo Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs.* IEEE Embedded Systems Letters, 2009.

[167] M. Slijepcevic, et al. Dtm: Degraded test mode for fault-aware probabilistic timing analysis. In *ECRTS*, 2013.

[168] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[169] M. Schoeberl et al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *NOCS*, 2012.

[170] M. Panic et al. On-chip ring network designs for hard-real time systems. In *RTNS*, 2013.

[171] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1), 2003.

[172] R. Bourgade et al. MBBA: a multi-bandwidth bus arbiter for hard real-time. In *EMC*, 2010.

[173] R. Bourgade et al. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *ETFA*, 2011.

[174] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 15–20, 2001. ISBN 1-58113-297-2.

[175] PROXIMA. *EU-FP7 Project: www.proxima-project.eu.*

[176] Sebastian Kehr. *Parallelization of Automotive Control Software.* Cuvillier Verlag Gottingen, Technische Universitat Ilmenau.

[177] P-SOCRATES. *EU-FP7 Project: www.p-socrates.eu.*

[178] BSC and Evidence. *Parallel Programming Models for Space Systems.* ESA contract 4000114391 15 NL Cbi GM.