PUCRS

FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO RUARO

SELF-ADAPTIVE QOS AT COMMUNICATION AND COMPUTATION LEVELS FOR MANY-CORE SYSTEM-ON-CHIP

Porto Alegre
2018

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SELF-ADAPTIVE QOS AT COMMUNICATION AND COMPUTATION LEVELS FOR MANY-CORE SYSTEM-ON-CHIP

MARCELO RUARO

Thesis submitted to the Pontifícia Universidade Católica do Rio Grande do Sul in partial fulfillment of the requirements for the degree of PhD in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2018

Ficha Catalográfica

R894s Ruaro, Marcelo

Self-adaptive QoS at communication and computation levels for many-core system-on-chip / Marcelo Ruaro . – 2018.
148 f.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. System-on-Chip. 2. Many-Core. 3. Network-on-Chip. 4. Quality-of-Service. 5. Self-adaptation. I. Moraes, Fernando Gehm. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecário responsável: Marcelo Votto Texeira CRB-10/1974

Marcelo Ruaro

Self-adaptive QoS at communication and computation levels for many-core system-on-chip

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 16th, 2018.

COMMITTEE MEMBERS:

Prof. Dr. Mateus Beck Rutzig (UFSM)

Prof. Dr. Rodolfo Jardim de Azevedo (UNICAMP)

Prof. Dr. Ney Laert Vilar Calazans (PPGCC/PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

AGRADECIMENTOS

Cada Tese possui uma longa história por trás. São em média quatro anos de muitas experiências acadêmicas e de vida. Durante esses anos de Doutorado aprendi muito. Conheci novas tecnologias, aprimorei meu pensamento cientìfico, li, escrevi e publiquei artigos. Além disso, tive o provilégio de fazer minha primeira viagem internacional para apresentar um artigo, e logo para uma cidade fantástica que sempre sonhava em visitar, que foi a cidade do Cairo no Egito (sim, conheci as pirâmides é claro). Além disso, tive a felicidade de ser contemplado com uma bolsa saundíche, o que me proporcionou a inesquecível experiência de viver por seis meses em Viena na Áustria.

Nos bastidores dessas experiências estão as pessoas importantes das nossas vidas. A minha família: a namorada e meu amor Fabiana, meus pais Silvio e Marinês, minha irmã Thaís e meu tio Nelson. Eles são a base de tudo. Não há palavras para descrever o constante apoio que me deram, me trazendo tranquilidade para enfrentar os momentos difíceis e sempre compartilhando comigo os momentos de fecilidade.

Uma Tese sem orientador não existe, e tive a sorte de ter um excelente comigo. O professor Moraes não fez só o papel de orientar, ele foi um acreditador, que me recebeu lá no início do mestrado, e acreditou no meu sonho. Lembro claramente nossa primeira conversa, ele pediu quais eram meus objetivos, e respondi: “quero ser Doutor em Ciência da Computação”. Suponho que tenha sido uma frase difícil para ele acreditar, visto minha total inexperiência no assunto e o longo caminho para ser percorrido. Aquele meu objetivo se tornou realidade, e sou muito grato a ele por me orientar e me aguentar nesses seis anos, me proporcionando diversas oportunidades que me fizeram crescer muito.

Um doutorado também é feito de cooperações, por isso, meu agradecimento especial aos colegas e amigos do GAPH: mestrandos, doutorandos e aos bolsistas ICs, que cooperaram em diversos artigos, discussões de pesquisa e em momentos de descontração.

Um ambiente acadêmico é fundamental, aqui deixo meu agradecimento notório a PUCRS, que possui no prédio 32 (Faculdade de Informática) um ambiente igual ou supeior a muitos laboratórios internacionais, com pessoas qualificadas e fornecendo equipamentos, conforto e segurança. O campus como um todo proporciona um bem-estar que incentiva trabalhos de melhor qualidade. Os funcionários, em especial os da secretaria do PPGCC, sempre atenciosos em resolver questões o mais rápido possível.

Por fim, agradeço a CAPES pelo suporte financeiro através da bolsa de doutorado pleno.

AUTOADAPTAÇÃO DE QOS NOS NÍVEIS DE COMUNICAÇÃO E COMPUTAÇÃO PARA SISTEMAS MULTI-NÚCLEOS INTRA-CHIP

RESUMO

Sistemas multi-núcleos intra-chip são o estado-da-arte em termos de poder computacional, alcançando de dúzias a milhares de elementos de processamentos (PE) em um único circuito integrado. Sistemas multi-núcleos de propósito geral assumem uma admissão dinâmica de aplicações, onde o conjunto de aplicações não é conhecido em tempo de projeto e as aplicações podem iniciar sua execução a qualquer momento. Algumas aplicações podem ter requisitos de tempo real, requisitando níveis de qualidade de serviço (QoS) do sistema. Devido ao alto grau de imprevisibilidade do uso dos recursos e o grande número de componentes para se gerenciar, propriedades autoadaptativas tornam-se fundamentais para dar suporte a QoS em tempo de execução. A literatura fornece diversas propostas de QoS autoadaptativo, focado em recursos de comunicação (ex., redes intra-chip), ou computação (ex., CPU). Contudo, para fornecer um suporte de QoS completo, é fundamental uma autoconsciência abrangente dos recursos do sistema, e assumir técnicas adaptativas que permitem agir em ambos os níveis de comunicação e computação para atender os requisitos das aplicações. Para suprir essas demandas, essa Tese propõe uma infraestrutura e técnicas de gerenciamento de QoS autoadaptativo, cobrindo ambos os níveis de computação e comunicação. No nível de computação, a infraestrutura para QoS consiste em um escalonador dinâmico de tarefas de tempo real e um protocolo de migração de tarefas de baixo custo. Estas técnicas fornecem QoS de computação, devido ao gerenciamento da utilização e alocação da CPU. A novidade do escalonador de tarefas é o suporte a requisitos de tempo real dinâmicos, o que gera mais flexibilidade para as tarefas em explorar a CPU de acordo com uma carga de trabalho variável. A novidade do protocolo de migração de tarefas é o baixo custo no tempo de execução comparado a trabalhos do estado-da-arte. No nível de comunicação, a técnica proposta é um chaveamento por circuito (CS) baseado em redes definidas por *software* (SDN). O paradigma SDN para NoCs é uma inovação desta Tese, e é alcançado através de uma arquitetura genérica de *software* e *hardware*. Para QoS de comunicação, SDN é usado para definir caminhos CS em tempo de execução. Essas infraestruturas de QoS são gerenciadas de uma forma integrada por um gerenciamento de QoS autoadaptativo, o qual segue o paradigma ODA (Observar, Decidir, Agir), implementando um laço fechado de adaptações em tempo de execução. O gerenciamento de QoS é autoconsciente dos recursos do sistema e das aplicações em execução, e pode decidir por adaptações no nível de computação ou comunicação, baseado em notificações das tarefas, monitoramento do ambiente, e monitoramento de atendimento de QoS. A autoadaptação decide reativamente assim como proativamente. Uma técnica de aprendizagem do perfil das aplicações é proposta para traçar o comportamento das tarefas de tempo real, possibilitando ações proativas. Resultados gerais mostram que o gerenciamento de QoS autoadaptativo proposto pode restaurar os níveis de QoS para as aplicações com um baixo custo no tempo de execução das aplicações. Uma avaliação abrangente, assumindo diversos *benchmarks* mostra que, mesmo sob diversas interferências de QoS nos níveis de computação e comunicação, o tempo de execução das aplicações é restaurado próximo ao cenário ótimo, como 99,5% das violações de deadlines mitigadas.

Palavras-Chave: Sistemas intra Chip; Multi-núcleos; Redes intra Chip; Qualidade de Serviço; Autoadaptação.

SELF-ADAPTIVE QOS AT COMMUNICATION AND COMPUTATION LEVELS FOR MANY-CORE SYSTEM-ON-CHIP

ABSTRACT

Many-core systems-on-chip are the state-of-the-art in processing power, reaching from a dozen to thousands of processing elements (PE) in a single integrated circuit. General purpose many-cores assume a dynamic application admission, where the application set is unknown at design-time and applications may start their execution at any moment, inducing interference between them. Some applications may have real-time constraints to fulfill, requiring levels of quality of service (QoS) from the system. Due to the high degree of resource's utilization unpredictability and the number of components to manage, self-adaptive properties become fundamental to support QoS at run-time. The literature provides several self-adaptive QoS proposals, targeting either communication (e.g., Network-on-Chip) or computation resources (e.g., CPU). However, to offer a complete QoS support, it is fundamental to provide a comprehensive self-awareness of the system's resources, assuming adaptive techniques enabling to act simultaneously at the communication and computation levels to meet the applications' constraints. To cope with these requirements, this Thesis proposes a self-adaptive QoS infrastructure and management techniques, covering both the computation and communication levels. At the computation level, the QoS-driven infrastructure comprises a dynamic real-time task scheduler and a low overhead task migration protocol. These techniques ensure computation QoS by managing the CPU utilization and allocation. The novelty of the task scheduler is the support for dynamic real-time constraints, which leverage more flexibility to tasks to explore the CPU according to a variable workload. The novelty of the task migration protocol is its low execution time overhead compared to the state-of-the-art. At the communication level, the proposed technique is a Circuit-Switching (CS) approach based on the Software Defined Networking (SDN) paradigm. The SDN paradigm for NoCs is an innovation of this Thesis and is achieved through a generic software and hardware architecture. For communication QoS, SDN is used to define CS paths at run-time. A self-adaptive QoS management following the ODA (Observe Decide Act) paradigm controls these QoS-driven infrastructures in an integrated way, implementing a closed loop for run-time adaptations. The QoS management is self-aware of the system and running applications and can decide to take adaptations at computation or communication levels based on the task feedbacks, environment monitoring, and QoS fulfillment monitoring. The self-adaptation decides reactively as well as proactively. An online application profile learning technique is proposed to trace the behavior of the RT tasks and enabling the proactive actions. Results show that the proposed self-adaptive QoS management can restore the QoS level for the applications with a low overhead over the applications execution time. A broad evaluation, using known benchmarks, shows that even under severe QoS disturbances at computation and communication levels, the execution time of the application is restored near to the optimal scenario, mitigating 99.5% of deadline misses.

Keywords: System-on-Chip; Many-Core; Network-on-Chip; Quality-of-Service; Self-adaptation.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AM	Adaptation Manager
API	Application Programming Interface
APL	Application Profile Learning
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction Set Processors
BE	Best Effort
CS	Circuit Switching
CTP	Communicating Task Pair
DB	Database
DEL	Data Extraction Layer
DMA	Direct Memory Access
DMNI	Direct Memory Network Interface
DTW	Dynamic Time Warping
DVFS	Dynamic Voltage and Frequency Scaling
EB	Elastic Buffer
EDF	Earliest Deadline First
FIFO	First in First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GPP	General Purpose Processor
GPU	Graphical Processor Unit
GS	Guaranteed Service
GT	Guaranteed Throughput
GUI	Graphical User Interface
IDMA	Identifier Division Multiplexing Access
IP	Intellectual Property
IRT	Input Reservation Table
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
LST	Least Slack Time
LUT	LookUp Table
NoC	Network-on-Chip
MJPEG	Motion-Joint Photographic Experts Group
MMR	Memory Mapped Register
MMU	Memory Management Unit
MoC	Model of Computation
MPI	Messaging Passing Interface
M_{PE}	Manager Processing Element
MPEG	Moving Picture Experts Group
MPN	Multiple Physical Network
MPSoC	Multi-Processor System on Chip
NI	Network Interface

ODA	Observe Decide Act
ORT	Output Reservation Table
OS	Operating System
PE	Processing Element
PP	Parallel Probe
PS	Packet Switching
QoS	Quality-of-Service
RR	Round-Robin
RT	Real Time
RTL	Register Transfer Level
RTOS	Real Time Operating System
RT-REM	Real Time Run-time Energy Management
S_C	Cluster Scheduler
SDN	Software-defined Networking
SDM	Spatial Division Multiplexing
S_L	Local Scheduler
SM_{PE}	System Manager Processing Element
SoC	System-on-Chip
SOI	Silicon on Insulator
S_{PE}	Slave Processing Element
SQL	Structured Query Language
STM	Slack Time Monitoring
TDM	Time Division Multiplexing
TLM	Transaction-level Modeling
VC	Virtual Channel
VCS	Virtual Circuit Switching
VHDL	VHSIC Hardware Description Language
WCET	Worst Case Execution Time

TABLE OF CONTENTS

1 INTRODUCTION

The high performance demand of the embedded market, combined with the technological advances in the semiconductor field, led to the development of many-core SoCs (System on Chip) [WOL08]. Many-cores contain a set of PEs (Processing Elements) interconnected through a NoC (Network on Chip) [HEM00][BEN02]. Many-cores are the state-of-the-art regarding computing power, due the high density of processing cores, and the support of parallel communications among them due the presence of the NoC as the communication infrastructure.

Current commercial many-cores can reach tens to hundreds of PEs [TIL13][CLE13][INT13]. Intel recently announced the Xeon Processor E5-2699 [INT16], a many-core with 22 physical cores. There is an estimation of ITRS [ITR15] that the number of PEs will reach more than 1,000 after 2025. This prediction is possible due to recent advances in the transistors' fabrication [IBM17a][LOU17] and new computation demands that push forward the edges of technology, like virtual reality, new smartphone applications, robotic, internet of things, and cloud computing. Therefore, it is noticeable the potential that many-core systems have, and its relevance for research.

NoC-based many-cores are a concept that emerged in the beginning of the last decade [HEM00][BEN02] and evolved through several contributions in academic and industrial works. The research field in many-cores is broad, due to its high complexity, multidisciplinary, and its tight design requirements. Verification and debugging, programming models, fault tolerance, security, power reduction, energy reduction, aging, and QoS (Quality of Service) are examples of current research areas in the NoC-based many-core SoCs.

General purpose many-core are systems where the application set is unknown at design-time, as in office and entertainment markets, smartphones, and other devices were applications can be installed and removed during the system's life. The system's developer does not know which application will be inserted into the system. Thus, the system must be smart to detect the application profile and to dynamically allocate its resources according to the applications' real-time constraints, providing QoS.

While to provide QoS for systems with few PEs (less than a dozen) seems to be solved in the research community, QoS for large-scale many-cores is an open research topic. This fact occurs due to challenges as management bottlenecks, communication and computation unpredictability, and process variability [SHA14]. Unpredictability is a relevant factor in adopting self-adaptive techniques. As the number of components increase, the number of events and transactions triggered by several protocols, management, monitoring, and data exchange also increases, making the process of predict them at design-time impracticable. In such systems, task mapping is only the first step to provide QoS, and a self-adaptive property is indispensable to increase the system's resilience regarding performance. Process variability can also affect QoS. For example, a core that is not working at the same frequency of other cores may induce deadline misses if a real-time task with tight constraints is allocated in this core. A self-adaptive technique can detect the QoS

violations and migrate the task to another PE. Such challenges require a *comprehensive* and *self-adaptive* QoS support to meet, at run-time, the different QoS levels of the user's applications. To be comprehensive, the QoS support must cover both the *communication* and *computation* resources. To be self-adaptive, the system itself must be able to detect QoS violations and act on the resources to restore the QoS levels. Related works have self-adaptive proposals covering only one level of QoS. The *goal* of this Thesis is to propose a self-adaptation QoS *infrastructure* and *management* for many-core systems, covering both the computation and communication levels. The proposed design can be divided into infrastructure and management scopes. The infrastructure scope concerns the design of techniques that enable the system's resources to be reconfigured at run-time. The management scope concerns high-level management, implemented in software, that is self-aware of the resources and applications and can trigger self-adaptations to ensure QoS. The next two paragraphs detail the *contributions* of this Thesis, covering the infrastructure scope at both computation and communication level.

At the computation level, the proposal consists of a low overhead task migration protocol and a dynamic real-time task scheduler. The task scheduler was designed based on the LST (Least Slack Time) algorithm and adopts a hierarchical organization, with the concepts of local schedulers, and a global scheduler. The local scheduler runs at each core allocating CPU resources according to the tasks' RT constraints (period, deadline, execution time). The global scheduler is included as part of the self-adaptive QoS management at computation level, which can decide to migrate a task at run-time if the local scheduler cannot ensure QoS. The main feature and novelty of the proposed task scheduler is to support at run-time the change of the task's RT constraints (period, deadline, execution time), providing high flexibility to the task by enabling fine-tuning of its RT workload. The task migration protocol is based on task recreation. It has a low overhead due to the choice of not copy the messages produced by the migrated task. The produced messages data remain in the original PE and are delivered on-demand. The task migration works to ensure computation QoS by acting in the CPU allocation control. The task scheduler works to ensure computation QoS by acting in the CPU utilization control.

At the communication level, the proposed design is a Circuit-Switching (CS) approach managed through the Software-Defined Networking (SDN) paradigm. CS is supported by configurable SDN routers organized in a Multiple Physical Network (MPN). The SDN routers have a small area footprint, implementing Elastic Buffers as storage input units, and supporting the SDN paradigm. The SDN paradigm removes the control logic from the router level (hardware), assigning it to a high-level management called CS-Controller (software). The CS-Controller abstracts to the rest of the system the CS management, it is responsible for searching CS paths and configuring the SDN routers, defining the network paths. The SDN-based CS works to ensure communication QoS by acting in the NoC allocation control.

Those techniques compose the QoS-driven infrastructure at computation and communication levels. The other contribution of this Thesis comprises a distributed self-adaptive QoS management.

The QoS management is implemented by high-level software algorithms and heuristics that run distributed at the manager PEs.

The QoS management is based on the ODA (observe, decide, act) paradigm. The *Observation* is supported by three classes of monitoring messages sent from the PEs running real-time tasks to manager PEs. The manager PE is self-aware about the status of applications and system's resources and executes heuristics that can decide to reconfigure QoS when necessary. The *Decisions* can be reactive as well as proactive, and fire orders to start the last phase of the ODA paradigm, the actuation. The *Actuation* is handled by the QoS-driven infrastructure and concerns the physical reconfiguration of the system's resources at both the communication, by setting up a new circuit-switching connection between two communicating tasks, and computation level, by migrating a task to another core and scheduling a task according it requested RT constraints.

To support proactive decisions, this Thesis proposes an online application profile learning technique, which continually learns about the communication and computation profile of each real-time task at run-time.

The overall results show that the proposed self-adaptive QoS management can restore the QoS level for the applications with a low overhead over the applications execution time. A set of experimental results are presented throughout this Thesis. A broad evaluation, assuming several known applications benchmarks (presented in Chapter 6) shows that even under several disturbances of QoS at both computation and communication levels, the execution time of the application is restored near to the optimal scenario, with 99.5% of deadline misses mitigated.

1.1 Hypothesis Development

The Author's research starting at the end of his Master course and finishing at the beginning of the Ph.D. explored QoS separately at communication [RUA15a] and computation [RUA14a] levels.

In Ruaro et al. [RUA14a] the Author proposed a run-time adaptation focusing on QoS computation, through task migration and a Round-Robin task scheduler with priorities. The key idea was to act individually at PE scope. A monitoring infrastructure (implemented in hardware and software) observes the latency and throughput of RT tasks. When a QoS violation is detected, the QoS management tries to remove BE tasks that can be running together the affected RT tasks. If there is no best-effort (BE) task to migrate, the heuristic searches for other RT tasks to migrate. The goal is to mitigate the interferences of the affected task, without migrating it directly, but acting indirectly, removing tasks sharing the same PE. If task migration is impossible due to the lack of available resources, the option is to increase the scheduling priority of the affected task. Results showed that the throughput of the affected task is restored after detecting interferences, with a small reaction time due to the QoS management. A limitation of this work is the absence of a real-time task scheduler, which can improve the CPU utilization and ensure more accurate QoS fulfillment, by handling specific RT constraints, like period, deadline, and execution time.

In Ruaro et al. [RUA15a] the Author proposed and evaluated a run-time adaptation focused on QoS of communication, through dynamic flow priorities and CS establishment. This research was based on the communication infrastructure proposed by [CAR11][CAR14], a network with duplicated physical channels, support to CS, and two levels of flow priorities (high and low). The network has two channels: channel 0 and channel 1. Channel 1 is used by low priority flows, and channel 0 is used by high priority flows, or by one CS connection, which allocates all channel 0 for a given CS path. Based on the same latency and throughput monitoring infrastructure of [RUA14a], the goal was to dynamically adapt the priority for a given flow according to the number of QoS violations. If this action is not enough to fulfill the communication QoS, the management tries to establish a CS connection. While this technique is effective to provide QoS in low congested scenarios, its main limitation is the low path diversity provided by the duplicated physical channel network. When a CS is established, all the communication flows using channel 0 are switched automatically to channel 1, increasing the congestion on channel 1.

The mentioned research motivated the Author to perform a broad literature review (presented in Chapters 4, 5, and 6) to study *self-adaptive* techniques covering QoS at both computation and communication levels. The related works pointed to a significant gap: "a many-core architecture that provides QoS in a comprehensive, scalable, and self-adaptive design, addressing the management and reconfiguration of resources at the computation and communication level." This gap was increasingly confirmed with the advances in the state-of-the-art review, leading to a central question of *how to design such kind of support*. This study enabled to propose the *Thesis* hypothesis:

> *A comprehensive, scalable, and self-adaptive QoS support can be achieved by proposing reconfigurable QoS techniques covering both computation and communication levels, and by a management that is based on the self-adaptive model [SHA09][DUT16a][DUT16b][ANZ17], the ODA paradigm [HOF13], and the distributed resource management [KRA12][CAS13].*

The *self-adaptive model* [SHA09][DUT16a][DUT16b][ANZ17] provides the necessary properties to implement subsystems to reach self-adaptiveness gradually. The *ODA paradigm* [HOF13] comprises the steps involved in the execution of the QoS management, presenting an organized structure to develop the components for self-adaptation comprehensively. The *distributed resource management* [KRA12][CAS13] provides scalability to the system by adopting a cluster-based organization.

Other related works contributed to the formulation of the Thesis hypothesis. At the communication level, the choice to adopts CS, implemented in a multiple physical network (MPN) design, was motivated by the previous conclusions in literature [EJA13][YOO13][LIU15], which compare MPN with SDM and TDM designs. Authors [EJA13][LIU15] compare MPN to SDM. With an equivalent bandwidth implementation, MPN increase area in $O(n)$, while SDM increases $O(n^2)$. The path delay increases with the number of sub-channels (SDM) in $O(n)$, while stays unchangeable for

MPN. Yoon et al. [YOO13] evaluate virtual channels (TDM) with MPN, concluding that MPNs have a simpler implementation, lower area, and simpler floorplan.

The high flexibility that the SDN paradigm shown for computer networks [KIR13] motivated the proposition of an SDN-based organization at the SoC context, and its use to manage CS connections. Thus, the proposition of an on-chip CS-Controller, as a module that abstracts the CS connection from other system components, enables to create a generic and modular CS support, fitted to the high-level management required in self-adaptive systems.

At the computation level, the proposition of task migration and the task scheduler was motivated by gaps found during the state-of-the-art review. Regarding the task scheduler, the main observed gap was the lack of support for the run-time modification of the RT constraints, instead of using static constraints during the whole application execution. This dynamic workload behavior is present in, e.g., multimedia applications [JAV14]. Regarding task migration, the main challenge was to propose a low overhead protocol, since the QoS management could require many task migrations simultaneously, and related works have limitations in task migration techniques for distributed memory systems.

1.2 Thesis Goals

The general goal of this Thesis is to propose a *self-adaptive QoS support for a many-core system that covers QoS at computation and communication levels*. This general goal is divided into specific goals:

1. A low-overhead task migration protocol for distributed memory many-cores;
2. A dynamic RT task scheduler;
3. An SDN-based Circuit-Switching;
4. A high-level self-adaptive QoS management at computation and communication levels

To enable the high-level self-adaptive QoS management, an infrastructure needs to be addressed, covering computation and communication. Goal 1 concerns the proposition and implementation of the infrastructure to support task migration. Goal 2 concerns the proposition and implementation of the infrastructure to support an RT task scheduler. Goals 1 and 2 cover the QoS-driven techniques at the computational level. Goal 3 concerns the proposition and implementation to support CS, covering QoS-driven technique at the communication level. Finally, goal 4 is related to the proposition of the self-adaptive QoS management over these proposals.

1.3 Thesis Contributions

The main contributions of this Thesis are the following items:

1. Low overhead task migration protocol [RUA17b] (Chapter 4);
2. Dynamic RT task scheduler [RUA15b][RUA16a] (Chapter 4);
3. SDN-based CS management [RUA17a][RUA18] (Chapter 5);
4. Self-adaptive QoS management at computation and communication level (Chapter 6).

Side contributions arising from the work required to achieve the specific goals include:

5. DMNI: a specialized network interface for many-cores [RUA16b] (Chapter 2);
6. A many-core debugging framework [RUA14b][RUA16c] (Chapter 2).

The side contributions are not directly linked to QoS, but provide mechanisms to improve the general system performance, which is the case of the Direct Memory Network Interface (DMNI), and supported the Author throughout the validation processes to debug the many-core platform, which is the case of the debugging framework.

1.4 Thesis Organization

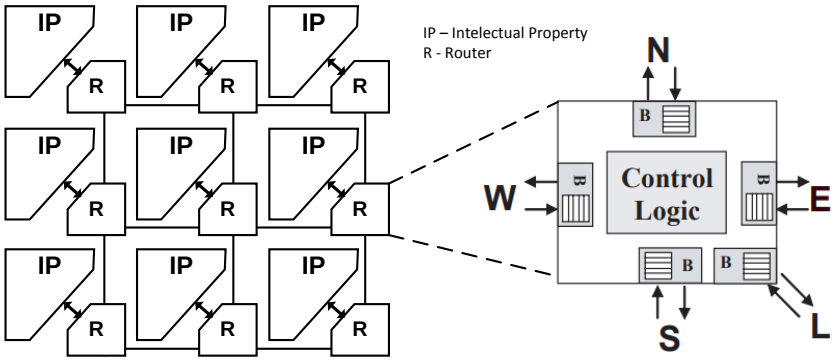
This Thesis proposal is organized as follows. Chapter 2 presents assumptions and general concepts in such a way to make the Thesis self-contained. Chapters 3 details architectural features of the reference many-core system. Additionally, this Chapter presents the contribution DMNI and the debugging framework, which are part of the baseline many-core system. Chapter 4 presents contributions of Thesis at the context of the QoS infrastructure at computation level. Chapters 5 presents contributions of Thesis at the context of the QoS infrastructure at communication level. Chapter 6 presents contributions comprising the self-adaptive QoS management. Chapter 4, 5 and 6 have a similar organization, with a state-of-the-art, description of QoS techniques, and conclusions. Chapter 7 presents the general conclusion of the Thesis and directions for future work. Appendix A presents the Author's publications during the Ph.D. period.

2 ASSUMPTIONS AND GENERAL CONCEPTS

This Chapter presents assumptions and general concepts used in this Thesis. Other specific concepts are detailed in the next Chapters.

2.1 Network-on-Chip

A Network-on-Chip (NoC) [HEM00][BEN02] is an on-chip communication infrastructure to interconnect Intellectual Property (IPs) components – Figure 1(a), decoupling the computation from communication. The NoC structure contains routers and wires. Routers have the function to implement the network control logic, as depicted in Figure 1(b), which defines the path for each exchanged packet between a source and target PE, routing a packet from an input port to an output port. Wires have the function to interconnect routers, and to connect each router with its local IP.



(a) Many-Core interconnected by a NoC (b) NoC's router overview

Figure 1 – (a) Network-on-Chip as interconnection infrastructure for a Many-core SoC. (b) a 5-port router overview [MOR04].

Due to the benefits of NoCs compared to bus-based connections, as parallelism and scalability, NoCs has been evolved from university prototypes to full commercial synthesis flows [MIC17]. For example, IBM's *TrueNorth* neuromorphic chip has 5.4-billion transistors, where 4,096 cores are connected through a NoC, aiming the design of a brain-inspired computer [IBM17b].

The main features of a NoC are the topology, routing algorithm, flow control, and switching mode [MOR04]. Regarding topology, currently stand-out the 2D-mesh topology, due to its easy implementation and scalability. Regarding the routing algorithm, turn-based algorithms (as XY, West-first, Negative-first) are mostly used, due to its simplicity, low area footprint, minimum path length, and deadlock-free features. A common example of flow control method is the synchronous credit-based, which provides lower latency compared to handshake methods. The most used switching mode is the wormhole packet-switching (PS) since it requires the smaller amount of space in buffers and avoids the reservation of network resources for a large amount of time.

This Thesis adopts the Hermes [MOR04] PS router, with a 2D mesh topology, XY routing algorithm, credit-based flow control and wormhole packet-switching.

2.2 Many-Core

Many-core [WOL08] is a system-on-chip composed of several PEs, interconnected by a NoC. The term many-core is related to the presence of several PE units in a single chip. Another definition for many-core is MPSoC (Multiprocessor System-on-chip), the switching from the label "multi" to "many" cores/processors is due to the growing number of PEs on the same chip. However, this nomenclature is not well defined in the literature, because works mix the terms MPSoC, multi-core, and many-core. This work adopts the many-core term, assuming systems from a dozen to hundreds of cores.

Many-cores can be classified into two large groups: heterogeneous and homogeneous.

- *Homogeneous* many-cores: divided into asymmetric and symmetric architectures. In s*ymmetric* many-cores, all PEs have the same architecture and are replicated over the system. Example of *asymmetric homogeneous* many-core is the big.LITTLE architecture, developed by ARM [ARM17][GUI17]. In this architecture, the system has two PE types, "LITTLE" PEs are designed for power efficiency while "big" PEs are designed for performance. The workload distribution between "LITLE" and "big" PEs is managed at run-time according to the applications' profile and the system budgets.
- *Heterogeneous* many-cores: different PEs architectures are assumed, as GPPs, GPUs, DSPs, and ASIPs.

Heterogeneous many-cores target specific applications, because their PEs (or some of them) are specialized to a given application profile. On the other hand, homogeneous many-cores (symmetric or asymmetric) target general-purpose applications, such as multimedia and office applications' classes, and use the replication of processing components to increase its processing power. Homogeneous many-cores have advantages over heterogeneous due to the possibility to achieve better load balancing, fault tolerance, time-to-market (due to the redundancy of components), and the support of easy task migration protocols. Currently, the industry already designs homogeneous many-core from a dozen to thousands of cores. Some examples are the 72 PEs chip from Tilera [TIL13], and *KiloCore* composed of 1000 PEs [BOH17].

While this Thesis adopts a symmetric homogeneous many-core as the baseline architecture, this does not limit (after some modifications) the applicability of the techniques herein proposed for heterogeneous or asymmetric many-cores. The homogeneous many-core is only used as a proof-of-concept.

2.2.1 Computation and Communication Resources

Resources of a many-core system can be divided into computation and communication parts. Figure 2 presents an example of a PE architecture, its components, and the division between the computation and communication resources. Computation resources are related to data processing, collecting and manipulating data inputs to create a meaningful information that is available as data

output. Distributed memory many-core systems have their computational resources located inside the PE instance, composed basically by a CPU and a local memory. A shared memory system has its computational resources decoupled from the PE instance. They assume an off-chip memory or a memory instantiated as on-chip IPs. Hybrid memory systems assume a shared memory and a small local memory (cache memory) within the PE, which is used to speed-up instruction access and to reduce the communication load [MAD16]. Besides the CPU and memory, a Direct Memory Access (DMA) or Direct Memory Network Interface (DMNI) [RUA16b] also can be adopted to improve memory transactions.

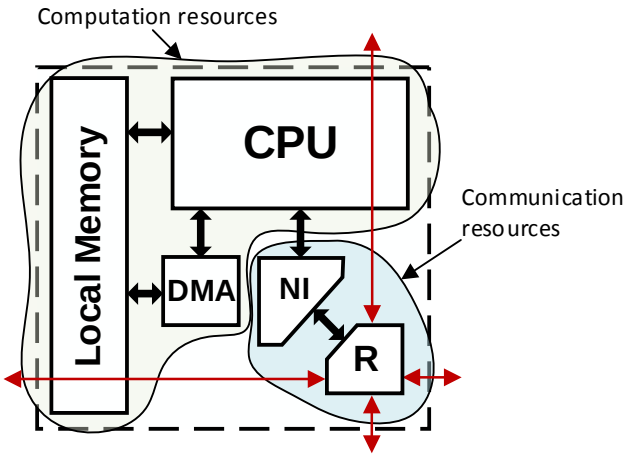


Figure 2 – Example of the components of a many-core Processing Element (PE), divided into computation resources (local memory, CPU, and DMA) and communication resources (router, wires, and NI).

Communication resources are concerned in transmitting the inputs and outputs of the data processing system. The communication resources correspond to the NoC resources, which can be assisted by a Network Interface (NI) to make the interface between the PE and the NoC. The NI decouples computation from communication, receiving data from local memory and converting to the NoC packet format, and converting NoC packets in inputs for the computational resource.

This Thesis proposes QoS techniques that reconfigure computation and communication resources. The computational resources comprise the CPU utilization (RT task scheduler) and CPU allocation (task migration) at run-time. At the communication level, this Thesis proposes multiples physical networks-on-chip (MPN), and the run-time management of the MPN resources.

2.3 Real-Time Applications

Real-time applications [LIU00] have timing constraints to finishes its jobs. A job can be defined as a segment of computation and/or communication that applications need to execute, for example, a loop iteration, the resolution of a mathematical equation, sorting an array, etc. Real-time applications can be characterized according to their job execution frequency: as *periodic*, *aperiodic*, *sporadic*. Periodic applications request to execute their jobs continually in fixed periods of time. Aperiodic applications request to execute their jobs continually in unfixed periods, but with a minimal interval of job release. Sporadic applications do not have a continual execution of their jobs,

they can request to execute they at any time, and there is not a guaranteed that the application will request to execute again, i.e., there is not a minimum time between jobs release.

Another classification of RT applications concerns *deadlines*: hard RT and soft RT. A deadline is a temporal constraint that indicates when the system must finish the execution of the application job. Hard RT applications do not tolerate any deadline miss, they are safe-critical applications and are commonly implemented in dedicated parts of the system. An example of hard RT application is a self-driving car algorithm, which can not miss deadlines to take its decisions, with the consequence of injury a human life. On the other hand, soft RT application tolerates some deadline misses. The number of tolerated deadline misses is not a constant, it depends on the application characteristics. Nevertheless, the system must work to avoid all deadline miss. Business and multimedia applications are well known applications from this class. Consider an audio-video decoder running on a smartphone. The system must work to deliver 30 frames per second to the end user, but if the frame rate drops for a brief moment to 29 or 28, this may not have critical impact to the end user.

Applications that do not have any deadline to finish their jobs are classified as Best Effort (BE). The system explores the idle time of the CPU or the slack time of RT application to execute those applications.

This Thesis assumes BE and soft RT applications with periodic, aperiodic and sporadic job releases. The support for the three job classes is due to the possibility of tasks to change its period, deadline and execution time at run-time.

2.4 Quality of Service for Performance

Quality of service (QoS) for performance is a system capability to dedicate some of its resource to fulfill the RT constraints of the applications [CAR11].

The term QoS originated from telecommunication networks, where QoS constraints are related to the communication throughput and latency of interconnections. For many-core systems, the concepts of QoS assumes a broader definition, covering not only communication but also computation resources. The communication QoS is focused in the allocation of communication resources (routers and wires), according to the application communication characteristics, as latency, throughput, jitter, and the application communication graph between its tasks. Flow priorities and circuit-switching are the most adopted techniques to perform this support. The computation QoS for many-cores is focused on allocating CPU resources. This support is mainly enabled by RT task schedulers, task mapping, and task migration techniques. Chapter 4 and 5 detail the most adopted techniques in literature to provide QoS at computation and communication levels.

Beside QoS for performance, there are also other QoS targets: examples are the QoS support against system failures (fault tolerance) and data integrity (security). This Thesis focuses on QoS for performance, addressing QoS at the communication and computation levels.

2.5 Self-awareness and Self-adaptation

Challenges that the design of the next many-core generations must address include the presence of hundred to thousand cores in a single chip, tightly application-specific QoS constraints, physical budgets for power and temperature, and low energy consumption. Design-time phases are the first steps to deal with such challenges. A large-scale system, due to its high level of unpredictability about the resources usage, requires autonomy to be aware of its resources and be the actor of run-time adaptation to fulfill constraints. Thus, the system must be self-aware, meaning that it must have the autonomy to observe the status of its resources and running application, gathering the necessary information to start a reconfiguration when necessary. While in an adaptive system the adaptation process may be triggered from the outside, e.g., by an application management layer or a human operator, a self-adaptive system identifies the triggering condition and initiates the adaptation process.

Recent works address these challenges, by investigating and proposing modular self-aware infrastructures and architectures for many-cores. The main goal is to provide a unified self-aware design phase, which is generic, avoiding the necessity to design a new self-awareness system for each new project [DUT17]. Models based on a closed loop have been addressed, as the one presented in Figure 3 [SAR15][DUT16b]. This kind of self-awareness model implements the ODA (Observe, Decide, Act) paradigm, enabling the system to be continually working toward a better understating of its behavior. Different layers implement independent self-awareness subsystems [DUT16b], with submodules for *observation* (by virtual and physical sensors), *decision* (adaptation policies composed of heuristics aware of design and applications constraints) and *actuation* (infrastructure support to reconfigure the hardware and the software level). A dedicated network can be assumed to enable fast communication of management packets, avoiding interference in the application flows [DUT16b]. One of the main advantages of this model is that each layer of the many-core can be made self-aware and self-adaptive, as detailed in Figure 3.

A recent survey conducted by Professor Axel Jantsch (Technological University of Vienna) and co-Authors provides a comprehensive overview related to the research status and challenges in self-awareness for SoCs [JAN17b]. The Authors present a precise definition and review of terms such as self-awareness, self-adaptation and other definitions required to construct such functionalities. Figure 4 reviews the pyramidal organization related to *self-x* functionalities to achieve self-adaptiveness [SHA09]. Self-optimization is related to the ability to tuning system resources to satisfy the requirements of different applications. Self-configuration is the capability related to the infrastructure support that enables the system to be reconfigured at run-time. At the top of the pyramid is self-adaptiveness. This functionality is related to the system ability to decide when and which resource to adapt to fulfill a given constraint. Other *self-x* properties such as healing and protection can be supported, covering fault-tolerance and data integrity, respectively. Note that for the system to be self-adaptive it needs to be self-aware.

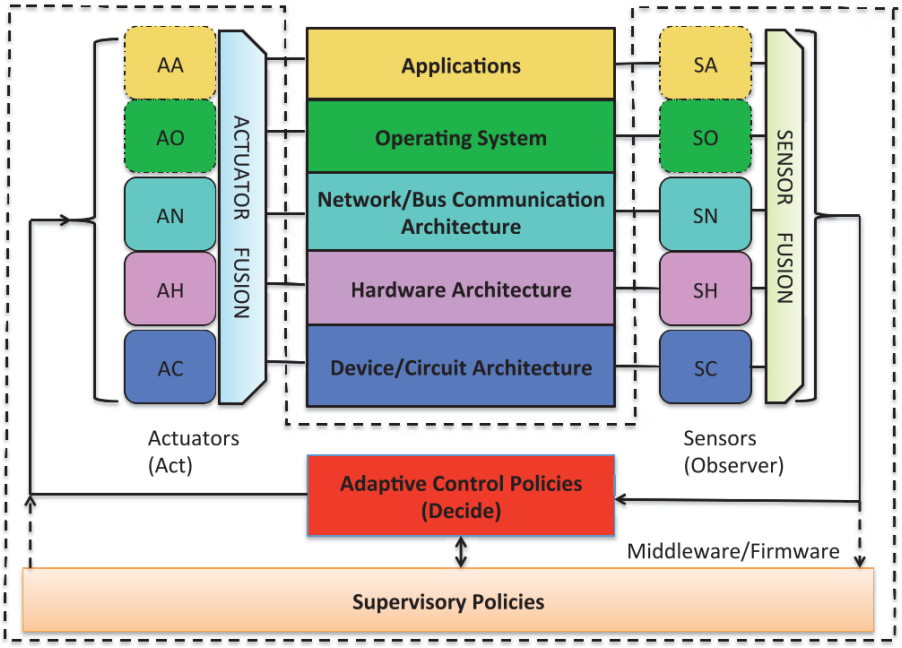


Figure 3 - Cross-layer virtual sensing and actuation at different layers of Cyber-Physical SoC (an example of many-core) [SAR15][DUT16b].

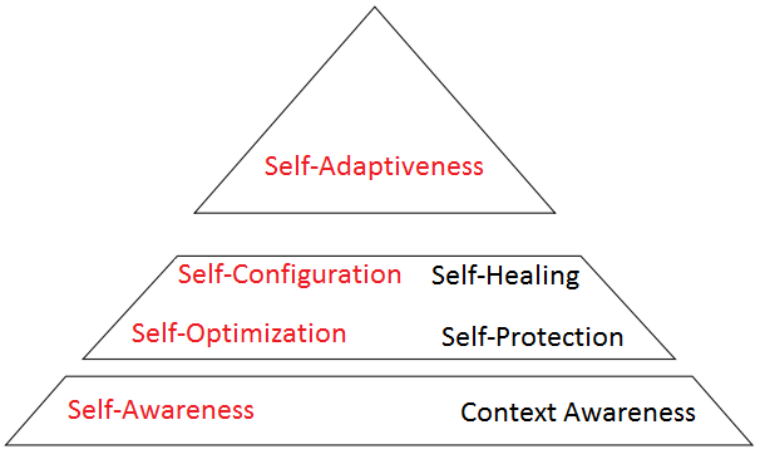


Figure 4 – Self-adaptiveness hierarchical organization [SHA09].

This Thesis is based on the *self-x* concepts to propose the self-adaptation QoS support at communication and computation levels, detailed in Chapter 6.

2.6 ODA Paradigm

The Observe-Decide-Act (ODA) is a method to implement self-adaptation, proposed in Hoffman et al. [HOF13] (SEEC Framework) at the SoC context, which consists in implement a closed loop (Figure 5) of observation, decision, and actuation, enabling the system to be self-aware as well as self-adaptive. **Observation** is implemented by information provided by monitors/sensors and sporadic feedbacks that are sent to a high-level management (software). The application developer defines goals/constraints to the application at design-time, and that must be met at run-time. The management module reads these goals comparing them to the received monitoring information.

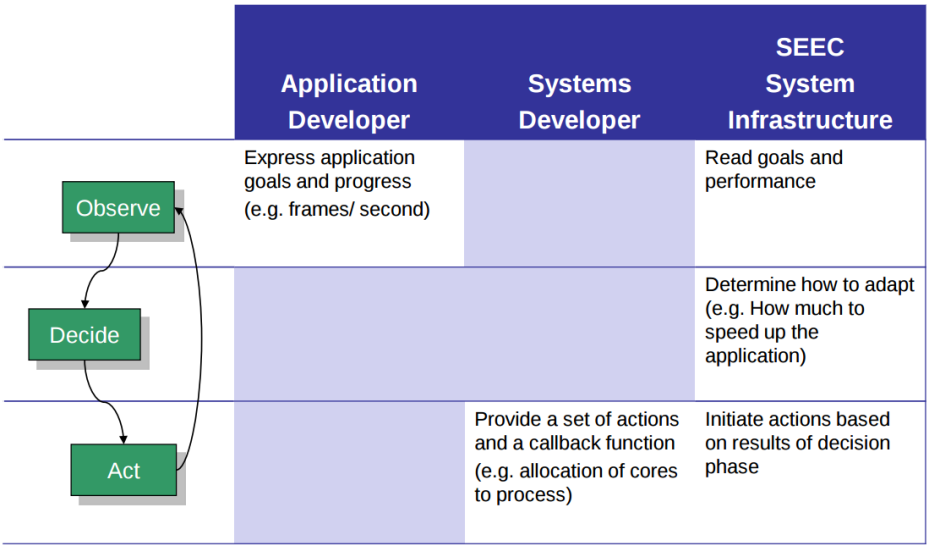


Figure 5 – ODA loop providing self-awareness and enabling self-adaptation [HOF13].

Decision is implemented by a high-level management module, which is aware on the situation and can determine how to adapt, for example, if a given task is losing QoS, deciding on which resource to act to restore the QoS level. **Actuation** is implemented by a QoS-driven infrastructure (such as real-time task scheduler, task migration protocols, circuit-switching designs), and by systemic protocols, which work to reconfigure the computation or/and communication resources at run-time. The actuation is triggered in the decision phase.

The ODA paradigm enables to implement a comprehensive self-adaptive management because it divides the subsystems by their roles, providing scalability for inserting future features, and also helps the validation of the self-adaptive implementation.

2.7 Distributed Resource Management

The high number of resources to manage can quickly overload a single manager of a many-core. A scalable management design method consists in dividing the management of the system in regions. Each manager handles a cluster of PEs [CAS13][GOT16]. This organization also enables to implement hierarchical management, as proposed by Faruque et al. [FAR10] and depicted in Figure 6. A low-level management is applied at the PE level, such as task scheduling and monitoring. The monitored information is sent to cluster managers, which concentrate these information enabling to have a holistic view the cluster. A third level can be inserted by assuming a global manager, which can receive high-level information from cluster managers, providing a systemic view. The global manager takes coarse grain decisions, like choosing which is the best cluster to receive a new application or managing reclustering protocols that change the shape of the cluster at run-time.

This Thesis assumes a distributed resource management implemented on dedicated managers. The management is divided into three levels, as proposed in some works of literature [FAR10][CAS13][GOT16]. Chapter 3 provides more details related to this infrastructure.

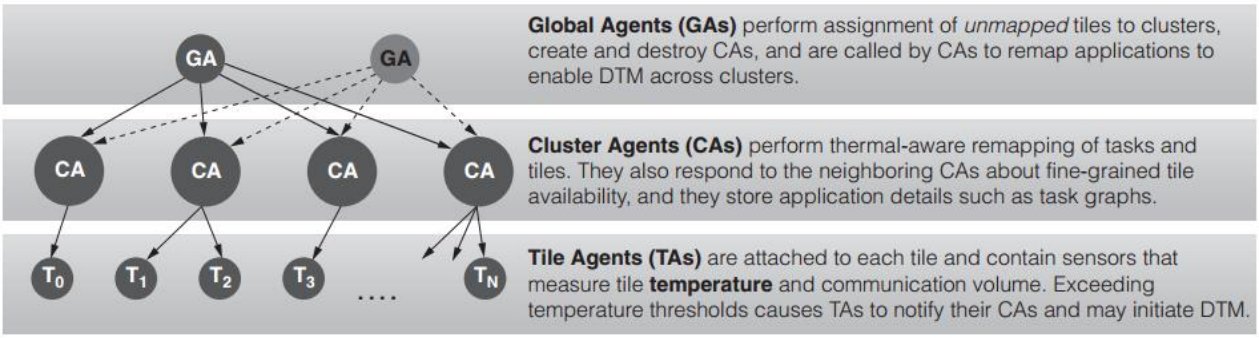


Figure 6 – Overview of distributed resource management proposed by [FAR10] to dynamically control the temperature on many-cores.

2.8 Monitoring

Monitoring means observation, i.e., the capability of the system to observe a given behavior and to enable self-adaptation. The literature presents many monitoring proposals [KOR13]. Monitoring for thermal, power and energy management are the most explored monitoring techniques in the literature. These techniques guide managers to balance the on-chip temperature or voltage by monitoring such parameters directly [RAN15][ZHA11][WAN10], or indirectly, through the monitoring of the communication frequency [FAR10], or memory accesses [GRA13]. Monitoring for fault-tolerance and reliability has also been proposed, measuring parameters as soft error failures, and wear-out of devices [ZHA11][LIU14a].

There are some proposals specific for QoS monitoring. In general, the QoS-driven monitoring targets two performance parameters: (*i*) performance constraints; (*ii*) system resource availability.

QoS monitoring can be applied at the task level in hardware [STA11] or software [SAI10]. Monitoring uses as constraints data obtained by a profiling step at design-time. At run-time, the monitored performance is compared to the constraints defined at design-time. In this context, a common technique is to observe the communication buffers. Stan et al. [STA11] monitor the communication buffers in the memory space. Matos et al. [MAT10a] monitor the communication buffers to apply buffer resizing. Faruque et al. [FAR12] apply buffer resizing and includes an adaptive route allocation algorithm. Communication buffer monitors can be implemented exclusively into the NoC [MAT10a], or into the NI [FAR12][KOR12][MOT12].

Kornaros et al. [KOR12] propose an NI monitor that can be software programmed to serve QoS. The proposal describes a *cluster manager monitor*, a hardware unit that concentrates the data from the low-level NI monitors, and generates interrupts to a generic management system when a predefined threshold is violated. Motruk et al. [MOT12] propose an NI monitor to collect information about the usage of the system shared resources. Monitoring is used to achieve composability in a shared-memory many-core by using temporal and spatial division.

There are also proposals focusing on generic monitoring infrastructures [AMB13] [FAT11][FIO10][KRA12]. The main strategy to achieve monitoring scalability, low area overhead, and low intrusion is to adopt a hierarchical monitoring organization [FAT11][KRA12]. Monitors can be

divided into hierarchical levels, each one handling different granularities of the monitored data. Such approaches include hybrid monitors (mixed hardware/software implementation) because they can combine the non-intrusiveness of hardware monitors (at lower levels), with the system's view and flexibility of the software monitors (at higher levels).

A representative work is the proposal of Kramer et al. [KRA12] that addresses a scalable monitoring infrastructure for many-cores. The Authors describe a hybrid monitoring organized in three levels, as presented in Figure 7(a). At the lower level, several Associative Counters Arrays (ACAs) can be attached to different hardware or software components of a PE - Figure 7(b). The ACA extracts system's performance parameters, as communication rate, buffer occupancy, and the number of instructions. ACAs send the monitored data to a Lower Level Monitoring unit (LLM). An LLM acts as an interface between the ACAs and high-level monitoring.

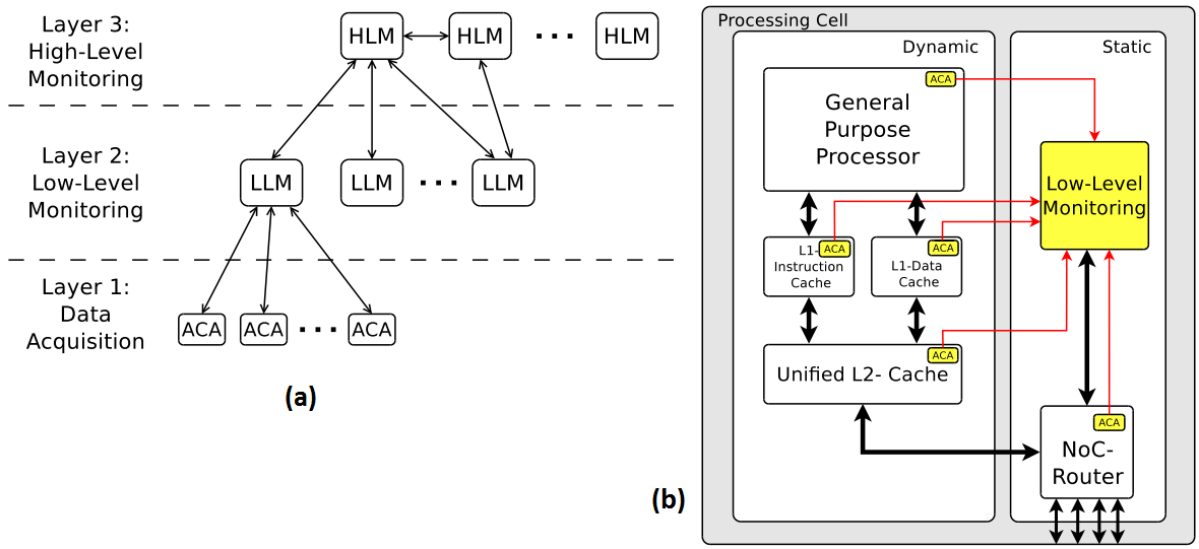


Figure 7 – Kramer et al. [KRA12] hierarchical and hybrid monitoring.

An advantage of hierarchical monitoring over centralized approaches is that its organization is easily settled with a cluster-based resource management [GOR13][GUA10][FAR10]. The lower level monitor units can be attached to PEs, medium levels monitors can be attached to cluster managers, and high-level monitors can be attached to a system many-core manager.

The key issue to support run-time monitoring implementations is the ability to program monitors dynamically. Such feature can be easily achieved in software monitoring [SAI10] or can use memory-mapped registers in hardware proposals [KOR12][AMB13][FIO10][STA11].

Another recent monitoring metric for many-cores is the task slack time monitoring. Slack time monitoring (STM) is an important metric to enable QoS adaptation focused on computation. It can be used together with an RT scheduler to observe the task, or CPU slack time (idle time monitoring). STM at the task level computes the slack time based on an earlier task execution completion. The literature shows that such approaches can be applied to energy management through DVFS [LI13][KOT10], which can be combined with task migration [SIN13a]. Paterna et al. [PAT12] propose an energy-efficient task allocation method considering process variation. That work mentions STM

for RT tasks based on a frame rate monitoring of multimedia applications, however, without detailing the implementation.

This Thesis adopts monitoring methods targeting self-adaptive techniques: deadline miss monitoring, latency miss monitoring, CPU slack-time monitoring, and application profile monitoring. While the monitoring implementations are not original contributions, they are fundamental to enable the self-adaptive techniques herein proposed.

2.9 Final remarks

The list presented below summarizes the assumptions adopted in the current Thesis:

- *NOC*: Hermes PS router, with a 2D mesh topology, XY routing algorithm, credit-based flow control and wormhole packet-switching.
- *Many-core*: symmetric homogeneous many-core.
- *Application types*: BE, and soft-RT applications with periodic, aperiodic and sporadic job release.
- *QoS*: target applications' performance.
- *Self-adaptation*: paradigm adopted to meet QoS at communication and computation levels.
- *ODA*: method to implement a comprehensive self-adaptive management.
- *Distributed Resource Management*: 3-layer structure to ensure scalability for the proposed methods.
- *Monitoring*: deadline miss monitoring, latency miss monitoring, CPU slack-time monitoring, and application profile monitoring.

3 REFERENCE MANY-CORE ARCHITECTURE

The *HeMPS* platform (http://www.inf.pucrs.br/hemps/) is the reference many-core architecture adopted in this Thesis. *HeMPS* is an open-source many-core architecture framework developed by the GAPH research group, used to implement and evaluate proposals for many-cores. The Author of this Thesis contributed significantly to the development of the last versions of *HeMPS* (since version 7.0). Figure 8(a) overviews the *HeMPS* architecture.

The application repository is an off-chip memory used to store the applications that will enter in the system at run-time. The system has a set of homogeneous PEs, organized in clusters. Each cluster has a Manager PE (M_{PE}), dedicated to management purposes, as task mapping and application admission control. The M_{PE} can receive other algorithms to gather monitoring information and to manage the system at run-time. This Thesis implements the self-adaptive algorithms in the $M_{PE}s$ kernels.

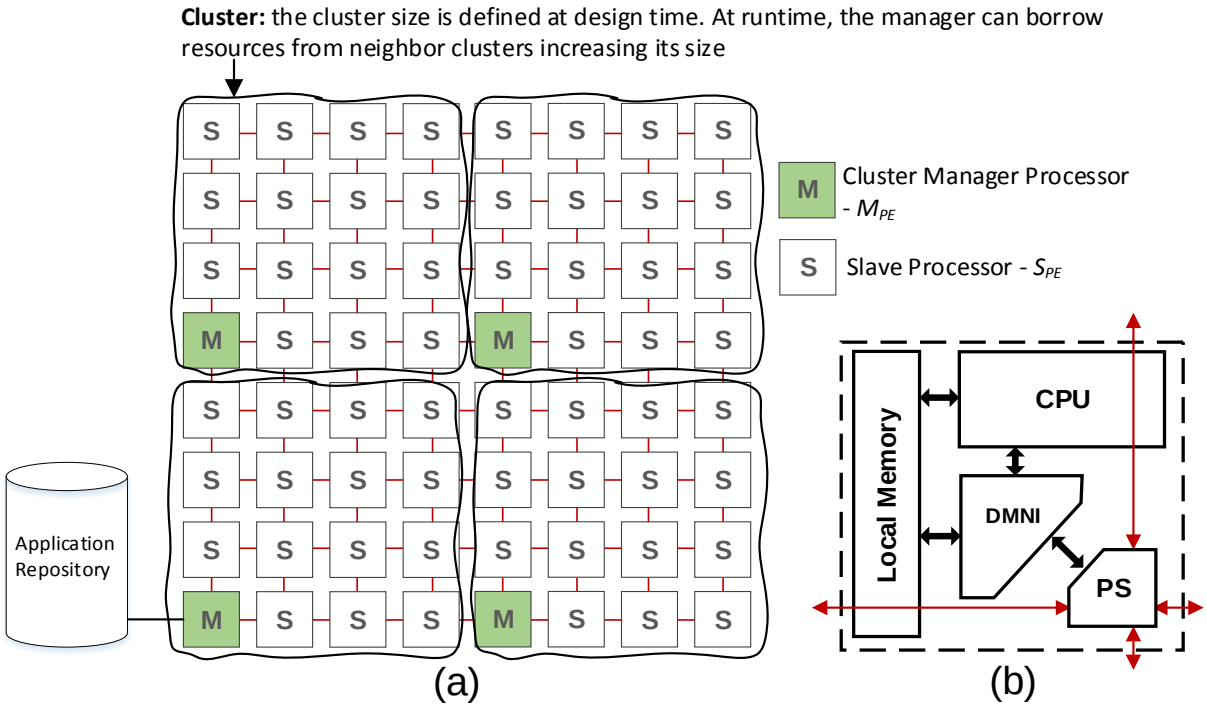


Figure 8 – Baseline many-core architecture. (a) system architecture [RUA17a]; (b) PE architecture.

Slave PEs (S_{PE}), execute user's tasks, with inter-task communication and multitasking support. The differentiation between M_{PE} and S_{PE} occurs at the software level, with each one having a different *kernel* (small operating system). Figure 8(b) overviews the PE's hardware architecture. All PEs have a dual-port scratchpad memory, a CPU, and a DMNI (Direct Memory Network Interface) connected to PS router (an original Hermes router [MOR04]). The DMNI merges the functionalities of the NI with a DMA module, reducing control logic and buffering, and creating a direct path between the NoC and memory. The system can also support other S_{PE} architectures enabling heterogeneity. This Thesis assumes a homogeneous symmetric architecture, as stated in the previous Chapter.

This Chapter contains five sections:

- Section 3.1 – describes the applications' model.
- Section 3.2 – reviews the main features of the operating system running in the processing elements (*kernel*).
- Section 3.3 – explains the communication method between tasks, based on message passing.
- Section 3.4 – presents one of the side contributions of this Thesis, the *Direct Memory Network Interface* (DMNI) [RUA14b].
- Section 3.5 – describes the modular verification, corresponding to the second side contribution of this Thesis [RUA16b].

3.1 Applications' Model

Figure 9 presents an application model example. An application *A* is modeled as task graph: $A=\{T, D\}$, where $T=\{t_1, t_2, ..., t_n\}$ is the task set. The set *D* represents the application descriptor, which contains the communicating task pairs (*CTP*). The *CTP*s of the application presented in Figure 9 are: $t_A \rightarrow t_B$, $t_A \rightarrow t_C$, $t_B \rightarrow t_D$, $t_C \rightarrow t_D$. A given task may be in 3 states: (*i*) *ready*, the task is ready to be executed; (*ii*) *waiting*, the task is blocked waiting for a message delivery; (*iii*) *running*, the task is running, using the CPU and changing its dynamic memory sections.

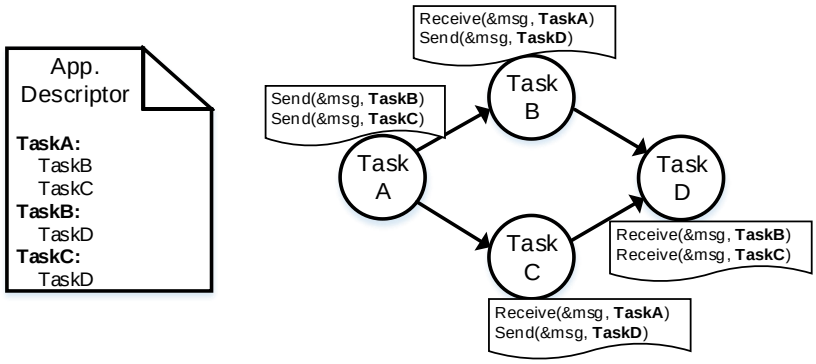


Figure 9 - Application Model Example [RUA17a].

The reference many-core system has a task scheduler supporting BE applications only. One of the contributions of this Thesis, presented in Chapter 4, provides the RT application support by proposing an original RT task scheduler.

3.2 Kernel (Distributed Operating System)

Figure 10 presents the kernel hierarchy of the *HEMPS*, implementing a distributed management. As previously mentioned, the system has three types of kernels: S_{PE}s and M_{PE}s. There is also one specific version of M_{PE}, called SM_{PE} (S stands from System). The SM_{PE} implements exactly the same functionalities of the M_{PE}, with the addition of other functions that perform a system management, controlling the interface with the application repository and a heuristic to assign new applications to the clusters [CAS13].

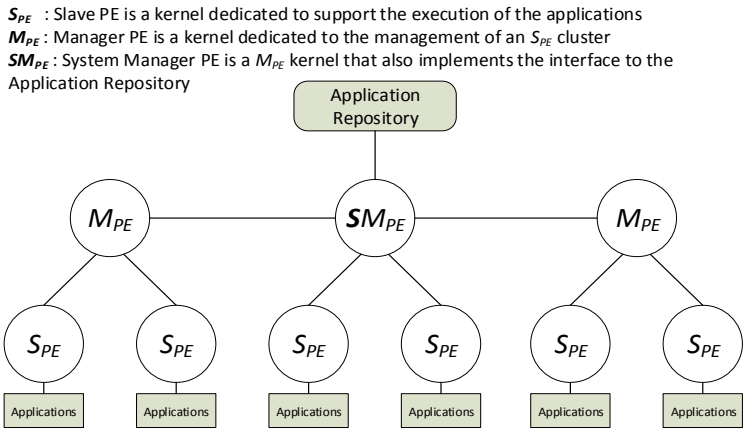


Figure 10 – Kernel Hierarchy of *HeMPS*.

Clusters have a dynamic size, i.e., clusters can change their size at run-time if the number of applications' tasks is greater than the number of available cluster's resources (local memory page of a given PE). This capability is supported by a reclustering protocol [CAS13] implemented between M_{PE}s that allows to borrow resources from a cluster to another.

3.2.1 Application's Admission Management

The kernels (S_{PE}, M_{PE}, SM_{PE}) implement an application admission protocol presented in Figure 11. When an application request to execute in the system, the application repository interrupts the SM_{PE}. The SM_{PE} handles the interruption retrieving the application description D and executes a cluster mapping algorithm, which selects the appropriate cluster to receive the application (based on the cluster's occupation). The SM_{PE} sends the application description D to the M_{PE} of the selected cluster. The M_{PE} receives the application description, maps the tasks into S_{PE}s and requests to the SM_{PE} the tasks object code. The SM_{PE} handles such request configuring the application repository to transfer the object tasks code to the S_{PE}s. When an S_{PE} receives a task code, it sends a message to its M_{PE} informing the successful load of the task code. The M_{PE} receive these messages from each S_{PE} with the application tasks. When all tasks are loaded, the application is released by the M_{PE} to execute in S_{PE}s.

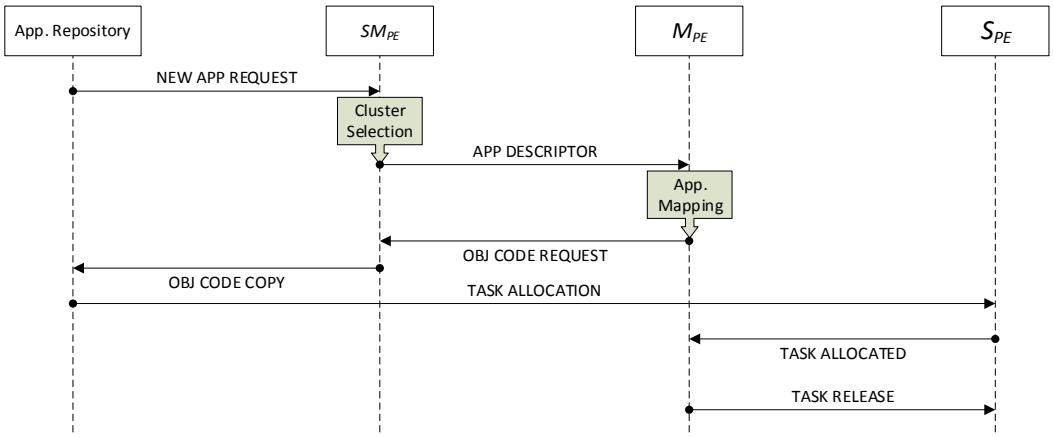


Figure 11 – Application Admission Protocol.

3.2.2 Inter-task Communication Management

The S_{PE} supports multitasking, interruptions, context saving and provides a communicating API for tasks by implementing two MPI-like primitives: *Send* and *Receive*. The *Receive* is a blocking primitive called by the consumer tasks. The *Receive* generates a MESSAGE_REQUEST packet to the PE of the producer task, which deliveries the data by sending a MESSAGE_DELIVERY packet when the producer task calls the *Send* primitive. When a consumer task calls the *Receive*, it goes to a waiting state until the reception of the message from the producer tasks.

Each S_{PE} kernel has a message FIFO, named *pipe*, which stores the messages produced by the running tasks. The *kernel* of the producer tasks manages the *pipe*. When a MESSAGE_REQUEST arrives at the producer kernel, it searches for the message in the *pipe*. If the message is in the *pipe*, the message is delivered to the consumer task. If the message is not in the *pipe*, the kernel stores the message request, and when the producer task calls the *Send* primitive, the produced message is forwarded to the consumer task.

3.2.3 Multitasking Management

The S_{PE} kernel support multitasking using a paging mechanism, which divides the memory into fixed-size pages during the kernel's boot. Figure 12(a) present the PE architecture detailing the logical division of local memory. Page 0 stores the kernel, and the remaining pages the applications' tasks. The number and size of pages (equal for all pages) are defined at design-time. Figure 12(b) details the memory structure for each page. The page has static and dynamic data sections. The static section is called *text*, which stores the task's object code. The dynamic sections are composed of the *data* (initialized date), *bss* (uninitialized data) and *stack*. This memory structure is defined by the compiler (e.g., GCC).

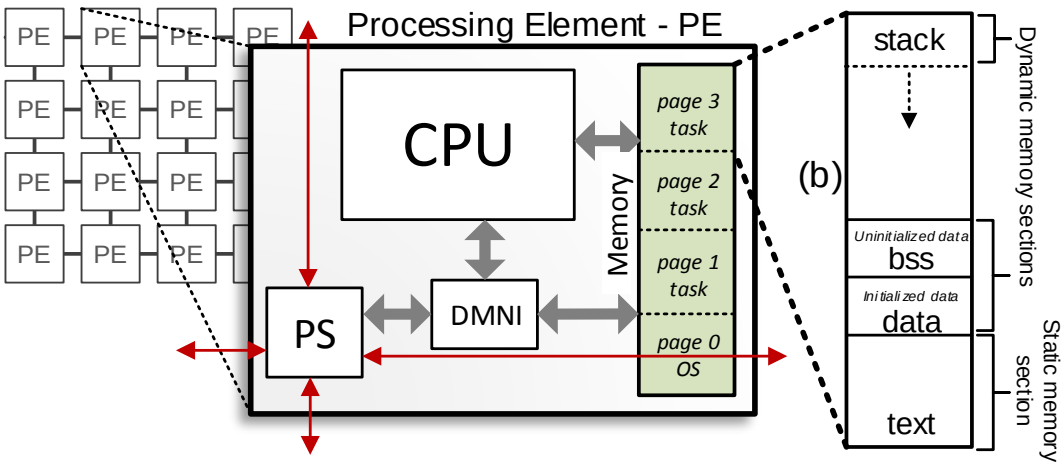


Figure 12 – Memory paging organization [RUA17b].

3.3 Communication between PEs

The communication between PEs occurs through unicast messages. The kernel manages the communication by sending and receiving packets. Packets may be related to the user inter-task

communication model, or to the system management. Figure 13 details the packet and message structures. From the NoC point of view, the packet has a header and a payload. The packet header has the target router address and the payload the packet size. From the kernel point of view, a message has:

- *message header*: encapsulates the packet header and the payload size and adds other control fields useful by the kernel, as the service header, which informs the service type (e.g., message delivery, message request, task allocation);
- *message payload*: optional field. It holds raw data used in the service header context (e.g., user data of a message delivery or an object code of a task allocation service).

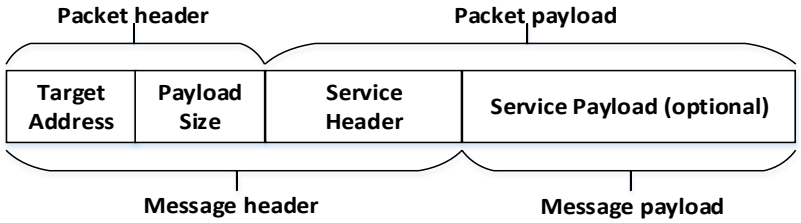


Figure 13 – Packet and message structures [RUA17b].

Figure 14 presents the flow to send and to receive a packet between two different PEs. When the kernel of the producer PE needs to send a packet, it calls a *send_packet()* function that programs the DMNI to start to send the packet, coping the data from memory and transmitting to the NoC. At the consumer side, when the DMNI receives a packet it interrupts the processor. The interruption handler calls the *read_packet()*, which programs the DMNI to read the packet coping it from the NoC to memory. Once the packet is completely received, the kernel executes functions related to the contents of the packet. For example, if the packet has data to a user task *t*, the packet (message in the user task context) is written in the *t* memory space, and *t* is scheduled to execute. The next Section presents more details about the API functions *send_packet()* and *read_packet()*.

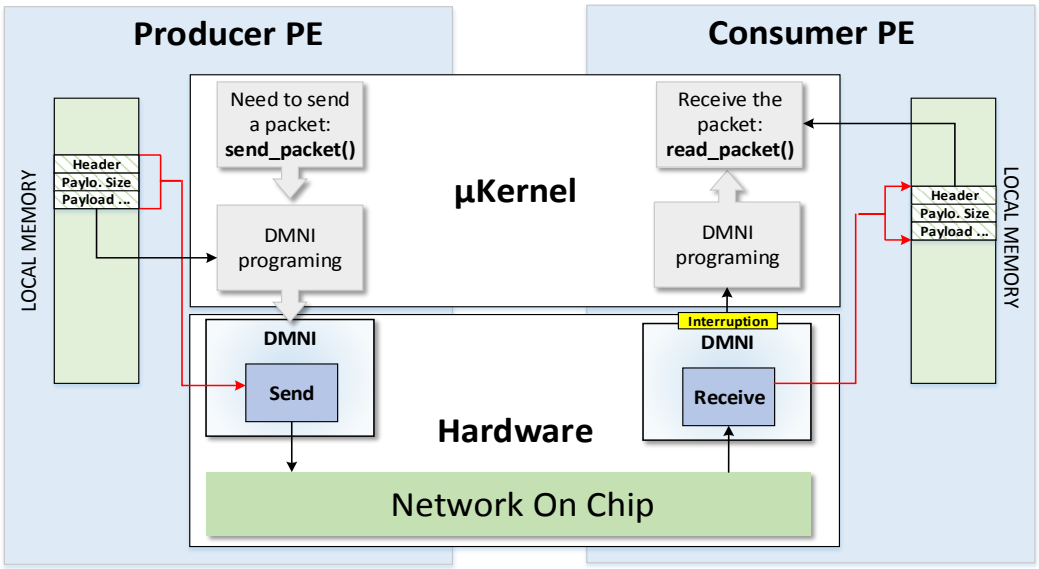


Figure 14 - Inter-PE communication flow [RUA16b].

3.4 Direct Memory Network Interface (DMNI)

As previously mentioned, the integration of many-core SoC with a NoC requires an NI. The goal of the NI is to make the interface between computation and communication, an interface between the processor and the NoC that implements the communication protocol to send and receive packets. Figure 15(a) shows a many-core architecture. Figure 15(b) presents a typical PE architecture with an NI integrated on the PE.

The architecture of Figure 15(b) is inherited from bus-based architectures and commonplace in NoC-based many-core designs. The processor has interfaces with different modules (NI, DMA), and the software has APIs to control each one. The processor programs the DMA writing in memory mapped registers the initial memory address and the block size to transfer. After programming the DMA module, the processor resumes the execution while the DMA transfers data to/from memory. For performance reasons, DMA is broadly used in systems that support real-time applications [LAP11].

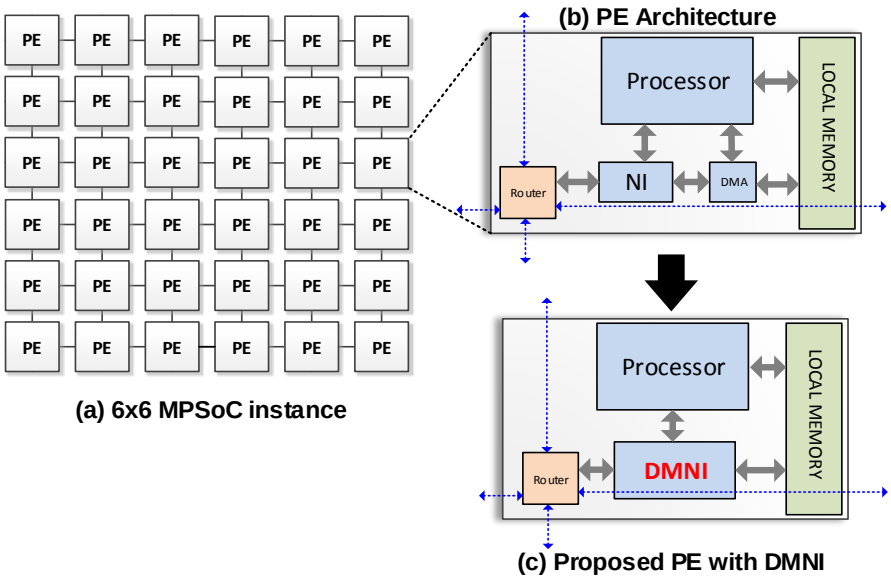


Figure 15 – Many-core and PE organizations [RUA16b].

Several works in the literature mention a design including a DMA and an NI [PAL12][ARN14]. Derin et al. [DER13] briefly mention an NI with DMA capability, but the work only groups the two modules, without an effective integration. Molnos et al. [MOL12] mention the use of a DMA to send and receive data between two memories (local and shared) through the NoC, without design details.

Attia et al. [ATT11] present a pipelined NI architecture for NoCs. The work presents a modular design, separating the injection and extraction path between the IP and the network sides. The proposed design outperforms other works in terms of latency and power, but the analysis is restricted to the NoC and the NI, without evaluating a complete system. Chouchene et al. [CHO11] add a low-power technique in the NI design of [ATT11], using a stoppable clock technique. The NI is turned off when there is no data to be handled.

Designs proposed in Matos et al. [MAT10b] and Swaminathan et al. [SWA12] target heterogeneous many-cores by using asynchronous communication architectures. Das et al. [DAS12] propose a fault-tolerant NI to be used in SDM (Spatial Division Multiplexing) NoCs, with serializers and deserializers to support the spatial division concept.

Kariniemi et al. [KAR10] propose an NI aiming to reduce the interruption frequency in the Micronmesh many-core by an interrupt batch mechanism. The results demonstrated a throughput improvement with longer messages. The work assumes the use of a DMA to improve communication latency but without specifying implementation details. Fanfga et al. [FAN10] propose an NI design combining a Lookup Table (LUT) mechanism and DMA features. The proposal is focused on the packet reception process. The tag segment of the LUT (programmed by the CPU) is compared with the tag information in the packet, and if matches, the address stored in the LUT can be used to start the DMA transfer directly. The goal, as in Kariniemi et al. [KAR10], is to reduce the interruption handler overhead. The Authors evaluate the latency to receive packets, with performance gains in larger packets.

Chen et al. [CHE10] and Ma et al. [MA15] employ an NI named DMC (*Dual Microcoded Controller*), targeting architectures with distributed shared memory organization. The DMC is a programmable hardware module that connects the memory, processor, and NoC. The DMC programming is eased using a microcode approach within two mini-processors. One mini-processor is used to handle local memory requests and the other to handle remote memory requests, by accessing the virtual shared memory space. A synchronizer ensures atomic memory access between the two mini-processors.

Some proposals for NoC-based many-cores does not assume a DMA implementation [ATT11][CHO11][MAT10b][SWA12][DAS12], focusing only on the NI design. Other works separate the DMA from the NI [PAL12][ARN14] or lack implementation details [DER13][MOL12][KAR10]. Works focusing on the integration of both modules, either lack validation data [FAN10] or cover specific implementations [CHE10][MA15]. As our proposal, works [KAR10][FAN10][CHE10] explore an NI design including a system perspective, identifying bottlenecks not addressed in previous works, as the cost to handle interruptions by the processor attached to the router NoC.

3.4.1 DMNI Design

This Section presents one of the side contributions of this Thesis. The goal of this proposal is to merge both modules (NI and DMA), into a new one, named *Direct Memory Network Interface* (DMNI), as shown in Figure 15(c). The main contribution is to give a specialized interface for NoC-based many-cores that directly connects the NoC router to the internal memory using a single module. The DMNI supports simultaneous packet reception and transmission, managed by a memory access arbiter, which interleaves the memory access when the send and receive modules are both active. A simplified programming interface exposes the DMNI services to the software layer.

A relevant feature of the DMNI is the access to two distinct memory blocks to transfer a packet. This feature is important because some transmissions between two PEs can come from two distinct memory regions, as, the message header and the message payload of the packet structure early explained.

Figure 16 details the DMNI architecture. The DMNI has three main modules: *send*, *receive*, and *arbiter*. The arbiter manages the memory accesses for both modules, enabling simultaneous send and receive operations. The kernel controls the DMNI through memory-mapped registers (MMRs). The DMNI design is generic because it enables to send and receive any type of data, not necessarily related to the message structure presented in Figure 13.

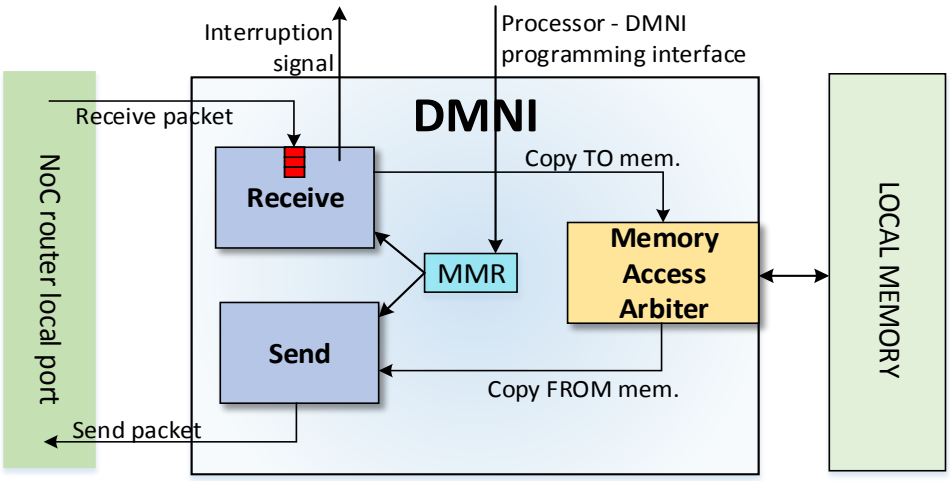


Figure 16 – DMNI architecture [RUA16b].

3.4.1.1 *Send Module*

This Section details the process related to the producer PE (left side of Figure 16). The role of the send module is to inject a packet into the NoC. The particular feature of this module is the possibility to transfer two memory blocks as a single transfer.

Figure 17 presents the *send_packet()* function provided in the DMNI API. It receives, respectively, the first and second memory sizes and addresses. If the DMNI is transmitting a packet (`DMNI_SEND_ACTIVE=1`), the procedure stays at line 2 until the release of the DMNI module. At lines 3 and 4, the first memory block is configured. If the message has a payload, at lines 6 and 7, the second memory block is configured. At line 8, it is written the operation type, i.e., read from memory. Finally, at line 9, the DMNI is released to start the packet transmission.

```
1.  void send_packet(mem_size_1, mem_addr_1, mem_size_2, mem_addr_2){
2.      while (MemoryRead(DMNI_SEND_ACTIVE));
3.      MemoryWrite(DMNI_SIZE, mem_size_1);
4.      MemoryWrite(DMNI_ADDRESS, mem_addr_1);
5.      if (mem_size_2 > 0){
6.         MemoryWrite(DMNI_SIZE_2, mem_size_2);
7.         MemoryWrite(DMNI_ADDRESS_2, mem_addr_2);
8.      MemoryWrite(DMNI_OP, READ);
9.      MemoryWrite(DMNI_START, 1);
10. }
```

Figure 17 – Send_packet() function, executed in the kernel of the processor [RUA16b].

Figure 18 presents the Finite State Machine (FSM) controlling the Send module. Initially, the FSM waits for the configuration of the MMRs (**WAIT** state) by the *send_packet()* function. When lines 8-9 of the function are executed, the FSM goes to the **LOAD** state, and the FSM assert a *send_active* signal to the arbiter to request access to the memory. The **LOAD** verifies if the local port of the router may receive data (*credit=1*) and if the arbiter allows a read operation (*read_enable=1*). If both conditions are satisfied, the data is read from memory and injected into the router local port (state **COPY_FROM_MEM**). Whenever the arbiter or the local port disables the transmission, the FSM returns to the **LOAD** state. The FSM sends the first memory block and then changes the address pointer to the second memory block (if configured) to transmit the remaining data.

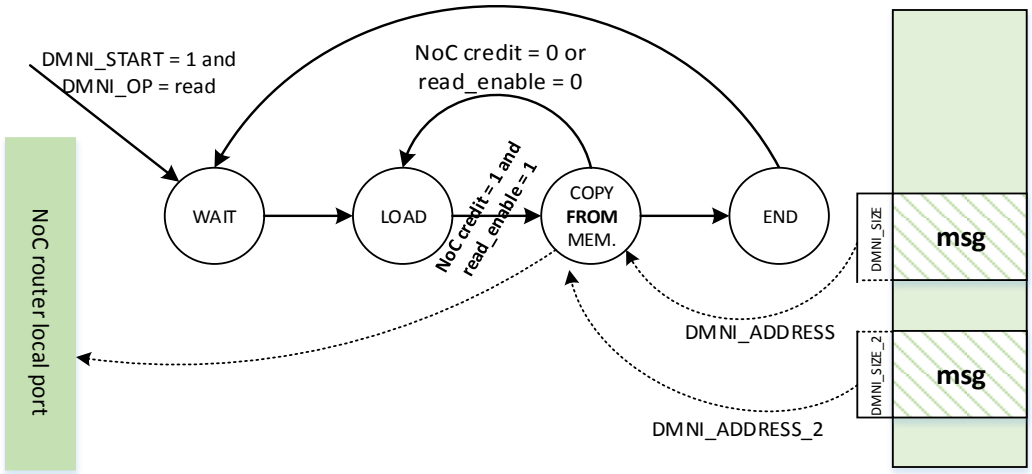


Figure 18 - FSM controlling the send module [RUA16b] .

Figure 19 presents the transmission of a packet configured with two memory regions: one starting at address 0x910 with contents {1, 7, A1, A2, A3}, and the second one starting at address 0x8c8 with contents {B1, B2, B3, B4}. Between cycles 3 to 10, the first memory block is transmitted (signal *data_out*). At cycle 11, the *send_size* becomes zero, changing the *mem_addr* signal to the second memory region, and the second part of the packet is transmitted. *The gap to change the memory region is only two clock cycles*. In a standard implementation (DMA+NI), which requires programming the DMA twice, the minimal gap is 22 clock cycles, penalizing the transmission of packets with a small payload.

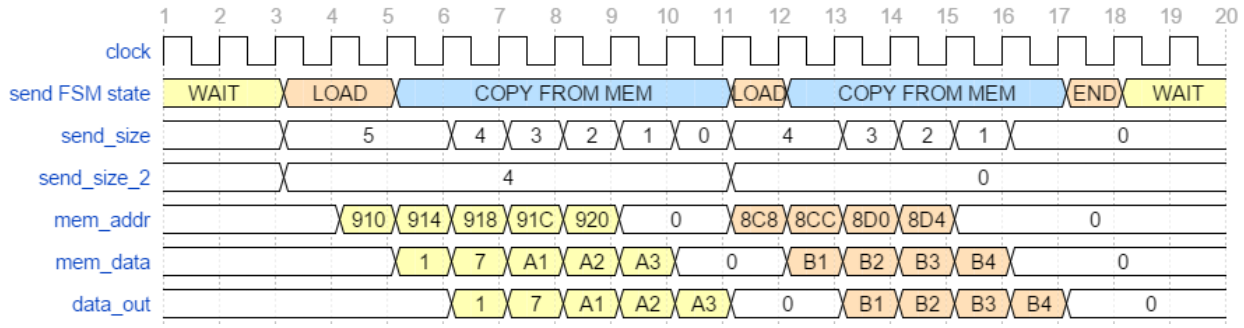


Figure 19 - Packet transmission by accessing two memory blocks [RUA16b].

3.4.1.2 *Receive Module*

This section details the process related to the consumer PE (right side of Figure 16). Figure 20 details the receive module. It contains two FSMs and a 16-flit buffer. The buffer depth is parameterizable at design-time.

When a packet arrives at the local port of the NoC router, the **HEADER** state reads the first flit of the packet, interrupting the processor (the interruption is not masked in order to avoid the packet to stall the NoC path). Next, the **PAYLOAD_SIZE** state reads the payload size and advances to the **DATA** state, which reads the remaining flits of the packet. The buffer receives all incoming flits. The NoC stalls when the buffer becomes full.

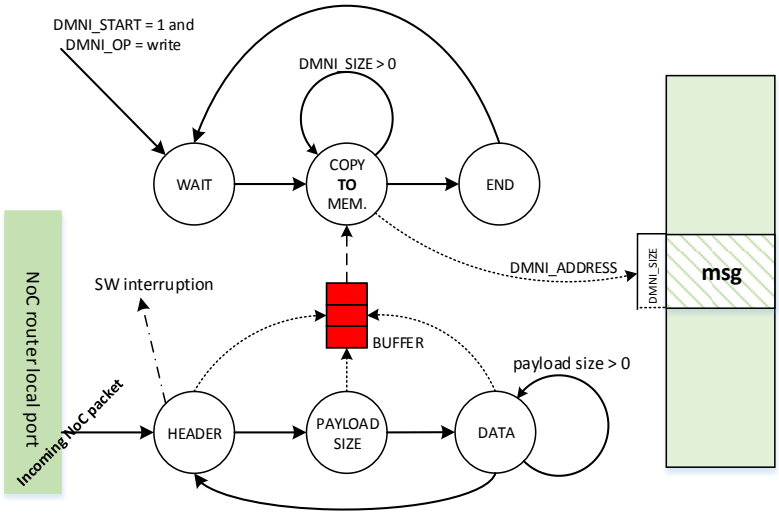


Figure 20 - FSM controlling the receive module [RUA16b].

Figure 21 presents the interruption handler process (*read_packet()* function). The *read_packet()* function writes into MMRs the amount of data to receive (line 2), the memory address to store the packet (line 3), the DMNI operation (line 4), and a *start* command (line 5). The kernel waits the complete reception of the packet (line 6) to safely read the packet content from memory, and executing the actions related to the packet service.

```
1. void read_packet (init_addr, packet_size)
2.       MemoryWrite(DMNI_SIZE, packet_size);
3.       MemoryWrite(DMNI_ADDRESS, init_addr);
4.       MemoryWrite(DMNI_OP, WRITE);
5.       MemoryWrite(DMNI_START, 1);
6.       while (MemoryRead(DMNI_RECEIVE_ACTIVE));
7. }
```

Figure 21 – *Read_packet()* function, executed in the μkernel of the processor [RUA16b].

The *write* and *start* conditions start the FSM at the top of Figure 21 (lines 4 and 5 of the *read_packet()*). This FSM transfer the data stored in the buffer to the local memory (state **COPY_TO_MEM**). To write into the memory, this second FSM asserts the *receive_active* signal to the arbiter to request access to the memory. The arbiter can grant access to the memory by asserting the signal *write_enable*. If the arbiter does not grant access to the memory, the FSM stays blocked

in the **COPY_TO_MEM** state.

Note that both FSMs of the receive module work in parallel. The first one receives data from the NoC storing the flits into the buffer, and the second one reads the buffer storing the data into the memory.

3.4.1.3 *Memory Access Arbiter*

The arbiter enables concurrent memory accesses to receive and to send packets. With such feature, the PE may receive new data and concurrently inject new packets into the NoC, interleaving the memory accesses. A round-robin (RR) arbiter enables this feature, by controlling two signals: *read_enable* (send) and *write_enable* (receive). A timer (DMNI_TIMER) controls the amount of time each module may access the memory.

Figure 22 presents the FSM controlling the arbiter. A signal named *round* selects the module to grant access. The receive and send FSMs assert the signals *send_active* and *receive_active*, respectively. When the arbiter goes to **SEND** state, the *read_enable* signal is asserted, enabling the send module to access the memory. The FSM stays in this state while *send_active* is asserted, or the timer expired and the other module requested access to the memory. Note that the arbiter may stay in the **SEND** state for periods larger than the timer limit if the other module does not request access to the memory. The **RECEIVE** state has the same behavior of the **SEND** state. When the FSM returns to the **ROUND** state, the *round* signal inverts, changing the order to verify which module must be served.

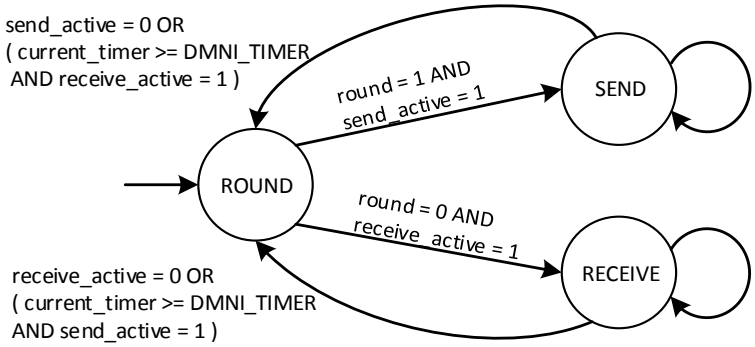


Figure 22 – FSM controlling the arbiter module [RUA16b].

Figure 23 presents the arbiter operation. For the sake of clarity, the DMNI_TIMER was configured to 5 cycles (*cc*). At the cc=2, the *receive_active* becomes true, signalizing to the arbiter that the send module needs to be stopped, and the RR scheduling executed. Note the timer value at this cc (0xF) is larger than the DMNI_TIMER (5) because only the send module is active. The RR execution occurs at cc=3, the timer returns to zero, the *round* signal is inverted, the *read_enable* becomes false, and the arbiter releases the receive module by activating the *write_enable* signal. The RR executes again at cc=10, and now the send module is released (*receive_active*←*1*). This interleaved operation continues while both *receive_active* and *send_active* signals remain asserted.

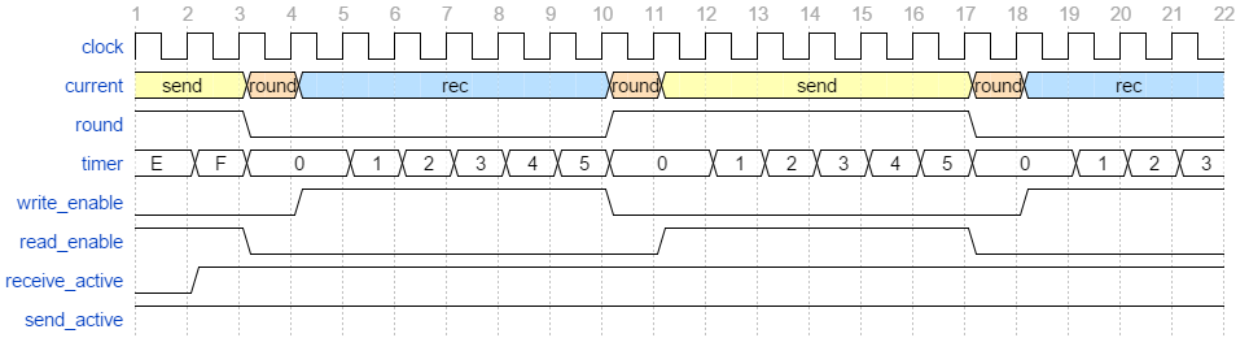


Figure 23 - Memory access scheduling [RUA16b].

3.4.2 DMNI Results

The DMNI was implemented using synthesizable VHDL, integrated into the reference many-core previosly presented. The baseline design, with separated modules, is named DMA+NI.

3.4.2.1 *Latency to transmit packets*

Figure 24 presents the latency to transmit packets with different sizes. The latency is measured from the moment when *send_packet()* is invoked up to the end of the execution of the *read_packet()*. Note that this latency includes the network latency, the interruption handling, and the context saving. The network latency represents a small fraction in the total latency, corresponding to 5 clock cycles per hop (in non-congested scenarios).

It is possible to observe that in both scenarios the latency grows linearly with the packet size. DMNI had a latency decrease of 116 cycles per 128 flits compared to DMA+NI. This reduction comes from two main reasons. The first one is related to software. For the DMA+NI implementation, the processor must wait the transmission of the first memory section and then program the DMA to transmit the second memory section. Using the DMNI, the processor programs once the memory regions, without be blocked due the transmission of the first section. The second reason is related to hardware. The unified DMNI design can transmit 1 flit per clock cycle, while in the DMA+NI it is necessary 2 clock cycles to inject one flit into the NoC due to the interface protocol between the two modules.

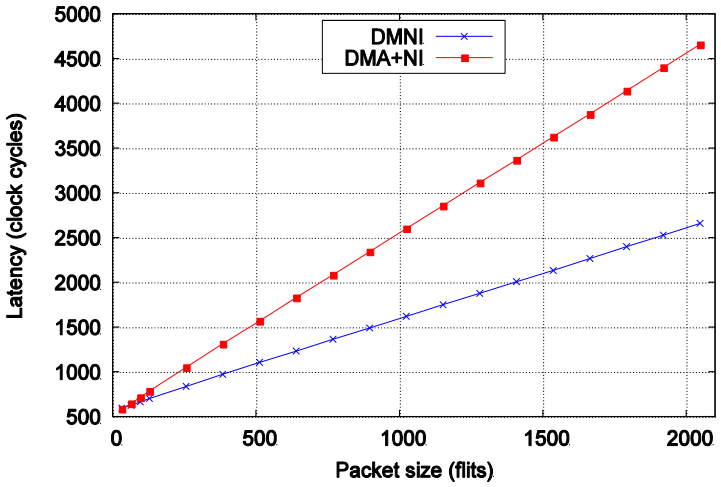


Figure 24 – DMNI and DMA+NI latency comparison [RUA16b].

A second latency evaluation concerns the latency in a real application, an MPEG decoder. The latency to decode one frame with the DMNI presented a reduction of 12.3% compared with DMA+NI. Further, the impact on the application execution time of MPEG is 15% lower with the DMNI design. Such results highlight the performance improvement offered by the DMNI, which specializes and simplifies the PE design.

3.4.2.2 *Area and State-of-the-Art Comparison*

Both designs, DMA+NI and DNMI, were synthesized using the Cadence ASIC design flow for a 65nm CMOS technology and prototyped in FPGA (Xilinx XC5VLX330), both using 100 MHz of clock frequency. Table 1 presents the area for the proposed DMNI, DMA+NI, and related works (those that have area report).

Comparing the baseline design (DMA+NI) with the proposed DMNI, there is a small area reduction (3.47%) when targeting an ASIC implementation. On the other side, for FPGAs, an important reduction in the number of flip-flops is observed – 48%, with an increased number of LUTs – 11.5%. The reduction observed in the number of flip-flops comes from the smaller number of registers required by the DMNI implementation.

Comparison to related works is difficult due to different specific goals and use different technologies. Observing the table, works from [DER13] and [MA15] (that use a DMA and a NI) have similar FPGA area results compared with DMNI.

Table 1 – Area comparison related to NI implementations [RUA16b].

Author	FPGA/ASIC	Work Goal	LUTs	FFs	Area
[DER13]	FPGA (Xilinx XC6VLX240T)	Network Adapter (DMA + NI)	879	577	N/A
[CHO11]	FPGA (Xilinx XC5VLX30)	Power-efficient NI (Credit Based)	420	590	N/A
[MAT10b]	ASIC (0.18um)	Asynchronous NI	N/A	N/A	18735 um^2
[MA15]	FPGA (Zynq7000)	Programmable NI (DME)	1163	313	N/A
Baseline design	***FPGA (Xilinx XC5VLX330) / ASIC (65 nm)***	Standard PE arch.	682	787	22141 um^2
This proposal		***Unified design - DMNI***	***761***	***409***	***21371 um^2***

3.5 Debugging

Previous Sections focused on the architectural features and protocols of the reference many-core platform. This Section details the proposition of a modular verification framework where it is possible to debug high-level computation and communication events in a many-core concurrently. It was used to validate and to debug the implementation of the reference many-core features, previously described, corresponding to the second side contribution of this Thesis.

The debugging framework has two contributions. The first one is a generic Data Extraction Layer, named DEL, which collects computation and communication events from the simulated many-core and stores such data into a database. The process of data extraction can be integrated with RTL, TLM, or virtual platforms descriptions. Such flexibility comes from an abstraction of the target many-core architecture implemented by DEL, which defines a generic data extraction method

combined with a standard database insertion. The second contribution is a graphical debugging toolset that explores the debugging database generated by DEL to create several GUI used for high-level communication and computation debugging. The data extraction (*back-end*) is decoupled from the graphical debugging tools (*front-end*), enabling the development of other custom front-end debuggers (graphical or not).

The originality of this proposal is a generic and scalable approach for data collection combined with a graphical toolset for debugging, it can be used jointly with state-of-the-art debugging methods acting as a complementary debugging approach.

Table 2 presents works related to NoC and many-core debugging. Debugging methods for many-cores and NoCs have gained increased attention due to the increased processor density, which makes the debugging process even more complex.

Table 2 - Related works in NoC and many-core debugging [RUA16c].

Work	Data Extraction	Target Debugging	Many-core description	Scalability concern	DB	GUI
[MUR14]	event inside cores	Parallel Software Concurrency	Virtual Platform	No, 4 cores	No	No
[GEO14]	software API	Software Errors	Virtual Platform	Yes, 32 cores	No	Yes
[WEN12]	core events	Parallel software data race	Virtual Platform	Yes, 64 cores	No	No
[CUE12]	breakpoints	Multimedia App. periodic conflicts	RTL	No, 3 cores	N/A	Yes
[PRA11]	SW instrumentation	Generic Observation	RTL	No, 3 cores	No	No
[HED11]	virtual HW events	Parallel software data race	Virtual Platform	Yes, 16 cores	No	No
[NEI12]	NI	NoC transactions and data race	RTL	No	No	No
[FRI14]	at run-time by a host unit	Visualize core logs	RTL (FPGA)	Yes, 45 cores	No	No
[ALH10]	router links	NoC link usage statistics	RTL (FPGA)	Yes, 16 cores	No	Yes
[MÖL10]	router links	NoC usage statistics	RTL	No, 9 cores	No	Yes
This Thesis	***router links, CPU events***	***Communication / Computation***	***RTL and Virtual***	***Yes, 400 cores***	***Yes***	***Yes***

Data extraction (2nd column of Table 2) addresses how the debugging method collects the data to be used in the debugging process. Some works extract data from the cores, enabling the debug of parallel applications [MUR14][GEO14][WEN12][CUE12][PRA11][HED11]. Others works adopt a communication-based data extraction by extracting data from the NoC links [ALH10][MÖL10] or the NI (Network Interface) [NEI12] to debug the communication events. This current proposal extracts the data from the router's links covering communication debug and extracts CPU events to cover computation debug. This approach provides a broad view of the system resources. Non-intrusive monitors extract these computation and communication events.

The *target of debugging* (3rd column) corresponds to the focus of the debugging. Three main debugging methods are identified in the literature: (*i)* parallel software debuggers; (*ii*) NoC debuggers; (*iii*) FPGA emulation. Most of the works adopt parallel software debug. Other works focus on improving the debug over FPGA implementations [FRI14][ALH10], and the works [ALH10][MÖL10] debug the NoC structure (NoC debuggers). The proposed method mixes the debug of computation and communication resources.

The many-core description (4th column), is related to abstraction level adopted to model the many-core. Authors [MUR14][GEO14][WEN12][HED11] adopt virtual-based platform descriptions. Such choice enables to speed up the simulation time to direct all debugging efforts to improve the software development. Other platforms are designed at the RTL level, with some proposals including FPGA emulation [FRI14][ALH10]. As the proposed framework adopts a generic data extraction, the designer implements the DEL according to the platform model. Therefore, both HDL models or virtual models are supported.

The scalability column (5th column) evaluates whether the debugging methods can handle large-scale data sets. The proposed framework leverages scalability by adopting a database (DB – 6th column of Table 2) to provide an efficient solution (described in the Results section) to access structured data with SQL queries.

The graphical representation (GUI) of the system events (7th column) eases and reduces the time spent debugging. Cueva et al. [CUE12] propose a simple GUI to observe the periodicity conflict between applications. Alhonen et al. [ALH10] and Moller et al. [MÖL10] propose graphical tools to generate NoC statistic. All the aforementioned works adopt simple graphical interfaces. The proposed framework presents several GUIs allowing a fast interpretation of the monitored system events.

3.5.1 Data Extraction Model

Figure 25 details the proposed *data extraction model*. It contains DEL and the database. The upper part of Figure 25 corresponds to a standard simulation environment. It assumes a generic NoC-based many-core, with a set of PEs interconnected by a NoC. The software part contains the kernel and a set of applications to execute on the system.

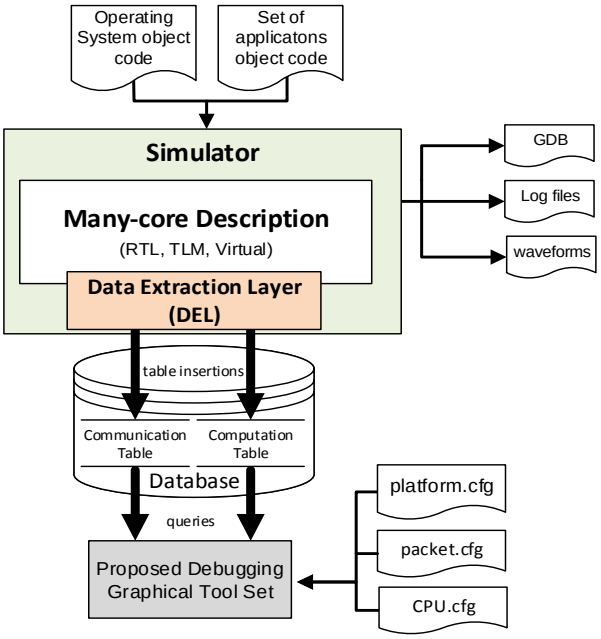


Figure 25 - Overview of the proposed debugging data extraction method [RUA16c].

A simulator receives the hardware and software parts of the system, simulating it using some

hardware description abstraction. A set of analysis may be executed according to the adopted simulator, as waveforms, log files, and GDB. This Thesis follows a different approach by proposing a generic DEL to extract communication and computation events and save them into a database following a standard structure (subsection 3.5.1.1). The front-end debugging tools communicate with the database according to a management protocol detailed in subsection 3.5.1.2.

3.5.1.1 *DEL: Data Extraction Layer*

The DEL collects data generated by the simulation and inserts them into database tables (e.g. latency, throughput, selected functions). *Communication events* are represented by packets arriving in the NoC router's input port. *Computation events* are specific software addresses executed into CPU. Such events enable to achieve a holistic view of the platform functional behavior. The DEL can be seen as the interface between the many-core platform and a front-end debugger (graphical or not), decoupling both parts of the system.

To achieve non-intrusiveness over the application execution, the DEL is implemented as a part of the hardware platform. This part is used only during the simulation at design-time, and it is removed for synthesis or prototyping. The DEL operation comprises two parts: *monitoring* and *database insertion*. The monitoring is in charge to extract the system data. For communication data, sniffers are implemented inside the router instance monitoring all input ports. The computation data extraction is implemented in a similar fashion, sniffing the CPU instruction address.

The database insertions are done directly from the hardware in the same process of the monitoring. The DEL can vary according to description level of the target platform. An RTL SystemC simulation enables database insertions directly from the SystemC code. A VHDL simulation can use a SystemC wrapper to execute the same task. The implementation of the DEL is a responsibility of the designer, once that it knows the platform details.

The data created by the DEL must follow a standard format to enable the insertion of the monitored events into the debugging database. The database implements two different tables. One for storing the communication (monitored packets) and the other for computation events (monitored addresses).

3.5.1.2 *Database Management*

The database is modeled in SQL. The communication and computation tables are created at the beginning of a new simulation. Figure 26 shows the sequence diagram for a new debugging scenario detailing the database management protocol necessary to the interaction between the Simulator, DEL, database, and front-end debugger.

At step *1*, the simulator initializes the DEL; at step *2*, DEL connects to the database server and creates a new database scenario for that simulation. A backup of previous databases is also created at this step. At step *3*, the front-end debugger can start a new debugging session, setting up a new connection with the database. The database connection of the DEL, at step *2*, and the connection of the debugger, at step *3*, are performed using a 5-tuple {*hostname, remote_port, scenario_name,*

user, password}. After the connection of the debugger with the database in step *3*, the debugging can start, in step *4*. The DEL extracts the platform communication and computation events and inserts into the database (step *5*). The front-end debugger can read such information, at step *6*, by requesting information from the database using SQL queries.

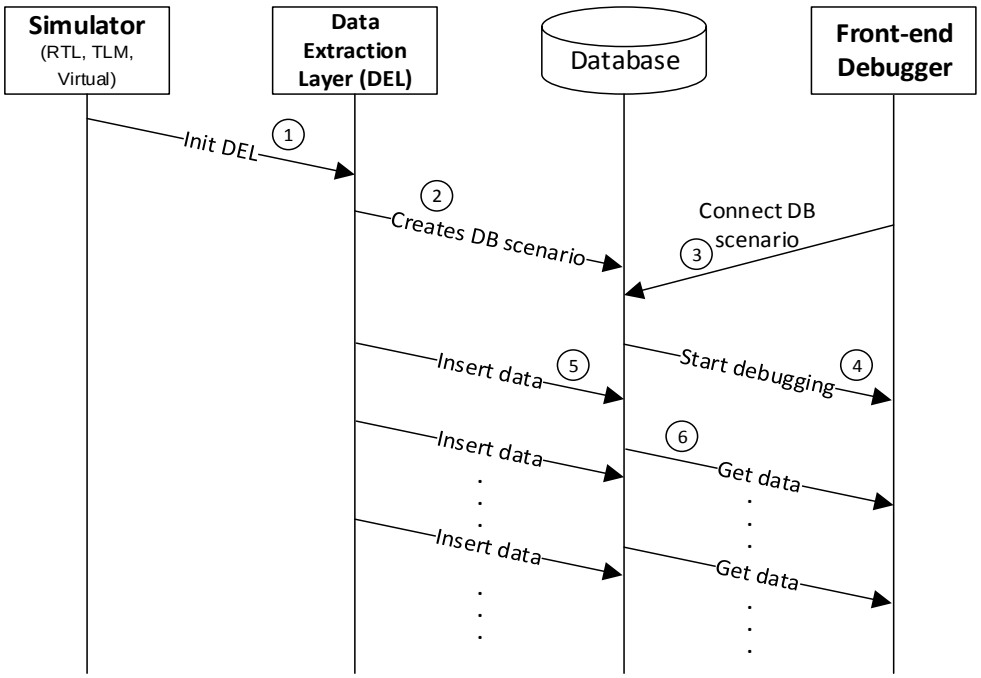


Figure 26 - Sequence diagram for database management [RUA16c].

3.5.2 Graphical Debugging Toolset

The proposed graphical debugging toolset (referenced from this point as *debugger*) works by reading communication and computation events stored in the database and converting such data into graphical information.

The focus of the debugger is to provide high-level awareness of the system status to the designer, enabling the analysis of computation and communication events. Figure 25 detailed the debugger inputs. The debugger receives configuration files at the initialization and performs queries over the communication and computation database tables.

3.5.2.1 *Configuration Files*

The debugger uses three configuration files: *platform*, *services*, *cpu*. These files can be generated automatically during the system compilation phase.

The *platform* file configures the debugger by providing parameters about the many-core architecture (e.g. size of the system) and the application tasks' set. The set of tasks to be executed in the many-core are listed as a tuple {*task name, identifier*}.

The *service* file contains the services supported by the platform. A *service* identifies the function of a given packet, and a *protocol* is defined by a set of *services*. For example, the message exchange protocol requires two *services*: MESSAGE_REQUEST and MESSAGE_DELIVERY. Each line of the

service file contains a tuple {*service name, identifier*}.

The *cpu* file describes the CPU addresses to be monitored. These addresses are monitored by the DEL, which generates computation events to informs that the addresses were executed. Examples of observation points are the addresses of the scheduler, system calls, task execution, interruption handler.

3.5.2.2 *Main View*

The main view shows an overview of the many-core architecture. By using this view, the system designer can debug the *communication* behavior, as NoC routing and link utilization, to validate system management protocols or task communication messages, and supervise parallel communications.

Figure 27 illustrates the view for a 4x4 many-core with 2x2 clusters. The green PEs represent cluster managers, the orange PE represents the cluster manager with access to the application repository (global manager), the blue PEs execute user applications. Each PE of this view contains the input channel utilization of each router port. Those values represent the percentage of the channel bandwidth usage, computed for a fixed time window (parameterizable in the tool).

Figure 27 – Main View: throughput and communication event views [RUA16c].

The zoom in PE 1x3 (Figure 27) details the channel utilization. It is possible to observe that the south port of the PE router has a channel utilization equal to 2.89% in this particular time window.

Each packet traveling into the PS NoC is displayed with a red arrow according to the packet advances to the next PE. The packet traces are colored in red. Figure 27 shows three packets traveling in the NoC at the time 71,225 (observed into *Speed Control* panel): 1x0 to 0x1, 1x2 to 1x3,

and 2x3 to 3x3.

3.5.2.3 *Mapping View*

Figure 28 presents the mapping view addressing *computation* debugging. With this view, the designer can validate task mapping algorithms, view the occupation of the PEs, and observe the task execution status.

Tasks belonging to the same application have the same color. Each task is displayed according to its name and ID detailed in the *platform* file. The designer can choose to see all task status, only the running tasks, or only the terminated tasks (tasks that already finished its execution). Tasks are displayed dynamically as they are mapped.

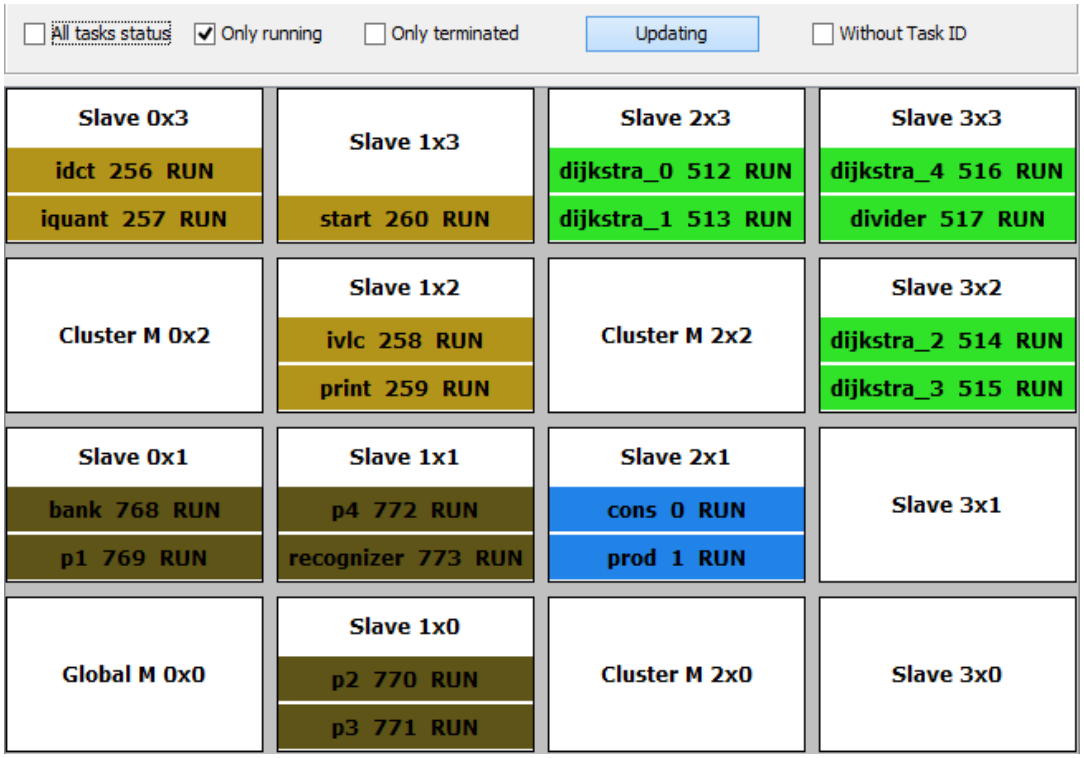


Figure 28 - Mapping view for a scenario with 4 applications, each one represented by a different color [RUA16c].

3.5.2.4 *CPU Utilization View*

The CPU utilization view addresses *computation* debugging. It enables the designer to verify the CPU use by different software parts over the time. The logged CPU events are related to kernel functions and the execution of a task. The addresses are described in the *cpu* file.

Figure 29 shows an example of the CPU utilization view for the PE 0x3. The y-axis corresponds to the monitored software events, and the x-axis the simulation time. This view also shows at the bottom left corner CPU utilization statistics, as the total CPU simulation time, and the percentage of CPU utilization for each kernel functions and each task executed by the CPU (in this example *idtc* and *iquant*). With this view, the designer can validate scheduling algorithms by verifying if a given task meets its constraints, evaluate the processing load to validate task mapping algorithms, correlate the processor events with communication protocols. Additionally, this view enables to

debug kernel or software bugs.

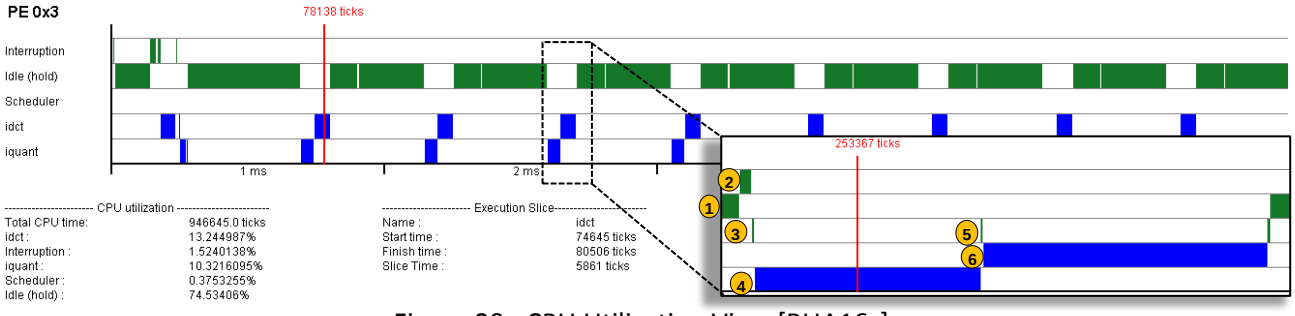


Figure 29 - CPU Utilization View [RUA16c].

3.5.3 Results

This Section presents results using the baseline many-core platform modeled in SystemC RTL. Applications (as DTW, MPEG, Dijkstra) and kernel are described in C language.

Figure 30 presents the debugging of a large-scale many-core (256 PEs), organized in a 16x16 PS mesh, with 4x4 clusters.

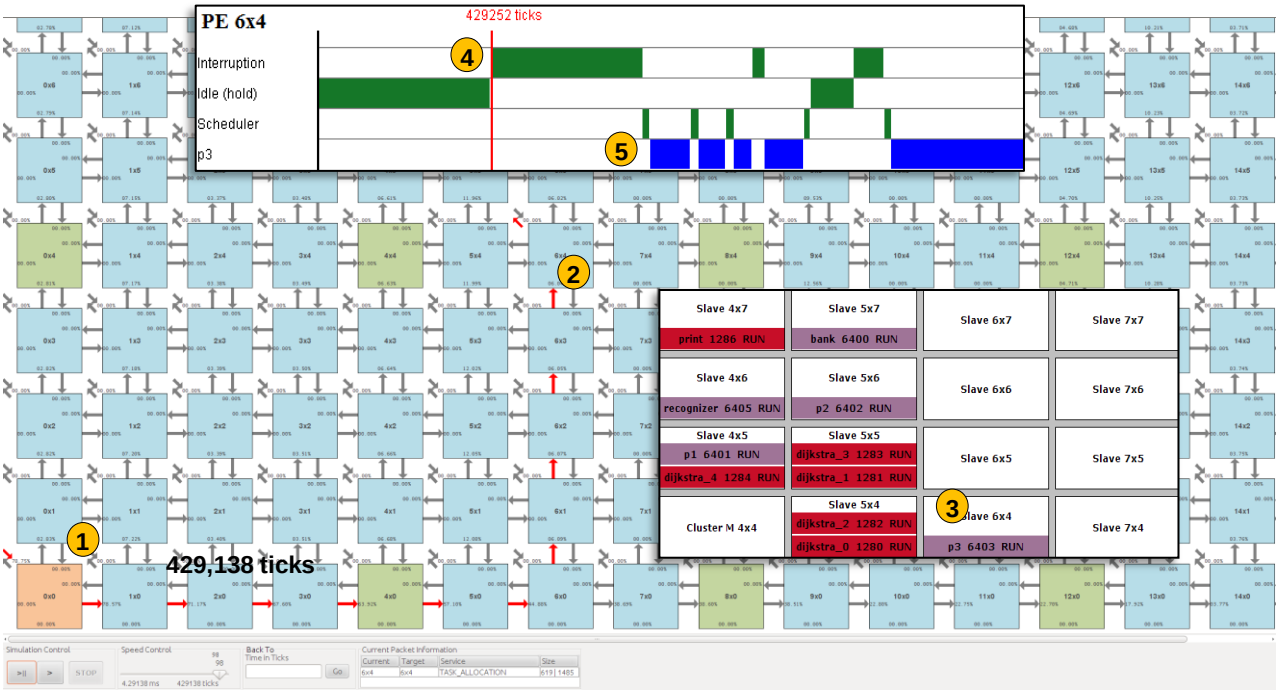


Figure 30 - Case-study debugging an MPSoC with 256 PEs [RUA16c].

The example in Figure 30 is a task allocation service, correlating the debugging of computation and communication events. The main view enables the user to observe a TASK_ALLOCATION packet leaving PE 0x0 (label 1 in Figure 30) and arriving at PE 6x4 (2) at time 429,138 ticks (clock cycles). This packet carries a task's object code. When the target PE (6x4) receives the packet, the mapping view shows the task *p3* allocated at this PE (3). The CPU view enables to observe that the kernel of the target PE was in the idle state. At time 429,252 the kernel executes the interruption handler (4), to consume the packet received at the local port of the router, with the task code (*p3*). Next, the scheduler selects the task *p3* and starts its execution (5).

Table 3 summarizes results related to simulation time and data storage, obtained from a SystemC RTL simulation, comparing the use of the database with an approach using log files (events stored in text files). The simulation time is penalized in 15% due to the operations to manage the database. On the other side, the required space to store the events reduces 26%. Executing an application in a 20x20 many-core for 10 seconds would require a data storage space of ~7 GB using log files, and ~5 GB using the database approach.

Table 3 - Results related to simulation time and data storage, for 100 ms of simulation (DB: database) [RUA16c].

System size	Simulation time (sec)			Data Storage (MB)		
	DB	log files	DB/log	DB	log files	DB/log
8x8	3,131	2,740	1.14	9.41	12.39	0.76
10x10	15,546	8,446	1.84	13.34	17.55	0.76
12x12	11,299	8,261	1.37	18.41	24.60	0.75
14x14	9,354	11,470	0.82	25.13	33.98	0.74
16x16	19,660	17,880	1.10	31.96	43.71	0.73
18x18	16,993	17,146	0.99	41.78	57.29	0.73
20x20	26,552	25,886	1.03	50.87	69.98	0.73

These results advance two key points related to debugging large-scale many-cores. The first one is the *graphical representation* of events generated during the simulation. It requires intuitive tools for accelerating debugging. The proposed framework provides GUIs that enables the designer to verify the system operation quickly. The second point is *scalability* since large-scale many-cores request flexible methods to handle datasets in a structured way.

4 QOS AT THE COMPUTATION LEVEL

This Chapter presents the contributions of this Thesis related to computation QoS. Section 4.1 presents the state-of-the-art review. Section 4.2 presents the first contribution, a low overhead task migration protocol. Section 4.3 presents the main contribution, the dynamic LST-based task scheduler. Section 4.4 presents a collaborative Ph.D. work, combining the proposed task scheduler with a run-time energy management. Section 4.5 finishes this Chapter presenting the conclusions.

4.1 State-of-the-Art

QoS at the computation level is mainly achieved by RT task schedulers at the processor level. Table 4 presents related works targeting task scheduling and QoS constraints. Most of the many-cores schedulers consider design-time steps integrated into frameworks [GAN13a][ROS14][BAM12] [TAF11]. Such works often employ a Model of Computation (MoC) (e.g., PPN, DAG, and SDGA), to model the applications at design-time. Applications are modeled as graphs, with nodes representing the actors, i.e., the application tasks, and edges representing the application communication dependency [BAM12][ROS14], which enables the developer to obtain accurate estimations about the communication and computation demands of each application. This approach makes easy to predict the applications behavior, helping to enable hard RT scheduling [DAV11]. Task scheduling based on MoC can be static (design-time scheduling) [GAN13a][BAM12][ROS14], dynamic (run-time scheduling) [OLI11], or both [TAF11]. Static or partial static (mixing design-time and run-time steps) scheduling is a conservative approach to guarantee hard RT behavior. Those proposals are only effective when the set of applications to execute in the system is known at design-time.

Table 4 – Works focused on task scheduling and QoS.

Work	Focus	Design-Time/Run-time	QoS
[GAN13a]	Processor resource scheduling framework	Design-time Framework	Multimedia stream app deadlines
[BAM12]	Scheduling periodic task modeled by CSDF graph using hard RT scheduling concepts	Design-time Framework	Throughput of periodic task
[ROS14]	Task mapping and task scheduling framework based on MoC	Design-time Framework	Fulfill RT deadlines
[TAF11]	Two task scheduling algorithms for temperature improvements while satisfies QoS	Mixed	No violation in real time tasks
[PAR14]	Handle I/O interrupts without disturbing hard RT tasks	Run-time	No interference in hard RT task
[OLI11]	Dataflow MoC-based MPSoC RTOS Scheduler	Run-time	Application execution deadlines
[HWA10]	Hybrid LST-based task scheduler	Run-time	Application execution deadlines

Different from a static scheduler, dynamic schedulers perform the scheduling decision at run-time. Such behavior comes at the cost of scheduling overheads and unpredictability [PAR14][OLI11]. *Pfair* (*Proportionate-fair*) is a state-of-the-art dynamical task scheduler for many-cores that claims to support hard RT tasks. It assumes run-time task migrations to improve system's utilization. Theoretically, hard RT support is achieved by assuming an excessive number of task migrations and

preemptions, without considering their interferences over the system. Indeed, this assumption leads to a high overhead which makes the *Pfair* applicability inviable to many-cores [CHA01].

Park et al. [PAR14] propose HPGP, a hybrid scheduler for many-cores based on *Pfair*. The hybrid feature means that the *Pfair* algorithm is executed inside local processors combined to a global scheduler, which makes task migrations. The goal is to reduce the number of task migrations compared to *Pfair*. The manager selects the ready tasks to be scheduled into a PE. Each PE executes a schedulability analysis. If the task is not schedulable, the core invokes the global scheduler to perform task migration. While the scheduler is labeled as dynamical, the application set if fixed at design-time. The proposal considers only periodic tasks with deadlines equal to their respective periods and constant execution time. The evaluation is carried out only with four cores, which does not enable validating the algorithm based on modern large many-cores systems.

The LST (*Least Slack Time*) scheduler is a task scheduler where the priorities of each job are based on their slack time, with smaller slack times resulting in a higher priority. At a given time *t,* the slack (or laxity) of a job with deadline *d* is equal to *d* - *t* minus the time required to complete the remaining portion of the job. While the LST is an optimal scheduling for mono processed systems, it was proved to be non-optimal to multiprocessor systems [LIU00]. Hwang et al. [HWA10] propose LSTR, a dynamic scheduling algorithm based on LST, with additional features to be optimal for multiprocessor systems. LSTR was designed to support only periodic tasks. The limitation of the proposal is that it is based on a theoretical evaluation, not addressing system overheads.

The design of a dynamic RT scheduler for many-cores should consider, among other factors, how to inform the RT task constraints to the system, and how to handle interruptions. The management of interruptions may interfere with the execution of RT tasks. Interruptions can be handled immediately using specific system routines or can be pooled at fixed and predetermined times [HAN11]. Another alternative is to redirect interruptions handling to free cores [PAR14]. However, this option only can be useful when interruptions come from external devices and are not related to inter-task communications. The kernel of *HeMPS* handles interruption immediately using system routines, enabling to quickly handle the interruptions, reducing the time to buffer them into the NoC, which could create a significant communication interference. However, this approach does not enable the scheduler to support *hard* real-time applications.

Dynamical RT task schedulers are integrated into an RTOS (Real-Time Operating System). An RTOS, besides the RT scheduler, must also provide support for fast preemptions, improved context saving, and low intrusive interruption mechanism. Those features conduct to a higher-level task execution composability and small OS execution time overhead [OLI11], opening more space to the RT task execution.

Provide QoS to a system consists of managing its resources. A typical approach is to work with worst-case execution time (WCET) estimation of applications to guarantee that the reserved resources are enough to all application corner cases. However, some Authors, presented in Table 5, argue that current mechanisms for WCET estimation are inefficient and conduct to system

underutilization. Thus, there are proposals focusing on relaxing WCET [NOW13][YU09] or proposing improvements of the WCET estimation [SHA12].

Table 5 – Works focused in WCET and QoS.

Work	Focus	Design-Time/Run-time	QoS
[NOW13]	Increase MPSoC utilization by a soft worst-case estimation	Run-time time slice manager	Full fill hard RT constraints
[YU09]	Relax worst-case estimation	Design time framework	Meet soft RT constraints
[SHA12]	Priority budget memory scheduler based on precise WCET estimation	Run-time	Memory access priorities

Power, energy, and temperature are critical parameters to be controlled. QoS policies become tightly coupled with such parameters due to the growing need for energy saving and temperature reduction. Table 6 presents works focusing on temperature, power and energy reduction, and simultaneously providing some degree of QoS (STM stands for *slack time monitoring*).

Table 6 – Works focused on temperature, power and energy reduction and QoS.

Work	Focus	Design-Time/Run-time	QoS	Technique	STM
[TUV13]	Run-time Adaptive task migration and power reduction	Run-time	Meet RT task constraints	Clock gating	Yes
[HAN11]	Composable OS design	Design methodology	Task execution composability	Clock gating	Yes
[YUN13]	Temperature aware scheduling	Run-time	Preserve task deadlines	Power gating	Yes
[GAN14]	Dynamic Power Management for temperature reduction	Run-time	Fulfill video application latency	Power gating	Yes
[JAV14]	Energy efficient pipelined MPSoC management	Run-time	Fulfill multimedia application deadlines	Clock and Power gating	Yes
[YU14]	Temperature aware scheduling	Run-time	Aims to increase QoS budget	Frequency scaling	No
[ABB14]	Dynamic Frequency Scaling based on PID controller	Run-time	Fulfill real time throughput deadlines	Frequency scaling	No
[GUI13]	Dynamic Frequency Scaling with software management	Run-time	Fulfill real time throughput deadlines	Frequency scaling	No
[DAS14]	Temperature aware task mapping, scheduling and PE's DFVS levels	Design time	Fulfill throughput constraints	DVFS	No
[KOT10]	Energy aware scheduling by exploring task slacks	Design time using linear programming	Fulfill task deadlines	DVFS	Yes
[MOL10]	Composable and energy efficient MPSoC architecture	Theoretical proposal	RT application composability	DVFS	Yes
[JUN14]	Energy efficient task mapping	Run-time and a design time behavior specification	hard RT application constraints	DVFS	No
[LI13]	Temperature aware scheduling with task migration support	Both	Execution deadlines	DVFS	Yes
[SIN13]	Energy reduction methodology based on slack time management	Both	RT application throughput	DVFS	Yes

There are two main classes of works that mix computation QoS policies with low power techniques: (*i*) clock and power gating; (*ii*) DFVS. Clock and power gating techniques can be used to the management of power [TUV13], energy [JAV14], and temperature [GAN14][YUN13]. Clock and power gating techniques are well suited to work with task migration due to the possibility to remove

the workload of a hot or overused PE, opening space to apply such techniques. The PE's slack time can be exploited to make the processor run in low power mode, achieving significant power reductions [TUV13][YUN13]. Clock and power gating can also be explored at the software level, working in conjunct with a composable OS design [HAN11].

Some works opt to employ frequency scaling technique to provide QoS. This technique can be used at design-time [YU14] or run-time [ABB14][ALM11][GUI13]. At design-time, using complex heuristics, the methods search for the appropriated temporal scenarios where the frequency can be scaled while satisfying QoS. This estimation requires a vast number of architectural characteristics variables as well as a robust application's description [YU14]. At run-time, frequency scaling can be applied by using monitoring and run-time management modules, which tune the processor's frequency according to the application's deadlines.

DVFS-based techniques can gradually scale the system performance and system power and are suitable when different QoS levels are needed. In fact, DVFS is a broadly adopted technique for power, energy, and temperature reduction and at the same time, offering some degree of QoS. DVFS can be applied at design-time (or offline) [DAS14] or mixing a design-time with the run-time phase [JUN14][LI13][SIN13a]. DVFS-based task scheduler has been demonstrated efficient to save energy and to reduce the temperature. Task and CPU slack time exploitation can be used to reduce the impact of voltage and frequency decrease without compromise QoS [KOT10][MOL10][LI13] [SIN13a][MAR17a].

Due to the number of PEs in many-cores, the memory access can generate bottlenecks in shared-memory architectures [OLI11], requiring QoS techniques. Table 7 presents works focusing on memory access scheduling QoS. QoS-aware memory scheduling mechanisms are required to manage the memory access according to a QoS policy either to prioritize real-time tasks access [SHA12][LIN10][KIM10][WAN14] or to provide memory access composability [LIU13].

Table 7 – Works focused on memory access scheduling QoS.

Work	Focus	Design-Time/Run-time	QoS
[SHA12]	Priority budget memory scheduler based on precise WCET estimation	Run-time	Memory access priorities
[LIN10]	Hierarchical DRAM memory access scheduling	Run-time	Memory access priorities
[KIM10]	Cluster memory access scheduling	Run-time	Memory access priorities
[LIU13]	Memory scheduling policy based in slack time management	Run-time	Composability
[WAN14]	Memory scheduling optimized for heterogeneous computing	Run-time	Memory access priorities

4.1.1 State-of-the-Art Discussion

Analyzing the reviewed works, essential features to provide QoS at the computation level include: (*i*) real-time task scheduling; (*ii*) task slack and processing idle time monitoring; (*iii*) DVFS for energy and temperature reduction; (*iv*) a relaxed WCET estimation.

The QoS, in the computation context, is mainly represented by RT task schedulers. Several proposals addressed scheduling algorithm for multiprocessor systems. However, the proportion of

those targeting many-cores is still low and have limitations related to clock-cycle validation, scalability, and dynamic behavior.

As dark silicon era imposes power reductions [SHA17], designers must find a way to ensure QoS in those scenarios. DVFS outstands as the main technique to reduce energy and temperature while allows controlling system's performance. Some Authors also use clock or power gating techniques. These techniques can be applied either in design-time methods or run-time managers. In the same way, DVFS also can be applied at design-time, by expensive formal models and an earlier and detailed system characteristics behavior, or adaptively, by using run-time power, frequency, and voltage managers. A system monitoring scheme continuously feeds these managers, with task's slack time and idle time of each processor, which enables the execution of heuristics that find an efficient tradeoff between QoS, temperature, and energy consumption.

Another recent topic in the design of many-cores is the concern by relaxed WCET. This research is motivated by the dark silicon era that makes task mapping and task scheduling more complex and challenging [SHA17]. Relaxing WCET can provide a significant efficiency in resource utilization and save energy while the real-time present negligible deadline misses.

This Thesis addresses QoS at the computation level by proposing a *dynamic* and *hierarchical* RT task scheduler based on the LST algorithm. The proposed LST scheduler is dynamic because it supports an unknown application set (dynamic workload), and enable tasks to change its RT constraints (period, deadline, execution time) at run-time. Faced with an RT workload change the scheduler sent this feedback to the QoS management (also is a global scheduler), which decide to employ an original task migration protocol. The hierarchical property is related to the organization of the scheduler, with local schedulers (running in the S_{PE}s) and global schedulers (running in M_{PE}s). The main function of the local scheduler is to manage the CPU allocation, while the main function of the global scheduler is related to the S_{PE}s task allocation, executing task migration during the application execution.

The proposed task scheduler was also integrated into an original energy management technique proposed in a collaborative work with the Ph.D. student Andre del Mestre. The QoS manager is aware of the slack-time of the whole application and exploits this time to apply DVFS and to save energy.

4.2 Task Migration

Self-adaptive techniques, as thermal and power management, load balancing, QoS, and fault tolerance commonly use task migration to perform run-time management in many-core systems. In thermal and power management [SAL14], task migration can be employed together DVFS to distribute the tasks according to the performance of the cores. For load balancing and aging control [JOH12][MAR11], task migration can help to distribute the workload of the system, by migrating several tasks simultaneously to PEs with lower utilization. For QoS [ABB14], task migration may be used to reserve CPU resources for real-time tasks. For fault tolerance [DAS13], task migration can

be used to move tasks running in faulty PEs.

Proposals related to task migration addressing many-core systems and assuming a distributed memory have at least one of the following features: (*i*) use of checkpoints, which requires source code annotation; (*ii*) task replication, with a replica of the task in one or more PEs that can receive the task to migrate (wasting memory); and (*iii*) significant migration latency.

The goal of this proposal is to develop a task migration technique between PEs having the same ISA (Instruction Set Architecture), for many-core systems with a distributed memory hierarchy. The contribution is a protocol with a lower latency compared to the related works. Further, this proposal eliminates the need for checkpoints, not requiring task replicas, and enable to migrate tasks of the same applications simultaneously.

The key point to reduce latency comes from a two steps procedure. First, the text memory section (object code) of the task to migrate, T_M, is migrated separately from the dynamic memory sections: data, bss, and stack. While the T_M's text is transferred from the source PE (Src_{PE}) to the target PE (Tgt_{PE}), T_M keeps running at the Src_{PE} being blocked only during the dynamic memory migration. The DMNI module is programmed to copy the text section while the CPU keeps running T_M. The second step comes from an inter-task synchronization protocol, which does not migrate the messages produced by T_M. The synchronization is performed on-demand after the migration, helping to reduce the migration data volume while ensures no message loss.

Table 8 summarizes the main features of the related works in task migration for distributed memory many-core systems. The 2nd column details the adoption of checkpoints. Checkpoints simplify task migration because they statically define migration points where the task context is safely saved. These checkpoints are inserted on task's source code, being difficult in practice to the application developer define safe states to migrate. The 3rd column addresses the inter-task synchronization. El-Antably et al. [ANT15] and Fu et al. [FU13] transfer all pending messages during the migration process (FIFO copy), Canella et al. [CAN12] wait for the consumption of all messages to execute the migration (FIFO release), and Saint et al. [SAI08] use the communication primitives as checkpoint to enable the migration. The proposal herein presented adopts an approach similar to the theoretic proposal of Munk et al. [MUN15]. The synchronization occurs on-demand according to the messages request by the consumer tasks.

The 4th column of Table 8 evaluates the use of an MMU (memory management unit) in the migration process. The adoption of MMU simplifies the migration process since the operating system handles virtual addresses, at the cost of additional hardware. As most of the works, the present proposal does not adopt an MMU. The memory management is simplified by using a paged memory organization managed by the OS (operating system).

Table 8 - Comparison of task migration works [RUA17b].

Proposal	Check.	Inter-task sync	MMU	Migration
[MUN15]	no	Message forward	no	recreation
[ANT15]	yes	FIFO copy	no	replication
[FU13]	no	FIFO copy	no	replication
[CAN12]	no	FIFO release	no	replication
[JAH11]	no	N/A	yes	pre and post copy
[SAI08]	yes	Send / Receive	no	recreation
This proposal	***no***	***Message forward***	***no***	***recreation***

The 5th column presents how the migration process is executed. Task replication [ANT15][FU13][CAN12] keeps a task replica at different processors. This procedure reduces the task migration latency because the *text* section is not transferred, but incurs in memory overhead. Task recreation [MUN15][JAH11][SAI08] stops the task execution, transfer the *text* and the context to the target PE, and then resume the execution. Aided by an MMU, Jahn et al. [JAH11] adopt a mixed mechanism, which pre-copies the most frequent data memory sections before the migration of the task context, and transfers the remaining memory sections by requests sent to the Src_{PE} after the task migration.

4.2.1 Proposed Task Migration

The task migration proposal is implemented as a kernel service. Figure 31 presents the task migration protocol. The task migration is triggered when a migration order arrives at the Src_{PE} (the generation of this event is out of the scope of this specific proposal). The kernel handles this order by configuring the DMNI to transfer the T_M *text* section (event 1). The *text* section is sent to the Tgt_{PE} through a MIGRATION_CODE message (event 2). After the DMNI configuration, T_M continues its execution up to reach a *safe* state, where the dynamic data memory sections can be safely migrated. The safe state is automatically defined by the kernel at run-time.

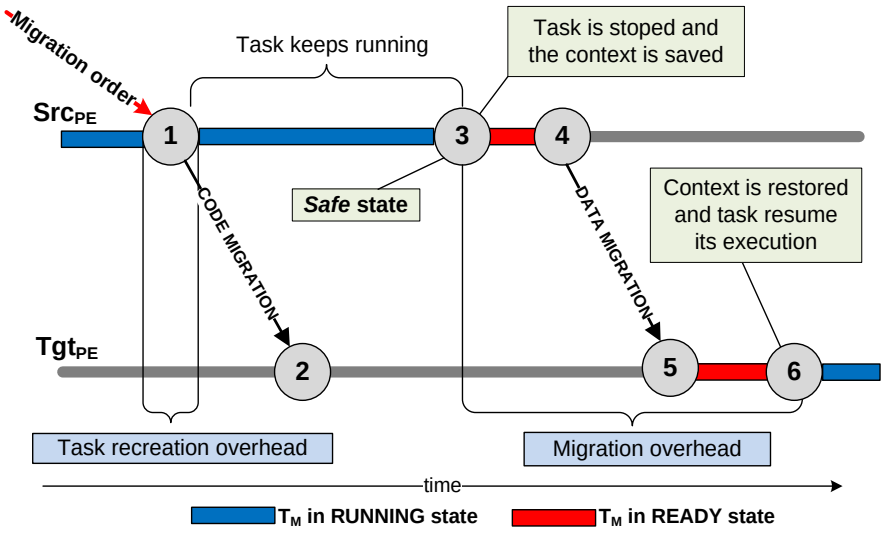


Figure 31 – Overview of the proposed task migration protocol [RUA17b].

Safe state definition. A given task may be in 3 states at the task scheduler perspective: (*i*) *ready*, the task scheduler can schedule the task to execute; (*ii*) *waiting*, the task is blocked waiting

for a message delivery; (*iii*) *running*, the task is running, using the CPU and changing the dynamic memory sections. *A safe state is defined when T_M is in the ready state.* If a migration occurs during the *running* state, the dynamic data memory sections are in use, corresponding to an unsafe state. If migration occurs during the *waiting* state, the delivery of the requested message will arrive at the old PE of T_M, inducing a message loss. The delay to start the data migration is a function of the task scheduler time slice and the time to the producer task to deliver the requested message.

Event 3 in Figure 31 represents T_M in a safe state. The task stops, and the kernel saves the context, transferring the context (CPU registers values) and the data memory sections to the Tgt_{PE} by a DATA_MIGRATION message (event 4). At event 5, the DATA_MIGRATION arrives at Tgt_{PE}. At event 6, the kernel of the Tgt_{PE} restore the T_M context (by copying the DATA_MIGRATION payload to T_M page), and T_M goes to the *ready* state. The overhead of the proposed protocol comes mainly from event 3 to event 6, which varies according to the size of the data section, and the interconnection infrastructure. The overheads are detailed in Subsection 4.2.2 assuming a 2D-mesh NoC as the interconnection infrastructure.

4.2.1.1 *Inter-Task Synchronization*

A key feature of the task migration protocol is the synchronization of the messages exchanged between tasks. Each S_{PE} kernel has a task *location table*. The *Send* and *Receive* primitives use this table to find the address of the communicating tasks. When T_M migrates, this table must be updated in all PEs with tasks that send and receive messages to/from T_M. The update is executed on-demand according to the following rules:

Rule 1: when T_M migrates to Tgt_{PE}, all produced messages by T_M stay in the *pipe* of Src_{PE}. After T_M migration, the consumer tasks continue sending MESSAGE_REQUESTs to Src_{PE}. If there *is a message in the pipe*, the message is removed from the *pipe* and delivered to the consumer task. Figure 32(a) presents this scenario. At event 1, task A migrates from PE0 to PE1. Messages produced by task A, to tasks B and C, stay in the pipe of PE0. At event 2, task B sends a MESSAGE_REQUEST to PE0 (old address of task A). As there is a message in the *pipe* to task B, this message is delivered to task B (event 3).

Rule 2: If there is *no message for the consumer task* in the *pipe* of Src_{PE}, the MESSAGE_REQUEST is forwarded to Tgt_{PE} and Tgt_{PE} deliveries the requested message to the consumer task. The kernel of the Src_{PE} has the new address of T_M, enabling to forward the MESSAGE_REQUEST. Figure 32(b) presents this scenario. Task B sends a MESSAGE_REQUEST to task A (event 1), and the *pipe* at PE0 does not contain messages to task B. Thus, the MESSAGE_REQUEST is forwarded to the new address of task A (event 2). Task A at PE1 handles the request and deliveries the message to task B (event 3). Figure 32(c) corresponds to the consumption of the last message produced by T_M in PE0 after the migration.

Rule 3: The task location table is updated in the kernel of the consumer task with the Tgt_{PE} address when a MESSAGE_REQUEST is received by Src_{PE}, and there is *no message produced by T_M in the*

pipe of Src_{PE}. Thus, a MESSAGE_REQUEST is forwarded to Tgt_{PE} as in rule 2, but a second message is sent to the kernel of the consumer task to update the task location table with the Tgt_{PE} address (UPDATE_LOCATION message). After this update, the kernel of the consumer task uses Tgt_{PE} address to send all requests to T_M. Figure 32(d) presents this scenario. When PE0 receives a MESSAGE_REQUEST (event 1), this message is forwarded to PE1 (event 2), and a UPDATE_LOCATION message is sent to PE2 (event 3) to update the task location table with the new location of task A (PE1). After receiving the forwarded request, PE1 deliveries the message to PE2 (event 4).

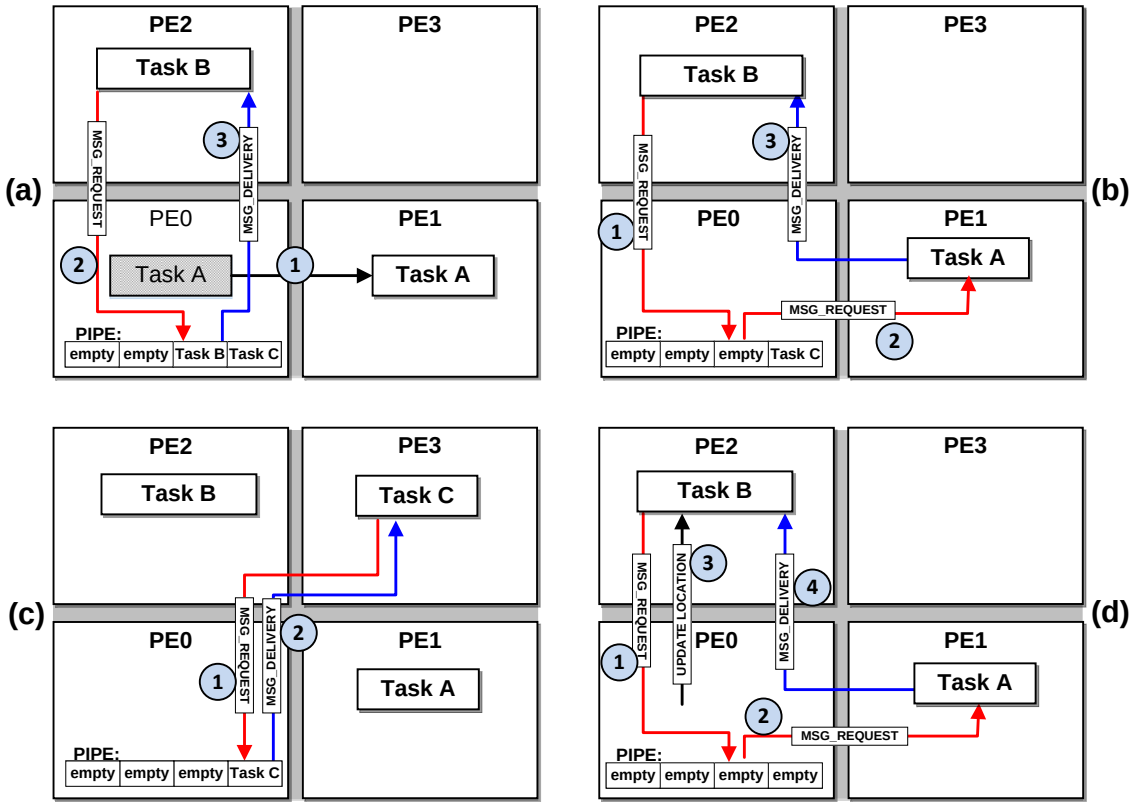


Figure 32 – Representation of the rules involved in the inter-task synchronization [RUA17b].

4.2.2 Results

The results were conducted in an RTL SystemC description of the reference many-core system. The migration latency is due to the *task recreation* step and the *dynamic data migration*. The task recreation latency is constant, regardless the T_M *text* size, once we adopt a dual-port memory and a the DMNI (which is able to to copy two blocks of memory by making a single programming). In systems with a single-port memory, this latency is proportional to the *text* size because the processor stalls during the transference. The task recreation time is due to the time spent by the kernel to configure the DMNI with the memory address and size of the *text*. Once configured the DMNI module, T_M resumes it execution while its *text* is injected into the NoC. This process is similar at the destination PE, when it receives the *text* message, it stops the execution of the current task and programms the DMNI. After the programming it resumes the execution of current task while the *text* is copied to the local memory by DMNI. The task recreation step (DMNI configuration phase) takes, on average, 2,700 *cc* (clock cycles). This value may suffer small variations due to the status of

the DMNI at moment of kernel request. The migration latency corresponds in fact to the time spent to transfer the dynamic memory section. The overhead grows linearly with a complexity of *O(n)*, where *n* is the sum of all dynamic memory sections and the T_M context (PC, SP, registers). Transferring this data requires to stop T_M, and restart T_M at Tgt_{PE} after the end of transference.

The evaluation of the dynamic data migration latency is carried out with a synthetic task and a variable data size. Figure 33 presents the task migration latency between PEs at 1 hop of distance and without NoC disturbing (each NoC router takes 5 *cc* to route a packet header). After a constant task recreation latency, 2,709 *cc*, the migration latency increases linearly with the data size, approximately 500 *cc* per KB, using 32-bit NoC channels. When compared to the related works (more details in Subsection 4.2.2.2), this value corresponds to a small task migration latency.

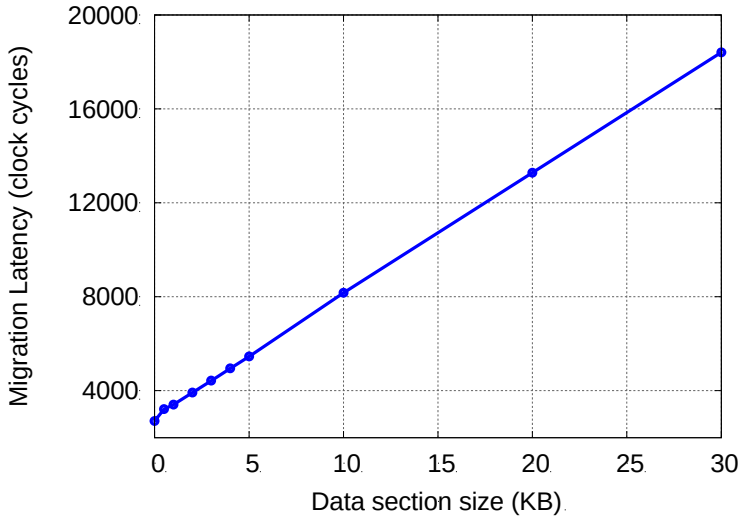


Figure 33 – Task migration latency according to the task data size, 32-bit NoC channels, 1 hop between PEs [RUA17b].

4.2.2.1 *Impact of the Task Migration in the Applications' Performance*

Figure 34 presents the migration latency considering simultaneous task migration for the MJPEG application. Each point of Figure 34 represents the latency to decode one frame with 128 bytes. The MJPEG application is modeled as a pipeline, with five tasks: *input*, *idct*, *ivlc*, *iquant*, *output*. The tasks are allocated alone in different PEs, and the task migration moves tasks to idle PEs. The curve "*without task migration*" corresponds to the minimal application latency that the platform can sustain – baseline.

The migration order arrives when the simulation reaches 400,000 *cc*. The overhead over the baseline latency measured at the 4th frame was (in *cc*) 2,282 (+4%), 7,976 (+14%), and 10,794 (+19%), for 1, 2 and 3 migrations, respectively. The latency reduction at the 5th frame comes due the message buffering caused by the blocking of the tasks, after the migration these buffered messages are consumed with high throughput. The application latency is restored one frame later.

The overhead over the application execution time, considering only the first *11 frames* of Figure 34, was 0.05% (600 *cc* of penalty) for 1 migration, and 0.6% for 2, and 3 migrations. The overhead for 2 and 3 migrations is not cumulative due to the parallel execution of the migration in

different PEs, and the inter-task message synchronization that masks the migration overhead due to its on-demand behavior to deliveries the pending messages. For many received frames, and frame sizes, this overhead is even smaller.

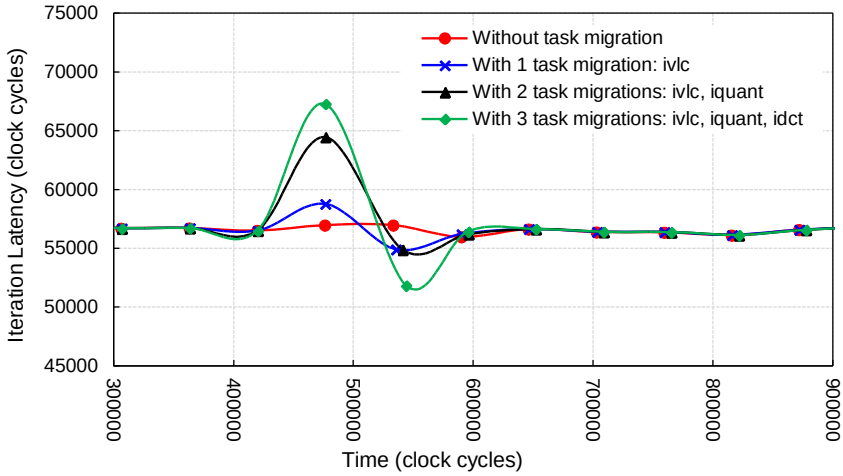


Figure 34 – MJPEG frame decoding latency for simultaneous task migrations [RUA17b].

Figure 35 explores the proposed task migration targeting QoS. The MJPEG application is mapped dynamically on the system. At time **A**, a disturbing application is mapped increasing the frame decoding latency. At time **B,** a QoS heuristic (out of the current scope) fires 3 concurrent tasks migrations (all tasks belonging to MJPEG application). The migration process finishes at time **C**, with the migrated tasks moved to free PEs, this restores the MJPEG latency to the baseline value. The MJPEG execution finishes at time **E** with task migrations. Without disturbing, the MJPEG finishes at time **D**, and with disturbing but without task migration it finishes at time **F**. The migration overhead over the application was 6,152 *cc* for 3 simultaneous task migrations (besides the speedup of 27% in the execution time).

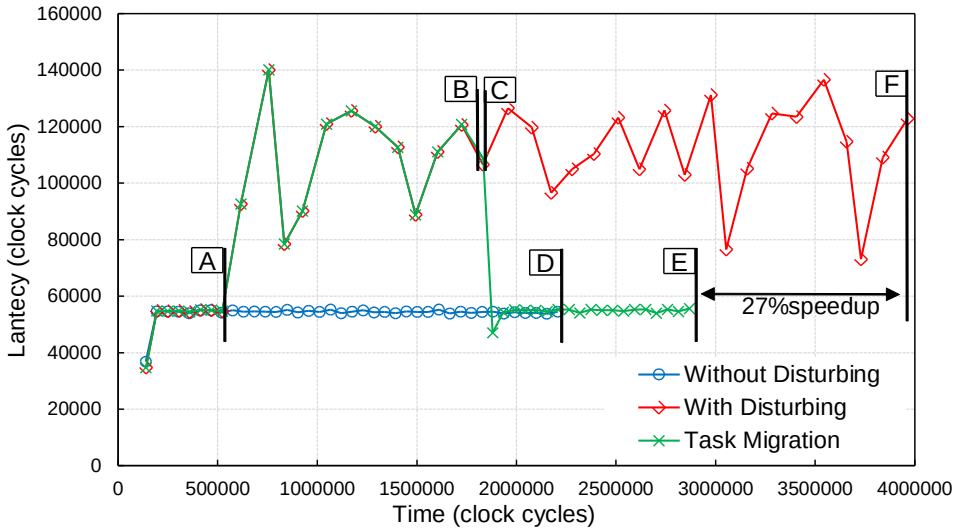


Figure 35 – Task migration applied for Quality of Service at MJPEG application [RUA17b].

4.2.2.2 *Latency Comparison with the State-of-the-Art*

Table 9 compares the proposed migration latency with results reported in the literature. This is a approximate comparison since the works' architectures are not exactly the same. El-Antably et al. [ANT15] evaluate the migration overhead using a synthetic application with three tasks: generator, processing, and consumer. The migration overhead is a function of the task size. The minimum overhead achieved was 530,923 µs for a task size of 4KB, increasing at a rate of 42.2 µs for each 4KB. Our proposal achieves an overhead of 4,945 *cc* based on Figure 33 (equal to 49.45 µs @ 100MHz) for the same data size, at an increasing rate of 500 *cc* (5 µs) per KB.

Table 9 - Comparison of the migration latency against the state-of-the-art works [RUA17b].

Proposal	NoC details	OS	Benchmark	Migration Latency
[MUN15]	Theoretical proposal			
[ANT15]	3-D torus	DNA-OS	Synthetic Apps	530,923 µs
[FU13]	N/A	uC/OS-II	Matrix Mult.	10,457 *cc*
[CAN12]	2D mesh	in-house	MJPEG	25,000 *cc*
[JAH11]	2D mesh 4 Gbit/s per link	CARAT middleware	x264 / 7Zip / robotic app.	106,505 *cc* (average value)
[SAI08]	2D mesh	HS-Scale OS	MJPEG	919,450 *cc*
This proposal	***2D mesh 3.2 Gbit/s per link***	***in-house***	***MJPEG***	***600-6,152 cc***

Fu et al. [FU13] use a matrix multiplication benchmark to evaluate its migration protocol. NoC details are not available. The obtained application overhead was 10,457 *cc*, and the migration latency for one task was in average 7,500 *cc*. Our proposal achieves an MJPEG application overhead of 600 *cc* for one task migration. Fu et al. do not allow more than one simultaneous task migration and use task replication.

Cannella et al. [CAN12] adopt as the benchmark a Sobel Filter and an MJPEG decoder. The MJPEG's task migration latency takes 25,000 *cc* in that work. Our overhead for MJPEG was, in the worst-case, equivalent to 24% to the one obtained in that work. Cannella et al. also employ task replication and require instrumentation in the task source code.

Jahn et al. [JAH11] adopt three benchmarks: x264, 7Zip, and an embedded-systems robotic application. With the CARAT task migration mechanism, the migration latency may vary because the migration transfers the memory section dynamically according to the task behavior. For this reason, the application benchmarks presented different migration latency overheads. A video application, x264, resulted in a latency overhead 303,118 *cc*.

Saint et al. [SAI08] use an MJPEG benchmark. The Authors employ a task migration in one task of the MJPEG application. The task migration process required 131.35 *ms* in a MIPS R3000, corresponding to a migration latency of 919,450 *cc*. For comparison purposes, the scenario of Figure 34 detailed an overhead in the MJPEG decoding of 6,152 *cc* to migrate 3 tasks simultaneously.

4.3 Dynamical Real-Time Task Scheduler

A key feature of a complex system, such as a many-core, is the ability to support dynamic workloads. Applications may have moments of heavy computational load and can have moments of a state close to the idle, waiting, for example, an external input, as a user interaction or a message from another task. For this reason, it is necessary to allow applications to tune the computational workload, avoiding unnecessary resources allocation. *This Section presents the main contribution of this Thesis related to QoS at the computation level*. The proposal focuses on a self-adaption technique for many-core, proposing a dynamic Real-Time (RT) task scheduler that can support run-time reconfiguration of the tasks' RT constraints. This reconfiguration starts with an API that enables the application developer to characterize the RT workload of each task at different execution points. At run-time, according to the task execution, the API triggers changes in the task RT constraints. The proposed task scheduler handles these changes at run-time to fulfill the new tasks' RT constraints.

Scalability is ensured by dividing the scheduler into two hierarchical levels: S_L (*Local Scheduler*), and S_C (*Cluster Scheduler*). The S_L runs at the processor level within $S_{PE}s$, using the LST (Least Slack-Time) algorithm. The S_C runs at the cluster level, within $M_{PE}s$. The S_C receives messages from the S_Ls, informing the CPU slack-time, deadline violations, and RT changes. The S_C implements an RT adaptation heuristic, triggering task migrations according to RT reconfiguration or deadline misses.

The scheduler works according to the ODA paradigm - Figure 36. The monitoring and notification messages (CPU slack-time, deadline miss, RT changes), produced by the S_Ls, are the inputs of the S_C (observation), the S_C executes a heuristic (decision), which can trigger an adaptation by a task migration (actuation). This process is repeated along the execution of the applications.

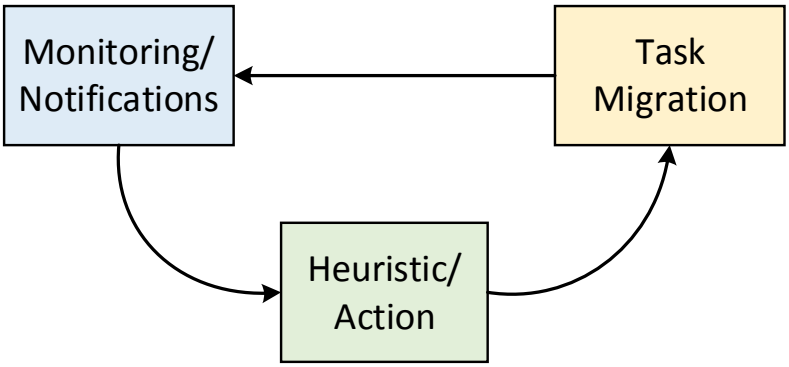


Figure 36 – Scheduler support for self-adaptation at run-time based on the ODA paradigm [RUA16a].

The task scheduler proposed in this Thesis is a dynamic scheduler, in the sense that tasks can change its real-time constraints at run-time. The task constraints are transferred to the kernel using task code annotation, a common approach found in the literature. Theodoropoulos et al. [THE13] use task code annotation that are used by a run-time manager to perform task mapping. Canella et al. [CAN12] employ task annotation to implement a task migration mechanism based on task replication.

As can be observed in Section 4.1, there is a lack of works in the literature addressing dynamic RT schedulers for large-scale many-cores. The *originality of this proposal* is related to the scheduler ability to support dynamic RT reconfigurations while it satisfies the task RT constraints. Task mapping or schedulability analysis in out of this Thesis' scope. Such techniques are explored in the literature [SIN13b][JUN14][PAR14] and can be easily combined with the proposed scheduler. Aware of the current state-of-the-art, this proposal is the first to support a run-time reconfiguration of the RT task constraints. Besides, the evaluation of the proposed scheduler is executed in a clock cycle accurate description of the many-core system.

4.3.1 Real-Time Task Model

An *m*-task of a real-time application $A=\{t_1, t_2, \ldots, t_m\}$ is modeled by a task graph G(*T*, *E*), with each vertex $t_i \in T$ representing a task and the directed edge (e_i, e_j), denoted as $e_{ij} \in E$, representing the communication between tasks t_i and t_j. Tasks communicate using non-blocking *Send* and blocking *Receive* MPI-like primitives (described in Subsection 3.2.2). A given task t_i can assume four states: *waiting*, *ready*, *running*, and *sleeping*. The ***waiting*** state implies that the task is blocked, waiting for a producer task to send it a message. The ***ready*** state means that the task already achieved its release time, and is ready to be scheduled. The ***running*** state implies that the task is executing on the CPU. Note that, with the proposition of the task scheduler, and consequently, the RT support, a new state, called ***sleeping*** is included in the scheduling of RT task only. The ***sleeping*** state means that the task already finished its execution time and its period does not end yet, so the task must be suspended.

As mentioned, the reference many-core system supports only BE tasks. This proposal adds the support for RT tasks. BE tasks do not have time bounds and explore the slack-time of RT tasks. RT tasks have *soft* temporal requirements. Figure 37 details the RT task constraints model.

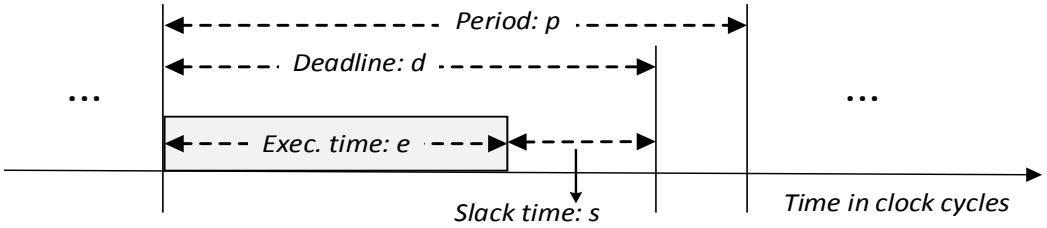


Figure 37 - RT constraints model [RUA16a].

RT tasks properties definition: RT task is a 6-tuple *{p, d, e, u, r, s}* with a period (***p***), relative deadline [LIU00] (***d***), execution time (***e***), utilization (***u***), remaining execution time (***r***), and slack-time (***s***). Utilization corresponds to $u = e * 100 / p$. The remaining execution time is the amount of time that the RT task has to finish it execution time within a given period.

To make the system aware of the constraints of a given RT task, the task must execute a system call named *RealTime*. This *RealTime* system call is the API provided to the application developer to reconfigure the RT constraints. Figure 38 presents an example of an RT task source code that configures the RT constraints (*d*, *p*, and *e*) dynamically. At line 3, the RT task *A* calls *RealTime*,

notifying the kernel about its RT constraints. Lines 4-8 execute processing code. Due to the RT constraints configuration, the scheduler can execute *A* according to the predefined RT requirements. Next, at line 10 the task calls *RealTime* again to notify the scheduler about its new constraints. The code between lines 11-15 executes according to these new constraints.

Note that the RT reconfiguration can be called in any task code point allowing the task can change its period (characterizing an aperiodic behavior), deadline, and/or execution time. Such behavior, where an RT task can change its RT constraints at run-time, is typical in real scenarios. For example, a voice recognition application. The application can assume two workloads: the first one is a listening state, where the application is waiting for the user to pronounce some sound, this state requires a moderate RT workload. The second one is the recognition state, where the task uses voice recognition algorithms. In this case, the application tasks can configure two workloads, with different RT requirements. Another example, is an image decoding application, according to the decoding frame properties, it can dynamically increase or decrease its RT constraints to decode the frame in a specified time. This feature of dynamic RT configuration help to not waste resources, avoiding the system always to assume the WCET scenario.

It is important to mention that the time used by the developer to configure the RT constraints is independent of the target archiceture, such contratins can be represented by timming parameters instead clock cycles. The platform can perform a simple conversion taking in consideration the clock cycle period.

To handle with inter-task dependencies this proposal assumes that an iteration of a given application defines a *hyper-period*, i.e., an RT application has all its tasks configured with the same *p*. This hyper-period can handle with inter-task dependencies because it is composed of the sum of the execution time of all application tasks and the worst-case of communication between the application's tasks.

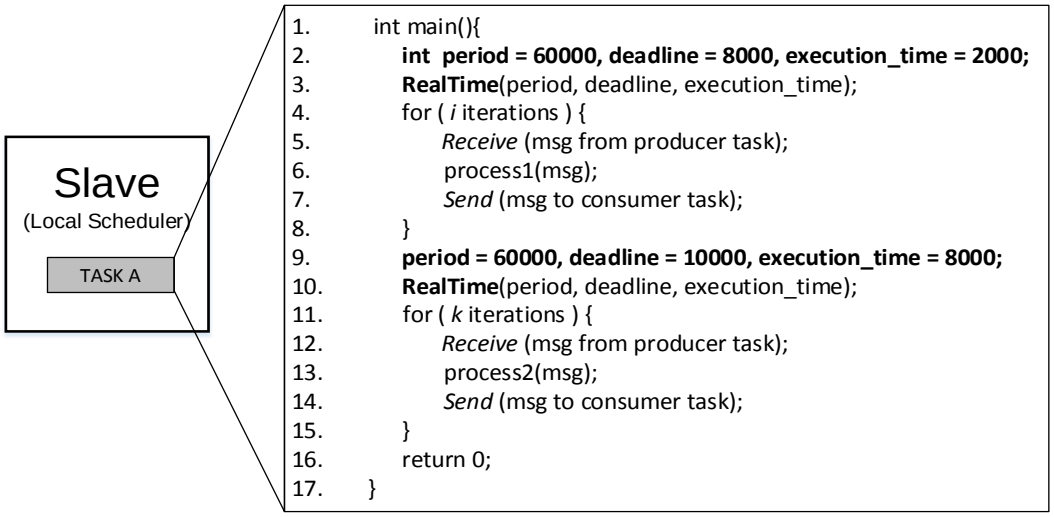


Figure 38 - Example of a task code with run-time RT configuration. It calls the *RealTime* syscall twice to configure the constraints (in lines 3 and 10) [RUA16a].

This task dependency model does not restrain the application model, accepting sequential,

parallel and pipelined applications. The designer can obtain the appropriated RT constraints running the application alone in the system (profiling phase).

4.3.2 Proposed Task Scheduler

Table 10 presents the classification based on the works of [LIU00][DAV11] for the proposed scheduler.

Table 10 - Proposed scheduler classification [RUA16a].

Criterion	Classification
Organization (Global, Partitioned)	*Hybrid (Mixes Global, Partitioned), and clustered*
Scheduling decision (Static, Dynamic)	*Dynamic*
Allocation (Clock, Table, Priority)	*Dynamic Priority-driven*
Migration (Job level, Task level)	*Task-level Migration*
Processor Number (Uni.,Multi.)	*Multiprocessor (on chip)*
Preemption (Yes, No)	*Yes – Priority based on task's slack time*
Supported task	*Periodic, aperiodic, sporadic*
Real-time (Hard, Soft)	*Soft real-time*

The S_Ls send messages to S_C. The messages are: (*i*) *CPU slack-time monitoring*; (*ii*) *deadline miss*; (*iii*) *RT change*. Messages (*ii)* and (*iii)* are reactive messages, sent when a task misses a deadline, or when a task calls the *RealTime* syscall, respectively. The slack-time messages are generated periodically.

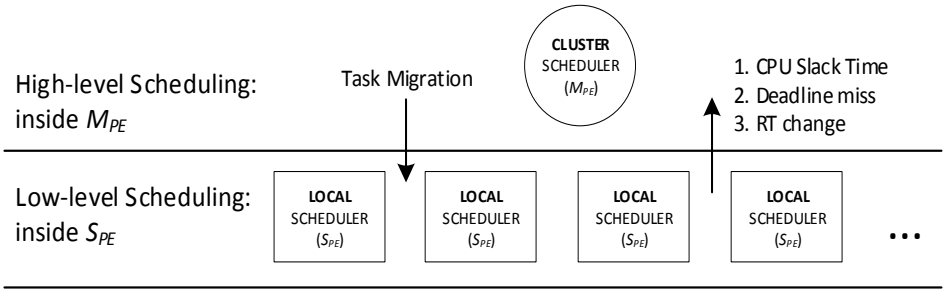


Figure 39 – Hierarchical scheduler organization [RUA16a].

The CPU Slack-Time Monitoring (STM) provides to the S_c the actual slack-time of each S_{PE}*'s* CPU. The STM has a hardware/software implementation at each S_{PE}. The hardware part corresponds to a timer, which generates an interruption to the kernel according to a monitoring window. This monitoring window is configurable at design-time and can be adjusted to provide a tradeoff between NoC communication load and the STM update frequency. The software part corresponds to a slack-time counter and a monitoring interruption handler function. When the STM timer interrupts the kernel, the kernel achieves the amount of time that CPU stay in idle, computes a percentage of this time over the monitoring window, sends a message with the current slack-time to S_C, and reset the hardware slack-time counter.

4.3.2.1 *Cluster Scheduler (S_C)*

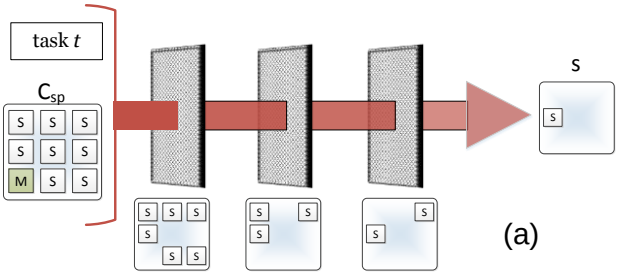
The S_C has the goal of handling the messages sent by the S_{PE}s and execute the RT adaptation if necessary. Each message has a different treatment. For the *slack-time* messages, the S_C only updates the percentage of the idle state of each S_{PE}. For *deadline miss* messages the S_C executes a heuristic called *RT_adaptation*, which can select a new processor to migrate the penalized task. Finally, for *RT change* messages, the S_C verify if the current processor of the task has enough utilization to execute the task, if not, as well as occurs for *deadline miss* message, the S_C executes the *RT_adaptation* heuristic.

The *RT_adaptation* heuristic works as a set of decision layers applied to the S_{PE} of the cluster as represented in Figure 40(a). Figure 40(b) presents the heuristic's pseudo-code. As input it receives the set $C_{SP} = \{P(c_i) - M\}$ (corresponding to set of S_{PEs} in the cluster), and the task *t* to be migrated into a given element of C_{SP}. As output the algorithm returns the selected S_{PEs}, corresponding to the new S_{PE} to receive the task *t*. If no one S_{PE} is available to receive the task, the adaptation process is suspended, the affected task will start to miss deadlines triggering the adaptation process again until an available processor can be found.

The decision functions only select the S_{PE} which fulfill the function's requirement. The following decision functions are used:

- *utilization*: selects the $S_{PE(s)}$ that have a remaining utilization enough to receive the task *t*;
- *max_avg_ST*: selects the $S_{PE(s)}$ with the largest average CPU slack-time, information obtained from STM;
- *min_RT_task:* selects the $S_{PE(s)}$ with the minimum number of RT tasks allocated to it;
- *min_abs_ST:* selects the $S_{PE(s)}$ with the largest absolute CPU slack-time measured at the last STM window;
- *min_alloc_tasks:* selects the $S_{PE(s)}$ with the minimum number of allocated tasks.

Finally, in line 6, the first S_{PE} in C_{SP} is selected to receive task *t*. Using this heuristic, the S_C takes advantage of the monitored slack-time of its slave processors (lines 2 and 4), together with the RT metrics (line 1 and 3). The information provides a trade-off between the processor's RT utilization and load balancing. After the execution of the heuristic, a task migration order is sent to the current S_{PE} of task *t,* and the task is migrated to the selected processor *s*.



(b)

RT_Adaptation **(C_{sp}, *t, s*)**

Input: C_{sp}, t
Output: s
begin
1: $C_{sp} = C_{sp} \cap$ *utilization*(C_{sp}, t)
2: $C_{sp} = C_{sp} \cap$ *max_avg_ST*(C_{sp})
3: $C_{sp} = C_{sp} \cap$ *min_RT_task*(C_{sp})
4: $C_{sp} = C_{sp} \cap$ *min_abs_ST*(C_{sp})
5: $C_{sp} = C_{sp} \cap$ *min_alloc_tasks*(C_{sp})
6: s = *first*(C_{sp})
end

Figure 40 - (a) layered decision flow. (b) *RT_adaptation* heuristic [RUA16a].

Note that S_C in this proposal individually performs the role to make RT adaptations, covering self-adaptation at computation level, the S_C role was integrated into a unified self-adaptive QoS management that will be presented in Chapter 6.

4.3.2.2 *Local Scheduler (S_L)*

Assuming the task model in Subsection 4.3.1, the S_L executes as a conventional LST scheduler [LIU00]. The LST algorithm was chosen because it has been proved optimal for single cores, and due to its support to deadlines different from the period, which is not supported in EDF (Earliest Deadline First). Note that the multiprocessor scheduling problem can be reduced to the single core problem, due the presence of the S_C, which acts as a high-level manager, migrating a task when the S_L that is not able to provide its RT requirements.

The RT tasks have scheduling priority over BE tasks. RT tasks are scheduled according to its least slack-time priority. If there are two or more RT tasks with the same slack-time, a round-robin algorithm is used to select the next scheduled task. BE tasks are scheduled only by the round-robin algorithm.

As the system uses an MPI-like communication API, some tasks (either BE or RT) can be in a *waiting* state. In this state, BE tasks are blocked. However, when an RT task goes to the *waiting* state, the S_C handles the RT task as a sleeping task, i.e., the scheduler verifies the end of task period, but do not update its remaining execution time neither schedules the task. When the task receives the requested message, the kernel changes the state of the task to ready and calls the S_L. The scheduler then updates the slack-time and the remaining execution time for all its RT ready tasks, scheduling the task using the LST priority.

Most schedulers use a fixed scheduler timer (S_T), or *quantum*, to schedule the tasks [LIU00][PAR14] (for example, EDF uses fixed time slices). This *quantum* is the interval between the scheduler calls. The proposed S_L adopts a variable S_T. Setting the appropriate S_T is challenging, because it may induce deadline misses caused by excessive scheduler executions. The proposed method to compute S_T is executed after selecting a given task to be scheduled t_s. The goal is to let t_s run, minimizing scheduler interruptions but without compromising other RT tasks. Let ψ be the set of RT tasks allocated into a given S_{PE}, except by t_s.

The S_T value is computed applying the following steps:

1. Selection of the *first end of period* for all *sleeping* and *waiting* tasks $s \in \psi$, using Equation 1:

$$S_{tc1} = \begin{cases} C_t - first\_end\_of\_perido(s_1 \ldots s_n), & n>0 \\ t_{s(r)}, & n=0 \end{cases} \quad (1)$$

   where C_t is the current system time, and n is the number of tasks $s \in \psi$. The value S_{tc1} ensures that the scheduler will be called at the first end of period of a task $s_i \in \psi$. The S_{tc1} value ensures a scheduler call to awake a sleeping task or to verify if a waiting task missed a deadline. The default value, if n=0, is the remaining execution time of t_s, which is stated by $t_{s(r)}$.

2. Selection of the minimum slack-time: *minST()* for all ready tasks $r \in \psi$, using Equation 2:

$$S_{tc2} = \begin{cases} minST(r_1 \ldots r_n), & n > 0 \\ t_{s(r)}, & n = 0 \end{cases} \quad (2)$$

where n is the number of tasks $r \in \psi$. The S_{tc2} value ensures the execution of t_s up to the end of the smallest slack-time of a task $r_i \in \psi$.

3. Selection of the scheduler timer S_T, using the Equation 3:

$$S_T = \min(S_{tc1}, S_{tc2}, t_{s(r)}) \quad (3)$$

If $t_{s(r)}$ is smaller than the previously computed values, it is adopted as the *quantum* value. After selecting S_T, tasks t_s start their execution using the S_T value as *quantum*.

4.3.3 Results

Results were obtained using a clock-cycle accurate RTL SystemC model of the reference many-core. Results use two latency metrics:

- *Task iteration latency*, time to execute a task iteration, which can e.g. be a loop.
- *Application iteration latency*, time for an application to execute its hyper-period.

4.3.3.1 *CPU Slack-Time Monitoring*

The evaluation of the STM includes: *accuracy* and *performance overhead*. The accuracy evaluation employs an 8x8 dimension many-core, divided into four 4x4 clusters. To estimate the S_{PE}s slack-time, the S_{PE}s received only RT tasks. Figure 41(a) presents the annotated *utilization* for each S_{PE} (%). Figure 41(b) presents the monitored *slack-time* achieved from STM (%). It is possible to note that the monitored slack-time is in practice the remaining utilization of Figure 41(a), with the sum of S_{PE}s utilization with the monitored slack-time reaching to 99%. The remaining 1% is related to kernel overheads. *Such results demonstrate the accuracy of the monitored slack-time.*

9	11	34	7	81	77	82	10
6	29	15	10	57	5	62	14
21	55	15	44	12	25	80	24
M_{PE}	58	86	6	M_{PE}	6	12	14
16	47	12	67	69	13	66	9
50	5	10	14	29	15	14	46
9	19	9	74	81	11	12	54
M_{PE}	40	79	59	M_{PE}	27	15	46

(a) SPs utilization (%)

90	88	65	92	18	22	17	89
93	70	84	89	42	94	37	85
78	44	84	55	87	74	19	75
M_{PE}	41	13	93	CM	93	87	85
83	52	87	32	30	86	33	90
49	94	89	85	70	84	85	53
90	80	90	25	18	88	87	45
M_{PE}	59	20	40	CM	72	84	53

(b) Monitored SPs slack-time (%)

Figure 41 – (a) S_{PEs} utilization using RT tasks. (b) Monitored SPs slack-time. Each square with a number represents an S_{PE} [RUA16a].

To evaluate the performance overhead due to STM, a 12x12 many-core divided into nine 4x4 clusters was used in experiments, running a mix of RT and BE applications. The monitoring window was set to 10 ms. Figure 42 presents the STM overhead for each S_{PE}. This overhead is related to the time required to handle the STM interruption and to send the slack-time message to the S_C. As can

be observed, the overhead in most S_{PEs} falls between 100 and 150 *clock cycles* (cc), with an average of 132 cc. There are a few large values, which can be explained by NoC congestion, forcing the packet to wait for the router to be released. The overhead in the M_{PE} to handle the STM packets was 1620 cc (only software execution). *Such result shows the small penalty to monitor the slack time.*

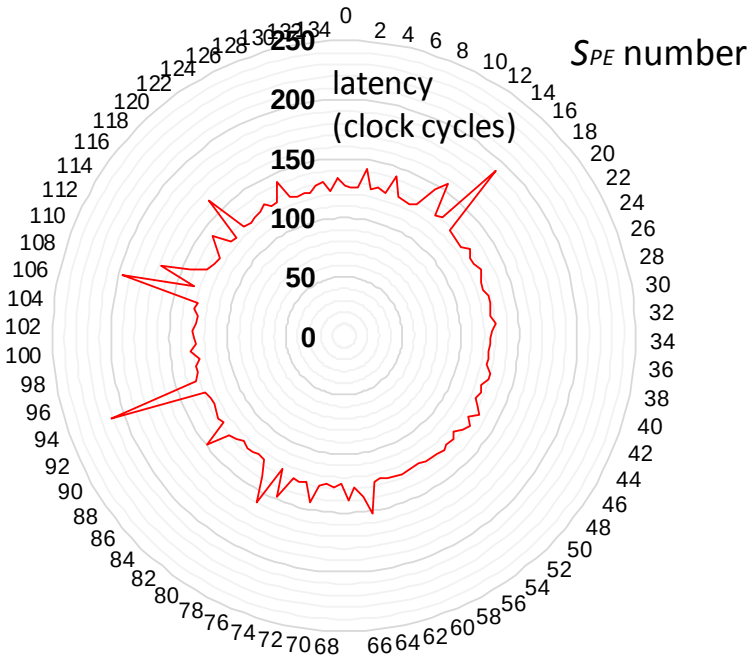


Figure 42 – STM overhead for S_{PEs} in a 12x12 many-core [RUA16a].

Different STM windows were evaluated: 1, 2, 5 and 10 ms. The cost of handling a monitoring message by S_C does not change with the STM window. Reducing the monitoring window may reduce the time to adapt the system, at the cost of increased processing in M_{PE}, due to the larger number of packets to deal. The monitoring window is a design choice, enabling to establish a trade-off between reaction time and performance overhead in the manager processors.

4.3.3.2 *Real-Time Adaptation Support*

This Subsection evaluates the RT adaptation support, observing the scheduler behavior when a *RealTime* syscall occurs. Figure 43 presents the CPU utilization for a given S_{PE} running two RT tasks: *t1* and *t2*. At the beginning of the execution, both tasks configure its RT constraints: $t1_{(u)}$=20%, $t2_{(u)}$=30%. Near 52 ms, *t2* changes its RT constraints, configuring $t2_{(u)}$=65%. It is possible to note that after the second *RealTime*, *t2* executes for a longer period, corresponding to the utilization configured in the second *RealTime* call.

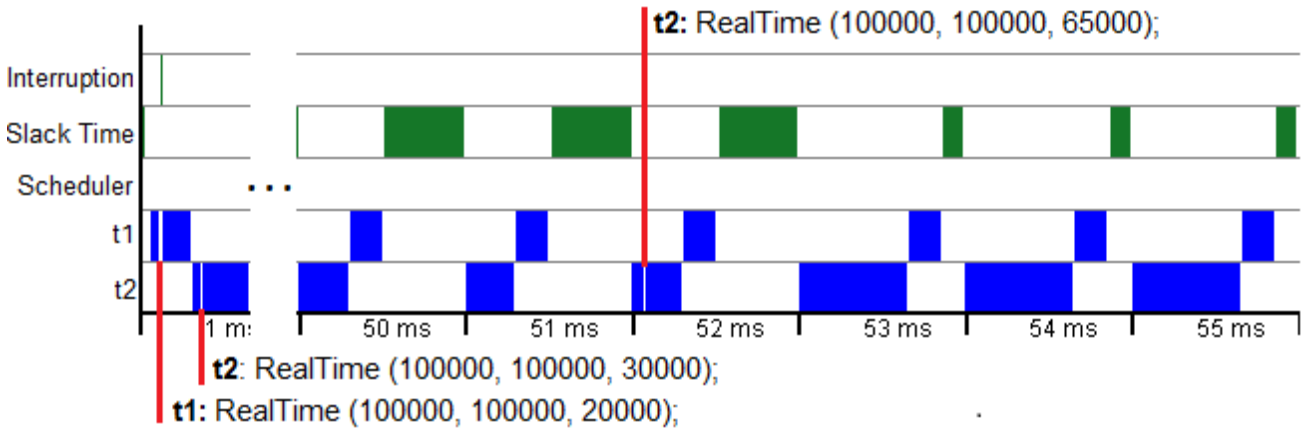


Figure 43 - Change in the CPU time utilization during an RT adaptation (rectangles represent the CPU utilization) [RUA16a].

Figure 44 presents the *task iteration latency* for *t1* and *t2*, considering the scenario presented in Figure 43. It is possible to note that the *t1* latency is not affected when the *t2* workload increases, demonstrating the capability of the S_L to preserve the RT constraints in a shared CPU scenario. The small peak near 52 ms observed in the graph occurs due the RT adaptation process. Figure 44(b) shows that after *t2* request more CPU resources the latency decreases in the same proportion.

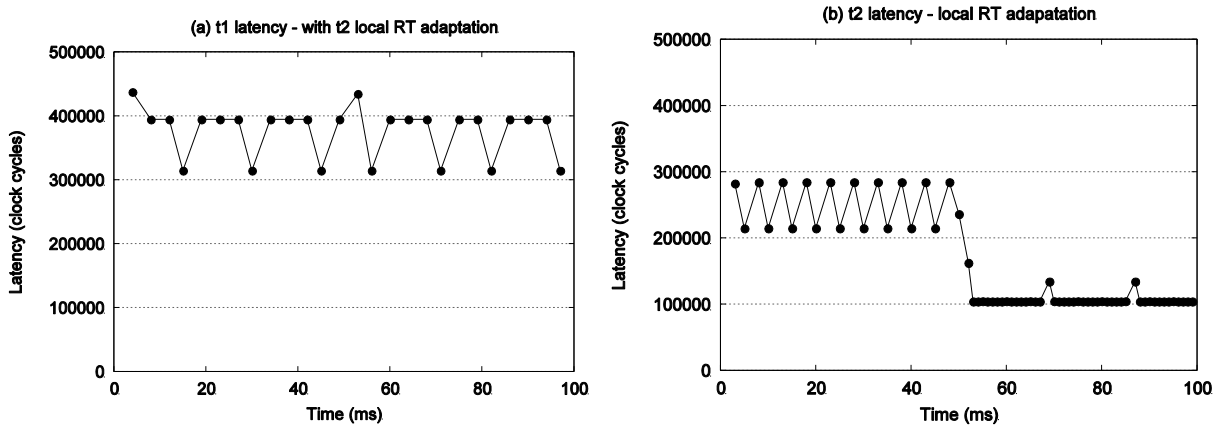


Figure 44 - Task iteration latency change during an RT adaptation. (a) t1 latency. (b) t2 latency [RUA16a].

4.3.3.3 *Real-Time Adaptation with Task Migration*

This Subsection presents an RT adaptation scenario using task migration (Figure 45). The same scenario of the previous Section is used. However, the second RT configuration of *t2* exceeds the S_{PE}'s CPU utilization: $t2_{(u)}$=85%. This utilization, plus $t1_{(u)}$, would result in a CPU utilization equal to 105%. When S_C receives the *RT change* message related to the second *t2* RT change, it detects that the CPU utilization in the S_{PE} is above 100%, and executes the *RT_adaptation* (Figure 45(b)) to select an S_{PE} with enough remaining utilization (neighbor PE at 1 hop of distance).

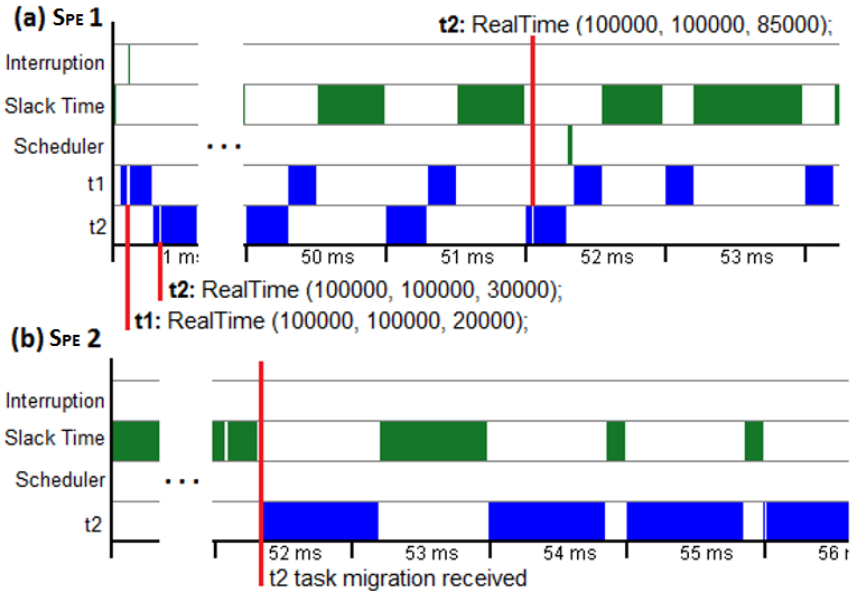


Figure 45 - Change of the CPU time occupation during an RT adaptation with task migration. Task t2 start to execute in S_{PE} 2 when the RT constraint changes [RUA16a].

Figure 46(a) and Figure 46(b) present the *task iteration latency* for *t1* and *t2*, considering the scenario presented in Figure 45. It is possible to observe the negligible impact of RT adaptation even with task migrations taking place. Observing the chart of Figure 46(a) is possible to note that near 52 ms *t2* call the second *RealTime*, inducing a task migration from S_{PE} *1* to S_{PE} *2* (Figure 46 (b)). The total time between the start of RT adaptation until the end of task migration was 8906 cc, with 2651 cc (29.7%) required to the RT adaptation process, and 6255 cc (70.3%) required to the task migration protocol.

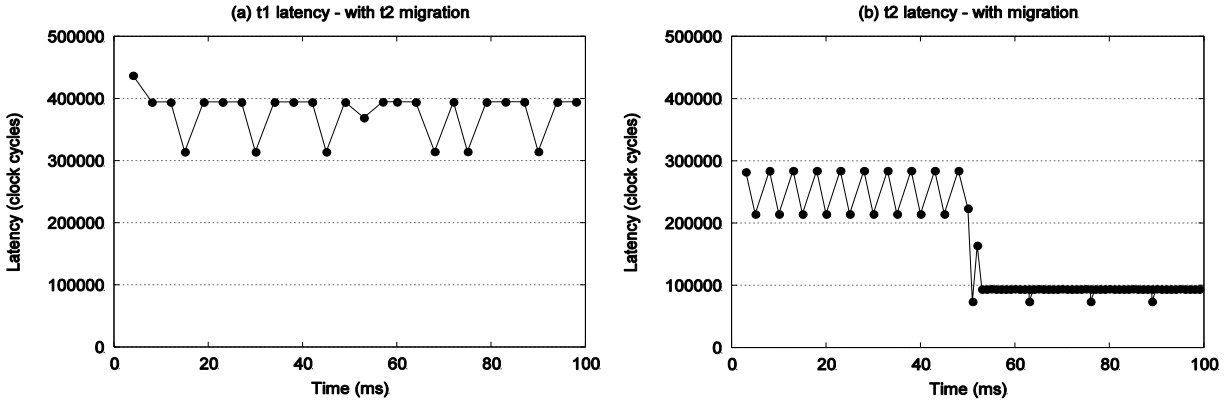


Figure 46 - t1(a) and t2(b) task iteration latency during an RT adaptation with task migration [RUA16a].

Note that in this experiment *t1* is not impacted by the *t2* migration. Besides the low overhead of the task migration protocol, another factor that contributes to this non interfecerence is due *t1* is in *sleeping* state when the order to migrate *t2* arrives at S_{PE} *1*. The protocol is fast enough to ensure *t2* migration until the next job release of *t1*.

4.3.3.4 *Real-time Applications Execution Time Evaluation*

This Subsection evaluates two RT applications: DTW (computation intensive, six tasks) and MPEG (communication intensive, five tasks). These applications – named *target applications*, were evaluated in the presence of disturbing applications (RT and BE). The goal is to observe the scheduler behavior with multiple tasks allocated in the same S_{PE} and the impact of the disturbing applications over the target applications.

Initially, the target applications execute alone in the system aiming to collect the reference RT constraints for its tasks (profiling step). Next, new simulations were performed aiming to insert RT interference over the target applications. The functionality of the target and disturbing applications is not important in this scenario. The goal is to observe how the insertion of either RT or BE interferences affect the RT quality provided by the proposed task scheduler over the target applications. Figure 47 depicts the results for DTW application. Each bar shows the results over different interferences. In the *minimal* bar, the disturbing tasks are only communicating, not presenting significative CPU interference over the DTW tasks. For the bars from 10% RT to 90% RT, the disturbing applications are set to induce interference by setting RT constraints through the *RealTime* syscall. The percentage represents the amount of the remaining CPU utilization left by the target tasks, which was explored by the disturbing task. Thus, the available time to execute disturbing tasks, $t_{disturb}$, corresponds to $R_t(p)$- $R_t(e)$, where R_t is the target task. The experiments vary $t_{disturb}$ from 10% (0.1* $t_{disturb}$) to 90% (0.9 * $t_{disturb}$).

Figure 47(a) presents the DTW average *application iteration latency*. The first column presents the *minimal* latency, next columns present the interference of an RT disturbing application varying $t_{disturb}$ from 10% (10% RT) to 90% (90% RT), the column "BE" corresponds to a BE application interference with $t_{disturb}$ equal to 100%. The last column presents the latency when a round-robin scheduler (i.e. without RT support) is used. Is possible to observe that for all RT disturbing scenarios the DTW latency close to the minimal latency (36220.03 cc.), with an average latency increase of 2%, and a standard deviation of 316.4 cc. Such results demonstrate the scheduling ability to preserve the RT application constraints even with high RT resource sharing load. BE disturbing applications do not influence the latency values. Disabling the RT support the latency increase 97.13% compared with the minimal latency, demonstrating the importance of the RT scheduler to meet deadline constraints on this scenario. The deadline miss rate for DTW with RT interferences was 0.66%, and for the disturbing application (which is also an RT application) was in average 2.7%. Figure 47(b) presents the DTW execution time. The results are similar to the latency results. The additional column "*minimal–no STM*", corresponds to the scenario to obtain the minimal latency, but disabling the STM. Note the negligible impact of STM monitoring, increasing the application execution time by 0.4% (adopting, as worst-case, STM window of 5 ms).

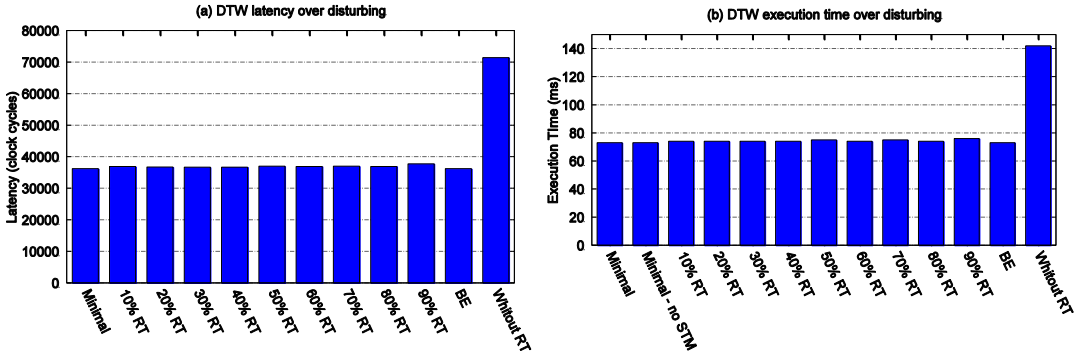


Figure 47 - (a) DTW application latency over disturbing. (b) DTW execution time over disturbing [RUA16a].

The *application iteration latency* observed in the MPEG application is similar to the DTW application. All latencies with RT disturbing remain close to the latency of the minimal scenario (55358.5 cc.), with an average latency increase of 0.5% and standard deviation equal to 181.1 cc. The frame rate achieved in the minimal scenario was 178 frames per 100ms. With RT disturbing applications, the frame rate has presented a maximum of 2 frames decrease compared to the minimal scenario. Disabling the RT support, the frame rate drops to 81 frames per 100ms. The impact of STM monitoring in the MPEG application was 0.07% (STM windows of 5 ms). The deadline miss rate for MPEG with RT disturbing application was 0.18%, and for the disturbing application was equal to 2.1%.

The deadline miss difference between the disturbing application and the RT application occurs because the disturbing application is computationally intensive, which makes such application more sensitive during RT scheduling. This same observation can be used to explain the difference of deadline misses between DTW and MPEG.

4.3.3.5 *Real-time Applications Latency Time Evaluation*

This Subsection evaluates how the latency of the RT application is affected faced to run-time events of interferences. The DTW application is mapped on the system and starts its execution at the beginning of the simulation, the MPEG start at 15ms. Three BE applications are inserted at 25ms, 35ms, and 45ms.

The graph of Figure 48(left) represents the latency for each MPEG iteration (frame decoding). This latency is collected at the last MPEG task (*output*).

As can be observed, the RT support scenario provides a reduction of 209.3% of the iteration latency compared with the scenario without RT support. Is possible to observe the high degree of interference caused by BE application in this experiment. The RT scenario ensures a latency of 45.8% higher in comparison to the optimal scenario.

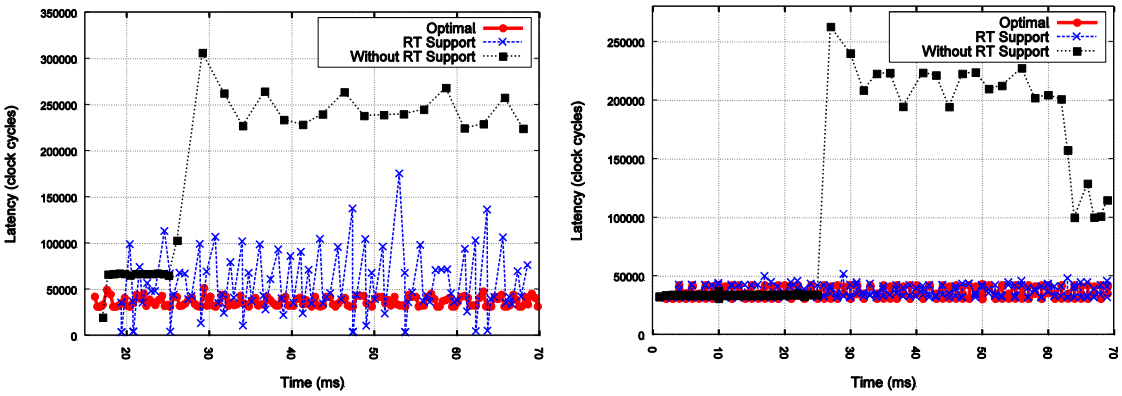


Figure 48 – (left) MPEG iteration latency; (right) DTW iteration latency [RUA15b].

The graph presented in Figure 48(right) addresses the DTW iteration latency. The effect of the RT support for DTW application is easily observed. The scenario with RT support can maintain the application requirements even with the load of the BE applications. The average latency increases only 1.43% comparing the RT support against the optimal scenario. The average latency reduction comparing the RT support against without RT is 97.7%.

As shown in Figure 48, the proposed scheduler can sustain the RT constraints of tasks. The high impact of disturbing over the MPEG application occurs because this application is highly communicating, which allows that the disturbing caused by BE applications can be observed. In the case of DTW, which is computation intensive, the scheduler can prevent deadline misses even in a high computation disturbing. As the scope of the scheduler is, essentially, computation resource management, it is expected the behavior shown in both graphs. This analysis emphasizes the need to complement a real-time scheduler with a run-time QoS mechanism at communication level.

4.4 Energy Management combined to Real-Time Task Scheduling

The proposed task scheduler was integrated into a run-time energy management technique [MAR17a][MAR17b]. The goal of the work is to propose an adaptive run-time energy management designed to support soft RT applications, called RT-REM. The energy management observes the energy per PE and exploits the slack-time of RT application to guide task mapping, task migration and to apply DVFS while meeting the application's RT constraints.

4.4.1 DVFS Design

The DVFS is applied at the computational resources (CPU and local memory). To support DVFS, the original DMNI design presented in Section 3.4 was modified, as overviewed in Figure 49. The DMNI can support two different frequency domains between the NoC and the PE. The main goal of such design is to enable PEs to work at different frequencies while the NoC transmits packets using the nominal frequency.

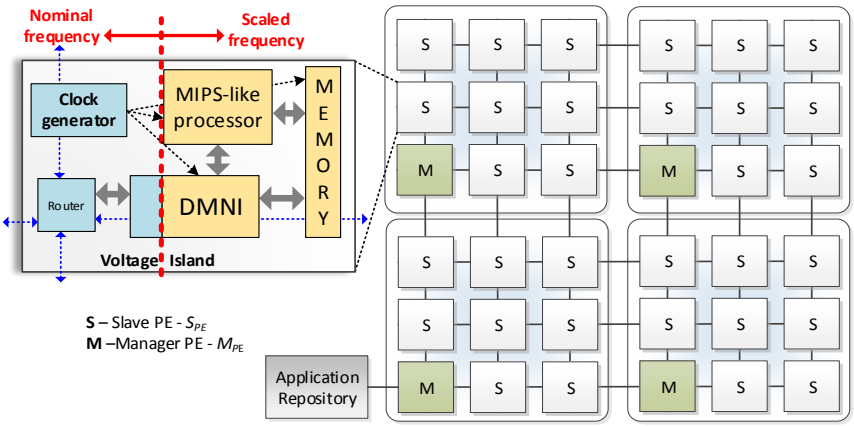


Figure 49 - A 6x6 instance of the reference many-core system, with four 3x3 clusters [MAR17b].

The work creates a set of voltage and frequency levels that can be applied at run-time, each level is called *vf-pairs* (from voltage and frequency pairs): (1.1V, 4ns), (1.0V, 4.5ns), (0.9V, 5.5ns). Such values were obtained from the *Liberty* files and timing analyses of the components netlist.

4.4.2 Power/Energy Characterization and Energy Profiling

Energy monitors extract the energy during time windows from the processor, memory, and router. These energies values are obtained based on a characterization phase (at design time), performed to each component. The processor's characterization classifies the ISA (arithmetic, logical, shift, etc.) achieving average power (mW) and energy (pJ) for each instruction class.

The router's characterization is similar to the processor's characterization. As the router adopts wormhole PS and credit-based flow control, the packet's header reserves the channel temporally until the end of the packet. Thus, the switching activity of the router can be classified in idle (0% of activity) and active (100% of activity), with each activity class presenting its energy (dynamic and leakage values). The memory characterization is performed by the CACTI-P tool [LI11]. The CACTI-P tool enables distinct model types of memories, generating estimations as access time, silicon area, and power.

After the characterization phase, an energy profiling phase is performed considering the *vf-pairs* and the characterization step. The energy profiling enables the system developer to know the minimum and maximum energy that an S_{PE} can consume (best and worst cases). This knowledge is useful to calibrate the run-time energy management algorithms. Figure 50 presents the energy consumption for each S_{PE} component. The histograms assume three supply voltages according to the *vf-pairs*, and a sampling window equal to 1 millisecond. The histograms show the consumption of the three main modules (processor, router, and memory). When the system operates at "max energy" (processor running applications) and router transmitting data, the processor is responsible for half of the consumed energy, and roughly 30% for the memory, and 20% for the router. For "min energy" (processor and router in idle states), the parcel due to the router increases because its frequency is not reduced.

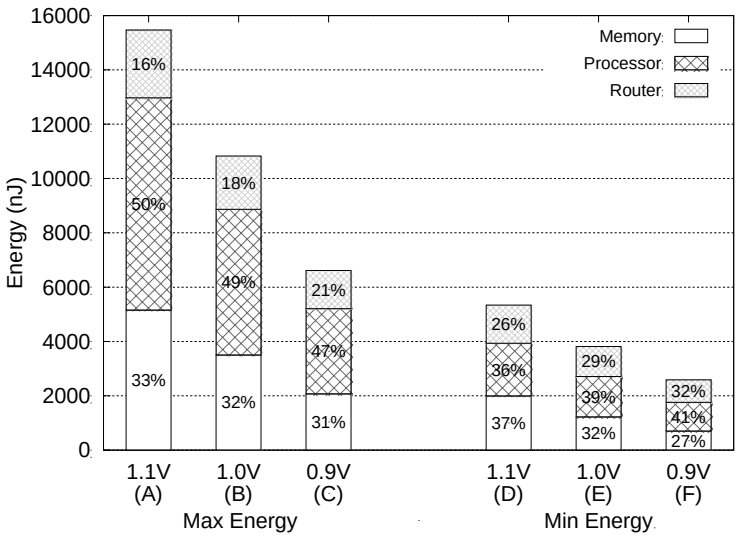


Figure 50 – Energy profiling of the PE for all voltage supplies. The total energy (y-axis) corresponds to the energy spent in a monitoring window of 1 ms [MAR17b].

4.4.3 Energy Monitoring and Estimation

Hardware and software components make the energy monitoring. At the hardware level, registers count the number of instruction classes (by monitoring the processor and the memory), and the number of flits (by monitoring the router), periodically, according to a given time window. These values are read by the kernel of the $S_{PE}s$ periodically after the end of each monitoring window. The S_{PE} uses the received data to perform estimation functions, computing the dynamic and leakage energy of the processor, memory, and router.

4.4.4 Application's Slack-time Monitoring

The RT support assumes the previously described LST task scheduler. Figure 51 details how to compute the application slack time. The $A_{RT}(p)$ is the application hyper-period. $A_{RT}(x)$ is the application execution time, corresponds to the time to execute all tasks during one $A_{RT}(p)$. This time is composed of the CPU usage and the communication overhead between tasks. $A_{RT}(s)$ is the application slack time, which corresponds to the difference between $A_{RT}(p)$ and $A_{RT}(x)$.

The reference (RT constraints) $A_{RT}(p)$ and $A_{RT}(x)$ are provided by the syscall *RealTime*. The *RealTime* is called inside each task. Note that by removing the *deadline* field from *RealTime* the proposal assumes only periodic task with their deadlines equal to the period.

The **current** $A_{RT}(p)$, $A_{RT}(x)$, and $A_{RT}(s)$ are computed at run-time by monitoring, triggered by a *syscall* inserted on the task's code, called *PeriodMonitoring*. This function sends the current time to the M_{PE} (which executes the energy management), allowing M_{PE} to measure the current $A_{RT}(x)$, $A_{RT}(s)$ and $A_{RT}(p)$ for each application, and to detect when a hyper-period is violated. This *syscall* is required in the first task (e.g., task *a* in Figure 51(b)) and the last task (e.g., task *e* in Figure 51(b)) of each application. Figure 52 presents an example of task's code calling the *RealTime* and *PeriodMonitoring*. As can be observed the *RealTime* is called once, at the beginning of the loop. The *PeriodMonitoring* is called at the end of the loop, enabling the system to measure the QoS at each

iteration.

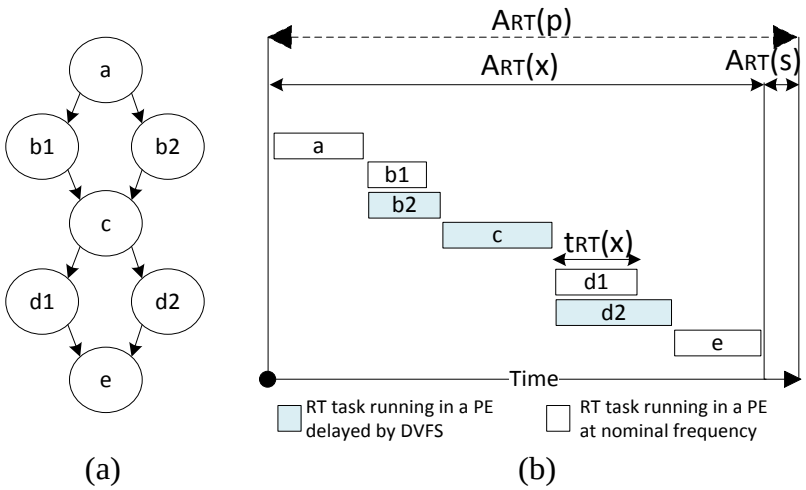


Figure 51 - Task graph of an RT application (a) and its scheduling (b) [MAR17b].

```
RT task e: code
1 int hyper_period=60000, exec_time=8000;
2 RealTime(hyper_period, exec_time);
3 for k iterations do
4   Receive(msg from d1);
5   Receive(msg from d2);
6   DoSomething();
7   PeriodMonitoring(get_nominal_tick_counter());
8 end for
```

Figure 52 - Code snippet for the last RT task of an application [MAR17b].

4.4.5 Run-time Energy Management (RT-REM)

The RT run-time energy management implemented within M_{PE}, it consists of heuristics that act during the application admission (task mapping), and at run-time according to RT constrains changes or hyper-period violations. During the application admission, the SM_{PE} redirect the application to the cluster with the lowest energy consumed, balancing the energy at the system level. During the task mapping phase, the M_{PE} of the cluster performs a task mapping heuristic based on a set of cost functions, which aim to select $S_{PE}s$ with the lower number of RT task running on it, and the minimal energy consumed. This approach balances the energy at the cluster level.

At run-time, after the task mapping, the M_{PE} detects when a given task change its RT constraints (due to a *RealTime* syscall) and executes the RT-REM of Figure 53 to set the appropriated *vf-pair* for the PEs running the task. The M_{PE} receives the new RT constraints of the task and uses it to compute the task's utilization $t_{RT}(u)$ according to line 4 of Figure 53. Two thresholds are used to guide the DVFS process based on the obtained $t_{RT}(u)$. HIGH_UTILIZATION: $t_{RT}(u)$ corresponding to more than 70% of $A_{RT}(p)$; LOW_UTILIZATION: $t_{RT}(u)$ corresponding to less than 30% of $A_{RT}(p)$. The $A_{RT}(p)$ is the constraint to meet in RT applications. As the voltage scaling introduces delays on the task's execution time, RT-REM heuristic selects *vf-pairs* based on $t_{RT}(u)$ assuming a low utilization task is less likely to generate $A_{RT}(p)$ violations than a high utilization one.

Besides RT changes, the M_{PE} can also detect hyper-period violations (from the syscall *PeriodMonitoring*), in such case, the M_{PE} reset the *vf-pair* of all application tasks to the nominal voltage.

Algorithm 3: DVFS for t_{RT} (executed at the manager PEs)

```
1   Input: t_RT, x_pos, y_pos
2   Output: send to SP(x_pos,y_pos) new settings of VF
3   Begin
4       t_RT(u) ← (t_RT(x)/ * 100) / A_RT(p)    // equation 2
5       if t_RT(u) > HIGH_UTILIZATION then
6           sendDVFS_CHANGE(x_pos,y_pos,VF_PAIR(1.1V, 4ns))
7       elsif t_RT(u) < LOW_UTILIZATION then
8           sendDVFS_CHANGE(x_pos,y_pos,VF_PAIR(0.9V, 5.5ns))
9       else
10          sendDVFS_CHANGE(x_pos,y_pos,VF_PAIR(1.0V, 4.5ns))
11      end if
12  end
```

Figure 53 – RT-REM Heuristic [MAR17b].

PEs running at 1.0V and 0.9V delays a task execution by 12.5% and 37.5%, respectively, due to the frequency reduction. Thus, an S_{PE} running a task with high utilization operates at the nominal voltage (lines 5-6) while an S_{PE} running a task with low utilization, operates at *vf-pair* (0.9V, 5.5ns), delaying the task by 37.5% (lines 7-8).

4.4.6 Results

Figure 54(a) presents the hyper-period of an RT application, the baseline execution time (black curve) and the execution time using RT-REM (red curve). The green line represents the application hyper-period. The hyper-period defined by *RealTime* for each iteration is 60K clock cycles. The distance from the baseline execution time curves to the hyper-period corresponds to the application slack-time. The baseline execution time presents a significative slack-time. Using RT-REM, after the definition of the *vfpairs*, the slack-time reduces because the S_{PE}s running the application are delayed due to the frequency reduction, with few violations in the hyper-period (smaller than 10% of the hyper-period value). Figure 54(b) plots the energy consumed by the system for the baseline and the system with RT-REM. As the simulation advances the energy savings increases, reaching 22% after executing 100 iterations of the applications.

The variability in the execution time observed in both scenarios (baseline and REM) are induced by branches of the application, kernel events, interruptions, and network congestion.

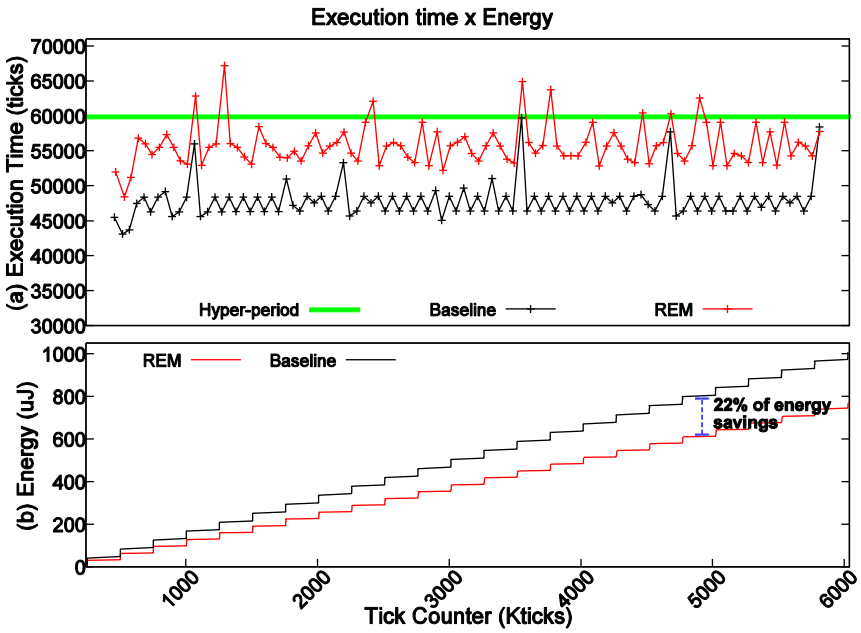


Figure 54 - Execution time and energy of an RT application with and without REM for 100 iterations [MAR17b].

Table 11 presents the evaluation of hyper-period violations and energy savings for systems having up to 144 PEs. Five many-core systems (first column), divided into different cluster sizes (second column), run the applications set. The number of RT and BE tasks to process is the same of the number of S_{PE}s of the many-core (third column). For instance, the 6x6-many-core has one SM_{PE}, three M_{PE}s, and 32 S_{PE}s. The fourth column presents the number of executed task migrations.

Table 11 – Violations of hyper-periods and energy savings of RTREM compared to the baseline system [MAR17b].

Many-core size	Cluster size	Number of tasks for each type	Number of task migrations	Violations of hyper-period	Energy savings
6x6	3x3	32	3	1,47%	15%
8x8	4x4	60	6	2,09%	18%
9x9	3x3	72	8	1,84%	20%
12x12	3x3	128	8	1,87%	18%
12x12	4x4	135	14	1.97%	18%

The fifth column of Table 11 shows the percentage of hyper-period violations (the execution of the applications in the baseline system does no present violations. The RT-REM produces a small number of hyper-period violations (< 2.1 %), with an amplitude inferior to 10% of the defined constraint. This result shows that the proposed RT-REM is suitable for soft-RT applications. For example, few hyper-period violations decoding a video frame do not affect the performance of the applications [FEG07]. The last column of Table 11 presents the energy savings, for a simulation time of 50ms. The average energy reduction observed is in average 18%. The energy saving is proportional to the amount of execution time. Therefore, for applications execution for long periods, important energy savings are expected.

4.5 Conclusions

This Chapter presented background about QoS at the computation level and the contributions of this Thesis in this context.

The proposed task migration [RUA17b] achieves results towards to demystify the high-cost of the task migration in distributed memory many-core systems, presenting a low latency protocol compared to the State-of-the-Art. Additional relevant features of the task migration proposal include: there is no need to replicate the code of the tasks; it is not necessary to modify the source code neither to add checkpoints; support to simultaneous migrations; and correct inter-task synchronization without migrating produced messages. A direction for future work is to evaluate the protocol with others self-adaptive techniques, as fault-tolerance.

The proposed dynamic task scheduler [RUA15b][RUA16a] is self-adaptive, supporting dynamic task RT constraints and CPU slack-time monitoring. The evaluation demonstrated an ability to fulfill RT applications with soft deadlines even for communication or computation intensive applications with the interference of other RT applications. The average latency increase was 2%, even assuming scenarios with 90% of CPU dedicated to disturbing RT applications. Such results demonstrate the scheduling ability to preserve the RT application constraints even with high RT resource sharing load. The STM presented a negligible impact on the execution time of application, with the worst execution time increase of 0.4%. Future works include integrating the scheduler with a task mapping and schedulability analysis algorithm.

The integration of the task scheduler to an energy management [MAR17b] reduces the consumed energy in many-core while executing soft RT applications meeting the applications' constraints. The proposed energy management executes monitoring (hyper-period slack time), decision (RT-REM heuristic), and actuation (DVFS and task mapping/migration). As shown in the results Section, the proposal is the scalable, with similar energy savings for different system sizes (from 36 up to 144 PEs), producing a small number of hyper-period violations (<2%). Future works include: (*i*) define consumption limits to cope with dark silicon issues; (*ii*) include other actuation techniques, as power gating; (*iii*) evaluate the approach for SOI technologies.

5 QOS AT THE COMMUNICATION LEVEL

This Chapter addresses the contributions of this Thesis covering QoS at the communication level. Section 5.1 reviews the state-of-the-art. Section 5.2 presents the proposition and evaluation of the Software-Defined Networking (SDN) for NoC-based many-cores. Section 5.3 presents a CS infrastructure and management based on the SDN paradigm. Section 5.4 presents the results. Section 5.5 finishes this Chapter presenting the conclusions and direction to future works.

5.1 State-of-the-Art

Table 12 presents the state-of-the-art related to QoS of communication. A common approach in the literature is to employ flow priority [KAK11][WIS11][CAR14][WAN12][JOV13][PAL12][GRO12] (third column). Flow priority enables differentiated treatment of packets according to different priorities. For instance, high priority packets can first receive the access to an output port by the arbiter or can to explore adaptive routing algorithms [CAR14]. Flow priority mechanisms are effective to offer a simple QoS support, where some deadlines may be missed, as in soft real-time applications. Most techniques that implement flow priority use Packet-Switching (PS), being susceptible to inter-packet interference and increased jitter.

The most adopted option to provide a robust QoS support is the end-to-end path reservation, employing a connection-oriented Circuit-Switching (CS) communication [LUS11][WIN11][STE12a] [PAL12][JOV13][CAR14][ZHA14][RUA15a]. CS is widely adopted in NoC-based many-cores to guarantee RT constraints at the communication level, being employed since 2004 with MANGO [BJE05] and AEthereal [GOS05] NoCs. CS communication reserves the NoC resources along the path, as links, buffers, and crossbar. It ensures that communication will not suffer traffic interference from other flows, providing guaranteed throughput, minimal latency and jitter to the communication flow. The path reservation is performed by a setup phase – which verify the resource availability and perform the reservation, and a release phase – which consists in releasing the reserved resources. The NI commonly manages the connection establishment and release phases. In summary, the differences between flow priorities and CS and is that flow priority assumes communication sharing and CS reserves the communication resources exclusively to a given flow.

There are two main approaches to implement CS. One reserves a single physical channel to a given flow [ZHA14][CAR14], where the flits are forwarded in a pipeline fashion up to the destination, without arbitration and routing overheads. The other approach is also known as Virtual Circuit Switching (VCS, or as mentioned in some works Switched Circuit Switching). Different from traditional CS, this technique allows sharing the physical channel with more than one flow. It ensures guaranteed services by isolating the flows spatially or temporally.

Table 12 – State-of-the-art about works addressing communication QoS.

Work	Focus	QoS Technique	QoS Guarantees	QoS implementation place
[KAK11]	Fault tolerance and QoS for NoC	Flow Priority using 2 physical channels	BE with soft guarantees	Router level
[WIS11]	QoS for prioritized packets	Priority-based Switch allocator	Latency of real time packets	Router level
[WAN12]	High throughput QoS-Aware router	Arbitration with dynamic congestion control and adaptive routing using multi physical channels	GS	Router level
[RUA15a]	Run-time QoS Adaptation	Run-time flow priority and CS adaptation according with monitored communicating task flow	BE and RT	Router/NI/OS level
[GRO12]	QoS topology-aware NoC	Elastic Buffers / Flow priorities/Dedicated QoS routers / VC and Preemptive Virtual Clock	BE and RT	NoC level: topology and router implementation
[JEO12]	Memory Access Sharing	Priority-based QoS-Aware memory controller	-	Memory Controller
[PAL12]	Hybrid Switching	Flow priority and CS using VC	BE and RT	Router Level
[CAR14]	Different QoS Services	Flow Priority and CS using 2 physical channels	BE and RT	Router level
[ZHA14]	Bufferless NoC with QoS guaranties	CS support and four services classes separated according to the traffic type and four priority levels managed by two stage routing algorithms	Real-time traffic QoS guarantees	Router level with IP support
[SAL11]	QoS Router	Priority-based scheduler supporting allocating VC according GS flows	BE and RT	Router level
[JOV13]	HW/SW integration to QoS control at the application level	QoS API that expose NoC QoS services of flow priorities and CS using VC at application level to openMP and ocMPI programming model	Guaranteed Services	System level, integrated in all instances
[OUY10]	QoS NoC for real-time flows	Flit scheduling (LFS) and flow control (FRS), supporting TDM connection	BE and RT	Router level
[BEY13]	HW/SW bridge infrastructure for connect NoC-based system	TDMA connections thought dedicated input ports and dedicated buffer to connection flow control in RT traffics	BE and RT	Software layer expose the QoS services to the IPs
[STE12a]	TDM NoC with QoS support	Distributed routing and TDM CS support	Only RT	Router level
[HE14]	Power Efficient QoS	Design time TDM scheduling and shortest path definition using a ILP algorithm	BE and RT	Design time
[ROS14]	DSE(Design Space Exploration) QoS framework	Includes TDMA scheduling based in the worst case communication analyses and bus bandwidth	Real-time RT	Bus-based system
[EJA13]	NoC with CS support	SDM (Spatial Division Multiplexing) with 1, 2 or 4 sub-channels and sub-networks.	BE and RT	Router and NI level
[LUS11]	QoS for RT applications	NoC with PS + CS using SDM up to 7 physical channels	BE and RT	Router level
[FAN11]	QoS for NoC routers	Dynamic VC allocation and NI flow injection control	BE and RT	Router and NI level
[WIN11]	Run-time VC allocation	A NoC Manager unit allocates at run-time guaranteed service VCs	BE and RT	Router and NoC level – global management
[PAL12]	Hybrid Switching	Two VC one for PS and other for CS managed by NI	BE and RT	Router and NI level
[SAM14]	Guaranteed Services based in a connection oriented technique	Flit-to-flit routing based in reservation and routing management table that uses the flit ID	BE and RT	Router level
[MAT14]	Multi-switching in NoCs	Router change two CS modes according with the path wire timing delay	Real-time flows	Router level
[SEI14]	Elastic buffer architecture for NoCs	Elastic Buffer with VC support	BE	Router level
[YAO14]	QoS support in bufferless NoC	Flit with high priorities are less deflected and not throttled	Soft guarantees	Router level with IP support
[HEI12]	High management communication QoS	Cluster regions are created to limit the number VC into the region	BE and RT	OS - Dedicated Managmnt. Unit

VCS is more flexible in terms of resource utilization than conventional CS since a single physical channel can be divided into virtual channels (VC) or virtual links. The VC separation is performed by flow control techniques applied during the channel allocation, and by different buffer organizations [PAS08]. VC implement blocking prevention using techniques for temporal link scheduling applied in routers and NI. Its major benefit is it higher resource utilization due to the flit interleaving of different packets in the same physical link. The VC technique can also be used as an alternative to avoid packet blocking in the wormhole flow control [PAS08]. As drawbacks, VC-based CS have increased complexity and power consumption against conventional CS [YOO13] due to the necessity of multiplexing of the physical channels over the time.

An option to implement VC flow control is to use one buffer for each VC [SAL11][PAL12], with VC used to BE flows and the others to GS flows. This approach has a high silicon cost due to the buffer size increase and requires complex scheduling implementation to avoid conflicts between different scheduling along the path [PAS08]. Another alternative to implementing the VC flow control is to assign the entire physical channel to a single buffer, and the VC flow control is performed by the Time Division Multiplexing (TDM) technique [LER05][OUY10][STE12a][BEY13] [HE14].

TDM is typically used as a VCS technique in NoC and buses. Its purpose consists in to divide the channel bandwidth into time slots. Each VCS connection receives an exclusive time slot according to its bandwidth requirements. TDM also reduces the buffer size due to the consecutive time slot allocation in the routers [LER05]. The flits that are crossing a path will be scheduled to not stall into the routers, and thus not requiring buffering. However, TDM-based designs suffer from high power demand and routers' complexity [LER05].

An alternative to TDM is Spatial Division Multiplexing (SDM). In SDM [LER05][LUS11][EJA13] the channel sharing occurs spatially instead temporally. SDM technique divides the wires between routers' channels, creating sub-physical channels at run-time. This enables to establish and to release a connection with a lower management effort when compared with TDM because in SDM the routers' configurations are performed at the beginning of the connection and remain active until its release, and in TDM the router configuration (TDM wheel) needs to be updated at each time slice. The major problem of the SDM is the switch complexity due the necessity to support the dynamic reconfiguration of wires [LER05].

Recent works showed the benefits to adopts a simpler alternative to support CS by using multiples instances of simple NoCs. Such design creates Multiples Physical Networks (MPN) [EJA13][LIU15][YOO13]. Authors [EJA13][LIU15] compare MPN to SDM designs. With an equivalent bandwidth implementation, MPN increases the area following an $O(n)$ complexity, while SDM increases $O(n^2)$. The path delay increases with the number of sub-channels (SDM) with a complexity $O(n)$, while remains unchangeable for MPN. The work of Yoon et al. [YOO13] presents a comparison of MPN and VC. They conclude that MPN presents a better area scalability and critical path flexibility to use in DVFS, also MPNs scale better regarding power dissipation regarding new technology nodes.

For distributed traffic patterns, VCs have a high sustained throughput and latency. However, when the traffic generates hotspots, MPNs have a better throughput.

Other works also provide QoS with CS with differentiated proposals: mixing multiple channels with SDM [LUS11], MPN with TDM [EJA13], TDM with SDM [LUS12], the IDMA technique proposed by [SAM14], or the adaptive CS buffering of [MAT14].

Table 13 compares qualitatively the CS NoC designs found in the literature. TDM has a higher adoption [OUY10][STE12a][BEY13][HE14]. However, the TDM drawback is its large area overhead due to the router's tables required to store the time slot assignment of each flow. Further, a switching algorithm is needed to be executed at each time slice, contributing to increases the dynamic power consumption. Another TDM limitation is the scheduling flexibility because each flow allocation must fit in the TDM wheel running at each router. If one router of the path has the time slot used by another flow, the CS establishment is suspended.

Table 13 – Comparison CS NoC designs.

Virtual CS Technique	Channel sharing	Main drawbacks	Main advantages	Literature adoption
TDM	Time	Expensive router's tables, higher power, switching needed at each time slice, limited scheduling	Simple implementation. High literature adoption	++++
SDM	Spatial	Serialization and de-serialization in the NI, large router area	Simple connection and easy CS control into routers	++
MPN	Not	Inflexibility to manage high variety of throughput grains.	Easy and simple implementation, floorplaning, place and routing. Low area, low critical path, flexibility to frequency scale	+++
IDMA	ID	Expensive router's tables, high routing overhead (2x clock Athereal), low adoption	Ease multicast. Low granularity	+
TDM+SDM	Time and Spatial	High management overhead, Low literature adoption	Low granularity	+

The SDM was proposed after TDM for NoCs, with the goal to reduce power and area compared to TDM. However, some works achieve SDM by defining statics sub-channels between routers, and the NoC acts as a set of disjoint NoC configured at design-time [EJA13]. By the other hand, Leroy et al. [LER05][LER08] propose a multistage SDM NoC based on an adaptive switch. This proposal supports dynamic CS bandwidth and adapts the bandwidth according to the flow requirements at the cost of a complex switching mechanism. Is easy to conclude in that work that the complexity of the switches required for SDM can make it applicability for SoC inviable if a low granularity of SDM is assumed. Leroy et al. [LER05] also compared the SDM with a TDM NoC implementation, showing lower power (-8%) and area in SDM (-31%), at the cost of a higher critical path (+ 37%).

MPNs have an increasing adoption over the last years [FAN11][WAN12][CAR14]. MPNs have the advantage of simple implementation, low area and easy floorplanning [YOO13]. However, MPNs have the drawback of static bandwidth per channel once that the number of MPNs is defined at design-time with a fixed bandwidth. The design can benefit from frequency scale or DVFS techniques to speed up or slow down an specific MPN. This approach is well fitted with the current dark silicon

context, where some system resources need to be turned off due to excessive power dissipation.

As alternative CS designs, the proposal of IDMA [SAM14] appears to be interesting due to the flexibility of multiples CS flow sharing the same link. The proposal adopts a flit routing based on flit's ID. IDM uses the packet header to reserve the required bandwidth to a given flow. Next, the routing is performed flit-to-flit by using an ID tag inserted at each flit. The ID is the same for all flits belonging to a packet. The router uses the ID to select the output port and provides the appropriated bandwidth to each flow, based on a routing reservation table, which is configured at run-time by the header. Results showed success to meet the flow's bandwidth requirement and a low area if compared with Ethereal NoC, which employs VCS using TDM.

As another option of CS design there is the exploratory study of Lusala et al. [LUS11], which merges TDM and SDM. This work adopts a TDM over an SDM concept. This approach leverages to a complex NoC design, divided into sub-channels that are defined at design time with each sub-channel multiplexed over the time. The work concludes that increasing the number of sub-channels on the SDM+TDM NoC creates a higher impact into the NoC area compared to the increased number of slots in a TDM NoC.

Recently, new bufferless-based designs for NoCs have been proposed. Elastic Buffers (EB) enable to reduce the power and area overhead due to the buffers by employing a latch-based design [MIC11]. This implementation creates a distributed FIFO along the communication path and allows to remove the credit protocol, which is required in wormhole flow control. To implement the forwarding mechanism between each EB a ready-valid handshake is employed. This control logic is smaller compared to traditional routers' switch allocators [MIC11]. EB can provide QoS either mixing them with VCS [GRO12] or using a VCS with EB [SEI14]. In this Thesis, and as will be detailed in next Sections, an original design combines EB with MPNs.

Another buffer-centered approach is to design bufferless NoC. Bufferless NoC removes the routers' buffers and saves area and power. Deflection or retransmissions techniques perform the correct packet communication. However, QoS is sacrificed. Some recent works have addressed QoS in bufferless approaches by using a priority-based deflection [ZHA14][YAO14].

Other works address communication QoS by a high-level communication management. High-level management is required to provide a system view over the communication infrastructure, and thus, providing communication QoS by run-time resource management. The implementation of the management protocol can be in central processor as in Winter et al. [WIN11], which presents and evaluates different implementations of a central hardware unit at the NoC level, named *NoCManager*. The manager allocates at run-time guaranteed service VCs to flows. The *NoCManager* contains the status of all links and is responsible for finding a CS path in the NoC. Authors argue that the central *NoCManager* is superior to the distributed technique. Besides this conclusion, the Authors mention scalability issues and point out a hierarchical method as future work.

High-level communication management also can be exposed to user's task by APIs [HEI12][CAR14][MOT11][SAP14][JOV13]. In Heisswolf et al. [HEI12] for example, it was proposed a decentralized policy to NoC management and communication resource allocation according to the application requirements, informed by an API. The policy consists in to create NoC regions (Figure 55) to limit the VC number inside the region by allowing a maximum number of VC assigned to flows GS and BE. The NoC regions are defined with the OS support that is responsible for configuring each router. This approach enables to configure a VC limit in entire regions or only the region border, aiming to accurately distribute the VC budget over the applications running inside the regions.

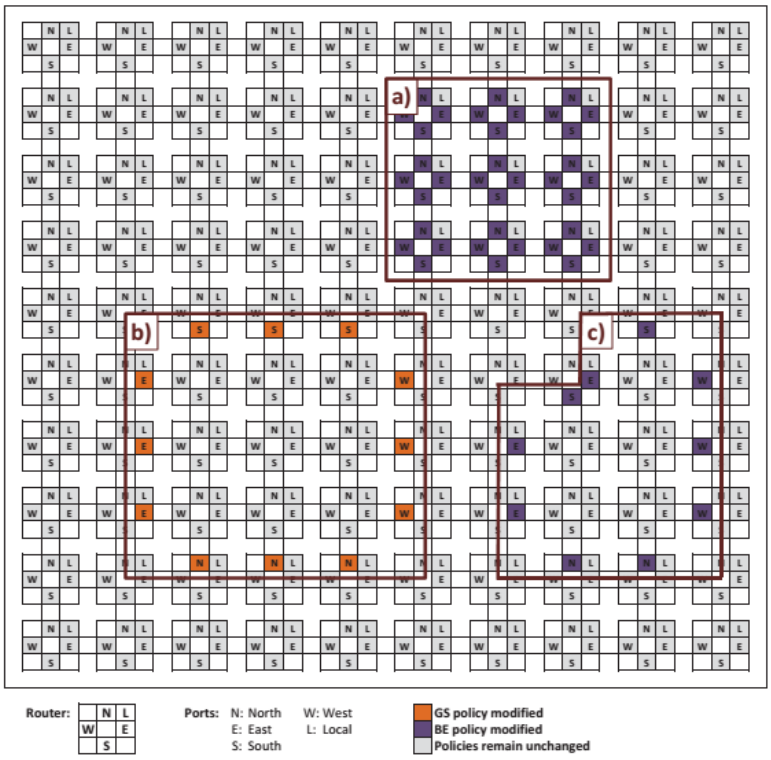


Figure 55 – Cluster regions which can tune the number of VC [HEI12].

The high-level QoS management is supported by a software implementation that consists of a mapping aware of application QoS constraints, jointly with a distributed QoS management within the OS of PEs, which defines the NoC regions.

5.1.1 State-of-the-Art Discussion

QoS targeting communication constraints for many-cores received many proposals in the last decade. Observing the reviewed works, it is possible to observe the adoption of the following techniques:

- flow priority;
- end-to-end throughput guarantees using CS;
- area efficient buffer approaches;
- high-level communication management.

Several proposals for QoS at the communication level have hardware implementations. These proposals can divide the flow priority into different classes, provide CS by mainly using TDM, SDM or MPN, and area efficient approaches centered in the buffer design. Observing the state-of-the-art related to CS NoCs, it is possible to find several design options. The TDM and SDM have a high capacity to meet the QoS levels. The drawback of such proposals is scalability, due to the area and power overheads. As shown by [YOO13], the use of MPNs can equally compete, and in some scenarios to overcome VC-based approaches.

With the increasing number of PEs on the same die, has emerged the need to provide high-level QoS management. Therefore, QoS becomes to be managed at the system level and some system services exposed to applicationm developer [CAR14]. At system level by run-time QoS management policies, and at developer level by QoS APIs that exposes the hardware services. However, assigning to the application's developer the job to configure the QoS statically is not a suitable solution. Self-adaptive systems are the alternatives, where the constraints are monitored at run-time, and a manager act over the system resources to meet the constraints. Thus, the application developer only has the role to inform the application's constraints to the system, without the need to act directly on the system resources.

This Chapter proposes techniques to achieve QoS at the communication level, aware of the main characteristics previously stated in the state-of-the-art. First, Section 5.2 proposes an SDN technique for NoCs, targeting high-level communicating management. At the hardware level, this Thesis proposes an MPN design with routers based on EB, enabling run-time reconfiguration and low area overhead. At the software level, a NoC-Controller implements the SDN paradigm by defining at run-time the paths in the MPN. Next, Section 5.3 proposes the adoption of CS targeting end-to-end throughput guarantees achieving QoS at the communication level. The SDN architecture is the support to implement the CS proposal. A CS management protocol implemented in a CS controller defines the CS paths. Section 5.4 evaluates the SDN architecture and the CS protocol.

5.2 Software-defined Networking (SDN) for NoC-based Many-Cores

SDN is a computer network paradigm that has as the central concept the routers' simplification. The diversity of routers available on the market of computer networks made difficult the process to configure and manage a network, motivating the SDN development. Figure 56 overviews a classical SDN organization [JAR14]. SDN was conceived assuming simple architectures, moving the control logic from the router to a high-level manager, called *Network Controller*, implemented in software. With this paradigm, routers act as simple forwarding units, programmed by the controller at run-time according to network policies defined by the user or the network status.

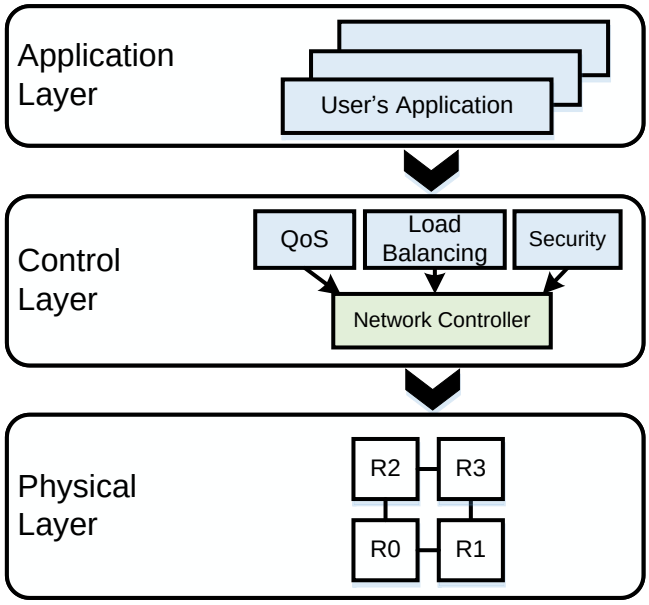


Figure 56 – SDN organization, and its adoption for NoC design [RUA17a].

The same scenario may occur today in the context of many-core systems. NoC designs targeting QoS adopt large buffers, several virtual channels, and complex arbitration/routing schemes [LER08][YOO13] to meet the applications' requirements. The complexity of current NoCs motivated this research to explore SDN applied to many-core systems, with potential advantages to reduce the NoC cost (area and power [SHA17]) concomitantly with a flexible management (e.g., QoS policies defined by software). Also, SDN can provide better reusability because routers are generic and simple hardware components, configured by software.

Recent works started to explore the SDN paradigm for SoC communication. Cong et al. [CON14] propose an SDNoC architecture where the control plane is deployed as a distributed unity at each router. The routers' control plane exchanges messages to implement the communication management protocol and to define the path for the flows. That work presents few details related to the architecture and no RTL validation. Sandoval et al. [SAN15] propose an SDN organization with three layers: operating system, network operating system, and infrastructure. The work assumes routers that can have the routing algorithm defined by the SDN controller. Flows that are not managed by the SDN controller use the XY routing algorithm. Work [SAN16] evaluated the configuration time for several routing algorithms, implementing them in the SDN controller. Results showed that the performance of the SDN to configure the routers varies according to the routing algorithm and the injection rate. For congested scenarios, worst results were obtained with adaptive routing algorithms. Scionti el al. [SCI16] propose the SDN architecture to explore dynamic changes in the network topology. Each PE has specific instructions to control the network topology by software, including switch off the links which are not used. The SDN paradigm is implemented by these specific instructions and not by an SDN Controller. Table 14 compares de SDN proposals for NoCs.

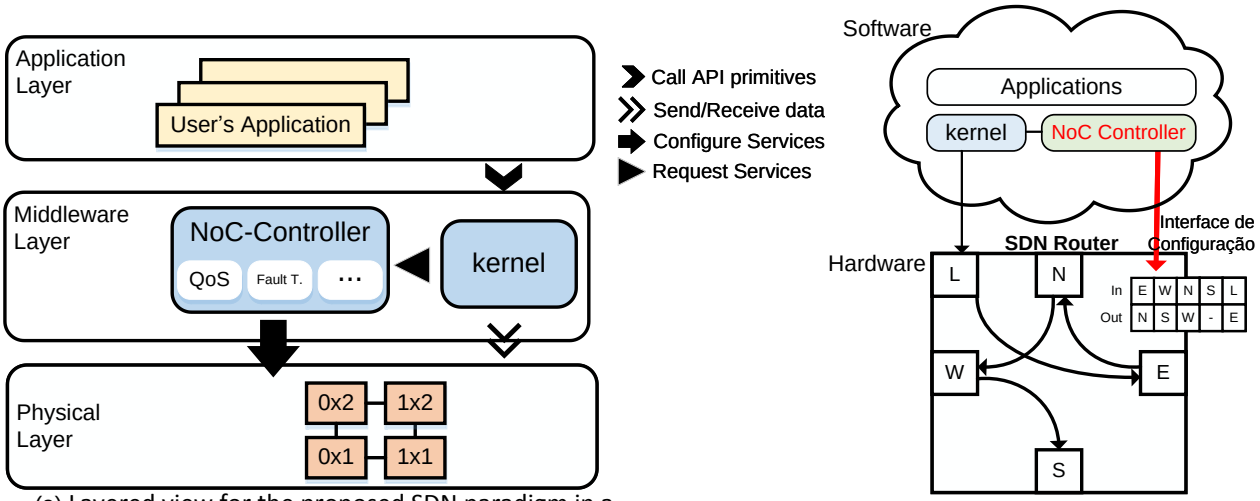
Table 14 – Related works on SDN architectures for Many-Core SoCs.

Works	Implementation Details	RTL Validation	SDN Controller
[CON14]	Few Details	No	One per router
[SAN15][SAN16]	Arch. organization overview	Yes (SystemC)	One per system
[SCI16]	Only router level details	Yes	N.A.
This Thesis	***Arch. organization and implementation***	***Yes (VHDL, SystemC)***	***One per system***

The SDN architecture herein proposed covers two gaps observed in the literature. The first one is in to provide a comprehensive SDN architecture, describing the hardware and software layers. The second one is the SDN evaluation against a state-of-the-art hardware method for defining the paths. The proposed SDN provides a generic and flexible architecture to manage the communication in many-core systems and to provide differentiated communication services. This Thesis is focused on the QoS service by establishing CS. Other services could be supported, for example, fault-tolerance, security, and load balancing.

5.2.1 SDN Architecture Overview

Figure 57(a) presents the proposed layered SDN organization for the many-core context. The application layer has the users' applications. The middleware layer contains the embedded OS (kernel) and the NoC-Controller (NC). The NC implements the SDN services to the kernel. Figure 57(b) presents the SDN-based communication directions. The kernel asks the NC to define paths. The NC set the paths by configuring SDN routers, which act as a forwarding unit, linking an input port to an output port. The NC configure several SDN routers to make a path. After the connection, the kernel injects/receives the applications' data by using the local port of the SDN router connected to it.



(a) Layered view for the proposed SDN paradigm in a many-core organization

(b) SDN-based communication

Figure 57 – (a) Layered view of the SDN paradigm in a many-core organization; (b) SDN-based communication [RUA17a].

Despite the advantages of the SDN-based management, it also imposes an important challenge that is the *software overhead* to manage NoC flows at run-time. The layered SDN

organization proposed in Figure 57(a) helps to mitigate this issue by assuming two main *principles*:

i. the NoC Controller provides differentiated communication services (QoS on this Thesis), being *decoupled from the PS architecture*. The system may work with the kernel and the PS NoC, independently of the SDN infrastructure;

ii. the NoC Controller is a *parallel system service* of the kernel.

Figure 58 presents details the physical layer and the modifications required to support the SDN paradigm. Figure 58(a) shows a standard many-core architecture, with PEs connected to one PS router (R in the Figure). Figure 58(b) presents the SDN architecture, with an NC managing the connection between SDN routers (S_R). Figure 58(c) shows the integration of the SDN architecture to the many-core architecture. The communication architecture presented in Figure 58(c) corresponds to MPNs, with one PS network and a set of SDN networks, named ***subnets***. The PS network is used for management packets and to transmit data packets when there is no path between two PEs. It also has the role of configuring the S_Rs.

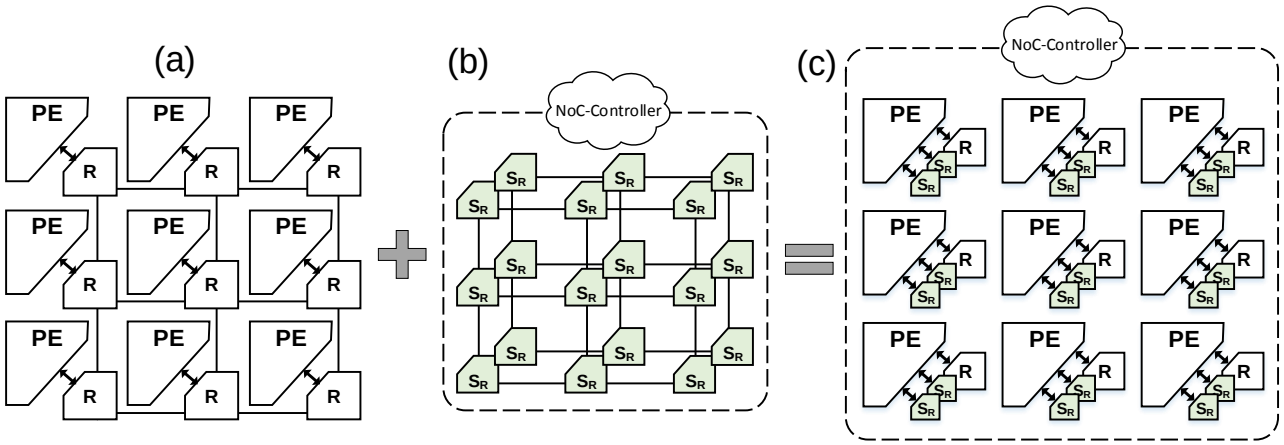


Figure 58 – Integration of the SDN in a standard NoC-based many-core architecture. (a) Standard NoC-based many-core architecture, (b) proposed SDN-based architecture (c) integration of the SDN in a NoC-based manycore architecture.

5.2.2 Hardware Architecture – SDN Router

Figure 59 details the proposed S_R. According to the SDN paradigm, the S_R should act as forwarding unit. To reduce area, Elastic Buffers (EB) [MIC11][MIC13] replace input buffers – Figure 59(a). The router has five inputs and five output ports (East, West, North, South, Local). Each input port has one EB. An EB contains a master and a slave latch, controlled by the EB control logic. The master latch is enabled when the clock signal is low, and the slave latch when the clock signal is high. This latch-based design allows the latches to be used as two independent storage locations. An EB uses a ready-valid handshake protocol. The *ready* signal notifies the upstream router that the current EB can receive a flit in its master latch. The *valid* signal notifies the downstream router that current EB has a flit to transmit. A flit is transferred to the downstream EB when both the *ready* and *valid* are asserted at the rising clock edge.

The EBs retains data for one clock period, avoiding long wires, ensuring a reduced clock period. EBs also enable to reduce the silicon cost compared to a two-slot FIFO [MIC13], once EBs need only one master-slave flip-flop instead of two.

Figure 59(b) details the S_R architecture. Besides the EB, each port has an upstream *req* signal employed by the consumer PE to request data from the producer PE. The *req* is used for the inter-task communication, being asserted when the consumer task requests a message to the producer task (this replaces the MESSAGE_REQUEST packet).

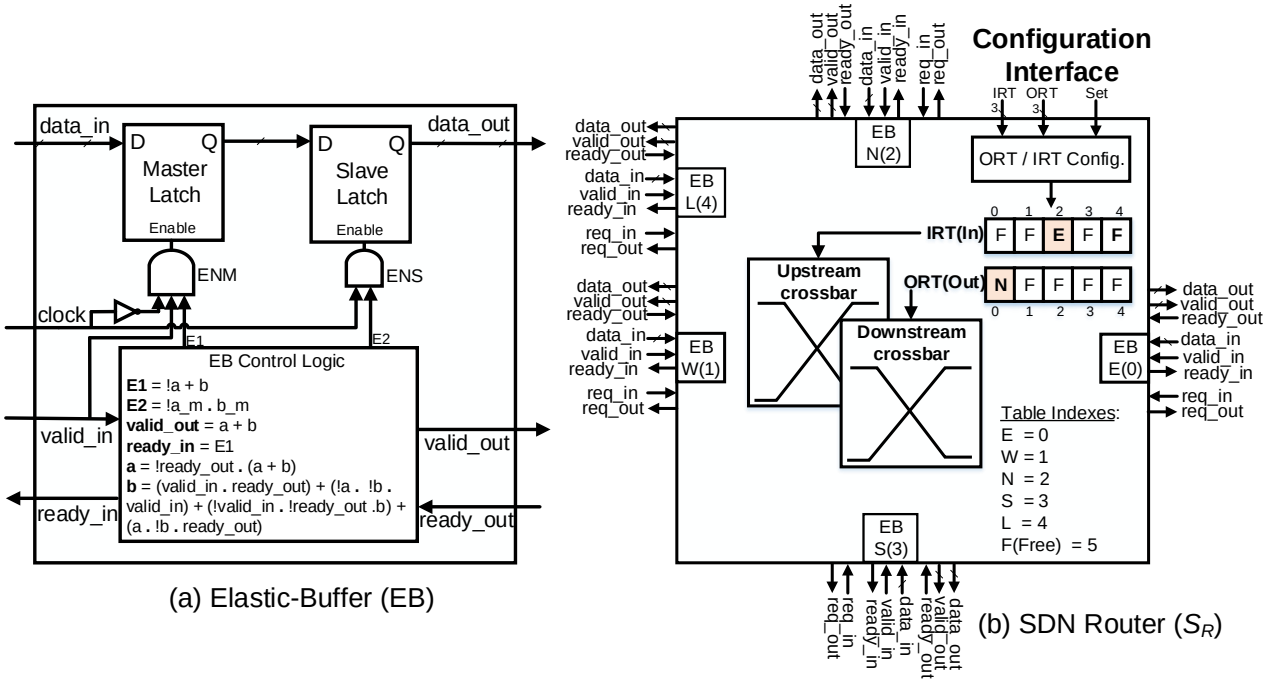


Figure 59 - (a) EB architecture [MIC11][MIC13], (b) SDN router architecture, with 5 EB [RUA17a][RUA18].

The S_R has internally two crossbars to connect the upstream (*ready*, *req*) and the downstream (*valid*, *data*) signals between the input and output ports. They are configured by the Input Reservation Table (IRT) and Output Reservation Table (ORT), respectively. Each table is a 5-entry array (number of input ports) with 3 bits at each slot (enabling to store six states: E, W, N, S, L, Free). In Figure 59(b), the North *inport* is forwarding data to the East *outport*. A *configuration interface* enables to programs the IRT and ORT tables. This interface is the key feature to make the router simple, avoiding logic for routing and arbitration modules. After configuring the S_R routers, data is transmitted in streaming without interferences by the path.

Figure 60 presents the process to configure an S_R. The SDN configuration is independent of the PE architecture once the configuration process does not include the NI. The NC sends through the PS network a configuration packet to program the IRT/ORT tables. Each configuration packet has 3 flits: header, with the target address and a flag specifying that the packet must be consumed by a given S_R and not by the NI; payload size, which is always 1; configuration, with 3 fields: input port, output port, SDN network number. It is not necessary to clear the IRT/ORT tables because the configuration process is managed by software.

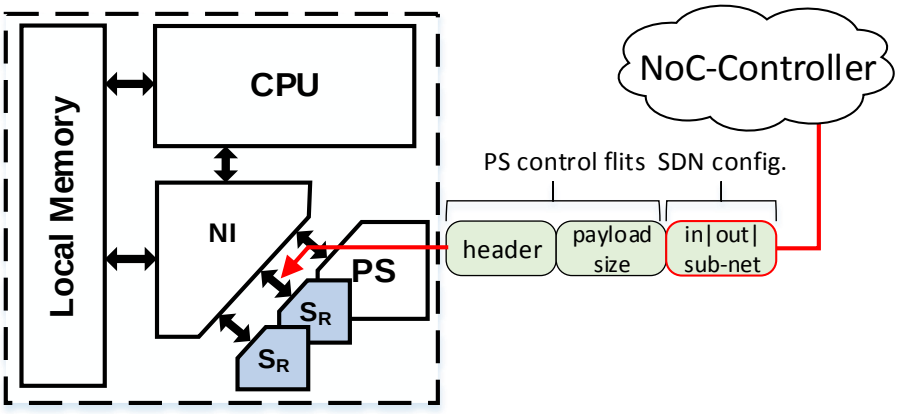


Figure 60 – PE architecture and configuration process of an S_R [RUA17a].

5.2.3 Software Architecture

The software architecture concerns the implementation of the NC, which handles path establishment requests generated by the kernel. As the NC is decoupled from the kernel (Figure 57), it can also can handle path requests from other system's components. Algorithm 1 presents the pseudo-algorithm of the NC.

Algorithm 1: NOC-CONTROLLER (NC)

```
Input: Source, Target
begin
1: while TRUE do
2:     path_request = read_path_request()
3:     if path_request == VALID then
4:         path[], subnet = SEARCH-PATH(Source, Target)
5:         if path[] ≠ Ø then
6:             configure_SDN_routers(path[], subnet)
7:             send_ack_to_requester(subnet)
8:         else
9:             send_nack_to_requester()
10:        end if
11     end if
12: end while
end
```

The NC continuously observes for new path requests (lines 1 and 2 of Algorithm 1). If there is a request (line 3), the NC calls the SEARCH-PATH algorithm at line 4. The role of the SEARCH-PATH algorithm is to define a path between a source and a target PE, implementing the control logic of the network (removed from the router to make it simple) according to a given path definition policy, for example, to fulfill QoS. The SEARCH-PATH algorithm returns the *path[]*, which consists of an array composed of the path routers' addresses, and the selected sub-net of the MPN. If the path is valid (*path[]* ≠ Ø), the NC configures each S_R of the path by sending the configuration packet (line 6). Next, at line 7, the NC sends an *ack* message to the OS (requester). If the path cannot be defined (*path[]* == Ø), the algorithm sends a *nack* to the requester at line 9.

5.3 Circuit-Switching (CS) based on the Software-defined Networking (SDN) Paradigm

This proposal aims to provide QoS support by adopting the CS approach managed by the SDN paradigm. CS has an initial phase called connection (or setup), which searches and allocates the path's resources. The connection phase is the subject of several researches, with the goal to reduce the connection latency. Table 15 presents related works on CS NoCs with proposals *addressing the search path* method. The 2nd column details the hardware infrastructure to support CS. Most techniques are based on the TDM and SDM approaches. The 3rd column of Table 15 presents the resulting path length of the search path mechanism. Minimal path length corresponds to the Manhattan distance between the source and target PEs. Deterministic search uses a deterministic routing algorithm, as XY. The shortest path searches for all possible paths in the NoC, returning the minimal path between the source and target PEs. Works [LER08][STE12b][LUS11] addresses only minimal or deterministic search methods. Such approaches limit the path exploration, thus reducing the success rate to establish the CS connections. Works [LIU15][CHE16][LIU14b] adopt the search for shortest paths. This approach increases the success rate to establish CS connections because the search space increases.

Table 15 - Related works on CS NoCs, with search path proposals [RUA17a].

Works	CS	Path Length	Implementation	Organization	Target
[STE12b]	TDM	Minimal	SW	Centralized	NoC
[CHE16]	TDM	Shortest	HW	Centralized	NoC
[LER08]	SDM	Deterministic	SW/HW	Centralized	NoC
[LIU14b][LIU15]	TDM/MPN	Shortest	HW	Distributed	NoC
[LUS11]	SDM	Deterministic	HW	Distributed	NoC
This Thesis	***MPN***	***Shortest***	***SDN-based***	***Centralized***	***System***

Works that search for the shortest path use dedicated hardware implementations (4th column), with centralized or distributed approaches (5th column). Chen et al. [CHE16] propose a centralized design, implementing a dedicated hardware unit to establish the connection. Liu et al. [LIU14b] implement a distributed approach called Parallel Probe that uses a flood-based search, eliminating redundant paths, and achieving a constant setup time per hop. That work was extended to MPN and SDN [LIU15].

Most proposals address CS at the NoC level and CS infrastructures with hardware-centered connection algorithms and constant setup time. Hardware-based search path designs are faster than software approaches, at the cost of increased silicon complexity. Software-based solutions are scarce because the search path evaluation has as target the NoC context (6th column) with a comprehensive CS management not being addressed.

This Section presents a CS infrastructure and management based on the SDN paradigm previously described. This Section proposes a *CS-Controller*. The *CS-Controller* is an instance of the *NoC-Controller* dedicated to QoS services. The *CS-Controller* defines the paths by running a SEARCH-PATH based on the Hadlock's algorithm [HAD77] and configuring SDN routers of the MPN at run-

time. The *CS-Controller* handles requests from the M_{PE} to establish connections during the RT application admission and to release connections when the application finishes its execution. After the CS establishment phase, the RT applications start their execution with all task exchanging data by dedicated CS paths, established during the application admission. This approach brings two main benefits: (*i*) removes the overhead to execute several connections and releases during the application lifetime, e.g., for each decoded frame [LER08]; (*ii*) provides communication predictability and QoS during the application lifetime, once each path is dedicated to serve one flow.

The CS technique herein proposed adopts a software-based search path algorithm, proposing a high-level CS management based on the SDN paradigm. This proposition fills a lack in the related works regarding a systemic and self-aware CS support for applications, i.e., aiding the CS provisioning from the applications' beginning up to its ending.

5.3.1 CS-Controller

The *CS-Controller* is an instance of the *NoC-Controller* implementing the QoS service by managing CS establishment and releasing at run-time. Figure 61 presents a many-core architecture example detailing the presence of the *CS- Controller* (CS). The MPN is used for CS connections, with the SDN routers managed by the *CS-Controller*. The *CS-Controller* is developed as a system task and mapped in the most central S_{PE} of the system.

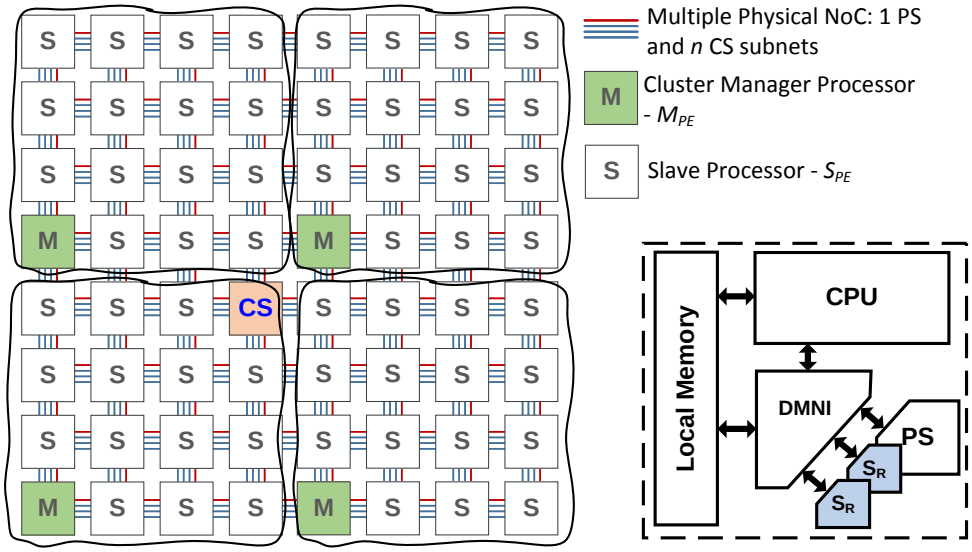


Figure 61 – Proposed many-core, with the SDN-based CS.

A distributed implementation of the *CS-Controller* seems to be a natural choice because it distributes the search path computation load in several instances of the controller. However, we observed in experiments that one single unit of *CS-Controller* does not represent a bottleneck for CS establishment in system's sizes up to 400 PEs (maximum system size assumed in the evaluations). The reason for this performance is due to the low complexity of the SEARCH-PATH algorithm. Its computation time, in the average case, grows linearly with the system size. Additionally, a distributed implementation assuming one CS-Controller for each cluster creates a communication

bottleneck between the CS controllers to obtain a systemic view of the system. The CS controllers need to communicate between them when it is necessary to create paths that cross the clusters borders. It is possible to adopt a distributed version of the CS-Controller for larger systems (> 20x20) without the cooperation between CS-Controllers, at the cost of reducing the path search exploration.

The *CS-Controller* executes two actions: (*i*) searches and configures CS paths; (*ii*) release CS paths. Equation 4 presents the memory storage requirements of the *CS-Controller* (in bytes).

$$S_{mem} = (R_n * S_n * [p + 1]) + 3 * R_n \quad (4)$$

where: R_n is the number of routers, S_n is the number of SDN subnets, and *p* is the number of ports per router (5 - N, S, E, W, L_IN) plus one representing the input port of the NI (L_OUT). The first multiplication of the Equation 4 computes the amount of data required to store the status of the input port for each SDN router, in a 3-dimensional array called ***inport***. The *inport* stores zero when a given port is free, or the $path_{ID}$, when the port is allocated. The {$3*R_n$} multiplication corresponds to the size of inherited variables from the Hadlock's algorithm (propagation, neighbors, and detour).

Section 5.4.5 evaluates the memory storage requirements of the *CS-Controller*.

5.3.2 SEARCH-PATH Algorithm

The SEARCH-PATH algorithm implements a heuristic that explores the path diversity of the MPN to select a *subnet* and a $path_{ID}$ from a source *S* to a target *T*.

The Hadlock's algorithm is a sub-function of the SEARCH-PATH algorithm. It searches for a path at each subnet. The Hadlock's algorithm is a grid routing search method, proposed in Hadlock [HAD77] and described in Sherwani [SHE05], originally used in VLSI synthesis. It is a bread-first algorithm that makes it expansion guided by a detour number *d*, assigned for each visited node. For minimal paths, *d(T)*=0. The detour number is increment according to the number of hops away from the minimal path. When a path *P* is found, the length of *P, len(P),* connecting *S* and *T* is defined according to Equation 5.

$$len(P) = MS(S,T) + 2 * d(T) \quad (5)$$

where: $MS(S,T)$ is the Manhattan distance between *S* and *T*.

Figure 62 present the SEARCH-PATH algorithm. The algorithm initially searches minimal paths in all *subnets*. A *min_path* variable controls the Hadlock's algorithm search path mode. When *min_path* is TRUE, the Hadlock's search is limited to minimal paths (*d(T) = 0*), when *min_path* is FALSE, the Hadlock's search is released to find the shortest path (*d(T) > 0*), if it exists.

Initially, line 2 sets *min_path* as TRUE. The first round of the loop (lines 4-16) explores all SDN subnets, stopping when a minimal path is found. Line 5 tests if the source (L_IN) and target (L_OUT) routers' interface with the DMNI are free or not. If TRUE, the Hadlock's algorithm is called (line 6), returning the $path_{ID}$ if a path was found, 0 otherwise. If all subnets were explored and no minimum

path exists (line 12), the algorithm switches the search to non-minimal paths (*min_path*=FALSE), at lines 13-14. The search stops at the first path found. By employing this approach, the complexity of the SEARCH-PATH is $O(R_n^2)$ in the worst case and $O(R_n)$ in the best case.

Algorithm 2: SEARCH-PATH

Input: *S, T*
Output: $path_{ID}$, *subnet*
begin
1: $path_{ID}$ = 0
2: *min_path* = TRUE
3: *subnet* = 0
4: **while** *subnet* < S_n **do**
5: **if** *inport*[*S*][*subnet*][L_IN]== 0 **and** *inport*[*T*][*subnet*][L_OUT]==0 **then**
6: $path_{ID}$ = HADLOCK**(***S, T*, *min_path***)**
7: **if** $path_{ID}$ **>** 0 **then**
8: **break**
9: **end if**
10: **end if**
11: *subnet* = *subnet* + 1
12: **if** *subnet* == S_n **and** *min_path* == TRUE **then**
13: *subnet* = 0
14: *min_path* = FALSE
15: **end if**
16: **end while**
17: **return** $path_{ID}$**,** *subnet*
end

Figure 62 - SEARCH-PATH algorithm [RUA17a].

When the Hadlock's algorithm reaches router *T*, it executes the *retrace* phase to define the path's routers. This phase is used to store the $path_{ID}$ in the *inport* structure (software allocation), and to configure the CS routers (hardware allocation).

5.3.3 RELEASE-PATH Algorithm

The RELEASE-PATH algorithm uses the $path_{ID}$ stored in the *inport* array to release the connection. For each path, the algorithm removes the $path_{ID}$ value associated with the allocated input ports, setting it to 0 (free). The release phase is only executed in software. It is not necessary to clear the IRT/ORT tables because the software manages the allocation. The RELEASE-PATH is a simple algorithm with complexity *O(len(P))*.

The *CS-Controller* exposes to the system the CS services (connection and release) generically because it only requires the source/target addresses to make connections. Thus, it also can handle requests from other IPs (rather than $M_{PE}s$), without a software stack (a kernel for example). Additionally, as the IPs are not involved in the process to configure the SDN router, the connections can be established between generic IPs (memories, GPUs, I/O modules).

5.3.4 CS Management Protocol

Figure 63 presents the sequence diagram detailing the protocol for CS connection and release, detailing the relationship between the components of the system and how they work together to provide CS for the RT applications. The CS management is performed for each new RT application. It is performed in parallel to the application admission management (Subsection 3.2.1).

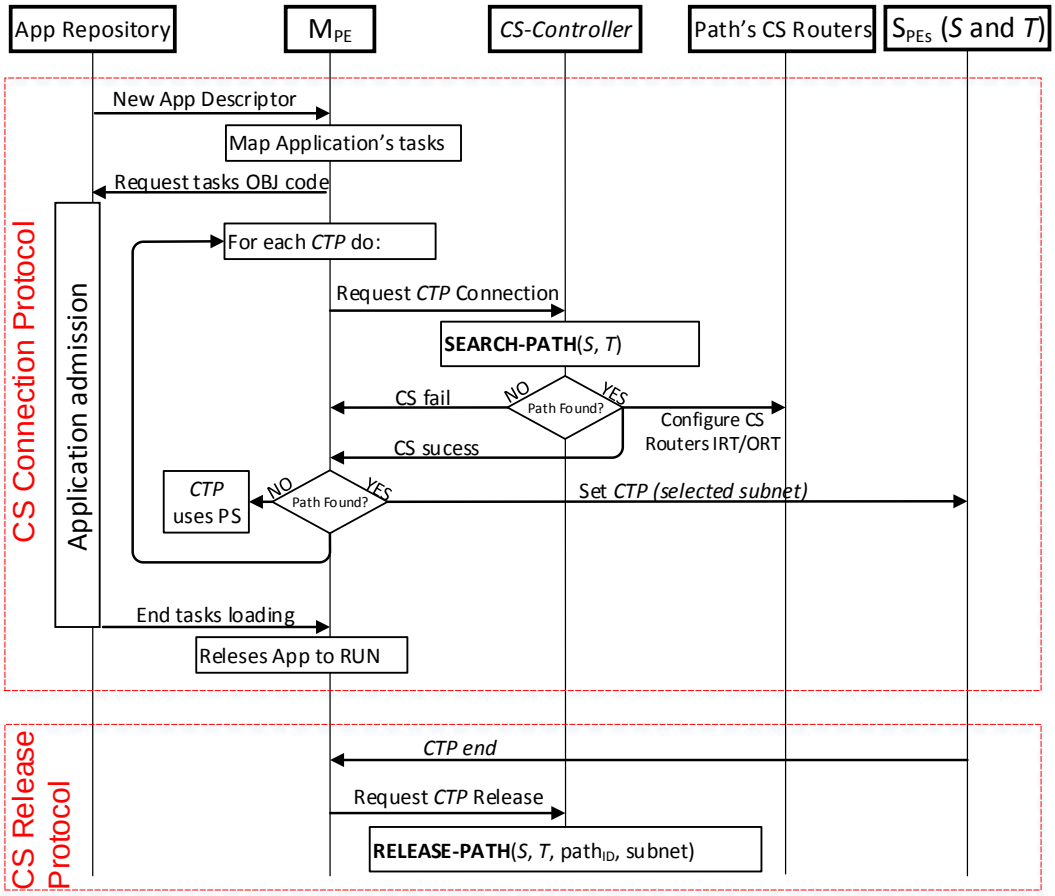


Figure 63 - CS management protocol [RUA17a].

When a new RT application requests to execute in the system at run-time, the *App. Repository* sends the *app descriptor* to an M_{PE}, which execute a task mapping heuristic to select the S_{PE} to each application's task. Next, the M_{PE} requests to the *App. Repository* to load the object code of the mapped tasks into its corresponding S_{PE}s (application admission phase). The load of the task code uses the PS subnet. In parallel to the load of the tasks' code, the M_{PE} requests connections to *CS-Controller*.

Best-effort applications do not have timing bounds and use the PS subnet. RT applications request CS for all *CTPs* (communicating task pairs). For each *CTP*, the M_{PE} requests the *CS-Controller* to establish a connection between the source and target S_{PE}s. The *CS-Controller* executes the SEARCH-PATH algorithm and returns to the M_{PE} if the CS establishment failed (returning 0) or succeeded (returning the $path_{ID}$ and *subnet*). If succeeded, the M_{PE} sends messages to the *S* and *T* S_{PE}s, notifying the selected *subnet*. This message configures the kernel of *S* and *T* S_{PE}s to inject and receive data by CS in the configured subnet. If failed, the *CTP's* communication uses the PS subnet

by default. When both the *CTPs'* allocation finishes, and the tasks are allocated into S_{PE}s (each task notifies the M_{PE} when it is ready), the M_{PE} releases the application sending a message to each task to start the execution.

When a task finishes, the S_{PE} notifies the M_{PE} the end of a given *CTP*. If the *CTP* communicates through CS, the M_{PE} uses the task identifier (unique for each running task), to recover the *subnet* and $path_{ID}$, and requests the *CS-Controller* to release the connection through the execution of the RELEASE-PATH algorithm.

5.4 Results

In this section, we evaluate the silicon area, power, performance, memory requirements, and latency of the proposed SDN-based CS. The many-core hardware is described in VHDL (for synthesis – Cadence's Genius) and RTL SystemC (for performance evaluation of large-scale systems). Applications and kernel are implemented in C language.

5.4.1 Performance Evaluation

The experimental setup addresses many-core sizes from 64 PEs (4 M_{PEs}, 60 S_{PEs}) to 400 PEs (16 M_{PEs}, 384 S_{PEs}), with three SDN subnets configurations: 4, 6, 8. The PS routers use 32-bit flits, and the CS routers 16-bit flits. Results use the following definitions:

Path Diversity (P_D): number of available CS paths (equation 6).

$$P_D = \#PEs * \#SDN\ subnets \tag{6}$$

Path Exploration (P_{EX}): percentage of CS requests w.r.t the *path diversity* (equation 7).

$$P_{EX} = \frac{\#\ CS\ requests}{P_D} \tag{7}$$

Success Rate: percentage of established connections per CS requests (equation 8).

$$success\ rate = \frac{\#\ CS\ connections}{\#\ CS\ requests} \tag{8}$$

The performance evaluation addresses several scenarios with different RT benchmarks: DTW (Dynamic Time Warping – pattern recognition algorithm), MPEG decoder, MPEG-2 decoder, and VOPD. Figure 64 presents the applications' communicating graph. We selected a heterogeneous set of applications aiming to create a fair evaluation. Each S_{PE} executes simultaneously two tasks. The evaluated scenarios execute several instances of the applications in such a way to have all S_{PE}s executing two tasks (s*ystem occupation* equal to 100%). These scenarios enable to stress the SDN infrastructure. Real-life scenarios hardly run with a system occupation equal to 100%, also mixing

best-effort application (which do not need CS) and RT applications.

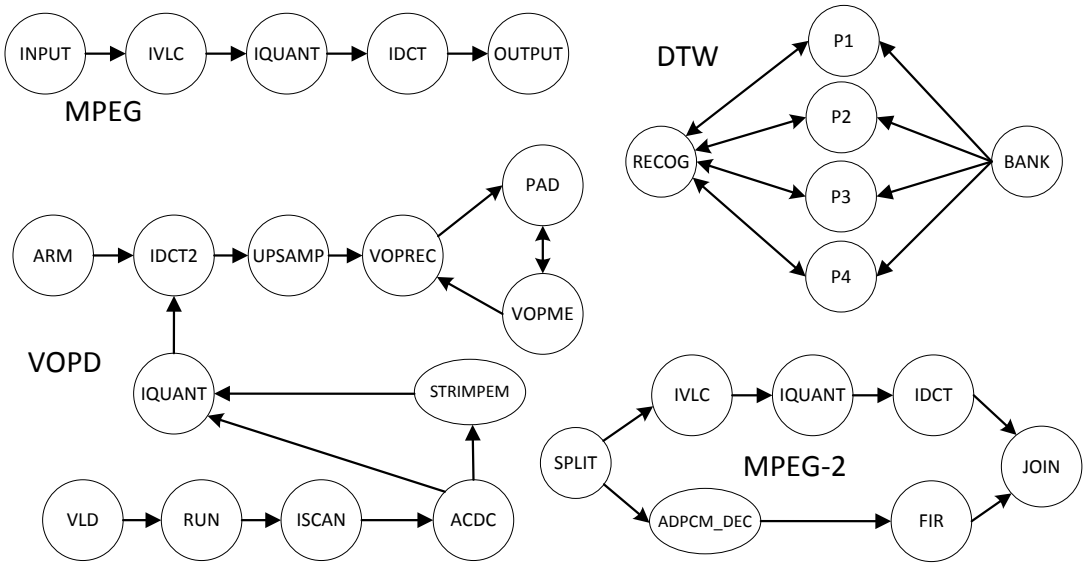


Figure 64 - Communicating task graph of the applications' benchmarks [RUA17a].

Table 16 details the results. The 1st and 2nd columns present the scenario's configuration. The 3rd column presents the number of tasks executing for each scenario. The 4th column presents the P_D for each scenario.

Table 16 - Results for simulations with 100% of RT applications and a 100% of system occupation [RUA17a].

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Scenario – system size	**SDN nets**	**Simultaneous Running Tasks**	**Path Diversity (P_D)**	**CS requests (P_{EX})**	**Success Rate**	**Manhattan Distance** *Avg. / Std.Dev / Max.*	**Hops** *Avg. | Std.Dev / Max.*	**SEARCH-PATH time** *Avg. / Worst.*	**App. Start Overhead** *Avg. (ms.)*
8x8 (64 PEs)	4	120 (100% occupation)	256	127 (49%)	94.4%	2.6 | 1.6 | 8	2.8 | 1.9 | 14	4258 / 48614	1.4
	6		384	127 (33%)	99.2%		2.7 | 1.6 | 8	3821 | 13166	1.6
	8		512	127 (24%)	100%		2.6 | 1.6 | 8	3725 | 12158	1.1
16x16 (256 PEs)	4	480 (100% occupation)	1024	623 (60%)	88.6%	2.5 | 2.0 | 21	2.7 | 2.1 | 21	5336 | 52965	3.1
	6		1536	623 (39%)	97.7%		2.6 | 2.1 | 22	5326 | 102433	3.9
	8		2048	623 (30%)	99.8%		2.5 | 2.0 | 21	5220 | 38361	3.5
20x20 (400 PEs)	4	768 (100% occupation)	1600	916 (57%)	90.94%	2.7 | 2.1 | 26	3.0 | 2.7 | 30	8521 | 486180	4.0
	6		2400	916 (38%)	98.25%		2.7 | 2.3 | 26	7716 | 174033	4.5
	8		3200	916 (28%)	100%		2.7 | 2.1 | 26	7121 | 107214	2.9

5.4.2 Success Rate

The 5th column of Table 16 presents the number of CS requests and the achieved P_{EX}. The number of CS requests is a function of the applications' graph. Increasing the number of SDN subnets decreases the P_{EX} due to a higher P_D. For example, the 20x20 scenarios generate 916 CS

requests, resulting in a P_{EX} =57% for 4 subnets, and a P_{EX}=28% for 8 subnets. This result shows the rich path diversity provided by the MPN design. Even with a system occupation of 100%, the amount of CS requests is below the P_D. The 6th column of Table 16 presents the *success rate*. The *success rate* increases with the number of subnets, due to the higher P_D. The scenarios with 6 and 8 subnets reach a success rate superior to 97%. Figure 65(a) presents the *success rate* as a function of the P_{EX} (bottom x-axis) and CS request (top x-axis), for scenario 20x20:6 (most complex scenario). This scenario has a P_{EX}=38%, with a *success rate* of 98.25% (highlighted in the Figure).

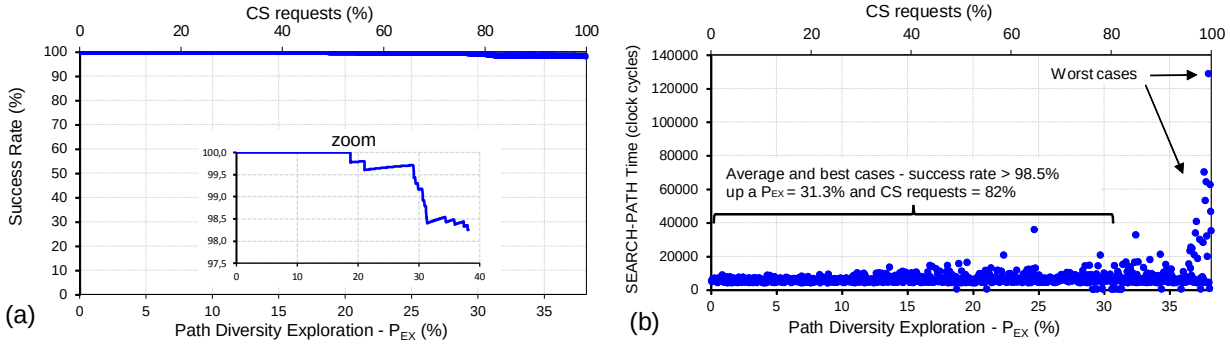


Figure 65 – (a) success rate for 20x20:6 CS subnets; (b) SEARCH-PATH execution time for 20x20:6 [RUA17a].

The mapping heuristic also contributes to the high success rate. This Thesis adopts a task mapping heuristic which distributed the RT tasks in free PEs [RUA16a]. Other heuristics [SIN13b] may increase the proximity of the communicating tasks, contributing to increasing the success rate further.

The achieved success rate can be compared to related works that address the shortest path search [LIU15][CHE16][LIU14b]. It is worth to mention that each proposal uses a different method to evaluate the success rate. Our success rate is computed according to the number of CS requests. In Liu et al. [LIU14b], the success rate is computed according to the *router rate*, a parameter that refers to the portion of clock cycles in which a node is used for transferring data. In Chen et al. [CHE16] the success rate is computed according to the requested BW. Work of Liu et al. [LIU15] does not present the success rate. Additionally, works [CHE16][LIU14b] addresses TDM while ours use MPN.

In Liu et al. [LIU14b] it is presented the success rate for a 16x16 mesh with a *master percentage* of 20% and 50%. The *master percentage* is similar to ours P_{EX}. Our success rate for P_{EX} =50% is 92% (20x20:4, worst scenario), while Liu et al. [LIU14b] achieves 30% of success rate with a master percentage=50%. That work also evaluates the HAGAR solution with the same configuration, which reaches a success rate near to 28%.

In Chen et al. [CHE16] the authors vary the requested BW to evaluate the success rate. The Authors also considers a background traffic (*bk*) as the percentage of used TDM slots. The *bk* can be roughly compared to ours P_{EX}. Assuming 20% of *bk*, the success rate achieved for a requested BW of 16% (maximum presented) was 82% for a 4x4 mesh. Our success rate for an 8x8 mesh with the P_{EX} of 20% was 100% regardless the number of subnets (4, 6 or 8).

5.4.3 Path Length

The 7th column of Table 16 presents the Manhattan distance between communicating tasks. It is worth to note that the average and standard deviation values for the scenarios are similar, regardless the system size. The main cost function of mapping heuristics is to place tasks belonging to the same application near to each other. As the system occupation increases, the number of contiguous regions to map the tasks reduces, explaining the maximum hop number.

The 8th column of Table 16 presents the path length (hops). The algorithm succeeds to establish short paths, reducing the interference for the next connections, which contributes to increasing the *success rate*.

5.4.4 SEARCH-PATH Execution Time

The 9th column of Table 16 presents the SEARCH-PATH execution time, in clock cycles (*cc*), including the execution time of the algorithm and the configuration of the SDN routers through the PS subnet. For all 20x20 scenarios (worst cases evaluated scenarios), the higher average value is 8,521 *cc* or 8.5 μs@100 MHz. Figure 65(b) plots the SEARCH-PATH execution time as a function of the P_{EX} (bottom x-axis) and CS requests (top x-axis), for 20x20:6 scenarios, respectively. Compare Figure 65(a) with Figure 65(b). The execution time in Figure 65(b) remains near 7,029 *cc* up to a P_{EX}=31.3% (third quartile). After this point, the minimal paths become scarce, requiring the search of non-minimal paths. With a P_{EX}=38%, the execution time reaches the worst case, failing to establish some connections.

Figure 66 presents the average execution time for the SEARCH-PATH algorithm as a function of the number of PEs and SDN subnets. In this scenario was used random source and target addresses generation (random task mapping), thus presenting a pessimistic evaluation. The execution time grows linearly with the system size at an average rate of 33.27 *cc*/PE. The 10th column evaluates the impact of the CS establishment phase on the applications' start time. In the worst scenario, applications were delayed by 4.5 *ms*. This delay only occurs when the applications begin the execution, being not noticed by the end user during the application execution. Some related works [LER08][LIU14b] may assume a new CS connection for each message, with an overhead proportional to the communication volume.

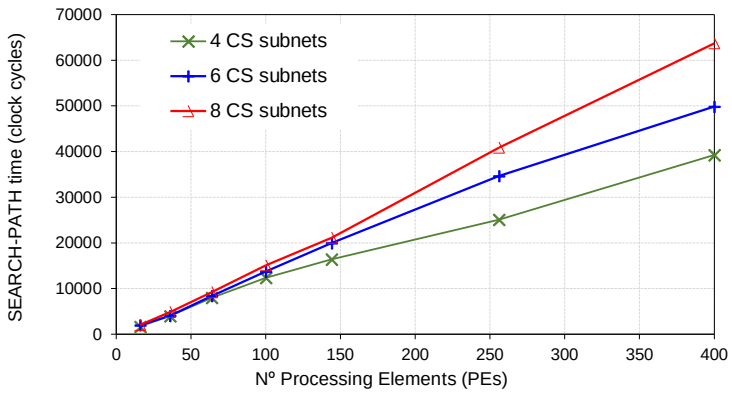
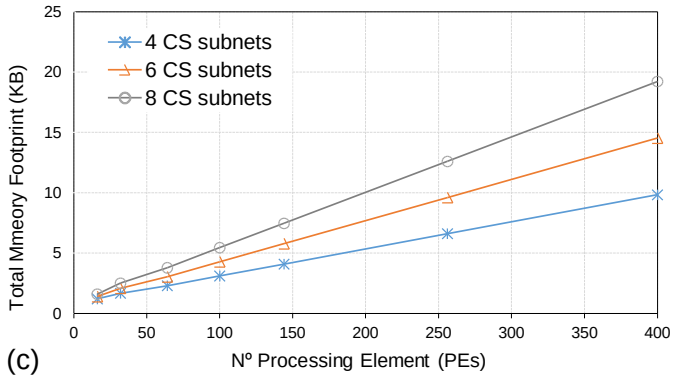


Figure 66 - SEARCH-PATH average execution time for the worst scenarios [RUA17a].

5.4.5 Memory Footprint

Figure 67 presents the memory footprint as a function of the number of the PEs. This memory usage includes data (Equation 4) and code of the *CS-Controller*. The memory requirement increases at a linear rate of 0.03, 0.05, and 0.06 KB/PE, for 4, 6, 8 SDN subnets, respectively. The total storage required for a 20x20:8 system is 19.22 KB, a small amount of memory for a many-core system.



(c)

Figure 67 - *CS-Controller* memory requirement [RUA17a].

5.4.6 Application's Latency Evaluation

Figure 68 presents the frame decoding latency of the MPEG-2 decoder using scenarios: PS without disturbing traffic (*PS Baseline*); PS with disturbing traffic (*PS+Disturbing*); *CS*.

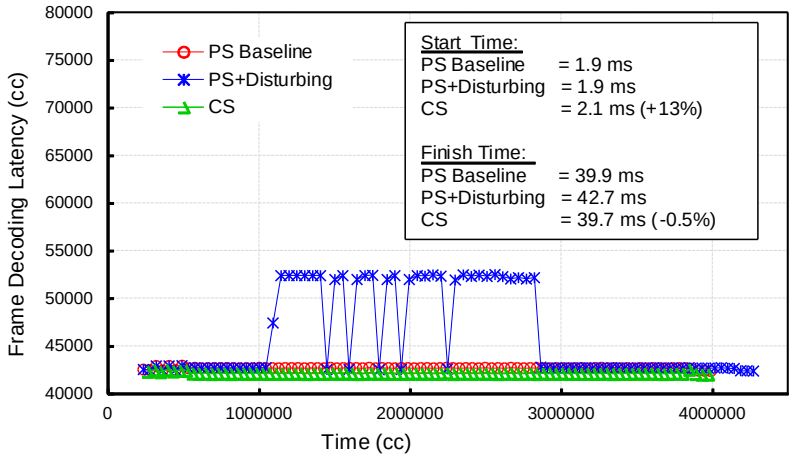


Figure 68 - MPEG-2 start time and frame decoding latency [RUA17a].

The *PS+Disturbing* scenario shares the PS NoC with packets from other applications, resulting in increased frame decoding latency and jitter. The *CS* scenario presents a constant and predictable latency, with a latency smaller than the *PS Baseline* scenario due to the streaming behavior of the CS routers, transmitting one flit/cycle (without routing and arbitration logic). Note that the application executes faster using CS, despite the delay to start. The results of Figure 68 addresses an application running for only 39.9 *ms*. The impact of the delay to start the application is even more negligible for longer execution time.

5.4.7 Comparison of the SDN-based CS to a Hardware-based Approach

The SDN-based CS was compared regarding latency to set a path and quality of paths (success rate) to a state-of-the-art search path mechanism called *Parallel-Probe* (PP) [LIU12], which is hardware implemented.

The PP method adopts a dedicated NoC responsible for finding the paths. As shown in Figure 69(a), each PE receives a PP router, connected to neighbors PP routers and locally to the SDN routers (S_Rs) (note that in this case the S_R, although it is the same hardware component, is configured by the PP router instead the *CS-Controller*). Figure 69(b) presents an example of the search method, with router 1 being the source and router 9 the target. The PP method finds the shortest path by propagating a wave of probes, which floods the PP network and unveils the shortest path by selecting the first probe to reach the target. When the first probe reaches the target, a backtracking process starts, releasing the other pre-allocated paths, and setting up the current path by configuring the S_Rs. The PP method enables to find the shortest path within a constant setup time of 3*D+6 clock cycles, where D is the Manhattan distance between the source and target PP routers.

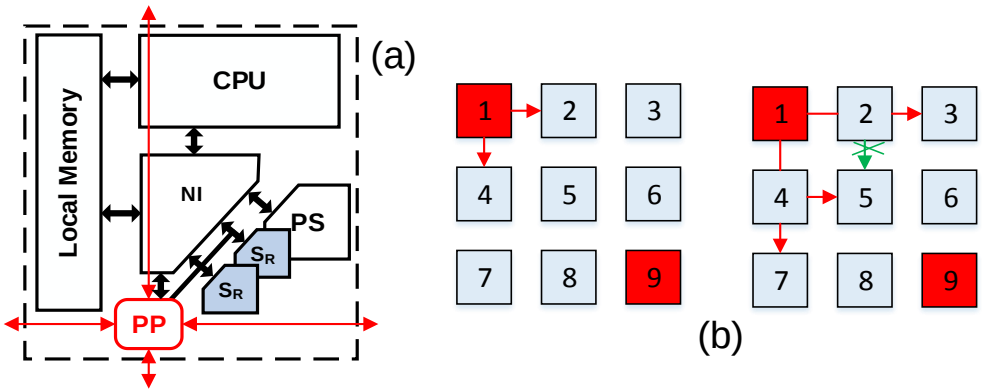


Figure 69 – (a) PE architecture including the Parallel-Probing router. (b) Example of PP algorithm [LIU12].

The *CS-Controller* acts only as a synchronizer in the PP implementation. As in the SDN implementation, the *CS-Controller* receives CS requests. These requests are stored in a FIFO because the PP network handles one propagation at a time. If there is a request in the FIFO, the *CS-Controller* handles it by sending a message to the source S_{PE} kernel to start the PP method. The message contains the target address and the subnet that the S_{PE} should use (the subnet is selected according to the subnet utilization, selecting the less used subnet). The S_{PE} starts the PP propagation by configuring its PP router. When the propagation reaches the target, the backtracking process starts. During the backtracking, the pre-allocated S_Rs not belonging to the path are released, and the S_Rs belonging to the path are configured using the programming interface. When the backtracking reaches the source PP router, it interrupts the S_{PE}'s kernel. If the search fails, the S_{PE} tries the next subnet, until finding a path. When this process finishes, the S_{PE} sends a message to the *CS-Controller*, reporting success or failure. As in the SDN implementation, after the search path process, the *CS-Controller* sends a ack/nack to the M_{PE} (the CS requester).

Table 17 presents the results, addressing many-core sizes from 36 to 256 PEs (1st column, system size-cluster size), with three SDN subnet configurations: 4, 6, 8 (2nd column). Each S_{PE} can execute simultaneously two tasks. The evaluated scenarios execute several benchmarks instances (DTW, MPEG, MPEG-2, VOPD) in such a way to have all S_{PE}s executing 2 tasks (system occupation equal to 100%), with the goal to stress the paths' diversity of subnets. The 3rd column presents the total number of requested paths by the selected benchmarks, i.e., the total number of CTPs. The 4th column corresponds to the method: PP or the proposed SDN.

Table 17 – PP and SDN evaluation, path length and connection time, for 6x6 to 16x16 many-core systems. Success rate: (*min hops* + *non min hops*)/*N# paths*.

System size: Cluster size	N# of SDN subnet	N# Paths	Method	Avg hops	Hops			Success Rate (%)	Connection time (clock cycles)		
					min	non min.	not found		Avg	Std dev	Max
6x6:3x3	4	70	PP	2.25	65	2	3	95.71	450	237	1,248
			SDN	2.24	64	2	4	94.29	2,840	1,663	8,637
	6		PP	2.31	66	4	0	100	435	252	1,800
			SDN	2.20	70	0	0	100	2,946	2,036	16,792
	8		PP	2.20	69	1	0	100	377	214	1,800
			SDN	2.20	70	0	0	100	2,913	2,103	16,284
8x8:4x4	4	127	PP	2.57	113	9	5	96.06	467	265	1,252
			SDN	2.60	119	3	5	96.06	3,765	3,111	25,810
	6		PP	2.48	122	5	0	100	420	215	1,801
			SDN	2.40	127	0	0	100	5,084	5,324	27,793
	8		PP	2.45	124	3	0	100	401	199	1,511
			SDN	2.40	127	0	0	100	5,166	5,645	29,465
12x12:4x4	4	328	PP	3.03	269	28	31	90.55	501	311	1,247
			SDN	3.02	273	21	34	89.63	5,094	5,570	61,557
	6		PP	2.85	304	20	4	98.78	443	288	1,835
			SDN	2.70	323	1	4	98.78	4,772	3,392	42,024
	8		PP	2.76	318	9	1	99.70	406	243	2,431
			SDN	2.67	327	0	1	99.70	4,468	2,514	26,077
16x16:4x4	8	623	PP	2.62	608	12	3	99.52	423	289	2,486
			SDN	2.62	620	0	3	99.52	5,453	3,522	48,542

The 5th column presents the *avg hops*, which corresponds to the average distance between a CTP. The SDN and PP methods present similar results (difference smaller than 5%), showing the effectiveness and scalability of the proposed software method compared to the hardware method. The reduced average number of hops is due to the mapping heuristic, which maps communicating tasks near to each other [SIN13b]. Next, the table presents the number of *minimal*, *nonminimal*, and *not found* paths. The SDN slightly overcomes the PP when evaluating the path length, since from 6 CS subnets all found paths were minimal for 6x6 and 8x8 systems, 98.5% for a 12x12:8 system, and 99.52% for a 16x16:8 system. The column *not found* is related to non-established paths. As expected, smaller number of SDN subnets induces a larger number of failures. The 9th column corresponds to the success rate. For small to medium systems sizes, 6 subnets were enough to find all paths. For large system sizes (12x12 and 16x16), 8 SDN subnets enabled to route more than 99% of the paths. In summary, the SDN method has a similar success rate to establish connections compared to the hardware implementation, with a slight advantage related to the path length (higher number of minimal paths).

The last 3 columns compare the latency to search the CS paths. The PP latency presents a small variation (small standard deviation values). On the other hand, the SDN latency tends to increase with the system size. The highest average latency was 5,453 and 501 clock cycles (cc) for SDN and PP, respectively. This is expected since the comparison occurs between software (SDN) and hardware (PP) implementations. If we assume CS connections established at the beginning of the application execution, with connections staying active during the application lifetime, the SDN search path latency only impacts on the application startup. For example, consider a system running at 500 MHz (T = 2ns), an average latency equal to 5,000 cc, and an application with 10 CTPs. The total latency would correspond to 100 µs, and would not be noted by the end user. We argue that SDN can be a viable option for communication management, with reduced area and management flexibility, features that hardware-centric techniques are not able to provide.

Graphs in Figure 70 detail the search path latency (Y-axis) as a function of the system occupation (X-axis), for scenario 8x8-4x4:8. All other experiments present similar behavior. As mentioned, the PP latency presents a small variation (Figure 70(a)). The average search path latency is 401 cc, and only 5 paths (3.9%) presents a latency higher than 630 cc. The latency increases when the available paths become scarce, inducing the search mechanism to explore alternative CS subnets. SDN presents a more significant variation in the search path latency (Figure 70(b)), due to the features of the Hadlock's algorithm, which increases the search space according to the failures to set a given path. The latency stays below 5,000 cc for 82.7% of the paths. As in the PP method, the SDN achieves worst latency when the system occupation increases, reaching 29,465 cc in the worst result.

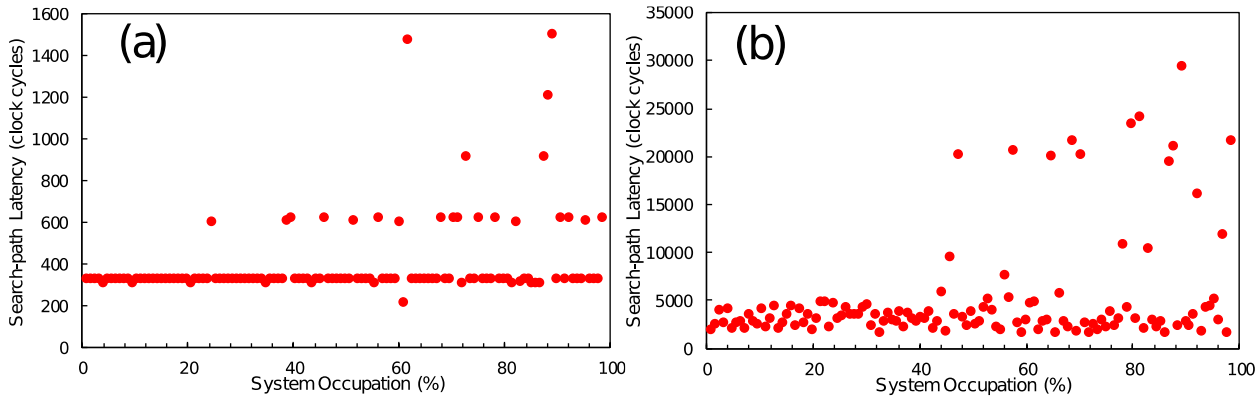


Figure 70 – Search path latency for PP (a) and SDN (b) - 8x8-4x4:8 system size.

5.4.8 Area and Power of the SDN Router

Table 18 compares the SDN router with a low-area PS router, for different flit widths. The PS router is configured as follows: 8-flit buffer depth, round-robin arbitration, XY routing, no virtual channels. The SDN router corresponds, in average, to 21% of the area and 16% of the power of the PS router. The area and power difference increase with the flit size.

Table 18 - Gate number (comb. and seq. gates), area (μm²) and estimated power (μW) for the CS and PS routers (28 nm SOI technology **@1GHz**, using the Cadence ASIC design flow) [RUA17a].

Flit width	5-port SDN router				5-port PS router			
	Comb. gates	Seq. gates	Area μm²	Power μW	Comb. gates	Seq. gates	Area μm²	Power μW
8	398	135	907	701	1079	472	3521	3683
16	517	215	1267	1154	1672	832	5948	6121
32	802	375	2011	1334	2927	1552	10921	10606

Such results show the low cost to adopt multiple physical networks based on simple design of SDN routers. Consider a PS router configured as follows: 32-bit flit width, 8-flit buffer depth, round-robin arbitration, XY routing, no virtual channels. The area of this router is 10,921 μm². As reported in the literature, the adoption of 2 virtual channels (VCs) almost doubles the router silicon area of the PS router [MEL05]. One S_R with 32-bit flit, as detailed in Figure 59, requires 2,011 μm². As the current work adopts 16-bit flits to reduce the MPN area while providing a sufficient throughput, the S_R requires 1,291 μm². Thus, the a MPN composed of 1 PS and 8 SDN subnets has a silicon area equivalent to a 2-VC packet switching NoC. Such result demonstrates the low cost to adopt MPNs compared to TDM-based NoCs, achieving similar results achieved in literature [YOO13].

The *Parallel-Probing* technique achieves better latency for search path, however, presents an additional area incurred by the presence of the dedicated PP router inside the PE instance. Such area increases according to the number of subnets and can be quantified as 800, 962 and 1,130 μm² for 4, 6, and 8 SDN subnets respectively. The overhead of the PP router is equivalent to one S_R. Thus, replacing the PP method by the SDN enable to create one additional subnet in the network, used for application data instead search path management, which contributes to increasing the communication throughput between cores.

5.5 Conclusions

This Chapter presented a background about QoS at the communication level and the contribution of this Thesis in this context.

This research investigated the pros and cons of the SDN paradigm, evaluating the proposal in a cycle-accurate many-core model. The work fills a lack in the literature by proposing a generic SDN architecture, addressing both hardware and software implementation details [RUA17a]. The reference hardware implementation (PP) enables fast connection establishment (small latency), with a small area overhead. Comparing the proposed SDN to the PP, we observe a similar path quality (i.e., average number of hops), with a slight improvement in the number of minimal paths, and higher latency. The higher latency is not an actual drawback since the latency only affects on the application startup (in the order of μs). The advantages of adopting SDN include simple hardware architectures, reusability, and management flexibility, features not available in hardware-centric approaches.

This research also proposed self-adaptive CS managed through the SDN paradigm for many-cores. This approach enabled to design a simple CS infrastructure, with configurable SDN routers

based on Elastic-Buffers. The adoption of a systemic CS management fills the gap in the literature related to the evaluation only at the NoC context. This proposition provides permanent CS for the RT applications, providing communication predictability and QoS. The search path based on the Hadlock's algorithm, plus the rich path diversity offered by the MPN, enabled the proposal to achieve a connection success rate higher than 97% for MPNs with 6 and 8 subnets executing real applications.

Future work includes the proposition and evaluation of other communication management policies using the SDN paradigm herein proposed, as fault-tolerance and security.

6 SELF-ADAPTIVE QOS MANAGEMENT AT COMPUTATION AND COMMUNICATION LEVEL

This Chapter details the main contribution of the Thesis, corresponding to the high-level management that offers self-adaptiveness for QoS at both the computation and communication levels. The Author developed this part of the Thesis during an internship in the ICT (*Institut für Computertechnik*) in TU Wien (*Technische Universität Wien*), Vienna – Austria, from April 2017 to September 2017, supported by CAPES-PDSE program. During this period, the Author deepened the knowledge related to self-awareness models with the SoC group under the supervision of Prof. Dr. Axel Jantsch [JAN17a]. Professor Jantsch, as well as the SoC group, have expertise in self-awareness, advancing the understanding about the means to implement and organize such property in the SoC environment. Related work about self-awareness related to this group and cited by this Thesis include [ANZ17][DUT16a][DUT16b]. Additionally, professor Jantsch has a significant experience in the NoC field, being as one of the Authors that formulated this concept [HEM00], and co-Author of the following works that are referenced by this Thesis: [EJA13][LIU14b][LIU15][LIU16][MA15].

The high-level QoS management herein proposed is only possible due the QoS techniques previously presented at computation and communication levels. The proposal presented in this Chapter modifies the organization of the QoS management presented in the previous Chapters. The *Cluster Scheduler* (described in Section 4.3) and the *CS Management Protocol* (described in Section 5.3) implement the high-level management for QoS at computation and communication levels, respectively. These managers were replaced by a unified QoS management herein presented, deciding by task migrations and CS at run-time.

6.1 Introduction

Many-core systems provide outstanding processing power [BOH17], but also pose challenges for temperature management, energy consumption, security, and quality of service (QoS). Due to the high amount of resources to manage and the unpredictability that many-core SoCs have, self-adaptive properties become fundamental to address such challenges [DUT16a].

Related works [QUA16][PAR14][MAN10][RUA15a][WIN11][ABO13][JOV13][RUA17a] [RUA17b][JUN14] on self-adaptation mechanisms for QoS in many-core SoCs focus on techniques addressing either communication (NoC) or computation (CPU). The focus of this proposal is to ensure QoS for soft real-time applications through a self-adaptive QoS management. The QoS management is aware of system's communication and computation resources and acts reactively and proactively at both levels. The QoS management receives QoS fulfillment monitoring data (deadline miss, latency miss) and an application QoS feedback (notifications about run-time workload changes), enabling the QoS management to act reactively according to the severity of the events. Additionally, this Thesis proposes an Application Profile Learning (APL) technique, which on-the-fly learns about the communication and computation profile for each RT task. This feature

enables the QoS management to assume also a proactive behavior guided by the computation and communication that each RT application demands.

The main contributions of the proposal presented in this Chapter are as follows:

- A self-adaptive QoS management for soft real-time applications that acts reactively and proactively, and covers both computation and communication levels;
- An Application Profile Learning mechanism, which traces the application tasks' profile at run-time and provides this information to the QoS management.

6.2 State-of-the-Art

This Section discusses related works according to the two contributions: the APL at the scope of application profile extraction, and self-adaptation for QoS of performance.

6.2.1 Related Works in Application Profile Extraction

Works [JUN14][QUA16] propose to obtain the applications' task graph mixing design-time and run-time steps, a dynamic mechanism has access to the application graph, using it to optimize run-time decisions, such as application remapping. This approach simplifies the work of run-time techniques because it provides a detailed application profile. However, it inserts more complexity during the application development; hence, the developer is in charge to provide the application profile correctly to the system. Ganeshpure et al. [GAN13b] propose a full run-time technique that extracts the communication task graph of the applications. A middleware implements this extraction by observing the execution phases for each task.

Our proposal adopts a more flexible approach. As in Ganeshpure et al. [GAN13b], the middleware (kernel) extracts the behavior of the tasks. Differently from [GAN13b], which uses less than 200 iterations to profile the application, our technique is continuously learning about the application profile due to continuous monitoring. This feature enables to support applications with dynamic behaviors, i.e., the workload changes at run-time (common in multimedia applications). Additionally, we are concerned with scalability, as the APL follows a hierarchical organization with monitors at the task level (at each core) that send information to the QoS management (cluster manager).

6.2.2 Related Works on Self-Adaptation for QoS

Several proposals in the literature provide run-time QoS mechanisms for many-core systems, with self-adaptive techniques, targeting resource management (dynamic mapping, task migration, task scheduling, flow priority, and CS. Table 19 presents the self-adaptive QoS proposals most related to this Thesis.

Works [QUA16][PAR14][RUA16a][JUN14] adopt task remapping/migration to answer to workload changes or real-time violations, addressing QoS at computation level. Some works

[JUN14][QUA16] adopt a hybrid task remapping heuristic, assuming that application characteristics are known at design-time. In contrast, we assume the application set is unknown at design-time. The proposed APL enables the system to learn about the running applications, creating an online profile. In Jung et al. [JUN14], applications can tune the workload at run-time by using an API. This feature provides high flexibility to an application to change its workload. Our work also enables such workload reconfiguration by an API, where each task can change its real-time constraints.

Table 19 – Related Works on Self-Adaptive QoS for Many-Cores

Works	QoS Focus	Method	Technique
[JUN14]	Computation	Dynamic Specification Behavior and Dynamic Mapping	Task Remapping
[QUA16]	Computation	Dynamic mapping based on prediction	Task Migration
[PAR14]	Computation	Hierarchical Scheduler	Task Migration
[RUA16a]	Computation	Dynamic Task Scheduler	Task Migration
[MAN10]	Communication	Bandwidth self-adaptation	Flow priority
[JOV13]	Communication	Expose NoC QoS services by a API	Flow priority / CS
[ABO13]	Communication	Proactive CS establishment	CS
[RUA15a]	Communication	Self-Adaptive QoS Management	Flow priority / CS
[RUA17a]	Communication	SDN-based self-adaptive QoS management	CS

Most works assume a hierarchical management organization [QUA16][PAR14][RUA16a], which distributes the management load by adopting a cluster-based organization, with a set of cores managed by a cluster manager. Our proposal also adopts a hierarchical organization, with slave cores running the user's applications and sending monitored/feedback data to a cluster manager, which executes the QoS management.

While the aforementioned works satisfy the computation constraints and also try to reduce the communication cost by mapping tasks closer to each other, these approaches address communication QoS indirectly and are not able to handle unpredictable events that can disturb the traffic in the network, e.g., a task migration packet crossing the communication path between two RT tasks. To mitigate this disturbance works focusing only on computation QoS should migrate the affected tasks to other processors, instead to act directly on the communication level.

Works [MAN10][RUA15a] address QoS at the communication level. Several works develop techniques to implement a QoS-driven infrastructure considering only the NoC (e.g. [WIN11][ABO13]). Joven et al. [JOV13] expose the communication QoS support to the software layer enabling the developer to define the QoS constraints. Abousamra et al. [ABO13] observe the message requests to set proactive CS, used for future message deliveries. Authors in Mangano et al. [MAN10] propose a self-adapting mechanism that exposes the hardware through a set of registers to program the QoS constraints for a bus-based SoC. Work of Ruaro et al. [RUA15a] proposes a self-adaptive flow priority management and CS establishment based on latency and throughput constraints. Authors of [RUA17a] propose a run-time CS based on a Software-Defined Networking (SDN) paradigm, enabling to establish CS paths during the whole application lifetime.

As also can be observed in related works, task migration and CS stand out as techniques to provide QoS at computation and communication levels, respectively. The novelty of the proposal presented in this Chapter is a unified self-adaptive QoS management addressing QoS of computation (task migration) and communication (CS) for soft real-time applications. Additionally, we propose a dynamic Application Profile Learning technique, which enables to take proactive decisions.

6.3 Application Profile Learning (APL)

This Section details the Application Profile Learning (APL). The APL has its implementation divided into two hierarchical levels. The lower level, implemented in the S_{PE}s, and the higher level, implemented in the M_{PE}s. At each S_{PE}, the kernel monitors the tasks' profile at run-time. The kernel monitors for each task *t* the following parameters: (*i*) computation, T_p, the part of time where *t* is using the CPU; (*ii*) communication, T_m, the part of time where *t* is blocked, waiting for a requested message from a producer task; (*iii*) idle time, T_i.

The monitoring extracts the relative amount (percentage) of T_m, T_p, and T_i, for each task periodically, over non-overlapping windows, where $T_m + T_p + T_i = 1$. The task scheduler computes T_p and T_i. The communication API computes T_m. T_m is computed from the perspective of the consumer task, evaluating the time spent between the requisition of a message until its reception. T_m is a function of three factors: (*i*) the time spent on the producer task to generate the requested message; (*ii*) the message size; (*iii*) the NoC congestion. Note, that the communication percentage is computed only for the received messages because the kernel adopts a non-blocking send operation (produced but not consumed messages are locally stored in a buffer). In scenarios with congestion in the NoC, the observed communication profile tends to increase when the consumer task spends more time waiting for messages. This behavior helps to mitigate network congestion because the management will pay more attention to affected communications.

Each S_{PE} sends the monitored profile periodically to its M_{PE}, which implement the APL upper level. The M_{PE} handles the received profile by applying an accumulated mean of the received profile with the past profiles. The self-adaptive QoS management uses the resulting value to estimate the profile of each task.

Consider as an example the task graph of Figure 71(a) and assume that each task executes the same computation load periodically. Figure 71(b) presents the profile graphs according to the APL method. Task A does not receive packets from other tasks, thus $T_m = 0$. Tasks B and C receive packets from task A, resulting in a mixed profile with T_p = 26% and T_m = 15% (tasks B and C have similar graphs). Finally, task D has two communication flows, receiving packets from tasks B and C. In task D the communication is higher, as depicted in the graph of Figure 71(b).

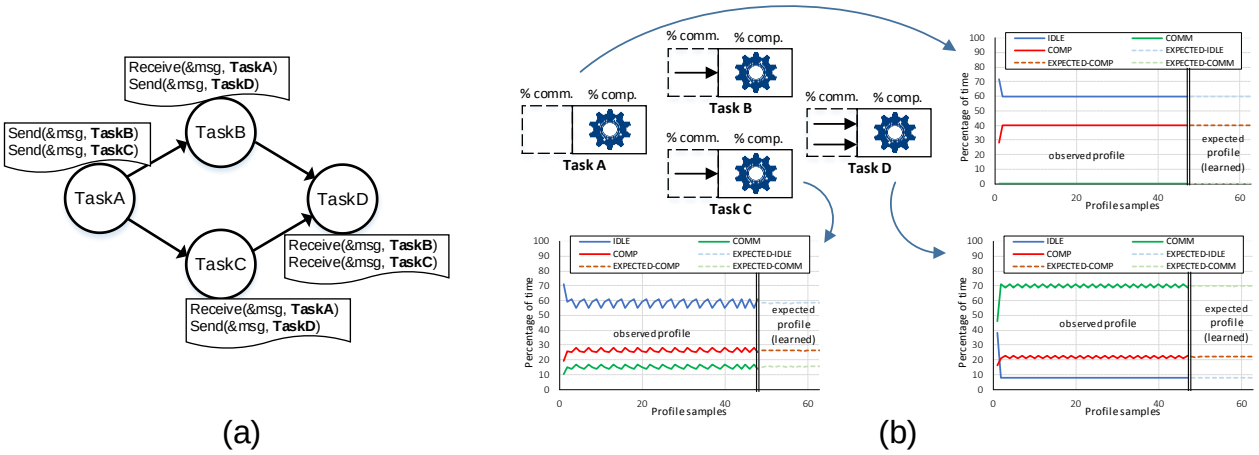


Figure 71 – (a) Example of a task communicating graph of an application. (b) Overview of the application profile learning method.

6.4 Self-Adaptive QoS Management

This Section details the self-adaptive QoS management. The approach is distributed, implemented inside each M_{PE}. It adopts the ODA (Observe, Decide, Act) paradigm [HOF13]. The ODA method includes a loop that is constantly aware of the system status. It is generic and can be adapted to different many-core architectures. Figure 72 summarizes the main contribution of this proposal, by presenting an overview of the self-adaptive QoS management.

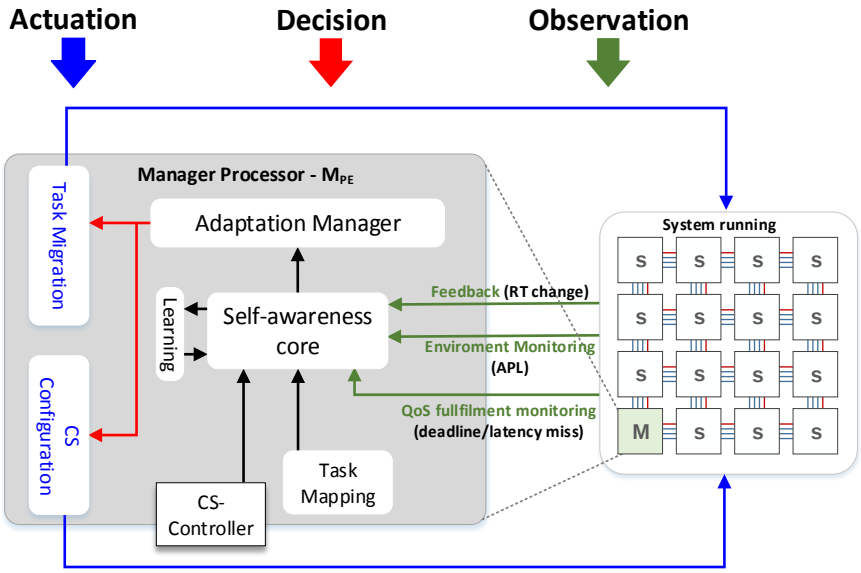


Figure 72 – Organization of the self-adaptive QoS management.

Observation: There are three message classes sent by S_{PE}s to the QoS management: *feedback*, *environment monitoring*, and *QoS fulfillment monitoring*. *Feedback* messages provide performance figures related to the tasks. The tasks generate the *feedback* messages, reporting changes in the RT constraints (period, deadline, execution time). The tasks may update at run-time the RT constraints using the *RealTime* API, enabling flexible workloads [RUA16a][JUN14]. The *environment monitoring* messages allow the manager to gather periodic information about the status of the resources and applications. These messages include the APL data. *QoS fulfillment monitoring* messages warn the

manager about violations of the QoS fulfillment. These messages address both communication (packet latency miss) and computation (deadline miss) levels.

Decision: The *self-awareness core* implements the awareness by accessing: (*i*) observation messages; (*ii*) tasks' location (provided by the task mapping heuristic); (*iii*) CS paths (provided by the CS-Controller); (*iv*) statistics obtained from the APL. This rich information set allows for accurate run-time decisions. The *self-awareness core* acts as a trigger deciding which system component is the target of the adaptation and when the adaptation occurs. The *self-awareness core* decides reactively as well as proactively. The *Adaptation Manager* (AM) handles the decisions, reconfiguring the system resources. Thus, the AM acts as the actuator managing the adaptation process.

Actuation: The AM manages task migration and CS (establishment/releases) protocols, according to decisions made by the self-awareness core. These protocols change physically the resources of computation and communication. The AM also ensures that the resources will be correctly updated after the adaptation to be used by the self-awareness module in future decisions.

Figure 73 presents the self-adaptation flow. The method is similar to a health check-up, with *quick* and *complete* check-ups. The quick check-up acts like when a symptom appears, such as a feedback message notifying an RT constraint modification or a deadline/latency miss, leading to reactive actions. According to the received message, there are two quick check-up functions, one to deal with computation events (RT change and deadline miss) and the other one to deal with communication events (latency miss). The complete check-up enables to evaluate the application behavior in detail, analyzing all application's tasks instead only one or one *CTP*. It may be triggered by the quick-checkup function leading to a reactive action, or it is invoked periodically (by the *routine overhaul* trigger), potentially leading to proactive actions.

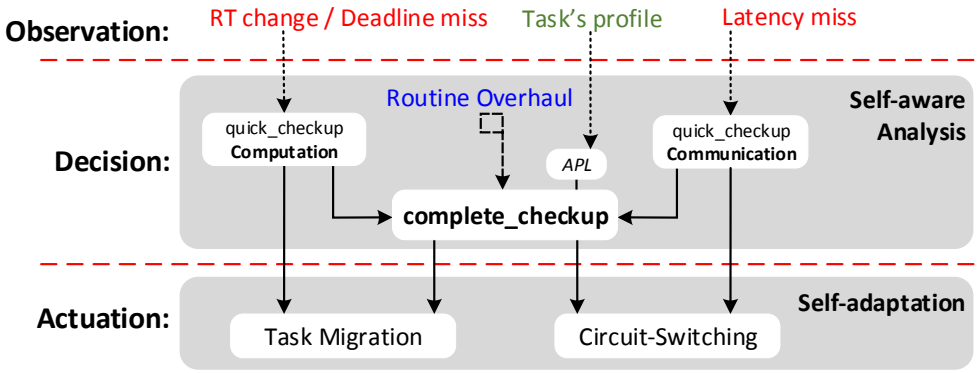


Figure 73 – Self-adaptation QoS management flow, executed by the manager processors.

The self-adaptive QoS management has its decision part implemented by the quick and complete check-up algorithms. The algorithms adopt the following design-time parameters:

1. ***cpu_TH***: maximum allowed CPU utilization per processor;
2. ***comp_profile_TH***: threshold percentage used to recognize a high computation profile task;

3. ***comm_profile_TH***: threshold percentage used to recognize a high communication profile task;
4. ***comp_profile_sum_TH***: threshold percentage used to fire a proactive adaptation, corresponding to the total load of the tasks on a given processor;
5. ***deadline_TH***: maximum percentage of deadline misses in a sampling period;
6. ***latency_TH***: maximum number of latency misses for a given Communicating Task Pair (CTP).

6.4.1 QUICK-CHECKUP Algorithms

The *feedback* and *QoS fulfillment monitoring* messages fire the quick check-up algorithms, presented in Algorithms 2 and 3.

The goal of the QUICK-CHECKUP-COMPUTATION algorithm is to evaluate when it is necessary to migrate some task due to a computation interference. The algorithm inputs are the task identification (*task*) and the CPU address executing the task (*task_cpu*). Line 1 computes the CPU utilization where the task is running, and line 2 calculates the deadline miss rate (percentage) for the task. The deadline miss rate is the relationship between the number of missed deadlines divided by the number of tasks' periods since the last application adaptation. Line 3 of the Algorithm verifies the task status using these two parameters (*cpu_util* and *deadline_miss_rate*). If the *cpu_util* or the *deadline_miss_rate* is higher than the predefined thresholds (*cpu_TH* and *deadline_TH*), the decision is to migrate the task. If the migration fails (no available processor), the COMPLETE-CHECKUP (line 5) is called.

Algorithm 2: QUICK-CHECKUP-COMPUTATION

```
Input: task, task_cpu
begin
1:  cpu_util = get_cpu_utilization(task_cpu)
2:  deadline_miss_rate = get_deadline_miss_rate(task)
3:  if cpu_util > cpu_TH or deadline_miss_rate > deadline_TH then
4:      if task_migration(task) = FALSE then
5:          COMPLETE-CHECKUP(task's application)
6:      end if
7:  end if
end
```

The goal of the QUICK-CHECKUP-COMMUNICATION algorithm is to evaluate when it is necessary to establish CS connections for some *CTPs* due to a communication interference. The algorithm receives as input a *CTP*, with a producer (*prod_task*) and consumer (*cons_task*) task identifiers. Line 1 obtains the total number of latency misses since the last application adaptation. When the latency misses exceed *2*latency_TH* (line 2), the COMPLETE-CHECKUP is invoked. If the latency misses exceed *latency_TH* (line 4) the algorithm tries to establish a CS path between *prod_task* and *cons_task*. Note, that in Algorithm 3 the call of the COMPLETE-CHECKUP only occurs if the CS establishment fails.

Algorithm 3: QUICK-CHECKUP-COMMUNICATION

Input: *prod_task, cons_task*
begin
1: *ctp_latency_count* = get_ctp_latency_number(*prod_task*, *cons_task*)
2: **if** *ctp_latency_count* > *2*latency_TH* **then**
3: COMPLETE-CHECKUP(task's application)
4: **else if** *ctp_latency_count* > *latency_TH* **then**
5: **CS_configuration(***prod_task*, *cons_task***)**
6: **end if**
end

6.4.2 COMPLETE-CHECKUP Algorithm

The main goal of the proposed QoS management is to reduce the reactive actions, acting proactively when possible to avoid future QoS violations. Algorithm 4 presents the COMPLETE-CHECKUP algorithm, which receives as input an *application* identifier. This algorithm has two operating modes, reactive and proactive.

Algorithm 4: COMPLETE-CHECKUP

Input: *application*
begin
1: *task_migration_list* = ∅
2: *task_rank[]* = computes_task_score(*application*)
3: **if** *task_rank[]* ≠ EMPTY **then**
4: *selected_task* = get_high_task_score(*task_rank[]*)
5: **task_migration(***selected_task***)**
6: **else**
7: **for** $t_i \in$ *application* **do**
8: *comp_task_num* = get_num_comp_task(t_i_*cpu*)
9: **if** comp_task_num > 1 **or** get_comp_sum(t_i_*cpu*) ≥ *comp_profile_sum_TH* **then**
10: **task_migration(**t_i**)**
11: *task_migration_list* = t_i
12: **end if**
13: **end for**
14: **for** $t_i \in$ *application* **and** $t_i \notin$ *task_migration_list* **do**
15: **if** get_comm_profile(t_i) ≥ *comm_profile_TH* **or** get_comm_profile(t_i) ≥ get_comp_profile(t_i) **then**
16: **for** $ctp_i \in C$ which t_i is consumer **do**
17: *prod_task* = get_producer(ctp_i)
18: **if** prod_task ∉ task_migration_list **and** ctp_i = *PS* **then**
19: **CS_configuration(**prod_task, t_i**)**
20: **end if**
21: **end for**
22: **end if**
23: **end for**
24: **end if**
end

The activation of the reactive mode occurs when a quick checkup algorithm fails, including the code lines 2 to 5. Line 2 ranks the application tasks according to the QoS violation severity, using Equation 9.

$$r_T = d_m + l_m + 10(u_{CPU} > cpu\_TH\ ?\ 1 : 0\ ) \quad (9)$$

where: r_T is the task rank, d_m is the number of deadline misses, l_m is the number of latency misses, u_{CPU} is the CPU utilization where the task is executing.

According to Equation 9, the rank of a given task is higher when it is running on a CPU with a utilization higher than *cpu_TH*. Line 4 selects the most critical task, (i.e., the one with the highest rank), and the task migration is fired to *selected_task* (line 5). Note that the reactive mode acts only in one task of the application. As one single QoS adaptation can affect the whole application performance, gradual steps are preferable to simultaneous adaptations.

The activation of the proactive mode occurs periodically (lines 7-23). The trigger to activate the COMPLETE-CHECKUP algorithm in this mode is the *overhaul routine*, which calls the COMPLETE-CHECKUP at the end of ten hyper-periods of the application. This number is a trade-off, a higher value reduces the COMPLETE-CHECKUP calls, delaying the time to take proactive actions; and smaller values increase the M_{PE} CPU usage. The proactive mode starts as in the reactive mode, ranking the tasks according to Equation 9. As the COMPLETE-CHECKUP was invoked by the *overhaul routine* and not by quick checkup algorithms, it is expected that the *task_rank[]* set be empty, i.e., all application tasks are fulfilling their constraints. The proactive mode acts first on the computation (lines 7-13) and then on the communication (lines 14-22).

At the computation level, for each task t_i of the *application*, the algorithm verifies if there are more than two high computation tasks in t_i's core (tasks exceeding *comp_profile_TH*), or if the sum of the computation profile for all tasks sharing t_i's core exceeds *comp_profile_sum_TH* (line 9). If true, t_i is migrated to a free processor (line 10), and the task identifier is added to the set *task_migration_list*. The goal is to reduce the CPU sharing between high computation tasks proactively. Also, as tasks may change their RT constraints dynamically, this action can prevent deadline misses when a given task increases the CPU utilization.

At the communication level, for each task t_i of the *application* that is not in the *task_migration_list*, the algorithm verifies if t_i's communication profile exceeds *comm_profile_TH* or is higher than the computation profile (line 15). If true, t_i is a candidate to have its communication mode changed to CS. The loop in lines 16-21 sets CS for each *CTP* that has t_i as a consumer task. Line 18 verifies if the producer task is not in *task_migration_list* and if communication is assigned to the PS network. If this condition is true, a CS is established.

In summary, the proactive QoS actions try to reduce the CPU sharing between high computation tasks and to establish CS mode between tasks with a high communication profile.

6.5 Results

The many-core system is modeled at the RTL level (VHDL and SystemC), and all software components are implemented in C language. The RT benchmarks correspond to real applications' previously addressed, as DTW and MPEG-2. Synthetic tasks run in parallel on the system with the purpose to induce computation and communication disturbances. The experiments adopt the following parameters: *cpu_TH*=99%, *comp_profile_TH*=50%, *comp_profile_sum_TH*=75%, *comm_profile_TH*=10%, *deadline_TH*=1%, *latency_TH*=2.

6.5.1 Comparison with Single Objective QoS Managers

This section compares this proposal with works covering self-adaptive QoS of communication [RUA15a] and computation [RUA16a]. Figure 74(a) shows a mapping scenario using the DTW application with interference at both computation (task *A*) and communication levels (df_1). The interference of computation occurs from 20ms to 55ms, where task *A* increases its utilization from 10% to 50%. The interference of communication occurs from 66mm to 90ms.

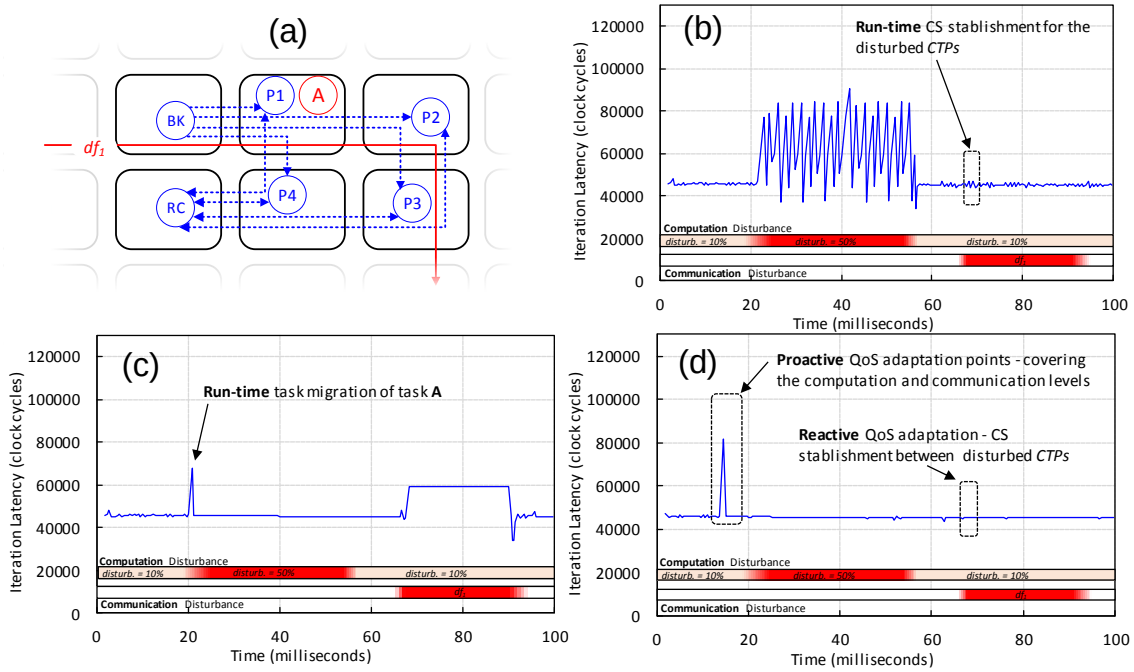


Figure 74 - Comparison with [RUA15a] and [RUA16a]: (a) scenario setup; (b) iteration latency of [RUA15a]; (c) iteration latency of [RUA16a]; (d) iteration latency of the proposed work.

Figure 74(b) shows results from [RUA15a]. The work establishes run-time CS for the disturbed *CTPs*, but it cannot mitigate the computation interference induced by task *A*. Figure 74(c) shows the results from [RUA16a], which counteracts the computation disturbance by migrating task *A* to another processor, but the communication interference remains. Figure 74(d) shows the results of the proposed work, which mitigates both computation and communication interference using proactive and reactive actions.

6.5.2 Application Profile Learning Overhead

The APL is implemented at the M_{PE} level since the learning process is spread across multiple manager processors. However, at the S_{PE} level, the APL can penalize the tasks' execution time due to the monitoring process. The kernel adopts two actions to minimize this overhead: (*i*) the monitored profile transmission occurs preferably in idle periods of the S_{PE}; (*ii*) each S_{PE} uses a different counter to trigger the sending of the information, thus distributing the monitoring load.

Figure 75(b) compares the overhead of the APL on application's execution time with a different number of tasks running at the same PE. This scenario uses two applications, Figure 75(a): DTW (Dynamic Time Warping algorithm recognizing 2500 patterns), and MPEG-2 (to decode 500 frames/audio arrays). The functionality of the application is not relevant, only the number of tasks per PE. The APL messages are transmitted from S_{PE} to M_{PE} at each 10 ms.

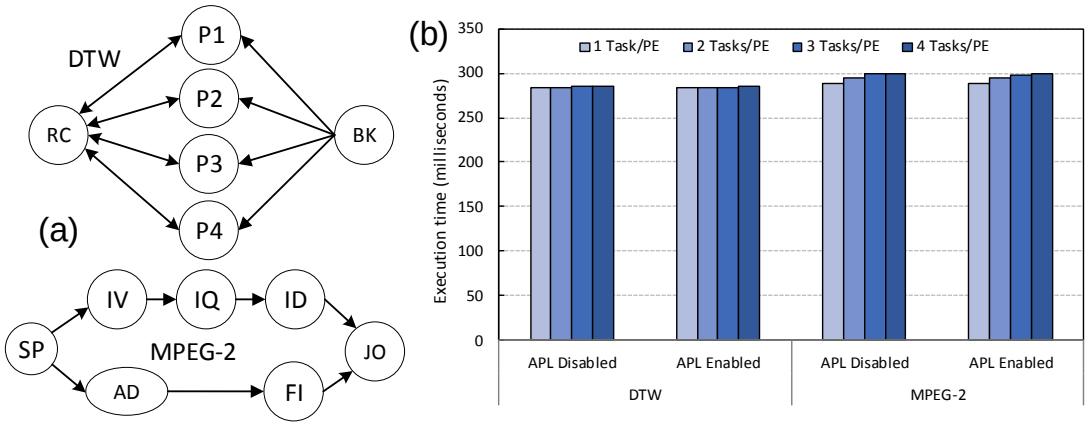
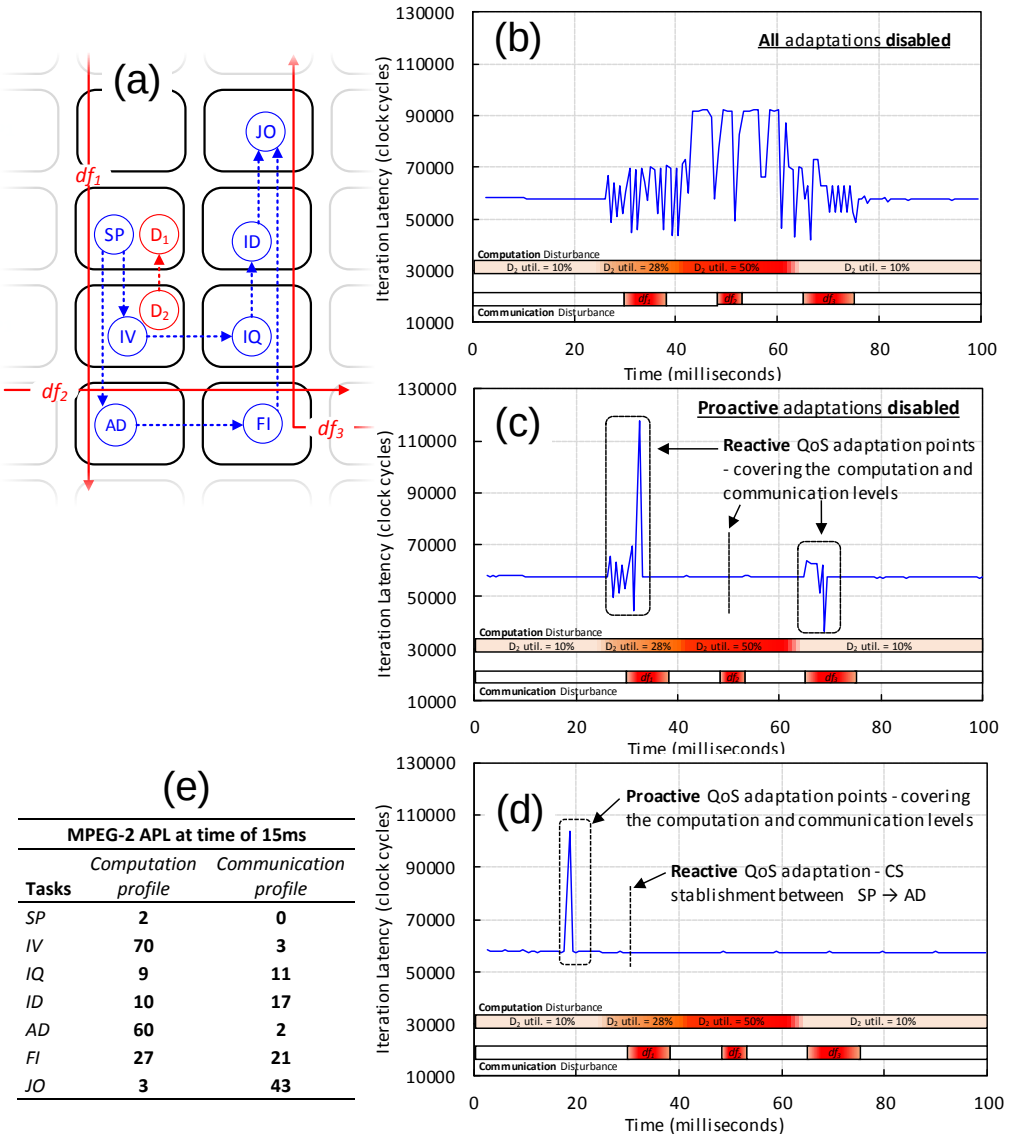


Figure 75 - (a) DTW and MPEG-2 application task graphs. (b) Overhead evaluation of the application profile learning.

All tasks in the "APL enabled" scenarios have the APL monitoring enabled, with an expected overhead at each core increasing according to the number of tasks per PE. However, the results show that the impact on the application execution time is negligible. The worst overhead is achieved in the MPEG-2 scenario, with 3 tasks/PE, corresponding to an increase of 0.0024% in the application execution time.

6.5.3 Self-Adaptive QoS Adaptation Evaluation

This subsection evaluates the proposed QoS management. Figure 76(a) shows the MPEG-2 task mapping, represented by blue circles. Tasks D_1 and D_2 share the CPU with tasks *SP* and *IV*, respectively. They belong to another QoS application, exemplifying a disturbance at the computation level. The red arrows denote disturbances of communication, other messages that interfere with the application flows. On the y-axis of Figure 76(b)-(d) shows the latency of one iteration of the MPEG-2 application, measured at the task *JO* (joint). The QoS constraint is 5,800 clock cycles, corresponding to the time to decode one audio/image frame of 576 bytes. Bars below the x-axis represent the CPU utilization of task D_2 (computation disturbance) and when the disturbing flows occur (communication disturbance).



(e)

MPEG-2 APL at time of 15ms		
Tasks	*Computation profile*	*Communication profile*
SP	**2**	**0**
IV	**70**	**3**
IQ	**9**	**11**
ID	**10**	**17**
AD	**60**	**2**
FI	**27**	**21**
JO	**3**	**43**

Figure 76 - Evaluation of the self-adaptive QoS management over the MPEG-2. (a) App. mapping. (b) No adaptation, deadline miss = 20.3%. (c) Only reactive adaptations, deadline miss = 2.3%. (d) Proactive and reactive adaptations, deadline miss = 0.5%. (e) APL for MPEG-2 at 15ms of simulation.

The first evaluated scenario has all QoS adaptations disabled - Figure 76(b). When the CPU utilization of task D_2 increases, at 25 and 40 ms, the latency increases due to the CPU sharing. In the same way, disturbing flows affect the latency. Note that when flow df_3 is active the task D_2 presents a low CPU utilization and the latency also increases. This result shows that the communication disturbance also impacts the QoS constraint.

The second evaluated scenario (Figure 76(c)) activates only reactive adaptations. When D_2 increases its CPU utilization from 10% to 28%, the total S_{PE} utilization reaches 98% (28% + 70% from task *IV*). The task migration is not immediately triggered because the CPU utilization remains below *cpu_TH*. Thus, task *IV* starts to generate deadline misses, and the QoS management decides to migrate task *IV* to a free processor at 31.4ms. Also, flow df_1 induces latency misses in the flows *SP→IV* and *SP→AD*, making the QoS management to decide to establish CS for these flows at 30.3

and 32ms. There is no impact on the latency when D_2 increases its utilization to 50% because task *IV* was previously migrated. Flow df_2 induces latency misses in the flow *AD→FI*, resulting in a new CS establishment at 49.2ms. As the CS establishment for one *CTP* affects in average 150 clock cycles of the application's latency, its effect is not perceptible in the graph. Finally, flow df_3 starts, disturbing three MPEG-2 flows: *IQ→ID*, *FI→JO*, and *ID→JO*. The consequence is several latency misses, and CSs are reactively established for all penalized pairs at 66.5, 67.4 and 68.5 ms. Note that *after* the reactive adaptations the QoS level is restored.

The third evaluated scenario activates proactive and reactive adaptations - Figure 76(d). The first call to the COMPLETE-CHECKUP occurs at 15ms due to the execution of the *overhaul routine*, with the learned profile presented in Figure 76(e). The COMPLETE-CHECKUP decides to migrate proactively task *IV* because it is sharing the CPU with D_2 and the sum of its computation profile is higher than *comp_profile_sum_TH* (at 17,6ms). Additionally, according to the obtained profile, tasks *ID*, *FI*, *JO*, and *IQ* have a high communication profile (higher than *comm_profile_TH*). Therefore, the QoS management proactively establishes CS for the flows: *IQ→ID* (18.5 ms), *AD→FI* (18.6 ms), *ID→JO* (19.1ms), *FI→JO* (19.7 ms), and *IV→IQ* (23 ms). This scenario also has a reactive QoS adaptation, which is a CS establishment between *SP→AD* at 32 ms due to the disturbance caused by df_1. The QoS management did not establish CS's previously since the APL revealed that *AD* has a communication profile smaller than the *comm_profile_TH*.

Comparing deadline miss rates in all three scenarios, we observe a miss rate of 20.3% when QoS is disabled, 2.3% when only reactive adaption is used, and 0.5% with both proactive and reactive adaptions.

While Figure 76 shows the MPEG-2 case in detail, Figure 77 summarizes results for 8 benchmarks. It compares execution time (a), and deadline miss rate (b), for: (*i*) baseline scenario; (*ii*) *DIST* - disturbances and no QoS mechanism; (*iii*) *REACT* - disturbances and only reactive QoS enabled; (*iv*) *P+R* - disturbances and both QoS mechanisms enabled (proactive + reactive). The disturbances consist of 2 tasks providing computation disturbance (randomly mapped within PEs running benchmark's tasks), and 3 disturbing communication flows. Compared to the baseline execution time, the *DIST* increases the execution time, on average, by 224.7% (severe disturbance). Applying QoS management, the execution time is restored close to the baseline: 13.8% for *REACT* and 2.4% for *P+R* above the baseline. Note that *REACT* exhibits a higher average execution time than *P+R*, highlighting the benefit of proactive actions. As Figure 77(b) shows, the deadline misses were reduced, on average, by 98% for *REACT*, and 99.5% for *P+R*, with a rate below 0.6% for all *P+R* benchmarks.

Those experiments show the synergy between proactive and reactive actions. The proactive adaptation reconfigures the system according to the application profile learning, preventing/minimizing future deadline misses due to interference. The reactive actuation operates during the learning period or responds to unpredictable disturbing events.

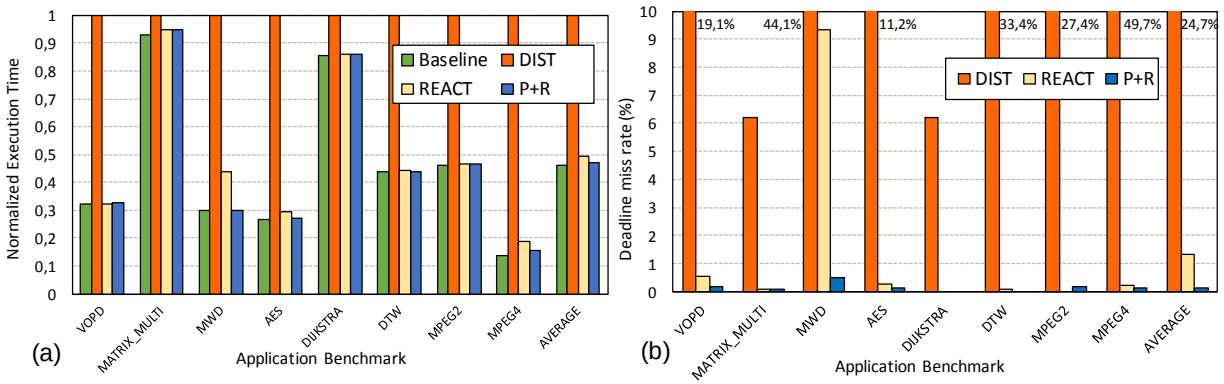


Figure 77 - Benchmark evaluation: (a) execution Time; (b) deadline miss rate.

6.5.4 Self-Adaptive QoS Trade-off

Subsection 6.5.1 showed that the QoS manager must act in both computation and communication conjointly. Subsection 6.5.3 presented the effectiveness of the proposed method to provide QoS for a set of benchmarks in the presence of disturbing events. This last evaluation stresses the proposed method, with all PEs executing tasks with QoS constraints, and increasing the CPU utilization gradually. This evaluation adopts three synthetic applications, with different profiles: (*i*) *COMP* – computation intensive (66% comp., 10% comm.); (*ii*) *COMM* - communication intensive (32% comp., 61% comm.); (*iii*) *HYB* – hybrid, a mix of computation and communication profile (45% comp., 30% comm.).

The simulated workload corresponds to 5 applications executing simultaneously: 1 *COMM*, 1 *COMP*, and 3 *HYB*. The applications were randomly mapped in a cluster with 16 PEs (1 M_{PE} and 15 S_{PE}s). The S_{PE}s were configured to run 2 tasks concurrently. To use all S_{PE}s of the cluster, all S_{PE} received two tasks. With such configuration, task migration is disabled due to the full system usage. All applications start their execution at the beginning of the simulation, with a warm-up period of 10 ms. After the warm-up period, each S_{PE} has 10% of CPU utilization (5% from each mapped task). As the simulation advances, all tasks increase its CPU utilization steadily. The utilization increases by 5% for each task at each 10 ms, resulting in a total CPU utilization increase of 10% at each 10 ms.

Figure 78(a) presents the deadline miss rate for this experiment (y-axis: percentage of deadline misses, x-axis: CPU utilization per S_{PE}). The deadline miss rate remains 0 up to 80% of CPU utilization. After 80% of CPU utilization, all applications start to miss deadlines. Due to the system unpredictably behavior with a higher CPU utilization, some application misses more deadlines than others (*Apps 3, 4, 6*). *As task migration is not possible due the full system occupation, this result showed the effectiveness of the task scheduler to ensure QoS at computation level*. The proposed method can ensure QoS for several applications running concurrently, even with a large CPU utilization (80% in this scenario).

Figure 78(b) presents the latency miss rate. All applications start their execution communicating using the PS NoC. As the simulation advances, the QoS management aided by the application profile learning, identify the applications' profile and set CS to the *COMM* and *HYB*

applications, reducing the latency miss to less than to 0.5%. The exception is the *COMP* application (*App 5*), which continues using the PS NoC because it does not satisfy the requirements for proactive CS. With the increase in the CPU utilization, this application receives one latency miss at each 10ms, resulting in latency miss rate of 12.5%. As *App 5* is computation intensive, this latency miss does not impact the deadline misses, as can be observed in Figure 78(a), not justifying the establishment of CS.

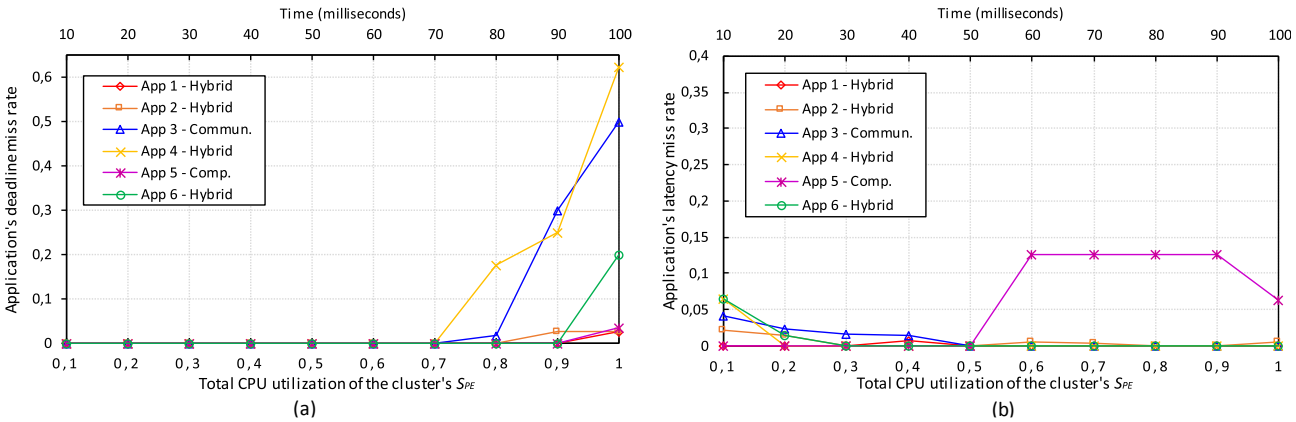


Figure 78 – QoS provisioning trade-off: (a) Deadline miss rate; (b) Latency miss rate.

This experiment enabled to observe the QoS provisioning trade-off at the computation and communication levels in a stressed scenario. The computation QoS starts to be affected after 80% of CPU utilization. The method ensures communication QoS for all communication sensitive applications by establishing CS at run-time. The adoption of an MPN provides sufficient CS paths even in a cluster with all S_{PE}s running the maximum number of tasks.

6.6 Conclusions

This Chapter proposed a dynamic profiling and self-adaptive QoS management for soft real-time applications. A run-time application learning profiling (APL) technique allows the QoS management system to take proactive actions, and when necessary, react to cope with the interference induced by the dynamic workload. The low overhead of the APL demonstrates that run-time techniques can be used to characterize applications.

The results show two relevant facts. First, mechanisms for regulating both communication and computing are essential for the overall QoS management. Second, proactive techniques can ease the avoidance of future deadlines and latency misses.

While we have demonstrated the benefits of the proposed APL and QoS management in a specific setting, there is little in the NoC, the PEs, and the middleware, that we require for our techniques to work. Hence, we expect similar benefits on very different platforms, which, however, should be demonstrated in future work.

7 CONCLUSION

This Thesis incrementally proposed and developed solutions to meet QoS constraints. Initially considering computation and communication in a separate way (Chapters 4 and 5), and then merging the techniques to build a comprehensive solution (Chapter 6).

7.1 QoS at the Computation Level

Chapter 4 explored QoS techniques at the computation level, including task migration and a dynamic task scheduler. The task migration protocol has as main feature the absence of code checkpoints; adopts task recreation, which reduces the memory overhead due to the absence of task replicas in potential target PEs; and transmits the task messages to the new PE on-demand.

Experimental results showed that the latency to migrate one task is lower that in related works, and the proposed protocol achieved a low overhead in the applications execution time: 0.05% for 1 migration, and 0.6% for 2 and 3 migrations while ensuring correct inter-task message synchronization.

The proposed task migration protocol is important towards demystifying the high penalties to adopt task migration in distributed memory systems. This is a significant achievement for self-adaptive techniques, since frequent task migrations may occur, and a low overhead protocol reduces the adaptation overhead.

The proposed dynamic task scheduler ensures a flexible RT time support for soft-RT applications. The scheduler has a hierarchical organization, with a cluster and a local scheduler. The local scheduler controls the CPU usage according to the task constraints, at the PE scope. When the local scheduler cannot meet the constraints, it warns the cluster scheduler, which can decide to fire a task migration, moving the task to a PE with more CPU availability. The main feature and contribution of the scheduler concerns the support to change at run-time the tasks' constraints (period, deadline, execution time).

Experimental results showed that the task scheduler is efficient to support soft RT applications, without deadline miss up to 80% of CPU utilization. Additionally, the scheduler handled dynamic workloads with tasks changing the RT constraints at run-time, and when necessary, migrating the task to another PE with a low overhead due to the low cost of the task migration technique.

The dynamic feature of the scheduler is important for many-cores because it allows a fine-grain QoS tuning at the computation level, enabling a better QoS management due to the avoidance of resource overuse. This is fundamental in the dark silicon era, where software development and execution must be efficient to offset the power and temperature restrictions.

7.2 QoS at the Communication Level

Chapter 5 explored QoS techniques at the communication level, introducing the Software-Defined Networking (SDN) paradigm to the many-core context, and addressed QoS by providing a run-time CS management based on the SDN infrastructure.

The motivation to adopt the SDN paradigm is its flexibility to manage network resources at run-time. The SDN approach transfers the hardware complexity of the network control to a software layer called Network Controller (NoC Controller). The underlying hardware supporting the SDN paradigm is an MPN NoC, with area and power efficient SDN routers.

The NoC Controller handles the requests to define paths, search the paths, and configure the MPN's routers in the paths. The NoC Controller is a generic concept related to the system management based on the SDN paradigm. This Thesis proposed a NoC-Controller called CS-Controller, which finds shortest paths using the Hadlock's algorithm for RT applications during the application admission management phase. The goal is to set CS for all application's flows before the application start its execution. At the end of the application, all connections are released. A CS management protocol ensures the communication between the M_{PE}, CS-Controller, and the S_{PE} running the applications' tasks.

Experimental results showed that the low cost of MPN combined to the routers design based on Elastic Buffers enable to create a rich path diversity for the SDN management, achieving a connection success rate higher than 97% for MPNs with 6 and 8 subnets. Additionally, the SDN paradigm when compared to a state-of-the-art search path mechanism achieved a similar path quality (i.e., average number of hops), with a slight improvement in the number of minimal paths, and higher latency. The higher latency was expected since the comparison was done assuming a software-based approach against a hardware-based approach. The overhead is not an actual drawback since the latency only affects the application startup (in the order of µs). The advantages of adopting SDN include simpler hardware architectures, reusability, and management flexibility, features not available in hardware-centric approaches.

The features provided by the SDN approach enables to create a self-awareness of the communication resources on-chip. Other important feature offered by the SDN-based NoC management it the hardware simplicity, which leads to a high degree of hardware reusability. Each SDN router is a simple unit, with the goal to link a given input port to an output port, according to the definition made in software and stored in a table inside the router.

7.3 QoS at both Computation and Communication Levels

Chapter 6 unified the QoS management, integrating the communication and computation techniques. The proposed management gathers several system and application information at run-time to create a self-awareness of computation and communication resources combined to the QoS fulfillment monitoring of the applications. Additionally, an application profile learning was able to observe the profile of the application on-the-fly. Based on all this information the QoS management

can act reactively, as well as proactively. Reactively, when a given task or CTP loses a deadline or latency constraint. Proactively, by early identifying if a given task has a computation or communication profile, allocating resources to the task before future QoS violations.

Experimental results showed the capability of the proposed self-adaptive QoS management to act reactively and mainly, proactively, avoiding potential QoS violations due to the identification of the correct tasks profile. In a scenario executing several benchmarks, the deadline miss rate stayed below 0.6% even with severe interferences at the computation and communication levels. The application profile learning achieved an accurate observation of the task profile with an insignificant overhead (0.0024% in the application execution time), demonstrating that run-time techniques can be used to characterize applications.

Comprehensiveness and self-adaptation are essential features for the next generation of many-cores due to the high complexity of such designs. Comprehensiveness enables defining trade-offs when multi-objective optimizations are required, such as a power, temperature, and QoS. Self-adaptation is essential to deal with challenges as unpredictably, dynamic workloads, and lack of composability.

7.4 Side Contributions

This Thesis presented two side contributions not related to QoS.

The DMNI enabled to create a direct link between the memory and the NoC router. The processor manages the DMNI, allowing to speed-up the communication between PEs. Experimental results showed that the DMNI speed-up the application execution time up to 12.3% when compared to a traditional PE architecture, with a DMA and NI modules (inherited from bus-based architectures). Such design optimizes the communication performance of NoC-based many-core systems, being important for the proposed QoS techniques.

The proposal of a graphical framework for many-core debugging presented a generic and intuitive way to decrease the validation time of system techniques (as the ones proposed in this Thesis), intuitively and holistically. The DEL layer defines a generic data extraction method combined with a standard database. The graphical windows enable to observe what is occurring during the simulation without looking into long log files or waveforms. A set of windows covering both debugging of computation and communication levels creates an intuitive debugging environment.

7.5 Hypothesis Support

The Introduction of the Thesis stated the following hypothesis:

A comprehensive, scalable, and self-adaptive QoS support can be achieved by proposing reconfigurable QoS techniques covering both computation and communication levels, and by a management that is based on the self-adaptive model [SHA09][DUT16a][DUT16b][ANZ17], the ODA paradigm [HOF13], and the distributed resource management [KRA12][CAS13].

Comprehensiveness was achieved by addressing the communication and computation levels at both the infrastructure scope (task scheduler, task migration, and SDN-based CS) and management scope (the self-adaptive QoS management). Related works so far address specific techniques for communication or computation separately. Acting only in the computation contributes significantly to the system performance. However, due to the number of PEs, the NoC traffic can impact significantly on overall applications' performance, requiring holistic techniques.

Due to the unpredictable profile of general-purpose applications, it is necessary to profile them at run-time and have tools to provide the correct resource allocation according to the application needs. This Thesis advanced the state-of-the-art related to this issue. The proposed self-adaptive QoS management supports the self-adaptation QoS stated in hypothesis, by presenting a run-time resource reconfiguration according to the RT tasks QoS monitoring status (reactively), and its profile (proactively). All these features were deployed according to the ODA paradigm, enabling a better understanding of the adaptation flows, and a better organization of the components.

The QoS techniques and management were distributed in a cluster-based organization, which ensure scalability to the proposal. This organization proved to be scalable, thanks the distribution of the resource management load [CAS13].

Based on the presented techniques throughout this Thesis and supported by the experiments, the Author concludes that the hypothesis was fulfilled.

7.6 Future Works

Future works include the gaps identified in the Thesis' proposals, presented in the next subsections.

7.6.1 Memory Access QoS

The DMNI was designed to support the parallel transmission of data to PS and SDN routers. However, the read and write memory access are atomic operations performed by the DMNI. Assume that two QoS flows are sending data in streaming from memory to SDN routers. The DMNI will provide the same priority for them. A priority-based memory access control over an MPN design is a relevant research that can fulfill this gap. The control can communicate with the task scheduler to know which QoS flow is more urgent, assigning higher priority to it.

7.6.2 A distributed implementation of SDN Controllers

The CS-Controller evaluations target many-core up to 400 PE (20x20 dimension). For larger system sizes, the SDN Controller can be deployed distributed at each cluster. However, if they not communicate among them, the SDN-based management will be performed in individual clusters without a global awareness of the network status. To implement system awareness of the SDN control in higher system dimensions, it is required the research and development of a distributed implementation of the SDN Controller. Such global awareness is challenging, once the protocols to

synchronize the network status among SDNs spreads over the cluster has the potential to cause a high communication overhead or a high latency to answer for a requested service.

7.6.3 Comprehensiveness is the key word

In the Author's opinion, current self-adaptive techniques, focused on only one scope (computation or communication), are mature in literature. The increasing complexity of many-cores will require increasingly degrees of comprehensiveness. This Thesis made one step towards a comprehensive system by integrating self-adaptation for QoS of the computation and communication resources. However, other self-adaptive mechanisms can be integrated. At the computation level, aging-aware task mapping can be assumed; at the communication level, other NoC controllers can be developed, targeting fault tolerance and communication security, for example. Additionally, systemic energy and temperature management are necessary, and they need to interact with the QoS heuristics to reach a good trade-off between applications performance constraints and system budgets.

REFERENCES

[ABB14] Abbas, N.; Ma, Z. "Run-time Parallelization Switching for Resource Optimization on an MPSoC Platform". *Design Automation for Embedded Systems*, vol. 18-3, Sep. 2014, pp. 279-293.

[ABO13] Abousamra, A.; Jones, A. K.; Melhem, R. "Proactive circuit allocation in multiplane NoCs". In: DAC, 2013, pp. 1–10.

[ALH10] Alhonen, A.; Salminen, E.; Nieminen, J.; Hämäläinen, T. D. "A scalable, non-interfering, synthesizable Network-on-chip monitor". In: NORCHIP, 2010, pp. 1-6.

[ALM11] Almeida, G.M.; Busseuil, R.; Carara, E. A.; Hébert, N.; Varyani, S.; Sassatelli, G.; Benoit, P.; Torres, L.; Moraes, F.G. "Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip". In: ISCAS, 2011, pp. 1500-1503.

[AMB13] Ambrose, J. A.; Cassisi, V.; Murphy, D.; Tuo, L.; Jayasinghe, D.; Parameswaran, S. "Scalable performance monitoring of application specific multiprocessor Systems-on-Chip". In: ICIIS, 2013, pp. 315-320.

[ANT15] El-Antably, A.; Gruber, O.; Rousseau, F.; Fournel, N. "Transparent and portable agent-based task migration for data-flow applications on multi-tiled architectures". In: CODES+ISSS, 2015, pp. 183-192.

[ANZ17] Anzanpour, A.; Azimi, I.; Götzinger, M.; Rahmani, A. M.; TaheriNejad, N.; Liljeberg, P.; Jantsch, A.; Dutt, N. "Self-awareness in remote health monitoring systems using wearable electronics". In: DATE, 2017, pp. 1056-1061.

[ARM17] ARM. "big.LITTLE technology". Available at: https://developer.arm.com/technologies/big-little, November 2017.

[ARN14] Arnold, O.; Fettweis, G. "Adaptive run-time management of heterogeneous MPSoCs: Analysis, acceleration and silicon prototype". In: SoC, 2014, 4p.

[ATT11] Attia, B.; Wissem, C.; Noureddine, A.; Zitouni, A.; Torki, K.; Tourki, R. "A new pipelined network interface for Network on Chip with latency and jitter optimization". In: ICM, 2011, pp. 1-6.

[BAM11] Bamakhrama, M.; Stefanov, T. "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications". In: EMSOFT, 2011, pp. 195-204.

[BAM12] Bamakhrama, M.A.; Zhai, J.T.; Nikolov, H.; Stefanov, T. "A methodology for automated design of hard-real-time embedded streaming systems". In: DATE, 2012, pp. 941-946.

[BEN02] Benini, L; Micheli, G. "Networks on chips: a new SoC paradigm". *Computer*, vol. 35-1, Jan. 2002, pp. 70–78.

[BEY13] Beyranvand, A. N.; Molnos, A.; Martinez, M. E.; Goossens, K. "A hardware/software platform for QoS bridging over multi-chip NoC-based systems". *Parallel Computing*, vol. 39-9, Sep. 2013, pp. 424-441.

[BJE05] Bjerregaard, T.; Sparso, J. "A router architecture for connection-oriented service guarantees in the Mango clockless network-on-chip". In: DATE, 2005, pp. 1226-1231.

[BOH17] Bohnenstiehl, B.; Stillmaker, A. J.; Pimentel, J.; Andreas, T.; Liu, B.; Tran, A. T.; Adeagbo, E.; Baas, B. M. "KiloCore: A 32-nm 1000-Processor Computational Array". *IEEE Journal of Solid-State Circuits*, vol. 52-4, Apr. 2017, pp. 891–902.

[BUR10] Burgio, P.; Ruggiero, M.; Esposito, F.; Marinoni, M.; Buttazzo, G.; Benini, L. "Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems". In: ICCD, 2010, pp. 187-194.

[CAN12] Cannella, E.; Derin O.; Meloni, P.; Tuveri, G.; Stefanov, T. "Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks". *VLSI Design*, article nº 2, Jan. 2012, 17p.

[CAR07] Carara, E.; Moraes, F.; Calazans, N. "Router Architecture for High-performance NoCs". In: SBCCI, 2007, pp. 111-116.

[CAR09] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. "HeMPS - a Framework for NoC-based MPSoC Generation". In: ISCAS, 2009, pp. 1345-1348.

[CAR11] Carara, E. "Serviços de Comunicação Diferenciados em Sistemas Multiprocessados em Chip Baseados em Redes Intra-Chip". Tese de Doutorado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2011, 107p.

[CAR14] Carara, E. A.; Calazans, N. L. V.; Moraes, F. G. "Differentiated Communication Services for NoC-Based MPSoCs". *IEEE Transactions on Computers*, vol. 63-3, Mar. 2014, pp. 595-608.

[CAS13] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F. "Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes". In: ISVLSI, 2013, pp. 153-158.

[CHA01] Chandra, A.; Adler, M.; Shenoy, P. J. "Deadline fair scheduling: bridging the theory and practice of proportionate fair scheduling in multiprocessor systems". In: RTAS, 2001, pp. 3-14.

[CHE10] Chen, X.; Lu, Z.; Jantsch, A.; Chen, S. "Supporting Distributed Shared Memory on multi-core Network-on-Chips using a dual microcoded controller". In: DATE, 2010, pp. 34-44.

[CHE16] Chen, Y.; Matus, E.; Fettweis, G. P. "Trellis-search based Dynamic Multi-Path Connection Allocation for TDM-NoCs". In: GLSVLSI, 2016, pp. 323-328.

[CHO11] Chouchene, W.; Attia, B.; Zitouni, A.; Abid, N.; Tourki, R. "A low power network interface for network on chip". In: SSD, 2011, pp.1-6.

[CLE13] CLEARSPEED. "CSX700". Available at: http://www.clearspeed.com/products/csx700.php, December 2013.

[CON14] Cong, L.; Wen, W.; Zhiying, W. "A configurable, programmable and software-defined network on chip". In: WARTIA, 2014, pp. 813–816.

[CUE12] Cueva, P. L.; Bertaux, A.; Termier, A.; Méhaut, J. F.; Santana, M. "Debugging embedded multimedia application traces through periodic pattern mining". In: EMSOFT, 2012, pp. 13-22.

[DAS12] Das, A.; Kumar, A.; Veeravalli, B. "Fault-tolerant network interface for spatial division multiplexing based Network-on-Chip". In: ReCoSoC, 2012, pp. 1-8.

[DAS13] Das, A.; Kumar, A.; Veeravalli, B. "Communication and Migration Energy Aware Design Space Exploration for Multicore Systems with Intermittent Faults". In: DATE, 2013, pp. 1631-1636.

[DAS14] Das, A.; Kumar, A.; Veeravalli, B. "Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs". In: DATE, 2014, pp. 1-6.

[DAV11] Davis, R.I.; Burns, A. "A survey of hard real-time scheduling for multiprocessor systems". *ACM Computing Surveys*, vol. 35-4, Oct. 2011, 44p.

[DER13] Derin, O.; Cannella, E.; Tuveri, G.; Meloni, P.; Stefanov, T.; Fiorin, L.; Raffo, L.; Sami, M. "A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project". *Journal Microprocessors & Microsystems*, vol. 37-6. Aug. 2013, pp. 515-529.

[DUT16a] Dutt, N.; Kurdahi, F. J.; Ernst, R.; Herkersdorf, A. "Conquering MPSoC complexity with principles of a self-aware information processing factory". In: CODES+ISSS, 2016, pp. 1-4.

[DUT16b] Dutt, N.; Jantsch, A.; Sarma, S. "Toward Smart Embedded Systems: A Self-aware System-on-Chip (SoC) Perspective". *ACM Transactions on Embedded Computing Systems*, vol. 15-2, Feb. 2016, 27p.

[DUT17] Dutt, N.; Rahmani, A. M.; Jantsch, A. "Empowering autonomy through self-awareness in MPSoCs". In: NEWCAS, 2017, pp. 73-76.

[EJA13] Ejaz, A.; Jantsch, A. "Costs and benefits of flexibility in spatial division circuit switched networks-on-chip". In: NoCArc, 2013, pp. 41-46.

[FAN10] Fangfa, F.; Xin'na, H.; Jinxiang, W.; Mingyan, Y. "A novel communication strategy between PE and NI in NoC-based MPSoC". In: RCSLPLT, 2010, pp.374-377.

[FAN11] Fan, W.; Xiang, L.; Hui, S. "The QoS mechanism for NoC router by dynamic virtual channel allocation and dual-net infrastructure". In: ICCP, 2011, pp. 1-5.

[FAR10] Faruque, M. Al; Jahn, J.; Ebi, T.; Henkel, J. "Run-time Thermal Management Using Software agents for Multi- and Many-Core Architectures". *IEEE Design & Test*, vol. 27-6, Sep. 2010, pp. 58-68.

[FAR12] Faruque, M. Al; Ebi, T.; Henkel, J. "AdNoC: Run-time Adaptive Network-on-Chip Architecture". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20-2, Feb. 2012, pp. 257-269.

[FAT11] Fattah, M.; Daneshtalab, M.; Liljeberg, P.; Plosila, J. "Exploration of MPSoC Monitoring and Management Systems". In: ReCoSoC, 2011, pp. 1-3.

[FEG07] Feghali, R.; Speranza, F.; Wang, D.; Vincent, A. "Video quality metric for bit rate control via joint adjustment of quantization and frame rate". *IEEE Transactions on Broadcasting*, vol. 53-1, Mar. 2007, pp. 441-446.

[FIO10] Fiorin, L.; Palermo, G.; Silvano, C. "A Monitoring System for NoCs". In: NoCArc, 2010, pp. 25-30.

[FRI14] Friederich, S.; Heisswolf, J; Becker, J. "Hardware/software debugging of large scale many-core architectures". In: SBCCI, 2014, pp. 1-7.

[FU13] Fu, F.; Wang, L.; Lu, Y.; Wang, J. "Low Overhead Task Migration Mechanism in NoC-based MPSoC". In: ASICON, 2013, 4p.

[GAN13a] Gangadharan, D.; Chakraborty, S.; Zimmermann, R. "Quality-aware media scheduling on MPSoC platforms". In: DATE, 2013, pp. 976-981.

[GAN13b] Ganeshpure, K.; Kundu, S. "On run-time task graph extraction in MPSoC". In: ISVLSI, 2013, pp. 171-176.

[GAN14] Gangadharan, D.; Teich, J.; Chakraborty, S. "Quality-aware video decoding on thermally-constrained MPSoC platforms". In: ASAP, 2014, pp. 256-263.

[GEO14] Georgiev, K.; Martin, V. M. "MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach". In: EUC, 2014, pp. 73-80.

[GOR13] Gorski, P.; Timmermann, D. "Centralized traffic monitoring for online-resizable clusters in Networks-on-Chip". In: ReCoSoC, 2013, pp.1-8.

[GOS05] Goossens, K.; Dielissen, J.; Radulescu, A. "Aethereal network on chip: concepts, architectures, and implementations". *IEEE Design & Test of Computers*, vol. 22-5, Sep.-Oct. 2005, pp. 414-421.

[GOT16] Götzinger, M.; Rahmani, A. M.; Pongratz, M.; Liljeberg, P.; Jantsch, A.; Tenhunen, H. "The Role of Self-Awareness and Hierarchical Agents in Resource Management for Many-Core Systems". In: MCSOC, 2016, pp. 53-60.

[GRA13] Grammatikakis, M.D.; Papagrigoriou, A.; Petrakis, P.; Kornaros, G. "Non-intrusive NoC DFS for Soft Real-Time Multimedia Applications". In: DSD, 2013, pp. 60-63.

[GRO12] Grot, B.; Hestness, J.; Keckler, S.W.; Mutlu, O. "A QoS-Enabled On-Die Interconnect Fabric for Kilo-Node Chips". *IEEE Micro,* vol. 32-3, May-Jun. 2012, pp. 17-25.

[GUA10] Guang, L.; Bo Yang; Plosila, J.; Latif, K.; Tenhunen, H. "Hierarchical power monitoring on NoC - a case study for hierarchical agent monitoring design approach". In: NORCHIP, 2010, pp. 1-6.

[GUI13] Guindani, G.; Moraes, F.G. "Achieving QoS in NoC-based MPSoCs through Dynamic Frequency Scaling". In: SoC, 2013, pp. 1-6.

[GUI17] Guikang Chen, X. Li.; Wen, W. "Energy-efficient execution for repetitive app usages on big.LITTLE architectures". In: DAC, 2017, pp. 1-6.

[HAD77] Hadlock, F. O. "A shortest path algorithm for grid graphs". *Networks*, vol. 7-4, Jan. 1977, pp. 323-334.

[HAN11] Hansson, A.; Ekerhult, M.; Molnos A.; Milutinovic, A.; Nelson, A.; Ambrose, J.; Goossens, K. "Design and implementation of an operating system for composable processor sharing". *Microprocessors and Microsystems*, vol. 35-2, Mar. 2011, pp. 246-260.

[HE14] He, H.; Yang, G.; Hu, J. "Algorithms for power-efficient QoS in application specific NoCs". In: ISLPED, 2014, pp. 165-170.

[HED11] Hedde, D.; Petrot, F. "A non-intrusive simulation-based trace system to analyze Multiprocessor Systems-on-Chip software". In: RSP, 2011, pp. 106-112.

[HEI12] Heisswolf, J.; Zaib, A.; Weichslgartner, A.; Konig, R.; Wild, T.; Teich, J.; Herkersdorf, A.; Becker, J. "Hardware-assisted Decentralized Resource Management for Networks on Chip with QoS". In: IPDPSW, 2012, pp. 234-241.

[HEM00] Hemani, A.; Jantsch, A.; Kumar, S.; Postula, A.; Öberg, J.; Millberg, M.; Lindqvist, D. "Network on chip: An architecture for billion transistor era". In: NORCHIP, 2000, 8p.

[HOF13] Hoffman, H. "Seec: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems". Tese de Doutorado, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (MIT), 2013, 172p.

[HWA10] Hwang, M.; Choi, D.; Kim, P. "Least Slack-time Rate First: New Scheduling Algorithm for Multi-Processor Environment". In: CISIS, 2010, pp. 806-811.

[IBM17a] IBM. "IBM Research Alliance Builds New Transistor for 5nm Technology". Available at: https://www-03.ibm.com/press/us/en/pressrelease/52531.wss, September 2017.

[IBM17b] IBM. "TrueNorth's - Brain-inspired Computer". Available at: http://www.research.ibm.com/articles/brain-chip.shtml, November 2017.

[INT13] INTEL. "Teraflops Research Chips". Available at: http://www.intel.com/pressroom/kits/Teraflops/index.htm, December 2013.

[INT16] INTEL. "Intel® Xeon® Processor E5-2699 v4". Available at: http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz?_ga=1.154754040.1577199730.1471540418, August 2016.

[ITR15] International Technology Roadmap for Semiconductors. "ITRS 2011 edition". Available at: http://www.itrs.net/reports.html, April 2015.

[JAH11] Jahn, J.; Faruque, M.; Henkel, J. "CARAT: Context-aware Run-time Adaptive Task Migration for Multi Core Architectures". In: DATE, 2011, pp. 1-6.

[JAN17a] Axel Jantsch's web-site. Available at: http://jantsch.se/AxelJantsch, October 2017.

[JAN17b] Jantsch, A.; Dutt, N.; Rahmani, A. M. "Self-Awareness in Systems on Chip – A Survey". *IEEE Design & Test*, vol. 34-6, Dec. 2017, pp. 8-26.

[JAR14] Jarraya, Y.; Madi, T.; Debbabi, M. "A Survey and a Layered Taxonomy of Software-Defined Networking". *IEEE Communications Surveys & Tutorials*, vol. 16-4, Apr. 2014, pp. 1955-1980.

[JAV14] Javaid, H.; Shafique, M.; Henkel, J.; Parameswaran, S. "Energy-Efficient Adaptive Pipelined MPSoCs for Multimedia Applications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33-5, May 2014, pp. 663-676.

[JEO12] Jeong, M. K.; Erez, M.; Sudanthi, C.; Paver, N. "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC". In: DAC, 2012, pp. 850-855.

[JOH12] Johann, S.; Aguiar, A.; Magalhães, F.; Longhi, O.; Hessel, F. "Task Model Suitable for Dynamic Load Balancing of Real-time Applications in NoC-based MPSoCs". In: ICCD, 2012, pp. 49-54.

[JOV13] Joven, J.; Marongiu, A.; Angiolini, F.; Benini, L.; Micheli, G. "An integrated, programming model-driven framework for NoC-QoS support in cluster-based embedded many-cores". *Parallel Computing*, vol. 39-10, Oct. 2013, pp. 549-566.

[JUN14] Jung, H.; Lee, C.; Kang, S.; Kim, S.; Oh, H.; Ha, S. "Dynamic Behavior Specification and Dynamic Mapping for Real-Time Embedded Systems: HOPES Approach". *ACM Transactions on Embedded Computing Systems*. vol. 13-4, Apr. 2014, 26p.

[KAK11] Kakoee, M.; Bertacco, V.; Benini, L. "ReliNoC: A Reliable Network for Priority-based On-chip Communication". In: DATE, 2011, pp. 1-6.

[KAR10] Kariniemi, H.; Nurmi, J. "High-Performance NoC Interface with Interrupt Batching for Micronmesh MPSoC Prototype Platform on FPGA". In: NORCHIP, 2010, pp. 1-6.

[KIM10] Kim, Y.; Papamichael, M.; Mutlu, O.; Harchol-Balter, M. "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior". In: MICRO, 2010, pp. 65-76.

[KIR13] Kirkpatrick, K. "Software-defined networking". *Communications of the ACM*, vol. 56-9, Sep. 2013, pp 16-19.

[KOR12] Kornaros, G.; Pnevmatikatos, D. "Real-Time Monitoring of Multicore SoCs Through Specialized Hardware Agents on NoC Network Interfaces". In: IPDPSW, 2012, pp. 248-255.

[KOR13] Kornaros, G.; Pnevmatikatos, D. "A survey and taxonomy of on-chip monitoring of multicore systems-on-chip". *ACM Transactions on Design Automation Electronic Systems*, vol. 18-2, Apr. 2013, 38p.

[KOT10] Kooti, H.; Bozorgzadeh, E. "Unified theory of real-time task scheduling and dynamic voltage/frequency Scaling on MPSoCs". In: ICCAD, 2010, pp. 139-142.

[KRA12] Kramer, D.; Karl, W. "A Scalable Monitoring Infrastructure for Self-Organizing Many-Core Architectures". In: DSD, 2012, pp. 42-49.

[LAP11] Laplante, P. A.; Ovaska, S. J. "Real-time systems Design and Analysis". Wiley-IEEE Press, 4th edition, 2011, 584p.

[LER05] Leroy, A.; Marchal, P.; Shickova, A.; Catthoor, F.; Robert, F.; Verkest, D. "Spatial Division Multiplexing: a Novel Approach for Guaranteed Throughput on NoCs". In: CODES+ISSS, 2005, pp. 81-86.

[LER08] Leroy, A.; Milojevic, D.; Verkest, D.; Robert, F.; Catthoor, F. "Concepts and Implementation of Spatial Division Multiplexing for Guaranteed Throughput in Networks-on-Chip". *IEEE Transactions on Computers*, vol. 57-9, Sep. 2008, pp. 1182-1195.

[LI11] Li, S.; Chen, K.; Ahn, J. H.; Brockman, J. B.; Jouppi, N. P. "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques". In: ICCAD, 2011, pp. 694-701.

[LI13] Li, X.; Jia, X.; Ju, L. "Slack-Time-Aware Energy Efficient Scheduling for Multiprocessor SoCs". In: HPCC_EUC, 2013, pp. 278-285.

[LIN10] Lin, Y.; Yang, C.; Lin, T.; Huang, J.; Chang, N. "Hierarchical memory scheduling for multimedia MPSoCs". In: ICCAD, 2010, pp. 190-196.

[LIU00] Liu, J.W.S. "Real-Time System". Prentice Hall, 1st edition, 2000, 456p.

[LIU12] Liu, S.; Jantsch, A.; Lu, Z. "Parallel probing: Dynamic and constant time setup procedure in circuit switching NoC". In: DATE, 2012, pp. 1289-1294.

[LIU13] Liu, S.; Su, M.; Wu, R.; Li, L. "Exclusive Memory Scheduling for Multimedia MPSoC". In: HPCC_EUC, 2013, pp. 2022-2026.

[LIU14a] Liu, W.; Wang, X; Xu, J.; Zhang, W.; Ye, Y.; Wu, X.; Nikdast, M.; Wang, Z. "On-chip sensor networks for soft- r tolerant real-time multiprocessor systems-on-chip". *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10-2, Mar. 2014, 20p.

[LIU14b] Liu, S.; Jantsch A.; Lu Z. "Parallel probe based dynamic connection setup in TDM NoCs". In: DATE, 2014, pp. 1-6.

[LIU15] Liu, S.; Jantsch, A; Lu, Z. "MultiCS: Circuit switched NoC with multiple sub-networks and sub-channels". *Journal of Systems Architecture*, vol. 61-9, Oct. 2015, pp. 423-434.

[LIU16] Liu, S.; Lu, Z.; Jantsch, A. "Highway in TDM NoCs. In Proceedings of the International Symposium on Networks on Chip". In: NOCS, 2015, 8p.

[LOU17] Loubet, N. et al. "Stacked nanosheet gate-all-around transistor to enable scaling beyond FinFET". In: VLSIT, 2017, pp. 230-31.

[LUS11] Lusala, A.; Legat, J-D. "Combining SDM-BASED Circuit Switching with Packet Switching in a NoC for Real-Time Application". In: ISCAS, 2011, pp. 2505-2508.

[LUS12] Lusala, A.; Legat, J-D. "A hybrid NoC combining SDM-TDM based circuit-switching with packet-switching for real-time applications". In: NEWCAS, 2012, pp. 17-20.

[MA15] Ma, R.; Hui, Z.; Jantsch, A. "A packet-switched interconnect for many-core systems with BE and RT service". In: DATE, 2015, pp. 980-983.

[MAD16] Madalozzo, G.; Duenha, L; Azevedo, R.; Moraes, F. G. "Scalability evaluation in many-core systems due to the memory organization" In: ICECS, 2016, pp. 396-399.

[MAN10] Mangano, D.; Strano, G. "Enabling Dynamic and Programmable QoS in SoCs". In: NoCArc, 2010, pp. 17–22.

[MAR11] Marwedel, P.; Bacivarov, I.; Lee, C.; Teich, J.; Thiele, L.; Xu, Q.; Kouveli, G; Ha, S.; Huang, L. "Mapping of applications to MPSoCs". In: CODES+ISSS, 2011, pp. 109-118.

[MAR17a] Martins, A.; Ruaro, M.; Santana, A.; Moraes, F. G. "Distributed Run-time Energy Management for Many-Core Systems Running Real-Time Applications". *Journal of Low Power Electronics*, v. 13-3, Sep. 2017, pp. 402-418.

[MAR17b] Martins, A.; Ruaro, M.; Santana, A.; Moraes, F. G. "Run-time Energy Management Under Real-Time Constraints in MPSoCs". In: ISCAS, 2017, pp. 2589-2592.

[MAT10a] Matos, D.; Concatto, C.; Kologeski, A.; Carro, L.; Kastensmidt, F.; Susin, A.; Kreutz, M. "Monitor-Adapter Coupling for NoC Performance Tuning". In: SAMOS, 2010, pp. 193-199.

[MAT10b] Matos, D.; Carro, L.; Susin, A. "Associating packets of heterogeneous cores using a synchronizer wrapper for NoCs". In: ISCAS, 2010, pp. 4177-4180.

[MAT14] Matos, D.; Kreutz, M.; Reinbrecht, C.; Carro, L.; Susin, A. "Adaptive multiple switching strategy toward an ideal NoC". In: ISCAS, 2014, pp. 1014-1017.

[MEL05] Mello, A.; Tedesco, L.; Calazans, N.; Moraes, F. G. "Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC". In: SBCCI, 2005, pp. 178-183.

[MIC11] Michelogiannakis, G.; Becker, D.U.; Dally, W.J. "Evaluating Elastic Buffer and Wormhole Flow Control". *IEEE Transactions on Computers*, vol. 60-6, Jun. 2011, pp. 896-903.

[MIC13] Michelogiannakis, G.; Dally, W. "Elastic Buffer Flow Control for On-Chip Networks". *IEEE Transactions on Computers*, vol. 62-2, Feb. 2013, pp. 295-309.

[MIC17] Micheli, G. De; Benini, L. "Networks on Chips: 15 Years Later". *Computer*, vol. 50-5, May 2017, pp. 10-11.

[MÖL10] Möller, L.; Jesus, H.; Moraes, F. G.; Indrusiak, L. S.; Hollstein, T.; Glesner, M. "Graphical interface for debugging RTL Networks-on-Chip". In: BEC, 2010, pp. 181-184.

[MOL10] Molnos, A.; Ambrose, J.A.; Nelson, A.; Stefan, R.; Cotofana, S.; Goossens, K. "A composable, energy-managed, real-time MPSOC platform". In: OPTIM, 2010, pp. 870-876.

[MOL12] Molnos, A.; Nejad, A. B.; Nguyen, B. T.; Cotofana, S.; Goossens, K. "Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms". In: SCOPES, 2012, pp. 13-21.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". *Integration, the VLSI Journal*, vol. 38-1, 2004, pp. 69-93.

[MOT11] Motakis, A.; Kornaros, G.; Coppola, Marcello. "Dynamic Resource Management in Modern Multicore SoCs by Exposing NoC Services". In: ReCoSoC, 2011, pp. 1-7.

[MOT12] Motruk, B.; Diemer, J.; Buchty, R.; Ernst, R.; Berekovic, M. "IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality". In: HASE, 2012, pp. 24-31.

[MUN15] Munk, P.; Saballus, B.; Richling, J.; Heiss, H. "Position Paper: Real-Time Task Migration on Many-Core Processors". In: ARCS, 2015, 4p.

[MUR14] Murillo, L. G.; Wawroschek, S.; Castrillon, J.; Leupers, R.; Ascheid, G. "Automatic detection of concurrency bugs through event ordering constraints". In: DATE, 2014, 6p.

[NEI12] Neishaburi, M. H.; Zilic, Z. "An enhanced debug-aware network interface for Network-on-Chip". In: ISQED, 2012, pp. 709-716.

[NOW13] Nowotsch, J.; Paulitsch, M. "Quality of service capabilities for hard real-time applications on multi-core processors". In: RTNS, 2013, pp. 151-160.

[OLI11] Oliva, Y.; Pelcat, M.; Nezan, J.-F.; Prevotet, J.-C.; Aridhi, S. "Building a RTOS for MPSoC dataflow programming". In: SoC, 2011, pp. 143-146.

[OUY10] Ouyang, j.; Xie, Y. "LOFT: A High-Performance Network-on-Chip Providing Quality-of-Service Support". In: MICRO, 2010, pp. 409-420.

[PAL12] Palumbo, F.; Pani, D.; Congiu, A.; Raffo, L. "Concurrent hybrid switching for massively parallel systems-on-chip: the CYBER architecture". In: CF, 2012, pp. 173-182.

[PAR14] Park, S. "Task-I/O Co-scheduling for Pfair Real-Time Scheduler in Embedded Multi-core Systems". In: EUC, 2014, pp. 46-51.

[PAS08] Pasricha, S.; Dutt, N. "On-Chip Communication Architectures System on Chip Interconnect". Elsevier, 2008, 522p.

[PAT12] Paterna, F.; Acquaviva, A.; Caprara, A.; Papariello, F.; Desoli, G.; Benini, L. "Variability-Aware Task Allocation for Energy-Efficient Quality of Service Provisioning in Embedded Streaming Multimedia Applications". *IEEE Transactions on Computers*, vol. 61-7, Jul. 2012, pp. 939-953.

[PRA11] Prada-Rojas, C.; Marangozova-Martin, V.; Méhaut, J; Santana, M. "A Generic Component-Based Approach to MPSoC Observation". In: EUC, 2011, pp. 261-267.

[QUA16] Quan, W.; Pimentel, A. D. "A Hierarchical Run-time Adaptive Resource Allocation Framework for Large-scale MPSoC Systems". *Design Automation for Embedded Systems*, vol. 20-4, Dec. 2016, pp. 311–339.

[RAN15] Ranieri, J.; Vincenzi, A.; Chebira, A.; Atienza, D.; Vetterli, M., "Near-optimal thermal monitoring framework for many-core systems on chip". *IEEE Transactions on Computer*, vol. 64-11, Nov. 2015, pp. 3197-3209.

[ROS14] Rosvall, K.; Sander, I. "A constraint-based design space exploration framework for real-time applications on MPSoCs". In: DATE, 2014, pp. 1-6.

[RUA14a] Ruaro, M.; Carara, E. A.; Moraes, F. G. "Run-time QoS Support for MPSOC: A Processor Centric Approach". In: SBCCI, 2014, pp. 1-7.

[RUA14b] Ruaro, M.; Carara, E. A. Moraes, F. G. "Tool-Set for NoC-Based MPSoC Debugging - a Protocol View Perspective". In: ISCAS, 2014, pp. 2531-2534.

[RUA15a] Ruaro, M.; Carara, E. A.; Moraes, F. G. "Run-time Adaptive Circuit Switching and Flow Priority in NoC-Based MPSoCs". *IEEE Transactions on VLSI*, vol. 23-6, Jun. 2015, pp. 1077-1088.

[RUA15b] Ruaro, M.; Madalozzo, G.; Moraes, F. G. "A Hierarchical LST-Based Task Scheduler for NoC-Based MPSoCs with Slack-Time Monitoring Support". In: ICECS, 2015, pp. 308-311.

[RUA16a] Ruaro, M.; Moraes, F. G. "Dynamic Real-Time Scheduler for Large-Scale MPSoCs". In: GLSVLSI, 2016, pp. 341-346.

[RUA16b] Ruaro, M.; Lazzarotto, F.; Marcon, C.; Moraes, F. G. "DMNI: A Specialized Network Interface for NoC-based MPSoCs". In: ISCAS, 2016, pp. 1202-1205.

[RUA16c] Ruaro, M.; Rubin, F.; Chamorra, H.; Amory, A. M.; Moraes, F. G. "A Data Extraction and Debugging Framework for Large-Scale MPSoCs". In: ICECS, 2016, pp. 616-619.

[RUA17a] Ruaro, M.; Medina, H. M.; Moraes, F. G. "SDN-Based Circuit-Switching for Many-Cores". In: ISVLSI, 2017, pp. 385-390.

[RUA17b] Ruaro, M.; Moraes, F. G. "Demystifying the Cost of Task Migration in Distributed Memory Many-Core Systems". In: ISCAS, 2017, 4p.

[RUA18] Ruaro, M.; Medina, H.; Moraes, F. G. "Software-Defined Networking Architecture for NoC-based Many-Cores". In: ISCAS, 2018, 4p.

[SAI08] Saint-Jean, N.; Benoit, P.; Sassatelli, G.; Torres, L.; Robert, M. "MPI-Based Adaptive Task Migration Support on the HS-Scale System". In: ISVLSI, 2008, pp. 105-110.

[SAI10] Said, M. B.; Loukil, K.; Ben Amor, N.; Abid, M.; Diguet, J.P. "A timing constraints Control technique for embedded real time systems". In: DTIS, 2010, 6p.

[SAL11] Salah, Y.; Tourki, R. "Design and FPGA Implementation of a QoS Router for NoC". In: NGNS, 2011, pp. 84-89.

[SAL14] Salami, B.; Baharani, M.; Noori, H. "Proactive Task Migration with a Self-Adjusting Migration Threshold for Dynamic Thermal Management of Multi-Core Processors". *The Journal of Supercomputing*, vol. 68-3, Mar. 2014, pp. 1068-1087.

[SAM14] Samman, F. A. "Run-time connection-oriented guaranteed-bandwidth network-on-chip with extra multicast communication service". *Microprocessors and Microsystems*, vol. 38-2, Mar. 2014, pp. 170-181.

[SAN15] Sandoval-Arechiga, R.; Vazquez-Avila, J. L.; Parra-Michel, R.; Flores-Troncoso, J.; Ibarra-Delgado, S. "Shifting the Network-on-Chip Paradigm towards a Software Defined Network Architecture". In: CSCI, 2015, pp. 869-870.

[SAN16] Sandoval-Arechiga, R.; Parra-Michel, R.; Vazquez-Avila, J. L.; Flores-Troncoso, J.; Ibarra-Delgado, S. "Software Defined Networks-on-Chip for multi/many-core systems: A performance evaluation". In: ANCS, 2016, pp. 129-130.

[SAP14] Saponara, S.; Bacchillone, T.; Petri, E.; Fanucci, L.; Locatelli, R.; Coppola, M. "Design of a NoC Interface Macrocell with Hardware Support of Advanced Networking Functionalities". *IEEE Transactions on Computers*, vol. 63-3, Mar. 2014, pp. 609-621.

[SAR15] Sarma, S.; Dutt, N.; Gupta, P.; Nicolau, A.; Venkatasubramanian, N. "Cyberphysical-system-on-chip (CPSOC): A self-aware MPSOC paradigm with cross-layer virtual sensing and actuation". In: DATE, 2015, pp. 625-628.

[SCI16] Scionti, A.; Mazumdar, S.; Portero, A. "Software defined Network-on-Chip for scalable CMPs". In: HPCS, 2016, pp. 112-115.

[SEI14] Seitanidis, I.; Psarras, A.; Dimitrakopoulos, G.; Nicopoulos, C. "ElastiStore: An elastic buffer architecture for Network-on-Chip routers". In: DATE, 2014, 6p.

[SHA09] Salehie, M.; Tahvildari, L. "Self-adaptive software: Landscape and research challenges". *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4-2, May 2009, 42p.

[SHA12] Shah, H.; Raabe, A.; Knoll, A. "Bounding WCET of applications using SDRAM with Priority Based Budget Scheduling in MPSoCs". In: DATE, 2012, pp. 665-670.

[SHA14] Shafique, M.; Garg, S.; Henkel, J.; Marculescu, D. "The EDA challenges in the dark silicon era". In: DAC, 2014, 6p.

[SHA17] Shafique, M.; Garg, S. "Computing in the Dark Silicon Era: Current Trends and Research Challenges" *IEEE Design & Test*, vol. 34-2, Apr. 2017, pp. 8–23.

[SHE05] Sherwani, N. "Algorithms for VLSI Physical Design Automation". Springer, 2005, 3rd edition, 572p.

[SIN13a] Singh, A.; Das, A. Kumar, A. "Energy optimization by exploiting execution slacks in streaming applications on Multiprocessor Systems". In: DAC, 2013, pp. 1-7.

[SIN13b] Singh, A.K.; Shafique, M.; Kumar, A.; Henkel, J. "Mapping on multi/many-core systems: Survey of current and emerging trends". In: DAC, 2013, pp. 1-10.

[STA11] Stan, A.; Valachi, A.; Barleanu, A. "The design of a run-time monitoring structure for aMPSoC". In: ICSTCC, 2011, pp. 1-4.

[STE12a] Stefan, R.; Molnos, A.; Ambrose, A.; Goossens, K. "A TDM NoC supporting QoS, multicast, and fast connection set-up". In: DATE, 2012, pp. 1283-1288.

[STE12b] Stefan, R.; Nejad, A.; Goossens, K. Online allocation for contention-free-routing NoCs. In: INA-OCMC, 2012, pp. 13-16.

[SWA12] Swaminathan, K.; Lakshminarayanan, G.; Seok-Bum Ko. "High Speed Generic Network Interface for Network on Chip Using Ping Pong Buffers". In: ISED, 2012, pp. 72-76.

[TAF11] Tafesse, B.; Raina, A.; Suseela, J.; Muthukumar, V. "Efficient Scheduling Algorithms for MpSoC Systems". In: ITNG, 2011, pp. 683-688.

[THE13] Theodoropoulos,D.; Pratikakis,P.; Pnevmatikatos,D. "Efficient run-time support for embedded MPSoCs". In: SAMOS, 2013, pp. 164-171.

[TIL13] Tilera Corporation. "Tile-GX Processor Family". Available at: http://www.tilera.com/products/processors/TILE-Gx_Family, November 2013.

[TUV13] Tuveri, G.; Secchi, S.; Meloni, P.; Raffo, L.; Cannella, E. "A run-time adaptive H.264 video-decoding MPSoC platform". In: DASIP, 2013, pp. 149-156.

[WAN10] Wang, Y.; Wang, Yu; Xu, J.; Yang, H. "Performance Evaluation of On-Chip Sensor Network (SENoC) in MPSoC". In: ICGCS, 2010, pp. 323-327.

[WAN12] Wang, C.; Bagherzadeh, N. "Design and Evaluation of a High Throughput QoS-Aware and Congestion-Aware Router Architecture for Network-on-Chip". In: Euromicro, 2012, pp. 457-464.

[WAN14] Wang, H.; Singh, R.; Schulte, M. J.; Kim, N. S. "Memory scheduling towards high-throughput cooperative heterogeneous computing". In: PACT, 2014, pp. 331-342.

[WEN12] Wen, C. N.; Chou, S. H.; Chen, C. C.; Chen, T. F. "NUDA: A Non-Uniform Debugging Architecture and Nonintrusive Race Detection for Many-Core Systems". *IEEE Transactions on Computers*, vol. 61-2, Feb. 2012, pp. 199-212.

[WIN11] Winter, M. and Fettweis, G. "Guaranteed Service Virtual Channel Allocation in NoCs for Run-Time Task Scheduling". In: DATE, 2011, pp. 1-6.

[WIS11] Wissem, C.; Attia, B.; Noureddine, A.; Zitouni, A.; Tourki, R. "A Quality of Service Network on Chip based on a New Priority Arbitration Mechanism". In: ICM, 2011, pp. 1-6.

[WOL08] Wolf, W.; Jerraya, A.; Martin, G. "Multiprocessor SystemFonFChip (MPSoC) Technology". *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 27-10, Sep. 2008, pp. 1701-1713.

[YAO14] Yao Z.; Sui, X.; Xu, T.; Ma, J.; Fang, J.; Mckee, S.A.; Fu, B.; Bao, Y. "QBLESS: A case for QoS-aware bufferless NoCs". In: IWQoS, 2014, pp. 93-98.

[YOO13] Yoon, J. Y.; Concer, N.; Petracca, M.; Carloni, L.P. "Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance". *IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems*, vol. 32-12, Dec. 2013, pp. 1906-1919.

[YU09] Yu, Y.; Ren, S.; Hu, X.S. "A Metric for Judicious Relaxation of Timing Constraints in Soft Real-Time Systems". In: RTAS, 2009, pp. 163-172.

[YU14] Yu; H.; Syed, R.; Ha, Y. "Thermal-aware frequency scaling for adaptive workloads on heterogeneous MPSoCs". In: DATE, 2014, pp. 1-6.

[YUN13] Yun, B.; Shin, K. G.; Wang, S. "Thermal-Aware Scheduling of Critical Applications Using Job Migration and Power-Gating on Multi-core Chips". In: TRUSTCOM, 2011, pp. 1083-1090.

[ZHA11] Zhao, J.; Madduri, S.; Vadlamani, R.; Burleson, W.; Tessier, R. "A Dedicated Monitoring Infrastructure for Multicore Processors", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19-6, Jun. 2011, pp. 1011-1022.

[ZHA14] Zhang, N.; Gu, H.; Yang, Y.; Fan, D. "QBNoC: QoS-aware bufferless NoC architecture". *Microelectronics Journal*, vol. 45-6, Jun. 2014, pp. 751-758.

APPENDIX A – PUBLICATIONS OF THE AUTHOR

Table 20 presents the set of publications (sorted by date) held since the beginning of the PhD. The description column links the paper to this Thesis section, when applicable, or to the main theme of the publication.

Table 20 – Author's publications.

	Publication	Description
1	**Tool-Set for NoC-Based MPSoC Debugging - a Protocol View Perspective** Ruaro, Marcelo; Carara, Everton Alceu; Moraes, Fernando Gehm. In: ISCAS, 2014.	Section 3.5
2	**Run-time QoS Support for MPSoC: A Processor Centric Approach** Ruaro, Marcelo; Carara, Everton Alceu; Moraes, Fernando Gehm. In: SBCCI, 2014.	Section 1.1
3	**A Hierarchical LST-Based Task Scheduler for NoC-Based MPSoCs with Slack-Time Monitoring Support** Ruaro, Marcelo; Madalozzo, Guilherme; Moraes, Fernando Gehm. In: ICECS, 2015.	Section 4.3
4	**Run-time Adaptive Circuit Switching and Flow Priority in NoC-Based MPSoCs** Ruaro, Marcelo; Carara, Everton Alceu; Moraes, Fernando Gehm. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v. 23, p. 1077-1088, 2015.	Section 1.1
5	**Hierarchical Energy Monitoring for Many-Core Systems** Martins, André; Ruaro, Marcelo; Moraes, Fernando Gehm. In: ICECS, 2015.	Section 4.4
6	**A Data Extraction and Debugging Framework for Large-Scale MPSoCs** Ruaro, Marcelo; Rubin, Felipe; Chamorra, Henrique; Amory, Alexandre de Morais Moraes, Fernando Gehm. In: ICECS, 2016.	Section 3.5
7	**Dynamic Real-Time Scheduler for Large-Scale MPSoCs** Ruaro, Marcelo; Moraes, Fernando Gehm. In: GLSVLSI, 2016.	Section 4.3
8	**DMNI: A Specialized Network Interface for NoC-based MPSoCs** Ruaro, Marcelo; Lazzarotto, Felipe; Marcon, César; Moraes, Fernando Gehm. In: ISCAS, 2016.	Section 3.4
9	**System Management Recovery Protocol for MPSoCs** Fochi, Vinicius; Caimi, Luciano; Ruaro, Marcelo; Wachter, Eduardo; Moraes, Fernando Gehm. In: SOCC, 2017.	High-level protocol to support fault tolerance at the system management level.
10	**Demystifying the Cost of Task Migration in Distributed Memory Many-Core Systems** Ruaro, Marcelo; Moraes, Fernando Gehm. In: ISCAS, 2017.	Section 4.2
11	**SDN-Based Circuit-Switching for Many-Cores.** Ruaro, Marcelo; Medina, Henrique; Moraes, Fernando Gehm. In: ISVLSI, 2017.	Section 5.3
12	**Distributed Run-time Energy Management for Many-Core Systems Running Real-Time Applications.** Martins, André; Ruaro, Marcelo; Santana, Anderson; Moraes, Fernando Gehm Journal of Low Power Electronics, v.13, Sep. 2017, p. 402-418.	Section 4.4
13	**Software-Defined Networking Architecture for NoC-based Many-Cores** Ruaro, Marcelo; Medina, Henrique; Moraes, Fernando Gehm. In: ISCAS, 2018, early accepted.	Section 5.2
14	**Self-Adaptive QoS Management at Communication and Computation Levels for Many-Core SoCs** Ruaro, Marcelo; Jantsch, Axel; Moraes, Fernando Gehm. Submitted to ACM Transactions on Embedded Computing Systems (TECS), 2018	Chapter 6

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br