PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

ANDERSON ROBERTO PINHEIRO DOMINGUES

ORCA RT-TOOLS: A SUITE OF TOOLS TO SUPPORT REAL-TIME ANALYSIS ON MANYCORE SYSTEMS

Porto Alegre
2024

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ORCA RT-TOOLS: A SUITE OF TOOLS TO SUPPORT REAL-TIME ANALYSIS ON MANYCORE SYSTEMS

ANDERSON ROBERTO PINHEIRO DOMINGUES

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2024

Ficha Catalográfica

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

ANDERSON ROBERTO PINHEIRO DOMINGUES

ORCA RT-TOOLS: A SUITE OF TOOLS TO SUPPORT REAL-TIME ANALYSIS ON MANYCORE SYSTEMS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on 26th March, 2024.

COMMITTEE MEMBERS:

Jarbas Aryel Nunes da Silveira (Pós-Graduação em Eng. de Teleinformática/UFC)

Rafael Fraga Garibotti (Vector Trading)

Prof. César Augusto Missio Marcon (PPGCC/PUCRS)

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

ORCA RT-TOOLS: UMA SUITE DE FERRAMENTAS PARA AUXILIAR A ANÁLISE DE TEMPO-REAL EM SISTEMAS MANYCORE

RESUMO

A pandemia de COVID-19 permitiu que muitas empresas entrassem no mercado oferecendo soluções para auxiliar no combate contra o vírus SARS-CoV-2, incluindo robôs para desinfecção por UV-C e veículos autônomos. Com o crescimento na demanda de recursos computacionais nestes sistemas, a comunidade passou a explorar recursos de plataformas manycore, que possuem um maior potencial de paralelismo quando comparadas às plataformas single- e multi-core utilizadas atualmente. Entretanto, sistemas da missão-crítica necessitam de suporte para vários requisitos não-funcionais para comunicação e computação, incluindo tempo-real (TR). Ao longo dos anos, garantias de TR para comunicação e computação foram abordadas por diferentes comunidades de pesquisa. De um lado, a pesquisa sobre computação de TR avançou significantemente para plataformas single- e multi-core. Do outro lado, a pesquisa em comunicação de TR originou as chamadas redes intra-chip de tempo-real (RT-NoCs). Como resultado, a literatura carece de uma abordagem que trata ambas as perpectivas destes sistemas, majoritariamente negligenciando uma análise combinada de computação e comunicação. Esta tese apresenta uma abordagem para a análise de tempo-real em systemas manycore baseados em redes intra-chip, tratando tanto computação quanto comunicação. Nós usamos nossa abordagem para garantir os requisitos de tempo-real para computação e comunicação de um sistema sem uma NoC de tempo-real, usando uma NoC *open-hardware* de baixo custo. Nós validamos nossa abordagem em ambiente de simulação RTL, usando tanto aplicações sintéticas quanto benchmarks.

Palavras-Chave: Redes intrachip, Sistemas de Tempo-Real, Manycores.

ORCA RT-TOOLS: A SUITE OF TOOLS TO SUPPORT REAL-TIME ANALYSIS ON MANYCORE SYSTEMS

ABSTRACT

The COVID-19 pandemic allowed plenty of companies to enter the market offering solutions to help fight the SARS-CoV-2 virus, including robots for UV-C disinfection and autonomous vehicles. With the demand for computational resources growing in the critical-mission systems domain, the community began exploring resources from manycore platforms, which have a potential for parallelism compared to the single- and multi-core platforms currently used. However, critical-mission systems need support for several non-functional requirements for communication and computation, including real-time (RT). Over the years, different research communities have addressed RT guarantees for communication and computation. On the one hand, research on RT computing has advanced significantly for single- and multi-core platforms. On the other hand, RT communication research developed the so-called real-time intra-chip networks (RT-NoCs). As a result, the literature lacks an approach that tackles both perspectives of a system, mostly neglecting the joint analysis of computing and communication operations synchronization. This Thesis presents an approach to RT analysis in NoC-based manycores, which tackles computation and communication jointly. We used our approach to guarantee the RT requirements for computation and communication on a system without an RT-NoC, using a low-cost, open-source NoC. We validate our approach for synthetic and benchmark applications in an RTL simulation environment.

Keywords: Network-on-chip, real-time systems, manycores.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

A-TDM – Asynchronous TDM
AD – Absolute Deadline
AGS – Adaptive Guided Search
AI – Artificial Intelligence
API – Application Programming Interface
ASIC – Application-Specific Integrated Circuit
BE – Best Effort
BFS – Breadth-First Search
CFAR – Constant False-Alarm Rate
CPA – Critical Path Analysis
CP – Capacity
CPU – Central Processing Unit
CS – Circuit Switching
DAG – Directed Acyclic Graph
DCT – Discrete Cosine Transform
DDMA – Dual-Model DMA
DES – Discrete-Event Simulation
DFS – Depth-First Search
DM – Deadline Monotonic
DMA – Direct Memory Access
DNN – Deep Neural Networks
DRA – Dynamic Resource Allocation
DTA – Design-Time Configuration
DTG – Design-Time Generation
DVFS – Dynamic Voltage-Frequency Scaling
EDF – Earliest Deadline First
ET – Execution Time
FPGA – Field Programmable Gate Array
FNS – Fast Network Scheduler
GAPH – Grupo de Apoio ao Projeto de Hardware
GCC – GNU's C Compiler
GCM – Graph Collapse Method

GCPA – Graph Critical Path Analyser
GDE – Graph Data Extractor
GSE – Grupo de Sistemas Embarcados
GUI – Graphical User Interface
GS – Guaranteed Service
HAL – Hardware Abstraction Layer
HRT – Hard Real-Time
I/O – Input and Output
ILP – Integer Linear Programming
ISA – Instruction Set Architecture
ISO – International Organization for Standardization
LET – Logical Execution Time
LLF – Least Laxity First
LSTF – Least Slack Time First
MBCF – Most Bandwidth Consuming First
MCPF – Most "Critical Path" First
MRT – Minimum Release Time
ORT – ORCA RT TOOLS
OS – Operating System
PCD – Period, Capacity, and Deadline
PE – Processing Element
PF – Pruning Factor
PIC – Programmable Interruption Controller
PO – Partial Outcome
PRT – Pre-Runtime
PS – Packet Switching
PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul
QoS – Quality-of-Service
NoC – Network on Chip
NS – Network Scheduler
RA – Reference Architecture
RAT – Resource Allocation Technique
RAM – Random Access Memory
RGG – Random Graph Generator

RM – Rate Monotonic
ROM – Read-Only Memory
RR – Round-Robin
RT – Real-Time
RTL – Register-Transfer Level
RUN – Runtime
S-TDM – Synchronous TDM
SA – Switch Allocation
SE – Standard Edition
SIARR – Selective Iteration-Aware Round Robin
SDM – Spatial-Division Multiplexing
SRA – Static Resource Allocation
ST – Switch Traversal
TCNI – Time-Controlled Network Interface
TDM – Time-Division Multiplexing
UAV – Unmanned aerial vehicle
UCX – Microcontroller Executive
UV-C – Ultra Violet class C
UVD – Ultra Violet Disinfection
VC – Virtual Channel
WCET – Worst Case Execution Time
WCTT – Worst-Case Transmission Time
WRR – Weighted Round Robin

CONTENTS

1. INTRODUCTION

Critical-mission applications struggle with legacy single- and multi-core platforms [Chakaravarthy et al., 2021] due to their limited scalability as the demand for computation and communication performance increases, propelled by applications such as delivering drones [Chen, 2023], autonomous cars [Cellan-Jones, 2020], and service robots [UVD Robots, 2024]. As an alternative to single- and multi-core platforms, the manycore technology gathered attention from the industry in the last decades due to its potential for parallelism, reduced area, low energy consumption, attractive fabrication cost, and scalability [Wolf et al., 2008]. However, adopting manycores imposes challenges, mainly due to their inherently distributed programming model [Nürnberger et al., 2014] and the lack of commercial off-the-shelf platforms. Besides, strict non-functional requirements of applications such as real-time (RT) require native support from the target platform, often requiring custom hardware solutions, e.g., field programmable gate array (FPGA), to supply applications needs.

In this Thesis, we present approaches for preventing violations of hard real-time constraints in applications running on network-on-chip-based (NoC) manycores, approaching both computation and communication. We dedicate the remainder of this chapter to present motivation (Section 1.1), research context (Section 1.2), thesis statement (Section 1.3), goals (Section 1.4), scope (Section 1.5), our contributions (Section 1.6), and organization of the remainder of the Thesis (Section 1.7).

1.1 Motivation

Since late 2019, the world has faced a Covid-19 pandemic, whose worldwide victims are countless. In response to the pandemic, the government and health organizations implemented new protocols to mitigate the SARS-CoV-2 contamination. The adopted strategies range from social distancing to heavy disinfection procedures, including using robots to replace humans as an alternative to avoid potential contamination. Even with the global economy bending to the effects of the pandemic, many corporations have seized opportunities within the robotics and embedded system markets, providing products such as delivering drones, disinfection robots, and personal assistant robots, augmenting the fight against the pandemic [Seidita et al., 2021].

The demand for critical-mission applications soared since the beginning of the pandemic [Berreby, 2020], whose dates nearly match with Amazon's announcements for new efforts in drone delivering [TechCrunch, 2019]. For instance, the Bishan-Ang Moh Kio Park [BBC News, 2020a] have tested in the field Spot [Boston Dynamics, 2020], a dog-

like robot from Boston Dynamics used to warn people about the social distancing policy in the park. Disinfection robots from UVD [UVD Robots, 2024] have been used in Chinese hospitals to disinfect rooms without human intervention [Ackerman, 2020]. The Brazilian company Instor has worked on Jaci [Instor – Projetos e Robótica, 2020], another robot for disinfection by UV-C (Ultraviolet, class C) [Global Lighting Association, 2021]. Robots such as Temi [Temi USA, 2020] and ARI [PAL Robotics, 2020] help hospitalized patients get in touch with relatives, as well as they help hospital staff to interact with patients from a distance [BBC News, 2020b]. Finally, Pfizer, one of the corporations producing Sars-CoV-19 vaccines, contacted WeRobotics to boot up a new delivering drone model project [WeRobotics, 2021]. Many more applications such as Moxi, from Diligent Robotics [Diligent Robotics, 2021]; Alexia, from Cocuus [NIUS, 2020b]; Rudy, from INF Robotics [Robotics, 2020]; and PP, ROC, and K9 from AIS [NIUS, 2020a] could be herein enlisted.

A second wave of investments in critical-mission application rose right after the pandemic[1], a few months after the announcement of OpenAI's ChatGPT [OpenAI, 2023] first open demonstration, in November 30, 2022 [Forbes, 2023]. Since the launching of ChatGPT, *big-techs* such as Microsoft [Microsoft, 2023] and Google [DeepMind, 2020] have made investments into applying ChatGPT technology into robotics. Also, a not-so-surprising outcome is the announcement of products with "embodied" language models (as in ChatGPT's technology). For instance, 1X Technologies, a company reportedly partnered with OpenAI, announced NEO [1X Technologies, 2023] – an android capable of learning – to be available by the end of 2023.

As more and more applications enter the market, the challenge of delivering reliable hardware-software systems to these products increases. The need for optimized embedded systems is even more demanding in specific applications such as UAV (Unmanned Aerial Vehicles [Sandamirskaya, 2022]. With the recent advancements of the artifical intelligence (AI) field regarding language models, such systems are expected to have dedicated hardware accelerators interacting with the core system to boost applications such as inference in deep neural networks (DNN) [Juracy, 2022].

1.2 Research Context

Re-engineering some state-of-the-art applications from multi-core to manycore architectures seems imminent [Azumi et al., 2020, Chitchian et al., 2013, Haggui et al., 2018, Minozzo et al., 2017, Zhang, 2014, Paul et al., 2014]. As an alternative to the multi-core architectures, manycores try to alleviate the communication throughput by introducing networks-on-chip (NoCs) in the design. Compared to buses, NoCs offer scalable support

[1] The Sars-CoV-19 pandemic officially ended in May 5, 2023 [WHO, 2023].

for massive parallel communication as wire length and clock frequency are unrelated to the number of cores [Heisswolf et al., 2013].

Applications such as autonomous vehicles run real-time software [Saeedi et al., 2018]. Eventually, the uncontrolled execution of real-time applications may put human lives at stake, as robots can operate, for example, UV-C lights, which inappropriate manipulation may result from eye damage to ozone intoxication [Global Lighting Association, 2021]. In manycores, the message-passing programming model [Nürnberger et al., 2014] requires both software and hardware to be adapted to guarantee the RT properties of applications. This is aggravated in applications that have strict bandwidth requirements, e.g., due to a large amount of sensors [Tergel Molom-Ochir, 2020, Yeong et al., 2021, Jeon et al., 2021]. While current RT research on single- and multi-cores extent mostly to the processing elements (PE) of manycores, the RT analysis of the underlying NoC must be accounted.

1.3 Thesis Statement

The community proposed Real-Time NoCs (RT-NoCs) to tackle real-time traffic in NoCs. However, current RT-NoCs mostly require custom routers to achieve RT guarantees. Guarantees appear as WCTT (worst-case transmission time) models [Chen et al., 2021] or schedulability analyses [Picornell et al., 2019]. In both cases, some hardware implementation must support the controlled execution of the schedule or prioritize packets at the runtime, resulting in area and power consumption overheads. The overhead generated by implementing RT routers sums to the realization of other non-functional requirements, leading to an unnecessarily complex NoC design.

Nevertheless, the literature treats non-functional requirements as isolated. Most studies approach at most one or two non-functional requirements and most techniques are bound to the baseline design and are unlikely to be employed in other projects. In such a scenario, this Thesis aims to answer the following research question:

> *"Can we guarantee the real-time properties of an application running on a manycore, from both communication and computation perspectives, using pre-runtime methods, on a non-RT manycore architecture? "*

1.4 Goals

The main goal of this Thesis is to "*provide real-time guarantees for manycore applications with no interference in the target hardware architecture, focusing mainly on mitigating hardware complexity growth due to the accumulated implementation of non-functional*

requirements." We present the specific goals in this Thesis below, showing how we achieve them.

1. Propose a framework instead of a NoC-specific solution: One of the goals of this Thesis is to propose a framework that can be adapted to suit the needs of different NoCs. Although we acknowledge the variety of router designs, we can set a base framework that one can extend to handle different architectures. Due to time limitations, we cannot test our proposal in multiple NoCs. However, we highlight several opportunities for using our approaches in different contexts throughout the Thesis.

2. Enable NoCs to run RT traffic even if they do not have native support: We construct a baseline manycore architecture using open-hardware and open-source software components, favoring the reproducibility of our study. Besides the operating system kernel, none of the software or hardware components supports RT[2]. We use the platform to validate our approaches.

3. Provide a scalable method, agnostic to the NoC dimensions: Scalability is one of the main concerns in NoC technology. For this reason, approaches must scale with the number of network nodes. We achieve this goal by using heuristics whenever the performance of the frameworks become intractable.

4. Develop tools to automate the analysis, reducing human error: One of the key features of the herein proposed frameworks is automation. During the development of the Thesis, we had to develop a set of tools, called ORCA RT-TOOLS (ORT). The tools are the main contribution of the Thesis, along with the underlying frameworks.

5. Contribute to lower the overall cost of designs: We contribute to lower overall design cost by employing only open-software and open-hardware technologies. The ORT suite is open-source and ships a base implementation in Python. All tools and libraries that interact with our tool suite are open-source. The adopted baseline platform has open-source software and open hardware modules only. Adopting open-source and open hardware components also contributes to the reproducibility of our study.

1.5 The Scope

Although we believe our approaches can be applied to a wide range of systems, we had to make assumptions as we could validate our approaches only for a few scenarios. We guarantee our approaches to work in a system where:

[2]The word "timing" has different definitions. In software engineering, it relates to the real-time properties of systems. In the digital and circuit systems area, it relates to the propagation of signals. To avoid misunderstandings, we couple to the term "real-time" when discussing software engineering, leaving the term "timing" only for hardware-level signal analysis.

1. Application-level properties:
   (a) Tasks have a WCET (worst-case execution time) model.
   (b) Packets are periodic, and provide either a rate model (e.g., bytes per second) or a PCD description (period, capacity, and deadline).
   (c) There is no priority between tasks, and all tasks have hard deadlines.
2. System-wide properties:
   (a) Application tasks are statically mapped into cores.
   (b) Task admission cannot occur at runtime, i.e., static workload.
   (c) The whole architecture runs at the same frequency.
   (d) Cores can be asymmetric.
   (e) Processing elements may have at most one central processing unit (A-CPU) core.
   (f) Communication between tasks in the same core is allowed.
3. NoC-Level Properties:
   (a) Routing algorithm must be deterministic and deadlock-free.
   (b) The NoC must provide a zero-load latency model, i.e., analytical latency in the absence of contention.
   (c) Topology must be described as a directed graph.

1.6 The Proposal

This Thesis presents approaches for preventing violations of hard real-time constraints in applications running on NoC-based manycores. We propose three approaches, presented throughout the Thesis, where one approach supersedes the previous approach, the latter being our final proposal. The approach does not impose any hardware implementation on the manycore. Instead, we present a pre-runtime approach based on both scheduling through optimization and discrete-event simulation. The problem of real-time guarantees is approached both from computation and communication perspectives.

1.6.1 Contributions

This Thesis encompasses the following contributions:

- Novel Contributions
  - A method based on discrete-event simulation to model kernel execution, focusing on mono-processing systems (Chapter 4).
  - A model for the characterization of periodic, real-time flows in NoCs; a method for scheduling that flows (Chapter 5).
  - An improved model for modeling flows workflow applications in NoCs; a heuristic algorithm for scheduling traffic; and a method for finding the minimum frequency necessary for such a schedule to work (Chapter 6).
  - A heuristic method for clustering task graphs (Chapter 7).
  - A mixed method for guaranteeing the execution time of real-time tasks and flows in manycore systems (Chapter 8).
- Other Contributions
  - A manycore design on the top of Hermes NoC and HFRisCV CPU (Chapter 3).
  - A NoC driver for the UCX-OS kernel (microcontroller execute)
  - A network interface module to asynchrnous packet transmission in direct memory access (DMA) mode.
  - A set of tools that automatize the herein presented approaches.

1.6.2 Publications Timeline

During the development of this Thesis, the author published the following studies. All studies sum contributions to this Thesis[3]:

- ORCA RT-Bench: A Reference Architecture for Real-Time Scheduling Simulators [Domingues et al., 2021] (SBESC 2021): This work presents a reference architecture for a real-time scheduling simulator. The goal of the study is to deliver a software environment in which a single task set could be submitted to the several scheduling while offering parameters to tune the environment to mimic the task system underlying to several operating systems (OS). We later renamed the tool to ORT/BENCH, presented in depth in Chapter 4 and adopted in one of our frameworks, in Chapter 8.
- Design-time Analysis of Real-Time Traffic for Networks-on-Chip using Constraint Models [Domingues et al., 2022a] (ICECS 2022): In this work, we present our first attempt

[3]Although the title of the papers contains the term “design-time”, the term “pre-runtime” would be correct as the approaches consider that the manycore hardware cannot be changed. The methods proposed in the papers only define the scheduling of flows and tasks prior to the execution of the application.

of scheduling network flows at the pre-runtime. We adapt the abstract notion of resource occupation to networks links, inspired by the job-shop problem and its variants. With the aid of Minizinc [MiniZinc Team, 2024], we search the solution space in a reasonable computing time for small problems. Chapter 5 discusses this work in depth.

- Design-time Scheduling of Periodic, Hard Real-Time Flows for NoC-based Systems [Domingues et al., 2022b] (SBCCI 2022): In this work, we represent a process for finding a network schedule bounded to the target NoC clock speed. By replacing Minizinc by a custom search algorithm we could achieve better execution speed while searching the solution space for valid schedules. Then, we used the searching algorithm as a part of an iterative search process to evaluate the real-time communication flows and find a minimum operating frequency for the target NoC such that packets do not miss any deadline. We discuss this work in Chapter 6.

1.7 Document Organization

Figure 1.1 presents the organization of the manuscript.

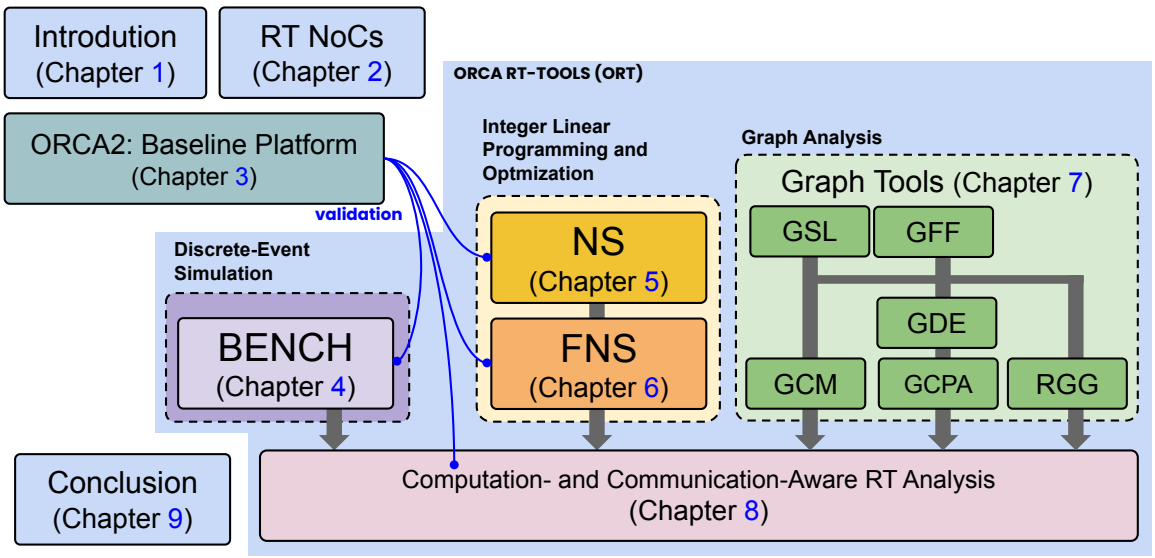


Figure 1.1 – Organization of the Thesis.

- Chapter 2 presents the background and the state-of-the-art approaches for guaranteeing real-time in NoCs, mostly referring to real-time NoCs. We also present a taxonomy on approaches for guaranteeing real-time constraints in NoCs.
- Chapter 3 introduces the ORCA2 manycore, our baseline platform. We discuss both hardware and software components and design decisions.

- Chapter 4 presents the ORT/BENCH tool. The tool is a scheduling simulator focusing on real-time scheduling modeling for mono-processed systems.
- Chapter 5 presents the ORT/NS tool. The tool introduces a real-time flows analysis framework, relying on the concept of the network resources. We present a proof-of-concept using Minizinc to find valid schedules and prevent packet collisions.
- Chapter 6 presents the ORT/FNS tool that finds the minimum frequency necessary to schedule a set of flows in a NoC system using an iterative process and an optimized search algorithm.
- Chapter 7 presents the ORT/GCM, a tool for fast application graph clustering.
- Chapter 8 presents a combined approach for determining the minimum frequency necessary for scheduling both traffic and tasks in manycore systems.
- Finally, Chapter 9 presents the final thoughts and concludes this Thesis.

2. REAL-TIME NOCS

Networks-on-chips (NoCs) are attractive options to interconnect manycore systems due to their potential for massively parallel communication and scalability while inserting low overhead for area and energy consumption to the design [Kumar et al., 2002, Benini and De Micheli, 2002, Jantsch et al., 2005]. In a typical NoC, routers provide the communication infrastructure, allowing multiple data streams to traverse the system simultaneously. As routers represent only a tiny portion of the chip area, NoCs can suit the communication needs of high-end and resource-constrained systems.

Quality-of-Service NoCs (QoS-NoCs) regard the guarantee of non-functional requirements of the communication sub-system, including security and safety [Charles and Mishra, 2020, Guo et al., 2020, Daoud, 2018]. Timing is another non-functional requirement and the primary concern of RT-NoCs: a particular case of QoS-NoCs in which determinism of the system is critical to guarantee predictability. RT-NoCs are particularly important in application domains such as control-theoretic systems, cyber-physical systems, and robotics applications such as autonomous driving systems that must comply with stringent safety constraints and rules [AUTOSAR, 2021, ISO, 2020]. For instance, the failing operation of a self-driving car system may result in catastrophic outcomes such as financial loss, environmental damage, and risk to human lives [Cellan-Jones, 2020]. In such a scenario, RT-NoCs are an alternative to provide timing guarantees in manycores.

The community proposed many RT-NoCs in the last two decades. NoCs timing behavior is generally difficult to predict due to the non-determinism imposed by several components such as buffers, arbiters, and routing algorithms. Although one may predict the behavior of some individual components, representation models rarely correctly represent applications [Renfa et al., 2008]. The community tackles the challenge of real-time communication in manycore systems by employing RT-NoCs, in which the behavior of the external behavior can be predicted, mainly adopting a non-optimal, worst-case scenario. [Bjerregaard and Sparso, 2005, Bolotin et al., 2004, Diemer et al., 2010, Göhringer et al., 2011, Heisswolf et al., 2013, Heisswolf et al., 2013, Liu et al., 2012, Zhan et al., 2014].

We discuss RT-NoCs throughout the remainder of this chapter. Section 2.1 presents classifications of RT-NoCs. Section 2.2 discusses mechanisms for resource allocation, mainly concerning real-time traffic. Section 2.3 discusses the techniques proposed so far and their implementation requirements. Section 2.4 discusses requirements for resource allocation. Finally, we discuss the contributions of this Thesis and how they fit gaps in the literature in Section 2.5.

2.1 Classifications of RT-NoCs

We dedicate this section to discussing the classification of RT-NoCs, which Figure 2.1 briefly illustrates. Hesham et al. [Hesham et al., 2017] proposed most of the classification, including studies on RT-NoCs up to 2017. We refer to more recent work when convenient. It is important to note that RT-NoCs may adopt the same classification employed to NoCs, although discussing general classification for NoCs is beyond the scope of the chapter. We emphasize hard real-time networks-on-chips (HRT-NoCs).

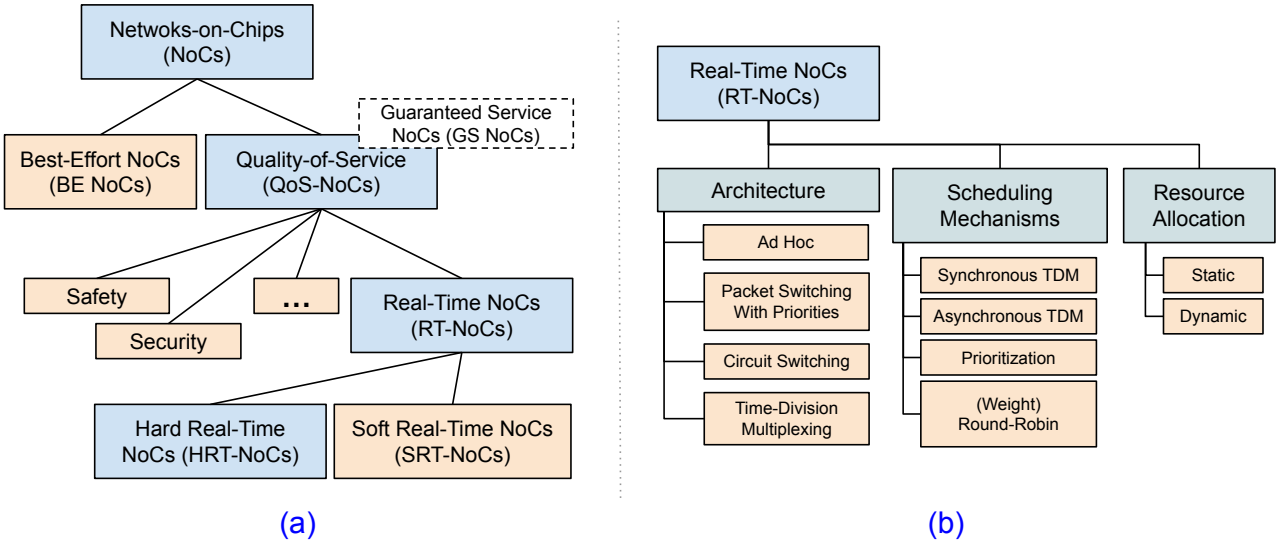


Figure 2.1 – A general taxonomy for NoCs, showing HRT-NoCs a subclass of RT-NoCs and QoS-NoCs (a), and a classification schema for RT-NoCs comprising architecture, scheduling mechanisms, and resource allocation classification (b).

2.1.1 RT-NoCs Architectures

Hesham et al. [Hesham et al., 2017] categorize NoC architectures into Best-Effort NoCs (BE-NoCs) and Guaranteed Service NoCs (GS-NoCs). While BE-NoCs focus on delivering correctness and completion of transmission, GS-NoCs have performance bounds and predictability as their primary concern. GS-NoCs are referred to as quality-of-service NoCs (QoS NoCs). QoS NoCs are further classified as RT-NoCs when concerning timing guarantees. RT-NoC architectures are classified into (i) ad hoc, (ii) packet-switching networks with priorities, (iii) circuit-switching (CS), and (iv) time-division multiplexing. Designs falling into more than one category are often referred to as hybrid NoCs [Hesham et al., 2017]. We briefly describe these categories below.

- Ad Hoc NoCs: A first approach for achieving guaranteed response time in NoCs is to handcraft the design to meet specific real-time requirements of applications, e.g., dimensioning parameters such as buffer width and depth. Xpipes [Bertozzi and Benini, 2004] is known to be one of the first studies to explore the design of ad hoc NoCs. Although the authors could successfully demonstrate the technique for an MPEG4 application, the approach may require additional effort due to the increased number of tasks and flows in modern applications [Shi et al., 2010, Paul et al., 2014]. The main advantage of this approach is the near-optimal utilization of network bandwidth. Disadvantages include the lack of support for scalability and composability.

- Packet-Switched NoCs (PS-NoCs) with Priorities: Packet-switching is a connectionless communication schema in which data traverses the network fragmented into one or more packets [Harmanci et al., 2005, Hesham et al., 2017, Mello et al., 2009]. In PS-NoCs, the path and transmission time of a packet are bound to the control flow mechanism, routing algorithm, arbitration rules, and current network traffic [Hesham et al., 2017]. Since the behavior of PS-NoCs is hard to predict [Hesham et al., 2017], the community proposed prioritization mechanisms [Balakrishnan and Ozguner, 1998, Hyojeong Song et al., 1997, Jong-Pyng Li and Mutka, 1994, Shi and Burns, 2008]. These mechanisms permit high-priority flows to preempt low-priority flows and commonly implement virtual channels (VCs) [Mesidis and Indrusiak, 2011]. Advantages of PS-NoCs include (i) none or low set-up time for transmission and (ii) robustness against network congestion and failure [Millberg et al., 2004a]. Disadvantages include (i) VCs performing worse than multi-plane NoCs for non-regular (aperiodic) traffic [Yoon et al., 2010], (ii) an increased number of VCs per physical channel negatively impact the overall buffer utilization, throughput, and latency of the network [Avani and Agrawal, 2018].

- Circuit-switched NoCs (CS-NoCs): Circuit-switched NoCs (CS-NoCs): In circuit switching, the network establishes a dedicated connection between source and destination routers so that packets can freely traverse the network without interruptions [Wiklund and Dake Liu, 2003, Wolkotte et al., 2005]. Three phases depict the circuit-switching process [Hesham et al., 2017]: (i) path setup – the network checks for path availability, allocating the necessary resources –, (ii) data transfer, and (iii) path release. Another noteworthy concept in CS-NoCs lies in spatial-division multiplexing (SDM). SDM physically isolates flows, occupying one or more lanes, a subdivision of the communication channel between two routers [Wolkotte et al., 2005]. Advantages of CS-NoCs include low area overhead of routers as buffering becomes unnecessary [Wolkotte et al., 2005], and (ii) data transferring depends only on the topological distance between routers [Wiklund and Dake Liu, 2003]. Disadvantages include the costs of establishing and managing circuit connections (e.g., connection setup time and resource reservation) [Bolotin et al., 2004].

- Time-Division Multiplexing NoCs (TDM-NoCs): TDM-NoCs combine virtual channel and CS/PS techniques in a single approach, supporting best-effort and real-time flows to traverse the network concurrently [Millberg et al., 2004a, Ye et al., 2005]. In this approach, the reservation of routers in a path affects only a couple of virtual channels, so the path is still available to handle other flows. In TDM-NoCs, link sharing occurs through time slots, whose assignments are kept in slot tables. Each router has its slot table, updated through configuration messages. These tables store information about paths and packets and the required path. TDM-NoCs forward packets using either circuit- or packet-switching [Yin et al., 2014]. Advantages of TDM-NoCs include the low buffer requirements at the router level [Stefan et al., 2012]. As a disadvantage, mechanisms required for time slot synchronization makes the implementation of TDM-NoCs prohibitively difficult for large chips [Hansson et al., 2009].

2.1.2 Scheduling Mechanisms in RT-NoCs

Heisswolf et al. [Heisswolf et al., 2013] classify scheduling mechanisms as (i) synchronous TDM, (ii) asynchronous TDM, (iii) prioritization, (iv) round-robin, and (v) weighted round-robin. Except for synchronous TDM, all other mechanisms perform asynchronous scheduling, which requires routers to implement buffering [Heisswolf et al., 2013]. Techniques for implementing arbitration are discussed elsewhere [Rao and Sarma, 2017].

- Synchronous TDM (S-TDM): In S-TDM scheduling, path allocation occurs on the same virtual channel address (index) for all routers in the path. Although it reduces the degree of freedom (fewer possibilities for paths), it has simpler hardware when compared to asynchronous TDM (A-TDM) implementations. However, S-TMD has reduced overall NoC utilization compared to A-TDM.
- Asynchronous TDM (A-TDM): A-TDM scheduling allows distinct virtual channels for path allocation. Consequently, the A-TDM overhead tends to be higher than the S-TDM, although it has better overall utilization due to its flexibility in the path allocation.
- Prioritization: Prioritization is an asynchronous scheduling mechanism in which the VCs do not need to be reserved in advance, as the higher-priority VCs will preempt lower-priority VCs. However, one disadvantage of prioritization is the lack of guarantees for resource isolation [Heisswolf et al., 2013], a requirement for QoS NoCs [Grot et al., 2009].
- Round-robin (RR) and Weighted Round-Robin (WRR): RR serves VCs one after another in a fair bandwidth distribution [Heisswolf et al., 2013]. WRR breaks service fairness and allows some VCs to be scheduled more often than others, using a weight parameter.

2.2 Resource Allocation for Hard Real-time NoCs

Heisswolf et al. [Heisswolf et al., 2013] categorize resource allocation in NoCs as (i) dynamic and (ii) static [Brandner and Schoeberl, 2012, Picornell et al., 2019, Picornell et al., 2020, Schoeberl et al., 2012]. Dynamic allocation refers to the allocation of resources at the run-time, usually considering dynamic task allocation and non-periodic flows. In contrast to dynamic allocation, static allocation refers to the design-time allocation (DTA) of resources, primarily considering fixed task mapping and periodic flows. We extend their classification to further categorize static resource allocation into three other categories: (i) pre-runtime (PRT), (ii) design-time configuration, and (iii) design-time generation (DTG). The approaches proposed in this Thesis focus on static resource allocation. Thus, we do not discuss dynamic resource allocation in depth. Figure 2.2 illustrates our classification schema.

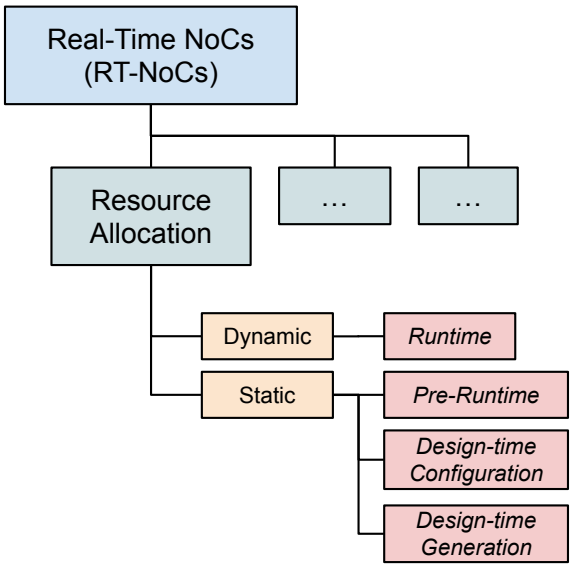


Figure 2.2 – The proposed classification schema, depicting dynamic resource allocation (performed at the runtime) and static resource allocation (performed at the pre-runtime or design-time). Design-time refers to configuration or generation of NoCs hardware.

- *Dynamic Resource Allocation (DRA)* — Runtime: Runtime allocation includes techniques to configure the network traffic during the execution of applications. We do not explore this classification further because this Thesis proposes static approaches. Besides, the literature indicates that a static resource allocation approach is necessary for the system to operate as close to the maximum theoretical bandwidth in the hard real-time domain as possible. For instance, Wiklund et al. [Wiklund and Dake Liu, 2003] write that "*[...] a pre-runtime static scheduling of communications approach is necessary for a hard real-time embedded system to be able to use a significant percentage of the theoretical maximum bandwidth*".

- *Static Resource Allocation (SRA)*: In contrast to DRA, the SRA approach rarely can adapt to different workloads or even overcome unexpected events in the network traffic, e.g. sporadic control traffic.

  – Pre-runtime: Pre-runtime SRA approaches allow for (i) a more straightforward assertion of deadline and constraints of the target application, (ii) reduced or no runtime overhead, and (iii) a vast number of techniques for scheduling generation at disposal when compared to runtime scheduling [Xu and Parnas, 2000]. Pre-runtime scheduling was introduced as a technique for process scheduling (static schedule), later appearing in the RT-NoCs domain. Wiklund et al. [Wiklund and Dake Liu, 2003] and Wolkotte et al. [Wolkotte et al., 2005] were the first authors to apply SRA approaches to control RT traffic. Pre-runtime approaches require mature modeling of the target system and minimal hardware support for arbitrating flows. In contrast to other SRA approaches, pre-runtime approaches require little or no modification to the hardware, mostly requiring software updates in a production setting. However, pre-runtime approaches rarely achieve the best resource usage due to idle hardware in the project. The approaches proposed in chapters 5, 6, and 8 fit in this classification.

  – Design-time Hardware Configuration: Some NoCs may allow for configuration at the design time, e.g., VCs routing [Millberg et al., 2004a]. Approaches in this group present a fixed hardware implementation that can be adjusted to serve different application workflow profiles. Configurations can be performed by hard-writing to flash memories, for example. These approaches differ from design-time generation as the network design does not change as the system parameters change, e.g., one cannot change buffer depth [Moraes et al., 2004]. This classification relates to performance-bound and delay-bound approaches [Bolotin et al., 2004, Diemer et al., 2010, Göhringer et al., 2011, Heisswolf et al., 2013, Heisswolf et al., 2013, Liu et al., 2012].

  – Design-time Hardware Generation: Design-time hardware generation approaches modify the design to match the needs of the application workflow [Diguet, 2014, Goossens et al., 2005b, Hansson et al., 2009, He et al., 2014, Millberg et al., 2004a, Stefan et al., 2014]. For instance, Evgeny et al. proposed QNoC [Bolotin et al., 2004]. Their approach aims to minimize the design cost by trimming unused hardware and allocating resources through optimization. Since changes in the application requirements may require re-running hardware generation, some approaches may rely on FPGA platforms. Compared to application-specific integrated circuits (ASIC), FPGAs can only achieve lower clock frequencies, which can be a barrier to adopting such platforms for the hard real-time domain [Paul et al., 2014].

2.3 RT-NoCs: Proposed Approaches

We extended the list of RT-NoCs implementations presented by Hesham et al. [Hesham et al., 2017] in their survey, adding approaches up to February 2024. We removed some studies from the list, considering only studies on hard real-time. Also, we removed NoC targeting restrict application domains [Li et al., 2016]. Table 2.1 shows an overview of the approaches regarding the adopted switching technique, evaluation metrics for RT, and resource allocation approach (RAT). We emphasize static analytical approaches (see Section 2.2).

1. PCC (SoCBus) [Wiklund and Dake Liu, 2003]: SoCBus addresses RT communication through a run-time, 5-phase circuit-switching (CS) protocol. They claim their approach to be deadlock-free, requiring no VC implementation or specific routing algorithm. They call their approach Packet Connected Circuit (PCC), resembling their CS protocol. The authors did not report information on the scheduling or application characterization.

2. SDM-CCN [Wolkotte et al., 2005]: The study mitigates the configuration issue reported in SoCBus [Wiklund and Dake Liu, 2003] by inserting dedicated links between routers while isolating configuration packets from payload transferring. They introduce the concept of lanes, where a single data path is split into several lanes, regulating lane usage according to the bandwidth needs of the system by a central coordination node (CNN). Although the authors present energy/power results for a set of benchmarks, they did not report information on the characterization of applications and scheduling.

3. Parallel Probe [Liu et al., 2012]: The authors propose a parallel probing method for path setup in CS networks. They claim to achieve reduced area overhead and reduced path setup time. One advantage of this method over previous CS is the predictable setup time. However, the path between transmitting and receiving nodes may vary according to the selected path. However, the authors present no real-time workload to validate their approach or suggest any technique to ensure real-time guarantees in their design.

4. Service Levels [Bolotin et al., 2004]: The study proposes a design-time NoC generation approach in which the parameters of the NoC are tailored according to the target application requirements while minimizing design area and power dissipation. They generate QoS requirements for four classes of traffic: (i) Signaling, (ii) RW/WR, (iii) real-time, and (iv) block. The authors provide benchmarks on uniform and non-uniform traffic scenarios to validate hardware cost minimization.

5. Traffic Shapers [Diemer et al., 2010]: This work presents an approach for optimizing network traffic during runtime. The author argues that pre-run-time analysis can generate optimal routing at reservation time, although it cannot guarantee optimal routing in

changing workload scenarios. Although validating their approach with synthetic flows, the authors do not present a WCTT model or RT benchmark experimentation.

6. Loop Containers [Millberg et al., 2004a]: In contrast to previous work, the author adapts a packet-switched network, Nostrum, to provide guaranteed service through virtual channels. They introduce the concept of loop containers, in which the VC enables the path reservation between source and target nodes. The reservation occurs in both directions, where a container (a packet with special flagging) loops in a virtual circuit. Routing resolves at the design time. The authors do not provide a WCTT model or validation using RT workload.

7. Contention-Free [Goossens et al., 2005b]: This work introduces Æthereal NoC, a CS network with TDM mechanisms. Slot tables regulate the connection between inputs and outputs for each router. Each input can be mapped to only an output for a single time slot, enabling contention-free traffic by construction. The behavior of the connections progresses with time, according to the configuration of the slots. The configuration of slot tables occurs at runtime by control packets. The work showed no validation for RT workload nor a WCTT.

8. Flit-Synchronous TDM [Hansson et al., 2009]: The authors propose a light version of the previously introduced Æthereal NoC. In this version, the authors explore TDM-based arbitration without requiring global synchronicity within the NoC, a limitation to TDM scalability. The authors provide no validation for RT workload or a WCTT model.

9. Setup-Tree/Multi-cast [Stefan et al., 2014]: This work extends a previous implementation of Æthereal NoC[Goossens et al., 2005b] to reduce chip area and packet header overhead. The authors report 10% to 91% of reduced area compared to other NoCs. Although the authors demonstrate that dÆlite surpasses Æthereal regarding connection setup time and traversal latency, they provide no validation for RT workload.

10. SurfNoC TDM [Wassel et al., 2013a, Wassel et al., 2013b]: The author presents SurfNoC, which implements a TDM mechanism over a mesh topology. The work presents non-interference (security) and scalability as the primary concerns. Regarding scalability, the authors demonstrate that their approach scales up with the number of nodes with minimum latency overhead. Moreover, the author provides a WCTT model on their network, which adopts dimension-ordered routing. The authors support their approach by demonstrating experiments for latency and bandwidth overhead. The author did not present validation for RT workload.

11. PhaseNoC TDM [Psarras et al., 2015]: PhaseNoC also has non-interference as a primary concern, introducing the concept of explicit pipelining. Their approach maximizes router utilization because different parts of the router work on different activities, e.g.,

VC allocation, switch allocation (SA), and switch traversal (ST). The authors did not present validation for RT workload.

12. Clockless VC NoCs [Bjerregaard and Sparso, 2005]: The authors present MANGO (Message-passing Asynchronous Network-on-Chip), providing guaranteed services through OCP (open core protocol) interfaces [Accellera, 2024]. The authors claim their NoC to be the first clockless NoC that supports the scalable implementation of independent buffering. The authors did not present results regarding RT workload.

13. WRR VC-Based TDM [Heisswolf et al., 2013]: This work propose a weighted round robin (RR) scheduling mechanism for a packet switching (PS) NoC. This mechanism provide fairness as no high priority packet preempts lower priority packets. The author provide WCTT models for their network and validate delay and throughput models using four RT applications.

14. Adaptive TDM & FIFOs [Diguet, 2014]: The authors present a self-adaptive approach for GS in NoCs in this study. Their approach consists of attaching a self-adaptive network-interfaces (SANIs) to the network. Their routers use reconfigurable TDM tables configured by local manager nodes. The authors do not provide experimentation results for RT workload.

15. Rerouting of Flits [Heisswolf et al., 2013]: The paper presents a mechanism for rerouting packets in intense load situations. The authors validate their work for random traffic scenarios, demonstrating that their approach increases up to 24% of successful GS connections, increasing the overall NoC utilization. However, the authors do not present validation for RT workload.

16. Adapting NoC Size [Göhringer et al., 2011]: The authors extend Star-Wheels NoC to support deterministic worst-case latency analysis in this work. They provide a method for design-time task mapping, although they adopt a greedy algorithm strategy. The work presents no validation for RT workloads.

17. Time Slack-Hetero-VFS [Zhan et al., 2014]: The authors adopt network calculus to model worst-case delay. They use a WCTT model combined with an algorithm to minimize the total communication energy of communication, switching between dynamic voltage-frequency scaling (DVFS) zones while maintaining time correctness.

18. ILP, Heuristics [He et al., 2014]: This work proposes an algorithm to achieve power efficiency in TDM-based NoCs. The authors claim to achieve 25% less energy dissipation with their proposed ILP (integer linear programming) approach approximating optimal solutions. The authors validate their approach using random test cases generated by Task Graphs For Free (TGFF) [Rhodes et al., 2021].

19. DCFNoC [Picornell et al., 2019]: The DCFNoC uses a channel dependency graph to generate layers that map packet collision. Their approach uses the layers to insert delays at the output of routers, mapping layers to TDM channels and preventing packet overlapping. The author validates area and delay, comparing a couple of TDM NoCs, although they do not mention the RT applications they used.

20. Argo 2.0 TDM [Sørensen, 2016]: The author modified the Argo 1.0, adding DMA controllers to the network interface. The routers store the configuration of the DMA within the TDM tables. The author presents a comparison demonstrating reduced scheduling entries in the TDM table. A case study appears in a later study [Rocha et al., 2016].

21. Communication Patterns [Brandner and Schoeberl, 2012]: The study explores core symmetry to provide a static schedule to a TDM NoC implementing VCs. The VCs deliver periodic messages through a fixed routing schedule computed off-line. The authors apply heuristics to reduce schedule length while adopting a constraint model for schedule construction.

22. Temporal Isolation [Perret et al., 2016]: This work constraint programming for a custom execution model, applied to the Kalray Mppa-256 [Kalray, 2024]. The execution model assumes TDM scheduling and relies on a hypervisor architecture running in privileged mode. The authors validate the study for a flight controller [Pagetti et al., 2014].

23. WCA (Worst-Case Analysis) [De Liz Bomer et al., 2023]: The study perform a worst-case analysis on a wormwhole, 2-D mesh NoC, with VCs named SoCIN-Q [Dahule et al., 2023]. The authors provide a model for bound network performance, although the model is architecture-specific. The study is evaluated using a System-C simulator named RedScarf [da Silva et al., 2019].

24. Management of In-Flight Deflection [Wasly et al., 2017]: The authors modify the Hoplite Noc [Kapre and Gray, 2015] to achieve determinism, bound to in-flight latency and queuing time at the source. The study present models for estimating delays with a reduction of 2% in area (LUTs) when compared to Hoplite.

25. Marathon NoC [Kapre, 2016]: The authors present a modification set on the Hoplite NoC to achieve reduced area overhead while targeting FPGA environments. The study discusses the expensive storage requirements as a challenge when deploying TDM techniques to FPGAs.

26. IPDeN NoC [González et al., 2022]: The study presents an approach that inserts small buffers in the network routers to avoid reordering flits in a deflection-based environment. The authors claim that inserting these buffers into routers reduces the overall response time and requires less hardware than VCs.

27. SHARP NoC [Chen et al., 2021]: SHARP NoC employs a priority-preemptive scheduling mechanism implemented within the router. They also propose an analysis model for worst-case latency predictability, also promoting single-cycle multihop as an alternative to the hop-by-hop NoCs.

28. Channel Mapping [Koenen et al., 2020]: The authors present a fault-tolerant hybrid TDM and PS NoC. The performance of the NoC is bound to the channel mapping, where nonoptimal mappings may compromise the NoC performance. They evaluate their NoC for two synthetic applications and several channel mapping scenarios.

29. SP^2 [Ueter et al., 2020]: The authors propose a switching mechanism named SP^2 in which either all the links in the flow path transmit one flit in that cycle or none of the links transmit. By doing so, the authors present the reduction of the NoC schedulability problem to the uniprocessor self-suspension problem.

30-32. Constraint Models [Domingues et al., 2022a, Domingues et al., 2022b]: This Thesis present 3 approaches for scheduling RT traffic in NoCs (see chapters 5, 6, 8). These approaches rely on the occupancy of network links and use optimization techniques to generate schedules at the pre-runtime. We validate the approaches at the end of the corresponding chapters.

    30. The ORT/NS tool (Chapter 5) guarantees deadlines for RT traffic in non-RT NoC (e.g., Hermes). However, it depends on an injection mechanism (Section 3.1.5). Configuring the schedule into the injection mechanism can be costly for extensive schedules, e.g., requiring additional memory space. Hardware-only injection mechanisms may require additional buffer space for storing the schedule. The approach also has impractical computing time for extensive schedules due to implementing ILP. ORT/NS does not account for CPU workload (traffic only).

    31. The ORT/FNS tool (Chapter 6) improves the ORT/NS tool performance, allowing for more extensive schedules. Given an application, the tool can determine the minimum required frequency of the target NoC to run the application. Nevertheless, ORT/FNS does not account for CPU workload (traffic only).

    32. Finally, we combine the ORT/NFS tool with other tools (Chapter 5) to achieve RT guarantees for both computation and communication. The framework relies on simulation to determine the release time of packets, based on the behavior of the task scheduler. The tool can determine the minimum system frequency necessary to achieve both CPU and network RT guarantees, requiring no specif hardware implementation. We validate the approach on the ORCA2 manycore (Chapter 3).

Table 2.1 – Overview of Proposed Techniques for Guaranteeing RT Traffic in NoCs (Thesis contributions in green).

#	Base NoC	Technique	SWT	RT Evaluation Metrics	RAT	CLA
1	SoCBus	PCC [Wiklund and Dake Liu, 2003]	CS	Design-time Setup	[Xu and Parnas, 2000]	PRT
2	—	SDM-CCN [Wolkotte et al., 2005]	CS	Design-time Setup	[Xu and Parnas, 2000]	PRT
3	—	Paralell Probe [Liu et al., 2012]	CS	Bounded Latency	SP	RUN
4	QNoC	Service Levels [Bolotin et al., 2004]	PS	Bounded Latency	SP	DTG
5	—	Traffic Shapers [Diemer et al., 2010]	PS	Bounded Performance	[Diemer and Ernst, 2009]	RUN
6	Nostrum [Millberg et al., 2004b]	Loop Containers [Millberg et al., 2004a]	PS	Design-time Allocation	–	DTC
7	Ætheral	Contention-Free [Goossens et al., 2005b]	TDM	Design-time Allocation	[Goossens et al., 2005a]	DTG
8	Ælite	Flit-Synchronous TDM [Hansson et al., 2009]	TDM	Design-time Allocation	[Hansson et al., 2007a]	DTG
9	dÆlite	Setup-Tree/Multi-cast [Stefan et al., 2014]	TDM	Design-time Allocation	[Hansson et al., 2007b]	DTG
10	—	SurfNoC TDM [Wassel et al., 2013a, Wassel et al., 2013b]	TDM	Static Scheduling	SP	DTG
11	—	PhaseNoC TDM [Psarras et al., 2015]	TDM	Static Scheduling	SP	DTG
12	Mango	Clockless VC NoCs [Bjerregaard and Sparso, 2005]	Hybrid	Bounded Performance	[Zhang, 1995]	RUN
13	—	WRR VC-Based TDM [Heisswolf et al., 2013]	Hybrid	Bounded Latency, Schedulability Analysis	SP	RUN
14	Self-adaptive	Adaptive TDM & FIFOs [Diguet, 2014]	TDM	Design-time Allocation	–	RUN
15	—	Rerouting of Flits [Heisswolf et al., 2013]	Hybrid	Bounded Latency, Schedulability Analysis	[Heisswolf et al., 2013]	RUN
16	Star-Wheels	Adapting NoC Size [Göhringer et al., 2011]	Hybrid	Bounded Latency, Schedulability Analysis	–	DTC
17	Mango [Bjerregaard and Sparso, 2005]	Time slack-hetero-VFS [Zhan et al., 2014]	Hybrid	Bounded Latency, Design-time Allocation	SP	DTG
18	—	ILP, Heuristics [He et al., 2014]	TDM	Design-time Allocation	SP	DTG
19	—	DCFNoC TDM and Channel Dependency [Picornell et al., 2019]	TDM	Bounded Performance	SP	DTC
20	Argo NoC 1.0 [Kasapaki et al., 2016]	Argo 2.0 TDM [Sørensen, 2016]	TDM	Design-time Allocation, Schedulability Analysis	—	DTC
21	S4NoC [Schoeberl et al., 2012]	Communication Patterns [Brandner and Schoeberl, 2012]	TDM	Design-time Allocation, Schedulability Analysis	SP	DTC
22	Kalray Mppa (R)-256 dual NoC	Temporal Isolation TDM [Perret et al., 2016]	TDM	Design-time Allocation	—	DTC
23	SoCIN-Q [Dahule et al., 2023]	Worst-Case Analysis [De Liz Bomer et al., 2023]	TDM	Schedulability Analysis	SP	DTC
24	Hoplite [Kapre and Gray, 2015]	HopliteRT [Wasly et al., 2017]	–	Bounded Latency	SP	DTC
25	Hoplite [Kapre and Gray, 2015]	Marathon NoC TDM [Kapre, 2016]	TDM	Schedulability Analysis	SP	DTC
26	—	IPDeN [Ribot González and Nelissen, 2020]	TDM	Bounded Latency	SP	DTC
27	SMART [Asgarieh and Lin, 2019], SSR-Net [Chen and Jha, 2016]	SHARP NoC [Chen et al., 2021]	—	Schedulability Analysis	SP	RUN
28	[Koenen et al., 2019]	Channel Mapping [Koenen et al., 2020]	Hybrid	Bounded Latency	[Goossens et al., 2005b]	RUN
29	—	SP^2 [Ueter et al., 2020]	—	Schedulability Analysis	SP	RUN
30	Hermes [Moraes et al., 2004]	Constraint Models [Domingues et al., 2022a]	PS	Schedulability Analysis	SP, ITT	PRT
31	Hermes [Moraes et al., 2004]	Constraint Models [Domingues et al., 2022b]	PS	Schedulability Analysis	SP, ITT	PRT
32	Hermes [Moraes et al., 2004]	Constraint Models, Graph Analysis, and Simulation	PS	Schedulability Analysis	ITT	PRT

"–" : Not informed, "SWT": Switching Technique, "CS": Circuit Switching, "PS": Packet Switching, "TDM": Time-Division Multiplexing

"RAT": Resource Allocation Technique, "SP": Same paper, "ITT": In This Thesis

"CLA": Static Resource Allocation Classification (Section 2.2)

"RUN ": runtime allocation, "PRT": pre-runtime, "DTC": design-time configuration, "DTG": design-time generation

2.4 Requirements for Resource Allocation

When allocating network resources (e.g., scheduling), algorithms and techniques might observe some properties that would be useful in practice. It is worth mentioning that these properties appear individually discussed in recent work, although their discussion dates from former publications on QoS NoC. [Grot et al., 2009, Stiliadis and Varma, 1996, Demers et al., 1989]. We discuss these properties below while presenting their alignment to this Thesis and the literature.

1. Isolation of flows and Fairness: Flows must be isolated from each other to avoid burstiness in the network, i.e., the system must preserve the behavior of other flows in the presence of a misbehaving flow. In the hard real-time domain, a failing flow results in a total failure of the application. However, when multiple applications share the same system, one application failing may cause no impact on the other applications running in that system. Approaches that run flows in separated channels, e.g., VCs, favor the isolation of flows. However, the cost of implementing physical flow separation may conflict with other system non-functional requirements, including cost and area overhead. Due to their adaptation features, approaches that fall in the dynamic resource allocation category are less successful in violating flow isolation. However, approaches relying on design-time hardware generation and configuration adopt pessimistic models. Pessimistic models may represent a waste of system resources, although they add possibilities for recovery in case of stuttering. We conclude that the isolation of flows is unbound to the resource allocation time, i.e., each technique must handle it individually. In this Thesis, we guarantee flow isolation by multiplexing the access to the network by flows, like most TDM approaches. (chapters 5, 6, and 8).

2. Low end-to-end latency and delay proportional to bandwidth usage: RT-NoCs insert performance overhead into the design due to the complexity of their mechanisms. For example, TDM techniques often rely on VCs to control traffic, and VC orchestration requires a more robust router. Approaches relying on design-time hardware generation and configuration can generate near-optimal delays due to offline analysis. On the other hand, runtime approaches generate the worst delays due to the unpredictability of the traffic, usually adopting complex worst-case models for guaranteeing RT. Finally, approaches based on contention-free traffic can present no added delay, where most of the analysis is performed offline. One of the drawbacks of contention-free approaches is the requirement for an injection mechanism, as packets must enter the network in an exact cycle to avoid packet collisions. In this Thesis, we guarantee no added delays by implementing a contention-free approach. The approach in Chapter 5 requires an injection mechanism, while we lift this restriction in approaches presented by chapters 6 and 8.

3. Efficient bandwidth utilization: Efficient bandwidth utilization corresponds to the capacity of the approach to reduce bandwidth waste. For example, the SP^2 [Ueter et al., 2020] approach retains flits in the buffer unless all routers in the path can forward flits to the next router (store-and-forward). The all-win behavior of the approach increases the number of cycles in which flits reside in the network, thus reducing the overall network utilization. On the other hand, approaches such as contention-free uses only the necessary network time to transfer packets. However, other packets, e.g., best-effort traffic or control packets, cannot use the idle bandwidth, representing waste in practice. The approaches of this Thesis (chapters 5, 6, and 8) focus on hard real-time applications only, so we are aware that the unused network bandwidth cannot be claimed by sporadic traffic.

4. Low performance overhead: Approaches should reduce resource provisioning overhead to a minimum. In a circuit-switching network, the channel reservation phase requires the setup of all routers in the path. During the setup, part of the routers in the path become occupied, even without data to transfer, until the setup has finished. After the transfer, there is also a release phase. Waste of bandwidth also happens between the transferring and the releasing of network resources. Waste also happens in TDM networks due to the effort necessary to manage VCs. The approaches presented in this thesis (chapters 5, 6, 8) employ offline provisioning of resources, thus requiring no setup time during the operation of the system.

5. Flexible bandwidth allocation: There is a trade-off between the grain of resource allocation and resource utilization. On the one hand, a coarse allocation grain may represent a waste of network bandwidth, although simplifying provisioning. On the other hand, a fine grain can present near-optimal resource utilization at the cost of a complex analysis framework. Approaches such as TDM allow for a configured allocation grain, i.e., time slots. The frameworks proposed in this Thesis employ a minimal constraint model while delegating the solving efforts to either ILP (chapters 5) or custom searching algorithms (chapters 6 and 8). This separation of concerns enables practitioners to use our frameworks through a full-automated process, even without deep knowledge of the underlying analysis.

6. Simplicity of Implementation: Most of the approaches in Table 2.1 require modifications to the design of the NoC. As one adds more and more features to the NoC, e.g., support for security, real-time, and energy savings, the more complex the design becomes. Approaches must consider making fewer modifications to the design, avoiding interference in the implemented features and favoring the implementation of new ones, reducing the complexity of the design. We observed that this requirement is discussed in very few papers. One of the most significant contributions of this Thesis is the proposal of approaches that cause no interference at the hardware level.

7. Scalability: Scalability is one of the key points of the NoC technology. Thus, approaches dealing with RT must scale up to the size of the NoC without compromising the performance of the system. Fortunately, most of the RT approaches in the literature prioritize scalability. Our first approach (Chapter 5) has scalability issues due to the adoption of ILP. Although not directly impacting the NoC performance, the approach cannot deal with larger problems. Our later approaches (chapters 6 and 8) surpass the scalability issues of the first approach.

8. Low energy and area overhead: Area and power consumption of the system increases with the size (number of routers) and complexity (kind of router) of the NoC. For instance, larger buffers in PS NoC lower network congestion at the cost of chip area overhead. We observed that most studies calculate the area and energy consumption increasing on the actual parameters of the baseline platform. However, most adopted baseline platforms present one or another features that approaches can take advantage. Our approaches consider no special feature from the baseline NoC, although the underlying models must consider the behavior of routing, switching and other parts of the NoC. Even without taking advantage of the baseline platform hardware, our approaches run offline (chapters 6 and 8), representing no energy or area overheads.

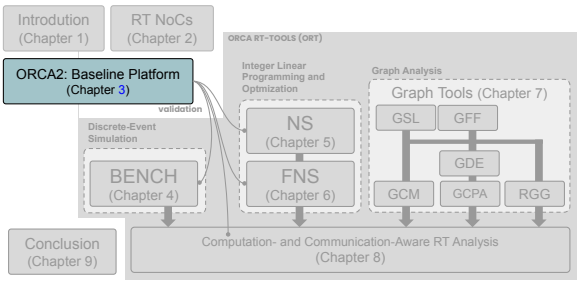
2.5 Final Remarks

This chapter presented a taxonomy on RT-NoCs, summarizing the state-of-the-art contributions of RT-NoCs and discussing requirements for resource allocation in manycores. Table 2.1 synthesizes studies on RT in manycores, where all approaches (1 to 29) require hardware support for their approach to work. Likewise, our first approach (30, Chapter 5) also requires hardware support.

Our other approaches (31 and 32) present similar benefits to the contention-free approaches while requiring no intervention at the hardware level, relying only on an offline optimization process. The approach is a novel contribution to the literature and, as far as we know, the only framework directed to general-purpose NoCs. As discussed in Section 2.4, our approach presents most of the guarantees regarding resource allocation at the advantage of allowing its application on low-cost NoC projects, e.g., Hermes [Moraes et al., 2004].

As a future work, we intend to extend this chapter to a systematic review of the literature on real-time NoCs and other approaches for guaranteeing RT in manycores. One concern to address in the review is to extend the classification of NoCs, mainly providing a clear separation of *ad hoc* studies from general approaches. Other classifications may apply, such as the target technology, e.g., FPGA (field programmable gate array) vs. ASIC (application-specific integrated circuit).

3. THE ORCA2 MANYCORE - BASELINE PLATFORM



We developed the ORCA2 manycore to experiment with the proposed approaches in a real-world environment, adopting production-ready technology as much as possible. As explained throughout this chapter, the manycore redesigns the ORCA manycore [Domingues, 2020], comprising updates in several platform components. To keep the reproducibility of our study, we adopt only open hardware and open-source software components. The ORCA2 manycore is available at https://github.com/andersondomingues/orca-rt-tools and partakes in the following chapters:

- Chapter 4: The ORT/BENCH implements discrete event simulation for a scheduling simulator. Although we do not run RTL simulations in the Chapter, the characterization of applications and kernel follow the configuration of the ORCA2 platform, e.g., scheduling and interruption handling model.
- Chapters 5 and 6: We evaluate the approaches by simulating our platform while replacing the CPU by the TCNI (time-controlled network interface, Section 3.1.5) module. A software-generated test-bench provides the interface between the TCNI and the NoC.
- Chapter 8: We run a full system simulation at the register-transfer level (RTL) level, including all hardware and software components discussed in this Chapter with exception to the TCNI.

To trim the time spent on building hardware from scratch, we adopted a couple of hardware modules from other projects. From the GAPH/PUCRS group, we adopted the Hermes NoC, which is a well-established project developed in 2004 [Moraes et al., 2004]. Since then, the project has received contributions from many MsC. and Ph.D. projects within the lab. From the GSE/GAPH group, we adopted the HFRISC processor core due to its compatibility with the RISCV32e instruction set architecture (ISA). Hardware developed in the context of this Thesis includes behavioral models for a dual-port scratchpad memory and TCNI (time-controlled network interface), as well as synthesizable modules, e.g., DDMA (Dual-Mode DMA). The TCNI, DDMA, and top-level design are novel contributions to this Thesis. We conduct all RTL simulations using Siemens' Modelsim SE-64 (Standard Edition), version 2021.07 (July 13, 2021). [Siemens, 2024]. We present the hardware of the ORCA2 platform in Section 3.1.

As for the software, we mainly relied on software from the GSE/PUCRS lab. We adopted the UCX-OS kernel [Johann, 2024d] due to its compatibility with the HFRISC CPU

core, and it ships necessary automation scripts to compile and run applications out-of-the-box, requiring only the GNU toolchain cross-compiler targeting the RISCV-32e instruction set. Our contributions regarding software include a network driver (upper and lower-level), applications, and HAL (hardware abstraction layer) configuration. Section 3.2 details the software.

3.1 The Hardware Stack: ORCA2 Manycore Architecture

The adopted baseline platform is a many-core system implemented on top of the Hermes [Moraes et al., 2004] network-on-chip (NoC). The many-core is organized in tiles, displaying a mesh-like topology, where routers connect to at most four neighbor routers (north, south, west, and east directions), forming a 2D discrete Cartesian plane of tiles. Routers also has a local port that connects each router to a processing element (PE). The PE hardware follows a fixed configuration, consisting of a DMA-like network interface (DDMA), the HFRISC/Riscv32-e CPU [Johann, 2024b], scratchpad memories (SPM) for boot ROM and RAM, and peripherals support (e.g., programmable counters). Figure 3.1 shows an overview of the many-core organization.

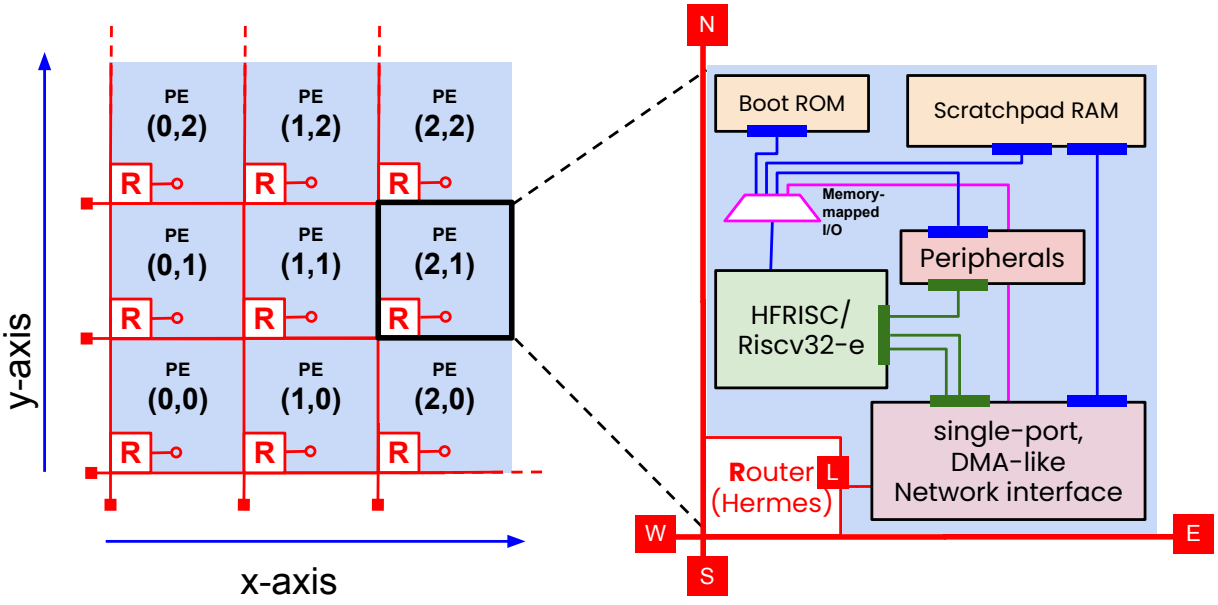


Figure 3.1 – Illustration of the tile-based architecture of the ORCA2 platform (left), depicting communication channels (red lines), routers (R), and processing elements (PE). PE tiles (magnification, right) comprise memory cores (ROM/RAM), CPU, router, and DDMA.

3.1.1 Hermes NoC and Communication Infrastructure

PE tiles communicate through packets. A packet is a sequence of flits. A flit corresponds to a fixed-length string of bits and represents the smaller data chunk transferable through the NoC. In Hermes, the length of a flit is configurable, although we adopted a 4-byte (32 bits) flit width. The depth of input buffers is also configurable, yet we choose a depth of 8 flits per buffer (one buffer per input port). Also, the number of routers in the NoC corresponds to the number of tiles, allowing for a theoretical value of 2^{16} = 65536 tiles, i.e., a 256 $\times$ 256 dimension mesh. The maximum packet size is $2^{32} - 1$ flits (approximately 4GB).

The leading flit of a packet identifies the address of the target router. We configure the address of routers at the design time following the physical addressing schema presented in Figure 3.2. When a router receives a packet whose destination address is the same as its address, it forwards the packet to its `LOCAL` port. When the destination address differs from the address of the router, the packet gets routed to one of the neighbor routers (`NORTH`, `SOUTH`, `EAST`, or `WEST`) using the XY algorithm.

In the XY routing algorithm, routers push flits in the X-axis of the NoC until they reach the same Y-coordinate of the destination router. Then, the router pushes flits toward the Y-axis until it reaches its final destination. The XY algorithm is deterministic and deadlock-free [Glass and Ni, 1992]. In Hermes, the leading flit (address flit) takes up to 7 cycles to route from one port to another. The remaining flits — address flit and beyond — follow the same path, one flit per cycle (contention-free environment).

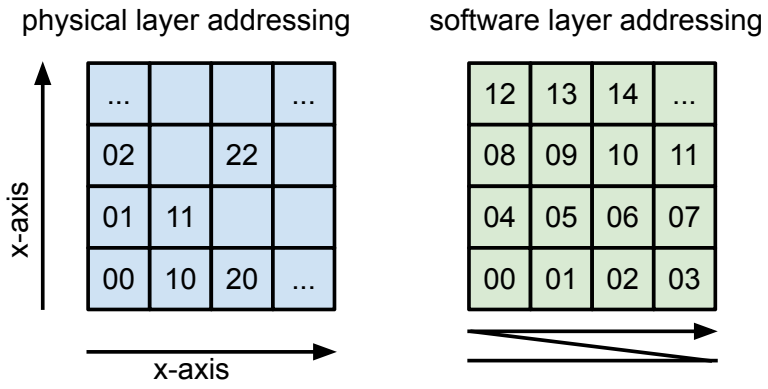


Figure 3.2 – Network addressing system used within ORCA2. Physical layer addressing (left) follows a pair (x,y) according to tile coordinate in the NoC. Software layer addressing (right) follows a sequential pattern (adapted from [Domingues, 2020]).

The second flit of a packet is the size flit; it carries the size of the packet (in flits). The theoretical maximum packet size is $2^{32} - 1$ flits. Flits arrive at the input buffers in the same order they leave the output buffer (in-order flit delivery). Routers implement the wormhole strategy for packet switching, transferring flits without data interleaving. The router serves ports following a round-robin policy in a circular queue fashion. Finally, the switch

control component performs routing and packet switching, while the crossbar component establishes the connection between ports. Figure 3.3 shows the Hermes router interface.

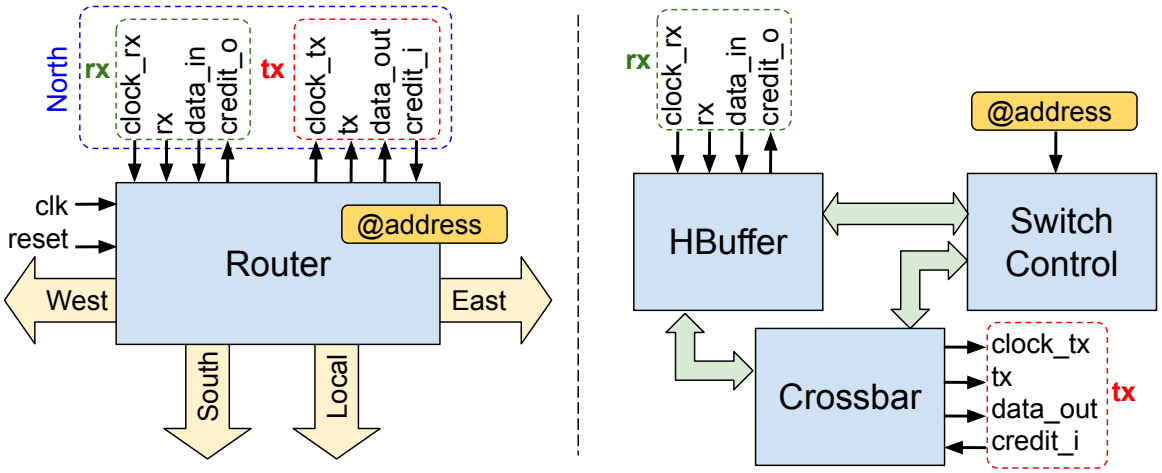


Figure 3.3 – A router (left) and its components (right) (adapted from [Domingues, 2020]). The interface of the five ports (North, South, East, West, and Local) is the same.

The interface of buffers is the same for all ports. Four signals control the receiving of new flits at each port: `clock_rx`, `rx`, `data_in`, and `credit_o`. Although Hermes individual clock domains per port, we assume the same clock signal for all modules; thus, the signal `clock_rx` of all routers is the same as the global clock signal. The signal `rx` (receive) raises when new data is available at `data_in`. Data stays at the buffer until routed to another port. When the buffer becomes full, the `credit_o` raises, preventing senders from injecting new data into the buffer [Moraes et al., 2004].

3.1.2 The HFRISC/rv32e Processor

The HFRISCV core is a 32-bit processor targeting the rv32e Risc-V standard [RISC-V International, 2023, Waterman and Asanović, 2017], in a 3-stage pipelined architecture (stages: fetch, decode, and execute), machine mode only. The architecture accounts for 32 user-level registers and four instruction formats (R, I, S, and U-type instructions). Instructions take 2 to 4 cycles to traverse the pipeline: (i) branches not taken spend 2 cycles, (ii) memory operations take 4 cycles, and (iii) the remaining operations take 3 cycles. Table 3.1 shows the instructions available in the architecture [Domingues, 2020].

The HFRisc/rv32e processor has two interface groups. The first group matches the wiring of the memory module (Mem. I/F, Figure 3.4). Although we could connect the processor directly to the RAM, we instead insert an intermediary multiplexer to resolve memory-mapped I/O registers (MMIO). The multiplexer switches between ROM, RAM, DDMA, and peripherals memory spaces (Figure 3.1, circles 1, 2, and 3). Figure 3.5 shows the memory zones (memory map) that our platform adopts.

Table 3.1 – Instructions of the HFRISC (rv32e) core.

Instruction Type	Subclass	Available Instructions
R-Type	Non-IMM Arith.	ADD, SUB
	Logical	XOR, OR, AND
	Shifts	SLL, SLR, SLA
	Comparison	SLT, SLTU
	M extension	MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU
I-Type	IMM Arith.	ADDI
	Logical	XORI, ORI, ANDI,
	Comparison	SLTI, SLTIU, SLLI, SLRI, SLAI, SLTI, SLTIU
	Loads	LB, LH, LW, LBU, LHU
	Synchronization	FENCE, FENCE.I
	System	SYSCALL, SBREAK
	Counters	RDCYCLE, RDCYCLEH, RDTIME, RDTIMEH, RDINSTRET, RDINSTRETH
S-Type	Stores	SB, SH, SW
	Braches (B-Type)	BEQ, BNE, BLT, BGE, BLTU, BGEU
U-Type	Addressing	LUI, AIUPC
	Jump-and-Link (J-Type)	JAL, JALR

There are two pins for external I/O, which we connect to the peripherals module. The peripherals module orchestrates incoming interruption requests, acting as a PIC (programmable interruption controller). The platform has three interruption lines attached to the processor: one for a programmable counter interruption (e.g., used by the scheduler) and two for the DDMA to use during packet reception.

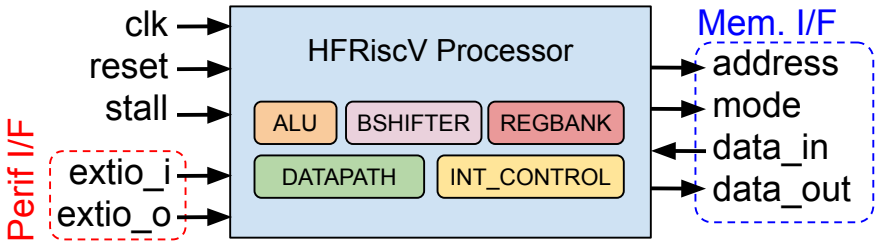


Figure 3.4 – The HFRISC processor interface (adapted from [Domingues, 2020]).

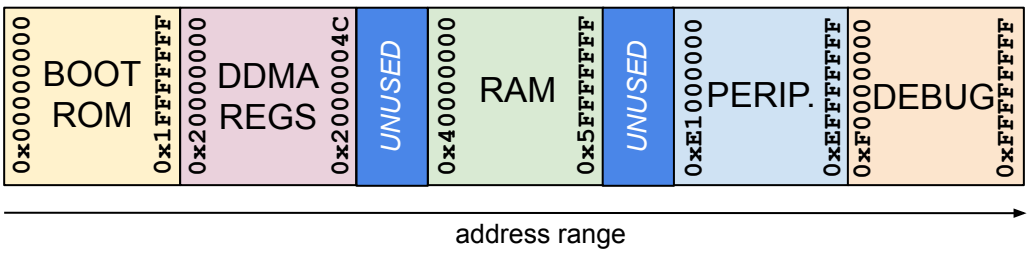


Figure 3.5 – Memory map of the ORCA2 platform.

3.1.3 DDMA - A Custom Network Interface featuring DMA

We avoid CPU stalls during data transferring by adopting a dual-port memory model within the PE. One memory port attaches to the CPU, while the other attaches to a DMA-like network interface module called DDMA (Dual-Mode DMA). The DDMA module implements an asymmetric DMA (direct memory access) mechanism, allowing the processor core to push/pull data from the underlying networking system asynchronously. In other words, the DDMA can transfer CPU-to-network (and network-to-CPU) data while minimizing CPU interruptions. Figure 3.6 shows the interface of the DDMA module.

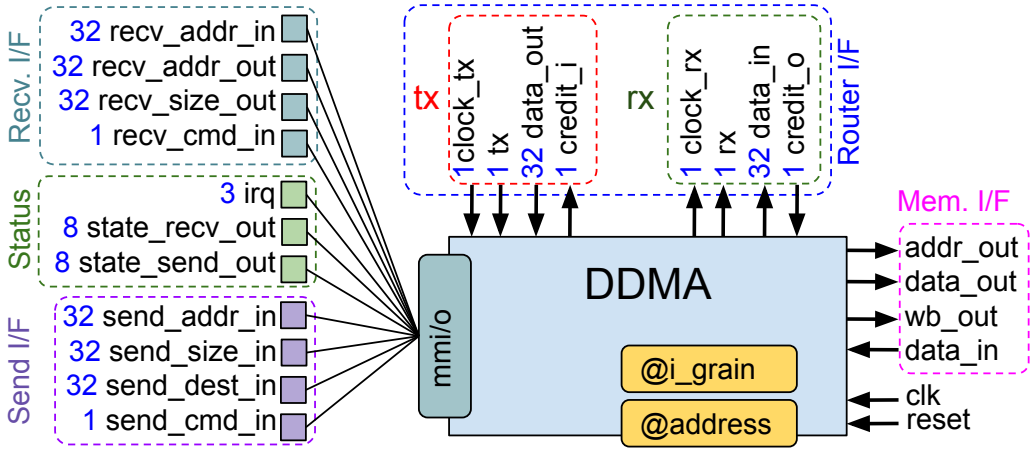


Figure 3.6 – The DDMA module and its interface. Signals are represented by arrows, where arrows pointing onto the module indicate input signals. Design-time parameters are tagged with a '@' character. Finally, memory-mapped registers are identified by an square shape near to their names.

The DDMA attaches to the CPU, memory, and router modules. On the CPU side, the DDMA interface provides controls for configuring send and receive operations. To send a packet, the network driver has to write to the following four registers: `send_addr_in`, `send_size_in`, `send_dest_in`, and `send_cmd_in`. The former three registers store information about the data to be copied (memory pointer, data size in bytes, and destination node number). The driver raises the command register flag (`send_cmd_in`) to confirm the sending operation. The command register flag is sensitive to the state of the internal sending process. Thus, the driver reads the status registers before confirming the sending operation. When the sending operation finishes, an interruption request signal rises.

When a packet reaches the local router (router `tx` signal), the DDMA raises an interruption request asking the CPU to generate a valid memory location to store the packet in, also informing the length of the incoming packet (`recv_addr_out`). Then, the CPU writes the address for the packet to the `recv_addr_in` and confirms the operation writing to the `recv_cmd_in` register. The DDMA discards the first two incoming flits (header and size), copying the following flits into the memory space. After the copy, another interruption request

tells the CPU that the packet is ready to be delivered to the software. The address of the received packet is pushed into the `recv_addr_in` to confirm the operation.

The status registers include three interruption request pins and two 8-bit registers to indicate the state of the send and receiving processes (one-hot). Receiving and sending operations only occur if the corresponding state machine resides at the `IDLE_STATE` (state zero). Finally, a fully combinational module, the arbiter, controls the access to the memory port. Since the memory port is shared by the send and receive processes, the arbiter implements a configurable (through the `i_grain` parameter) interleaving logic.

3.1.4 Scratchpad Cores

Our platform uses two scratchpad memory cores within its tile architecture, one for the ram (random access memory) and another for the ROM (read-only memory). By design, the ROM memory resembles the RAM module, although the writing pin is disabled in port A to avoid overwriting the ROM. Also, port B is wholly unused and removed from the design. The ROM memory is connected to the CPU through a multiplexer through port A. The multiplexer performs address space translation, connecting the Port A of the RAM to the CPU. Thus, the CPU core can read from both memory cores through the multiplexer (and other memory zones). Finally, port B of the RAM module connects to the DDMA module. Figure 3.7 shows the interface of the scratchpad memory core.

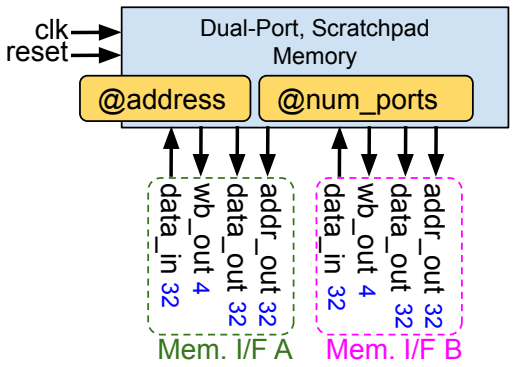


Figure 3.7 – The scratchpad memory core, displaying two ports (A and B).

3.1.5 TCNI - Time-Controlled Network Interface

The TCNI (time-controlled network interface) is a behavioral module that times the injection of packets in the network. The approaches presented in Chapters 5 and 6 rely on a constraint model to calculate the time in which packets must enter the network, guaranteeing

that packets will meet their deadline. The TCNI module is our design proposal to employ such approaches in the ORCA2 platform. Figure 3.8 shows the interface of the TCNI module.

Since our platform relies on the DDMA to push packets into the network, we insert the TCNI in the design between the MMIO multiplexer and the DDMA modules. The goal is to intercept the configuration of the DDMA, delaying it until the release time of packets. For this reason, the interface of the TCNI includes the `pointer_in` and `size_in` registers, resembling the `send_addr_in` and `send_size_in` registers of the DDMA module. The `time_in` register indicates the cycle in which the TCNI must activate the DDMA. Finally, the `status_out` reports the status of the DDMA, and the `valid_in` register activates the configuration of the TCNI. Figure 3.9 shows the positioning and interaction of the TCNI in the PE architecture.

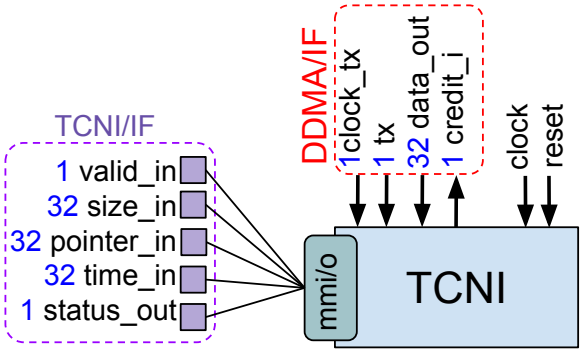


Figure 3.8 – The TCNI module interface.

We designed the TCNI module only at the behavioral level to use some high-level abstractions in the design, such as priority queues. Since the TCNI has no interaction with the task scheduler, we could not guarantee that the software would activate the TCNI by the order of packets. Ideally, we could sort activations in the software drive, although this could cause a possible delay in the operation of the TCNI. To keep the injection of packets strict to their release time, we discarded the out-of-order software driver. Another possibility is to implement a priority queue in hardware. In this case, we would have to determine the number of configurations to reside in the TCNI simultaneously, which would also require some sorting mechanism at the hardware level.

3.2 Software Stack: Operating System, Drivers, and Applications

We reduce the compilation time of the software image by compiling only a single binary for all PE. When simulating at the RTL (register-transfer level), we use a behavioral scratchpad that loads the image into the memory space for the multiple PE. As a result, we must include all application tasks and necessary software libraries in the same image. Memory-mapped registers generated at elaboration time indicate the X and Y coordinates

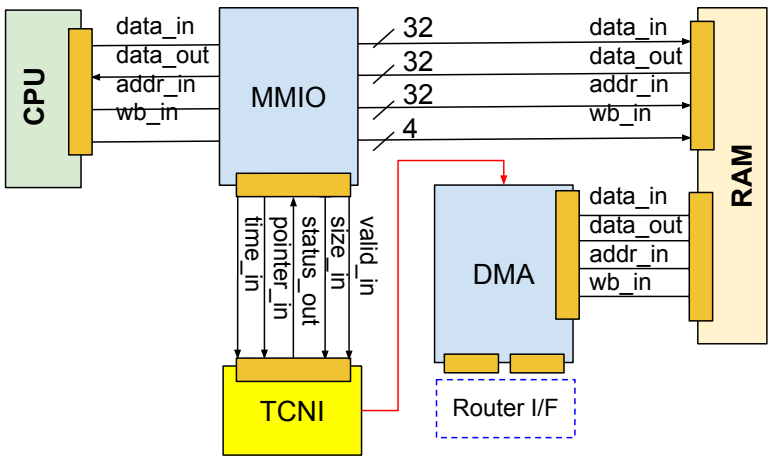


Figure 3.9 – The TCNI module within the PE in the ORCA2 architecture.

of each PE. During the boot of the kernel, the OS selects tasks to run accordingly to the XY coordinate of the PE.

We compiled all software using a cross-compiler GNU's C Compiler (GCC) version 12.2.0 targeting the riscv32e architecture. The whole software is written in C, with minimal assembly coding at the HAL level.

3.2.1 The UCX-OS Kernel

The UCX-OS Kernel (μ-Controller eXecutive) is a lightweight kernel targeting microcontroller and embedded devices [Johann, 2024d]. Components of the UCX-OS kernel include a simplified standard C library, a preemptive scheduler for best-effort and real-time application, and ports for a couple of processor cores — including the RISC-V/32i, RISC-V/32e, and MIPS1 standards. We adopt the RISC-V/32e port, mainly due to its compatibility with the HFRISC/32e processor core. Figure 3.10 shows the main components of UCX-OS.

3.2.2 Task Scheduler

We perform a couple of modifications to the UCX-OS kernel. First, we implement a blocking mechanism considering the dependency between tasks to avoid unnecessary task preemption. If two tasks communicate and they do not reside in the same CPU, we say that the receiving tasks depend on the sender task. In other words, the receiving task must receive one or more packets from the sender to start processing. We use this rationale to put tasks without the required input packets in a blocked state and avoid scheduling the tasks unnecessarily. Once the receiving task tries to receive a packet, it looks at the packet queue of the network driver. If the queue is empty, the task will voluntarily put itself in a blocked

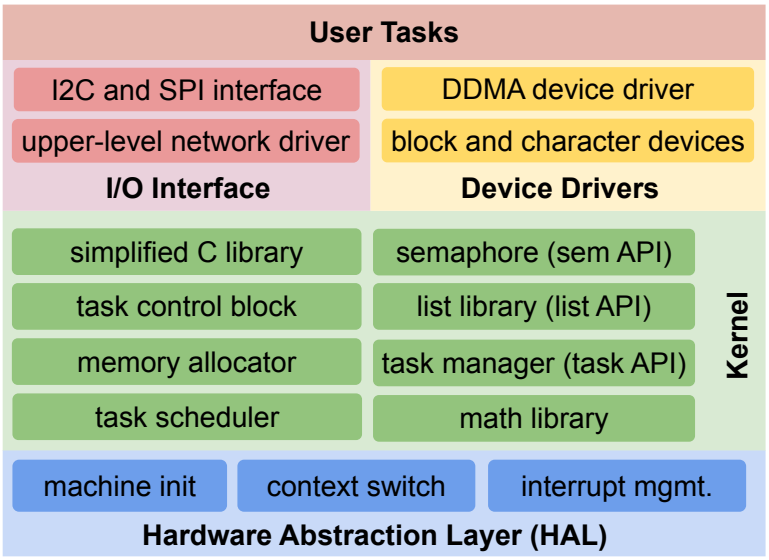


Figure 3.10 – The UCX-OS kernel.

state. Once that task receives a packet, the network driver changes the task state to ready so the scheduler can elect it to run in the next scheduling activity. If no task is able to run, the scheduler will artificially schedule the `idle` task, powering down the CPU.

Also, we added an interruption to enable input tasks periodically as if the application were receiving data from outside the system. With such a mechanism, we can control the rate of workflow applications.

In addition to the above features, our scheduler implements the SIARR (Selective Iteration-Aware Round-Robin) algorithm. The algorithm aims to mitigate momentarily performance slowdowns. In SIARR, the scheduler stores the current iteration number for each task. As tasks advance towards higher value iterations, tasks with lower ones get priority. If two non-blocked tasks share the priority, the successor tasks get priority over the predecessor tasks. The SAIRR extends the PBRR scheduling technique [Nithya and Itapu, 2023] and is a novel contribution to this Thesis.

3.2.3 The Network Driver

The UCX-OS has no native support for networking. We ported the network layer we used in previous work [Domingues, 2020] within the HellfireOS kernel [Johann, 2024a], performing a few modifications:

1. We split the driver into two layers: (i) upper-level network driver and (ii) lower-lever network driver.
2. We dramatically reduced the driver size by removing the shared packet queue and dynamic port mapping features.

3. We moved the interruptions call to the upper-level driver. The goal is to make the interruption handling implementation device-specific.

The lower-level network driver is responsible for interacting with the underlying network hardware. In this Thesis, the lower-level driver commands the DDMA attached to the NoC. Table 3.2 shows the application programming interface (API) for the lower-level network driver. The `send` operation relies solely on the `_ddma_send_status` and `_ddma_send` operations. Once the driver calls `_ddma_send`, the operation will internally call `_ddma_send` to check on the status of the sending state machine. Even if multiple tasks invoke the send operation, only one task can pass through the checking at a time, creating a virtual queue in the stack. Please note that tasks will not become blocked to avoid deadlock situations.

Table 3.2 – Operations of the Lower-Level Network Driver

Operation Alias	**Parameters**	**Function**
_ddma_init	*none*	Reset DDMA state machines and print configured node information, including node XY coordinates and NoC dimensions.
_ddma_send	destination, size, payload	Configures the DDMA to produce header flit equals to *destination* and size flit equals to *size*, also informing the DDMA that the beggining of data is at address equals to *payload*. After configuring the parameters, it activates the DDMA to send the packet.
_ddma_send_status	*none*	Report the status of the sending state machine. A zero value means that the state machine resides at the `idle` state. Non-zero values means that the state machine is busy.
_ddma_recv_size	*none*	Returns the size of an incoming packet. Called form the upper-level network driver once an incoming packet triggers an interruption.
_ddma_set_recv_addr	address	Set the address to which the DDMA must store the incoming packet. The address is generated by the upper-level network driver.
_ddma_get_recv_addr	*none*	Returns the last address used by the DDMA to store packets.
_ddma_recv_ack	*none*	Acknowledges to the DDMA that the received packet is stored into the packet queue and the driver is ready to receive more packets.
_ddma_recv_status	*none*	Report the status of the receiving state machine. A zero value means that the state machine resides at the `idle` state. Non-zero values means that the state machine is busy.
_ddma_node_addr	*none*	Return the software layer address of the current network node (PE). Nodes are adressed as explained in Section 3.1.1.
_ddma_atox	address	Converts the software layer *address* to a hardware layer address and returns the X-axis component of the converted address.
_ddma_atoy	address	Converts the software layer *address* to a hardware layer address and returns the Y-axis component of the converted address.
_ddma_xyta	xcoord, ycoord	Converts the hardware layer address (X = *xcoord*, Y = *ycoord*) into the corresponding software layer address.

The DDMA activates the `receive` operation once a packet hits the input port of the router. First, the DDMA receives the first flit in the network and checks whether the packet arrived at the right destination, dropping malformed packets. Then, the second received flit informs the size of the incoming packet in bytes. The DDMA forwards this information to the upper-level network driver, which accesses the internal register through the `_ddma_recv_size` operation. Then, the upper-level driver allocates enough memory to store the packet using the `_ddma_set_recv_size`, informing the DDMA where to store the packet.

As the lower-level network driver can only handle sending and receiving packets atomically, the upper-level driver aids the kernel in controlling the lower-level driver while providing an API for tasks to use the underlying network. Table 3.3 shows the operations of the upper-level network driver. The `send` operation occurs asynchronously due to the DDMA capabilities. The `receive` operation occurs asynchronously as well.

Table 3.3 – Operations of the Upper-Level Network Driver

Operation Alias	Parameters	Function
_ddma_init	*none*	Reset DDMA state machines and print configured node information, including node XY coordinates and NoC dimensions.
_ddma_send	destination, size, payload	Configures the DDMA to produce header flit equals to *destination* and size flit equals to *size*, also informing the DDMA that the beggining of data is at address equals to *payload*. After configuring the parameters, it activates the DDMA to send the packet.
_ddma_send_status	*none*	Report the status of the sending state machine. A zero value means that the state machine resides at the `idle` state. Non-zero values means that the state machine is busy.
_ddma_recv_size	*none*	Returns the size of an incoming packet. Called form the upper-level network driver once an incoming packet triggers an interruption.
_ddma_set_recv_addr	address	Set the address to which the DDMA must store the incoming packet. The address is generated by the upper-level network driver.
_ddma_get_recv_addr	*none*	Returns the last address used by the DDMA to store packets.
_ddma_recv_ack	*none*	Acknowledges to the DDMA that the received packet is stored into the packet queue and the driver is ready to receive more packets.
_ddma_recv_status	*none*	Report the status of the receiving state machine. A zero value means that the state machine resides at the `idle` state. Non-zero values means that the state machine is busy.

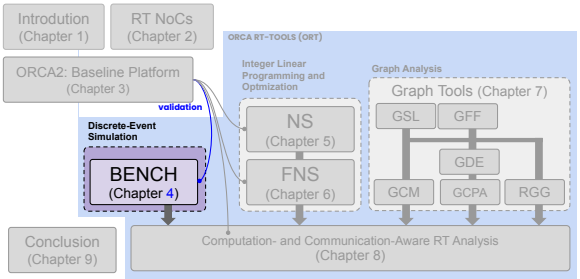
3.3 Final Remarks

This chapter presented ORCA2, the baseline manycore architecture adopted within this Thesis. We built the platform considering the reuse of components and the low cost of

implementation. In the future, we intend to augment the platform to work with the rest of the tools presented in the next chapters of this Thesis out-of-the-box. The goal is to create an environment for fast inspection of the system, e.g., scheduler status and debugging interface. The environment would include simulating the system in a coarser abstraction grain, e.g. instruction-level.

4. ORT/BENCH: PREDICTING THE BEHAVIOR OF TASK SCHEDULERS WITH DISCRETE-EVENT SIMULATION

This Chapter presents ORT/BENCH, a task scheduler simulator tool targeting mono-processing systems, part of the ORT suite and required by the approach in Chapter 8. The following publication relates to the tool:



> ORCA RT-Bench: A Reference Architecture for Real-Time Scheduling Simulators
> Domingues, A. R. P; Benno, J.; Amory, A. M.; Moraes, F. G.
> XI Brazilian Symposium on Computing Systems Engineering (SBESC). 2021.
> https://ieeexplore.ieee.org/document/9628369

We organize the rest of the Chapter as follows. Section 4.1 discusses scheduling tools available in the literature and the motivations for developing our tool. We present the architecture of our tool and its components in Section 4.2. Section 4.3 presents the user interface and trace file format. Finally, Section 4.4 concludes this Chapter by validating the proposed model.

4.1 Motivation and Related Works

Among the challenges we encountered when experimenting with the baseline platform was predicting the behavior of the scheduler. Studies in the literature discuss a variety of scheduler algorithms while presenting proof of their correctness, performance, and other guarantees. However, real-world systems have plenty of points of non-determinism and do not reflect some of the simplifications presented in the literature, e.g., scheduler time equals zero, absence of system interruptions, and absence of voluntary task preemption. [Li et al., 2019, Xia et al., 2021, Doan and Tanaka, 2019].

Before developing our tool, we searched the literature for other tools to simulate scheduling. Although we have found a couple of references to tools, we were unable to download any of the tools from the corresponding source. However, the associated studies could have hinted at the performance of the tools and the internal structures of their software implementation. We briefly discuss these studies as follows.

- [Yaashuwanth and Ramesh, 2010]: The authors propose an academic tool for teaching real-time scheduling, implementing a couple of scheduling algorithms, e.g., EDF

(earliest deadline first), LLF (least laxity first), RM (rate monotonic), and DM (deadline monotonic). The tool relies on a GUI (general user interface) environment focusing on teaching, with no reported automation capabilities, e.g., batch execution.

- [Casile et al., 1998]: The authors propose a tool for dealing with distributed real-time scheduling. They provide trace files as output, a GUI arrangement displaying simulation steps, and an uncoupled backend. However, they do not discuss simulation performance or the characterization of the task set.

- [RETIS Lab, 2013]: The RTSim tool is a simulator based on Metasim, a discrete-event simulation (DES) library. They provide a GUI tool for visualizing simulation traces called RTTracer. Unfortunately, the RTSim website is outdated (the last update was in 2011). The last public version of their tool was released in 2007. We could not find any documentation supporting the installation or extension of the tool.

- [Manacero et al., 2001, GSPD Lab, 2021]: RTsim (lowercase S) is another academic simulator. Their website reports that the simulator supports RM scheduling for single processors and other multiprocessor systems algorithms. However, we could not evaluate the simulator as the download link is broken.

Even if we could access the tools from their corresponding source, the tools, as reported in the corresponding reference, lack some key features that we consider to be requirements for developing other parts of our Thesis. For instance, performance is essential when collecting data for multiple system executions. The possibility of operating the command line is also desirable to enable automation in the project. Finally, exporting the results as trace files allows for data visualization. Table 4.1 compares the tools regarding these features, positioning the ORT/BENCH tool in the literature.

Table 4.1 – Comparison of scheduling simulators.

	Features* **			
Tool	A	B	C	D
[Yaashuwanth and Ramesh, 2010]				
[Casile et al., 1998]	+		+	
[RETIS Lab, 2013]	+		+	+
[Manacero et al., 2001]				
ORT/BENCH (this thesis)	+	+	+	+

(*) Features are the following: (A) command line execution, (B) performance analysis, (C) trace files, (D) allows for extension, e.g., new scheduling algorithms.

(**) The '+' denotes that the authors somehow claim their tool to have the said feature.

We first developed the ORT/BENCH tool in C++, aiming for performance and using the discrete-event simulation (DES) engine we proposed in a previous work [Domingues,

2020]. Later, we created another version of the tool in Python. We noticed that the performance of both tools was nearly the same. For this reason, we kept the Python version, developing a custom DES engine for the tool and ceasing the development of the C++ version. We believed that the Python version would be more accessible to students to modify and experiment with, and it would have a more straightforward setup.

During the development of the second tool, the complexity of extension points challenged us. We wanted to make the tool easily extendable so one could add more scheduling algorithms, modify the characterization of the task sets, and introduce interruption mechanisms. For this reason, it sounded reasonable to create a reference architecture (RA) [Taylor et al., 2007, Galster, 2015] instead of simply a tool. In practice, we created both as the ORT/BENCH tool follows the namesake reference architecture.

4.2 The ORT/BENCH Reference Architecture

A reference architecture (RA) is a collection of architectural design decisions and building blocks that targets a single application domain [Galster, 2015]. In other words, an RA groups the knowledge of several projects in the domain, serving as a reference for building new software. Since our RA aims to enable the development of new scheduling simulators, our goal is to define models and building blocks that can be reuse along multiple simulator projects. Due to the variety of scheduling algorithms, we limit the scope of our RA to the simulation of algorithms targeting mono-processed systems.

Our RA comprises five building blocks: (i) event model, (ii) system model, (iii) simulation model, (iv) scheduling algorithm, and (v) performance model. We split the event model further into the (a) task model and (b) interruption model. Both the interruption model and the performance model are optional. Figure 4.1 summarizes the building blocks of our RA.

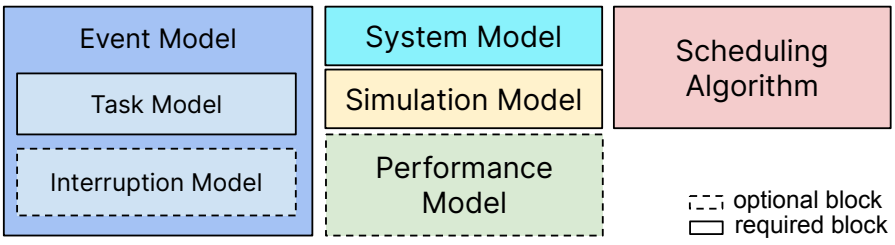


Figure 4.1 – Building blocks of the proposed reference architecture, comprising optional and mandatory blocks (adapted from [Domingues et al., 2021]).

The ORT/BENCH tool is an open-source scheduling simulator written in Python, developed following the principles of the namesake reference architecture. The tool was firstly distributed as a stand-alone application, and now it is part of the ORT suite, available

for download at https://github.com/andersondomingues/orca-rt-tools/. We dedicate the rest of the Chapter to explain the decisions made in during the development of the ORT/BENCH.

4.2.1 Event Model

In our RA, we use a DES engine to mimic the behavior of real-time schedulers. Thus, we describe the behavior of schedulers as events. In the context of our Thesis, we are interested in two types of events. First, we look into events triggered by tasks, e.g., system calls. The second kind of event relates to interruptions. A minimal scheduler must model at least one event, preferably to invoke a task scheduler, which we call *IRQ_SCHED*. In an actual kernel, the scheduler to elect the next executing task at each interruption, and the periodic calls to the scheduling give the system the possibility for multitasking. With that single interruption, we can trace the execution of most mono-processor scheduling algorithms. Figure 4.2 shows an example of a schedule trace for the EDF algorithm, where a *IRQ_SCHED* event occurs at each time unit. Tasks modeling follow the PCD (Period, Capacity, Deadline) model [Williams, 2006, Laplante and Ovaska, 2011, Berger, 2020, Kerrisk, 2021], where a task is $T =< P, C, D >$, where P is the task period, C is the capacity, and D is the task deadline.

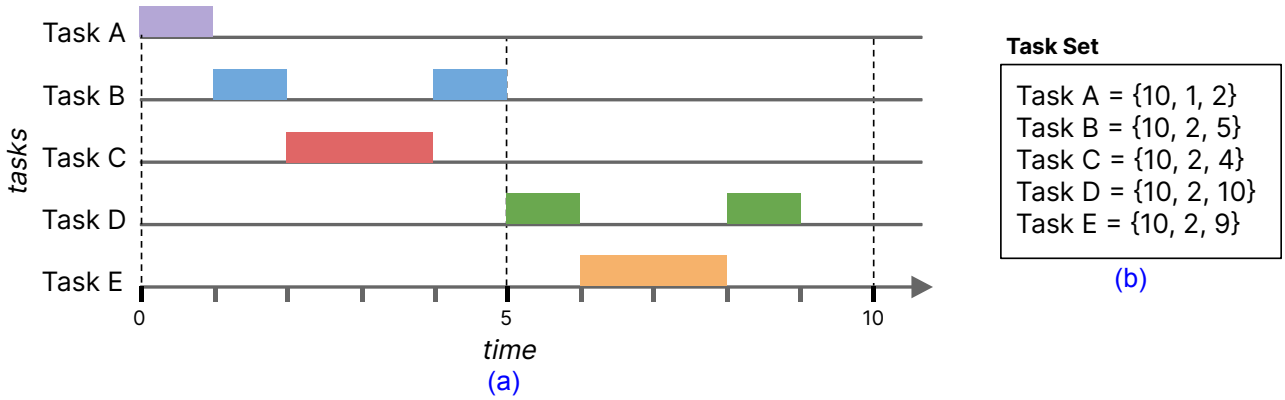


Figure 4.2 – Example of a simulation trace for the EDF algorithm (a) and the corresponding task set (b) (adapted from [Ali and Zakarya, 2020]).

Optionally, schedulers may implement voluntary task preemption. In this case, tasks can deliberately invoke the scheduler, giving up its execution time. We call this event *TASK_END*. Such an event can be helpful when tasks require a blocking resource (e.g., disk) or even during I/O (input and output) events. Figure 4.3 shows an example of an application using voluntary preemption. Please note that calling the scheduler in between two calls of the *IRQ_SCHED* event may require a more precise simulation time scale. Besides, voluntary task preemption allows for better CPU usage, although increasing the number of calls to the scheduler and overall scheduling time.

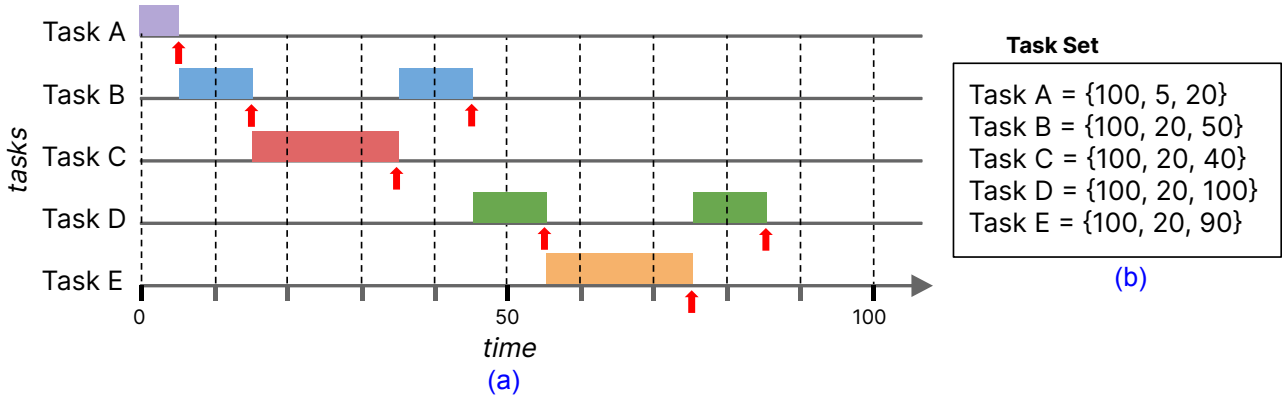


Figure 4.3 – Example of a simulation trace for the EDF algorithm with voluntary preemption.

Task Model

The task model corresponds to the representation of tasks within the simulator. For instance, a minimal task model would model the workload of the applications, i.e. the CPU time allocation of tasks. One may extend the model to admit other events, e.g., I/O simulation and system calls. In cases where resource-aware scheduling is necessary [Tillenius et al., 2015], one may add events for modifying counters to the model or even emulate sensor data.

We describe applications as sets of independent and periodic real-time tasks $T = \{t_0, ...t_n\}$ running in a mono-processed system. Each task is a 3-tuple $t_i =< p, c, d >$, where p is the period, c is the capacity, and d is the deadline, all them expressed in discrete time units (u). This model is a simplification over other models [Williams, 2006, Laplante and Ovaska, 2011, Berger, 2020], also inspired in the Linux kernel [Kerrisk, 2021].

The ORT/BENCH tool implements a parser in which applications are described as directed graphs $G =< V, E >$, where vertices represent tasks, and edges represent task communication. To vertices, we attach the parameters required by the task model, i.e. period, capacity, and deadline. Edges are unused in this Chapter, although we keep the graph structure for use in later Chapters. Also, some approaches may consider task dependency and real-time communication [Al-Kadi and Terechko, 2009]. Our parser also allows for more information to be added to the file without much effort. Figure 4.4 shows an example of an application description file.

```
1  [nodes]
2  #id task  capacity  deadline  period
3  01  T1    1         4         4
4  02  T2    2         5         5
5  03  T3    5         20        20
```

Figure 4.4 – Example of an application description file depicting an application taken from [Laplante and Ovaska, 2011] (adapted from [Domingues et al., 2021]).

Interruption

We implement the *IRQ_SCHED* interruption inside a execution looping. The user can inform the period of the interruption as a parameter. By default, the simulator calls the scheduler each 1 time unit, simulating the behavior of a system without voluntary task preemption. May the user define another period for the *IRQ_SCHED* interruption, voluntary task premption will be used whenever a task workload finishes before the scheduling period. The model can be extended to admit the simulation of other interruptions, peripheral I/O and watchdogs.

4.2.2 System Model and Scheduling Algorithms

The system model orchestrates the state of tasks within the system. One may implement the system model as a list of tasks in which a simplified task control block represents each task, mirroring the task abstractions of an actual real-time kernel. Another possibility is having individual lists, each storing tasks according to their state. If going for individual lists, one must consider that at least two lists are necessary, as the system must distinguish between running and idling tasks. Formally, the adopted model is a system of interconnected lists of tasks $Q = \{q_1, q_2, ..., q_n\}$, where $|Q| \geq 2$ (because the system must distinguish between running and idle tasks). One may add more lists if necessary (like we did in ORT/BENCH). A graph $G = V \times E$ represents the transition system (movement of tasks between lists), i.e., the set of edges E represents the possibilities for tasks to move from one list to another, and the set of vertices V represents the lists. Finally, a scheduling algorithm is a function $\Phi : Q_i \rightarrow Q_i$ that sorts the *ready* list, putting the next task to run at the top. After the sorting, the system exchanges the top tasks of the *ready* and *executing* lists. Figure 4.5 shows an example of a simulation model, depicting the trace in Figure 4.2.

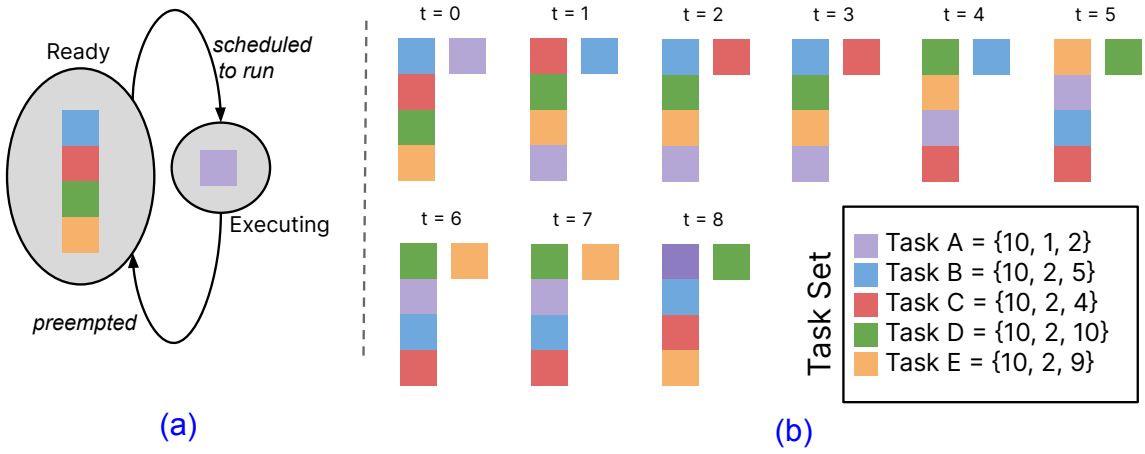


Figure 4.5 – An example of simulation model (a), and the corresponding simulation trace (b).

In the FreeRTOS kernel [FreeRTOS, 2024], tasks can assume one of the following four states: (i) suspended, (ii) blocked, (iii) running, and (iv) ready. For the Linux kernel [Babar, 2012], it uses four states: (i) runnable, (ii) interruptable sleeping, (iii) uninterruptable sleeping, (iv) zombie. In ORT/BENCH, we implements three lists: (i) running, (ii) ready, and (iii) blocked, as shown in Figure 4.6.

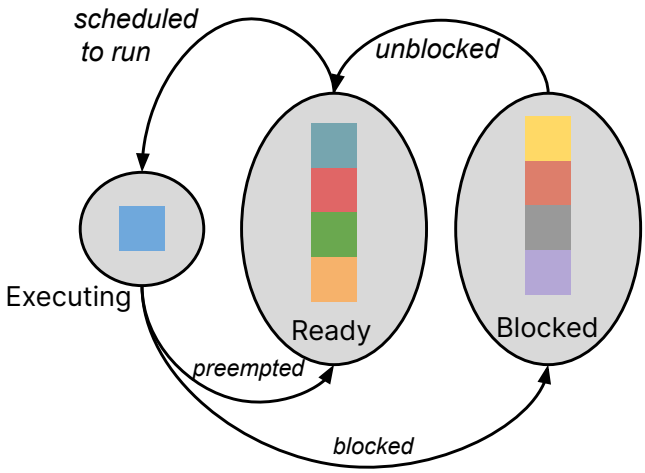


Figure 4.6 – System model implemented as an interconnected list system. Arrows indicate the direction in which tasks can move between lists. The sorting algorithm (scheduling) applies only to the ready list.

Our interconnected list system has four transitions: (i) scheduled, (ii) preempted, (iii) blocked, and (iv) unblocked:

- *Scheduled*: The running list stores the only executing task in the system, if any. Every time a task leaves the running list, another one (can be the same task) must take its place. The scheduling algorithm has to sort the ready list and move one task from the ready list to the running list. If no suitable task exists, the simulator introduces an idle task until it moves another task into the ready list.
- *Preempted*: A task moves from the running list to the ready list if its execution time window is over, i.e. *IRQ_SCHED* event. May the task finishes before the *IRQ_SCHED* event, it moves to the blocked list instead.
- *Blocked*: A task may block in two situations. First, when a resource blocks its execution, e.g. I/O. Second, when it finishes before the next period. In both cases, the task is moved into the blocked list.
- *Unblocked*: Tasks in the blocked list move to the ready list if they are within their release time window (period) and there is no resource blocking its execution.

4.2.3 Simulation Model

The simulation model is the method used by the simulator to generate the simulation trace. The simulation trace corresponds to the result of a simulation session, usually including the name of events and annotations indicating the time they would occur. The number of events that can occur in parallel is bound to the number of simulated CPUs; for mono-processor systems, two events cannot occur simultaneously.

One important aspect of the simulation model is its determinism. In some situations, events can occur simultaneously, even when considering a mono-processing environment. For instance, one interruption can occur at the same time as a task triggers the voluntary preemption mechanism. Consequently, the simulation model must adhere to an order of precedence that determines which event executes first in case of a conflict —— failing to resolve conflicts leads to non-deterministic behavior. A simulation model describes a function $F : s \rightarrow s', s \neq s'$, where s corresponds to the current simulation state and s' corresponds to the next simulation state.

In ORT/BENCH, the implemented simulation model relies on the discrete-event simulation [Fishman, 2001, Kofman et al., 2019, Domingues, 2020]. In this model, events are pushed into a list and sorted by their release time (priority queue). Events occur instantaneously, allowing the simulation to skip the remaining simulation time until the next event. For this reason, simulating a discrete-event system is indeed faster than executing any operating system kernel or scheduler. Figure 4.7 shows a comparison of a kernel with 10ms tick and a DES engine running tasks as events, assuming a performance of 1*ms* per event.

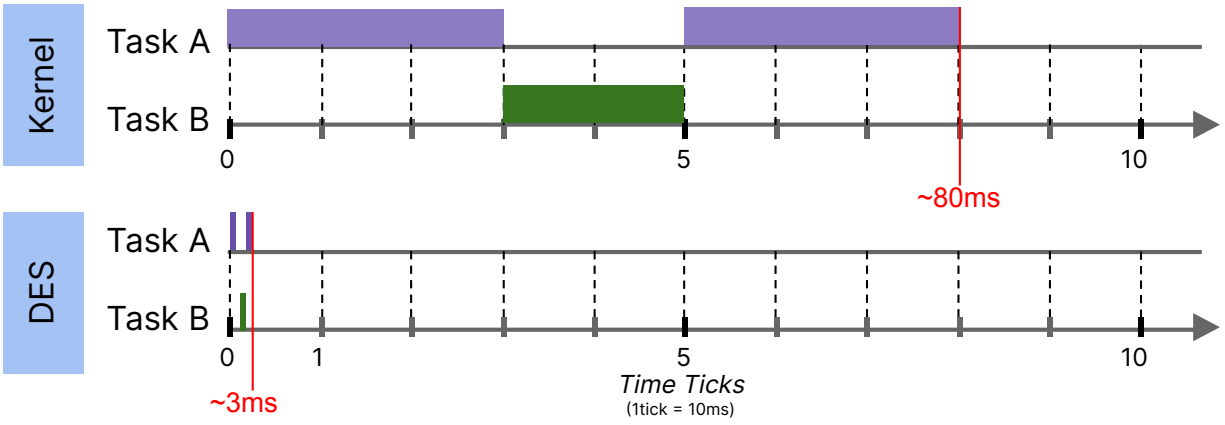


Figure 4.7 – Comparison between a kernel run and a DES engine simulating the same trace.

The simulation engine, a component of our simulator, implements the F function as an algorithm, which takes the current state of the simulation as input and updates that state, generating the next state. This operation repeats until the end of the simulation. We represent events as 3-tuples $e =< t, f, T >$, where t is the release time of that event, f is an activation function, and T is a period increment. For periodic events, T adds to t at each

successive release of e (period). The simulation function $f : Q \rightarrow Q$ corresponds to the movement of tasks in the system model, where Q is the set of all lists in that model. A simulation (Equation 4.1) is represented by a 5-tuple, where t_0 is the initial simulation time (usually equals to zero), n is the maximum simulation time, P is a priority-queue of events, E is the set of all events, and g is the computation function. The next state of the simulation can be achieved by applying g over the current state. Successive applications of g generate a trace of computation steps, ending when $t_m \geq n$, at the m^{th} step. We have applied a similar simulation model in our previous work [Domingues, 2020].

$$SIM =< t_0, n, P, E, g > \mid SIM' = g(SIM) \tag{4.1}$$

4.2.4 Performance Model

A performance model is a tool for assessing the simulation performance. This model must include the timing evaluation of events, as well as the scheduling algorithm. The goal of the performance model is to predict how much time the simulation will take to simulate a given task set. Formally, we describe a generic performance model as $P :< s', s'', F >\rightarrow \mathbb{N} \mid s' \neq s''$, where P maps the transition of states s' and s'', from a simulation model F to a number of time units.

Our performance model (Equation 4.2) can estimate the time taken to simulate a task set for a given hyper-period. Our estimation relies on the fact that we can predict the effort of the simulation model if we know the task set *a priori*. We must calculate the number of events to be simulated for that task set and multiply the resulting value for the effort of simulating a single event. In this case, our simulator must be deterministic, and the effort to simulate one event must be constant.

$$(O(Sort_n) + k) \times \left(\sum_{i}^{n} \frac{HP}{P_i} + \frac{HP}{time\ slice} \right) \tag{4.2}$$

Our simulator implements a routine that generates a trace of computation steps for a given initial simulation state. In that routine, the effort to simulate one event is constant (k), corresponding to moving the running task from the running list to another list and moving one task from the ready list to the running list. Before moving the one task back to the running list, the scheduling algorithm sorts the ready list. The cost of sorting the ready list is bound to the execution of the underlying sorting algorithm, which we denote $O(Sort_n)$, where n is the number of tasks in the ready list. A conservative approach would take n as the number of tasks in the whole system, which is always greater than the size of the ready list. In a few words, the effort of calling the scheduler once is given by $O(Sort_n) + k$.

The last component of our performance model corresponds to the number of simulated events, that is, the number of calls to the scheduler during the hyper-period (*HP*), which corresponds to the "*the smallest interval of time after which the periodic patterns of all tasks is repeated*" [Ripoll and Ballester-Ripoll, 2013]. The number of events must be at least $^{HP}/_{time\ slice}$, as the simulator calls the scheduler at the end of each time slice (`IRQ_SCHED`). Finally, we must consider the invocation of the scheduler at the end of each task, equals to $\sum_{i}^{n} {}^{HP}/_{P_i}$, where n is the number of tasks in the model and P_i is the period of the i^{th} task.

4.3 Other Features

Our tool implements the following scheduling algorithms: Deadline Monotonic (DM), Earliest Deadline First (EDF) [Liu et al., 2012], Least Laxity First (LLF), Least Slack Time (LST), and Rate Monotonic (RM) [Buttazzo, 2011, Laplante and Ovaska, 2011]. One may select the algorithm by entering the corresponding acronym of the algorithm as a parameter, e.g., `-EDF` selects the Earliest Deadline First algorithm. At the startup, the simulation engine selects the proper scheduling algorithms, that is, the algorithm sorting the ready list.

Due to performance reasons, our simulator runs in console mode, producing a *trace file* as output. The trace file stores a list of events produced during the simulation, indicating the time in which tasks enter and leave the running list and their deadline. Our trace file can serve as input for two visualization tools: KProfiler [Johann, 2024c] and ORB_KProfiller [Benno, 2024]. The former is a tool for visualizing system events for the HellfireOS operating system. Calling our tool with the `-kprofiler` parameter format the trace file to match the input of KProfiler. The latter, ORB_KProfiller, is a front end to our tool, capable of interactively generate information on the simulation, e.g., number of missed deadlines, schedulability tests, multiple simulation charts. Figure 4.8 shows an example of trace file. Figure 4.9 displays ORB_Kprofiller.

```
1  #id    #adl    #start   #end
2  1      30      1        2
3  1      30      2        3
4  1      30      3        4
5  1      30      4        5
6  2      35      5        6
```

Figure 4.8 – Trace file reporting the result of a simulation. The id field matches the one in the input file. Other fields are presented in discrete time units.

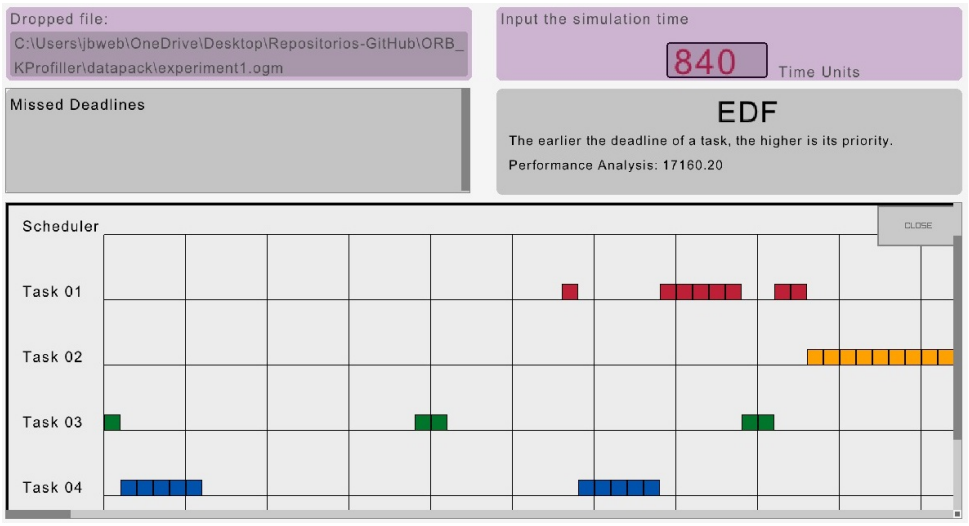


Figure 4.9 – ORB_Kprofiller interface depicting a simulation trace. Horizontal axis represents time, while vertical axis represents tasks. Performance estimation result (pink) and schedulability test (gray) are shown at the right-upper corner.

4.4 Performance Model Validation

This Section validates the performance model presented in Section 4.2.4. We simulated seven applications, 30 times each, collecting their execution time. Table 4.2 shows the characterization of applications, the simulated hyper-period, and their mean execution time. The hyper-period is given by $HP = lcm(T_p)$, where T_p is the set of all periods of all tasks in application T, and lcm is the least common multiplier function. We arbitrarily choose EDF as the scheduling algorithm for the experiment, and the value of $O(Sort_n)$ depends only on the number of tasks of each application. To eliminate noise during the experiment, we configured the simulator to collect the time tags (in milliseconds) in which the first and last events left the event queue for each run. Subtracting both tags give us the amount of time spent by the simulator to process the simulation events, ignoring file manipulation and startup routines in the process.

4.4.1 Results

As discussed in Section 4.2.4, the performance of our simulator is bound to the number of simulated events. When simulating shorter time slices, the number of calls to the scheduler (`IRQ_SCHED`) increases. Consequently, the number of events to be simulated increases, degrading the performance of the simulation. The worst performance is achieved when time slice is 1 time unit. For the experiment, we applied a normalized time slice of 1 time unit for all applications.

Table 4.2 – Characterization of applications A to G.

Application	Task Info.			Hyperperiod	Avg. Exec. Time (ms)
	Label	Period	Capacity		
A	T1	90	1	1260	3.406
	T2	4	2		
	T3	21	5		
B	T1	4	1	1540	8.438
	T2	14	2		
	T3	28	7		
	T4	10	1		
	T5	44	11		
C	T1	10	2	3600	11.689
	T2	12	2		
	T3	16	2		
	T4	18	2		
	T5	20	2		
	T6	200	2		
D	T1	90	10	6300	42.372
	T2	60	12		
	T3	105	19		
	T4	50	25		
	T5	150	5		
E	T1	30	5	50400	218.885
	T2	35	9		
	T3	45	15		
	T4	100	10		
	T5	800	40		
F	T1	24	8	840	3.737
	T2	30	10		
	T3	7	2		
G	T1	64	8	960	4.192
	T2	80	10		
	T3	20	2		
	T4	30	5		
	T5	60	20		

Execution time increases linearly to the number of events. This is true as long the number of tasks remains the same. The time taken to simulate a single event depends only on the number of tasks in the ready list. We observe that the ratio $\frac{ET}{HP}$ grows linearly on the number of events, where ET is the execution time of the simulation. Figures 4.10 and 4.11 show the approximation of collected data to a power series and linear functions, respectively. The average time to simulate one event roughly approximates 0.00448ms. Please note that this time is bound to the performance of the machine in which the experiments were executed, as well as the configuration of the installed tool-chain and compilation scripts.

The ratio $\frac{ET}{e}$, where e is the number of simulated events, presents the same linear behavior as the $\frac{ET}{HP}$ ratio. In this case, for a constant number of events, the time to simulate an application grows linearly to the number of tasks. This is true as long as the time slice is $1u$ and $HP = e$. We omit the charts as they would be trivially similar to the ones shown in Figures 4.10 and 4.11.

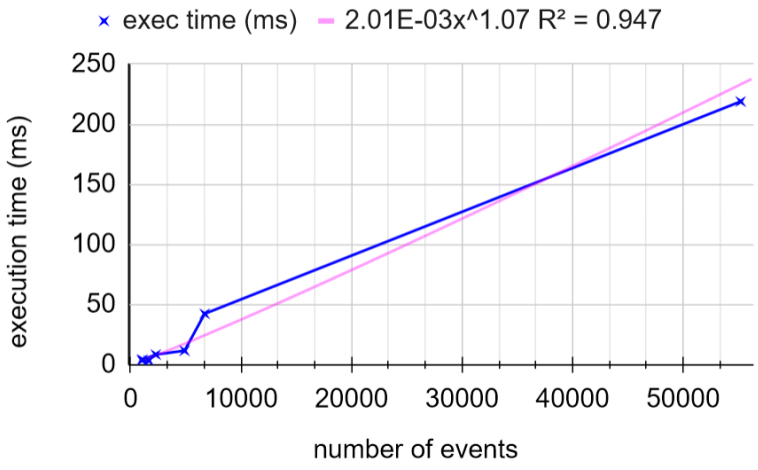


Figure 4.10 – Approximation of the collected data to a power series function. Points are sorted by execution time (ascending).

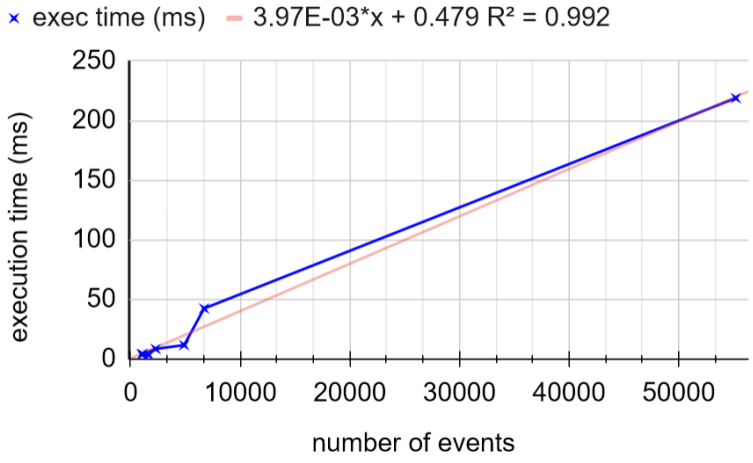


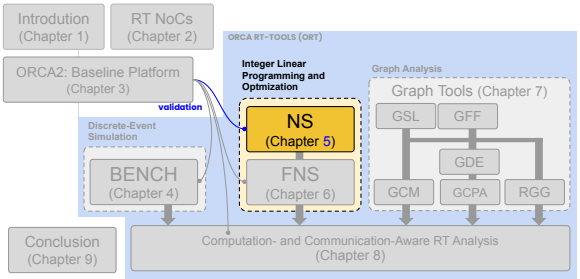
Figure 4.11 – Approximation of the collected data to a linear function. Points are sorted by execution time (ascending).

4.5 Final Remarks

This chapter presented ORT/BENCH, a tool for predicting the behavior of task schedulers. The main contribution of the chapter is to provide a method for collecting data from the scheduler without having to execute a whole system setup. The tool surpasses the performance of actual kernels due to the application of discrete-event simulation, providing a faster way of collecting results. Besides, our reference architecture introduces a novel contribution to the Thesis, allowing practitioners to create their own simulator based on our building blocks. We employ the ORT/BENCH tool in the approach discussed in Chapter 8. Finally, as a feature work, we intend to extend the simulator to adhere to other system parameters, e.g., voluntary task preemption. Besides, we want to create a more comprehensive user interface to allow using the tool in undergraduate courses.

5. ORT/NS: DESIGN-TIME ANALYSIS OF REAL-TIME TRAFFIC IN NOCS USING CONSTRAINT MODELS

This Chapter presents our first effort at modeling network traffic using constraint models to generate valid traffic schedules. As a result, we developed the ORT/NS tool, an offline network scheduler. The following publication relates to the tool:



> Design-time analysis of real-time traffic for networks-on-chip using constraint models
> Domingues, A. R. P; Johann Filho, S., Amory, A. D. M., and Moraes, F. G.
> 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI). 2022.
> https://ieeexplore.ieee.org/document/9893222

We organize the rest of the Chapter as follows. Section 5.1 briefly presents the *-shop problems we used to model network traffic. We present Net-Shop, our variant of the *-shop problem and network model, in Section 5.2. We developed a tool named ORT/NS (network scheduler) to automatize the execution of the Net-Shop model. We present the architecture of the ORT/NS tool in Section 5.3. We demonstrate the application of the ORT/NS tool over a synthetic application in Section 5.4. Finally, we discuss the research outlook on the ORT/NS tool in Section 5.5.

5.1 The *-Shop Problems

We name *-shop the family of problems that comprise the job shop, flow shop, and permutation shop problems, as well as many other variants in the literature. These problems have many applications, including manufacturing processes (process engineering) and task scheduling (computing). Regardless of the variant, such problems involve allocating jobs to resources, often called machines. Jobs represent sets of self-contained, smaller parts called tasks; each task has its execution parameters, e.g., minimum and maximum execution time. Also, tasks can be bound to specific machines, i.e., require particular machine features for execution. Other characteristics of tasks include task dependency or even a requirement for finishing before a specific due time. Although our goal is not to discuss *-shop comprehensively, we advise one to see Garey and Johnson [Michael R. Garey, 1979] for a comprehensive description of some *-shop variants and their proof for NP-completeness.

Formally, a *-shop problem $P =< M, J, T, \phi, \psi >$ is a 5-tuple where M is the set of machines, J is the set of jobs, ϕ is the cost function (schedule length), T is the discrete time domain, and ψ is the set of constraints in the problem (variant specific). The solution space, i.e., the set of all possible solutions, represents the relation $S : T \times M \times J$. Any solution $s \in S$ indicates the time and machine that each job must be released to minimize C. The difference between two arbitrary variants of the *-shop problem usually lies in the definition of constraint set ψ and rarely relates to the structure of the problem.

5.2 Net-Shop: Network Scheduling as a *-Shop Problem

To model network scheduling as a *-shop problem, we need to adjust the *-shop problem structure. First, we rename machines to “links” and jobs to “packets.” Since we are interested in NoCs, we must also rename tasks to “flits.” We formally describe our variant of the *-shop problem as: “Given a set of packets (J) consisting of a set of flits each, find an allocation ($s \in S$) of packets for links (M) such that all flits traverse all the links in accordance to their route, in order.” Of course, we must define the route of packets *a priori*, and we assume the XY routing algorithm for the rest of the Chapter. For readability, we name our problem variant the “Net-Shop” problem, formally $P =< M, J, T, \phi, \psi >$. We detail the parts of the problem as follows:

- M is the set of *network links* (previously, machines). For readability, we name links after their connecting routers, e.g., link `1-3` departs from router 1 and arrives at router 3. This naming convention is topologic agnostic and one may change it as necessary. Please note that links `1-3` and `3-1` are not the same link as they transfer data in opposite directions. The input and output links are named on the local port, e.g., `L-5` links the local port of the 5^{th} PE and router 5. Similarly, link `5-L` connects router 5 to the local port of the 5^{th} PE in the opposite direction. Figure 5.1 shows a 2D-mesh NoC topology, and the name given to its links.

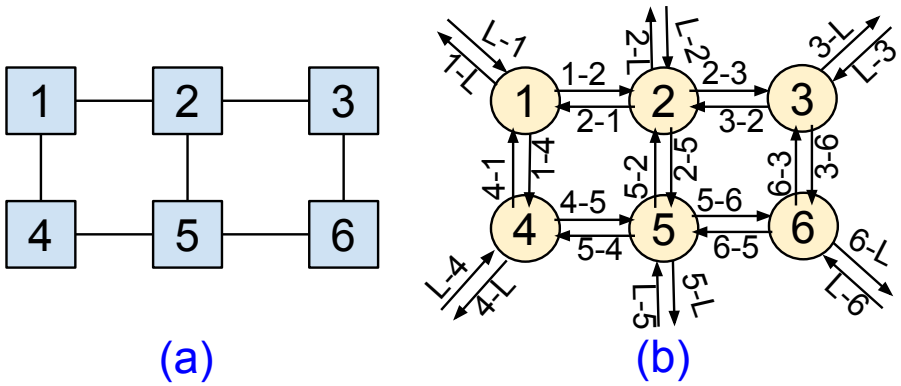


Figure 5.1 – Example of a NoC (a) and the names of links in the *-shop problem (b).

- J is the set of *packets* (previously, jobs). A packet has a sequence of flits, which can occupy any link of the network. However, since packets have source and destination nodes, the underlying routing algorithm limits the set of links that packets can traverse. In our problem variant, two flits cannot occupy the same link at the same time.
- T is the discrete time domain. We measure time in cycles instead of taking seconds or milliseconds as units. By doing so, we eliminate any dependency on the frequency of the target system.
- We measure solutions efficiency by ranking them by their schedule length, often called makespan, where the lower the value, the better. Let $\phi = C : s \in S \rightarrow \mathbb{Z}^+$ be a cost function that measures the schedule length of a single solution (in time units). The best solution is the one that matches $min(C_s)$.
- S is the solution space, and $s \in S$ is an $M \times T$ relation denoting which flit (or packet) occupies each link. An instance of the problem has no solution if $S = \emptyset$.
- ψ is the set of constraints on the problem.
    1. For any packet, all links in its path remain occupied for the same time during the traversal. Thus, $\forall(t_i, t_j \in j_k).(p(t_i) = p(t_j))$, where p_i is the capacity of each task. This is equivalent to the processing time in *-shop problems.
    2. We set a constraint on task release time $r(t_i)$ such that $\forall(t_i, t_j \in j_k).(r(t_i) = r(t_j))$, indicating that packets will allocate all links in their path, from the source router to their destination. This simplification slightly degrades the performance of the network, as it keeps links allocated until the last flit arrives at its destination. In practice, the maximum delay introduced is not greater than the dimension of the number of hops of the packet. We claim this delay to be negligible for a wormwhole NoC as the number of cycles to push packets from one router to another approximates 1 cycle as the length of the packet increases.

Equation 5.1 shows an example (K) and its components. The M comprises all network links for a 2x2 dimension NoC, and the J set has all the packets in the network. Packets carry information on which links they need to occupy during traversal. Figure 5.2 shows an illustration of the problem in a more intuitive drawing.

$$K = \begin{cases} M = & \{0\text{-}1, 1\text{-}0, 1\text{-}3, 3\text{-}1, 2\text{-}3, 3\text{-}2, 0\text{-}2, 2\text{-}0, L\text{-}0, 0\text{-}L, L\text{-}1, 1\text{-}L, L\text{-}2, 2\text{-}L, L\text{-}3, 3\text{-}L\} \\ J = & \{j_1 = \{L\text{-}0, 0\text{-}1, 1\text{-}L\}, \\ & j_2 = \{L\text{-}0, 0\text{-}1, 1\text{-}3, 3\text{-}L\}, \\ & j_3 = \{L\text{-}2, 2\text{-}3, 3\text{-}L\}, \\ & j_4 = \{L\text{-}2, 2\text{-}3, 3\text{-}1, 1\text{-}L\}, \\ & j_5 = \{L\text{-}3, 3\text{-}2, 2\text{-}0, 0\text{-}L\}\} \end{cases} \tag{5.1}$$

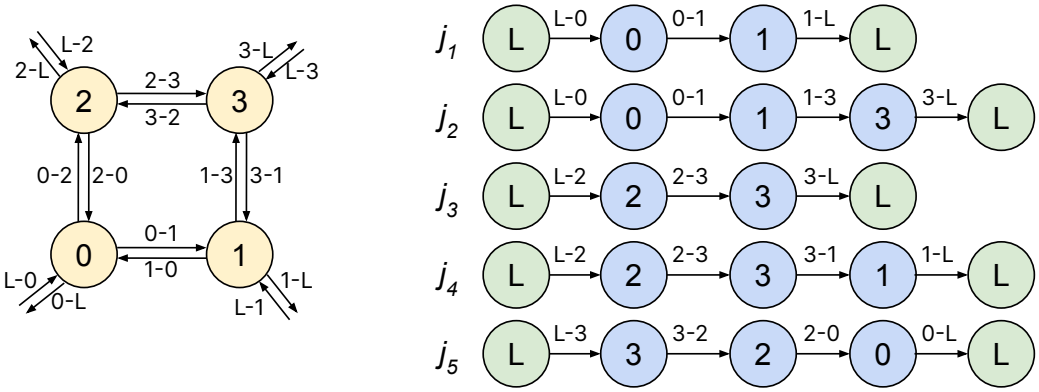


Figure 5.2 – Links of a 2x2 NoC instance (left), and an instance of Net-Shop (right).

5.3 Scheduling packets with the ORT/NS tool

Time Division Multiplexing (TDM) is an approach for guaranteeing real-time traffic in NoCs [Picornell et al., 2019, Picornell et al., 2020]. The TDM approach relies on the analysis of *conflicting packets* in the network. Two packets conflict if they share resources, getting separated into different TDM slots. Although we assume NoCs without virtual channels, we borrow the concept of *conflicting packets*, which we generalize to what we name *occupancy*. Occupancy corresponds to a relation $O : J \times M \times T$. We propose an approach in which a resource conflict corresponds to simultaneously using a single link by at least two different packets. Our approach aims to prevent conflicts by assigning packets to resources by controlling the injection time of packets in the network. The result of the approach is a table of injection time, to be used within the source of the packet (e.g., TCNI, Chapter 3). Figure 5.3 shows the five steps of our approach: (i) flow characterization, (ii) flow unwrapping, (iii) network delay analysis, (iv) schedule generation, and (v) configuration.

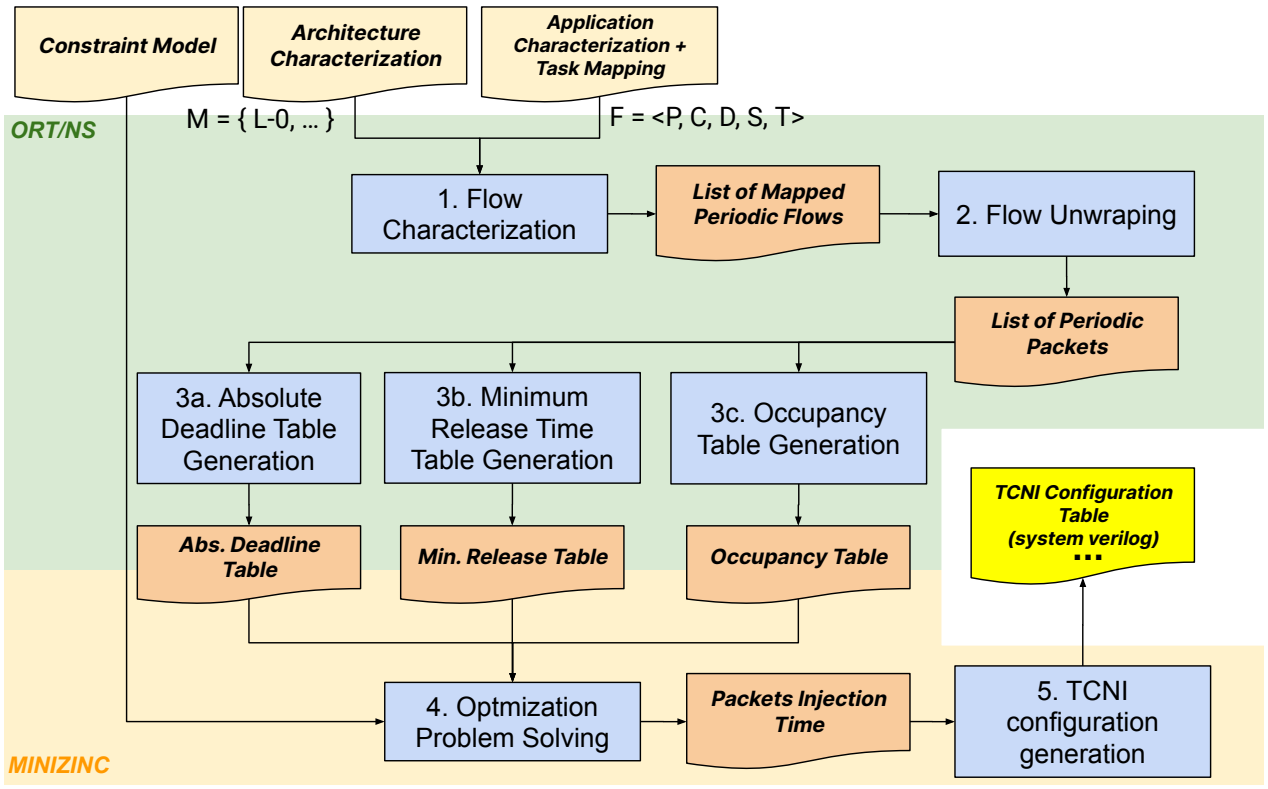


Figure 5.3 – The steps of our method (blue rectangles), their inputs and outputs (white shapes) and the tools that automatizes the steps (represented by different background colors). Arrows connect inputs to the corresponding steps. User inputs are represented by the yellow shapes at the top of the image.

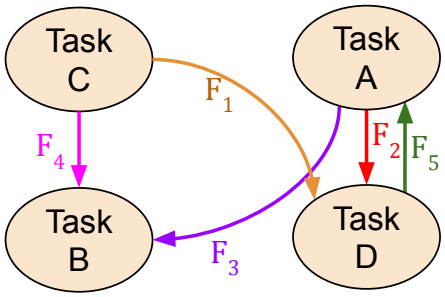
5.3.1 Flow Characterization

In our approach, a flow is the representation of an infinite set of packets, whose model is a 5-tuple $f =< p, c, d, s, t >$, where p is the flow period, c is the amount of data to traverse the network for each packet in that flow (worst-case assumption constant), d is the *relative* deadline, and s and t are the source and destination tasks for that flow. In practice, the flow set $F = \{f_0, f_1, ..., f_n\}$ is an extension of the J set, carrying now information about the periodic behavior of flows. Period and deadline can be represented in discrete time units, although one can easily assume any other time-measure unit (we use *cycles* for the rest of the Chapter). The same applies to data size (we use *bits* for the rest of the Chapter). We label both source and target tasks according to their position in the topology, so our approach requires task mapping to be performed *a priori*.

Our analysis extends to the hyperperiod, as we are interested in the periodic execution of tasks. The hyperperiod corresponds to "the smallest interval of time after which the periodic patterns of all tasks repeats" [Ripoll and Ballester-Ripoll, 2013]. We borrow this definition while replacing the word "tasks" with "flows." In our context, the optimal hyperperiod is the least common multiplier among the periods of all flows, i.e., $H = lcm(F_p)$. As the *lcm*

function tends to “misbehave” for larger sets, we suggest practitioners manipulate packets period to reduce the value of H as much as possible, e.g., rounding periods to the critical path or artificially increasing messages payload. Finally, we assume that the network state resets at the end of the hyperperiod, so any chances for error propagation by accumulated delay are eliminated.

We express traffic characterization using a cyclic direct graph (digraph). Also, we label edges on flow data and nodes on tasks, as Figure 5.4 shows. Information on their period, capacity (worst-case data size), and deadline must be present for each flow.



(a)

Flow	Period	Cp.	Deadline
F_1	55	160	55
F_2	55	416	55
F_3	55	192	55
F_4	55	256	55
F_5	55	128	55

(b)

Figure 5.4 – Example of application characterization using a digraph. The application *synthetic-flow-A* comprises four tasks, labeled from A to D and represented by nodes in the graph. Arrows indicate flows (a). In this application, flow have their period equals to their relative deadline (b), although this is not a limitation of our approach.

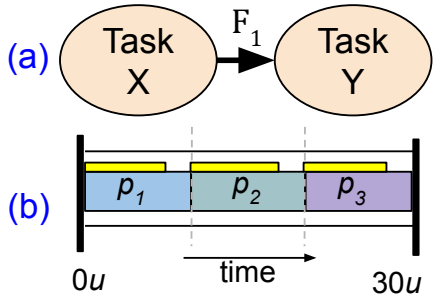
5.3.2 Flow Unwrapping

The set of all packets injected into the network during the hyperperiod is given by P. Equation 5.2 determines $|P|$, where n is the number of flows and f_i^p is the period of each flow $f \in F$. Each flow unwraps to generate a finite number of packets for the given hyperperiod. Packets originating from the same flow have the same data size (worst case data size), although they necessarily differ in their *minimum release time* (MRT) and *absolute deadline* (AD). A packet $p \in P$ is given by $p =< f \in F, r, a >$, where f is the corresponding flow, r is the minimum release time, and a is the absolute deadline. By unwrapping all flows, we achieve the set of packets $P = \{p_0, p_1, ..., p_k\}, k = |P| - 1$.

$$|P| = \sum_{i=0}^{n} \frac{f_i^p}{H} \tag{5.2}$$

For example, one flow with a period equal to 10 time units (u) would generate three packets for a hyperperiod of 30u. The first packet enters the network at the startup (time = zero), i.e., the minimum release time (p^r) for that packet is zero. The second packet enters the network after 10u, and the last packet enters the network after 20u. We assume packets

are evenly distributed within the hyperperiod. Their absolute deadline is given by $f^d + p^r$, their minimum release time, plus the relative deadline of the corresponding flow. Thus, the deadline of packets p_1, p_2, and p_3 would be 10u, 20u and 30u, respectively. Figure 5.5 presents an example of unwrapped flow.



Flow	Period	Capacity	R. DL
F_1	10	8	10

Packet	Min. RI.	Capacity	A. DL
p_1	0	8	10
p_2	10	8	20
p_3	20	8	30

(c)

Figure 5.5 – Example application and flow unwrapping for a hyperperiod of 30u (a). Three packets were generated: p_1, p_2, and p_3 (c). Packets are evenly distributed in time (b).

It is worth highlighting that flow unwrapping is automated by the NS tool. The result of the flow unwrapping step is a list of packets. Each packet is P = <MRT, CP, AD>, where *MRT* is the minimum release time, *CP* is the capacity, and *AD* is the absolute deadline. Please note that *CP* is the same as the capacity of the corresponding flow.

5.3.3 Table Generation

In this step, the process crosses the requirements of packets with the available system resources (R). The result of this step is three tables: (i) absolute deadline, (ii) minimum start, and (iii) capacity. Each table has one dimension representing the network resources (links) and another representing the corresponding packet attribute.

- *Occupancy* represents the relation $O : R \times P \rightarrow \mathbb{N}$, where R is the set of resources (links) and P is the set of all packets. The relation O represents the time each packet requires from each resource during transmission. We calculate occupancy on the worst-case transmission time (WCTT) of the target NoC.
- The *minimum release time* represents the relation $M : R \times P \rightarrow \mathbb{N}$, where A is the time each resource can be allocated to a packet. This table corresponds to the occurrence of periods in the timeline.
- *Deadline*: represents the relation $D : R \times P \rightarrow \mathbb{N}$, where D is the upper-bound time limit for a packet to use a link. If one packet keeps any link allocated further than the intended, the application may fail due to starvation.

5.3.4 Optimization

This step determines the time allocated to resources for each packet traversing the network during the hyperperiod. For example, each resource $r \in R$ can represent a link or router. In this Thesis, we consider only links due to the capabilities of the target NoC architecture [Moraes et al., 2004]. The optimization step aims to determine a combination of time spans such that no packet misses any deadline. The following constraints describe this rationale:

(C1) *Packet release time must be greater or equals to their minimum release time* (5.3). Since packets (P) are evenly distributed in the hyperperiod, packets cannot enter the network until the last packet of that flow reaches its destination. Please note that our approach treats resources (R) individually, and one packet may require more than one resource simultaneously. For this reason, we represent `min_release` as a matrix.

$$\forall(r \in R, p \in P)(release[r,p] \geq min\_release[r,p]) \tag{5.3}$$

(C2) *Two packets cannot share the same resource simultaneously* (5.4). We call this constraint the "non overleap constraint," as it enforces resources to be allocated only to a single packet. We write this constraint as a *predicate* (5.5) on the release time and occupancy of packets.

$$\forall(p1, p2 \in P).(nonoverlap(p1, p2)) \tag{5.4}$$

$$\begin{aligned} &nonoverlap(p_a, p_b \in P) = \forall(r \in R)(\\ &release[r,p_a] + occupancy[r,p_a] \leq release[r,p_b] \vee \\ &release[r,p_b] + occupancy[r,p_b] \leq release[r,p_a]) \end{aligned} \tag{5.5}$$

(C3) *Each packet must meet its deadline* (5.6). Since we treat resources individually, each pair of a packet and a resource (p, r) has its deadline and must be released before the allocation of that resource by other packets.

$$\forall(r \in R, p \in P)(release[r,p] + occupancy[r,p] \leq deadline[r,p]) \tag{5.6}$$

(C4) *A packet must allocate all the required resources for its transmission during the whole transmission (optional).* This constraint forks our analysis into two possibilities. First, we create a strict network model with information on each link. This model will generate

no waste of network bandwidth, although it can be tricky to develop depending on the target NoC. The second possibility is to design a heuristic constraint, accepting some waste of network bandwidth. In the former case, the constraint carries out the complexity of the network, and the allocation time for all packets is only the necessary time for the packet to traverse without congestion. In the second case (5.7), we assume that links allocate all the necessary links simultaneously, roughly corresponding to the circuit-switching situation (even for a wormhole network). For simplicity, we assume the latter case for the rest of the Chapter.

$$\forall(r_a, r_b \in R, p \in P)(release[r_a, p] = release[r_b, p]) \tag{5.7}$$

We state the Net-Shop problem as "*Given the relations occupancy, minimum release time, and deadline over* $P \times R$*, determine a relation* $S : P \times R \rightarrow T$ *that satisfies* $C1$*,* $C2$*,* $C3$*, and* $C4$*.*" Since occupancy gives us the time each packet requires from each of the links in the subject system, solving the problem involves optimization. However, we cannot compare two solutions regarding their quality at the time, so we take that any solution is valid. In other words, we approach our optimization problem as a decision problem, although still NP-complete. The solver finds multiple solutions in most cases, and the set of all valid solutions is not countable, although finite.

5.4 Proof of Concept and Discussion

We evaluate our approach for two applications targeting our baseline platform (NoC only). The goal is to find when packets must be injected into the network so they will not collide. Equation 5.8 presents the WCTT model of the Hermes [Moraes et al., 2004] NoC. The network protocol adds two extra flits to the payload (p), where the first flit (header) takes up to $7m$ cycles to route, with m corresponding to the Manhattan distance between source and target routers (5.9). Flits from the 3^{rd} and beyond take one cycle to perform a hop.

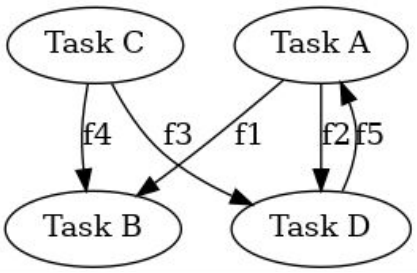
$$WCTT(cycles) = 7m + p + 1 \tag{5.8}$$

$$m = |x_b - x_a| + |y_b - y_a| \tag{5.9}$$

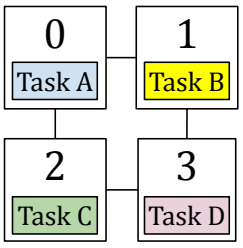
5.4.1 Application A — Synthetic Flow B

We use the *Synthetic-Flow-B* application to demonstrate the basic mechanics of our approach. Figure 5.6 shows the application characterization and task mapping for this

application for a 2*x*2 NoC instance. Please note that our analysis is sensitive to task mapping performed before characterization. Discussing task mapping is out of the scope of this proposal as task mapping is a well-established field of study.



(a)



(b)

Flow	P	Cp	D
F1	608	160	608
F2	608	416	608
F3	608	192	608
F4	608	256	608
F5	608	128	608

(c)

Figure 5.6 – The Synthetic-flow-B application task graph depicting flows interaction (a), the tasks mapped onto a 2 × 2 NoC (b), and the characterization of flows (c).

With the aid of Minizinc [MiniZinc Team, 2024], we developed a constraint model description using Minizinc language. The inputs for the model consist of three tables (occupancy, minimum start time, and deadline), as summarized in Table 5.1. We removed unused links from the tables to increase the performance of the solver. The tables will likely represent a sparse matrix as the number of links traversed by flows is less than the total number of links in the network. In this application, flows F_i unwrap to a single packet P_i each.

The configuration of the injectors resets at the beginning of each hyperperiod. For this reason, the deadline of packets must be less than the hyperperiod, guaranteeing that the state of the network will reset when packets reach their destination. The hyperperiod for the application is 608u (cycles), and all tasks share the same period.

Table 5.2 shows the results generated by the Minizinc solver (Gecode 6.3). Occupation and minimum start values for all links in the same packet are the same, and the results showed that all resources (links) necessary for each packet were allocated simultaneously and at the right time. Although the target NoC does not implement a mechanism for allocating links, the packets enter the network at the time indicated in the results for the first link in the path, as there is no congestion in the network. The non-overleap constraint in the model guarantees congestion avoidance. Thus, the path is accessible for packets to traverse the network during the allocation time.

As one can observe from Table 5.1, the solver suggests that packets P1 and P3 must enter the network at cycle 416. Packet P_5 enters the network at cycle 256, and other packets enter the network at cycle zero. Flows that conflict enter the network one after another, i.e., packets that share at least one link. In Figure 5.6 (a), we observe that flow F_3 conflicts with flows F_2 (on link `3-L`) and F_4 (on links `L-2` and `2-3`). The same stands for flow F_1, which also conflicts with flows F_2 (links `L-0` and `0-1`) and F_4 (on link `1-L`). One possibility (the one generated by Minizinc) consists of injecting P_2 and P_4 before P_1, P_3, and P_5.

Table 5.1 – Synthetic-flow-B characterization.

	Pck.	Links											
		L-0	0-1	1-L	1-3	3-L	L-2	2-3	3-1	L-3	3-2	2-0	0-L
occupancy	P_1	160	160	160	–	–	–	–	–	–	–	–	–
	P_2	416	416	–	416	416	–	–	–	–	–	–	–
	P_3	–	–	–	–	192	192	192	–	–	–	–	–
	P_4	–	–	256	–	–	256	256	256	–	–	–	–
	P_5	–	–	–	–	–	–	–	–	128	128	128	128
min. start	P_1	0	0	0	–	–	–	–	–	–	–	–	–
	P_2	0	0	–	0	0	–	–	–	–	–	–	–
	P_3	–	–	–	–	0	0	0	–	–	–	–	–
	P_4	–	–	0	–	–	0	0	0	–	–	–	–
	P_5	–	–	–	–	–	–	–	–	0	0	0	0
deadline	P_1	608	608	608	–	–	–	–	–	–	–	–	–
	P_2	608	608	–	608	608	–	–	–	–	–	–	–
	P_3	–	–	–	–	608	608	608	–	–	–	–	–
	P_4	–	–	608	–	–	608	608	608	–	–	–	–
	P_5	–	–	–	–	–	–	–	–	608	608	608	608

Note A: All units represented in cycles; **Note B**: Dash lines represent sparse matrix emptiness.

Table 5.2 – Optimization results for Synthetic-flow-B

	Pck.	Links											
		L-0	0-1	1-L	1-3	3-L	L-2	2-3	3-1	L-3	3-2	2-0	0-L
injection time	P_1	416	416	416	–	–	–	–	–	–	–	–	–
	P_2	0	0	–	0	0	–	–	–	–	–	–	–
	P_3	–	–	–	–	416	416	416	–	–	–	–	–
	P_4	–	–	0	–	–	0	0	0	–	–	–	–
	P_5	–	–	–	–	–	–	–	–	256	256	256	256

Note A: All units represented in cycles; **Note B**: Dash lines represent sparse matrix emptiness.

We configure behavioral injectors (TCNI, Chapter 3) with the results generated by Minizinc (Table 5.2), delaying the injection of packets in the network. We designed our project to output the time in which packets enter and leave routers. Then, we simulated the behavioral design for 608 cycles (the hyperperiod) using the ModelSim tool, achieving the results shown in Figure 5.7. From the simulation, we can observe that packets did not overlap or arrive at their destination after the corresponding deadline. However, as expected, packets are assumed to have all the links allocated along their path.

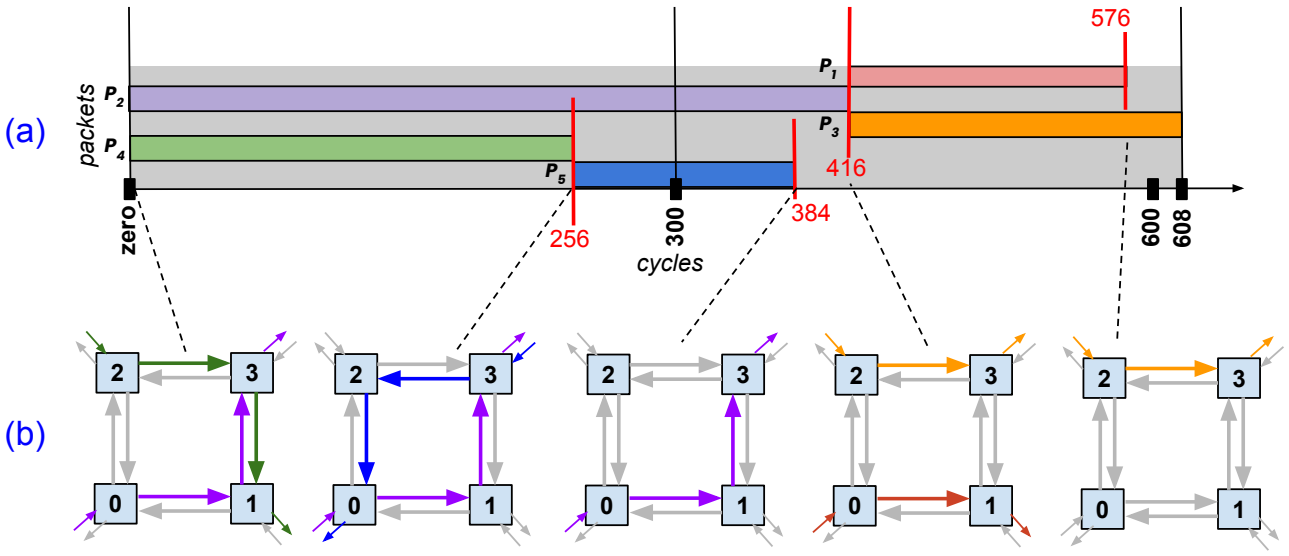
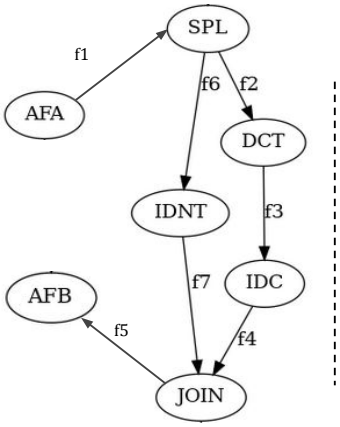


Figure 5.7 – Results of the simulation with the Synthetic-Flow-B application, depicting the spam lifetime of packets (a) and the usage of network links (b).

5.4.2 Application B — DCT-verify

We reproduced the same steps for Application A to characterize Application B. Figure 5.8 shows the application graph (a) and the corresponding task mapping (b). We applied our approach to the DCT-verify application (Discrete Cosine Transform), taken from the STR2RTS benchmark [Rouxel and Puaut, 2017]. Unlike Application A, this application is a workflow application with more strict deadlines for the tasks — tasks depend on the previous task in the workflow. Figure 5.8 (c) shows the characterization of Application B.



Flow	P	Cp	D
F1	78651	4096	36627
F2	78651	4096	37346
F3	78651	4096	53521
F4	78651	64	69632
F5	78651	8192	78607
F6	78651	4096	53521
F7	78651	4096	70253

(c)

Figure 5.8 – The DCT-Verify task graph (a), the corresponding mapping to 2x3 NoC (b), and its flows characterization (c).

Table 5.3 shows the optimization result gathered from Minizinc. We observed that the results took more time (0.523*s*) to generate compared to Application A (0.013*s*). We

expected this behavior due to the size of the problem, i.e., more links. It is important to note that Minizinc will keep running until it finds a solution (table) or a counter-proof of the model. For this reason, we may take any output result as a valid schedule.

Figure 5.9 shows the results of the RTL simulation collected from Modelsim. We configured the injector following Table 5.3, simulating the design for 78607 cycles (the hyperperiod). Once again, packets do not overlap, and their deadlines are less or equal to the one in the characterization. It is important to note that packets P_4 and P_5 could overleap as we did not restrict the routers regarding sending and receiving packets in parallel. However, we could, for instance, model local links (e.g., `L-2` and `2-L`) as the same link, preventing them from having two packets allocated at the same time.

Table 5.3 – Optimization results for DCT-Verify

injection time	Pkts.	Links									
		L-0	0-1	1-2	2-L	L-3	3-2	L-2	2-4	4-L	L-4
	P_1	29154	29154	29154	29154	–	–	–	–	–	–
	P_2	–	–	–	33250	33250	33250	–	–	–	–
	P_3	–	–	–	–	–	–	49425	49425	49425	–
	P_4	–	–	–	–	–	–	–	–	–	69568
	P_6	–	–	–	–	49425	–	–	–	–	–
		4-5	L-5	5-3	3-L	3-1	1-L	L-1	1-3	3-5	5-L
	P_4	69568	–	–	–	–	–	–	–	–	69568
	P_5	–	70415	70415	70415	–	–	–	–	–	–
	P_6	–	–	–	–	49425	49425	–	–	–	–
	P_7	–	–	–	–	–	–	65472	65472	65472	65472

Note A: All units represented in cycles.
Note B: Dash lines represent sparse matrix emptiness.

5.5 Final Remarks

This Chapter presented an novel approach for scheduling packets using a constraint model and the Minizinc solver. Our tool ORT/NS implements five steps to transform user inputs into a table of injection times. For the discussed applications, we achieved results in hundreds of milliseconds. Still, we did not apply the approach to larger hyperperiod (e.g., 70,000 cycles) values due to the extensive solving time; the solver took more than 48 hours to compute the injection times of the Car Application [Shi et al., 2010] (Intel® Xeon® CPU E3-1220 v5 @3.00GHz, 32GB memory). One drawback of our tool is the need for a synchronization mechanism; we suggested the TCNI module (Section 3.1.5) as a solution.

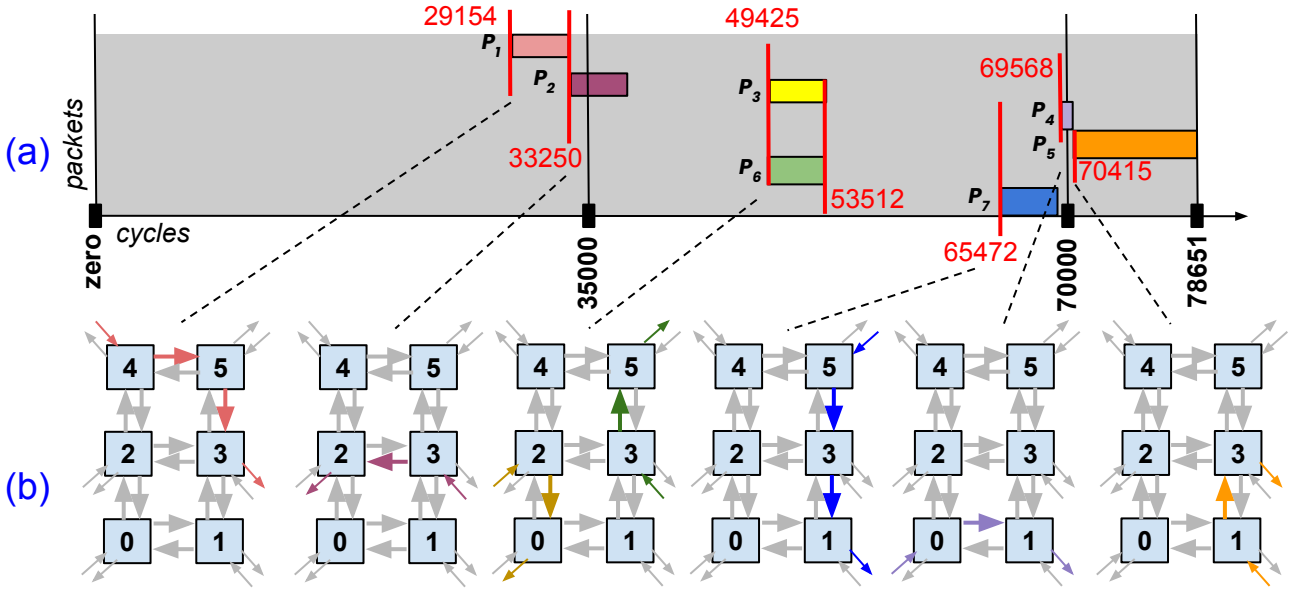
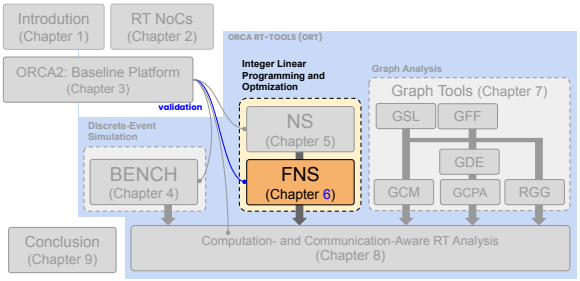


Figure 5.9 – Results of the simulation for the *DCT-Verify* application, depicting the spam lifetime of packets (a) and the usage of network links (b).

Chapter 6 discusses strategies to tune our approach for larger applications, mainly by implementing a custom solver. Besides, our approach does not consider computing time (we assumed the task with WCET equals zero). Chapter 8 shows how we include computation in our analysis using an improved ORT/NS tool version as part of a more extensive analysis framework.

6. ORT/FNS: FINDING THE MINIMUM FEASIBLE FREQUENCY FOR A VALID NETWORK TRAFFIC SCHEDULE

This chapter presents the ORT/FNS tool, made on the top of the ORT/NS tool (Chapter 5). The tool can find feasible schedules faster (F) than the ORT/NS tool due to a plenty of improvements. The following publication relates to the tool:



> Design-Time Scheduling of Periodic, Hard Real-Time Flows for NoC-based Systems
> Domingues, A. R. P. ; Johann Filho, S.; Amory, A. D. M. ; Ost, L.; Moraes, F. G.
> 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS). 2022.
> https://ieeexplore.ieee.org/document/9970868

Figure 6.1 shows the building blocks of our improved framework, where we replaced three building blocks of the ORT/NS, including the desired target frequency as a user input (Section 6.1). The goal of the new framework is to generate a traffic schedule while considering the target frequency. As the platform frequency influences the network throughput, we scale the application parameters (e.g., period) to match the target frequency using an *instantiator* component (Section 6.2). To improve the tool performance, we replaced the ILP solver with a new heuristic-based search engine (Section 6.3). A last modification uses binary search to find the minimum frequency required for the traffic to schedule (Section 6.4). We present results on the new framework in Section 6.5.

6.1 Application Model

In the ORT/NS tool (Chapter 5), we described applications using direct graphs $G = V \times E$, where vertices (V) model tasks and edges (E) model flows. A flow is a 5-tuple $f =< p, c, d, s, t >$, where p, c, and d correspond to the period, capacity, and deadline of flows, and s and t are the source and destination tasks. We measure components p and d in discrete time units, which we previously considered simply as cycles. Consequently, our tool could generate a valid schedule or output that the schedule is unfeasible. However, we know that increasing the system frequency also increases the NoC bandwidth. Thus, if we could measure p and d in standard time units (e.g., milliseconds), we could try to schedule the same traffic while targeting different frequencies. Figure 6.2 shows an example of task characterization.

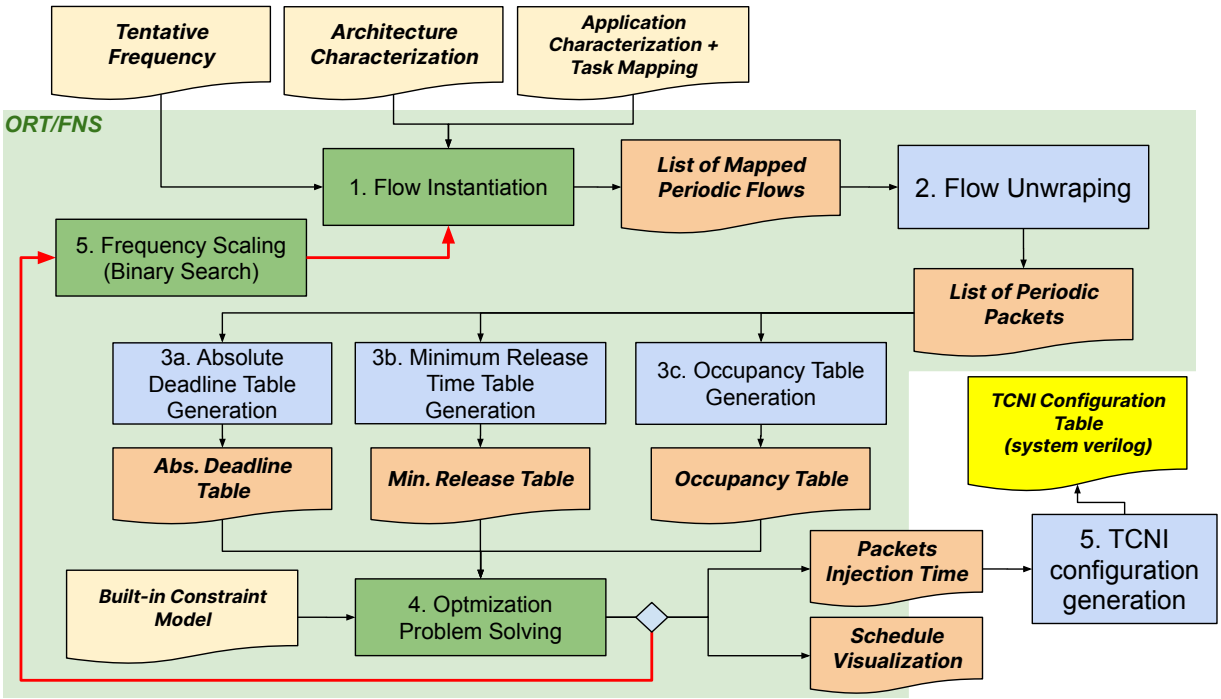


Figure 6.1 – The building blocks of the modified framework. The green color represents the newly added blocks. The red line is the optimization path for the target frequency.

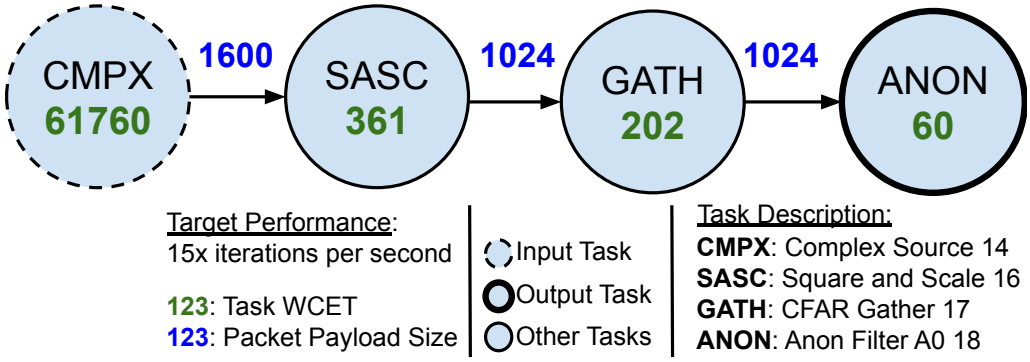


Figure 6.2 – Constant False Alarm Rate (CFAR) Detection application, from the STR2RTS benchmark [Rouxel and Puaut, 2017]. Green values represent tasks WCET, in cycles. Blue values represent network load, in bytes. The application period is $1/15$ seconds. Deadline is assumed to be equal to the period.

Our modified application model allows for workflow applications. In such applications, the whole application is bound to a single period. We assume the application deadline equals its period, and all tasks have the same period and deadline. The hyperperiod becomes the critical path execution time, comprehending from the beginning of the input task to the end of the output task. In a manycore environment, input and output tasks receive and send data from/to outside the system. In some applications, there can be multiple input and output tasks. Example applications include those processing data from sensors [Vancin et al., 2020].

Regarding flows, we assume the period of flows to be equals to the period of the application. In the CFAR application [Rouxel and Puaut, 2017], the period is 66.666*ms* (i.e., $^1/_{15}$ seconds, rounded down). Consequently, the period of all flows is the same. At this moment, we are still not considering the application CPU load.

6.2 Instantiator

The instantiator module adjusts the application parameters to match a given frequency. Although tasks WCET and flows payload size will not be modified, we must adjust the application period and deadline. By increasing the target frequency, we increase the number of cycles in the hyperiod and the chances of finding a valid traffic schedule. Figure 6.3 shows an example of frequency scaling for the CFAR application. The CFAR application running in 1GHz manycore with a period equal to $^1/_{15}$ seconds executes in $\sim 65M$ instructions per iteration. By decreasing the frequency to 500*MHz*, the number of instructions will be $\sim 32M$ for the same period of $^1/_{15}$ seconds ($^1/_{15}$ seconds $\simeq$ 66.666*ms*). Since the frequency of the target NoC is the same as the CPU, the number of flits per iteration does not change with the frequency. Table 6.1 and Table 6.2 show the characterization of the CFAR application for 1*GHz* and 500*Mhz* frequencies, respectively.

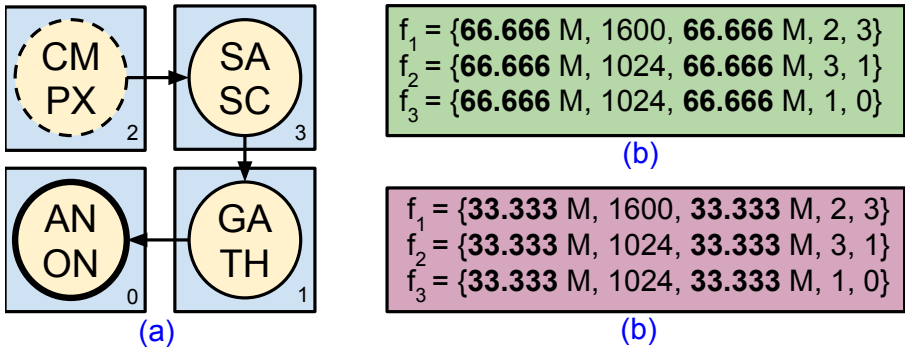


Figure 6.3 – CFAR application [Rouxel and Puaut, 2017] mapped to a 2x2 NoC (a), and application flows targeting frequencies 1GHz (b) and 500MHz (c).

6.3 Scheduling Packets with a Custom Solver

In our approach, the goal of the ILP solver, e.g., Gecode [Gecode, 2024], is to search the solution space for a feasible schedule, given our constraint model. The solution space corresponds to the intervals in which packets can be injected into the network, bound by their minimum and maximum release times. The maximum release time is given by *deadline* minus *occupancy*, and the minimum release time is zero. Table 6.3 shows the

solution space for the CFAR application. The number of possible schedules is given by $31733 \times 32309 \times 32309 \cong 33.125 \times 10^9$, i.e., 33+ trillions of alternatives. For instance, Minizinc and Gecode can search such a solution space for hours. Assuming larger applications, e.g., the CAR application [Shi et al., 2010], the number of possible schedules can be up to $\simeq 7 \times 10^{206}$. The scheduling problem is intractable, and an exhaustive search is impractical except for tiny problem instances. We developed a custom solver to address larger applications.

Table 6.1 – CFAR application characterization for 1GHz.

		Links								
Pkts.		L-2	2-3	3-L	L-3	3-1	1-L	L-1	1-0	0-L
occup.	P_1	1600	1600	1600	–	–	–	–	–	–
	P_2	–	–	–	1024	1024	1024	–	–	–
	P_3	–	–	–	–	–	–	1024	1024	1024
min.start	P_1	0	0	0	–	–	–	–	–	–
	P_2	–	–	–	0	0	0	–	–	–
	P_3	–	–	–	–	–	–	0	0	0
deadline	P_1	66.666	66.666	66.666	–	–	–	–	–	–
	P_2	–	–	–	66.666	66.666	66.666	–	–	–
	P_3	–	–	-	–	–	–	66.666	66.666	66.666

Note A: All units represented in cycles; **Note B**: Dash lines represent sparse matrix emptiness.
Note C: Deadline appear multiplied by 10^{-6}.

Table 6.2 – CFAR application characterization for 500MHz.

		Links								
Pkts.		L-2	2-3	3-L	L-3	3-1	1-L	L-1	1-0	0-L
occup.	P_1	1600	1600	1600	–	–	–	–	–	–
	P_2	–	–	–	1024	1024	1024	–	–	–
	P_3	–	–	–	–	–	–	1024	1024	1024
min.start	P_1	0	0	0	–	–	–	–	–	–
	P_2	–	–	–	0	0	0	–	–	–
	P_3	–	–	–	–	–	–	0	0	0
deadline	P_1	33.333	33.333	33.333	–	–	–	–	–	–
	P_2	–	–	–	33.333	33.333	33.333	–	–	–
	P_3	–	–	-	–	–	–	33.333	33.333	33.333

Note A: All units represented in cycles; **Note B**: Dash lines represent sparse matrix emptiness.
Note C: Deadline appear multiplied by 10^{-6}.

Table 6.3 – Search space for the CFAR application (at 500MHz).

		Links					
Pkts.		L-2	2-3	3-L	L-3	3-1	1-L
search space	P_1	[0;31733]	[0;31733]	[0;31733]	–	–	–
	P_2	–	–	–	[0;32309]	[0;32309]	[0;32309]
		L-1	1-0	0-L			
	P_3	[0;32309]	[0;32309]	[0;32309]			

6.3.1 Collision Checking

Our first effort while developing our solver was to develop a method for validating arbitrary solutions. Given an arbitrary solution, we had to decide whether the given solution was a valid schedule. Regardless of the search strategy, the verification of the solution would run for every solution during the search. Developing a CPU-intensive verification method could jeopardize the performance of the solver. We aimed to perform the least number of checks possible, eliminating invalid schedules as quickly as possible. Our solution included traversing the resulting matrix and searching for links shared by two or more packets. We use the non-overleap constraint for these links to ensure that packets do not collide. Table 6.4 shows the result matrix for the Synthetic-Flow-B application, where colored numbers indicate colliding links. We implemented the collision detection on the one-dimensional plane, as Equation 6.1, where A and B are the colliding packets, A_{rt} is the release time of A, and A_{at} is the arrival time of A ($A_{at} = A_{rt} + A_{occupancy}$).

Table 6.4 – Optimization results for Synthetic-flow-B. Values with the same colors represent colliding links (total of 6 collisions).

		Links										
Pck.		L-0	0-1	1-L	1-3	3-L	L-2	2-3	3-1	L-3	3-2	2-0	0-L
injection time	P_1	416	416	416	–	–	–	–	–	–	–	–	–
	P_2	0	0	–	0	0	–	–	–	–	–	–	–
	P_3	–	–	–	–	416	416	416	–	–	–	–	–
	P_4	–	–	0	–	–	0	0	0	–	–	–	–
	P_5	–	–	–	–	–	–	–	–	256	256	256	256

Note A: All units represented in cycles; **Note B**: Dash lines represent sparse matrix emptiness.
Note C: Colliding links are L-0, 0-1, 1-L, 3-L, L-2, and 2-3.

$$1D\_collision(A_{rt}, A_{at}, B_{rt}, B_{at}) \Rightarrow Max(A_{rt}, B_{rt}) \leq Min(A_{at}, B_{at}) \tag{6.1}$$

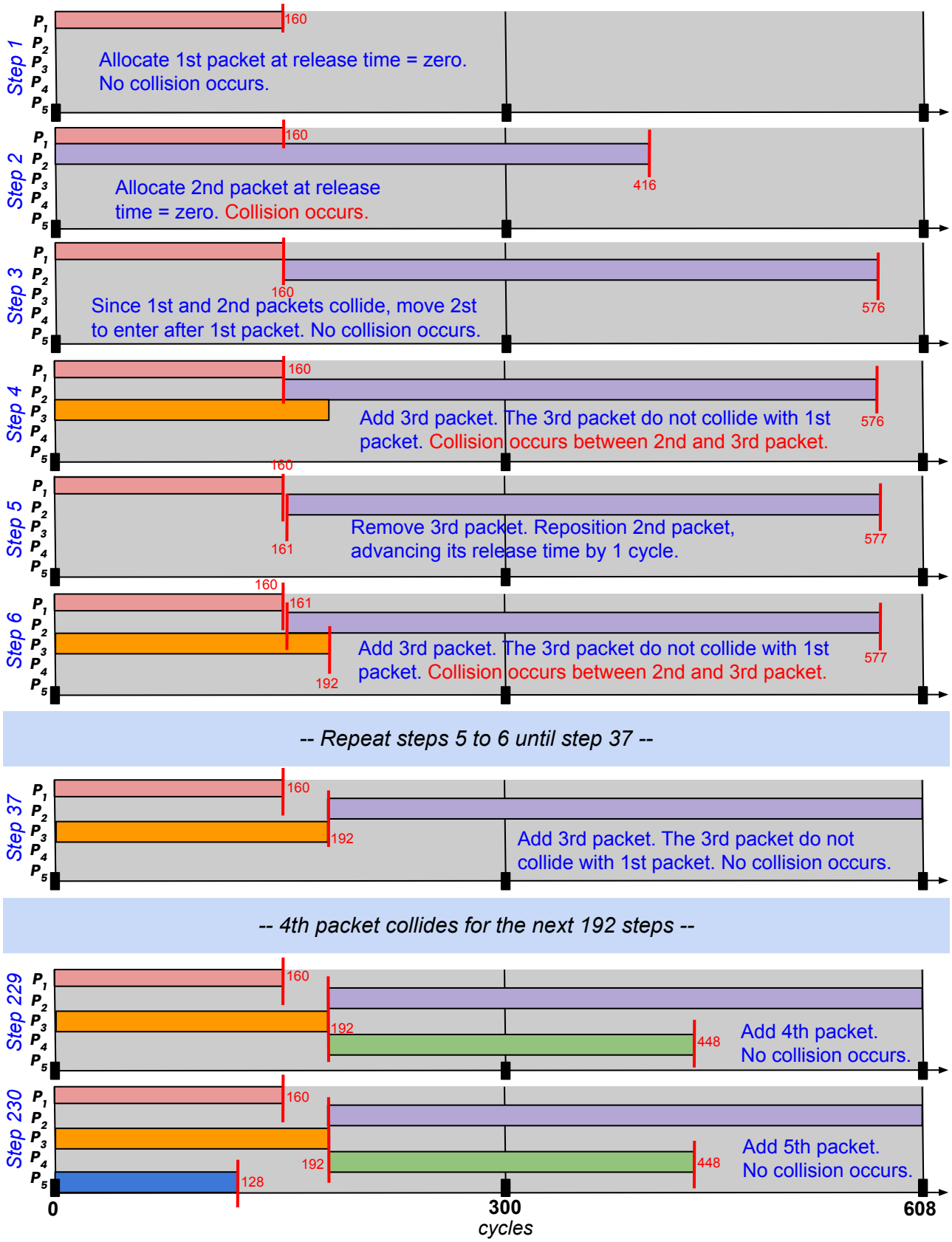


Figure 6.4 – Our custom solver performing the schedule of the Synthetic-Flow-B application. The algorithm takes 230 steps to find a feasible schedule.

6.3.2 Traversing the Search Space

To mitigate the collision checks during the search, we developed an algorithm that progressively adds packets to the solution. The goal is to add all packets such that there is no packet conflict. At the beginning, the solver adds a packet and checks for the consistency of the solution. If the solution is inconsistent, the algorithm backtracks, increasing the release time of the added packet. The algorithm progresses to another packet once the packet fits in the solution. If the packet does not fit, the algorithm backtracks to change the release time of the previous packet. If there are no packets to backtrack, the schedule is unfeasible.

Figure 6.4 shows our algorithm generating a schedule for the Synthetic-Flow-B. The resulting schedule differs from the one presented in Section 5.4.1 because our algorithm searches all possibilities for the release time, from the minimum release time to the maximum release time. Minizinc starts in the opposite direction, allocating packets as close to the hyperperiod as possible. Since we cannot compare solutions regarding quality, both solutions represent valid schedules.

6.3.3 The Pruning Factor

In the example of Figure 6.4, it took 192 steps to allocate a single packet in the solution. As one may have noticed, this number of steps is as large as the number of allocation possibilities for that packet. To accelerate the algorithm, we designed a *pruning factor* (PF) parameter. Instead of incrementing the release time by one cycle in each step, the increment is performed on the PF, skipping neighbor solutions unlikely to fit. For instance, if an allocation attempt just failed, we can skip ahead a couple of cycles as neighbor solutions are likely to fail. A PF of 1 matches has no effect, while any PF $n \in \mathbb{N}^+$ reduces the size of the search space n times. Figure 6.5 shows our custom solver adjusting the release time of packet P_2 in the schedule of the Synthetic-Flow-B for a PF of 10 cycles. The number of steps for allocating packet P_2 was reduced from 31 to 4 steps due to the number of skipped neighbor solutions. Please note that a large PF may skip over valid solutions, making the solver flag the schedule as unfeasible.

6.3.4 Guided Search

Pruning the search space reduces the number of alternatives for medium-to-small-sized problems. However, it is of little use in larger problems. For instance, it reduces the solution space of the Shi et al. [Shi et al., 2010] problem from $\simeq 7 \times 10^{206}$ to only

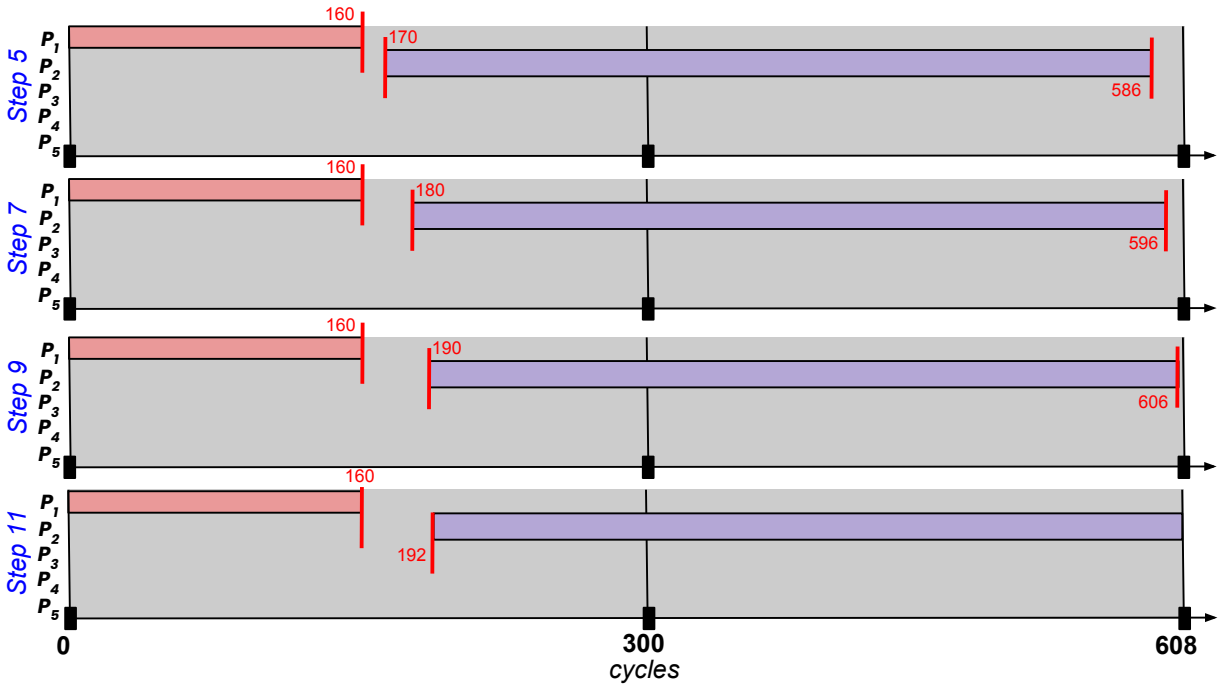


Figure 6.5 – Our custom solver performing the schedule of the Synthetic-Flow-B application using a Prune Factor of 10 cycles.

7×10^{204} if using a PF of 100, i.e., skipping 99 alternatives ahead in each step. We optimize our algorithm even further by selecting which packets to allocate first. In the example of Figure 6.4, finding the feasible schedule would take more steps if we tried to fit P_2 prior to packet P_1. To avoid such a situation, we sort a list of packets so that the solver will try to allocate packets using one of the following criteria:

- *LSTF (Least Slack-Time First)*: We call the slack-time of a packet the number difference between its maximum release time and its minimum release time. The LSTF criterion tries to allocate packets with the least number of alternatives first. Equation 6.2 shows the LSTF function.

$$LSTF(pkt) = pkt_{period} - pkt_{size} \tag{6.2}$$

- *MBCF (Most Bandwidth Consuming First)*: Allocates packets that consume more bandwidth first. The amount of bandwidth is given the size of the packet multiplied by the number of times it is injected during the hyperperiod, as Equation 6.3 shows.

$$MBCF(pkt) = pkt_{size} \times \frac{HP}{pkt_{period}} \tag{6.3}$$

- *MCPF (Most Critical Path First)*: The MCPF function gives each packet a score based on the links they traverse. First, we sort links by usage. We give each link a "usage ratio," given by the time the link is occupied during the hyperperiod (in cycles). Then,

for each packet, we sum the usage ratio of all links they traverse. This criterion aims to allocate packets in the network hotspots first.

With the guided search, our solver now performs a two-step algorithm. First, it uses LSTF, MCPF, or MBCF to generate an ordered list of packets. Then, it proceeds to try to add packets to the solution, as Section 6.3.2 discusses. Finally, the pruning factor is applied to reduce the number of tries per packet. Regardless of the chosen criteria, the guided search algorithm performs as Equation 6.4 shows, where $O(nlogn)$ is the effort for sorting the list of packets (e.g., using Quick Sort), $O(^{|P|\times HP}/_{PF})$ is the effort for finding the release time of all packets, P is the number of packets, and PF is the pruning factor. Please note that this is a very pessimistic analysis, as the effort for allocating a packet depends on the number of alternatives for that packet, not the length of the hyperperiod.

$$O(nlogn) + O\left(\frac{|P| \times HP}{PF}\right) \tag{6.4}$$

6.3.5 Adaptive Guided Search

The guided search (LSTF, MBCF, and MCPF) may eventually fail to allocate one or more packets if “critical packets” were left to be allocated lastly. Critical packets may cause link collision due to their increased period-size ratio. Allocating such packets early in the algorithm reduces the exploration of local minima in the solution space.

We tweaked the guided search algorithm, developing a new version called Adaptive Guided Search (AGS). Initially, we include a *threshold* to limit the number of tries when scheduling packets. Once the allocation of a packet fails, the algorithm resets, putting the failed packet on the top of the packet list (initially generated by LSTF, MBCF, or MCPF). Figure 6.6 shows the pseudo-code for the AGS algorithm.

The AGS algorithm generates an ordered list R of packets using LSTF, MBCF, or MCPF (line 3). The goal is to remove all packets from R and either schedule them (line 15) or flag them as critical (line 21). Every time the algorithm flags a packet as critical, it increments the number of tries for that packet (line 16). When the number of tries surpasses the configured *threshold*, the algorithm returns the generated schedule and the set of packets that could not be scheduled (line 22). The algorithm pushes the failed packet to the top of the list of packets, restarting.

```
Algorithm Adaptive Guided Search
 1. Inputs: the set of packets to be scheduled, P.
 2. Begin
 3.     Let R ← lstf(P) or mbcf(P) or mcpf(P)
 4.     Let Q ← an empty ordered set
 5.     Let S ← { } // the skipped packets
 6.     Let F ← the prunning factor,
 7.     Let G ← threshould (max. tries per packet)
 8.     Let T ← [ zeroes ] // tries per packet (critical pkts.)
 9.     While |R| ≠ 0:
10.         Let schedule ← { }, temp_schedule ← { }
11.         For q in Q:     // partial schedule
12.             schedule ← allocate(schedule, q, F)
13.         temp_schedule ← schedule
14.         For r in R:     // adaptive permutation
15.             If not allocate(temp_schedule, r, F) Then
16.                 T[r] ← T[r] + 1
17.                 If  T[r] < G Then
18.                     R.push_front(r) // next to be tried
19.                     Goto Line 7
20.                 Else // packet could not be scheduled
21.                     R.remove(r), S.push(r)
22.         Return  temp_schedule, S
23. End
```

Figure 6.6 – Adaptive guided search algorithm. Packets that could not be scheduled will be selected to be tried first at the next algorithm reset. The output is the set of scheduled packets and a list of packets that could not be scheduled.

6.4 The Minimum Frequency Required for a Feasible Schedule

We could use our solver to find the minimum frequency necessary to schedule applications. Our instantiator module scales the application to the given tentative frequency. Then, the AGS algorithm tries to generate a schedule. We configure the number of tries per packet and the pruning factor per application. The result is either a full schedule (all packets successfully added to the schedule) or a partial schedule (when at least one packet could not be scheduled).

We added a wrapper over the FNS tool (FNS-Wrapper) to perform a binary search on the tentative frequency (T_t). First, we apply the tentative frequency to the AGS algorithm. We should reduce the tentative frequency by half if we get a full schedule. If the schedule is partial, we multiply the tentative frequency by two. At this point, we keep track of the last two tentative frequencies, defining the search range as $[T_f\ ;\ T_c]$, where T_f is the lower frequency among the two last tentative frequencies, and T_c is the higher frequency among the two last frequencies. We update the search range using the formula in Equation 6.5, applying the ADS algorithm to T_t at each iteration.

$$[T_f' \; ; \; T_c'] = \begin{cases} [T_c \; ; \; 2 \times T_c], & \text{if half schedule} \\ [T_f \; ; \; \dfrac{T_f + T_c}{2}], & \text{if full schedule} \end{cases} \tag{6.5}$$

The FNS-Wrapper stops searching for a frequency when the difference between T_f and T_c is lower or equals the *inverted tolerance factor* (ITF). For instance, the algorithm would stop for search interval [98MHz ; 100MHz] with an ITF = 98%, taking 100MHz as the minimum frequency. We adopt ITF = 98% for the remainder of the Thesis. In practice, this means that the frequencies found for all applications can be $0 - 2\%$ lower than the frequencies reported in the results. We avoided $ITF \geq 99\%$ because we experienced rounding errors in preliminary runs (the algorithm hangs indefinitely as it never reaches the condition to quit). Thus, we assume the theoretical ITF = 100% to be impractical.

6.5 Proof of Concept and Discussion

We evaluate the performance of our search engine for the following applications: (i) Synthetic-Flow-D, (ii) DCT-Verify (see Section 5.4.2), and (iii) CAR [Shi et al., 2010]. We execute these applications for the different packet allocation criteria (LSTF, MBCF, and MCPF) using pruning factors of 10, 20, and 40. For the CAR application, we had to increase the PF 100 times due to the length of the hyperperiod. The *threshold* is equal to the PF. Table 6.5 shows the number of visited solutions (actual algorithm effort). Table 6.6 displays the number of skipped solutions in each run (saved effort). Lower values represent the best algorithm performance.

Table 6.5 – Execution Performance (visited solution).

	LSTF			MBCF			MCPF		
Apps.**	10	20	40	10	20	40	10	20	40
Synthetic-Application D	40	16	6	40	16	6	6	6	10
DCT-Verify	8	8	8	8	8	8	214	111	60
Car application*	660	660	660	660	660	660	660	660	660

* : Pruning Factor ×100

Table 6.6 – Execution Performance (skippped solutions).

	LSTF			MBCF			MCPF		
Apps.**	10	20	40	10	20	40	10	20	40
Synthetic-Application D	96	12	2	96	12	2	4	2	3
DCT-Verify	158	314	80	416	209	105	8970	2322	626
Car application*	13629	7267	3683	13641	7282	3684	5073	3002	1762

* : Pruning Factor ×100

The AGS algorithm performed better with the MCPF criteria for the Synthetic-Application-D. Since the application has just a few packets, solutions that allocate critical packets first outperform other solutions, i.e., LSTF $PF = 40$, MBCF $PF = 40$, and MCPF $PF = 20$. The MCPF criteria outperforms other criteria when $PF = 10$. The DCT-Verify application has more flows and a lengthened hyperperiod compared to Synthetic-Application-D. The number of possibilities for allocating packets was also higher. For this reason, the AGS algorithm could find a valid schedule in fewer steps. However, the MCPF criteria for the application did not work as expected. Allocating P_6 (see Table 5.3) early in the algorithm generated many backtrack operations as P_2 had fewer possibilities for allocation. Finally, the CAR application had the larger hyperperiod, number of packets, and allocation possibilities per packet from all evaluated applications. One may observe that, regardless of the chosen criteria, the number of steps to find a valid schedule was the same in all situations (660). The number matches the number of packets in the application. However, the number of skipped solutions was higher in the LSTF and MBCF criteria.

We observe that a higher PF helps the AGS algorithm to converge faster. However, an increased PF may have negative results depending on the number of packet allocation possibilities. For instance, Table 6.7 shows the minimum frequency required for a feasible schedule for the evaluated applications. For Synthetic-Application-D, a $PF = 40$ forced the AGS algorithm to skip too many allocation possibilities, leading to a sub-optimal solution. However, the solution is optimal for $PF = 10$ and $PF = 20$. Similarly, a $PF = 10$ is associated with the optimal frequencies in the DCT-Verify and CAR applications.

Table 6.7 – Minimum scheduling frequency (MHz) using the AGS algorithm.

		Criteria and Pruning Factor		
Applications	**PF***	LSTF	MBCF	MCPF
Synthetic-Application-D	10	1,929	1,929	1,929
	20	1,929	1,929	1,929
	40	2,183	2,183	2,183
DCT-Verify [Rouxel and Puaut, 2017]	10	793	1701	792
	20	795	1707	795
	40	800	1707	800
Shi et al. [Shi et al., 2010]	10	2,813	2,438	4,219
	20	2,813	2,438	4,219
	40	2,813	2,813	4,688

PF*: Prunning Factor, : Optimal frequency, : Worst resulting frequency

We observed that the pruning criteria had no effect on the quality of the frequency for the Synthetic-Application-D. Since packets had only a few alternatives for allocation, the ADS algorithm fit them in almost the same place for all criteria.

For the DCT-Verify application, the MCBF criterion showed the worst result among the criteria. The MCBF criterion prioritized packet P_6 over packet P_2, leading to an increased number of tries while fitting P_2. With the MCPF criterion, packet P_2 gets prioritized, resulting

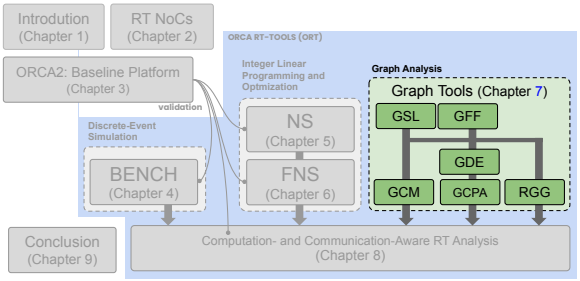
in fewer tries while fitting P_6. For this application, the MCPF criterion presented the best frequency, slightly outperforming the LSTF criterion.

Contrarily to the DCT-Verify application, the MBCF presented the best frequency for the CAR application. Since the application has a more significant number of conflicting packets, prioritizing the packets in the critical path increases the number of backtrack operations in the algorithm. Since we adopt a threshold equal to the PF (1000 for this application), most of the packets "timed out" before AGS found a fit for them, generating many resets in the algorithm.

6.6 Final Remarks

We presented presented a custom solver that replace ILP solvers by the AGS algorithm, increasing the performance when searching the solution space for schedules. The FNS-Wrapper uses the AGS algorithm to perform a binary-like search to find the minimum frequency to schedule applications. Both the solver and wrapper are novel contribution to this Thesis. We demonstrated the performance of our solver on a synthetic application (Synthetic-Application-D) and two other applications from the literature: DCT-Verify, from the STR2RTS benchmark [Rouxel and Puaut, 2017], and the CAR application [Shi et al., 2010]. Our framework reduced the time to find solutions by 70.58%, 95.18%, and 95.38% for the aforementioned applications, respectively. Finally, we use the ORT/FNS tool later in Chapter 8 as part of a larger real-time analysis framework.

7. ORT/GCM: COMMUNICATION PARTITIONING IN SHARED-MEMORY AND I/O SPACES



This Chapter presents the ORT/GCM tool (Graph Collapse Method), which implements a method for clustering applications graphs while partitioning the communication load between the memory and I/O spaces. This part of the Thesis has yet to be published.

We organize the chapter as follows. Section 7.1 presents the background on graph clustering and the GRAPH-COLLAPSE method. We discuss the cutting criteria of our method in Section 7.2. We evaluate our method in Section 7.3, while presenting minor tools for graph manipulation. Section 7.4 show the application of our method to an application from the literature. We conclude the chapter in Section 7.5 with a brief discussion of the proposed method.

7.1 Graph Clustering

For applications where the number of tasks exceeds the number of available computing nodes (CPUs) in the target system, one must adopt some scheduling strategy to guarantee CPU time sharing. Since we focus our frameworks on the static analysis of flows at the pre-runtime, we need to group tasks into different task sets and further map these sets into processing elements, assuming a manycore architecture. Tasks mapped to the same CPU communicate through shared-memory spaces, while tasks mapped to different CPUs will use the underlying network to communicate. Consequently, we must consider the benefits of mapping specific tasks together and the penalties of not doing so. For this reason, we develop a strategy for task clustering. As a requirement, the task clustering strategy must be scalable, i.e., capable of clustering graphs regardless of the application and NoC size.

Task set clustering consists of splitting a task set into multiple subsets, so one can eventually map each subset set to a different CPU core. Ideally, the strategy to group tasks for each CPU core relates to some aspects of the target system. For instance, one may group tasks targeting objective functions such as network cost reduction, minimizing or maximizing CPU usage, or even reducing overall communication cost. Regardless of the objective function, one may adopt algorithms such as HCS (Highly Connected Subgraphs) [Hartuv and Shamir, 2000] if clustering for vertices similarity. Other strategies for graph clustering include those based on MIN-CUT and MAX-CUT (see [Goldberg and Tarjan,

1988]). For example, the Stoer–Wagner algorithm [Stoer and Wagner, 1997] can solve MIN-CUT in polynomial time for undirected, weighted graphs. Nevertheless, we could not locate an algorithm for clustering directed graphs in polynomial time complexity in the literature.

7.1.1 The GRAPH-COLLAPSE Algorithm

We developed a greedy algorithm for clustering application graphs that targets directed, weighted graphs. Figure 7.1 shows the pseudo-code for our GRAPH-COLLAPSE algorithm, whose implementation resulted in the ORT/GCM tool. The rationale behind the GRAPH-COLLAPSE algorithm is to remove edges from the graph until we find a given number of vertices. At each iteration, the algorithm removes at least one edge from the graph, merging their connecting vertices (destination and departure vertices). At the end of the algorithm, the graph will collapse into a couple of vertices, each vertex representing a subgraph (cut). Our algorithm can be seen as a variation of the Stoer–Wagner algorithm, although we choose to eliminate edges (not vertices) from the graph. Instead of cutting the graph at each iteration, our algorithm greedily selects edges for removal. The selection occurs according to a *cutting criterion*.

Algorithm GRAPH-COLLAPSE

```
1.  Inputs: a weighted directed graph G = <E,V,w>
2.          a collapse criterion C
3.          a target number of nodes N
4.  Begin
5.      While |E| > N:
6.          Let a ← C(E)
7.          Let origin ← vertice from which a departs
8.          Let destination ←vertice to where a points to
9.          Let od ← node merging origin and destination
10.         For e in E:   // merge two vertices
11.             If e points to origin or points to destination:
12.                 e now points to od
13.             If e departs from origin or from destination:
14.                 e now departs from od
15.         remove(V, origin)  // remove old vertices
16.         remove(V, destination)
17.         add(V, od) // add new node
18.         For e in E:   // clean up circular edges
19.             If e departs from v and points to v | v ∈ V:
20.                 remove(E, e)
21.     Return G
22. End
```

Figure 7.1 – The GRAPH-COLLAPSE algorithm.

The GRAPH-COLLAPSE algorithm has two main components. First, the algorithm uses the input criterion to select an edge from the graph, remove the edge, and make the necessary adjustments to preserve vertices and edge properties. Regardless of the chosen criterion, the algorithm modifies the input graph to generate an in-place solution with space complexity of $O(1)$. For a graph containing $|V|$ vertices, it takes at most $(|V|-N)$ iterations for the algorithm to complete, where N is the number of clusters (vertices in the resulting graph). The second part of our algorithm is the cutting criteria. At each iteration, the time complexity of applying the criterion sums to the graph maintenance operations: vertex merging, edge removals, adding new vertices to the graph, and cleanup. Since ORT/GCM implements the graph model using arrays to store vertices and edges data, the time complexity of our algorithm is linear on the number of edges. Equation 7.1 shows the time complexity of our algorithm, where C is the time complexity of the selected criterion.

$$O(C \times \frac{|E|}{2}) \tag{7.1}$$

7.2 Cutting Criteria

In the context of this Thesis, application graphs are models representing whole task sets. As such, removing edges from a graph is the same as removing communication between tasks, which may sound odd in the context of manycores. However, since we group tasks in the same CPU core, removing edges means that we are not using the underlying NoC to transmit application data. Instead, the software writes the payload of removed links to the memory space. Due to the functioning of our network driver, packets transferred from one task to another in the same CPU core will use the memory space. The network driver pushes the packet pointer into the receiving task packet queue; this operation takes only a couple of instructions (see Section 3.2.3).

Selecting cutting criteria means we select which part of task communication will transfer through the underlying NoC and which part will transfer through the shared memory space. Removing edges with the most network payload may sound natural, leaving the system with the minimum network load possible. In our first attempt to create clusters with fair usage of system resources, we developed two criteria: MIN-COMM and MAX-COMM.

The MIN-COMM criterion keeps the edges with the minimum communication load and eliminates edges with the most communication load from the graph. Figure 7.2 shows the application of the MIN-COMM criterion over a random graph. The GRAPH-COLLAPSE algorithm removes the edge with the most network load from the graph at each iteration. The algorithm also merges the vertices before connected by the removed edge, reducing the size of the graph. Although the resulting clustering optimally reduces the communication load, the processing load of the clusters is unbalanced.

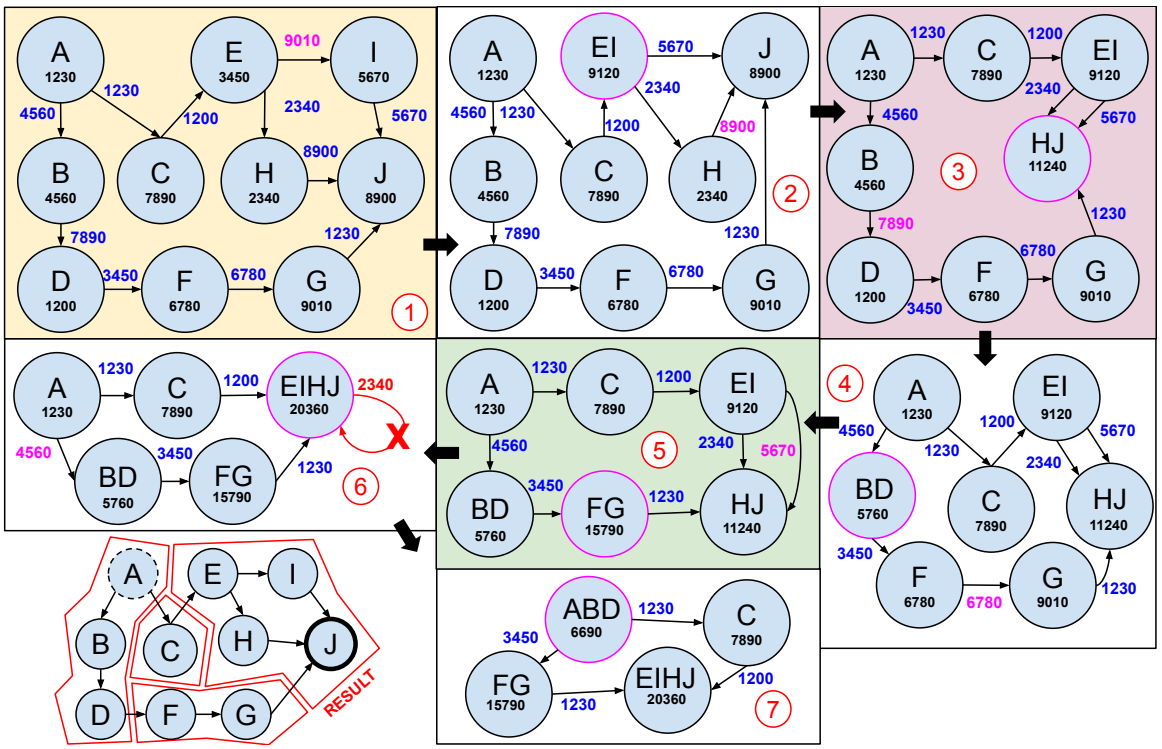


Figure 7.2 – The GRAPH-COLLAPSE algorithm applied to a graph using the MIN-COMM criterion and $N = 4$. The input graph is shown at the left-top corner (yellow background).

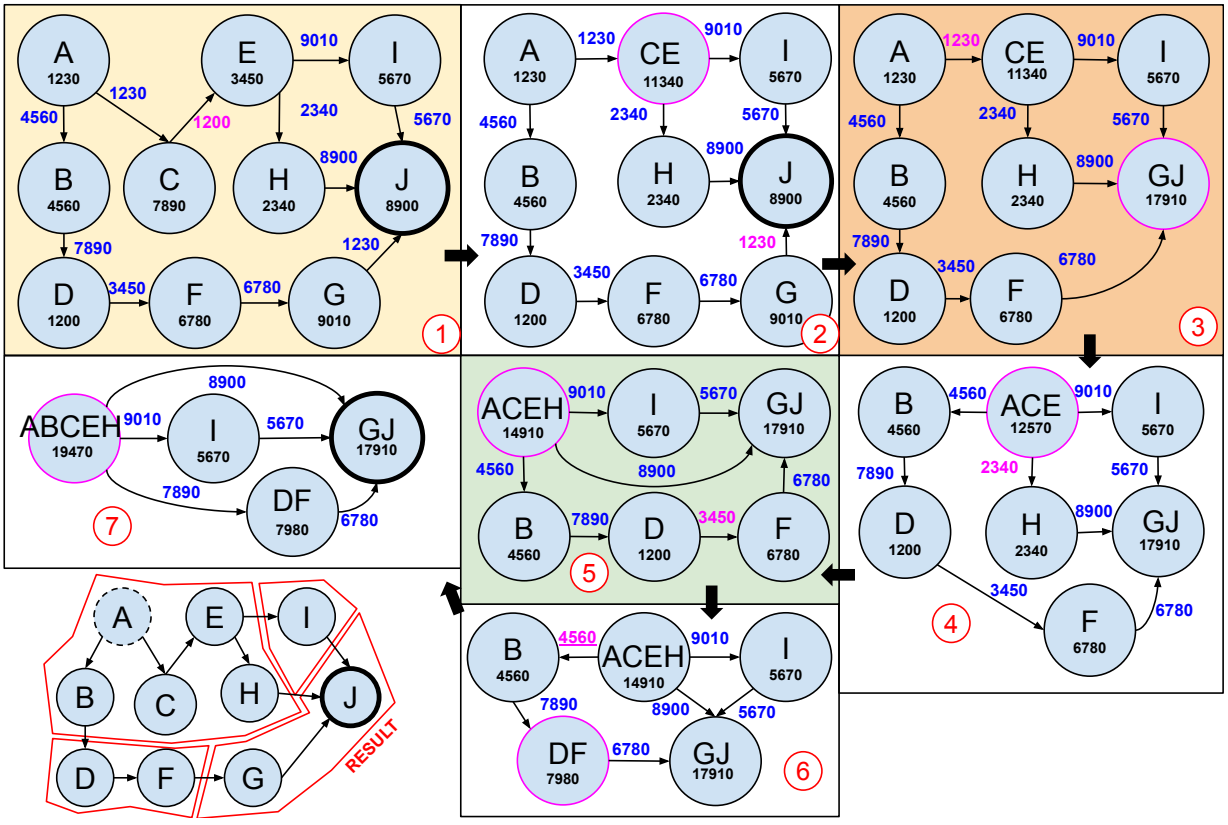


Figure 7.3 – The GRAPH-COLLAPSE algorithm applied to a graph using the MAX-COMM criterion and $N = 4$. The input graph is shown at the left-top corner (yellow background).

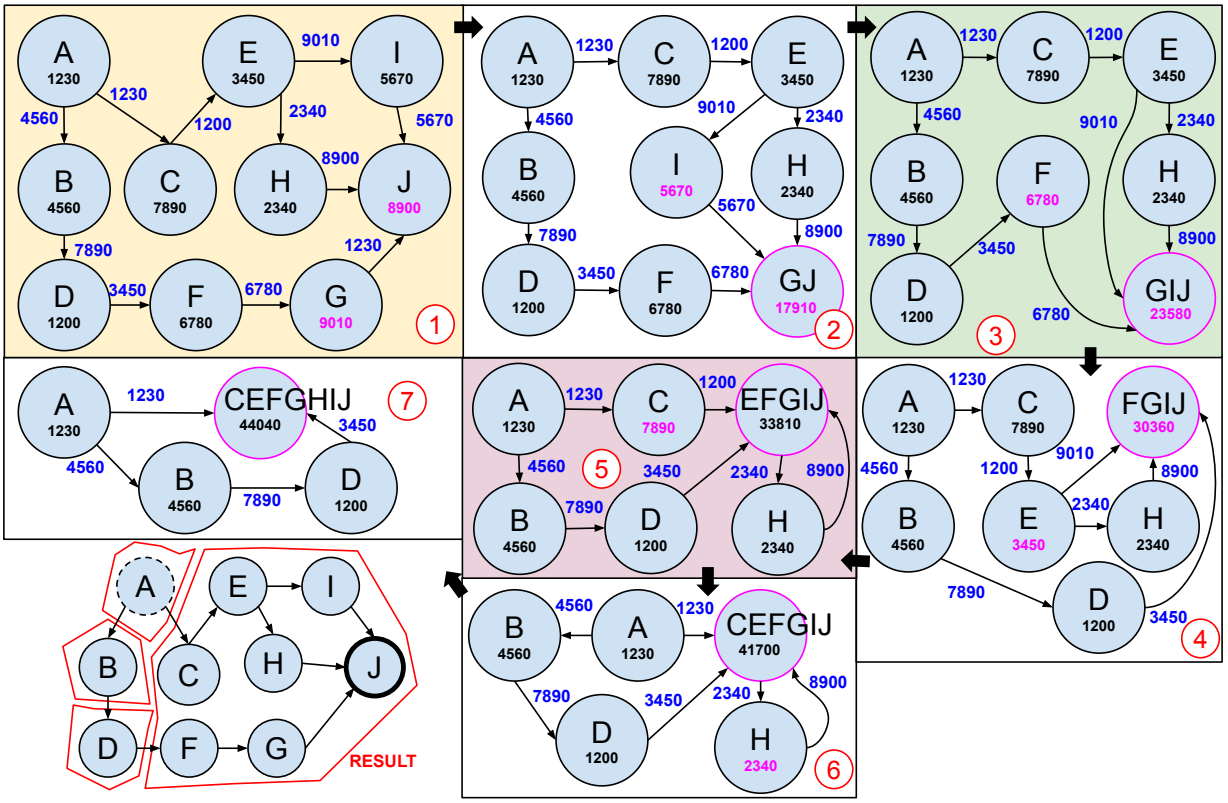


Figure 7.4 – The GRAPH-COLLAPSE algorithm applied to a graph using the MIN-PROC criterion and $N = 4$. The input graph is shown at the left-top corner (yellow background).

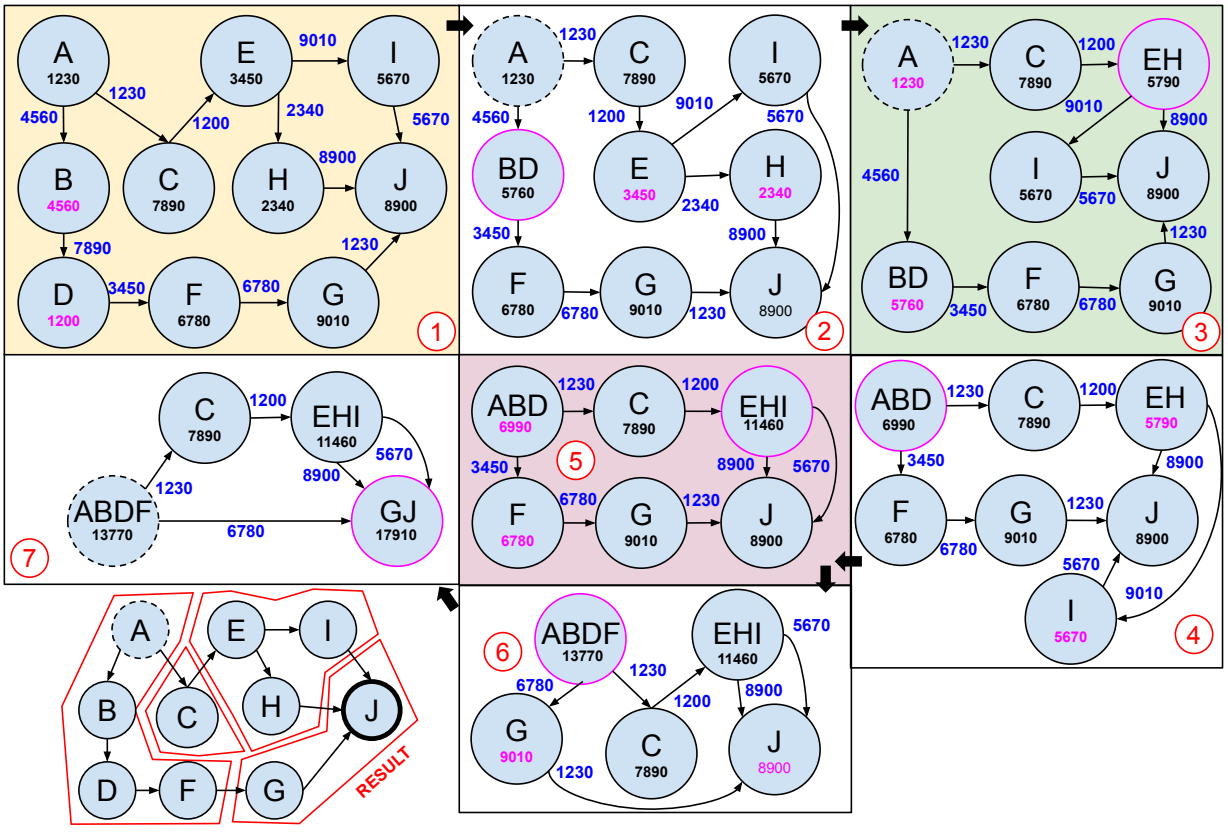


Figure 7.5 – The GRAPH-COLLAPSE algorithm applied to a graph using the MAX-PROC criterion and $N = 4$. The input graph is shown at the left-top corner (yellow background).

The MAX-COMM criterion acts similarly to the MIN-COMM criterion, except it removes edges with the minimum network load while keeping the edges with the most network load. Figure 7.3 shows the application of the GRAPH-COLLAPSE algorithm for the MAX-COMM criterion over the same graph as the MIN-COMM example. In practice, the MAX-COMM criterion generates the worst solution when considering network usage. We quickly observe that graphs generated from MIN-CONN and MAX-CONN criteria do not offer any balance between computation workload among the tasks. The reason for such results lies in the lack of relation between computation and communication in the criteria, which removes edges from the graph regardless of which vertices will be merged.

The MIN-PROC criterion selects edges whose sum of the CPU load of connected vertices is the highest among all pairs. In other words, our algorithm merges vertices with the highest CPU load first. Figure 7.4 shows an example of the GRAPH-COLLAPSE algorithm using the MIN-PROC criterion. Unfortunately, we observed that the vertex with the highest value absorbs almost all the vertices in the graph, making the MIN-PROC criterion of little use in a practical scenario. Contrarily, the MAX-PROC provides the most balanced resulting graph among the criteria discussed so far by first merging the vertices with the lower CPU load. Figure 7.5 shows an example of the GRAPH-COLLAPSE algorithm using the MAX-PROC criterion. However, the MAX-PROC criterion neglects the communication load, creating unbalanced clustering from the communication viewpoint.

7.3 Criteria Evaluation

To better understand the effects of criteria in the graph clustering process using the GRAPH-COLLAPSE algorithm, we experimented by running our algorithm over a set of randomly generated graphs. To generate the graphs, we use the ORT/RGG tool, which we developed in the context of this thesis. The ORT/RGG tool is a random graph generator in which one can generate graphs by adjusting (i) the number of vertices, (ii) edge density, and (iii) minimum and maximum values for values tagged in vertices and edges. The tool expresses edge density and limits for tagged values through normal distribution curves with configurable parameters σ and μ. In this Thesis, we use $\sigma = 10,000$ and μ equals the number of edges in the graph. We collect data from the original graphs generated by the ORT/RGG tool and the resulting graph from the GRAPH-COLLAPSE algorithm (ORT/GCM) to evaluate the criteria. Finally, we automate the evaluation using the ORT/GDE tool (Graph Data Extractor), implementing the following metrics.

M1) *Standard Deviation of Edge Values*. This metric represents the distribution of the traffic in the network links. Higher values may indicate network hotspots. This metric is as Equation 7.2 shows, where x_i is the value tagged in the i^{th} vertex of the graph, and μ is the average vertex tagged value.

$$M1 = \sqrt{\frac{\Sigma(x_i - \mu)^2}{|E|}} \tag{7.2}$$

M2) *Standard Deviation of Vertex Values*. This metric measures the homogeinity of the CPU load in the graph application. Lower values represent near-optimal CPU usage, while higher values represent unbalanced CPU usage among the clusters, assuming symmetric CPU cores. This metric is as Equation 7.3 shows, where x_i is the value tagged in the i^{th} edge of the graph, and μ is the average edge tagged value.

$$M2 = \sqrt{\frac{\Sigma(x_i - \mu)^2}{|V|}} \tag{7.3}$$

M3) *Normalized Computation/Communication Rating*. This metric indicate whether the application is computation-dominated ($M3 > 1$) or communication dominated ($M3 < 1$), assuming the instruction throughput of one CPU core aproximates a single 32-bit instruction per cycle (IPC) and the network link throughput approximates 32-bit per cycle. This metric is as Equation 7.4 shows, where x_i is the tagged value of the i_{th} vertex in the graph, and y_j is the tagged value of the j_{th} edge in the graph.

$$M3 = \left(\sum_{i=0}^{|V|} x_i\right)\left(\sum_{j=0}^{|E|} y_j\right)^{-1} \tag{7.4}$$

7.3.1 Comparing Criteria for Graphs with Distinct Number of Vertices

We use the ORT/RGG tool to generate 17 random graphs, varying the number of nodes for 10% of the scale factor, for $|V|$ from 20 to 90, and from 100 to 900. We applied the MIN-COMM and MAX-PROC criteria to minimize the graph targeting a number of nodes equals to 16 (e.g., a 4×4 mesh dimension NoC). The number of edges relates to the number of vertices in the original graph, where we generate graphs in which vertices is connected to 60% of other vertices in the graph. We use the ORT/GDE (Graph Data Extractor) tool to automate the analysis of the resulting graphs and collecting metrics. Values tagged on edges and vertices follow a normal distribution of center (loc) equals to zero and width equals to 1 (scale), adjusted to the interval $[200; 2000]$. Table 7.1 show the results for the evaluation.

M1) Standard Deviation of Edge Values:

- Since MIN-COMM removes edges with higher values, the sum of edges values decreases at each iteration of the algorithm, reducing the overall network load. As the number of edges in the input graph increases, the lower is M1. This behavior

Table 7.1 – Results for the evaluation of the MIN-COMM and MAX-PROC criteria, applied to a 4×4 NoC ($N = 16$). Lines represent graphs and columns represent metrics.

	Input Graph					Output Graph (MIN-COMM)				Output Graph (MAX-PROC)			
#	|E|	|V|	$\lfloor M1 \rfloor$	$\lfloor M2 \rfloor$	$M3$	|E|	$\lfloor M1 \rfloor$	$\lfloor M2 \rfloor$	$M3$	|E|	$\lfloor M1 \rfloor$	$\lfloor M2 \rfloor$	$M3$
1	228	20	3440.995	4232.595	0.089	55	2327	74797	0.390	180	3413	12074	0.113
2	522	30	2706.639	3874.520	0.070	97	2350	157437	0.379	378	2752	22097	0.096
3	936	40	3136.718	3947.190	0.043	128	2976	183704	0.335	724	3142	19479	0.056
4	1470	50	2660.779	4310.756	0.040	167	2473	229173	0.365	1122	2683	28083	0.052
5	2124	60	2634.919	4034.489	0.023	193	2480	225930	0.261	1616	2644	30693	0.031
6	2898	70	2405.791	4266.827	0.025	260	2176	346929	0.283	2198	2382	40310	0.034
7	3792	80	2525.702	3863.026	0.023	330	2338	378513	0.265	2838	2543	43878	0.031
8	4806	90	2167.588	3662.968	0.019	377	1998	389270	0.260	3622	2190	46176	0.027
9	5940	100	2483.120	3326.092	0.015	722	2288	433947	0.132	4401	2486	54298	0.021
10	23880	200	2375.536	3223.859	0.008	732	2231	932012	0.264	17612	2382	114144	0.011
11	53820	300	2195.770	3121.223	0.005	1030	2044	1504970	0.313	40015	2195	157841	0.008
12	95760	400	2150.283	3284.168	0.003	1429	2110	1667189	0.242	70363	2143	192568	0.005
13	149700	500	2013.879	3229.622	0.003	1805	1948	2232758	0.266	111938	2014	254529	0.004
14	215640	600	1986.697	2906.002	0.002	2188	1919	3059979	0.288	160464	1990	311619	0.004
15	293580	700	2062.924	2977.308	0.002	3309	1951	3424459	0.216	215048	2062	396542	0.003
16	383520	800	1913.514	2490.328	0.002	2860	1848	3598534	0.252	281927	1913	446394	0.003
17	485460	900	1882.212	2790.095	0.002	3123	1802	4080599	0.284	358667	1882	482700	0.002

Note A: Measures in blue depict gains over the input graph, while red values depict a worse achieved value.
Note B: Values of M1 and M2 of output graphs rounded up to the nearest integers.

repeats for the output graph. Since the value of edges is limited to the interval [200; 2000], the limit of M1 for infinite number of edges is 1.

- The MAX-PROC had little influence on M1, with low variations upper and lower the baseline value. This explains the lack of relation between computation and communication in the graphs, as the ORT/RGG tool sets edges and vertices based on a normal distribution with random assignment.

M2) Standard Deviation of Vertex Values:

- The MIN-COMM criterion had little effect on M2, similarly to MAX-PROC in M1. Once more, the assignment of values to the input graph is random; there is no association between the value of a vertex and the edges connected to it. However, as the algorithm removes edges, the values of vertices increases. By the end of the algorithm, the value of each vertex is either their initial value or the sum of a set of merged nodes, explaining the increase of M2 values.
- The MAX-PROC criterion also suffers from the increased M2 due to the merging of vertices. However, the algorithm merges vertices with the least value, leaving only the vertices with the most value in the graph. Also, newly added vertices has the sum of the merged vertices as values, reducing the value of M2 in comparison to MIN-COMM. To add, none of the criteria can generate a graph whose M2 is lower in comparison to the input graph. However, we observe that MAX-PROC present a better balance regarding CPU load.

M3) Normalized Computation/Communication Rating:

- The M3 metric indicates that all applications become more computation-dominated as they are clustered. Since eliminated edges carry communication load, it is expected the computation-communication rating to increase towards computation. As the number of vertices increases in the input graph, the clustering process causes less impact in M3, i.e. M3 is the same for the input and output graph if the input graph has infinity vertices (limit). The M3 metric increases more in MINCOMM as it removes edges with the most communication load from the graph.

7.4 Proof of Concept

As a proof of concept, we apply our algorithm to the CAR application, a stereo photogrammetry application, used by a car system to detect obstacles and populate a database. Figure 7.6 shows the application as presented in the reference publication [Shi et al., 2010]. Figure 7.7 shows the graph we build from the scratch. In the paper, the authors provide no characterization of tasks, i.e. WCET. However, the author present a mapping of the application on e 4×4 mesh manycore. We consider that their map is at most "perfect balanced", i.e. the CPU usage is the same for all CPU, regardless of the task set. We also consider the theoretical safe limit of 70% CPU usage [Laplante and Ovaska, 2011]. Finally, the WCET of tasks is the division of the CPU usage (70% for all CPU) by the number of tasks mapped to that CPU. The informed hyperperiod for the application is 1 second. Assuming an arbitrary frequency of 1MHz, the extrapolated number of cycles available for tasks is $700,000$ per second (70% of 1MHz).

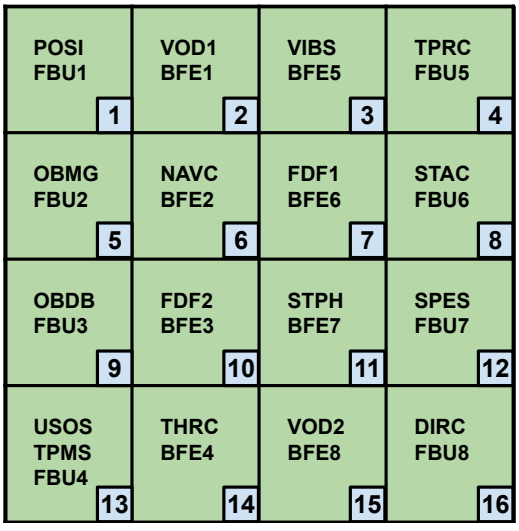


Figure 7.6 – Clustering of the CAR application (Adapted from [Shi et al., 2010]).

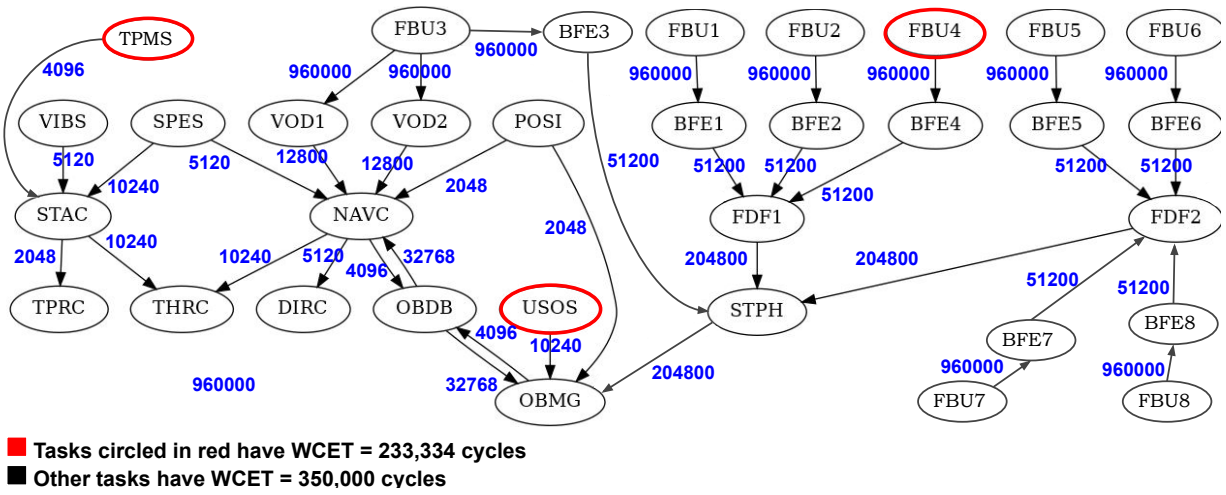


Figure 7.7 – The CAR Application.

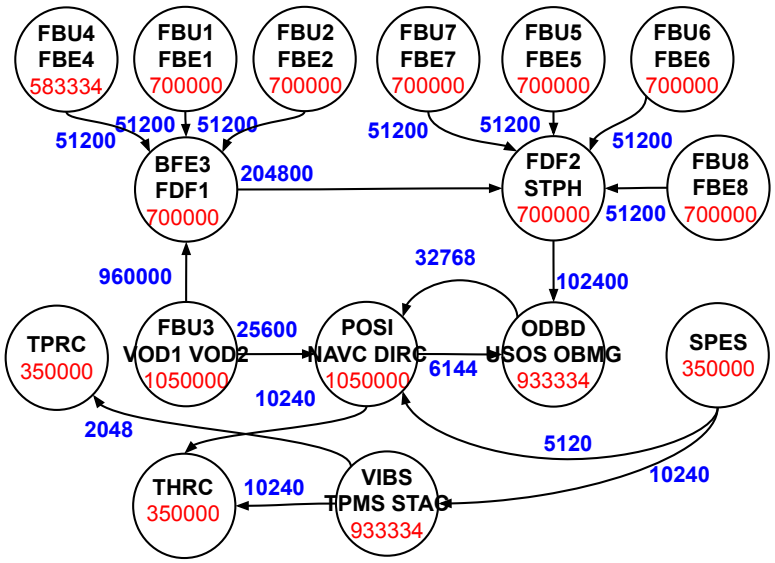


Figure 7.8 – CAR application, clustered with N=16 and MAX-PROC criteria.

We run the ORT/GCM tool to cluster the application (Figure 7.7) using the MAX-PROC criterion. Figure 7.8 shows the resulting graph. The reference clustering (Figure 7.6) has a standard deviation of vertices equals zero, as we assumed an optimal use of CPU. The clustered application, generated by the ORT/GCM tool has a standard deviation of vertices (M2) equals to $221,360$, rough representing 31% of CPU time waste. Regarding communication, the reference graph has a standard deviation of edges (M1) equals $410,267$, while the generated graph has $215,206$. On the one hand, our clustering process increased the CPU usage in 21%. On the other hand, our method reduced the network usage by 52.45%.

Figure 7.9 shows the result for the MIN-COMM criterion. The M2 of the resulting graph is $1,101,867$. As expected, the criterion eliminated edges with the least value from the graph, collapsing most of the tasks in a single cluster, increasing the CPU usage by 157%. Regarding the communication, the M1 value is $19,165.638$, rough corresponding to $0,004\%$ of the baseline value.

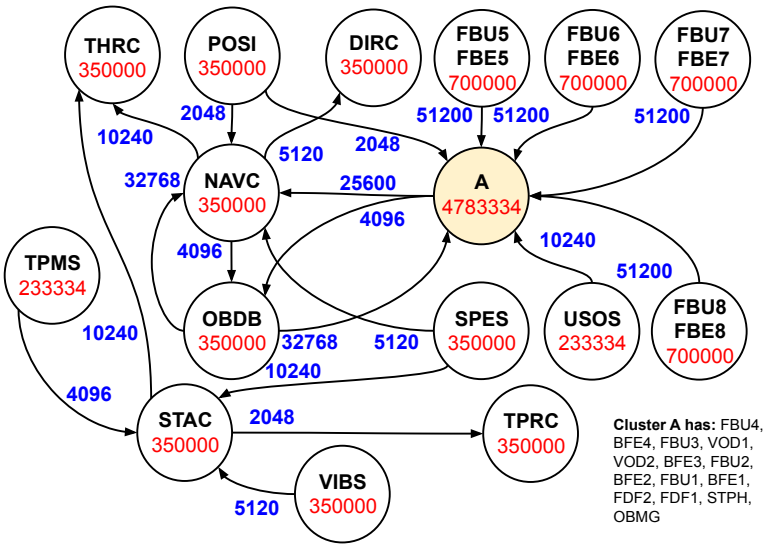


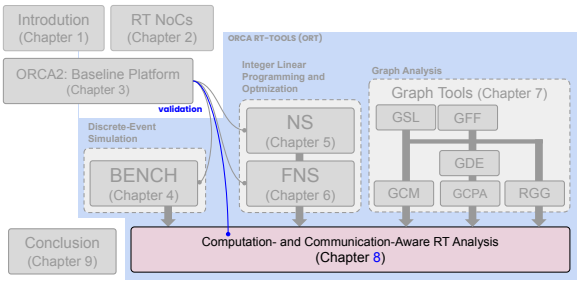
Figure 7.9 – CAR application, clustered with N=16 and MIN-COMM criteria.

7.5 Final Remarks

This chapter presented a method for clustering graphs whose performance is linear to the number of edges in the application graph. Our method is fully automated by the OR-T/GCM tool, allowing for fast analysis and clustering of application graphs.. One drawback of our method is the lack of balance between computation and computation, as the approach does not support executing multiple criteria. As a result, the output graph can be unrealistic, either surpassing the CPU or network capacities of the target system.

One may notice that the clustering criteria aggressively balance the computation or communication at the cost of unbalance the other. Although the algorithm can solve clustering in linear speed, adjustments must be made to generate realistic clusters. As a future work, we intend to create new criteria targeting the generation of more balanced graphs, minimizing both the distance (standard deviation) of edges and vertices of the application graph.

8. ORCA RT-TOOLS: SCHEDULING PERIODIC, REAL-TIME WORKFLOW APPLICATIONS IN NOC-BASED MANYCORE SYSTEMS USING THE ORT SUITE



This Chapter discusses the last of our three approaches for scheduling network traffic. It also accounts for CPU time and uses most of the tools presented before. The results from this part of the Thesis have yet to be published.

We organize the Chapter as follows. Section 8.1 presents the overview of our approach, presenting the building blocks of our approach. In Section 8.2, we present a proof of concept, evaluated in Section 8.3. Finally, Section 8.4 concludes the chapter while presenting the limitations of our approach.

8.1 The Approach

We propose an approach to deal with applications in which tasks and communication have real-time constraints. Handling computation and communication in the same approach is not novel [Khare et al., 2019, Benchehida et al., 2020]. However, the literature relies on hardware modules to control the time to inject packets into the network, leading to undesired area overhead and energy overhead. The main contribution to the Chapter is to show that we can perform RT analysis statically without modifying the underlying architecture. To achieve such a result, we use the tools presented in former chapters of this Thesis. Please note that the tools presented in past chapters automate their corresponding frameworks. Instead, the approach proposed in this chapter is not automated by any of the tools; thus, we had to collect results from the tools individually, doing part of the analysis manually. However, we included the automation of the approach as a future work.

When developing this approach, our first attempt would include computation and communication constraints in a single model to generate a single schedule. However, in the last chapters, we demonstrated that generating a network schedule via constraint models requires heuristics to handle the search complexity. Also, we demonstrated that a mechanism to enforce the behavior of the schedule is needed. On the one hand, the time that task schedulers take to switch tasks during the runtime is negligible. On the other hand, the NoC cannot switch flows from links unless it implements a mechanism such as VCs.

We relied on the distributed programming model property of manycores as an alternative to handle both computation and communication RT requirements. Even sharing

the same underlying network media, the processing elements of the many-core have individual task schedulers that handle a fraction of the application each. The core idea of our approach lies in predicting the time in which tasks will release packets into the network. If we can collect the release time of packets, we can also generate a valid network schedule considering a contention-free environment. The goal is to determine the minimum frequency so the system can schedule computation (tasks) and communication (flows). Figure 8.1 shows an overview of our approach and its building blocks.

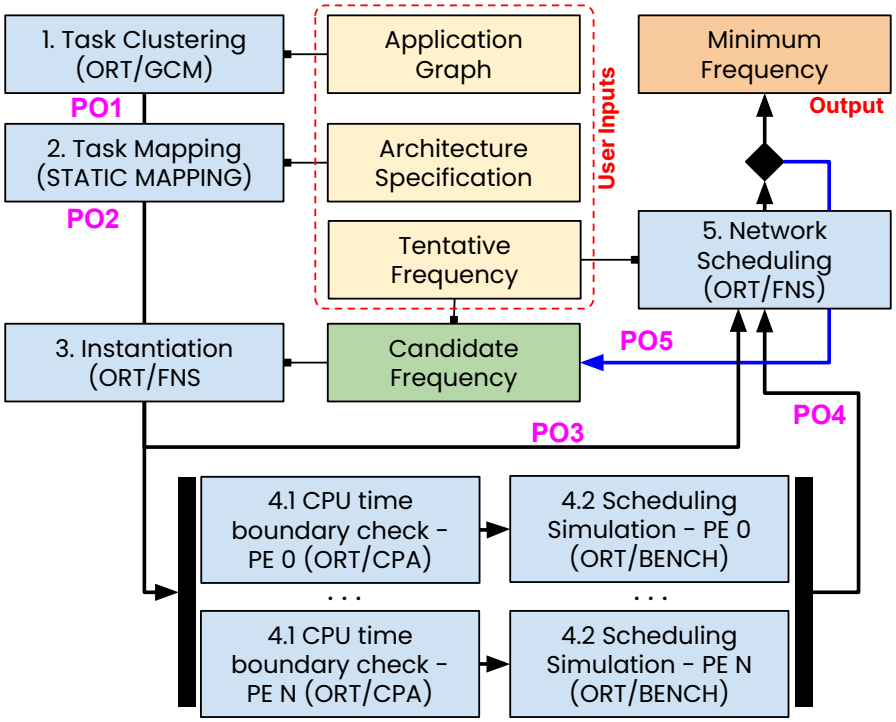


Figure 8.1 – The proposed real-time analysis framework for computation and communication. Blue shapes represent the building blocks (processes). Yellow shapes represent user inputs. The candidate frequency (green shape) changes during the execution of the workflow. Finally, the result of the workflow is the minimum frequency (orange shape). Purple labels (PO1-PO5) represent partial outcomes.

8.1.1 Task Clustering

We adopt the ORT/GCM tool to cluster tasks targeting the number of PEs of the baseline platform (Figure 8.1, process 1). The result of the process is an application graph whose number of vertices is equal to the number of PEs (PO1). To balance the CPU load between the PEs, we run the ORT/GCM tool using all the criteria discussed in Chapter 7, picking the result with the slightest standard deviation between vertices values. Due to the complexity of the GCM algorithm, the time to run all the criteria is negligible (see Section 8.2).

8.1.2 Task Mapping

Task mapping is a well-developed topic in the manycore literature [Singh et al., 2013]. One notorious application of task mapping in manycores is energy consumption savings [Maqsood et al., 2018, Quan and Pimentel, 2015], although other applications such as network throughput reduction, temperature reduction, and multiple resource optimization also apply. Task mapping becomes more challenging as the number of tasks, the number of processing elements, and the complexity of the mapping criteria increase, requiring heuristics to achieve solutions in a feasible time. Thus, most task mapping methods rely on genetic algorithms, integer linear programming, or simulated annealing search [Singh et al., 2013].

We map clusters to nodes after we merge vertices (tasks), reducing the number of mapping possibilities. By the end of the clustering process, the resulting graph has the number of vertices equal to the number of CPUs in the target system. In the worst case, a brute-force search would take up to $n!$ steps to find the optimal solution by exploring the whole search space, where n is the number of CPUs. For a 3×3 dimension manycore, it would take $9! = 3,628,80$ steps. We developed a custom heuristic algorithm called spiral mapping, which targets mesh-topological manycores. Exploring task mapping is beyond the scope of the Thesis, and there is no reason for practitioners to adopt the spiral algorithm. However, studies on the spiral algorithm may appear as a future work.

The ORT/SM tool implements the spiral mapping algorithm. The algorithm greedily selects task clusters whose communication load is the least. Then, it allocated that cluster as far as possible from the middle of the NoC. The “middle of the NoC” refers to the coordinates with the least variation from the middle value of axes. For example, the middle values of an 8×10 NoC are ($x = 4, y = 5$). Figure 8.1 shows the spiral algorithm applied to the CAR-APP (see Figure 8.2) on a 4×4 manycore. For the sake of readability, Table 8.1 shows the task clusters. In the example, the algorithm selects the cluster with the least volume (Figure 8.2, a). It repeats the same until it fills the outer border of the mesh (b-d). The behavior repeats until all clusters become mapped to the manycore (e-f). The algorithm performs $O(n)$ on the number of clusters, plus the cost of sorting clusters by communication volume.

Table 8.1 – Network volume of the clustered CAR application (MAX-PROC).

#	Cluster	Volume	#	Cluster	Volume
1	SPES	15360	9	FBU7:BFE7	51200
2	TPRC	2048	10	FBU8:BFE8	51200
3	THRC	20480	11	BFE3:FDF1	1164800
4	FBU1:BFE1	51200	12	FDF2:STPH	358400
5	FBU2:BFE2	51200	13	POSI:NAVI:DIRC	56832
6	FBU4:BFE4	51200	14	FBU3:VOD1:VOD2	972800
7	FBU5:BFE5	51200	15	USOS:OBDB:OBMG	141312
8	FBU6:BFE6	51200	16	VIBS:TPMS:STAC	22528

NOTE: Network volume is the sum of the weights of all incoming and outgoing edges of each cluster.

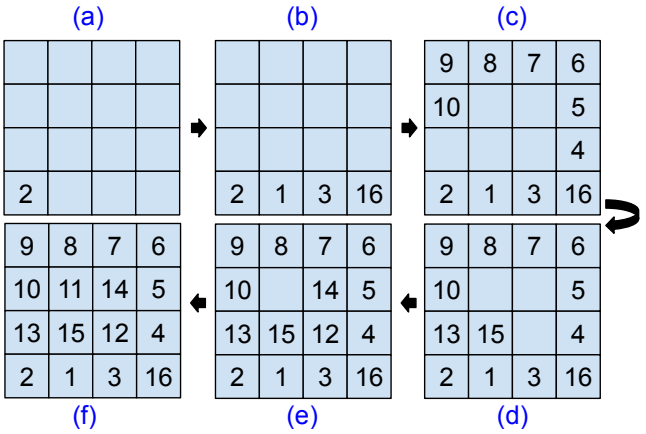


Figure 8.2 – Application of the the SM algorithm to the clustered CAR application (MAX-PROC). Numbers represent tasks and blue squares represent CPUs.

8.1.3 Instantiation

Our approach is iterative, similar to the approach in Chapter 6. At the beginning, the user informs a tentative frequency. After mapping tasks to the manycore, our approach requires an application instance targeting a candidate frequency (Figure 8.1, process 3). We use the tentative frequency value as the candidate frequency during the first iteration.

For the communication load, the instantiation of flows occur as describe in Chapter 6, except that we leave the minimum release time of flow as a variable (we assumed zero before). For the computation, our analysis fork into two possibilities: (i) tasks workload is describe as cycles, or (ii) tasks WCET is expressed in standard time units, e.g., seconds, milliseconds. In the former case, no modification is necessary, as the next processes treat time as cycles of the target architecture. In the latter case, we need to scale the application to the frequency of the system and adjust the number of cycles, i.e. $cycles = \frac{time}{frequency}$.

8.1.4 CPU time boundary analysis

This step performs schedulability tests on each task set produced by the clustering process. We perform the test using the ORT/GCPA (an implementation of the critical path analyses algorithm) and the ORT/BENCH tool, which we used for scheduling simulation in Chapter 4. We prefer to run the ORT/GCPA test first because it is faster than the ORT/BENCH test, although the order has no impact on the results. Also, there is no dependency on the tests, so one may run the entire test set in parallel. To simplify the automation of the step, we run tests sequentially.

ORT/GCPA - Graph Critical Path Analysis

The ORT/GCPA tool implements the critical path analysis method (CPA) [Kelley and Walker, 1959]. The goal of the tool is to find whether the system has enough CPU time to run the task sets. For this reason, we must apply the method for each task set. The CPA method calculates the critical path of a task set and appears in other contexts, e.g., logistics and manufacturing. In our Thesis, the method finds the minimum CPU time required to run a task set based on the WCET of tasks and the WCTT of communication.

The critical path of an application is the worst-case time to finish processing the last task in the workflow. Since CPUs carry only subsets of the application task set, we must account for internal communication (task-to-task communication through memory spaces) and external communication (through the NoC). Due to the capabilities of our network driver (see Chapter 3.1), internal communication always takes 10 cycles to occur. Please note that we model the number of cycles our network driver takes to move packets. If considering other platforms, one must account for the worst-case operation time of the driver.

The algorithm proceeds as Djkistra's algorithm for finding the shortest path between two nodes in a graph [Dijkstra, 1959], except that we multiply the weights of edges (communication load) by -1. Please note that we cannot apply topological sort to cyclic graphs, so the application graph must be a directed acyclic graph (DAG). To lift such a restriction, we use depth-first search (DFS) to find and remove cycles, adding the weight of the removed edge to all edges pointing to the vertex pointed by the cycle. Since cycles can only occur locally, we always add 10 cycles, i.e., the worst-case execution time of the NoC driver routine to move packets between applications in the same CPU. Since NoC driver operations cannot occur in parallel, we need to sum the value of vertices instead of choosing the one with the lowest value. Figure 8.3 shows an example of the adapted CPA method.

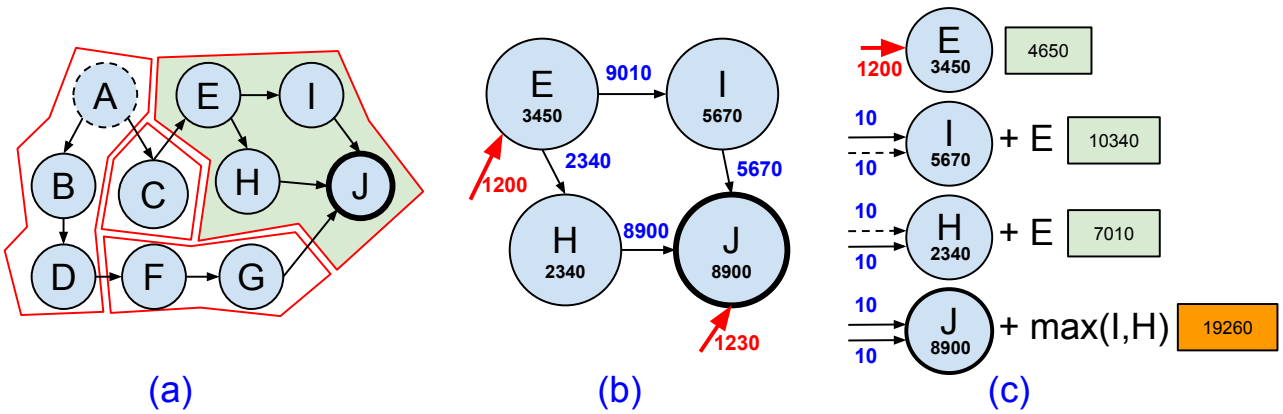


Figure 8.3 – A clustered application (a), the internal and external communication of a cluster (b), and the modified CPA method applied to that cluster (c). Red values represent external communication. Blue arrows represent internal communication with values transformed to 10. Indirect communication appear as dashed arrows. Green boxes represent the time to execute the task. Orange box is the critical path value for the cluster.

A final step in the method is calculating the CPU time spent by interruptions. In our baseline platform, two kinds of interruption concerns us: (i) task scheduling interruptions, and (ii) network interruptions. We perform the characterization on interruptions and the driver considering a worst-case scenario. Equation 8.1 gives the number of cycles required to run the task set (cluster), where $t_{cluster}$ is the execution time of the cluster, *epkts* is the number of incoming external packets, t_{driver_irq} is the amount of cycles to handle the driver interruption (incoming packets), t_{cpa} is the result of the CPA method (Figure 8.3), *sched_t* is the schedule period in cycles, and t_{sched_irq} is the schedule interruption processing time.

$$t_{cluster} = (epkts \times t_{driver\_irq}) + (t_{cpa}/sched\_t) \times t_{sched\_irq} \tag{8.1}$$

The $t_{cluster}$ value is the amount of cycles for the cluster to processing. We convert the application period, (e.g., Figure 6.2) to cycles, using the candidate frequency. The result is the amount of cycles necessary to run an application iteration, $t_{iteration}$. If $t_{cluster} < t_{iteration}$, the CPU can schedule the task set. Otherwise, the schedule is unfeasible.

ORT/BENCH - Extracting Packets Release Time from Simulated Scheduling

In Chapter 4, we proposed a scheduling simulator architecture named ORT/BENCH. The tool can simulate scheduling algorithms using discrete-event simulation, whose simulation speed surpasses the execution of actual kernels. Besides the performance, another advantage of adopting such a tool is configuring the execution time of system events. For instance, we can tune the ORT/BENCH tool to simulate multiple system events while setting an upper-bound execution time for them. In other words, the performance of the tool will remain the same regardless of the length of events. We can use the ORT/BENCH tool in parallel to the ORT/GCPA tool, as previously suggested. The ORT/GCPA tool implements an analytic model, and the performance of the model is bound to the size of the input system. Oppositely, the performance of the ORT/BENCH tool is bound to the number of events under simulation. Nevertheless, increasing the frequency of scheduling interruption in the system does not affect ORT/GCPA performance, although increasing the number of events for simulations in ORT/BENCH degrades simulation performance.

To use the ORT/BENCH tool, we need to characterize the system a priori. The parameters are the same as for the ORT/GCPA method (*epkts*, t_{drive_irq}, $sched_t$, and t_{sched_irq}), except for t_{cpa}. To add, we need to know the scheduling algorithm and the characterization of tasks, e.g., WCET model or PCD description, — period, capacity, and deadline. After the simulation, the ORT/BENCH tool indicates the number of cycles taken to process the entire cluster, i.e. $t_{cluster}$. Once again, if $t_{cluster} < t_{iteration}$, the schedule is feasible; otherwise, unfeasible.

8.1.5 Network Scheduling

Due to the voluntary preemption mechanism implemented by our network driver (see Section 3.2.3), we enforce the behavior of tasks to follow the Logical Execution Time (LET) model [Gemlau et al., 2021]. For this reason, we can use the ORT/BENCH to collect the time in which tasks release packets in the network. The release time of packets is a requirement for this step of our approach, so using ORT/BENCH to simulate scheduling is mandatory. We use the LET model to enforce the behavior of tasks, thus allowing us to predict the access to the memory. In our framework, we adapt the concept to predict the behavior of packets. Thus, application tasks can only start processing after receiving all required packets. Analogously, tasks send packets at the end of their processing phase.

8.2 Proof of Concept: The Synthetic Application E

We demonstrate our approach on the Synthetic Application E, shown by Figure 8.4. We scaled the application in Chapter 7 to make the application CPU dominated, increasing the WCET of tasks 100 times and the size of packets 10 times.

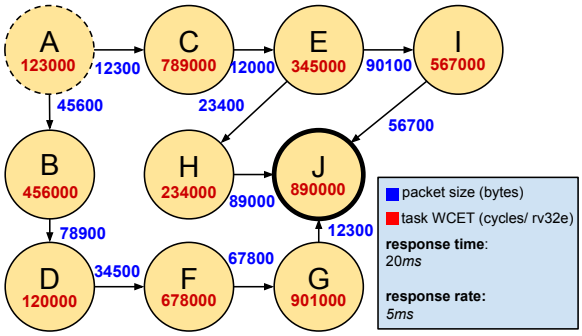


Figure 8.4 – The Synthetic Application E.

8.2.1 Task Clustering and Task Mapping

We perform task clustering on the Synthetic Application targeting a 2×2 instance of our baseline platform. For the criteria, we adopt the same strategy we described in Section 8.1.1, where we run the ORT/GCM for all the criteria presented in Chapter 7. Figure 8.5 shows the results of the clustering process for the Synthetic Application E. We select the clustering criterion whose result represents the slightest standard deviation between the total WCET of clusters, i.e., the most balanced CPU load.

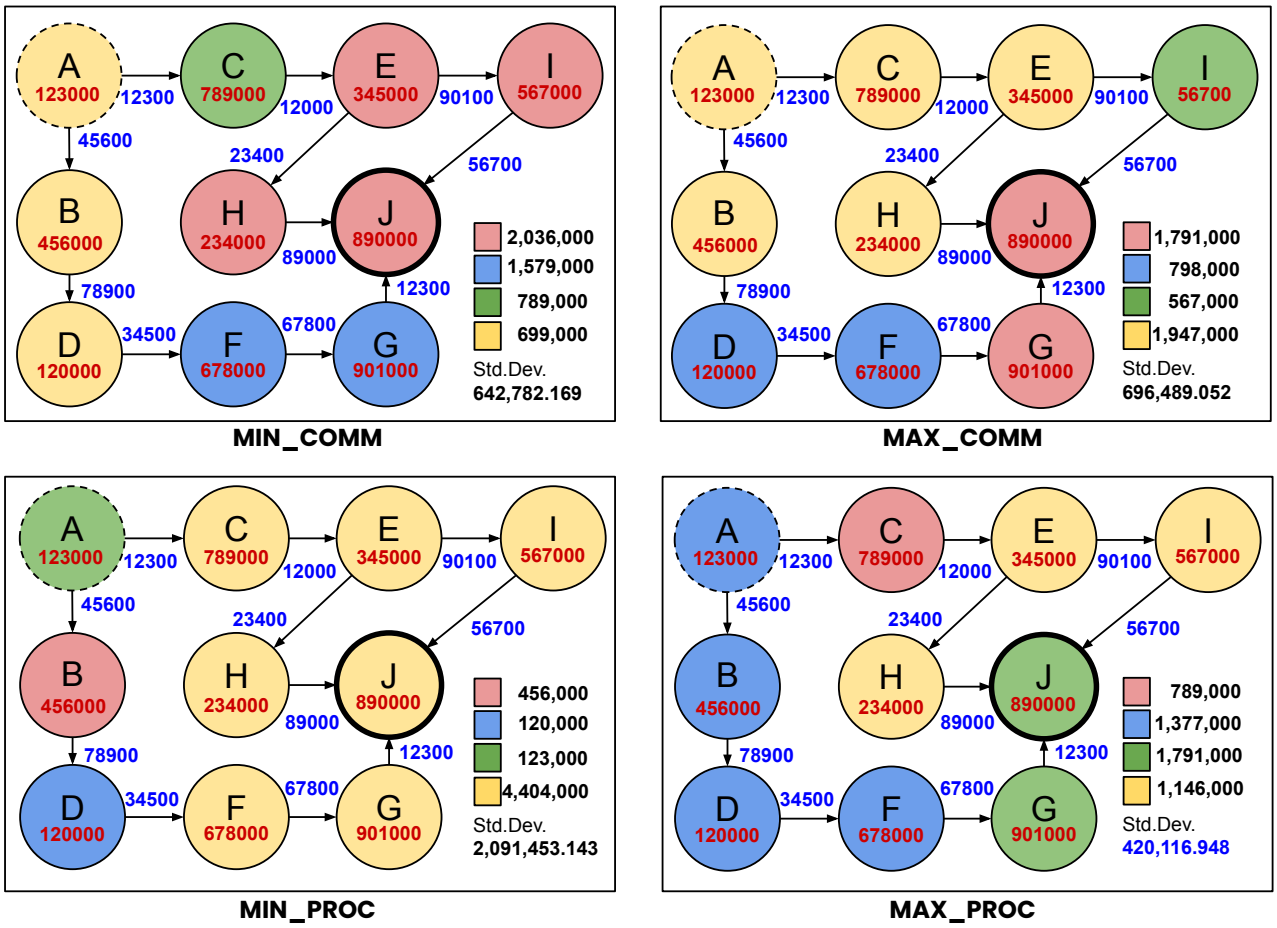


Figure 8.5 – Clustering of the Synthetic Application E for the MIN_COMM, MAX_COMM, MIN_PROC, and MAX_PROC criteria. Task colors represent the clusters. Legend shows the sum of CPU load for each cluster, followed by the standard deviation CPU load.

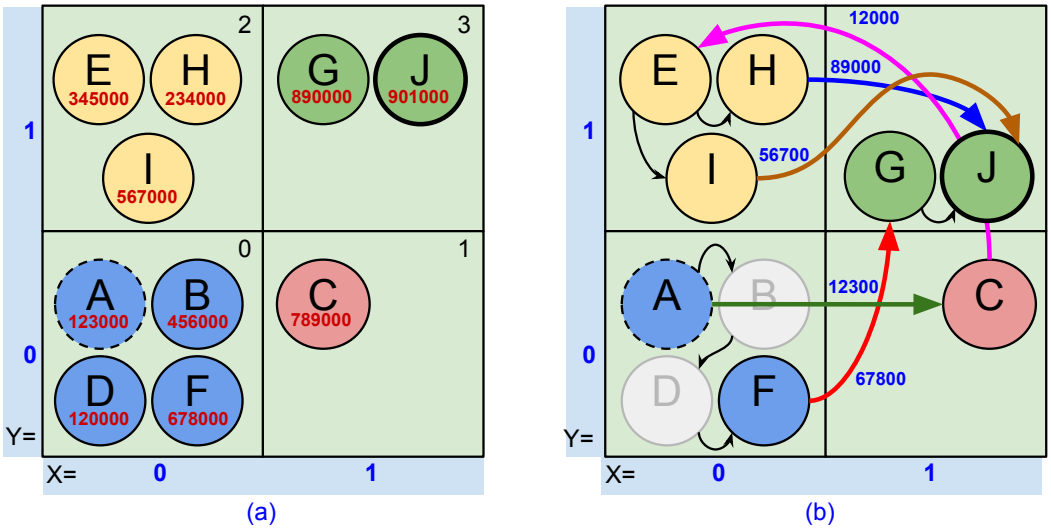


Figure 8.6 – The Synthetic Application E mapped to the ORCA2 manycore (a), and communication partitioning in internal and external contexts (b). Arrows indicate the communication direction. Black arrows indicate internal communication, while external communication is denoted by colored arrows. Black-and-white tasks do not communicate with external tasks.

We map the selected clustering (Figure 8.5, bottom-right) to our manycore using the SM algorithm. The map is as Figure 8.6 shows. We name clusters on the tasks they include; the application has the following clusters: ABDF, C, EHI, and GJ. Cluster C is a trivial example of a cluster, including only the task C. Cluster ABDF has the most tasks, although cluster GJ has the most CPU load (peak CPU time requirement of $1,791,000$ cycles). The total network load per iteration is $237,800$ bytes (communication between tasks from different clusters), representing 45% of total system communication.

8.2.2 Task Instantiation and Searching for the Minimum Frequency

The number of iterations our approach takes to find the least suitable system frequency depends on two factors. First, the larger the difference between the tentative and final frequencies, the more steps the approach takes. The performance of our binary-search-like algorithm (see Section 6.4) is $O(log\ n)$. Due to knowing the behavior of the application a priori, we informed $1.5Ghz$ as the tentative frequency, hoping to shrink the size of the demonstration presented herein. Second, due to the behavior of the binary search algorithm, the algorithm may not converge without a tolerance factor. For this application, we set a tolerance factor of 1%; the algorithm will converge once it finds a candidate frequency with less than 1% difference from the lowest frequency found so far. Figure 8.7 demonstrates the algorithm results.

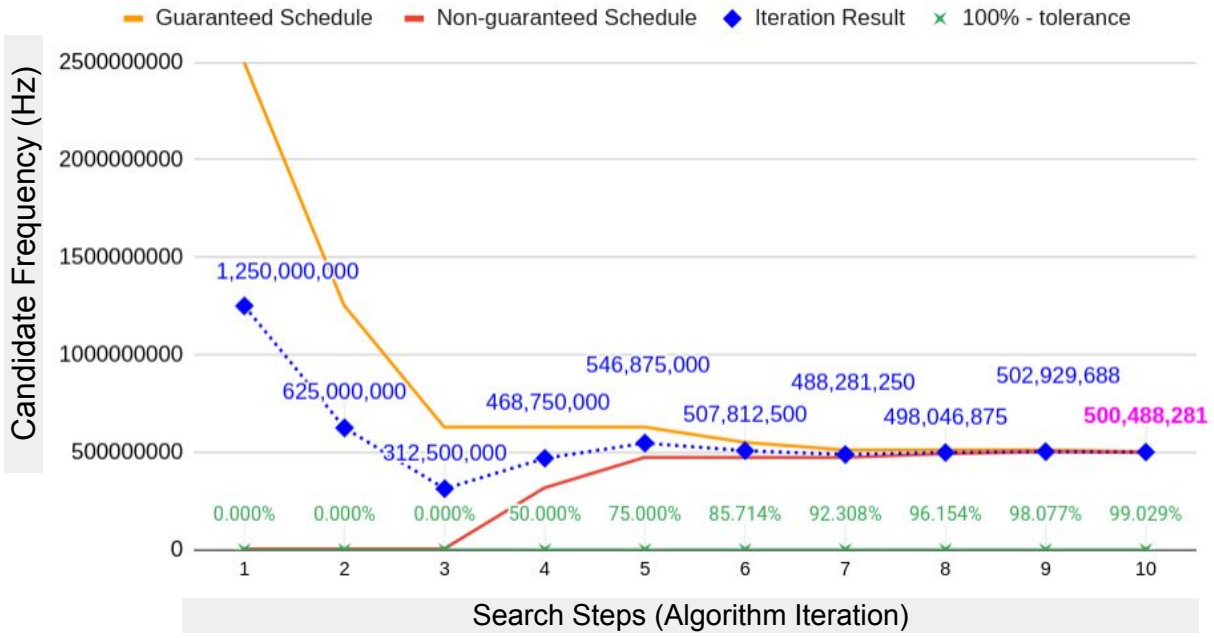


Figure 8.7 – Results of our binary-search-like algorithm. The tentative frequency is 1.5GHz. The algorithm converges at step 10, finding 500.488MHz as the final frequency.

In the proposed pre-runtime approach, the algorithm starts with the nominal frequency (1.5GHz) of the platform. The goal is to adjust the platform frequency to minimize power consumption while meeting RT constraints. If scheduling at the nominal frequency proves infeasible, the algorithm considers alternative clustering options. In scenarios where no viable solution emerges, the application is deemed unsuitable for execution on the current platform. In this case, the designer may use a larger platform to satisfy the constraints.

8.2.3 ORT/GCPA and the CPA method

We use the ORT/GCPA to find the required number of cycles per iteration. In the case of the Synthetic Application E, cluster GJ demands the most CPU time. Due to all CPUs running in parallel, the GJ cluster becomes the bottleneck of the system, limiting the number of iterations from the CPU perspective. Figure 8.8 synthesizes the results of the GCPA analysis. Our tool indicates that cluster GJ requires at least 2, 174, 500 cycles to run one iteration with a CPU load of 86.98%. Note that, at this point, our analysis extends to each CPU. We discuss the interaction between tasks of different clusters later in Section 8.2.4.

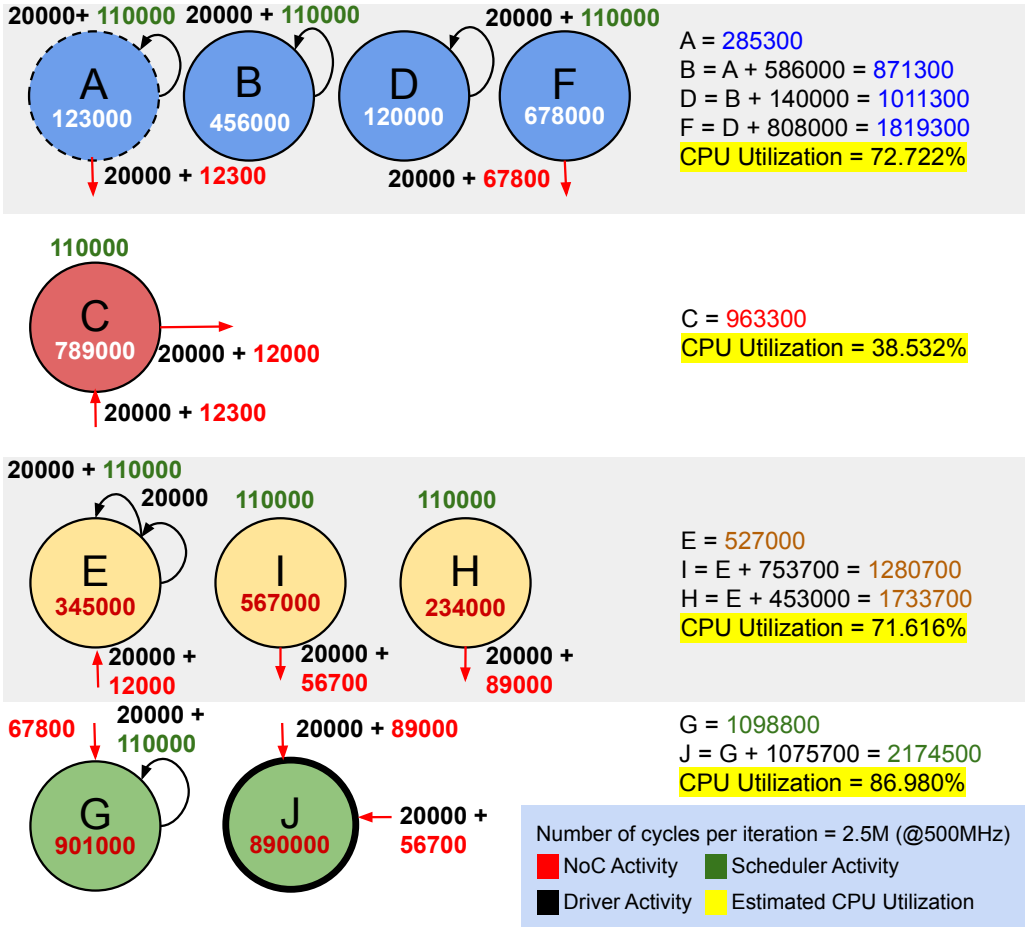


Figure 8.8 – The CPA method applied to the Synthetic Application E.

We used the following process to characterize the kernel. First, instead of counting instructions from the assembly, we inserted counters in the hardware to capture kernel events. We introduced macros in the source code that unwraps to assembly code. The assembly code has an overhead of two instructions. Given our simplifications regarding CPU time, we understand that accounting for such cycles in our analysis is unnecessary. We use the counters in two situations: (i) to characterize the number of instructions of the network driver and scheduler; and (ii) to capture the scheduling events during the RTL simulation.

The result of our characterization is as follows. Scheduling takes at most $110,000$ cycles to occur. We account for clock interruptions, dispatcher time, context switching, and the scheduling algorithm, considering the worst-case execution time of each part. Our network driver consumes $20,000$ cycles to send packets and $20,000$ cycles to receive packets (assuming the DMA has already transferred them to the memory). Both values are rough simplifications of the behavior of dynamic memory allocation (i.e., *malloc*), which consumes most of the time in both operations.

ORT/BENCH and Task Scheduling Simulation

We use the ORT/BENCH tool to simulate the behavior of clusters and assert their schedulability. The characterization of interruptions is the same as that of the ORT/GCPA tool. We simulate for 5ms (application response rate), i.e., $2,500,000$ instructions. Figure 8.9 shows the results of the simulation.

Since we provided the exact characterization for the ORT/GCPA and ORT/BENCH tools, the number of cycles computed by both tools is the same. Besides, the workload of most tasks is shorter than the scheduling tick, i.e., 5ms, resulting in a fixed number of calls to the scheduler during each application iteration. Task execution follows a round-robin algorithm because task scheduling uses data dependency to determine the next task to run. The system will schedule a task only if all data dependencies are met, i.e., received necessary packets.

A noteworthy feature of our baseline platform is voluntary preemption. When a task misses any data dependency, it puts itself in a blocked state to save CPU time through voluntary preemption. Although of utility in practice, this feature is challenging to simulate. For ORT/BENCH and ORT/GCPA, we added a worst-case arrival time to tasks to mimic the behavior of a late packet. Considering that the application will fail if one of the clusters misses their deadline, we consider that dependencies will always meet their deadlines, and the cluster will start processing once the input data is available. In the case of Synthetic Application E, tasks from cluster ABDF will execute one after another in pipeline-fashion execution, taking at most 20ms to execute. However, the response time rate of the application is 5ms.

Given that the application follows the LET model [Gemlau et al., 2021], tasks only release packets into the network at the end of their execution. We collect the time the schedule switches applications from the ORT/Bench tool simulation results and subtract the network driver time. Then, we use the resulting value as the minimum release time of packets in the ORT/FNS tool.

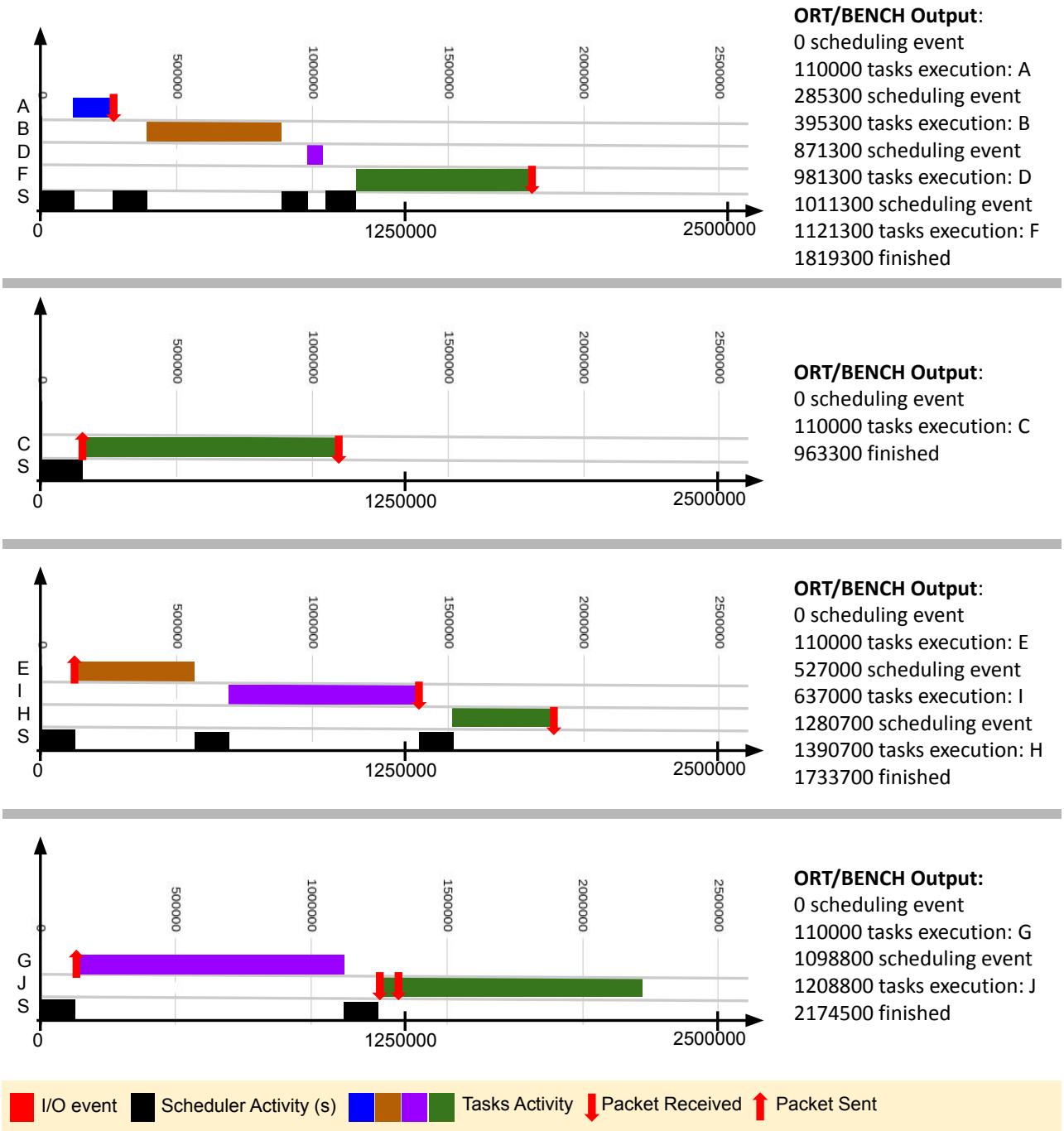


Figure 8.9 – ORT/Bench simulation results for the Sythetic Application E, including the simulation log (right) and the simulation trace (left).

8.2.4 Network Scheduling

As we demonstrate in Chapter 6, the ORT/FNS tool can find network schedules given the characterization of the network traffic using the parameters: (i) minimum release time, (ii) packet length in bytes, and (iii) a deadline value. Table 8.2 shows the parameters for the Synthetic Application E.

- We can obtain the length of packets from the application graph. Assumptions on the network are the same as in chapters 5 and 6.
- We set the flow deadline to the time the destination task is scheduled (from ORT/BENCH) minus the network driver time. From the resulting value, we subtract the packet occupancy (same as in Chapter 5).
- The minimum release time of packet, which we once assumed equals zero, now comes from the ORT/BENCH simulation. We take the time that tasks leave the CPU as the worst-case release time.

Table 8.2 – Characterization of flows for the Synthetic Application E.

Flow	Minimum Release Time	Source	Destination	Volume	Deadline
F_1	$285300 + (phase \times pv)$	A	C	12300	$110000 + (phase \times pv)$
F_2	$1819300 + (phase \times pv)$	F	G	67800	$110000 + (phase \times pv)$
F_3	$963300 + (phase \times pv)$	C	E	12000	$110000 + (phase \times pv)$
F_4	$1280700 + (phase \times pv)$	I	J	56700	$1208800 + (phase \times pv)$
F_5	$1733700 + (phase \times pv)$	H	J	89000	$1208800 + (phase \times pv)$

pv: phase time (5ms).

We must also adjust the release times and deadlines, accounting for temporal dependency between tasks. For instance, task C is scheduled simultaneously with task A in a different CPU (see Figure 8.9), so a packet cannot reach C from A in time. To fix the problem, we introduce a parameter named *phase*, representing the period in which a task waits for previous tasks in the workflow to finish. Since the scheduler only elects task C to run once each 5ms (schedule interruption time), we set the application phase to 5ms. In practice, the kernel will set the status of the receiving task to READY as soon as a packet is received. However, the task enters the CPU only in the next scheduling event. A phase of 5ms corresponds to $2,500,000$ cycles given a 500MHz frequency. Figure 8.10 illustrates the concept of a phase.

The Synthetic Flow E application has 4 phases. In practice, due to the missing data dependencies, the scheduler will not schedule the last task in the graph (output, J) to run in less than three phases. In phase 1, the scheduler of CPU 0 executes tasks A, B, D, and F

in order. While task B enters the CPU, the network transfers a packet from task A to C. The packet will arrive before the end of the phase, flagging task C as READY. However, the CPU will schedule task C only at the following scheduling event, i.e., at the beginning of the next phase. The same occurs for all clusters. The message from task C arrives at CPU 1 before the end of the phase, although the scheduler will schedule task E only at the beginning of the next phase. The packet from task F to G is ready by the end of the first phase, and task G will enter the CPU only in the next phase. Task C entered the CPU in phase 2, and Task E entered in phase 3. Finally, packets from tasks H and I will be ready by the end of phase 3. Task J enters the CPU in phase 4, the last phase.

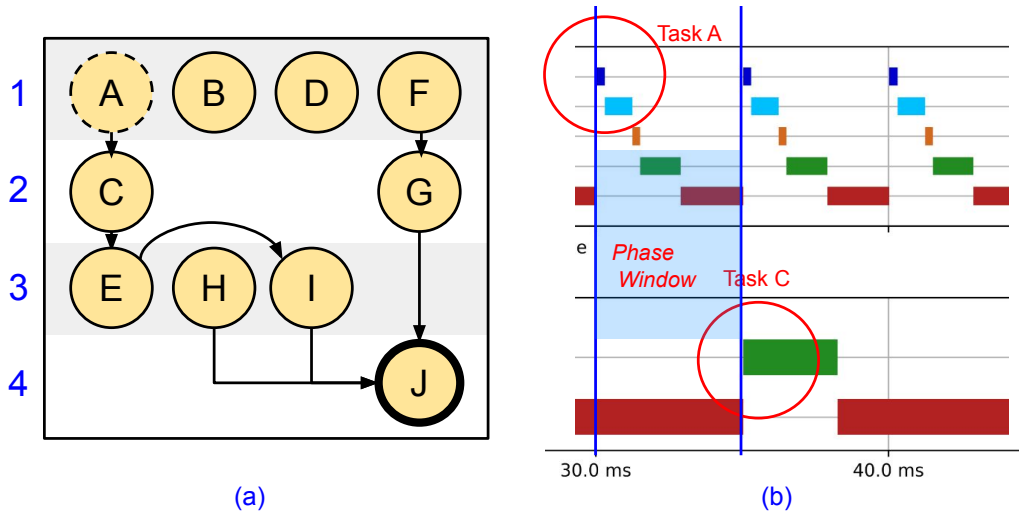


Figure 8.10 – Phases of the Synthetic Application E (a) and an example of phase (b), collected from RTL simulation. Blue lines represent scheduling activity (amortized).

8.3 Approach Evaluation

We evaluate our approach by simulating our baseline platform at the register-transfer level (RTL). We use ModelSim 2021.1 on a computer Intel® Xeon® E3-1220 v5 CPU @3.00GHz with 32GB RAM. The RTL project mixes VHDL and SystemVerilog languages. Additional configurations of the project include:

- A 2ns clock period, i.e. the system runs at 500MHz.
- We artificially release task A iterations each 5ms, following the application expected response rate of 200 iterations per second.
- Due to the configuration of the network driver at the startup, we delayed the execution of the first iteration of task A to occur at 30ms (warm-up time).

- We set the number of scheduling interruptions per second to 200, i.e., a 5ms task execution window. The scheduling algorithm selects tasks based on their data dependency; only tasks that meet data dependencies can enter the CPU.

We could capture the scheduling behavior during the RTL simulation with the counters we used during kernel characterization. A memory-mapped register stores the identifier of the running tasks (i.e., an integer number). Once the schedule changes the running task, the register captures the value from the software. Figure 8.11 shows the results of the simulation for the Synthetic Application E. In the Figure, tasks receive colors according to their identifier. The kernel adds the IDLE task to occupy CPU time when no task can run, having the identifier 0. The rest of the tasks receive numbers from one up to the limit of tasks in the system.

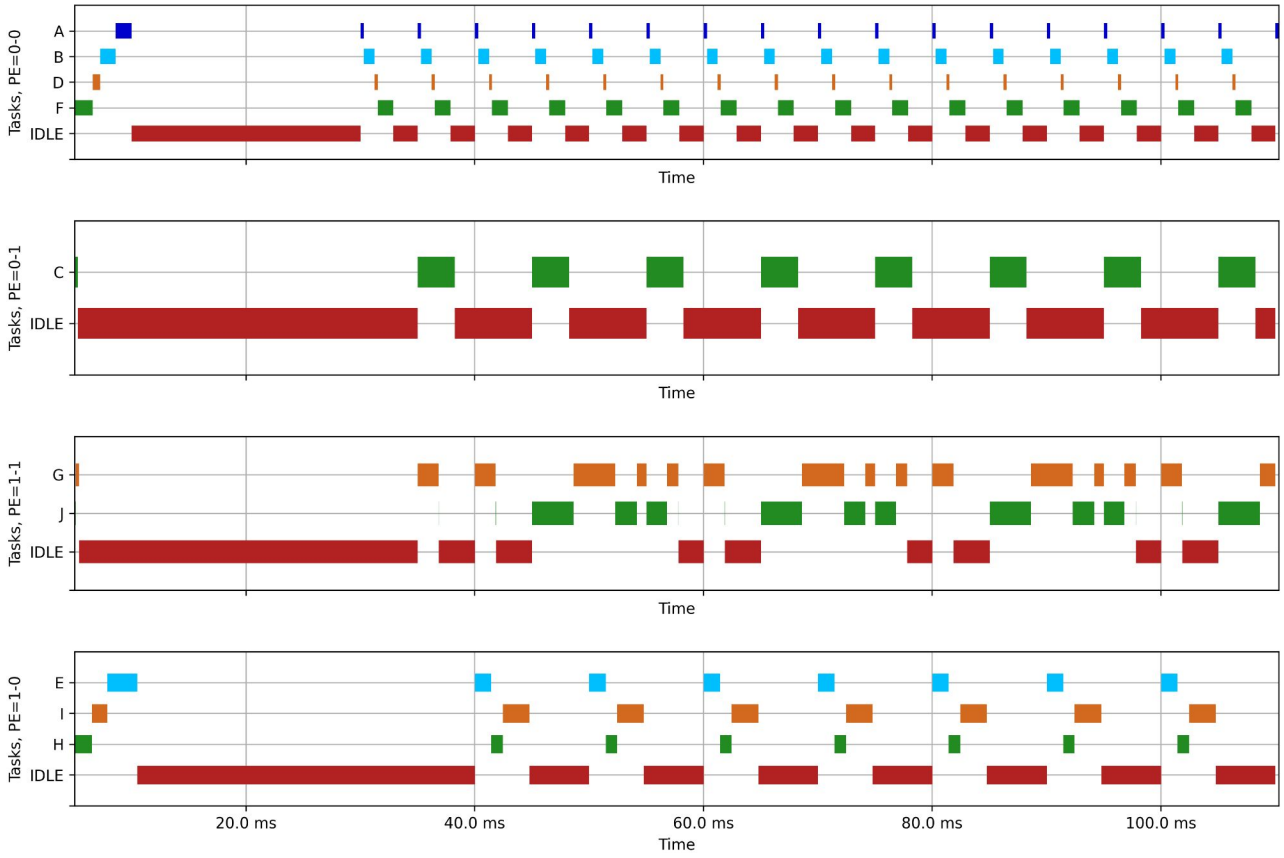


Figure 8.11 – Results of the RTL simulation for the Synthetic Application E. Colors follow tasks identifiers, where 0 is red, 1 is green, 2 is orange, 3 is light blue, and 4 is blue. Lanes represent the CPUs of the manycore.

The system will only schedule the task IDLE if no other task can run. A task can only run by meeting its data dependencies. Task A is a particular case, as its run condition is bound to time (5ms), which we artificially control by hardware. In the case of cluster ABDF, we observe that all tasks run in the same order as the task graph. The last task to run is Task F, leaving the rest of the CPU time to the IDLE task. Cluster ABDF executed in $1,439,000$

cycles. Our analysis indicated a requirement of 1,819,300 cycles, 126% of the actual value ($\simeq$ 26% overestimation). Table 8.3 presents the results for all clusters.

Table 8.3 – Results of RTL simulation for Synthetic Application E (CPU).

Cluster	Execution Time*	Estimation**	Difference	Difference %
ABDF	1,439,000	1,819,300	+380,300	+26.428%
C	801,300	963,300	+162.000	+20.217%
EHI	2,174,500	2,667,895	+493,395	+22.690%
GJ	2,384,400	3,014,400	+630.000	+26.421%

* Execution time in cycles, measured from RTL simulation in ModelSim.

** Estimated execution time in cycles, taken from ORT/GCPA tool.

NOTE 1: “Difference %” takes execution time as 100%.

Regarding CPU time, our approach overestimated the execution time of all clusters by an average of 23.934%. This overestimation represents a waste of $\simeq$ 24% of CPU time in all CPUs. Cluster ABDF had the worst estimation, representing $\simeq$ 26% of CPU time waste. However, please note that our worst estimation is within the limits of CPU utilization suggested by the literature, where a CPU utilization of $\geq$ 70% is often considered questionable [Laplante and Ovaska, 2011].

Figure 8.12 shows the phases of Synthetic Application E. We assert phases by collecting the beginning and end times of iterations, using the same hardware we used to collect the execution time of tasks. The hyperperiod of the application is 20ms, with the four iterations in the hyperperiod having a different behavior (e.g. iteration 4 took 3,994,080 cycles to complete). For instance, task C processes packets from the first and second iterations in the same phase. Since task C will be preempted only at the end of the phase, it checks for the queue and finds the packet from the second iteration. The same occurs in cluster EIH, where task E received two packets from task C. Task J enters the CPU with one packet in the queue, receiving a second packet during the processing. Task J processes iterations one and two after another, leaving the CPU afterward. Later, the task processes iteration 3, leaving the CPU once more. The task processes the later iteration while entering the CPU the third time. During the hyperperiod, iterations 1 and 3 took 16.793ms and 16.438ms to run. iterations 2 and 4 took 12.799ms and 12.486ms to run. None of the iterations took more than the 20ms response time limit. As no deadline violation occurred, the application has successfully met its real-time requirements.

8.4 Final Remarks

In this chapter, we propose an approach for asserting the real-time properties of applications, approaching computation and communication. The proposed approach relies

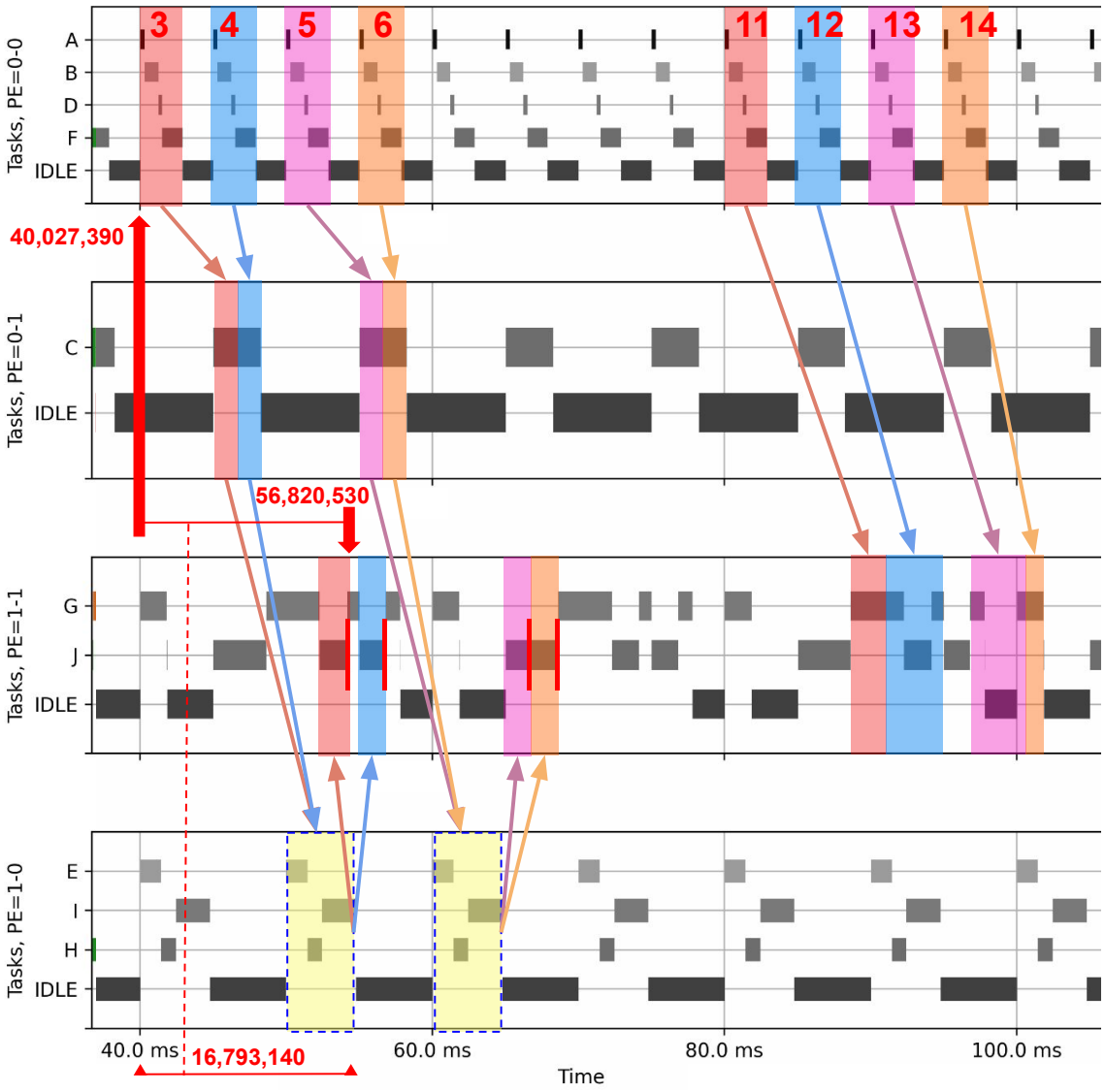


Figure 8.12 – Phases of the Synthetic Application E, collected from the RTL simulation. Colored rectangles represent phases of the same iteration. Interaction between tasks F and G is showed at the right for the sake of readability (phases 9 to 12). Other interactions appear at the left (phases 3 to 6). Arrows indicate the direction of the workflow. The yellow rectangle represents a phase overlapping. Phase 3 took 16, 793, 140 cycles to complete.

on tools proposed in earlier chapters of the Thesis. We demonstrate our approach for a synthetic application, which could guarantee both response time and response rate. We evaluate our approach using RTL simulation and our baseline manycore.

Our approach is novel as it approaches computation and communication without requiring modifications to the system and applies to general-purpose NoCs. We did not intervene with the manycore at the hardware level, preserving the same baseline platform described in Chapter 3. For the software, the only modification we made concerns the controlled execution of task A, guaranteeing its execution rate of 5ms. In a real-world system, the rate of the task would be controlled by the input, e.g., a sensor or other source from outside the system.

One drawback of our application is the lack of automation, which prevented us from experimenting with more applications. However, we want to use a single tool to group the tools presented throughout this Thesis in the future. We also want to apply our approach to real-world applications. Unfortunately, larger applications are rarely found in the literature.

9. CONCLUSION

This Thesis presented an approach for guaranteeing real-time properties of applications in manycores, approaching computation and communication aspects of the system. To develop a final approach, we had to develop an initial approach from scratch, as presented in Chapter 5, evolving the approach several times until we achieved a more comprehensive framework (Chapter 8). During the development of the Thesis, we employed techniques that included system simulation (Chapter 4), constraint programming (Chapter 5), and problem optimization (Chapter 6). By the end of the Thesis, we mixed all these techniques in a final approach (Chapter 8), which we could demonstrate using the ORCA2 manycore as the baseline platform (Chapter 3).

9.1 Discussion

In Chapter 1, we introduced the Thesis by questioning whether we could guarantee the real-time properties of an application, approaching both communication and computation aspects of the system, without making incursions into the manycore architecture. The questioning was propelled by a gap in the literature, whose approaches (Chapter 2) rely on specific features of the underlying hardware. The approaches in the literature were too specific and rarely applicable to other systems but the ones discussed in their demonstrations. Although unlikely to meet the requirements of all possible manycores, our approach has no hardware requirements that bind it to a single platform and is generic enough to be applied to a wide range of systems. Consequently, we can perform RT analysis on general-purpose NoCs despite properties such as routing algorithm, buffer size, and switching, requiring such properties to be abstracted in the RT model.

We achieved all the secondary goals of the Thesis. First, we could perform RT analysis on a manycore whose hardware has no native support for RT (subgoal 1). As we do not modify the hardware of the system, we understand that the area and energy requirements of the design are preserved. We programmed a suite of tools — ORT — to automate most of the analysis, except for a part of the approach in Chapter 8 (subgoal 4, partial fulfillment). The models used within the tools serve no specific system and have room for modifications, requiring no modifications to the tools in most cases (subgoal 2). Our approach is scalable due to the custom search method we developed in Chapter 6 (subgoal 3). Lastly, our tools and the baseline platform are open hardware and open source, requiring no commercial technology to run (subgoal 5).

9.2 Lessons Learned and Future Works

1. We initially proposed the ORT/BENCH tool as an academic tool, but later, we repurposed the tool to simulate our system as part of the herein-proposed approach. While experimenting with the tool, we struggled to model some characteristics of our system, notably the behavior of packets and the effect of interruption in the schedule. We solved the issue by characterizing the system, assuming a pessimistic worst-case execution time for the software components. As a future work, we would like to improve the tool, mainly providing support for the voluntary preemption mechanism of our manycore. In addition, we would like to experiment with a compiled model as an alternative to the discrete-event model, making the tool closer to the execution of an actual kernel.

2. One advantage of the constraint model from our initial approach is its optimization accuracy. Although the performance of ILP solvers is poor compared to our AGS algorithm, our constraints can deliver cycle-accurate network analysis. Automating the constraint generation from a higher-level model could help analyze smaller parts of the system, e.g., buses. The AGS algorithm, implemented in the ORT/FNS tool, overcomes the performance problem of constraint models. Unfortunately, we should have modeled the CPU using constraints. If we include the CPU models in the constraint set, an alternative for the analysis of Chapter 8 could be another heuristic-based tool, replacing both the ORT/GCPA and ORT/BENCH tool in the approach.

3. The graph tools in Chapter 7 are a promising approach for clustering tasks in manycores. However, the lack of a criterion to contemplate computation and computation makes the resulting graph unbalanced. In future work, we want to explore the algorithm, electing edges to remove from the graph based on other decision-making algorithms, e.g., simulated annealing.

4. The ORCA2 platform, used as the baseline platform in our studies, could be better integrated with the tools we developed. To make our approach independent of the platform, we collected data from the system through simulation logs, special registers in the architecture, and software output. We intend to create interfaces to ease runtime data gathering from the platform. One option could be to use SystemVerilog interfaces to avoid tampering with the hardware.

5. We intend to perform a systematic review of the literature to document and discuss the references of Chapter 2. As far as we know, the last review on RT NoCs dates from 2017 [Hesham et al., 2017] (almost seven years as of March 2024). Besides including newer studies in the review, we intend to include other studies on RT directed to manycores. The approaches presented in this Thesis, for instance, have the same purpose of RT NoCs, although they approach the problem from a different perspective.

6. Finally, we can improve the approach presented in Chapter 8 in several ways. One possible direction could be narrowing the scope of the approach, contemplating a smaller set of systems, e.g., only packet switching NoCs. However, one of the goals of the Thesis is the development of a generic approach. To keep the approach as generic as possible, we intend to work on a formal way to describe the approach inputs. Another future work includes the componentization of the approach, allowing fast switching of components specific to certain NoC features. For example, two components could be used in the same part of the workflow, one for PS networks and another one for CS networks. Another opportunity would be exploring the reuse of components to create different tools from a single tooling platform. One of the goals of creating custom tools is to reduce the waste of the system resources. Although our tool achieved $< 30\%$ of CPU waste, we can improve the results through model specialization. As for the NoC, a network utilization waste is acceptable as long as the application is computation-dominated like most of the applications herein discussed.

9.3 Final Thoughts

This thesis explored the domain of manycore systems, focusing on realizing real-time application requirements. We presented the evolution of our approach while discussing its applicability in a low-cost manycore. Our approach is novel, as other approaches in the literature require specific hardware on target platforms. Although requiring some polishing, the approach allowed for exploring RT analysis at the pre-runtime without overfitting our models to specific platforms. We plan to keep exploring the field in the future, helping to promote manycores in the hardware-software, critical-mission, and embedded applications domain.

REFERENCES

[1X Technologies, 2023] 1X Technologies (2023). NEO by 1X Technologies | Your Intelligent Android Assistant. Online: https://www.1x.tech/androids/neo.

[Accellera, 2024] Accellera (2024). Ocp (open core protocol). Online: https://www.accellera.org/downloads/standards/ocp.

[Ackerman, 2020] Ackerman, E. (2020). Autonomous Robots Are Helping Kill Coronavirus in Hospitals. Online: https://spectrum.ieee.org/automaton/robotics/medical-robots/autonomous-robots-are-helping-kill-coronavirus-in-hospitals.

[Al-Kadi and Terechko, 2009] Al-Kadi, G. and Terechko, A. S. (2009). A hardware task scheduler for embedded video processing. In Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., and Ungerer, T., editors, *High Performance Embedded Architectures and Compilers*, pages 140–152, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ali and Zakarya, 2020] Ali, A. and Zakarya, M. (2020). *Power and Performance Based Hybrid Multiprocessing for Real Time Scheduling*. PhD thesis, International Islamic University, Islamabad.

[Asgarieh and Lin, 2019] Asgarieh, Y. and Lin, B. (2019). Smart-hop arbitration request propagation: Avoiding quadratic arbitration complexity and false negatives in smart nocs. *ACM Transactions on Design Automation of Electronic Systems*, 24(6).

[AUTOSAR, 2021] AUTOSAR (2021). Standards – AUTOSAR. Online: https://www.autosar.org/standards/.

[Avani and Agrawal, 2018] Avani, P. and Agrawal, S. (2018). Efficient Dynamic Virtual Channel Architecture for NoC Systems. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2502–2507.

[Azumi et al., 2020] Azumi, T., Maruyama, Y., and Kato, S. (2020). ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4375–4382.

[Babar, 2012] Babar, Y. (2012). Understanding linux process states. Online: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[Balakrishnan and Ozguner, 1998] Balakrishnan, S. and Ozguner, F. (1998). A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):664–678.

[BBC News, 2020a] BBC News (2020a). Coronavirus: Robot dog enforces social distancing in Singapore park. Online: https://www.bbc.com/news/av/technology-52619568.

[BBC News, 2020b] BBC News (2020b). How robots have joined the battle against coronavirus. Online: https://www.bbc.com/news/av/technology-52105478.

[Benchehida et al., 2020] Benchehida, C., Benhaoua, M. K., Zahaf, H.-E., and Lipari, G. (2020). Task and communication allocation for real-time tasks to networks-on-chip multiprocessors. In *2nd International Conference on Embedded & Distributed Systems (EDiS)*, pages 9–14.

[Benini and De Micheli, 2002] Benini, L. and De Micheli, G. (2002). Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 418–419.

[Benno, 2024] Benno, J. (2024). Orb_kprofiller repository at github. Online: https://github.com/bennoXav/ORB_KProfiller.

[Berger, 2020] Berger, A. S., editor (2020). *Debugging Embedded and Real-Time Systems*. Newnes.

[Berreby, 2020] Berreby, D. (2020). The pandemic has been good for one kind of worker: robots. Online: https://www.nationalgeographic.com/science/article/how-pandemic-is-good-for-robots.

[Bertozzi and Benini, 2004] Bertozzi, D. and Benini, L. (2004). Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31.

[Bjerregaard and Sparso, 2005] Bjerregaard, T. and Sparso, J. (2005). A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Design, Automation Test in Europe Conference (DATE)*, pages 1226–1231 Vol. 2.

[Bolotin et al., 2004] Bolotin, E., Cidon, I., Ginosar, R., and Kolodny, A. (2004). QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2):105–128.

[Boston Dynamics, 2020] Boston Dynamics (2020). Spot(R). Online: https://www.bostondynamics.com/spot.

[Brandner and Schoeberl, 2012] Brandner, F. and Schoeberl, M. (2012). Static Routing in Symmetric Real-Time Network-on-Chips. In *International Conference on Real-Time and Network Systems (RTNS)*, page 61–70.

[Buttazzo, 2011] Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition.

[Casile et al., 1998] Casile, A., Buttazzo, G., Lamastra, G., and Lipari, G. (1998). A scheduling simulator for real-time distributed system. *IFAC Workshop on Distributed Computer Control Systems*, 31(32):161–167.

[Cellan-Jones, 2020] Cellan-Jones, R. (2020). Uber's self-driving operator charged over fatal crash. Online: https://www.bbc.com/news/technology-54175359.

[Chakaravarthy et al., 2021] Chakaravarthy, R. V., Kwon, H., and Jiang, H. (2021). Vision Control Unit in Fully Self Driving Vehicles using Xilinx MPSoC and Opensource Stack. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 311–317.

[Charles and Mishra, 2020] Charles, S. and Mishra, P. (2020). Securing Network-on-Chip Using Incremental Cryptography. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 168–175.

[Chen, 2023] Chen, C. (2023). Amazon reveals first photos of the new prime air delivery drone. Online: https://www.aboutamazon.com/news/transportation/amazon-prime-air-drone-delivery-mk30-photos.

[Chen et al., 2021] Chen, P., Liu, W., Chen, H., Li, S., Li, M., Yang, L., and Guan, N. (2021). Reduced worst-case communication latency using single-cycle multihop traversal network-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(7):1381–1394.

[Chen and Jha, 2016] Chen, X. and Jha, N. K. (2016). Reducing wire and energy overheads of the smart noc using a setup request network. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3013–3026.

[Chitchian et al., 2013] Chitchian, M., van Amesfoort, A. S., Simonetto, A., Keviczky, T., and Sips, H. J. (2013). Adapting Particle Filter Algorithms to Many-Core Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 427–438.

[da Silva et al., 2019] da Silva, E. A., Kreutz, M. E., and Zeferino, C. A. (2019). Redscarf: an open-source multi-platform simulation environment for performance evaluation of networks-on-chip. *Journal of Systems Architecture*, 99:101633.

[Dahule et al., 2023] Dahule, S. K., Golhar, R. V., and Ramteke, M. D. (2023). The behavior of round robin arbiter in noc architecture. *International Journal of Engineering and Innovative Technology (IJEIT)*, 3.

[Daoud, 2018] Daoud, L. (2018). Secure Network-on-Chip Architectures for MPSoC: Overview and Challenges. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 542–543.

[De Liz Bomer et al., 2023] De Liz Bomer, R. V., Zeferino, C. A., Seman, L. O., and Leithardt, V. R. Q. (2023). Worst-case communication time analysis for on-chip networks with finite buffers. *IEEE Access*, 11:25120–25131.

[DeepMind, 2020] DeepMind (2020). RT-2: New model translates vision and language into action. Online: https://www.deepmind.com/blog/rt-2-new-model-translates-vision-and-language-into-action.

[Demers et al., 1989] Demers, A., Keshav, S., and Shenker, S. (1989). Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4).

[Diemer and Ernst, 2009] Diemer, J. and Ernst, R. (2009). A link arbitration scheme for quality of service in a latency-optimized network-on-chip. In *Design, Automation Test in Europe Conference (DATE)*, pages 574–577.

[Diemer et al., 2010] Diemer, J., Ernst, R., and Kauschke, M. (2010). Efficient throughput-guarantees for latency-sensitive networks-on-chip. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 529–534.

[Diguet, 2014] Diguet, J.-P. (2014). Self-Adaptive Network On Chips. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, page 6.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

[Diligent Robotics, 2021] Diligent Robotics (2021). Moxi. Online: https://diligentrobots.com/moxi.

[Doan and Tanaka, 2019] Doan, D. and Tanaka, K. (2019). Adaptive local assignment algorithm for scheduling soft-aperiodic tasks on multiprocessors. In *IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6.

[Domingues et al., 2021] Domingues, A. R., Benno, J., Amory, A. M., and Moraes, F. G. (2021). Orca rt-bench: A reference architecture for real-time scheduling simulators. In *XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6. IEEE.

[Domingues et al., 2022a] Domingues, A. R., Johann Filho, S., Amory, A. D. M., and Moraes, F. G. (2022a). Design-time analysis of real-time traffic for networks-on-chip using

constraint models. In *35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. IEEE.

[Domingues et al., 2022b] Domingues, A. R., Sergio Filho, J., Amory, A. D. M., Ost, L., and Moraes, F. G. (2022b). Design-time scheduling of periodic, hard real-time flows for noc-based systems. In *29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4. IEEE.

[Domingues, 2020] Domingues, A. R. P. (2020). ORCA: A Self-Adaptive, Multiprocessor System-on-Chip Platform. Master's thesis, Programa de Pós-Graduação em Ciência da Computação. PPGCC/PUCRS. Online: http://tede2.pucrs.br/tede2/handle/tede/9344.

[Fishman, 2001] Fishman, G. S. (2001). *Discrete-Event Simulation – Modeling, Programming and Analysis*. Springer Science+Business Media New York, 1st edition.

[Forbes, 2023] Forbes (2023). A Short History Of ChatGPT: How We Got To Where We Are Today. Online: https://www.forbes.com/sites/bernardmarr/2023/05/19/a-short-history-of-chatgpt-how-we-got-to-where-we-are-today/?sh=63bb5e46674f.

[FreeRTOS, 2024] FreeRTOS (2024). Freertos task states and state transitions described. Online: https://www.freertos.org/RTOS-task-states.html.

[Galster, 2015] Galster, M. (2015). Software reference architectures: related architectural concepts and challenges. In *1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures (CobRA)*, pages 1–4.

[Gecode, 2024] Gecode (2024). Generic constraint development environment. Online: https://www.gecode.org/.

[Gemlau et al., 2021] Gemlau, K.-B., KÖHLER, L., Ernst, R., and Quinton, S. (2021). System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Transactions on Cyber-Physical Systems*, 5(2).

[Glass and Ni, 1992] Glass, C. J. and Ni, L. M. (1992). The Turn Model for Adaptive Routing. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 278–287, New York, NY, USA. ACM.

[Global Lighting Association, 2021] Global Lighting Association (2021). Position Statement on Germicidal UV-C Irradiation. Online: https://www.globallightingassociation.org/images/files/publications/GLA_UV-C_Safety_Position_Statement.pdf.

[Goldberg and Tarjan, 1988] Goldberg, A. V. and Tarjan, R. E. (1988). A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940.

[González et al., 2022] González, Y. R., Nelissen, G., and Tovar, E. (2022). Ipden: Real-time deflection-based noc with in-order flits delivery. In *IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 160–169.

[Goossens et al., 2005a] Goossens, K., Dielissen, J., Gangwal, O., Pestana, S., Radulescu, A., and Rijpkema, E. (2005a). A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Design, Automation Test in Europe Conference (DATE)*, pages 1182–1187.

[Goossens et al., 2005b] Goossens, K., Dielissen, J., and Radulescu, A. (2005b). AEthereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421.

[Grot et al., 2009] Grot, B., Keckler, S. W., and Mutlu, O. (2009). Preemptive Virtual Clock: A Flexible, Efficient, and Cost-Effective QOS Scheme for Networks-on-Chip. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 68–279.

[GSPD Lab, 2021] GSPD Lab (2021). :: RTsim Homepage. Online: https://www.dcce.ibilce.unesp.br/spd/rtsim/english/index.php.

[Guo et al., 2020] Guo, P., Hou, W., Guo, L., Cao, Z., and Ning, Z. (2020). Potential Threats and Possible Countermeasures for Photonic Network-on-Chip. *IEEE Communications Magazine*, 58(9):48–53.

[Göhringer et al., 2011] Göhringer, D., Oey, O., Hübner, M., and Becker, J. (2011). Heterogeneous and runtime parameterizable Star-Wheels Network-on-Chip. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 380–387.

[Haggui et al., 2018] Haggui, O., Tadonki, C., Lacassagne, L., Sayadi, F., and Ouni, B. (2018). Harris corner detection on a NUMA manycore. *Future Generation Computer Systems*, 88:442–452.

[Hansson et al., 2007a] Hansson, A., Coenen, M., and Goossens, K. (2007a). Undisrupted Quality-of-Service during Reconfiguration of Multiple Applications in Networks on Chip. In *Design, Automation Test in Europe Conference (DATE)*, pages 1–6.

[Hansson et al., 2007b] Hansson, A., Goossens, K., and Rădulescu, A. (2007b). A Unified Approach to Mapping and Routing on a Network-on-Chip for Both Best-Effort and Guaranteed Service Traffic. *VLSI Design*, 2007:68432:1–68432:16.

[Hansson et al., 2009] Hansson, A., Subburaman, M., and Goossens, K. (2009). Aelite: A flit-synchronous Network on Chip with composable and predictable services. In *Design, Automation Test in Europe Conference (DATE)*, pages 250–255.

[Harmanci et al., 2005] Harmanci, M. D., Escudero, N. P., Leblebici, Y., and Ienne, P. (2005). Quantitative modelling and comparison of communication schemes to guarantee quality-of-service in networks-on-chip. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1782–1785 Vol. 2.

[Hartuv and Shamir, 2000] Hartuv, E. and Shamir, R. (2000). A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4):175–181.

[He et al., 2014] He, H., Yang, G., and Hu, J. (2014). Algorithms for power-efficient QoS in application specific NoCs. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 165–170.

[Heisswolf et al., 2013] Heisswolf, J., König, R., Kupper, M., and Becker, J. (2013). Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603 – 2622.

[Heisswolf et al., 2013] Heisswolf, J., Singh, M., Kupper, M., König, R., and Becker, J. (2013). Rerouting: Scalable NoC self-optimization by distributed hardware-based connection reallocation. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–8.

[Hesham et al., 2017] Hesham, S., Rettkowski, J., Goehringer, D., and Abd El Ghany, M. A. (2017). Survey on Real-Time Networks-on-Chip. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1500–1517.

[Hyojeong Song et al., 1997] Hyojeong Song, Boseob Kwon, and Hyunsoo Yoon (1997). Throttle and preempt: a new flow control for real-time communications in wormhole networks. In *International Conference on Parallel Processing (ICPP)*, pages 198–202.

[Instor – Projetos e Robótica, 2020] Instor – Projetos e Robótica (2020). JACI. Online: https://www.instor.com.br/projeto/9.

[ISO, 2020] ISO (2020). ISO 26262:9 — Automative Safety Integrity Level (ASIL). Online: https://www.iso.org/obp/ui/#iso:std:iso:26262:-9:ed-2:v1:en.

[Jantsch et al., 2005] Jantsch, A., Lauter, R., and Vitkowski, A. (2005). Power analysis of link level and end-to-end data protection in networks on chip. In *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1770–1773 Vol. 2.

[Jeon et al., 2021] Jeon, J., Jung, S., Lee, E., Choi, D., and Myung, H. (2021). Run Your Visual-Inertial Odometry on NVIDIA Jetson: Benchmark Tests on a Micro Aerial Vehicle. *IEEE Robotics and Automation Letters*, 6(3):5332–5339.

[Johann, 2024a] Johann, S. (2024a). Hellfireos realtime operating system. Online: https://github.com/sjohann81/hellfireos.

[Johann, 2024b] Johann, S. (2024b). Hf-risc soc. Online: https://github.com/sjohann81/hf-risc.

[Johann, 2024c] Johann, S. (2024c). Kprofiler. Online: https://github.com/sjohann81/hellfireos/tree/master/usr/kprofiler.

[Johann, 2024d] Johann, S. (2024d). Ucx/os - microcontroller executive / os. Online: https://github.com/sjohann81/ucx-os.

[Jong-Pyng Li and Mutka, 1994] Jong-Pyng Li and Mutka, M. W. (1994). Priority based real-time communication for large scale wormhole networks. In *International Parallel Processing Symposium (IPPS)*, pages 433–438.

[Juracy, 2022] Juracy, L. R. (2022). *A framework for fast architecture exploration of convolutional neural network accelerators*. PhD thesis, School of Technology. Online: https://tede2.pucrs.br/tede2/handle/tede/10437.

[Kalray, 2024] Kalray (2024). Mppa (r) dpu coolidge (tm). Online: https://www.kalrayinc.com/products/processors-many-core/.

[Kapre, 2016] Kapre, N. (2016). Marathon: Statically-scheduled conflict-free routing on fpga overlay nocs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 156–163.

[Kapre and Gray, 2015] Kapre, N. and Gray, J. (2015). Hoplite: Building austere overlay nocs for fpgas. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8.

[Kasapaki et al., 2016] Kasapaki, E., Schoeberl, M., Sørensen, R., Müller, C., Goossens, K., and Sparsø, J. (2016). Argo: A real-time network-on-chip architecture with an efficient gals implementation. *I E E E Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492.

[Kelley and Walker, 1959] Kelley, J. E. and Walker, M. R. (1959). Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), page 160–173, New York, NY, USA. Association for Computing Machinery.

[Kerrisk, 2021] Kerrisk, M. (2021). sched(7) — linux manual page. Online: https://man7.org/linux/man-pages/man7/sched.7.html.

[Khare et al., 2019] Khare, A., Boben, I. A., and Chattopadhayay, S. (2019). Enhanced schedulability via minimal routing with mapping and priority assignment for real-time network-on-chip. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pages 564–568.

[Koenen et al., 2019] Koenen, M., Doan, N. A. V., Wild, T., and Herkersdorf, A. (2019). A hybrid noc enabling fail-operational and hard real-time communication in mpsoc. In Schoeberl, M., Hochberger, C., Uhrig, S., Brehm, J., and Pionteck, T., editors, *Architecture of Computing Systems – ARCS 2019*, pages 31–44, Cham. Springer International Publishing.

[Koenen et al., 2020] Koenen, M., Doan, N. A. V., Wild, T., and Herkersdorf, A. (2020). Exploring task and channel mapping strategies in fail-operational and hard real-time nocs. In *IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7.

[Kofman et al., 2019] Kofman, E., Muzu, A., and Zeigler, B. (2019). *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Elsevier Academic Press, 3rd edition.

[Kumar et al., 2002] Kumar, S., Jantsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., and Hemani, A. (2002). A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 117–124.

[Laplante and Ovaska, 2011] Laplante, P. A. and Ovaska, S. J. (2011). *Real-Time Operating Systems*, chapter 1-3, pages 79–147. John Wiley & Sons, Ltd.

[Li et al., 2019] Li, J., Zheng, G., Zhang, H., and Shi, G. (2019). Task scheduling algorithm for heterogeneous real-time systems based on deadline constraints. In *9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pages 113–116.

[Li et al., 2016] Li, X., Duraisamy, K., Bogdan, P., Majumder, T., and Pande, P. P. (2016). Network-on-chip-enabled multicore platforms for parallel model predictive control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(9):2837–2850.

[Liu et al., 2012] Liu, S., Jantsch, A., and Lu, Z. (2012). Parallel probing: Dynamic and constant time setup procedure in circuit switching NoC. In *Design, Automation Test in Europe Conference (DATE)*, pages 1289–1294.

[Manacero et al., 2001] Manacero, A., Miola, M. B., and Nabuco, V. A. (2001). Teaching real-time with a scheduler simulator. *Frontiers in Education Conference. Impact on Engineering and Science Education*, pages 15–19.

[Maqsood et al., 2018] Maqsood, T., Tziritas, N., Loukopoulos, T., Madani, S. A., Khan, S. U., Xu, C.-Z., and Zomaya, A. Y. (2018). Energy and communication aware task mapping for mpsocs. *Journal of Parallel and Distributed Computing*, 121:71–89. Online: https://www.sciencedirect.com/science/article/pii/S074373151830217X.

[Mello et al., 2009] Mello, A., Calazans, N., and Moraes, F. (2009). *QoS in Networks-on-Chip – Beyond Priority and Circuit Switching Techniques*, pages 1–22. Springer US.

[Mesidis and Indrusiak, 2011] Mesidis, P. and Indrusiak, L. S. (2011). Genetic mapping of hard real-time applications onto NoC-based MPSoCs — A first approach. In *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–6.

[Michael R. Garey, 1979] Michael R. Garey, D. S. J. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, first edition edition.

[Microsoft, 2023] Microsoft (2023). ChatGPT for Robotics: Design Principles and Model Abilities. Online: https://www.microsoft.com/en-us/research/group/autonomous-systems-group-robotics/articles/chatgpt-for-robotics/.

[Millberg et al., 2004a] Millberg, M., Nilsson, E., Thid, R., and Jantsch, A. (2004a). Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation Test in Europe Conference (DATE)*, pages 890–895 Vol.2.

[Millberg et al., 2004b] Millberg, M., Nilsson, E., Thid, R., Kumar, S., and Jantsch, A. (2004b). The Nostrum backbone-a communication protocol stack for Networks on Chip. In *International Conference on VLSI Design (VLSID)*, pages 693–696.

[MiniZinc Team, 2024] MiniZinc Team (2024). MiniZinc. Online: https://www.minizinc.org/.

[Minozzo et al., 2017] Minozzo, L., Rufino, J., and Lima, J. (2017). Experiences on object tracking using a many-core embedded system. In *International Conference on Applied Computing (AC)*, pages 195–204.

[Moraes et al., 2004] Moraes, F. G., Calazans, N., Mello, A., Moller, L., and Ost, L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration*, 38(1):69–93.

[Nithya and Itapu, 2023] Nithya, N. and Itapu, S. (2023). Design of low area interconnect architecture for cpu-gpu network-on-chips (nocs). In *2023 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–5.

[NIUS, 2020a] NIUS (2020a). AIS 5G The First and No.1 in Thailand. Online: https://www.ais.co.th/.

[NIUS, 2020b] NIUS (2020b). Alexia, el robot camarero anticontagio que atiende la terraza de un bar de Pamplona. Online: https://www.niusdiario.es/sociedad/camarero-robot-alexia-bar-pamplona_18_2956020165.html.

[Nürnberger et al., 2014] Nürnberger, S., Drescher, G., Rotta, R., Nolte, J., and Schröder-Preikschat, W. (2014). Shared Memory in the Many-Core Age. In *European Conference on Parallel Processing (Euro-Par)*, pages 351–362.

[OpenAI, 2023] OpenAI (2023). ChatGPT. Online: https://chat.openai.com/.

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

[PAL Robotics, 2020] PAL Robotics (2020). ARI. Online: https://pal-robotics.com/robots/ari/.

[Paul et al., 2014] Paul, J., Stechele, W., Kröhnert, M., and Asfour, T. (2014). Resource-Aware Programming for Robotic Vision. *CoRR*, abs/1405.2908.

[Perret et al., 2016] Perret, Q., Maurere, P., Noulard, E., Pagetti, C., Sainrat, P., and Triquet, B. (2016). Temporal isolation of hard real-time applications on many-core processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11.

[Picornell et al., 2019] Picornell, T., Flich, J., Hernández, C., and Duato, J. (2019). DCFNoC: A Delayed Conflict-Free Time Division Multiplexing Network on Chip. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Picornell et al., 2020] Picornell, T., Flich, J., Jose, D., and Hernández, C. (2020). HP-DCFNoC: High Performance Distributed Dynamic TDM Scheduler Based on DCFNoC Theory. *IEEE Access*, 8:194836–194849.

[Psarras et al., 2015] Psarras, A., Seitanidis, I., Nicopoulos, C., and Dimitrakopoulos, G. (2015). PhaseNoC: TDM scheduling at the virtual-channel level for efficient network traffic isolation. In *Design, Automation Test in Europe Conference (DATE)*, pages 1090–1095.

[Quan and Pimentel, 2015] Quan, W. and Pimentel, A. D. (2015). A hybrid task mapping algorithm for heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 14(1). Online: https://doi.org/10.1145/2680542.

[Rao and Sarma, 2017] Rao, B. N. and Sarma, N. M. (2017). A survey on QOS routing techniques used in network on chip. In *International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 3018–3024.

[Renfa et al., 2008] Renfa, L., Yan, L., and Cheng, X. (2008). A Survey of Task Scheduling Research Progress on Multiprocessor System-on-Chip. *Journal of Computer Research and Development*, 45(9).

[RETIS Lab, 2013] RETIS Lab (2013). RTSim Home Page. Online: http://rtsim.sssup.it/.

[Rhodes et al., 2021] Rhodes, D., Dick, R., and Vallerio, K. (2021). TGFF. Online: https://robertdick.org/projects/tgff/index.html.

[Ribot González and Nelissen, 2020] Ribot González, Y. and Nelissen, G. (2020). HopliteRT*: Real-Time NoC for FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3650–3661.

[Ripoll and Ballester-Ripoll, 2013] Ripoll, I. and Ballester-Ripoll, R. (2013). Period Selection for Minimal Hyperperiod in Periodic Task Systems. *IEEE Transactions on Computers*, 62(09):1813–1822.

[RISC-V International, 2023] RISC-V International (2023). Specifications - RISC-V International. Online: https://riscv.org/technical/specifications/.

[Robotics, 2020] Robotics, I. (2020). INF Robotics. Online: https://infrobotics.com/.

[Rocha et al., 2016] Rocha, A., Silva, C., Sørensen, R. B., Sparsø, J., and Schoeberl, M. (2016). Avionics applications on a time-predictable chip-multiprocessor. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 777–785.

[Rouxel and Puaut, 2017] Rouxel, B. and Puaut, I. (2017). STR2RTS: Refactored StreamIT benchmarks into statically analyzable parallel benchmarks for WCET estimation & real-time scheduling. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 1:1–1:12.

[Saeedi et al., 2018] Saeedi, S., Bodin, B., Wagstaff, H., Nisbet, A., Nardi, L., Mawer, J., Melot, N., Palomar, O., Vespa, E., Spink, T., Gorgovan, C., Webb, A., Clarkson, J., Tomusk, E., Debrunner, T., Kaszyk, K., Gonzalez-De-Aledo, P., Rodchenko, A., Riley, G., Kotselidis, C., Franke, B., O'Boyle, M. F., Davison, A. J., Kelly, P. H. J., Luján, M., and Furber, S. (2018). Navigating the Landscape for Real-Time Localization and Mapping for Robotics and Virtual and Augmented Reality. *Proceedings of the IEEE*, 106(11):2020–2039.

[Sandamirskaya, 2022] Sandamirskaya, Y. (2022). Rethinking computing hardware for robots. *Science Robotics*, 7(67):eabq3909.

[Schoeberl et al., 2012] Schoeberl, M., Brandner, F., Sparsø, J., and Kasapaki, E. (2012). A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *ACM/IEEE International Symposium on Networks-on-Chips (NOCS)*, pages 152–160.

[Seidita et al., 2021] Seidita, V., Lanza, F., Pipitone, A., and Chella, A. (2021). Robots as intelligent assistants to face COVID-19 pandemic. *Briefings in Bioinformatics*, 22(2):823–831.

[Shi and Burns, 2008] Shi, Z. and Burns, A. (2008). Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *ACM/IEEE International Symposium on Networks-on-Chips (NOCS)*, pages 161–170.

[Shi et al., 2010] Shi, Z., Burns, A., and Indrusiak, L. (2010). Schedulability analysis for real time on-chip communication with wormhole switching. *IJERTCS*, 1:1–22.

[Siemens, 2024] Siemens (2024). Modelsim hdl simulator. Online: https://eda.sw.siemens.com/en-US/ic/modelsim/.

[Singh et al., 2013] Singh, A. K., Shafique, M., Kumar, A., and Henkel, J. (2013). Mapping on multi/many-core systems: survey of current and emerging trends. In *DAC '13: Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA. Association for Computing Machinery. Online: https://doi.org/10.1145/2463209.2488734.

[Stefan et al., 2012] Stefan, R., Molnos, A., Ambrose, A., and Goossens, K. (2012). A TDM NoC supporting QoS, multicast, and fast connection set-up. In *Design, Automation Test in Europe Conference (DATE)*, pages 1283–1288.

[Stefan et al., 2014] Stefan, R. A., Molnos, A., and Goossens, K. (2014). dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594.

[Stiliadis and Varma, 1996] Stiliadis, D. and Varma, A. (1996). Design and Analysis of Frame-Based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks. *SIGMETRICS Perform. Eval. Rev*, 24(1):104–115.

[Stoer and Wagner, 1997] Stoer, M. and Wagner, F. (1997). A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591.

[Sørensen, 2016] Sørensen, R. B. (2016). *Time-Predictable Communication on a Time-Division Multiplexing Network-on-Chip Multicore*. PhD thesis, Technical University of Denmark (DTU).

[Taylor et al., 2007] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2007). *Software Architecture: Foundations, Theory and Practice*, page 119. Addison-Wesley.

[TechCrunch, 2019] TechCrunch (2019). A first look at amazon's new delivering drone. Online: https://techcrunch.com/2019/06/05/a-first-look-at-amazons-new-delivery-drone/.

[Temi USA, 2020] Temi USA (2020). Temi — The Personal Robot. Online: https://www.robotemi.com/.

[Tergel Molom-Ochir, 2020] Tergel Molom-Ochir (2020). Autonomous Driving and Its Sensor Technology The Levels of Autonomous Driving. Online: https://orise.orau.gov/resources/stem/documents/scholarships/autonomous-driving-and-its-sensor-technology.pdf.

[Tillenius et al., 2015] Tillenius, M., Larsson, E., Badia, R. M., and Martorell, X. (2015). Resource-aware task scheduling. *ACM Trans. Embed. Comput. Syst.*, 14(1).

[Ueter et al., 2020] Ueter, N., Chen, J.-J., von der Brüggen, G., Venkataramani, V., and Mitra, T. (2020). Simultaneous progressing switching protocols for timing predictable real-time network-on-chips. In *IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10.

[UVD Robots, 2024] UVD Robots (2024). UVD Robots. Online: https://www.uvd-robots.com/.

[Vancin et al., 2020] Vancin, P. H., Domingues, A. R. P., Paravisi, M., Johann, S. F., Calazans, N. L. V., and Amory, A. M. (2020). Towards an integrated software development environment for robotic applications in mpsocs with support for energy estimations. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

[Wasly et al., 2017] Wasly, S., Pellizzoni, R., and Kapre, N. (2017). HopliteRT: An efficient FPGA NoC for real-time applications. In *International Conference on Field-Programmable Technology (FPT)*, pages 64–71.

[Wassel et al., 2013a] Wassel, H. M. G., Gao, Y., Oberg, J. K., Huffmire, T., Kastner, R., Chong, F. T., and Sherwood, T. (2013a). SurfNoC: A Low Latency and Provably Non-Interfering Approach to Secure Networks-on-Chip. *ACM SIGARCH Computer Architecture News*, 41(3).

[Wassel et al., 2013b] Wassel, H. M. G., Gao, Y., Oberg, J. K., Huffmire, T., Kastner, R., Chong, F. T., and Sherwood, T. (2013b). SurfNoC: A Low Latency and Provably Non-Interfering Approach to Secure Networks-on-Chip. In *International Symposium on Computer Architecture (ISCA)*, pages 583–594.

[Waterman and Asanović, 2017] Waterman, A. and Asanović, K. (2017). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Online: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[WeRobotics, 2021] WeRobotics (2021). Teaming up with Pfizer on New Cargo Drone Project. Online: https://blog.werobotics.org/2021/01/12/teaming-up-with-pfizer-on-new-cargo-drone-project/.

[WHO, 2023] WHO (2023). World Health Organization Director-General's opening remarks at the media briefing – 5 May 2023. Online: https://www.who.int/news-room/speeches/item/who-director-general-s-opening-remarks-at-the-media-briefing---5-may-2023.

[Wiklund and Dake Liu, 2003] Wiklund, D. and Dake Liu (2003). SoCBUS: switched network on chip for hard real time embedded systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 8 pp.–.

[Williams, 2006] Williams, R., editor (2006). *Real-Time Systems Development*. Butterworth-Heinemann, Oxford.

[Wolf et al., 2008] Wolf, W., Jerraya, A. A., and Martin, G. (2008). Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713.

[Wolkotte et al., 2005] Wolkotte, P. T., Smit, G. J. M., Rauwerda, G. K., and Smit, L. T. (2005). An energy-efficient reconfigurable circuit-switched network-on-chip. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 8 pp.–.

[Xia et al., 2021] Xia, M., Liu, H., Li, J., and Li, M. (2021). Research on task scheduling algorithm based on multi-time period merging. In *4th World Conference on Mechanical Engineering and Intelligent Manufacturing (WCMEIM)*, pages 389–394.

[Xu and Parnas, 2000] Xu, J. and Parnas, D. L. (2000). Priority Scheduling Versus Pre-Run-Time Scheduling. *Real-Time Syst*, 18(1).

[Yaashuwanth and Ramesh, 2010] Yaashuwanth, C. and Ramesh, R. (2010). Web-enabled framework for real-time scheduler simulator (a teaching tool). In *International Conference on Computer Research and Development*, pages 826–830.

[Ye et al., 2005] Ye, Q., Liu, J., and Zheng, L. (2005). Switch Design and Implementation for Network-on-Chip. In *Conference on High Density Microsystem Design and Packaging and Component Failure Analysis (HDP)*, pages 1–7.

[Yeong et al., 2021] Yeong, D. J., Velasco-Hernandez, G., Barry, J., and Walsh, J. (2021). Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review. *Sensors*, 21(6).

[Yin et al., 2014] Yin, J., Zhou, P., Sapatnekar, S. S., and Zhai, A. (2014). Energy-Efficient Time-Division Multiplexed Hybrid-Switched NoC for Heterogeneous Multicore Systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 293–303.

[Yoon et al., 2010] Yoon, Y. J., Concer, N., Petracca, M., and Carloni, L. (2010). Virtual Channels vs. Multiple Physical Networks: A Comparative Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, pages 162–165.

[Zhan et al., 2014] Zhan, J., Stoimenov, N., Ouyang, J., Thiele, L., Narayanan, V., and Xie, Y. (2014). Optimizing the NoC Slack Through Voltage and Frequency Scaling in Hard Real-Time Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(11):1632–1643.

[Zhang, 1995] Zhang, H. (1995). Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1396.

[Zhang, 2014] Zhang, H. (2014). *Exploiting many-core processors to optimize the performance of simultaneous localization and mapping*. PhD thesis, University of Massachusetts Lowell.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br