# NoC-based Architectures for Real-Time Applications : Performance Analysis and Design Space Exploration

Frédéric Giroudot

# THÈSE

**En vue de l'obtention du**

# DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

**Délivré par :**

Institut National Polytechnique de Toulouse (Toulouse INP)

**Discipline ou spécialité :**

Informatique et Télécommunication

---

**Présentée et soutenue par :**

M. FREDERIC GIROUDOT

le vendredi 13 décembre 2019

**Titre :**

NoC-based Architectures for Real-Time Applications: Performance Analysis and Design Space Exploration

---

**Ecole doctorale :**

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

**Unité de recherche :**

Département Modèles pour l'Aérodynamique et l'Energétique (DMAE-ONERA)

**Directeur(s) de Thèse :**

M. AHLEM MIFDAOUI

M. EMMANUEL LOCHIN

**Rapporteurs :**

M. FRÉDÉRIC MALLET, INRIA SOPHIA ANTIPOLIS

M. LAURENT GEORGE, UNIVERSITE MARNE LA VALLEE

**Membre(s) du jury :**

M. LAURENT GEORGE, ESIEE NOISY LE GRAND, Président

M. AHLEM MIFDAOUI, ISAE-SUPAERO, Membre

M. CLAIRE PAGETTI, ONERA TOULOUSE, Membre

M. EMMANUEL LOCHIN, ISAE TOULOUSE, Membre

M. MARC GATTI, THALES AVIONICS, Membre

**Abstract**

Monoprocessor architectures have reached their limits in regard to the computing power they offer vs the needs of modern systems. Although multicore architectures partially mitigate this limitation and are commonly used nowadays, they usually rely on intrinsically non-scalable buses to interconnect the cores.
The manycore paradigm was introduced to tackle the scalability issue of bus-based architectures. It can scale up to hundreds of processing elements (PEs) on a single chip, by organizing them into computing tiles (holding one or several PEs). Intercore communication is usually done using a Network-on-Chip (NoC) that consists of interconnected on-chip routers on each tile.

However, manycore architectures raise numerous challenges, particularly for real-time applications. First, NoC-based communication tends to generate complex blocking patterns when congestion occurs, which complicates the analysis, since computing accurate worst-case delays becomes difficult. Second, running many applications on large Systems-on-Chip such as manycore architectures makes system design particularly crucial and complex. It complicates Design Space Exploration, as it multiplies the implementation alternatives that will guarantee the desired functionalities. Besides, given an architecture, mapping the tasks of all applications on the platform is a hard problem for which finding an optimal solution in a reasonable amount of time is not always possible.

Therefore, our first contributions address the need for computing tight worst-case delay bounds in wormhole NoCs. We first propose a buffer-aware worst-case timing analysis (BATA) to derive upper bounds on the worst-case end-to-end delays of constant-bit rate data flows transmitted over a NoC on a manycore architecture.
We then extend BATA to cover a wider range of traffic types, including bursty traffic flows, and heterogeneous architectures. The introduced method is called G-BATA for Graph-based BATA. In addition to covering a wider range of assumptions, G-BATA improves the computation time; thus increases the scalability of the method.
In a third part, we develop a method addressing design and mapping for applications with real-time constraints on manycore platforms. It combines model-based engineering tools (TTool) and simulation with our analytical verification technique (G-BATA) and tools (WoPANets) to provide an efficient design space exploration framework.
Finally, we validate our contributions on (a) a series of experiments on a physical platform and (b) a case study taken from the real world, the control application of an autonomous vehicle.

**Résumé**

Les architectures mono-processeur montrent leurs limites en termes de puissance de calcul face aux besoins des systèmes actuels. Bien que les architectures multi-cœurs résolvent partiellement ce problème, elles utilisent en général des bus pour interconnecter les cœurs, et cette solution ne passe pas à l'échelle.
Les architectures dites pluri-cœurs ont été proposées pour palier les limitations des processeurs multi-cœurs. Elles peuvent réunir jusqu'à des centaines de cœurs sur une seule puce, organisés en dalles contenant une ou plusieurs entités de calcul. Elles sont généralement munies d'un réseau sur puce permettant les échanges de données entre dalles.

Cependant, ces architectures posent de nombreux défis, en particulier pour les applications temps-réel. D'une part, la communication via un réseau sur puce provoque des scénarios de blocage entre flux, ce qui complique l'analyse puisqu'il devient difficile de déterminer le pire cas. D'autre part, exécuter de nombreuses applications sur des systèmes sur puce de grande taille comme des architectures pluri-cœurs rend la conception de tels systèmes particulièrement complexe. Premièrement, cela multiplie les possibilités d'implémentation qui respectent les contraintes fonctionnelles, et l'exploration d'architecture résultante est plus longue. Deuxièmement, déterminer de façon optimale la répartition des tâches à exécuter sur les entités de calcul n'est pas toujours possible en un temps raisonnable.

Ainsi, nos premières contributions s'intéressent à cette nécessité de pouvoir calculer des bornes fiables sur le pire cas des latence de transmission des flux de données empruntant des réseaux sur puce dits « wormhole ». Nous proposons un modèle analytique, BATA, prenant en compte la taille des mémoires tampon des routeurs et applicable à une configuration de flux de données périodiques générant un paquet à la fois.
Nous étendons ensuite le domaine d'applicabilité de BATA pour couvrir un modèle de traffic plus général ainsi que des architectures hétérogènes. Cette nouvelle méthode, appelée G-BATA, est basée sur une structure de graphe pour capturer les interférences possibles entre flux de données. Elle permet également de diminuer le temps de calcul de l'analyse, améliorant la scalabilité de l'approche.
Dans une troisième partie, nous proposons une méthode pour la conception de systèmes temps-réel basés sur des plateformes pluri-cœurs. Cette méthode intègre notre modèle d'analyse G-BATA dans un processus de conception systématique, faisant en outre intervenir un outil de modélisation et de simulation de systèmes reposant sur des concepts d'ingénierie dirigée par les modèles, TTool, et un logiciel pour l'analyse temps réel des réseaux, WoPANets.
Enfin, nous proposons une validation de nos contributions grâce à (a) une série d'expériences sur une plateforme physique et (b) une étude de cas d'application réelle, le système de contrôle d'un véhicule autonome.

# Acknowledgements

*A hero can be anyone.*

—Bruce Wayne, *The Dark Knight Rises*

To my supervisors, Ahlem and Emmanuel – I want to express my profound and genuine gratitude. I have never been a role model PhD student, so having role model supervisors was some kind of undeserved blessing. Many thanks for cheering me up in my worst days of discouragement, for believing in me and my work, for your immensely valuable input on papers and job applications, and most of all for letting me pursue my second passion.

To Guillaume, Stephen and Vincent, with whom I had the pleasure of working when I was in Sydney – thanks for making my experience at Data61 so amazing and valuable. I hope I see you again soon Down Under.

To Ludovic, who dragged me away from the vortex of my first contributions and opened to me the doors of TTool – I learnt so much working with you, and your efficiency at work was and continues to be an inspiration.

I would like to thank my thesis referees, Laurent George and Frédéric Mallet, for their feedback and the fresh look they provided on my work. Thanks, also, to my examinators, Claire Pagetti and Marc Gatti, for your input on my presentation and report.

Things would not have been the same without my work colleagues and friends, fellow PhD students and Doctors – I particularly want to mention:

Antoine, my equally-long-legged-sitting-in-front-of-me mate, mentor and famous WiFi expert, for joining me in the constant claim that 31000 is better than 31400; Eyal, fellow psytrance-listener, who left to another office too soon; Franco, the other bicycle-to-work-addict of my office; Ilia, for reminding me that no matter how much I struggle with red tape, there is always someone having it worse; Narjes, for her concerns about my (in)sanity and her inversely-proportional-to-her-size smile; Zoe, fellow adopted Aussie, for her knowledge about vermicomposting; Bastien, pilot, astrophysicist, TCP-specialist, seasoned artist in obnoxious puntastic jokes, for the always-appreciated observation nights that unveiled Saturn's rings, Jupiter's satellites, Mars and blinding views of the Moon; Doriane, Earth-explorer and sneaky raccoon who stole my NFC tag reader, for her shared love of electronic music and

for reminding me that there are no dumb questions; Henrick, inspirational alpha-vegetarian with a dash of perfectionism I sometimes wish I had, for what will go down in history as the "prank-that-went-too-far"; Lucas, for preventing my body from melting during the hottest days of summer, and whose will to experiment with ~~blockchain~~ rum was always appreciated; Clément, watercooling/lockpicking referent; John, whose PhD defence was even more impressive than his ability to ingest large amounts of ~~junk~~ food. Thanks for the shared laughs, meals, coffees, drinks, concerts, home parties, work hours; and for the many more things that helped me get through each day.
I also want to thank Christophe and Tanguy for trusting me with teaching part of their respective course in C and Java – it has been incredibly rewarding and I learnt a lot – as well as Odile, Alain, Rob and everyone at the DISC.

To Edo, Pauline, Eunbee and Lana – thanks for bearing your fair share of the living-under-the-same-roof-as-Fred pain. Your undiscontinued support makes you family to me now.
To my friends from prep class and engineering school – thanks for being part of the journey that brought me here. May you remain as brilliant as you are humble.
To my friends in Toulouse, in Sydney and elsewhere, and to my family – a thousand of thank yous for your comforting presence and for not asking too much about how my PhD was going. I can't name y'all but you know who you are.

To my aerial friends and acquaintances – I am so grateful for your support and for the breath of fresh air you have been when I was drowning under work. I still can't believe I managed to get through this PhD while being able to train and compete. It was worth every struggle, injury, breakdown and freakout.
Thanks Maureen, Popo, Anne-Claire, Chacha, Olivia, Marion, Max, Cami, Dennis, Matt, Adam; and all the boys and girls I met around the world.

Special thanks to those who helped with the corrections of this manuscript: Alexandra, Ants, Béné, Charles, Lana, Taylor – you are the best.

I dedicate this thesis to my best friend Rhita, who, hopefully, will never have to go through the agony of reading it.

Fred

# Contents


**I Problem Statement and State of the Art** **1**

**1 Introduction** **3**

**2 Context and Problem Statement** **7**
2.1 Real-Time Systems Context . . . . . 8
2.1.1 Characteristics . . . . . 8
2.1.2 Examples . . . . . 8
2.1.3 Requirements and Challenges . . . . . 9
2.2 Manycore Platforms . . . . . 10
2.2.1 Topologies . . . . . 11
2.2.2 Forwarding Techniques and Flow Control . . . . . 15
2.2.3 Arbitration and Virtual Channels . . . . . 18
2.2.4 Routing Algorithms . . . . . 20
2.2.5 NoC Examples . . . . . 23
2.3 Discussion: NoCs and Real-Time Systems . . . . . 26
2.4 Conclusion . . . . . 29

**3 State of the Art** **31**
3.1 Timing Analysis of NoCs . . . . . 32
3.1.1 Overview . . . . . 32
3.1.2 Scheduling Theory . . . . . 33
3.1.3 Compositional Performance Analysis . . . . . 36
3.1.4 Recursive Calculus . . . . . 37
3.1.5 Network Calculus . . . . . 38
3.1.6 Discussion . . . . . 40
3.2 System Design and Software/Hardware Mapping . . . . . 41
3.2.1 Design Space Exploration . . . . . 42
3.2.2 Task and Application Mapping on Manycore Architectures . 44
3.2.3 Discussion . . . . . 48
3.3 Conclusion . . . . . 48

**II Contributions** **51**

**4 BATA: Buffer-Aware Worst-Case Timing Analysis** **53**
4.1 Introduction . . . . . 54
4.2 Assumptions and System Model . . . . . 54
4.2.1 Network Model . . . . . 54
4.2.2 Flow Model . . . . . 57
4.3 Approach Overview . . . . . 57
4.3.1 Buffer-Awareness: An Example . . . . . 57

4.3.2 Main Steps of BATA . . . 59
4.4 Indirect Blocking Analysis . . . 61
4.5 End-to-End Service Curve Computation . . . 65
4.5.1 Direct Blocking Latency . . . 65
4.5.2 Indirect Blocking Latency . . . 67
4.5.3 Computation Algorithm . . . 69
4.6 Illustrative Example . . . 70
4.7 Performance Evaluation . . . 73
4.7.1 Sensitivity Analysis . . . 74
4.7.2 Tightness Analysis . . . 77
4.7.3 Computational Analysis . . . 79
4.7.4 Comparative Study . . . 82
4.8 Conclusions . . . 84

**5 G-BATA: Extending Buffer-Aware Timing Analysis 87**
5.1 Problem Statement . . . 88
5.1.1 Illustrative Example . . . 88
5.1.2 Main Extensions . . . 90
5.2 Extended System Model . . . 91
5.2.1 Traffic Model . . . 91
5.2.2 Network Model . . . 91
5.3 Interference Graph Approach for Indirect Blocking Set . . . 93
5.4 Refining Indirect Blocking Latency . . . 96
5.5 G-BATA: Illustrative Example . . . 98
5.6 Performance Evaluation . . . 100
5.6.1 Computational Analysis . . . 101
5.6.2 Sensitivity Analysis . . . 107
5.6.3 Tightness Analysis . . . 112
5.7 Conclusions . . . 115

**6 Hybrid Methodology for Design Space Exploration 117**
6.1 Introduction . . . 117
6.2 Overview and Extended Workflow . . . 118
6.3 System Modeling: Adding a NoC Component in TTool . . . 119
6.3.1 Implementation . . . 119
6.3.2 Functional View . . . 120
6.3.3 Architecture . . . 124
6.4 Verification, NoC Generation and Simulation . . . 124
6.5 Performance Evaluation . . . 126
6.5.1 Example Modeling . . . 126
6.5.2 Analysis and Results . . . 128
6.6 Conclusion . . . 131

**7 Practical Applications 133**
7.1 Experiments on TILE-Gx8036 . . . 134
7.1.1 Platform Characteristics . . . 134
7.1.2 Traffic Generation . . . 134
7.1.3 Latency Measurements . . . 136
7.1.4 Results and Discussion . . . 137
7.2 Control of an Autonomous Vehicle: Timing Analysis and Comparative Study . . . 138
7.3 Control of an Autonomous Vehicle: Modeling and DSE . . . 143
7.3.1 Functional Description . . . 144
7.3.2 Architecture Modeling . . . 145
7.3.3 Simulation . . . 145
7.4 Results and Conclusion . . . 146

**III Conclusion 149**

**8 Conclusion 151**
8.1 Summary of Contributions . . . 151
8.1.1 BATA . . . 151
8.1.2 G-BATA . . . 152
8.1.3 Hybrid Design Space Exploration . . . 152
8.1.4 Validation . . . 153
8.2 Perspectives . . . 153
8.2.1 Models and Approaches . . . 153
8.2.2 Tools . . . 155

**IV Appendix 157**

**A Résumé en français 159**
A.1 Introduction . . . 160
A.2 Contexte de la thèse . . . 162
A.2.1 Systèmes temps-réel . . . 162
A.2.2 Architectures pluri-cœurs . . . 163
A.3 État de l'art . . . 166
A.3.1 Analyse temps réel des réseaux sur puce . . . 166
A.3.2 Exploration d'architectures et mapping logiciel/matériel sur architectures pluri-cœurs . . . 170
A.4 Analyse temporelle pire cas des réseaux sur puce wormhole intégrant l'impact des mémoires tampon . . . 173
A.4.1 Modélisation du réseau et des flux . . . 173
A.4.2 Illustration du problème . . . 175
A.4.3 Formalisme et calculs . . . 176
A.4.4 Résumé de l'analyse de performance . . . 178
A.4.5 Conclusion . . . 182

A.5 Analyse temps réel des NoCs wormhole hétérogènes par graphe d'interférences . . . 182
A.5.1 Formalisme étendu . . . 182
A.5.2 Définition et construction du graphe d'interférence . . . 183
A.5.3 Analyse de performance . . . 185
A.5.4 Conclusion . . . 186
A.6 Approche hybride pour l'exploration d'architectures . . . 186
A.6.1 Workflow étendu . . . 187
A.6.2 Modélisation système du NoC . . . 187
A.6.3 Performance . . . 189
A.6.4 Conclusion . . . 191
A.7 Validation des contributions . . . 193
A.7.1 Analyse pire cas . . . 193
A.7.2 Modélisation sous TTool . . . 195
A.8 Conclusion . . . 196
A.8.1 Résumé des contributions . . . 196
A.8.2 Perspectives . . . 198

**B Network Calculus Memo** **201**
B.1 Basics . . . 201
B.2 Notations . . . 203

**C Case Study Data** **207**

**D List of Publications** **211**

**E List of Abbreviations** **213**

**Bibliography** **215**

# List of Figures


2.1 Ring network topology . . . . . 12
2.2 Mesh and Torus topologies . . . . . 13
2.3 Flattened Butterfly topology . . . . . 14
2.4 Bypass mechanism . . . . . 19
2.5 A typical deadlock scenario . . . . . 21
2.6 Intel SCC overview: grid and tile . . . . . 23
2.7 Tilera TILE-Gx8036 overview . . . . . 24
2.8 Kalray MPPA overview . . . . . 25

4.1 Typical 2D-mesh router . . . . . 55
4.2 Architecture of an input-buffered router and output multiplexing with arbitration modeling choices . . . . . 56
4.3 Example configuration and packet stalling . . . . . 58
4.4 Another configuration where flow 1 cannot be blocked by flow 3 . . . 59
4.5 Subpath illustration for the *foi k* . . . . . 62
4.6 Flow configuration on a 6×6 mesh NoC . . . . . 74
4.7 Buffer size impact on BATA end-to-end delay bounds . . . . . 75
4.8 Packet length impact on BATA end-to-end delay bounds . . . . . 75
4.9 Rate impact on BATA end-to-end delay bounds . . . . . 76
4.10 Results of the BATA computational analysis . . . . . 80
4.11 Number of calls to the function endToEndServiceCurve() . . . . . 81
4.12 A simple configuration from [1]. . . . . 82
4.13 Predicted bounds for different values of bandwidth . . . . . 83
4.14 Delay bounds of flow 1 vs buffer size under CPA and NC approaches 83

5.1 Example configuration and subpaths computation with BATA . . . . 89
5.2 Packet configuration with two instances of flow 2 . . . . . 89
5.3 Main steps of G-BATA . . . . . 90
5.4 Subpaths computation with G-BATA approach . . . . . 100
5.5 Compared runtimes of BATA and G-BATA approaches . . . . . 101
5.6 Comparative study of the algorithmic complexity . . . . . 103
5.7 Scalability of G-BATA on large flow sets . . . . . 103
5.8 Quadrants of the NoC and illustration of flows from families A, B and C . . . . . 104
5.9 Runtimes *vs* flow number . . . . . 106
5.10 Studying the correlation between average DB index (resp. average IB index) and total runtime for 32-flow configurations . . . . . 107
5.11 Compared buffer size impact on end-to-end delay bounds . . . . . 108
5.12 Packet length impact on G-BATA end-to-end delay bounds . . . . . 110
5.13 Packet length impact on BATA end-to-end delay bounds . . . . . 110
5.14 Compared flow rate impact on end-to-end delay bounds . . . . . 111

5.15 Determining which approach should be used . . . . . 114

6.1 Workflow of the hybrid approach . . . . . 118
6.2 Extended workflow with implementation details . . . . . 120
6.3 NoC functional view . . . . . 121
6.4 Functional structure of a router . . . . . 121
6.5 Read and Write operations in the router model . . . . . 123
6.6 Control events in the router model . . . . . 123
6.7 Architectural view of the presented router (with task mapping) . . . 124
6.8 Example configuration . . . . . 126
6.9 Functional view of the example . . . . . 127
6.10 Activity diagrams . . . . . 127
6.11 Mapping example . . . . . 128
6.12 Distribution of end-to-end delays . . . . . 130

7.1 Main application algorithmic view . . . . . 135
7.2 TX and RX processes . . . . . 136
7.3 Experiment configurations 1 and 2 . . . . . 137
7.4 Experiment results for configuration 1 and 2 . . . . . 138
7.5 Worst-case end-to-end delay bounds comparative . . . . . 140
7.6 Delay bounds with 4, 2 and 1 VC with buffer size = 2 flits . . . . . 141
7.7 Task graph of the case study . . . . . 144
7.8 Activity diagram of control task 12 . . . . . 145
7.9 Architecture of the platform . . . . . 146
7.10 End-to-end delay measurements on the case study . . . . . 147

A.1 Topologies en grille et tore 2D . . . . . 164
A.2 Architecture d'un router de mesh 2D et niveaux d'arbitrage . . . . . 174
A.3 Exemple de blocage avec étalement des paquets . . . . . 175
A.4 Calcul d'un sous-chemin relativement au flux $k$ . . . . . 177
A.5 Configuration test sur un réseau sur puce en grille 6×6 . . . . . 179
A.6 Résultats de l'analyse de scalabilité . . . . . 181
A.7 Processus de l'approche hybride . . . . . 187
A.8 Description fonctionnelle du réseau sur puce . . . . . 188
A.9 Description fonctionnelle d'un routeur 2 entrées, 2 sorties, 2 canaux virtuels . . . . . 189
A.10 Configuration considérée . . . . . 189
A.11 Vue fonctionnelle . . . . . 190
A.12 Diagrammes d'activité . . . . . 190
A.13 Architecture et mapping . . . . . 191
A.14 Distribution des délais mesurés . . . . . 192
A.15 Comparatif des bornes sur le délai de bout en bout . . . . . 194
A.16 Architecture de la plateforme . . . . . 196
A.17 Distribution des délais de bout en bout normalisés . . . . . 197

# List of Tables


2.1 Summary of main NoC topologies . . . . . 14
2.2 Forwarding techniques . . . . . 17
2.3 Flow control techniques . . . . . 18
2.4 Summary of design choices and their impact . . . . . 26

3.1 Summary of approaches for timing analysis of wormhole NoCs . . . 41

4.1 Tightness ratio results for the tested configuration . . . . . 78

5.1 Tightness Summary for both approaches, buffer size 4 flits and 16 flits 113

6.1 Request times, injection times, ejection times and delays for all flows 129

7.1 Average tightness and tightness differences for various buffer sizes . . 139
7.2 Relative increase of the worst-case end-to-end delay bounds for $B = 2$ 142
7.3 Runtimes of BATA and G-BATA for different NoC configurations . . 143

A.1 Résumé des choix de conception et leur impact . . . . . 166
A.2 Indices de finesse pour les configurations testées . . . . . 180
A.3 Résultats de simulation . . . . . 190

B.1 Summary of notations . . . . . 205

C.1 Original task-core mapping . . . . . 208
C.2 Flow set characteristics . . . . . 209
C.3 Computed tightness ratios for buffer size values 2, 100 and $\infty$ . . . . 210

# Part I

# Problem Statement and State of the Art

*Anything that can go wrong, will go wrong.*

—Murphy's Law

*Do you dance, Minnaloushe, do you dance?*

—W.B. Yeats, *The Cat and the Moon*

Chapter 1

# Introduction

For a long time, processor architectures have been organized around a single processing element (PE). As performance requirements consistently increase, various optimizations and improvements have been made on such architectures. Adding specialized components and various controllers or coprocessors to handle repetitive tasks (such as network interfacing) or computationally expensive tasks (such as cryptographic functions) can unload the processing element, without fundamentally improving its performance.
Increasing the clock frequency is an option as well, but it comes at the price of a higher power consumption that causes more thermal dissipation. As such, it may not be appropriate for systems where only a passive cooling system is available, and is inherently limited by the physical resiliency of the chip.
Optimizing the processor pipeline, or using out-of-order execution and branch prediction techniques has also led to significant performance increase. However, it introduces undeterminism and additional design complexity. Moreover, attacks exploiting out-of-order execution and/or speculative execution (Meltdown [2], Spectre [3]) have recently been implemented, which compromise memory isolation on many widely used processors. The immediate solutions to mitigate those vulnerabilities have a significant negative impact on performance [3].
Besides, none of the aforementioned techniques implements actual parallelism. Multiprocessor architectures have addressed this aspect. In addition to the various external devices or components, they classically feature several CPUs interconnected by a bus, but they are inherently limited in terms of scalability.

To cope with these limitations, the manycore paradigm was proposed [4]. It was made possible because chip technology has evolved to allow the integration of more and more transistors on the same silicon die. A manycore architecture is a set of "many"[1] simple processors on a single chip, usually organized as an array of

[1] Usually, "many" is understood as "in the order of magnitude of $10^2$ and above"

*tiles*. To avoid bottleneck issues, each tile holds its local memory and cache, hence cache coherency mechanisms and more generally memory requests must rely on an interconnect allowing inter-tile communication. Core-to-core message passing and access to external devices must be handled in a scalable way as well.
Several of such architectures are commercially available [5, 6, 7, 8].

In critical systems (*e.g.* avionics, aerospace and automotive), operating reliability is ensured through various costly certification processes. Therefore, there is still a strong trend to reuse components and architectures that are already certified; thus little incentive to migrate to a different architecture paradigm. Moreover, the multiplicity of processing elements and the concurrent execution of tasks make predictability of the system behavior harder to guarantee.
Nonetheless, the transition to manycore architecture seems unavoidable. Monoprocessor architectures are likely to become scarcely available as their performance will be outranked by more recent manycore chips. Besides, monoprocessor architectures also struggle to meet the increasing computing power requirements in critical and mixed-criticality systems. The current paradigm in avionics is to multiply the number of monoprocessor computers, however such a solution decreases inherently the system scalability.

Therefore, ensuring predictable execution of critical applications on manycore platforms appears as one of the main challenges of the transition from monoprocessor to manycore architectures. There are several ways to contribute to this challenge. Mostly, they regard aspects of executing an application on a processing resource, with a particular focus on the specificities of manycore platforms compared to monoprocessor architectures (shared memory, shared communication media, and concurrent execution). In particular, these aspects include:

- enforcing task and memory isolation;
- mapping applications and tasks onto cores in a way that ensures tasks execute within their deadlines, while taking into account the dependencies between tasks;
- getting guarantees on communication delays between tasks;
- handling various levels of criticality among the applications running on the platform, and ensure least critical tasks do not impact the safe execution of most critical tasks;

This thesis will present our work on several specific aspects of real-time applications execution on manycore chips, but try as much as possible to broaden the

applicability of our models to various platforms.

We start by giving the background we need for our work, and stating the problem we will be working on (Chapter 2). Then, we discuss the state-of-the art and the opportunities for possible contributions in Chapter 3. We then develop our contributions as follows:

- First, we address the intercore communication issue by introducing a timing analysis method applicable to a wide range of manycore architectures. Knowing an upper bound on the traffic generated and/or consumed by each task running on a manycore platform, such an approach allows to derive worst-case performance bounds relative to the data flows.
- The result can be exploited to prove whether the communication requirements are feasible with the given configuration and help orienting necessary alterations of the chosen architecture. In that respect, Chapter 4 presents a first approach, called BATA, for worst-case timing analysis of data flows on wormhole Networks-on-Chip, taking buffer size into account. We thoroughly evaluate BATA performances and find out that while it yields safe bounds with a tightness up to 80% on average for the tested configurations, the computation time needed grows rapidly with the number of flows, and it can only model constant-bit rate traffic.
- Second, in Chapter 5, we extend the first approach by introducing G-BATA, for "Graph-based BATA". It simultaneously allows to model bursty traffic and heterogeneous architectures, and improves the computation time of the delay bounds. We evaluate its performances as well and compare them to the first model. We find out computations are up to 100 times faster than BATA for relatively small configurations, and the method is able to analyze large configurations (800 flows) with a reasonable analysis time (9 seconds per flow). G-BATA approach also provides bounds with a similar tightness compared to BATA.
- Third, we tackle task mapping and Design Space Exploration in Chapter 6. We integrate our intercore communication analysis approach into a system design methodology. Our proposal allows to model complex manycore-based systems and determine early in the design steps whether the real-time constraints are met. To that end, we introduce additional models in the toolkit TTool [9] for design space exploration and implement our methodology using WoPANets [10, 11], a software for worst-case timing analysis of networks, to which we add a plugin supporting our timing analysis model.

- Fourth, in Chapter 7, we validate our contributions by confronting them to real-world applications. We perform experiments on a Tilera TILE-Gx8036 manycore chip to measure data flow end-to-end latencies and compare them to the bounds yielded by our model. We are able to prove that our approach is practically applicable to physical configurations and provides safe delay bounds on the configuration. We then apply our timing analysis and design space exploration methodology to a case study of an autonomous vehicle control application.
- Finally, Chapter 8 concludes the thesis and unveils the future developments and perspectives of our work.

*Pour cet écrit, j'ai dû aux sciences*
*Offrir mon âme et trois années.*
*Permets qu'il t'emmène, en apnée,*
*Où les fils de ma pensée dansent.*

*
* *

Chapter 2

# Context and Problem Statement

## Contents


**2.1 Real-Time Systems Context . . . . . 8**
2.1.1 Characteristics . . . . . 8
2.1.2 Examples . . . . . 8
2.1.3 Requirements and Challenges . . . . . 9
**2.2 Manycore Platforms . . . . . 10**
2.2.1 Topologies . . . . . 11
2.2.2 Forwarding Techniques and Flow Control . . . . . 15
2.2.3 Arbitration and Virtual Channels . . . . . 18
2.2.4 Routing Algorithms . . . . . 20
2.2.5 NoC Examples . . . . . 23
**2.3 Discussion: NoCs and Real-Time Systems . . . . . 26**
**2.4 Conclusion . . . . . 29**


In this chapter, we narrow down the context of our work and we specify the problem statement. We start by discussing the specificities of real-time systems in Section 2.1, and provide a few examples. We underline the main requirements and challenges raised by such systems. Then, we focus on manycore platforms and Networks-on-Chip, from an architectural and functional perspective, in Section 2.2. Afterwards, we discuss the relevance of the different paradigms in regard to real-time systems requirements and challenges in Section 2.3. Finally, based on the insight from the first sections, we conclude this chapter by a summary of the area we will explore in our research.

## 2.1 Real-Time Systems Context

### 2.1.1 Characteristics

A system is a *real-time system* when "the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" [12]. In other words, the result must be within a timing constraint (also called *deadline*) to be relevant. A computationally correct result that does not comply with the timing constraint may be useless, or dangerous.

Contrarily to non real-time systems (*e.g.* best effort systems), that seek to optimize average performance, real-time systems require first of all that all timing constraints are met, even in the worst-case execution scenario.

The impact of a deadline miss determines the criticality of the system or application, *i.e.* it quantifies how important it is that the system shall comply with its real-time constraints. If failure to meet the deadline leads to catastrophic consequences – death or huge material loss – the system is said to be *hard real-time*. If the deadline miss is of lesser importance and does not impact significantly the functionalities of the system, the system is said to be *soft real-time*. Finally, a system or part of a system that will be severely impacted by a deadline miss without catastrophic consequences is sometimes referred to as *firm real-time*.

Sometimes, complex systems are *mixed-criticality* systems. This means that they comprise several subsystems, applications or tasks that have different criticality levels. Analysis of such systems raises additional challenges, as the execution of a non-critical task should not impact the execution of a critical one.

### 2.1.2 Examples

In Aeronautics, Full Authority Digital Engine Controller (FADEC) [13] is an application in charge of controlling an aircraft jet engine. It receives information from sensors located in the engine, processes them, and, if need be, performs the appropriate action. Failure of this application to work properly may prevent an engine malfunction from being detected and mitigated. The possible consequences include the loss of an engine, damage to the aircraft and/or to the people on board. Therefore, FADEC is a hard real-time application.
The control application of an autonomous vehicle detailed in [14] is in charge of

(i) processing information from a stereo photogrammetric set of sensors; (ii) deriving the absolute position of detected obstacles and adding them to its database; (iii) adapting trajectory and monitoring vehicle stability. As these functions are critical to guarantee the integrity of the vehicle (and, whenever it is relevant, of the obstacles that may be humans), the system is hard real-time.

The video streaming service of Netflix is a real-time system, as failure to deliver the video frames in a timely manner could cause the movie being watched to freeze. However, a non-optimal streaming experience at home will not cause anything more serious than a few curse words being pronounced. It is certainly not critical. Therefore, such a system is soft real-time.

Australian Government's Department of Home Affairs is a best-effort system for visa applications. For instance, 75% of applications for a subclass 407 training visa will be processed in 84 days or less, and 90% of applications will be processed in 4 months or less [15]. [1] There is no guarantee on the worst-case response time of the system.

### 2.1.3 Requirements and Challenges

Expected properties of real-time systems include in particular [16]:

1. **Timeliness** – results must not only be correct in terms of value, but also meet the associated deadline;
2. **Design for peak load** – the system must comply with its requirements even in the worst case scenario;
3. **Predictability** – to ensure that the performance requirements are met, the behavior of the system must be predictable;
4. **Safety** – we expect a valid behavior of the system in all circumstances. This includes fault tolerance and resilience to malicious attacks.

We consider safety issues as beyond the scope of this thesis, thus we choose to focus on the first three requirements.

Requirement 3 (predictability) is greatly impacted by design choices. For instance, common cache mechanisms improve the average execution time of a memory request, but introduce undeterminism. When a cache miss occurs, the latency to perform the memory request can increase by several orders of magnitude. As we have to account for the worst case (Req. 2), it may be relevant to deactivate cache mechanisms to improve worst-case performance.

[1] These informations are updated on a regular basis and may have evolved by the time this manuscript is read.

We will mostly rely on the predictability requirement to narrow down our study and leave out architecture paradigms and mechanisms that introduce undeterminism in the execution.
Additionally, real-time systems raise challenges that must be addressed, such as:

1. **Scalability** – with the increasing demands and foreseen size of real-time systems in the short term, both the architecture paradigm and the related analysis methods must scale when considering large systems;
2. **Complexity** – to facilitate reconfiguration and decrease development and maintenance costs, designers favor simple architectures, from both a hardware and software point of view. For instance, industrials in avionics and automotive tend to prefer using commercial off-the-shelf (COTS) technologies to reduce costs.

Fulfilling real-time requirements should therefore not prevent these challenges to be taken into account.
Regardless, design choices will not be enough to ensure all real-time requirements are satisfied. Moreover, a trade-off must be found between addressing the aforementioned challenges while complying with the requirements of real-time systems. That is why there is a strong need for analytical models that are able to prove that the system complies with all requirements, particularly 1 and 2. In the following sections, we will review design choices of manycore platforms and determine which of them are most suitable to help with real-time systems requirements and challenges.

## 2.2 Manycore Platforms

Manycore architectures need efficient intercore communication to make the most out of the additional computing power provided by the multiplicity of their processing elements. To avoid creating bottlenecks, they usually follow a NUMA (Non-Uniform Memory Access) paradigm, where each computing tile (holding one or several processing elements) has its local L1 cache and memory. Memory requests are thus addressed either locally or to a distant tile, and tiles are interconnected to allow not only distant memory requests, but also cache coherency mechanisms, core-to-core message passing and I/O and external devices access. Therefore, the choice of an interconnect is crucial from a performance point of view.
A simple interconnect such as a bus may be sufficient when there are only a few cores, but such a paradigm does not scale well over a few to a dozen of cores:

- the per-user available bandwidth is inversely proportional to the number of

users competing for the use of the bus;
- the bus clocking frequency and synchronization are constrained by electrical properties of the chip technology.

Point-to-point wired communication, although solving the bandwidth issue, has a quadratic wiring complexity, thus does not scale well either. Essentially, an efficient interconnect will be a trade-off between hardware complexity and paradigm scalability, while guaranteeing communication predictability and timeliness.
In that respect, Networks-on-Chip, proposed in [17, 18, 19], appear as a promising solution for distributed, scalable interconnects. Networks-on-Chip, abbreviated NoC(s), follow a paradigm similar to classic switched networks – routers interconnected by links receive and forward data packets – but the on-chip nature of these networks carries specific constraints and characteristics, including mainly:

- the topology of the interconnection;
- the protocol(s) used to forward data from node to node and to handle congestion;
- the arbitration policy;
- the algorithm(s) used to compute packet routes;
- the limited on-chip area;
- the place and route complexity and power consumption.
- the hardware complexity;
- the amount of memory needed at each node;
- the wiring complexity.

In the next sections, we will review the existing NoC architectures from these various points of view.

### 2.2.1 Topologies

It is generally not easy, or even impossible, to modify the hardware of an on-chip component. In that respect, and unless they are designed to be reconfigurable, NoCs topologies are static. To the best of our knowledge, there are no COTS architectures offering reconfigurable topologies, but it is something that could be imagined. Hereafter, we will only consider static topologies. The choice of the NoC topology impacts the scalability of the architecture (Challenge 1). To exhibit this impact, we consider a NoC $\mathcal{N}$ that we can assimilate to a directed, finite graph. We assume routers are vertices of the graph, denoted $nodes(\mathcal{N})$, while links are edges of the graph, denoted $edges(\mathcal{N})$. We present two metrics to characterize a NoC : network diameter and router radix. We will first need the following definition.

**Definition 1.** Minimal Path Length

*Given any two routers $R_1, R_2$ in the network, the minimal path length from $R_1$ to $R_2$ is the minimal number of "hops" or inter-router links that a packet must use to go from $R_1$ to $R_2$. We denote it $\mathcal{L}(R_1, R_2)$. In graph terms, for any $R_1, R_2 \in nodes(\mathcal{N})$, $\mathcal{L}(R_1, R_2)$ is the shortest path from vertex $R_1$ to vertex $R_2$.*

**Definition 2.** Network Diameter

*The network diameter $D_{\mathcal{N}}$ is defined as:*

$$D_{\mathcal{N}} = \max_{R_1, R_2 \in nodes(\mathcal{N})} \mathcal{L}(R_1, R_2)$$

*In other words, network diameter is the maximum of all minimal path lengths over the network.*

**Definition 3.** Router Radix

*Considering only routers with the same number of inputs and outputs, the radix of router $R$ is the number of input/output pairs of $R$. In graph terms, the router radix of $R$ is the degree of vertex $R$.*

With these notions, we now review several topologies described in the literature. The ring consists of $N$ routers in a circular disposition. Each of them has 2 input/output pairs with its 2 closest neighbors (in a full duplex configuration) and an input/output pair with the local processing element (Figure 2.1). The radix of each router is constant and equals 3, but the diameter of this network topology is linear in the number of nodes ($\lfloor \frac{N}{2} \rfloor$ in a full duplex configuration, $N$ when the links to the neighbors are unidirectional). Moreover, the last point implies that the available bandwidth on a link can decrease rapidly with the number of flows.

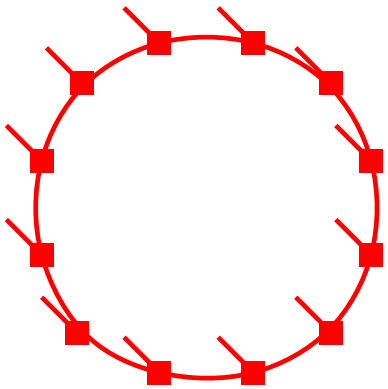


Figure 2.1 – Ring network topology

A mesh consists of routers disposed at the intersections of a grid, most commonly a 2D grid ([20, 21, 22, 23, 24, 25]), although 3D meshes also have been thoroughly studied [26, 27], and higher dimension meshes are possible as well [20]. Routers are connected to the local tile and to their $2n$ neighbors ($n$ being the dimension of the mesh), except for the border/corner routers (Figure 2.2). They have a constant radix, while the network diameter grows as $O(\sqrt{N})$ (provided the NoC is a square). A torus is similar to a mesh, but the border routers are connected to the routers of the opposite border (Figure 2.2). The radix remains the same, and the network diameter has the same asymptotic complexity (up to a constant multiplier).
Both these topologies exhibit a good scalability, mainly due to their acceptable router complexity. The main limiting factor is the diameter : for a $16 \times 16$ mesh NoC (256 nodes), the diameter is 31.

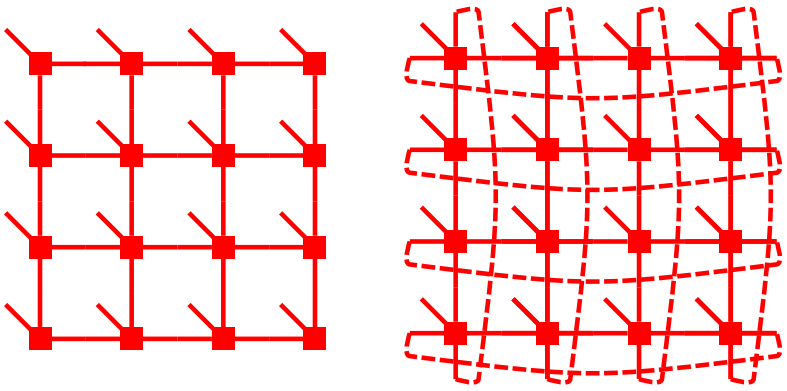


Figure 2.2 – Mesh and Torus topologies

Ring and $n$-dimensional torus are part of a larger family of topologies called the $k$-ary $n$-cubes [20, 28]. A $k$-ary $n$-cube contains $k^n$ nodes. For $k > 2$, each node has $2n$ neighbors (one in each dimension). For instance, a $k$-ary 1-cube is a ring with $k$ nodes, a $k$-ary 2-cube is a 2D torus with $k^2$ nodes, a 3-ary 3-cube is a 3D torus with 27 nodes.
In [29], the authors propose to adapt flattened butterfly topology to NoCs to prevent network diameter from increasing that fast with the number of nodes, and compare it to mesh topologies. The idea is that each router is connected to all routers on the same line and on the same column (Fig. 2.3, represented without the links to the tiles) and to one or more cores. If we connect each router to 4 cores, we are able to interconnect an $8 \times 8$-core chip using only 16 routers of maximum radix 10. The network diameter falls down to 2. Although this solution is not highly scalable, since following the same paradigm causes routers radix to explode rapidly

| Topology | Router radix | Diameter | Scalability |
|---|---|---|---|
| Ring | constant $= 3$ | $O(N)$ | Limited |
| $n$-dimensional mesh | constant $\leq 2n+1$ | $O(\sqrt[n]{N})$ | Good |
| $n$-dimensional torus | constant $= 2n+1$ | $O(\sqrt[n]{N})$ | Good |
| 2D Flattened Butterfly | $O(\sqrt{N})$ | constant, 2 for a 2D grid | Limited |

Table 2.1 – Summary of main NoC topologies *vs* scalability issue

$(O(\sqrt{(n)}))$, solutions for connecting several $8 \times 8$ flattened butterfly topologies together are presented. However, they are not evaluated in detail and may suffer from a non-optimized application mapping since they lead to non-uniform NoC topologies.

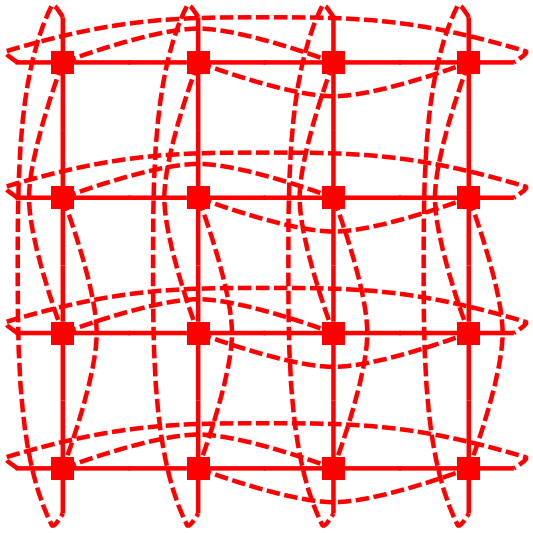


Figure 2.3 – Flattened Butterfly topology

Finally, let us mention that for large platforms, it is possible to design a hierarchical interconnect. For instance, the Kalray MPPA Bostan [7, 30] has 256 processing cores and uses a 2D mesh NoC. The NoC is a $4 \times 4$ grid, connecting 16 tiles. Each tile holds 16 cores that are interconnected and share the local network interface to access the NoC. Hybrid interconnects combining a mesh and buses have been mentioned in [31] to improve performance, but the scalability issue has not been addressed for larger interconnects.

We summarized our quantitative insights of NoC topologies in Table 2.1. Network diameter is the limiting factor for the ring topology as it increases linearly with the

number of routers.
Flattened butterfly fixes this issue by ensuring a constant diameter, at the expense of the router radix. Router radix depends on the size of the network, therefore flattened butterfly is not suitable for direct use in a NoC topology.
Mesh and torus appear to be a good compromise. They allow for simple routers with a constant radix (similar to what a ring topology offers) while keeping the diameter reasonable as the number of router increases.
For larger platforms, hierarchical topologies can be explored. The most common approach is to group several (2 to 16) processing elements on each tile [8, 32, 7]. This allows to increase the number of interconnected processing elements without needing more routers, which means the diameter does not increase. This implies an increased utilization rate of the routers at the network interface, as several processing elements will use the same router to access the NoC. More complex hybrid interconnects have not been thoroughly evaluated.

### 2.2.2 Forwarding Techniques and Flow Control

Forwarding packets and managing data flows on the NoC can be done in many ways. The characteristics of the chosen techniques impact:

- the timeliness (Req. 1) and particularly the traffic latency;
- the platform complexity (Challenge 2) and specifically the needed on-chip memory;
- the predictability (Req. 3), regarding lossless transmission.

There are several ways to transmit packets over a network. The classical way of doing so is Store-and-Forward (S&F): at every hop, each packet is forwarded to the next network node. Once it reaches the next node, the routing decision is made and the packet can be forwarded to the next node, and so on until it reaches its destination. With this technique, it is necessary to:

- have enough memory at each router to hold at least the largest packet;
- wait until each packet is completely stored to start forwarding it to the next node.

This last point causes an additional latency on the transmission of a packet, that depends on the link bandwidth, packet length, and number of nodes to cross. For instance, if a packet of length $L$ is transmitted over a path of $N$ nodes connected by links of bandwidth $C$, the latency to transmit the packet over the network without congestion is $\frac{L}{C}N$. It is proportional to the path length.

Circuit switching [20, 33] addresses this issue. The idea of Circuit switching is that the sender of a packet uses a (smaller) control packet to request the needed resources along the path and establish a circuit before transmitting the packet from source to destination. The packet will then travel without experiencing congestion, and thus will not need to be buffered in any intermediate node. If the size of the control packet is $L_c$, the network latency is $\frac{L_c}{C}N + \frac{L}{C}$. If $L_c$ is small compared to $L$, the length of the path has a minor impact on the network latency. However, this mechanism requires to reserve the whole path of a packet to proceed to the transmission. This fact may be problematic on heavy-utilized networks, because it blocks all other packets from using even one part of the reserved path during the whole packet transmission.

Virtual Cut-Through (VCT), presented in [34], also reserves links on the path using a header, but without requiring to wait until a complete circuit is established. The packet is sent right away and will be entirely buffered in an intermediate node if contention occurs ahead. As with circuit switching, there is no need to wait for the whole packet at each node. However, the packet has to be entirely removed from the network if it is blocked at some point. If the header length is $L_h$, the network latency without congestion is $\frac{L_h}{C}N + \frac{L}{C}$, with a negligible impact of the path length when $L_h$ is small compared to $L$. VCT has the same buffer requirements as S&F, since to cover worst-case congestion, it must be able to buffer an entire packet at each node.

Wormhole Routing proceeds with the same idea, by dividing a packet into flits of size $L_f$. The header progresses along the path, with the rest of the flits following in a pipelined way. The main difference with VCT is that when the header is blocked, the packet does not have to be entirely buffered in the corresponding node. Instead, a flow control mechanism blocks the remaining flits where they are, and the transmission resumes when the header flit can move again. This drastically reduces the amount of memory needed at each node. The network latency without congestion has the same form as VCT and circuit switching: $\frac{L_f}{C}N + \frac{L}{C}$.

As far as memory use is concerned, wormhole routing has an advantage over other techniques as it requires only enough memory to store one flit at each router. It exhibits a low network latency when no congestion occurs, and the packet buffering in case of congestion can be tweaked by varying the available buffer size at each router. For instance, increasing the buffer size will allow to store a packet in fewer

| Technique | Packet Latency | Per-node memory | Path reservation |
|---|---|---|---|
| S&F | $\frac{L}{C}N$ | $\geq L$ | no |
| circuit switching | $\frac{L}{C} + \frac{L_c}{C}N$ | none | yes |
| VCT | $\frac{L}{C} + \frac{L_h}{C}N$ | $L$ | no |
| wormhole | $\frac{L}{C} + \frac{L_f}{C}N$ | $L_f$ | no |
| bufferless | $\frac{L}{C} + \frac{L_f}{C}N$ | none | no |

Table 2.2 – Forwarding techniques and their requirements

nodes if it is blocked (or even more than one packet at each node), at the expense of memory requirements. For an on-chip system, carefully dimensioning memory will favorably impact power consumption and on-chip area.

Finally, we mention that bufferless techniques exist besides circuit switching, as described in [35, 36]. In [37], such a technique relies on a mechanism called *deflection* routing (also referred to as *hot-potato* routing). The idea is similar to wormhole routing, except that flits are never buffered. If the requested output is not available, flits are routed to a different one and never remain in a router. Such a technique is intrinsically limited to unicast transmission, because at each router, there has to be at most as many exiting flits than entering flits.

We synthesized the characteristics of forwarding techniques in Table 2.2, in terms of packet latency, buffer memory needs and path reservation.
To handle packets when congestion occurs, forwarding techniques rely on a mechanism called *flow control* (see [33]). We can mainly distinct 3 types of flow control mechanisms:
**Delay-based or Credit-based:** when a packet requests a resource that is unavailable, it is buffered and waits until it is granted the use of the resource. Delay-based flow control can be implemented using a system of credits issued from each input that grant the upstream output the ability to forward one flit. Such a mechanism enables lossless transmissions and can be used with most of the forwarding techniques mentioned earlier (S&F, VCT, wormhole). This mechanism induces a higher end-to-end delay when there is congestion to guarantee lossless transmission.
**Loss-based:** when a packet requests a resource that is unavailable, it is dropped after a certain time and has to be retransmitted. The retransmission is usually managed by a higher layer. Such a mechanism can be used with S&F, VCT and

| Flow control | Predictability |
|---|---|
| Delay-based | good |
| Loss-based | limited |
| Deflection-based | limited |

Table 2.3 – Flow control techniques and their predictability

wormhole routing as well. It may help reduce congestion in the network during periods of heavy utilization and improve the average latency, but introduces undeterminism due to the drop and retransmission issues. In particular, it is hard to bound the worst-case latency of a packet, considering it may be dropped an arbitrary number of times due to congestion.

**Deflection-based:** when a packet requests a resource that is unavailable, it is deflected from its original route. This mechanism also allows lossless transmission and improves load-balancing, because of its ability to reroute packets towards non-utilized links. Similarly to loss-based flow control, it comes at the price of an additional unpredictability regarding the delay experienced by a packet, because it is difficult to bound the number of times a packet may be deflected from its original path to destination.

We recap these characteristics in Table 2.3.

### 2.2.3 Arbitration and Virtual Channels

When several packets are competing for the use of one resource, the router has to decide which of them will be granted the use of the resource. The arbitration levels in the router may vary depending on the router architecture, but the underlying policies to favor one packet over another are generally among the following:

**First-Come First-Served (FCFS):** flits arriving at a router are served according to the time they arrived at the router, in a First-In First-Out way. This is the simplest policy.

**Round-Robin (RR):** at each router, all entities requesting the use of a resource are each allocated a certain amount of credit. Each of them is served until its credit runs out. Then, the arbiter serves the next entity, and so on. Each entity recovers its full credit amount when the arbiter switches to the next entity.

**Weighted Round-Robin (WRR):** WRR is the same as RR, except that the amount of credit assigned to the various entities may differ and favor some entities over others. The weight of an entity is the ratio of its amount of credit to the total amount of credit. This way, the sum of all weights equals 1. WRR and RR exhibit

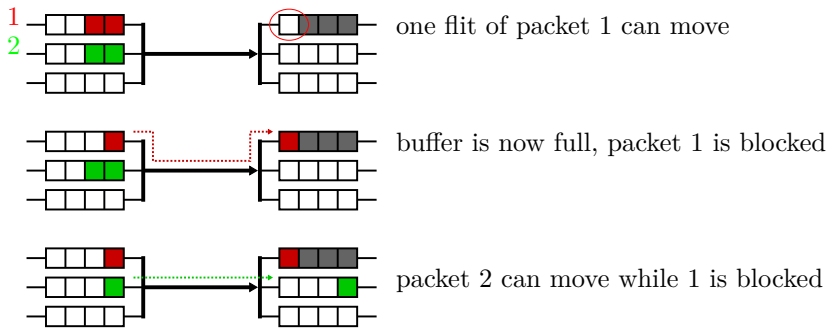



Figure 2.4 – Bypass mechanism

a good predictability (Req. 3) in terms of granted service.

**Fixed Priority (FP):** this policy requires packets to have a priority attribute. In case of concurrent request for the same resource, a packet with a higher priority will be served before a packet of a lower priority. This impacts the predictability. For higher priority packets, it increases predictability, as they are guaranteed a certain service. As such, it also improves timeliness (Req. 1) for higher priority levels. For lower priority packets, it decreases predictability, as their granted service depends a lot on the presence of higher priority packets competing for the same resources.

Virtual Channels (VCs), introduced in [38], are a way to share one physical link into separate logical channels, by implementing separate buffer queues at each router [39]. Packets in different VCs will use the same links from node to node but they will queue in different buffers. This especially allows one packet to bypass another that is blocked instead of having to queue behind it. On Figure 2.4, we present an example of a bypass scenario enabled by VCs. Initially, packet 1 is using the link and forwarding one flit to the next buffer, while packet 2 has to wait because the link is being used by packet 1. The next buffer on the path of packet 1 is then full, so packet 1 cannot move further. However, packet 2 is using another VC, and the next buffer of this VC is not full. Therefore, packet 2 can resume its transmission. Such a mechanism improves overall link utilization while reducing congestion. It may help with system scalability (Challenge 1), but at the expense of an increased complexity (Challenge 2).

Typically, VCs are used to provide different guarantees to different traffic classes, but they can also be used for preventing deadlocks (packets being forever blocked in the network, see Section 2.2.4), by breaking cyclic dependencies between the

resources requested by packets [40]. Packets may be statically assigned to one VC, or change VC during their transmission, depending on what VCs are used for.
A typical use of VCs is to implement a FP arbitration policy. To do that, one can map one priority level to a VC (this is done in [41]), or several priority levels to one VC. Another way, mentioned in [42], is to do a local mapping of priority levels to VCs at each node, depending on the flow communication pattern. Provided there are enough VCs, this can ensure only one flow is mapped to each VC at each router and/or minimize the number of needed VCs. This requires to know the path of the flows, and do an offline static mapping of priority levels on VCs.
Note that in routers, depending on the architecture, there may be several arbitration levels. For instance, one can arbitrate between packets depending on the input they come from or on the VC they use, with a different arbitration policy.

Arbitration policies that favor predictability are mostly (weighted) round-robin and fixed priority. FCFS, although simpler, provides a service that depends a lot on how many packets request the same resource, and at what time they do. RR is relatively simple to implement and provides the best fairness to all traffic classes. Fixed priority requires to handle priority attributes that can either be read from each packet, or determined using virtual channels, and in that way may increase hardware and/or software complexity. It is less fair than RR and degrades the predictability of the service granted to lower priority classes, but this may be a possibility worth exploring when dealing with mixed criticality traffic or flows with different timing requirements (real-time and best-effort).

### 2.2.4 Routing Algorithms

Knowing the flow control mechanism and forwarding technique used to transmit a packet from node to node is not enough to successfully transmit a packet over the NoC. In order for a packet to reach its destination, each node should know where a flit is supposed to be forwarded. This is ensured by choosing an appropriate way of determining the path of a packet from source to destination. Such a principle is called a *routing algorithm*. In this section, we will review different algorithms.

A routing algorithm, along with the chosen flow control mechanism, must ensure that all packets reach their destinations, and as such it should prevent two phenomena: deadlock and livelock. The former occurs when a packet or a flit is waiting to be granted the use of a resource used by another flit or packet, that is in turn

waiting for a resource to be freed, and so on so forth, with ultimately a packet or flit waiting for the resource used by the packet or flit of interest to become available. It occurs when resource requests dependencies form a cycle. Such a scenario is represented on Figure 2.5.

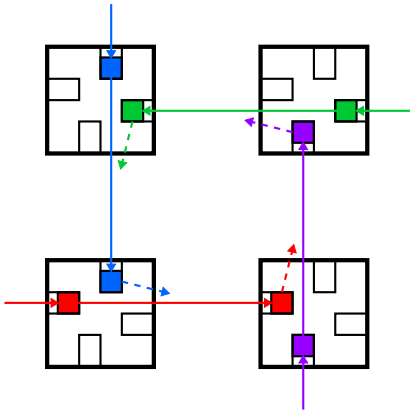


Figure 2.5 – A typical deadlock scenario

Here, each head flit of the purple, green, blue and red packet respectively is waiting for a resource used by a flit of green, blue, red and purple packet, respectively. The situation cannot evolve and the packets are blocked forever. A livelock occurs when a packet progresses in the network without ever reaching its destination. Typically, such a scenario is imaginable with bufferless techniques (a packet is never blocked) if the packet keeps being deflected from its destination due to an unfair arbitration policy with packets of contending flows.

Hence, the choice of a routing algorithm will impact the predictability requirement (Req. 3). We can distinct several characteristics of routing algorithms, as detailed in [33].

**Central *vs* distributed:** central routing relies on one common entity to compute the routes for all packets. Distributed routing has multiple entities capable of deciding all or part of the route a packet will use. Central routing can be done either offline for configurations where all data flows are known, or online when meaning the computing entity receives a request for each packet to be sent. The latter option does not scale well because it creates a bottleneck at the computing entity (Challenge 1). The former option allows to balance link utilization, but implies that all data flows must be known in advance. Distributed routing can also be done offline *e.g.* with the used of routing tables at each node, or dynamically.

It shows better scalability than centralized routing.

**Source routing *vs* hop routing:** both can be classified as subfamilies of distributed routing algorithms. Algorithms based on source routing compute the whole path of a packet before it is sent. Algorithms based on hop routing rely on each node to make the decision for the next hop. Choosing one or the other may slightly impact complexity, but the way it does depends on other factors.

**Deterministic *vs* adaptive:** Given a source SRC and a destination DST, a deterministic routing algorithm will always give the same route from SRC to DST. Running an adaptive algorithm twice with the same SRC and DST may, however, output different routes depending on the circumstances. It can adapt the path to live or unpredictable events, *e.g.* congestion or link failure. Adaptive routing can help mitigate congestion at the expense of predictability. In that respect, deterministic routing will fit better for real-time systems (Req. 3).

**Minimal *vs* nonminimal:** the path computed can be either (one of) the shortest paths available, or not. It is interesting to notice that in the general case of a $k$-ary $n$-cube, a deterministic, deadlock-free, minimal routing algorithm does not exist, but nonminimal routing algorithms have been introduced for such topologies [40, 20].

Routing algorithm possibilities are partly conditioned by the topology. For mesh and torus of an arbitrary dimension, the dimension-ordered routing (DOR) is a typical example of a deterministic, distributed and deadlock-free algorithm. We choose an order on the $n$ dimensions of the topology, from 1 to $n$. Knowing the position of the current node, $(x_1, \cdots, x_n)$, and the position of the destination node, $(x'_1, \cdots, x'_n)$, the packet is first routed along the dimension 1 until $x_1 = x'_1$. Then, it is routed along dimension 2, and so on, until finally $x_n = x'_n$. For a 2D mesh (or torus), the two possible DOR are X first and Y first, depending on which dimension is picked first.

Deflection routing, used especially in bufferless architectures, can be based on a preferred route, *e.g.* computed using a minimal routing algorithm. In the case two packets compete for the same output, the arbiter will deflect one of them from its preferred route. To avoid a packet being unfairly or endlessly deflected (which could cause a livelock), dedicated arbitration mechanisms have been developed when deflection is necessary [37].

Finally, the turn model, introduced in [43], allows to design partially adaptive or deterministic routing algorithms for $n$-dimensional meshes and $k$-ary $n$-cubes.

### 2.2.5 NoC Examples

#### 2.2.5.1 Intel SCC

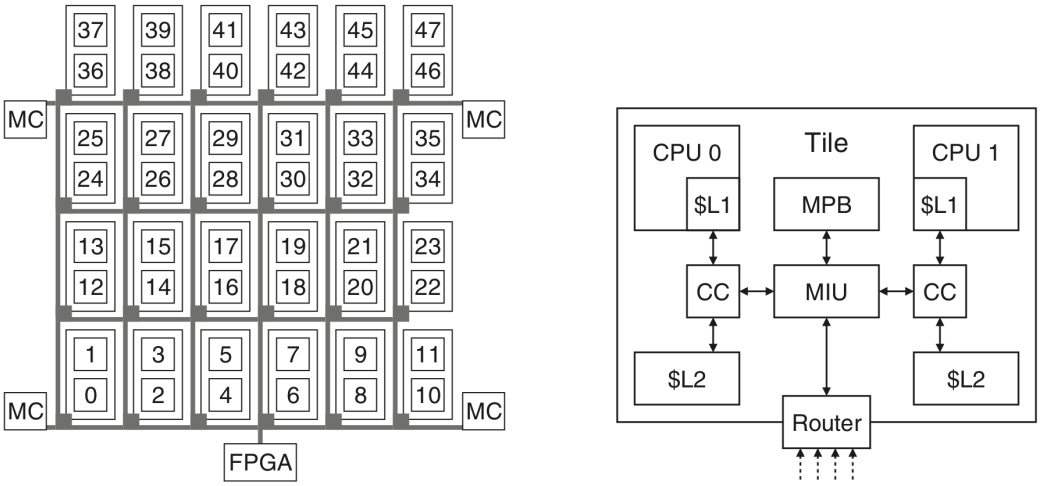



Figure 2.6 – Intel SCC overview: grid (left) and tile (right)

Intel's Single-chip Cloud Computer is a manycore chip developed for research purposes [44, 8]. It features 24 2-core tiles interconnected via a $4 \times 6$ mesh NoC. Routing policy is XY. Apart from the tiles, the SCC contains four memory controllers (MC) and an FPGA.
An overview of the SCC architecture is shown on Figure 2.6. On each tile, the two cores have their own L1 cache for data and instructions, and a L2 data cache with the associated cache controller (CC). A mesh interface unit (MIU) allows cores to access the NoC and handles the buffer that stores incoming packets, the message passing buffer (MPB).

Intel SCC has been widely used for research purposes, including in real-time-oriented papers [45, 46].

#### 2.2.5.2 Tilera TILE-Gx8036 and TILE-Gx Series

The TILE-Gx8036 [5] is a 36-core chip with a 2D-mesh NoC. The NoC is constituted of several independent networks for various types of traffic, but the user mostly has control over one of them, called UDN (User Dynamic Network). Among the other subnetworks, the I/O Dynamic Network (IDN) handles I/O devices access, while the memory system uses the QDN (reQuest Dynamic Network), RDN (Response

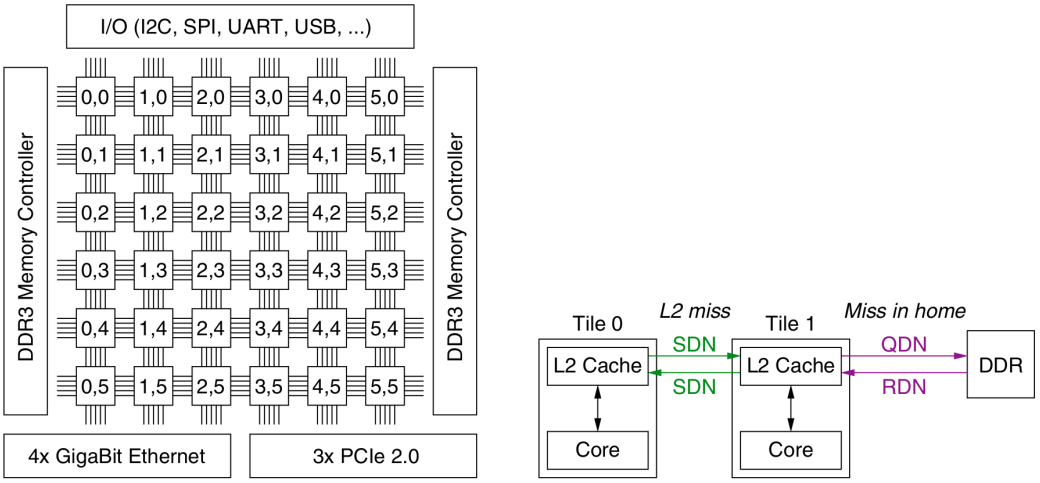



Figure 2.7 – Tilera TILE-Gx8036 overview: NoC and devices (left) and memory requests handling (right)

Dynamic Network) and the SDN (memory snoop network) to handle various operations. The UDN has no virtual channels and supports dimension-ordered routing (either X-first or Y-first). Buffers are located at the input of routers and can hold 3 flits. The technological latency of the routers is one cycle.
The arbitration mechanism of routers is central. Therefore, if two flits cross the router using different inputs and requesting different outputs, one of them will experience an additional delay of one cycle, as the arbitration mechanism will handle them one after the other. A far as arbitration mechanisms between input ports are concerned, the chip offers a classic Round-Robin policy and a Network Priority policy. The latter favors traffic that is already on the NoC over traffic requesting a local input port (injection).

Note that similar chips with more cores are also available, such as Mellanox TILE-Gx72 [6]. An overview of the TILE-Gx with 36 cores is shown on Figure 2.7.

#### 2.2.5.3 Kalray MPPA series

Kalray MPPA-256 [7] is a 256-core chip. It is organized in 16 tiles connected by two $4 \times 4$ 2D-torus NoC. One, the *Data-NoC*, is dedicated to data transfers, the other handles small control messages (*Control-NoC*). Buffers are located at the outputs routers, and there are 4 of these buffers at each output (one for each of the 4 other interfaces). The arbitration between contending flows is round-robin, and unlike the TILE-Gx, flows crossing a router without interface conflict do not delay each

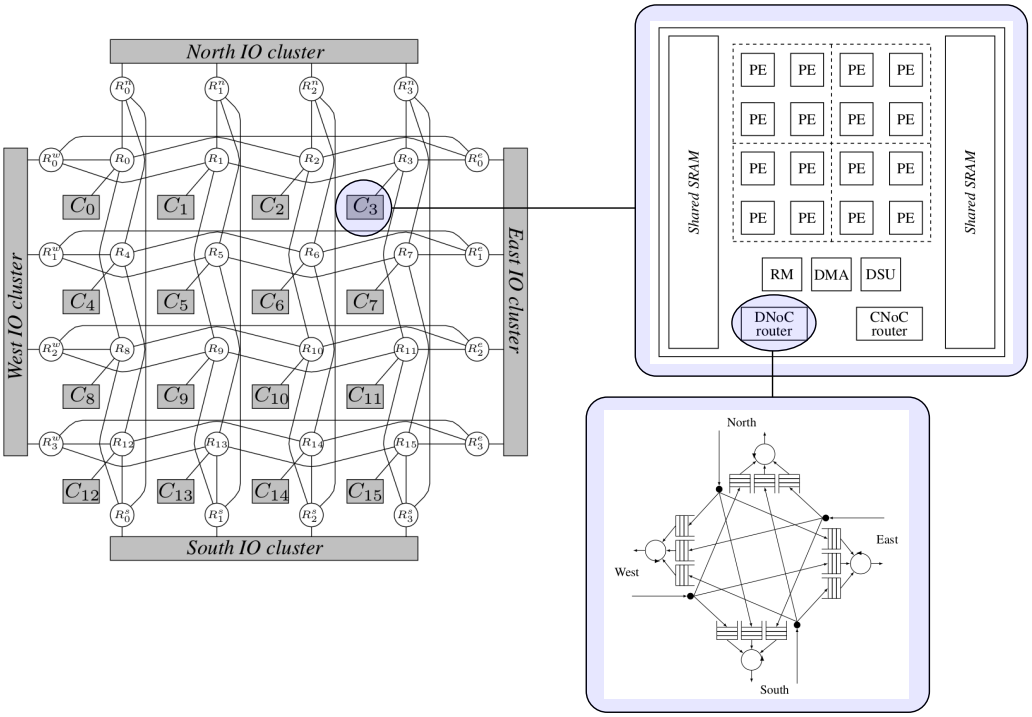



Figure 2.8 – Kalray MPPA overview: NoC (top left), tile (top right) and router (bottom)

other.

Both NoCs support source routing. All turns a packet must take to reach its destination are written by software in the header before the packet is sent. Flow control relies on traffic shapers, that can throttle the injection rate of flows in the NoC and subsequently ensure that buffers are never full (thus no backpressure happens).

NoCs also handle multicast (deliver the packet to some of the nodes on its route) and broadcast (deliver the packet to all nodes on its route). Note that these definitions of multicast and broadcast differ from what is usually assumed in classical networks.

Each tile contains 16 general purpose cores, an additional core to manage resources of the local tile (resource manager or RM), a DMA device to send data over the NoC, and memory.

We present the architecture of the Kalray MPPA on Figure 2.8. We also included the architecture of a router, as it is significantly different from many other manycore NoC routers.

| | Predictability | Scalability | Complexity | Restriction |
|---|---|---|---|---|
| **Topology** | | ✓ | ✓ | mesh and torus |
| **Forwarding** | ✓ | | ✓ | wormhole |
| **Flow control** | ✓ | | | delay-based |
| **Arbitration** | ✓ | | ✓ | RR, WRR, FP |
| **VCs** | | | ✓ | implementing FP |
| **Routing** | ✓ | ✓ | | deterministic, distributed |

Table 2.4 – Summary of design choices and their impact

## 2.3 Discussion: NoCs and Real-Time Systems

In this section, we first summarize our study of NoCs architectures and design choices and their impact on real-time requirements and challenges. Then, with the insight of existing architectures, we explain what our focus will be in the next sections. We tackle the predictability requirement by considering paradigms that favor deterministic behavior, so that bounding the worst-case is possible (Req. 3). Similarly, we discard paradigms that would prevent the system to scale (Challenge 1). Finally, we favor solutions that decrease memory requirements and on-chip complexity (Challenge 2).

Table 2.4 recaps the various design choices in NoCs for manycore architectures. We highlight the main impacts they have on requirements and challenges by putting a check mark ✓ in the appropriate cell, and use this insight to determine whether we restrict the focus of our work to specific paradigms. These restrictions are shown in the last column. No check mark in a column does not mean the considered design paradigm has no impact on the corresponding requirement, but rather that its impact is limited or considered not crucial, and therefore will not restrict the focus of our work.

From a predictability point of view, we want to avoid techniques that do not provide worst-case guarantees. In terms of topologies, there is no predictability-related incentive to pick one topology over another. The relevant criteria to favor one paradigm over another have to do with scalability and on-chip place-and-route complexity. Commercially available chips remain generally based on 2D-mesh and torus, and occasionally 3D-mesh. They keep the on-chip wiring simple, are adapted to routing algorithms that are simple to implement, and they scale well, especially

if they interconnect tiles with several PEs (like Intel SCC and Kalray MPPA). In that respect, they comply with Challenges 1 and 2. To the best of our knowledge, there is no strong incentive to use more complex topologies, at least not yet.

Forwarding techniques we reviewed impact predictability, but none of them has a decisive advantage over the other on predictability-related issues. Congestion, when it happens, will cause delays that are challenging to predict. The most relevant criteria in that matter regard other performance metrics (energy consumption, on-chip area, complexity...). We favor wormhole routing because it is widely used and allows to reduce the amount of memory needed at each router.

Deflection-based and loss-based flow control mechanisms suffer from an intrinsic unpredictability that makes worst-case scenarios extremely difficult to anticipate, and therefore are not adapted to a real-time context. Besides, most NoCs implement lossless transmission. The latter is usually credit-based, although some platform (like Kalray MPPA) additionally provide traffic shaping mechanisms at the injection level to leverage congestion-related delays. Thus, in the remainder of this report, we will consider wormhole NoCs with credit-based flow control.

Arbitration mechanisms also impact predictability, without being deciding factors. Round-Robin, Weighted Round-Robin and Fixed Priority exhibit a better predictability than First-Come First-Served. Fixed-Priority also allows to provide different services to traffic or tasks of different criticality levels, which may justify the additional complexity it induces. Many COTS platform use Round-Robin. Hence, we will consider the most common arbitration policies: Round-Robin and Fixed-Priority. We will also assume Fixed-Priority arbitration is implemented using virtual channels.

Finally, adaptive routing algorithms are not ideal choices as they introduce a lot of possible outcomes in a packet transmission that are hard to anticipate. Moreover, scalability concerns favor distributed routing algorithms over centralized ones. Routing algorithms are generally dimension-ordered when they are hardware-based. Kalray MPPA relies on software configurable source routing, which is highly configurable and allow to implement many different algorithms, but always in a source-routing way. Most of the available routing algorithms are deterministic, and as such comply with the derived requirements for NoC-based real-time systems. It appears, reading the literature, that adaptive routing may be justified only for non

critical systems, or systems that do not require real-time guarantees. Moreover, such a choice in routing algorithm should be motivated by an appropriate study to confirm it can improve overall performance.
Therefore, we will do our best to make our work routing algorithm-independent, and instead only assume the chosen one is deterministic and distributed.

Finally, we want to point out that a certain number of works in the literature introduce architectures specifically designed to provide real-time guarantees. For instance, authors in [47] present a NoC architecture able to handle critical communication with service guarantee needs while using the residual service for best-effort traffic. The work of [48] focuses on providing support for hard real-time traffic with a circuit-switching-like forwarding technique. More recently, an approach based on a dynamic path establishment was proposed in [49]. It is implemented at network layer and relies on a centralized control node to handle requests, to determine if adding a new real-time flow to the existing set while ensuring deadlines are met is possible.
Although such works are very interesting, we focus our work on providing performance bounds on architectures that do not have any special support for real-time applications and are suitable for best-effort traffic. The reason to that is that most widespread COTS architectures that we are aware of are originally designed with no primary focus on real-time applications [7, 6, 5, 8]. Second, exclusive real-time environment is often not suitable for best effort traffic [48], or causes resources to be underutilized in the average case, which may lead to overdimensioning the resources for best effort communication.

As a result, it appears that design choices alone do not allow to fulfill and favor the desired requirements and challenges mentioned in Section 2.1.3.
There is a strong need for means and formal proof to guarantee Requirements 1 and 2 to run real-time applications on manycore architectures. Besides, predictability will also be impacted by other factors that are not platform-related, for instance, mapping applications and tasks on a manycore chip. We outline two fields that are worth investigating.

First, real-time analysis of a wormhole NoC is a difficult problem, due to sophisticated congestion patterns that are (i) hard to avoid without severely impacting the utilization rate of the network links; (ii) hard to predict and model, because of the number of different configuration that can delay one packet; (iii) crucial to take into

account, as NoC delays directly impacts the execution of tasks that communicate using the NoC.
Second, design Space Exploration (including resource dimensioning, choosing between several COTS architectures) and task mapping (assigning application and tasks to processing elements of the chip) are sensitive to NoC communication performance bounds. They also have a huge impact on the network load (and therefore on the delays experienced by the flows), as geographical location of tasks generating a lot of traffic considerably changes the flow pattern.

## 2.4 Conclusion

Even assuming deterministic routing, lossless transmission and appropriate arbitration, many challenges remain to provide real-time guarantees for on-chip communication and consequently ensure safe execution of a real-time application on a manycore platform.
These challenges mostly regard timing analysis of NoCs and system design – both design space exploration and software/hardware mapping.
We see these two aspects as central and relevant issues to tackle in order to provide methods and tools for analysis and design of manycore, NoC-based architectures for real-time applications. Hence, in the next chapter, we will review the existing works addressing these two issues.

*Jadis sur le sable normand,*
*Alors que se couchaient le soleil et la lune,*
*D'amitié nous fîmes serment,*
*Enveloppés des sourds grondements de Neptune.*

*
* *

Chapter 3

# State of the Art

## Contents


**3.1 Timing Analysis of NoCs . . . . . 32**
3.1.1 Overview . . . . . 32
3.1.2 Scheduling Theory . . . . . 33
3.1.3 Compositional Performance Analysis . . . . . 36
3.1.4 Recursive Calculus . . . . . 37
3.1.5 Network Calculus . . . . . 38
3.1.6 Discussion . . . . . 40
**3.2 System Design and Software/Hardware Mapping . . . . . 41**
3.2.1 Design Space Exploration . . . . . 42
3.2.2 Task and Application Mapping on Manycore Architectures . 44
3.2.3 Discussion . . . . . 48
**3.3 Conclusion . . . . . 48**


In this chapter, we review the main existing works in NoC timing analysis and design space exploration. We identify the pros and cons of each approach and highlight our contributions, that will be developed in the following chapters.

There are two main parts to this review. Section 3.1, on one hand, focuses on timing analysis of NoCs. First, we explain why some methods of performance analysis are unfit for real-time and critical systems. Then, we recap the main approaches that are relevant to Timing Analysis of wormhole NoCs. We detail each of them in Sections 3.1.2 to 3.1.5, by briefly explaining the underlying theoretical elements before reviewing the most significant contributions in the field. Finally, Section 3.1.6 presents a synthetic table of relevant approaches and their limitations, and gives an overview of our main contributions for timing analysis of NoC-based platforms.

Section 3.2, on the other hand, is dedicated to problematics of system design and mapping software elements onto hardware platforms. We first review general DSE

approaches. Then, we focus on real-time application mapping on manycore architectures. We detail the main methods that address the problem and the most recent contributions. We also review a few approaches based on execution models and environments for predictable execution. We recap this part with an outline of our contributions.

## 3.1 Timing Analysis of NoCs

### 3.1.1 Overview

To evaluate the performance of a NoC, it is possible to use simulation. The aim is to reproduce the behavior of the system according to a model of its functionalities and architecture and measure metrics that are relevant to the evaluation goals. The more detailed and accurate the model is, the more insight the simulation will yield, but the more time- and resource-consuming it will be. Examples of NoC simulators include Noxim [50], Booksim, HNOCS. Some of them, like Noxim, are cycle-accurate. Although simulation does not provide worst-case bounds on end-to-end delays, it may be used to assess the tightness of the worst-case bounds obtained with analytical models.

If the system is too complex for simulation to give relevant results in a reasonable time, another approach is to use probabilistic models. Queueing theory-based models of NoCs have been developed in the past [51, 52]. For instance, a model detailed in [52] takes as input an application communication graph, a topology graph of the NoC, a mapping of the applications and a routing matrix, and outputs the average latency for each packet. The approach in [51] is different: the authors develop an algorithm based on a queueing model to assign an optimal depth value to each buffer. However, an inherent limitation of probabilistic models is that they give statistical results. Thus, they are not able to provide worst-case delay bounds.

For real-time systems, timing analysis is even more crucial. Neither simulations and experiments nor probabilistic approaches are sufficient to prove the system works as expected in the worst case, *i.e.* they do not guarantee that the worst-case execution time (in the case of an application) or the worst-case delay (for communications or data transmission) is covered. Instead, formal proof of the safe behavior of the system is needed. In particular, this requires to bound NoC communication delays, which can be done by computing (a bound on) the worst-case end-to-end delays

for all data flows using the NoC.

The exact worst-case end-to-end delay is generally hard to compute for complex systems. Hence, a classical approach is to determine an upper bound to the worst-case, which essentially comes down to a trade-of between complexity of the model and tightness of the result. On one hand, the more complex the model is, the more accurate the bound will be, but the more time it needs to output results. It is also more error-prone and harder to formally verify. On the other hand, overestimating the worst-case bound may incite to overdimension the resources, which will impede the efficiency of the system.
The most relevant worst-case timing analysis methods are mainly: Scheduling Theory (ST), Compositional Performance Analysis (CPA), Recursive Calculus (RC), and Network Calculus (NC). In the following sections, we will review ST-based, CPA-based, RC-based and NC-based methods for timing analysis of NoCs, as well as a couple of specific approaches.

### 3.1.2 Scheduling Theory

Scheduling Theory is originally used to bound response times of a set of tasks competing for the use of one or several computing resources [53, 54]. The idea is as follows [54]. We consider a set of tasks with priorities that must execute on a computing resource, and for each task $i$ (periodic or sporadic), the following associated parameters:

- $C_i$, the worst-case execution time, *i.e.* the time needed to execute the task with the available resource(s) when there is no interference;
- $T_i$, the lower bound on the time between two consecutive arrivals of $i$ (if the task is periodic, $T_i$ is its period);
- $D_i$, the deadline of the task.

The worst-case response time (WCRT) of the task $i$, that is the maximum time (counted from the task release) after which the task has been executed, is denoted $R_i$. We also use $hp(i)$ to denote the set of tasks of a higher priority than $i$. To simplify the notations and focus only on the computation principle, we consider the task $i$ can only be delayed by the execution of higher priority tasks, and we neglect the overhead due to context switches and other interferences with a constant duration. To bound the worst-case interference on task $i$ from other tasks, one can notice that during the duration of $i$ response time, there may be $n_j$ arrivals of a

task $j$, with

$$n_j = \left\lceil \frac{R_i}{T_j} \right\rceil$$

The corresponding duration of execution of this task would then be $n_j C_j$. Since only higher priority tasks may preempt the execution of task $i$, the total interference suffered by task $i$ over its execution due to higher priority tasks, denoted $I_i$, equals:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Then, we can write the worst-case response time of $i$ as:

$$R_i = C_i + I_i$$

that is the sum of the computation time of $i$ and the interference from other tasks. Combining these two equations, we get:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

This can be solved using an iterative computation, based on a sequence $(R_i^n)_{n \in \mathbb{N}}$ that may converge under the right conditions. The task set is schedulable if, for any task $i$, the sequence converges and $R_i \leq D_i$.
Enhancements of this result include for instance accounting for a release jitter $J_i$ for each task, that is considering the maximum duration the task may have to wait upon its arrival before being released; integrating other constraints, such as task dependencies; modeling other interference types, such as concurrent memory access in multiprocessor systems, etc.

Principles of Scheduling Theory have been adapted to bound message transmission delays in networks in [55]. The idea is to consider that flows need to use resources to reach their destination, and as such are conceptually similar to a task and its execution needs; while resources are network elements. The main problematic in that case is to refine interference model to accurately capture the complex blocking phenomenons that occur in a wormhole NoC.
In [56], these principles were applied to worst-case timing analysis of wormhole NoCs supporting multiple VCs and with no priority sharing. It is based on distinguishing two types of impact on the flow under analysis, called flow of interest (*foi*), caused

by higher priority flows:

- direct interference – the *foi* shares a physical resource with the contending flow, and can be delayed by it;
- indirect interference – the *foi* does not share any resource with the contending flow, but there is at least one intermediate contending flow between them that has a direct relationship with the *foi* or an intermediate flow on one hand, and the contending flow or another intermediate flow on the other hand.

This method has been extended in [41] to support priority sharing. It allows to consider that several flows can have the same priority level and share the associated VC. To handle this extended hypothesis, the authors introduce direct and indirect blocking, that refer to the previously defined notions of direct and indirect interference respectively, but caused by same-priority flows (instead of higher-priority flows).

However, the latter approach may lead to overly pessimistic results for large NoCs with a high number of flows and a limited number of virtual channels. Moreover, the authors in [57] have proved later that the method in [56] could be optimistic in specific situations. They refined the response time analysis through the distinction between downstream and upstream indirect interferences, which leads to a deeper understanding of the problem. Essentially, this distinction takes into account whether indirect interference propagates to the flow of interest by going upstream the path of intermediate flow, or downstream. Although the authors exhibited a case of the "multi-point progressive blocking" (MPB) problem that caused the optimism in previous ST-based works, their initial approach still suffered from optimistic behaviour, that they later corrected in [58]. However, they focus only on configurations with no priority sharing, which limits the applicability of such a proposal.

Enhancements of the work in [41] have been detailed in [59], where refined interference patterns between flows have been introduced through accounting for the physical contention domain impact, but considering only one-flit size buffers. Finally, [60] attempted to refine the model of [58], noticing that the amount of flits causing MPB-related latency to the *foi* cannot exceed the total buffer size on the considered part of the *foi* path to its destination. Nonetheless they provided no formal proof to their approach and instead relied on experimental results to support the validity of their model.

A more thorough work was presented in [42], but still suffered from the non-priority shared assumption. Authors instead rely on a per-node mapping allowing each

flow to have the exclusive use of its VC. They claim the number of available VCs in modern platforms and the foreseen number of VCs on manycore chips in a near future would be sufficient to make such mappings possible, allowing this method to scale. This assumption could be difficult to maintain for configurations with many traffic flows. Moreover, it requires the mapping to be done offline, prior to the system being deployed, and offers little to no possibility of dynamically adding flows (even flows with no real-time constraints).

### 3.1.3 Compositional Performance Analysis

Compositional Performance Analysis [61, 62] was introduced as a framework to derive worst-case timing behavior of embedded real-time systems. The formalism it uses is similar to ST-based approaches, but it allows in addition to use existing models for local, independent analysis of a part of the considered system, and link the results of different parts to get a global model. CPA is based on event-based models for resources and tasks, and provides performance bounds for different metrics such as buffer size and end-to-end delay.

Recently, the authors in [63] have developed a model for wormhole NoCs worst-case analysis based on CPA. The underlying timing analysis is based on the busy window approach [64]. They account for different types of blocking that the *foi* can undergo at a given router. These are as follows:

- Direct output blocking – the *foi* shares an output port with another flow, but no input port;
- Direct input blocking – the *foi* shares an input port with another flow, but no output port;
- Overlapping – the *foi* shares an input port and an output port with another flow;
- Indirect output blocking – the *foi* experiences direct output blocking from a flow $j$ that experiences direct input blocking from another flow $k$, without the *foi* experiencing any direct blocking from flow $k$.

All blocking delays are bounded and summed over the path of the *foi* to compute the end-to-end delay bound. This approach supports priority sharing and VC sharing, but ignores buffer backpressure.

Afterwards, this work was extended in [1] to support backpressure by modeling the

additional blocking delay caused by feedback control when buffers are full. This approach also refines the computation of the blocking delay caused by contending flows. However, the presented analysis has considered only a single VC and buffer sizes that do not go below one packet. This analysis has also ignored the flows serialization phenomena.

### 3.1.4 Recursive Calculus

Recursive Calculus was introduced in [65] to bound end-to-end flow delay on SpaceWire Networks, that also use wormhole routing. It relies on considering the possible contention on each link of the path of the *foi*. The authors compute a delay for each link of the path, integrating the impact of contending flows (flows sharing resources with the *foi*). Since these contending flows may be delayed as well, the inferred delay on the *foi* will be impacted. That is why the approach to compute the delay on a link is recursive.
This method as it is presented did not account for traffic specificities such as flow rate. It also did not cover priority mechanism for flows.

A recent work [66] proposed a revision of Recursive Calculus. They first exhibit a scenario with buffers that can hold more than one single flit and show RC gives an optimistic bound on that counter example. Then, they detail a revision of RC to cover such cases. The main idea of the approach is to refine the possible occupation of buffers. Noticing that one buffer can contain flits of packets belonging to different flows, they use a bound on the number of partial packets and complete packets in a given buffer. They integrate this result into a corresponding "maximum buffer delay". This term can either be computed using Integer Linear Programming, or bounded for a less computationally-expensive result (at the expense of tightness), and integrated when performing the analysis. They are also able to model packet fragmentation (absent from most other approaches). Packet fragmentation is the way to transmit an amount of data exceeding the maximal packet size by splitting the data chunk into several packets.
However, their work has a similar limitation as in [65]. Traffic specificities such as packet inter-arrival time are not taken into account. Moreover, the model does not allow to have multiple VCs.

In [67], the authors present an ad hoc method for deriving worst-case traversal time of wormhole NoCs in Tilera-like manycore platforms. Their approach improves

Recursive Calculus method by taking into account the pipeline effect in wormhole routing. They present three properties to refine possible contention effects on the *foi* and they integrate them in a recursive algorithm to compute a bound on end-to-end delay. The first one derives the local worst-case scenario at a given router from the point of view of the *foi*, allowing to make the general assumption that this type of scenario happens at every contention point and thus covering the worst-case delay. The second and third one bound the delay undergone by the *foi* due the contention with a flow sharing resources, depending on the contention undergone by these flows. In particular, the third property partially captures the packet spreading phenomenon in the NoC when contention occurs, but only when assuming buffers are one flit deep.
Besides, the model does not support Virtual Channels and considers only Round-Robin as the arbitration mechanism between router inputs.

Finally, we mention the work in [13] as an interesting enhancement of [67]. It extends the NoC analysis method to data flows between the cores and the I/O devices. In a context where external data streams are received by the chip, this approach allows to ensure that incoming Ethernet frames are not dropped due to congestion on the NoC.

### 3.1.5 Network Calculus

Network Calculus (NC) is a deterministic queueing theory based on $(\min, +)$ algebra. It was first introduced in [68] and developed in [69].
The main principle of this theory is to model traffic using cumulative functions, that is a function of time counting the amount of data injected in the network at one point. The cumulative function of a traffic flow can be bounded with an arrival curve, whereas the minimal service that can be provided by each of the network elements is modeled with a service curve. Subsequently, one can derive a bound on (i) the worst-case delay, that is the maximal duration it takes for a flit entering the system to be delivered at the output; (ii) the worst-case backlog, that is the maximal amount of data held inside the system.
NC has had a certain number of additional analytic contributions. We will particularly use one of these principles, known as “Pay Multiplexing Only Once” (PMOO), introduced in [70]. It models the serialization phenomenon of data flows when they cross several consecutive network nodes. This result improves the tightness of the

timing analysis by allowing to pay the additional delay caused by multiplexing flows only once.
Basics of NC and theorems relevant to the work in this report are presented in Appendix B.1, along with the notations used throughout the next pages.

Recent works based on Network Calculus have tackled timing analysis of wormhole networks, including SpaceWire Network [71, 72] and NoCs. Authors in [73] develop a NC-based analysis of delay bounds in a NoC, based on the work in [74] for sink-tree networks. They distinguish three basic contention patterns (nested, parallel, crossed) between the *foi* and two contending flows, and detail the associated service curve computation using Theorem 5 in Appendix B.1 for residual service with aggregate traffic. They construct a contention tree to capture impact of contending flows on one another and account for indirect interference, and apply the results of the basic patterns analysis to derive the end-to-end service curve. However, their model does not take into account the buffer size and ignore the effect of backpressure.

They refined their work in [75], focusing on wormhole NoCs and solving the chain blocking problem resulting from backpressure in a recursive way. They model the flow control system as a router component, whose behavior depends on the following router. This method infers complex fixed-point problems solving and has been validated only on relatively small NoCs with a simple flows configuration; thus limiting the applicability and scalability of their approach. Moreover, they have considered only a single-VC NoC routers.

Authors in [76] have provided tighter delay bounds, using improved arrival and service curves while taking buffer size and flows serialization into account. However, their approach seeks to provide a buffer size threshold to avoid the buffer backpressure. In addition, they do not consider wormhole routing as the switching technique; thus avoiding the complex chain blocking issue.
In [77], the authors computed end-to-end delay bounds on the Kalray MPPA2 Processor based on Network Calculus. This work is interesting from a practical perspective but it remains very specific to one architecture. The authors also rely on the Kalray MPPA on-chip traffic shapers to avoid backpressure, and the buffer size is significantly bigger than all considered packet sizes.

In [78] authors have started the exploration of the buffer size impact on the

interference patterns. The considered wormhole routers do not support the VC concept and the presented approach does not integrate the flows serialization phenomena.

### 3.1.6 Discussion

In the light of our review, we summarize the main advantages and drawbacks of reviewed approaches. We particularly consider: (i) the use of *wormhole routing* with *multiple VCs*; (ii) the support of *VCs Sharing* (*i.e.* several traffic classes per VC) and *priority Sharing* (*i.e.* several traffic flows per priority level) in each router; (iii) the integration of the *buffer size* and the flows *serialization* impacts ; (iv) the applicability for general traffic model and heterogeneous architectures assumptions.

We present a synthesis of the applicability of recent work in Table 3.1. Essentially, ST-based approaches suffer from their theoretical complexity that lead to optimism in the timing analysis in some previously published papers [57, 41]. Most recent works addressed this problem [58, 42] but do not take into account priority sharing and VC sharing, which may be a problem when implementing priority classes on NoCs with a limited number of VCs.
CPA-based and RC-based approaches that we are aware of usually lack the genericity that would make them applicable to a wide range of NoCs (VC support, arbitrary buffer size).
NC-based approaches show drawbacks in the limited applicability of the results [75, 77, 78], and occasionally in the surrounding analysis.

We choose to use Network Calculus for our contributions in worst-case timing analysis of NoCs for the following reasons:

- Network Calculus provides advanced theoretical results for worst-case timing analysis of networks. These cover not only the modeling of the service offered by network elements but also combinations of network elements, with principles such as pay burst only once and PMOO,
- NC is a field-proven method. It has been used to certify AFDX (Avionic Full Duplex Switched Ethernet). AFDX is featured in particular in the most recent Airbus aircraft such as A380;
- Due to the modularity of the way network elements are modeled, a NC-

based model can be updated and improved without having to alter the whole methodology.

Our first contribution in worst-case timing analysis will be an analytical model for NoC worst-case analysis, based on Network Calculus. It will integrate buffer size impact, flow serialization, VC sharing and priority sharing, be applicable to homogeneous platforms and support a basic traffic model.

Our second contribution will aim at tackling limitations of the first one. First, we will extend the network model to heterogeneous architectures and generalize the traffic model to cover a wider range of data flows. Second, we will improve the computational aspect of the model to improve the scalability of our approach.

| Approach | ST | | | | CPA | | RC | | | NC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Contribution | [58] | [41] | [59] | [42] | [1] | [63] | [65] | [66] | [67] | [75] | [76] | [77] |
| wormhole | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ |
| multiple VCs | ✓ | ✓ | ✓ | ✓ | | ✓ | | | | | ✓ | ✓ |
| priority sharing | | ✓ | ✓ | | ✓ | ✓ | | | | ✓ | ✓ | |
| VCs sharing | | | | | | ✓ | | | | | | |
| flows serialization | | | | ✓ | | ✓ | | | ✓ | | ✓ | ✓ |
| $B = 1$ flit | ✓ | | ✓ | ✓ | | | | ✓ | ✓ | ✓ | | |
| $L \leq B$ | ✓ | | | ✓ | ✓ | ✓ | | ✓ | | ✓ | ✓ | |
| $B \leq L$ | ✓ | | | ✓ | | | | ✓ | | ✓ | | ✓ |

Table 3.1 – Summary of approaches for timing analysis of wormhole NoCs (N.E.S.: not explicitly specified)

## 3.2 System Design and Software/Hardware Mapping

This section focuses on the design of NoC-based systems, especially – but not limited to – manycore-architecture-based systems, and keeping in mind real-time problematics. Essentially, system design involves two different areas of focus. First, starting from a relatively high level view (*e.g.* system-level), one should be able to narrow things down to one or a few architectures that are best suited to run all the applications constituting the system.

Second, given a hardware, general purpose architecture and a set of software-implemented (or implementable) functions, *e.g.* applications and tasks, another problem is to map the tasks to the processing elements of the architecture. This aspect has a particular importance when the chosen platform is very generic (for instance a COTS, general purpose manycore chip) or when many functions of the system will be software-implemented and run on a general purpose processing ele-

ment instead of a specialized hardware component (for instance an ASIC).
These two aspects cannot always be clearly separated in the design process, but we believe the underlying challenges raised by each of them deserve a section on their own. Therefore, we will tackle general DSE considerations in Section 3.2.1, and more specific mapping of tasks on NoC-based manycore architectures in Section 3.2.2.

### 3.2.1 Design Space Exploration

Increasing complexity of systems, coupled with the necessity to reduce production costs and design-to-market time, motivated the development of systematic and rigorous design methodologies.
Design space exploration has been thoroughly tackled with various approaches. Relevant aspects to take into account when considering an approach are (i) the level (or levels) of abstraction offered – system-level, Transaction Level Modeling (TLM), RTL (Register Transfer Level), CABA (Cycle-Accurate Bit-Accurate) – and how they are adapted to the steps of the design process; (ii) the orthogonality of functional and architectural models, as design process rapidly becomes tedious if these two views are not independent; (iii) the available design space exploration and mapping technique(s), if applicable.
One should also consider some practical aspects of the chosen methodology and/or the tool implementing it: (i) the input format (SysML, SystemC...); (ii) the available models for architecture components (processing components, communication media, memories, etc.); (iii) the ability to automatically generate code from a higher level view;
Several works have tackled NoC design. In [79], the author presents a design flow for telecommunication NoC-based systems. The method uses an application graph and an architecture graph to first perform an Algorithm-Architecture Adequation phase and determine several parameters for the NoC (size of the FIFO queues, network interface dimensioning, topology choice...). The design flow includes a tool to generate SystemC code for simulation and validation. However, real-time constraints of the system can only be checked with simulation results.
The approach detailed in [80] also considers an application-focused methodology to design a NoC. The tool presented, based on Arteris, allows designers to identify the limitations of an architecture in regard to the application considered and its requirements, and iterate the process to converge towards an implementation that satisfies the requirements. It also features a practical use case to show the applica-

bility of the approach on a real design problem. The focus on a specific application makes it hard to determine if such an approach would be adequate to tackle design space exploration using more generic NoC architectures.
In [81], the authors propose an extension to the MPSoCSim simulator that integrates a SystemC NoC model. It extends the original mesh NoC model of MPSoCSim to support multi and manycore clustered architectures, with more than one processor per node. Nodes architectures can be heterogeneous and the extension allows more flexibility than the original approach, but the architecture remains constrained.

Other works provide frameworks or toolkits for systematic design flow and they allow to model systems at different abstraction levels. Ptolemy [82, 83] addresses heterogeneous systems modeling and simulation. Its primary intent is not DSE. Ptolemy framework models heterogeneous systems that cannot be described with a global approach by using different *models of computation*.
Artemis [84] provides an environment for design space exploration of heterogeneous embedded systems with different abstraction levels, allowing to manage the modeling effort depending on the accuracy goal and the stage of design. It complies with the Y-chart model [85] by separating architecture and application models at the highest abstraction levels, and integrating an explicit mapping step.
Authors in [86] present MoPCoM, a co-design methodology for real-time embedded systems based on the Marte UML profile [87], offering three abstraction levels, the lowest of which can be used for VHDL code generation. This approach also separates, at each abstraction level, the application model from the platform model. A third "allocation" model is used for the mapping.
Another interesting work, particularly adapted to systems such as tiled manycore platforms and NoCs, is detailed in [88]. It consists of a package called Repetitive Structure Modeling (RSM), integrated in the Marte profile and allowing a compact representation of large systems with a regular structure. RSM was successfully used in the framework GASPARD [89] to model massively parallel embedded systems.
In [90], the authors develop a system level environment to perform DSE at a higher abstraction stage than RTL models, allowing simulations that are less time consuming at a stage where some aspects can be left unspecified. The presented approach complies with the Y-chart model: the functional abstraction of the system and the architecture model are handled separately before performing mapping and DSE. The design work flow can be practically implemented using the toolkit TTool [9].

An extended contribution was presented in [91] with an approach enabling to define system partitioning (hardware and software) before refining the models into behavioral and architectural models. The approach is also supported by TTool.
Finally, there has been recent effort to integrate network models and simulation in system design workflows and DSE. In [92], authors take network design aspects into account by adding a dimension to the design space exploration at the TLM refinement step. The proposed methodology allows to model system/network interactions and dependencies. It starts with a system/network partitioning, followed by a specification of system/network interactions. Cosimulation is then made possible using the introduced framework and existing simulation tools synchronized with one another.
In [93], authors tackle the limits of existing modeling approaches for communication architectures by proposing a formal model for network functional architectures. They detail an additional design space exploration step focused on network aspects, integrated in a design flow prior to the mapping and "classic" design space exploration phases. The presented formal model includes an abstraction for communication channels. Nonetheless, the approach does not address algorithmic aspects of DSE. Besides, decoupling network-related DSE from the remaining of the design flow may hide possible interactions or interference between communication infrastructure and system architecture.
The work presented in [94] pursued network architecture integration in design flow tools, using UML models.
Unfortunately, none of these approaches have integrated formal verification of real-time constraints in their workflows.

### 3.2.2 Task and Application Mapping on Manycore Architectures

Given a set of applications, each of which is composed of tasks with various constraints and needs (real-time deadlines, use of memory or external devices, precedence constraints), the problem is to assign each task to a processing resource so that the application(s) can run while satisfying the execution requirements. As mentioned in [95], a distinction is to be made between design-time mapping and run-time mapping strategies. The latter are able to generate a mapping or modify a previously generated mapping at runtime. They can react dynamically to a change in the workload to optimize resource utilization, while design-time mapping strategies are used offline and target systems requiring more determinism

in their execution. As we are mostly interested in real-time applications and critical systems, we will only focus on design-time mapping strategies.

#### 3.2.2.1 Heuristics and Optimum Approximations Methods

Finding an optimal mapping of a set of tasks on a set of processing resources is a NP-hard problem [96, 95]. Beyond a certain problem size, exhaustive algorithms take too long to run to be used in practice. Hence, many approaches rely on an heuristic to orientate the search when exploring possible mappings. Others use branch-and-bound algorithms. Algorithms based on heuristics converge to a solution that may not be optimal but is good enough and can be obtained in a reasonable time. Choosing an heuristic depends on the sought goal, and may regard performance metrics (such as execution time, delay, latency, throughput, etc.) or cost concerns such as energy consumption. In this section, we will review several approximation- and heursitic-based algorithms for task mapping. We stress out, however, that in a real-time context, such algorithms are not sufficient and must be paired with am adequate method of timing analysis to ensure the deadlines are met.

The use of a Genetic Algorithm to converge towards a near-optimal mapping is presented in [97]. The author's approach can be applied to heterogeneous, NoC-based SoCs, limited to mesh topologies. However, the underlying communication timing analysis is very rough, overestimates the delay for wormhole communications and is not congestion aware.
Authors in [98] have detailed an algorithm to map processing cores onto a mesh NoC, ensuring bandwidth requirements are met while minimizing the average communication delay. They extensively evaluate their approach with video processing application and provide comparisons with other state-of-the art algorithms.

In [99], the authors propose a branch-and-bound algorithm to yield a mapping minimizing energy consumption. Their approach uses an application graph to model processing cores and inter-core communication, and an architecture graph to model the tiles and communication channels of the target platform. They express the performance constraint as a condition on the aggregated bandwidth on each of the communication links. This approach suffers from applicability limitations, *e.g.* one computing tile can host at most one core. Besides, the model relies heavily on platform-specific parameters such as XY routing and mesh topology.

The work in [100] takes contention into account in the mapping algorithm. It is based on the formulation of an Integer Linear Programming (ILP) problem, taking into account the number of shared links in the cost function. The presented solving relies on a linear programming approximation of the ILP problem followed by an heuristic, so that the cores that communicate the most are picked first, and mapped to a tile in such a way that reduces path-based interference and distance to the tiles they communicate with. This solution outperforms by far the exhaustive ILP solving algorithms, and the resulting mappings also allow higher packet injection rates than those yielded by the energy-aware approach in [99].

Notice that these last three works [98, 99, 100] make no distinction between a processing core, or IP (Intellectual Property) and the task(s) it performs. In the light of Section 3.2.1, this impacts the flexibility of the approach because changing the task assignments of cores will change the application graph even though the cores remain the same.

More recently, the work presented in [101] explored the impact of mapping on NoC contention and introduced a mapping algorithm minimizing the number of shared paths. The approach uses a task graph to build a tree that models the dependencies and communications between tasks. The tree is then optimized before serving as a basis to map tasks on processing elements. This method can achieve significant predictability, making it interesting in a real-time context. However, its computation of the end-to-end latency when no contention occurs is pessimistic because it doesn't take into account pipeline effect.

The work in [13] tackles task mapping on mesh-based, general purpose manycore chips, but additionally considering the core-to-I/O data flows, and mixed-criticality applications. The mapping strategy follows two steps: (i) assigning rectangular regions of the mesh to applications, starting with critical applications, and placing the corresponding regions next to external interfaces (Ethernet, etc.) when they most need them; (ii) mapping tasks within application regions, minimizing the distance of tasks using I/O ports with the aforementioned ports, and taking into consideration the XY routing policy to avoid congestion on core-to-I/O data flows. This approach remains quite platform-specific, although the presented principles can be applied to similar 2D-mesh-based chips. Noticeably, real-time constraints are accounted for and verified in this approach.

Predictability has been addressed as well in the approach of [102]. Different heuristics can be used for task mapping (which tasks to select first during the mapping phase) and core mapping (where to map the selected task). The presented task mapping heuristics focus either on how much a task communicates, or on defining partitions of tasks that communicate a lot between each other and much less with tasks in other partitions. Core mapping is done either on selecting cores on the basis of their position in the NoC (middle, edge, corner) or by picking adjacent consecutive cores, in a spiral way.
Comparison of the heuristic-based approach with an exhaustive solver is provided, as well as experimental results obtained with a physical 64-core Tilera chip. Although the model does not guarantee deadlines, an interesting contribution of this paper is the proposed temporal frame abstraction (similar to a TDMA technique) that provides additional predictability in the execution.
These last works have in common improving predictability through task mapping. Additionally, [102] provides an additional mechanism to enhance predictable communication on the NoC. Providing predictability guarantees with dedicated mechanisms is one way to tackle determinism in manycore-based real-time systems. We will give an overview of relevant contributions in this area in the next section.

#### 3.2.2.2 Predictability-Enhancing Techniques

In the real-time and critical systems context, some approaches seek to make applications execution as predictable as possible. This is particularly challenging in manycore-based systems, as most of the available platforms are originally designed for best-effort applications, or lack some elements to enforce real-time guarantees.
A first approach is to use hardware extensions. This is what authors in [103] rely on. They add a controller component in the NoC routers to override default arbitration. Alongside, they use hardware locks and split each local RAM into banks, so that several initiators can access the local memory without interfering with one another. Finally, they define a set of mapping rules and use scheduling tables for NoC communication. Their method generates the mapping and the appropriate code to control the multiplexers of the NoC routers and the execution on each core. The main drawback of such a method is that it relies on hardware extensions, thus limiting its applicability and preventing the use of COTS chips as target platforms.

Another solution was presented by the authors in [46]. Using an execution model

based on 4 rules, they formalized the mapping problem and were able to execute a set of real-time tasks on three different COTS architectures, with guarantees on the timing requirements.
Their approach takes into account the precedence constraints between tasks and the cost of inter-task communication. The model outputs a spatial mapping and an offline schedule ensuring safe execution of the task set.
Similarly, the thesis of Quentin Perret [30] investigated the predictable execution of real-time applications. In particular, it determines requirements for predictable execution and implements an execution model for the Kalray MPPA 256 manycore processor.

### 3.2.3 Discussion

Many approaches in the literature have tackled DSE for NoC-based architectures. Multi-level design space exploration is less common, and overall, to the best of our knowledge, integrating formal real-time constraints checking in the workflow has not been done before.
As far as mapping heuristics are concerned, several methods have been proposed and tested. They are able to provide a good trade-off between obtaining an optimal solution and reducing the computation costs. Incentives to use one heuristic over the other depends mainly on the target system requirements. It appears the greatest limitation of these approaches is their specialization. Moreover, few of them take into account real-time constraints or formally prove the mapping they obtain complies with timing requirements.
Therefore, our third contribution will present a generic and modular system design methodology for NoC-based architectures, allowing to verify real-time constraints at an early design step. Such a methodology will combine Network Calculus and Simulation to evaluate system performance and can be used for timing-aware design space exploration. It will not be our focus to propose or implement a mapping heuristic, but rather to allow the use of any mapping heuristic in the workflow.

## 3.3 Conclusion

Our review of state-of-the-art approaches in NoC worst-case timing analysis and design space exploration for NoC-based architectures exhibited mostly two aspects. First, existing worst-case timing analysis techniques for wormhole NoCs generally rely on hypotheses limiting their applicability. Hence, we will present an approach

addressing these limitations. Then, we will extend it to improve its scalability and its applicability domain.
Second, design space exploration approaches of NoC-based systems do not often integrate analytical methods to verify timing constraints. Hence, we will propose a methodology combining both our timing analysis approach and classic design space exploration workflow to speed up the design process. We will not tackle mapping algorithms, as many works have addressed this problem. Instead, we aim at making our methodology able to integrate such algorithms.
The next chapters detail our contributions.

*Eveillé tôt, ordonné, vif,*
*De nature arrangeante, il m'a subi deux ans.*
*Où est sa mystérieuse griffe ?*

*Cherchez l'alpha des vers : son nom y est présent !*

*
* *

# Part II

# Contributions

*We need not to be let alone. We need to be really bothered once in a while. How long is it since you were really bothered? About something important, about something real?*

—Ray Bradbury, *Fahrenheit 451*

*I find that answer vague and unconvincing.*

—K-2SO, *Rogue One*

Chapter 4

# Buffer-Aware Worst-Case Timing Analysis of Wormhole NoCs for Homogeneous Platforms and CBR Traffic

Contents


**4.1 Introduction . . . . . 54**
**4.2 Assumptions and System Model . . . . . 54**
4.2.1 Network Model . . . . . 54
4.2.2 Flow Model . . . . . 57
**4.3 Approach Overview . . . . . 57**
4.3.1 Buffer-Awareness: An Example . . . . . 57
4.3.2 Main Steps of BATA . . . . . 59
**4.4 Indirect Blocking Analysis . . . . . 61**
**4.5 End-to-End Service Curve Computation . . . . . 65**
4.5.1 Direct Blocking Latency . . . . . 65
4.5.2 Indirect Blocking Latency . . . . . 67
4.5.3 Computation Algorithm . . . . . 69
**4.6 Illustrative Example . . . . . 70**
**4.7 Performance Evaluation . . . . . 73**
4.7.1 Sensitivity Analysis . . . . . 74
4.7.2 Tightness Analysis . . . . . 77
4.7.3 Computational Analysis . . . . . 79
4.7.4 Comparative Study . . . . . 82
**4.8 Conclusions . . . . . 84**

## 4.1 Introduction

This first contribution regarding worst-case timing analysis of wormhole NoCs originated from the necessity to account for buffer size in wormhole NoCs. A classic assumption in networks is to assume buffer overflow never happens, *i.e.* all buffers are large enough to hold traffic that stalls.
While this may be a reasonable assumption in Store and Forward Networks, achieving lossless transmission through a backpressure mechanism in wormhole NoCs with relatively small buffers may lead to the buffers being quickly full when contention occurs.
As pointed out in Chapter 3, most existing approaches in this area have limitations usually related to buffer size, flow serialization or limited applicability of the model. Therefore, our approach detailed in this chapter, based on Network Calculus, will address these issues. We will focus herein on homogeneous platforms and Constant Bit Rate (CBR) traffic.
The remainder of this chapter is organized as follows. We present the assumptions of our approach and the system model in Section 4.2. Then, in Section 4.3, we propose an example to illustrate the importance of buffer awareness and provide an overview of our approach, called Buffer-Aware worst-case Timing Analysis (BATA). In Sections 4.4 to 4.6, we detail the main steps of BATA approach and present an illustrative example. In Section 4.7, we evaluate our model through analytical studies (4.7.1, 4.7.2 and 4.7.3) and provide some comparative analyses with a state-of-the-art approach (4.7.4). Finally, Section 4.8 concludes this chapter. Network Calculus results as well as notations used in this chapter are recalled in Appendix B.

## 4.2 Assumptions and System Model

### 4.2.1 Network Model

Our model can apply to an arbitrary NoC topology as long as the flows are routed in a deterministic, deadlock-free way, and in such a way that flows interfering on their path do not interfere again after they diverge. Nonetheless, we consider the commonly used 2D-mesh topology with input-buffered routers and XY-routing, known for their simplicity and high scalability.[1] Besides, XY-routing is widely used

[1]Note that this model is also applicable without changes to 2D torus topologies and NoCs using a source routing algorithm that verifies the aforementioned properties.

in COTS architectures, *e.g.* on most Tilera-like chips [5].
We consider typical input-buffered 2D-mesh routers with 5 pairs of input-output, namely North (N), South (S), West (W), East (E) and Local (L), as shown on Figure 4.1. Output-buffered routers have buffers located at the output ports instead of the input port but remain similar otherwise.

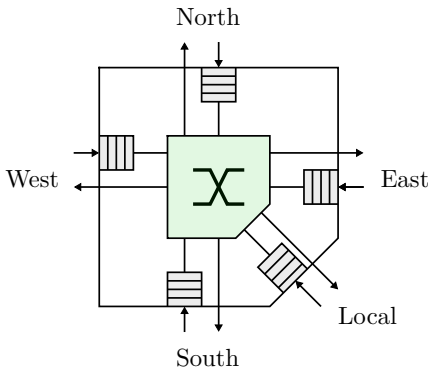



Figure 4.1 – Typical 2D-mesh router

It is worth noticing that NoCs using output-buffered routers can be modeled similarly to input-buffered routers NoCs. The idea is that from a flow point of view, whether the buffer is located at the input or at the output does not change the number of buffers and links crossed by the flow on its path, as introduced in [66].
The notations used in this paper will be introduced as they are needed and are also gathered in Table B.1 in Appendix B.2. As a general rule, upper indexes of a notation $X$ refer to a node or a subset of nodes, while lower indexes refer to a flow. $X_f^r$ means "$X$ at node $r$ for flow $f$".
The considered wormhole NoC routers are similar to the architecture presented in [24], illustrated in Figure 4.2 (left). They implement a priority-based arbitration of VCs and enable flit-level preemption through VCs. The latter can happen if a flow from a higher priority VC asks for an output that is being used by the *foi*. Hence, when the flit being transmitted finishes its transmission, the higher priority flow is granted the use of the output while the *foi* waits. Moreover, each VC has a specific input buffer and supports many traffic classes, *i.e.*, VCs sharing, and many traffic flows may be mapped on the same priority-level, *i.e.*, priority sharing. Finally, the implemented VCs enable the bypass mechanism, previously mentioned in Section 2.2.3 and illustrated in Figure 2.4.
We consider an arbitrary service policy to serve flows belonging to the same VC within the router, *i.e.*, these flows can be from the same traffic class or from different traffic classes mapped on the same VC. This assumption allows us to cover the

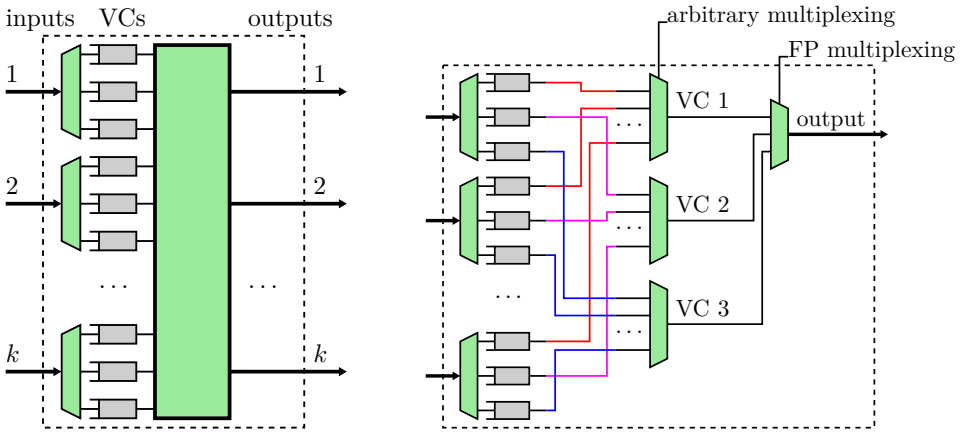



Figure 4.2 – Architecture of an input-buffered router (left) and output multiplexing (right) with the arbitration modeling choices

worst-case behaviors of different service policies, such as FIFO and Round Robin (RR) policies.

Hence, we model such a wormhole NoC router as a set of independent hierarchical multiplexers, where each one represents an output port as shown in Figure 4.2 (bottom). The first arbitration level is based on a blind (arbitrary) service policy to serve all the flows mapped on the same VC level and coming from different *geographical* inputs. The second level implements a preemptive Fixed Priority (FP) policy to serve the flows mapped on different VC levels and using the same output port. It is worth noticing that the independence of the different output ports is guaranteed in our model, due to the integration of the flows serialization phenomena. The latter induces ignoring the interference between the flows entering a router through the same input and exiting through different outputs, since these flows have necessarily arrived through the same output of the previous router, where we have already taken into account their interference.

Based on Network Calculus [68, 69] (see Appendix B for the main concepts used in this thesis), each router-output pair $r$ (that we will refer to as a *node* from now on) has a processing capacity that we model using a rate-latency service curve.

$$\beta^r(t) = R^r(t - T^r)^+$$

$R^r$ represents the minimal processing rate of the router for this output (which is typically expressed in flits per cycle, fpc) and $T^r$ the maximal experienced delay by any flit crossing the router before being processed (which is commonly called routing delay and takes one or few cycles).

For now, we assume the considered NoC architectures are homogeneous, meaning all buffers of all routers have the same size, denoted $B$, and all inter-router links have the same capacity, denoted $R$.

### 4.2.2 Flow Model

The characteristics of each traffic flow $f \in \mathcal{F}$ are modeled with the following leaky bucket arrival curve, which covers numerous different traffic arrival events, such as CBR traffic with or without jitter :

$$\alpha_f(t) = \sigma_f + \rho_f \cdot t$$

This arrival curve integrates the maximal packet length $L_f$ (payload and header in flits), the period or minimal inter-arrival time $P_f$ (in cycles) and the release jitter $J_f$ (in cycles) in the following way :

$$\begin{aligned} \rho_f &= \frac{L_f}{P_f} \\ \sigma_f &= L_f + J_f \cdot \rho_f \end{aligned}$$

For each flow $f$, its path $\mathbb{P}_f$ is the list of nodes (router-outputs) crossed by $f$ from source to destination. Moreover, for any $k$ in appropriate range, $\mathbb{P}_f[k]$ denotes the $k+1^{th}$ node of flow $f$ path (starting at index 0). Therefore, for any $r \in \mathbb{P}_f$, the propagated arrival curve of flow $f$ from its initial source until the node $r$, computed based on Theorem 6 in Appendix B.1, will be denoted:

$$\alpha_f^r(t) = \sigma_f^r + \rho_f^r \cdot t$$

The end-to-end service curve granted to flow $f$ on its whole path will be denoted:

$$\beta_f(t) = R_f \left(t - T_f\right)^+$$

## 4.3 Approach Overview

### 4.3.1 Buffer-Awareness: An Example

The key element to take into account the backpressure phenomenon induced by limited buffer size is based on how packets can spread in the network when stalled. We consider an illustrative example to better understand the impact of the buffer size on the packet spreading (Figure 4.3) and how it affects the interference a flow

can have on another flow when they do not share resources. We make the following assumptions: (i) each buffer can store only one flit; (ii) all flows have 3-flit-long packets; (iii) all flows are mapped to the same VC; (iv) the *foi* is flow 1.

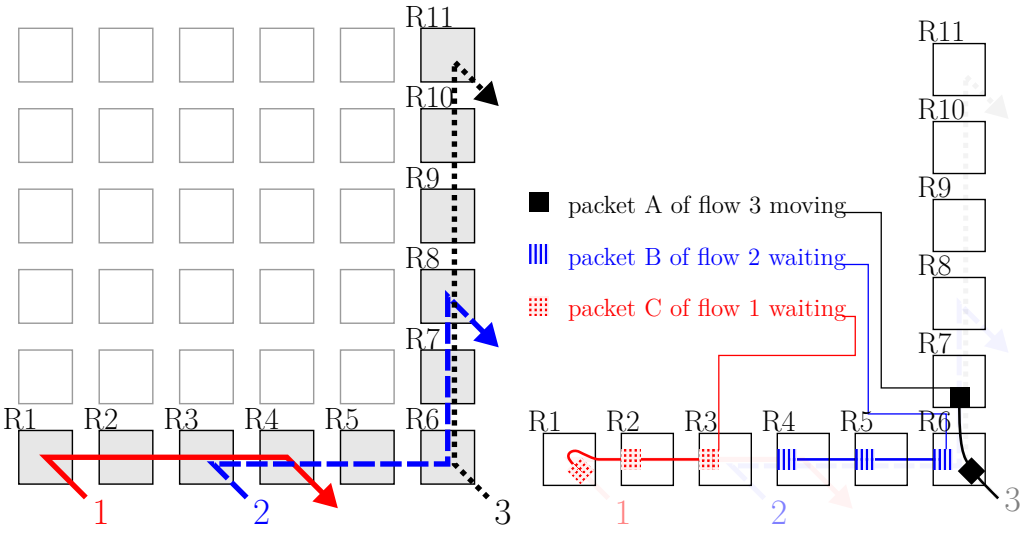



Figure 4.3 – Example configuration (left) and packet stalling (right)

We assume there is a packet A of flow 3 that has just been injected into the NoC and granted the use of the North output port of R6. Simultaneously, a packet B of flow 2 is requesting the same output, but as A is already using it, B has to wait. B is stored in input buffers of R6, R5 and R4. Finally, a packet C of flow 1 has reached R3 and now requests output port East of R3. However, the West input buffer of R4 is occupied by the tail flit of B. Hence, C has to wait. In that case, A blocks C *via* packet B. We say that flow 3 can indirectly block flow 1 even though they do not share resources. Note that such a scenario can happen only if all flows use the same VC. If they do not, one flow can bypass another that is blocked.

Now suppose flow 3 source is one hop further (Figure 4.4). While the modification seems minor, it impacts the indirect blocking possibilities. Consider that flow 3 has a packet A that has just been injected in the network and is using output port North of R7. As before, flow 2 has a packet B in the network that competes with A and has to wait. This time, however, the output requested by B is one hop further on flow 2 path. As a result, B is stored in input buffers of R7, R6 and R5. Finally, flow 1 has injected a packet C into the NoC. Since B is stalling one hop further than before on its path, C can request output port East of R3 and use input buffer West of R4 to reach its destination without contention.

An approach that does not consider buffer sizes would consider that flow 3 may impact flow 1 regardless of the configuration, because both of them share resources

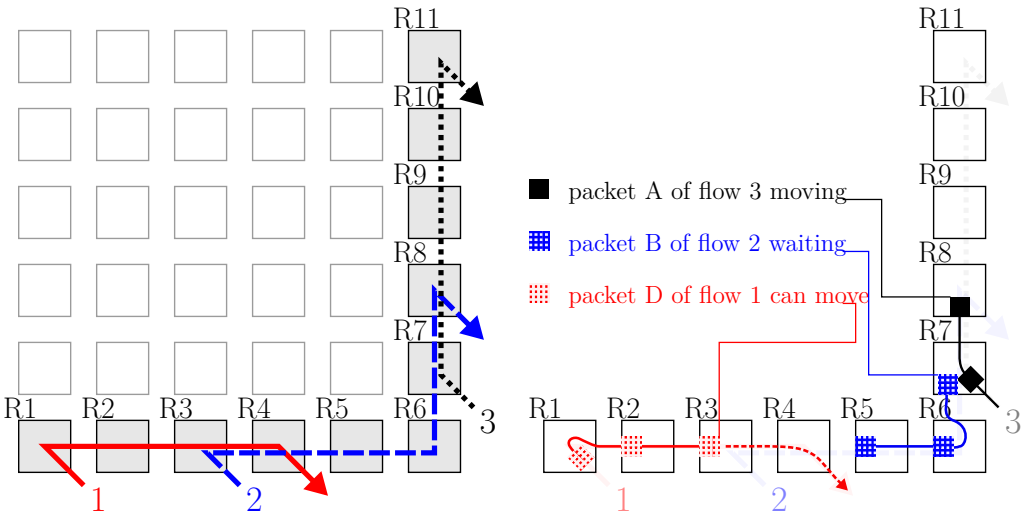



Figure 4.4 – Another configuration (left) where flow 1 cannot be blocked by flow 3 (right)

with flow 2. However, we just showed that it is not necessary. This illustrates the impact of the buffer size on packet spreading.
Hence, the limited buffer size reduces the section of the path on which a blocked packet can in its turn block another one; thus the indirect blocking delay as well.

### 4.3.2 Main Steps of BATA

In order to compute the end-to-end delay bound of a flow $f \in \mathcal{F}$, we present the following high-level view, that we will detail in the next sections. It consists of three parts: the buffer-aware analysis of the indirect blocking set, the service curve computation and the end-to-end delay computation.

**Step 1 – Buffer-aware analysis of the indirect blocking set**: To account for the impact of flows that do not physically share any resource with the *foi* $f$, but can delay it because they impact (directly or indirectly) at least one flow directly blocking $f$, we introduce the *Indirect Blocking set* of $f$, abbreviated IB set and denoted $IB_f$.

**Definition 4.** *The indirect blocking set of a flow $f$ is the set of flows that do not physically share any resource with $f$, but cause a delay to $f$ because they impact (directly or indirectly) at least one flow sharing resources with $f$. It is denoted $IB_f$ and contains pairs of the form {flow id, subpath} to specify, for each flow id, the subpath where a packet of that flow can cause blocking that may propagate to $f$ through backpressure.*

This step takes into account the impact of the limited buffer size on the way a packet can spread on the NoC; thus on $IB_f$, and will be detailed in Section 4.4.

**Step 2 – End-to-end service curve computation**: to get a bound on the end-to-end delay for a *foi* $f$, we need to compute its end-to-end service curve along its path $\mathbb{P}_f$. This service curve is denoted:

$$\beta_f(t) = R_f \left(t - T_f\right)^+ \quad ,$$

where $R_f$ represents the bottleneck rate along the flow path, accounting for directly interfering flows of same and higher priority than $f$, and latency $T_f$ consists of several parts :

$$T_f = T_{DB} + T_{IB} + T_{\mathbb{P}_f} \tag{4.1}$$

where:

- $T_{\mathbb{P}_f}$ is the "base latency", that any flit of $f$ experiences along its path due only to the technological latencies of the crossed routers;
- $T_{DB}$ is the maximum direct blocking latency, due to interference by flows sharing resources with the flow of interest (*foi*). We denote the set of such interfering flows $DB_f$;
- $T_{IB}$ is the maximum indirect blocking latency, due to flows in the indirect blocking set $IB_f$, that can indirectly block $f$ through the buffer backpressure phenomenon.

Aside from the base latency, there are two main components to the end-to-end service curve latency, namely the direct blocking latency and the indirect blocking latency.

Flows contributing to the direct blocking latency are said to be part of the Direct Blocking set of the *foi*, abbreviated DB set, and defined as follows:

**Definition 5.** *Let $f$ be the* foi. *The set of flows that share resources with $f$ on their paths is called the Direct Blocking set of $f$ and denoted $DB_f$. Moreover, the subset of flows in $DB_f$ sharing resources with $f$ along path is denoted $DB_f^{|path}$.*

This step integrates the flows serialization effects using the Pay Multiplex Only Once (PMOO) principle [70], detailed in Appendix B.1, and will be detailed in Section 4.5.1.

Flows contributing to the indirect blocking latency have been determined at the previous step. At this point, we use the IB set to compute $T_{IB}$.

**Step 3 – End-to-end delay bound computation**: Finally, knowing the initial arrival curve of the *foi* and the end-to-end service curve granted to the *foi* computed in Step 2, we derive the bound on the worst-case end-to-end delay, denoted $D_f$. Applying Theorem 6 of Appendix B.1, the delay bound in cycles is as follows:

$$D_f^{\mathbb{P}_f} = \left\lceil \frac{\sigma^{\mathbb{P}_f}[0]}{R_f} + T_{\mathbb{P}_f} + T_{DB} + T_{IB} \right\rceil \tag{4.2}$$

The reason we use the ceiling function is because a latency of half a clock cycle has no physical value.
In the next sections, we will detail each of the three steps of the analysis.

## 4.4 Indirect Blocking Analysis

In this section, we detail the first step of our approach BATA. To account for indirect blocking due to backpressure, we have to account for the effect of limited buffer size. Consider $k$ and $l$ two flows that are directly interfering with one another, $\mathbb{P}_k, \mathbb{P}_l$ their paths, and let $\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$ be the last node they share:

$$\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l) = \mathbb{P}_k[\max\{i, \mathbb{P}_k[i] \in \mathbb{P}_l\}]$$

Similarly, the first common node of $\mathbb{P}_k$ and $\mathbb{P}_l$ is called the convergence node of $k$ and $l$ and denoted $\mathrm{cv}(\mathbb{P}_k, \mathbb{P}_l)$.
Suppose the path of $l$ continues after $\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$. Even if the head flit of $l$ is not stored in a router of $\mathbb{P}_k \cap \mathbb{P}_l$, the limited buffer size available in each router can lead to storing the tail flit of $l$ in a router of $\mathbb{P}_k \cap \mathbb{P}_l$ under contention. In that case, $l$ blocks $k$.
Therefore, we need to quantify the way a packet of flow $f$ spreads into the network when it is blocked and stored in buffers. We denote $B$ the size of the buffer.

**Definition 6.** Spread Index
*Consider a flow $f$ of maximum packet length $L_f$ flits. The spread index of $f$, denoted $N_f$, is defined as follows:*

$$N_f = \left\lceil \frac{L_f}{B} \right\rceil$$

*where $B$ is the buffer size at node $r$ in flits.*

$N_f$ is the number of buffers needed to store one packet of flow $f$.
Using this notion and the last intuitive example, we call the section of the path of flow $k$ from $\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$ through $N_k$ nodes (at most) "subpath of $k$ relatively to $l$":

**Definition 7.** *The subpath of a flow $k$ relatively to a flow $l$ is:*

$$subpath(\mathbb{P}_k, \mathbb{P}_l) \quad = \quad \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l) + 1], \ldots, \mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l) + N_k]\Big]$$

*where* $Last(\mathbb{P}_k, \mathbb{P}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{P}_l\}$ *is the index of the last node shared by* $k$ *and* $l$ *along* $\mathbb{P}_k$, i.e. $\mathbb{P}_k[Last(k,l)] = \mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$.

We can extend this notion and define, in a similar fashion, the subpath of any flow $k$ relatively to a subpath $\mathbb{S}_l \subset \mathbb{P}_l$ of any flow $l$ (with $l \neq k$ or $l = k$). The previous notation still holds:

**Definition 8.** *The subpath of a flow $k$ relatively to any subpath $\mathbb{S}_l$ of any flow $l$ is:*

$$subpath(\mathbb{P}_k, \mathbb{S}_l) \quad = \quad \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + 1], \ldots, \mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + N_k]\Big]$$

*where* $Last(\mathbb{P}_k, \mathbb{S}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{S}_l\}$ *is the index along* $\mathbb{P}_k$ *of the last node shared by* $k$ *and* $l$ *within* $\mathbb{S}_l$. *By abuse of notation, we may denote* $subpath(k,l)$ *to refer to* $subpath(\mathbb{P}_k, \mathbb{P}_l)$, *and similarly* $subpath(k, \mathbb{S}_l)$ *to refer to* $subpath(\mathbb{P}_k, \mathbb{S}_l)$.

If $\mathbb{P}_l$ ends before reaching the $N_l$-th node after $\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$, then we ignore the out-of-range indexes. The notion of subpath is illustrated in Figure 4.5 for the *foi* $k$ and a spread index for the interfering flow $l$ equal to 3, *i.e.*, $N_l = 3$.

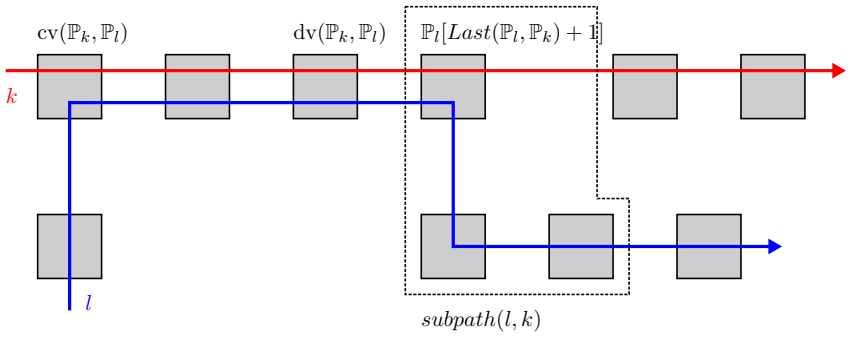



Figure 4.5 – Subpath illustration for the *foi* $k$

The main steps to determine the IB set for a flow $f$ are detailed in Algorithm 1 and are as follows:

1. with $sp(f)$ denoting the set of flows with same priority as $f$, determine all subpaths of flows in $DB_f^{|\mathbb{P}_f} \cap sp(f)$ relatively to flow $f$ (Line 2);
2. for each of these subpaths, check if they intersect with other flows. If they do, determine the subpaths of these other flows relatively to the subpath they intersect (Line 8);

3. add the new found subpaths to the set and reiterate until no new subpath is found (Line 11).

The function call computeDBset$(j, path)$ returns all the flows of $DB_j^{|path} \cap sp(j)$ and their associated subpath relatively to $j$.

**Algorithm 1** Determining $IB_f$ and the associated subpaths

indirectBlockingSet($f$, $\mathbb{P}_f$)

**Input:** $f$, the flow of interest, $\mathbb{P}_f$ the associated path

**Output:** $IB_f$, a set containing flow indexes and associated subpath involved in indirect interference on $\mathbb{P}_f$

1: $IB_f$ is initially empty
2: $init\_set \leftarrow DB_f^{|\mathbb{P}_f} \cap sp(f)$
*// initialize S*
3: **for** $i \in init\_set$ **do**
4: Append $\{i, i.\text{subpath}(f)\}$ to $S$
5: **end for**
6: **while** $S \neq \varnothing$ **do**
7: Pop a pair $\{j, \text{subj}\}$ from $S$
*// Compute subpaths relatively to j on subj :*
8: currentDB $\leftarrow$ computeDBset$(j, \text{subj})$
9: **for** $(k, \text{subk}) \in$ currentDB $\cap\, sp(j)$ **do**
10: **if** $(k, \text{subk}) \notin IB_f$ **then**
11: Append $(k, \text{subk})$ to $IB_f$ and $S$
12: **end if**
13: **end for**
14: **end while**
15: **return** $IB_f$

Notice that flows in $DB_f \cap sp(f)$ have no subpath in $IB_f$, since the influence of directly-interfering, same-priority flows is already integrated through the computation of the direct blocking latency $T_{DB}$, as explained in Section 4.5.1, Eq. 4.6b. Moreover, this algorithm can be used to compute the subset of $IB_f$ containing only flows that can cause indirect blocking on a subpath $subP \subset \mathbb{P}_f$. In that case, we need to consider $DB_f^{|subP}$ instead of $DB_f$, and we will note the result "partial indirect blocking set" $IB_f^{|subP}$.

**Property 1.** *(Complexity of IB set computation algorithm)*
*The complexity of Algorithm 1 for a flow set $\mathcal{F}$, denoted $\mathcal{C}(\mathcal{F})$, and expressed as the number of calls to* `computeDBset`*, is bounded according to the following expression:*

$$\mathcal{C}(\mathcal{F}) \leq 1 + |\mathcal{F}| \sum_{f \in \mathcal{F}} |\mathbb{P}_f|$$

*The maximal complexity of* `computeDBset()`*, denoted* $\mathcal{C}$(`computeDBset`)*, is:*

$$\mathcal{C}(\texttt{computeDBset}) = \mathcal{O}\left(\max_{f \in F} |\mathbb{P}_f|\right)$$

*Proof.* Assume that, at every call of the function `computeDBset()` on line 8, we store the result so that we don't have to compute the subpaths of all flows relatively to the same subpath twice. Then, the number of calls to the function `computeDBset()` is at most the number of possible subpaths in the whole flow set.
The subpath of a flow relatively to an arbitrary subpath is determined by only two factors: the divergence point (where the subpath starts) and the spread index (the length of the subpath). Thus, for any flow $f \in \mathcal{F}$, there are as many possible subpaths as there are nodes on the path of $f$, *i.e.* there are $|\mathbb{P}_f|$.
Therefore, the total number of possible subpaths in $\mathcal{F}$ is:

$$|\mathcal{F}| \sum_{f \in \mathcal{F}} |\mathbb{P}_f|$$

Finally, we call `computeDBset()` once before the `while` loop, hence the final result.

We now evaluate the complexity of `computeDBset()`. Applied to a flow $f$, this function computes the subpaths of the flows in $DB_f$ relatively to $f$. Assuming we have a preprocessed dictionary listing, for every node, the indexes of flows using this node,[2] we only have to run our algorithm through the path of $f$ and check if there are contending flows at this node. Comparing the indexes of the current node with those of the previous node, we can find divergence nodes of contending flows relatively to the flow of interest. We assume that, knowing the divergence point of a contending flow relatively to the flow of interest, it takes a constant time to find its subpath (we only need to compute the spread index). The complexity of `computeDBset()` called on a flow $f$ is thus proportional to the path length of $f$. Thus, we can bound its complexity as follows:

$$\mathcal{C}(\texttt{computeDBset}) = \mathcal{O}\left(\max_{f \in F} |\mathbb{P}_f|\right)$$

□

[2] We do run such a preprocessing on the configuration.

## 4.5 End-to-End Service Curve Computation

In this section, we detail the second step of BATA, consisting of the computation of the end-to-end service curve.

### 4.5.1 Direct Blocking Latency

The first component of the end-to-end service curve latency is the base latency.

**Theorem 1.** *The base latency for a flow $f$ on its path $\mathbb{P}_f$ in a NoC with strict service curve nodes of the rate-latency type $\beta_{R,T}$ is equal to:*

$$T_{\mathbb{P}_f} = \sum_{r \in \mathbb{P}_f} T^r \tag{4.3}$$

*where $T^r$ is the latency of the service curve for node $r$.*

*Proof.* The proof will be done with the proof of Theorem 2. □

To account for the flow serialization impact, we use results of [70], recalled in Appendix B.1 for FP policy. In that respect, we will need the following definitions:

**Definition 9.** *Let $f$ be the* foi.
*$hp(f)$ is the set of flows mapped to a VC of strict higher priority than $f$.*
*$sp(f)$ is the set of flows mapped to the same VC as $f$, $f$ excluded.*
*$lp(f)$ is the set of flows mapped to a VC of strict lower priority than $f$.*
*Moreover, we define $slp(f) = sp(f) \cup lp(f)$ (resp. $shp(f) = sp(f) \cup hp(f)$), that is all flows with a priority lower or equal (resp. higher or equal) than $f$, $f$ excluded.*

**Definition 10.** *(Extended convergence node) Consider $k, l$ two flows sharing resources, and their respective paths $\mathbb{P}_k, \mathbb{P}_l$. If we consider subsections of the paths of $k$ and $l$, denoted $\mathbb{S}_k$ and $\mathbb{S}_l$ respectively, the previous definition of the convergence node still holds. The first common node of $\mathbb{S}_k$ and $\mathbb{S}_l$ is denoted* $\mathrm{cv}(\mathbb{S}_k, \mathbb{S}_l)$. *By abuse of notation, we may denote* $\mathrm{cv}(k, l)$ *to refer to* $\mathrm{cv}(\mathbb{P}_k, \mathbb{P}_l)$ *and simplify the expressions.*

The maximum direct blocking latency, part of the maximum service latency defined in Eq. (4.1), is defined in the following Theorem.

**Theorem 2.** *(Maximum Direct Blocking Latency)*
*The maximum direct blocking latency for a* foi *$f$ along its path $\mathbb{P}_f$, in a NoC under*

*flit-level preemptive FP multiplexing with strict service curve nodes of the rate-latency type $\beta_{R,T}$ and leaky bucket constrained arrival curves $\alpha_{\sigma,\rho}$ is equal to:*

$$T_{hp} + T_{sp} + T_{lp}$$

*with:*

$$T_{hp} = \sum_{i \in DB_f \cap hp(f)} \frac{\sigma_i^{\mathrm{cv}(i,f)} + \rho_i \cdot \sum\limits_{r \in \mathbb{P}_f \cap \mathbb{P}_i} \left( T^r + \frac{L^r_{slp(f)}}{R^r} \right)}{R_f} \tag{4.4a}$$

$$T_{sp} = \sum_{i \in DB_f \cap sp(f)} \frac{\sigma_i^{\mathrm{cv}(i,f)} + \rho_i \cdot \sum\limits_{r \in \mathbb{P}_f \cap \mathbb{P}_i} \left( T^r + \frac{L^r_{slp(f)}}{R^r} \right)}{R_f} \tag{4.4b}$$

$$T_{lp} = \sum_{r \in \mathbb{P}_f} \frac{L^r_{slp(f)}}{R^r} \tag{4.4c}$$

*where:*

$$L^r_{slp(f)} = \max \left( \max_{j \in sp(f)} \left( L_j \cdot \mathbf{1}_{\{sp(f) \supset r\}} \right) , S_{flit} \cdot \mathbf{1}_{\{lp(f) \supset r\}} \right)$$

$$R_f = \min_{r \in \mathbb{P}_f} \left\{ R^r - \sum_{j \ni r, j \in shp(f)} \rho_j \right\}$$

*Proof.* The main idea is to integrate the impact of the flow serialization phenomena on the granted end-to-end service curve for the *foi* $f$ along its path $\mathbb{P}_f$. To achieve this aim, we adapt the results of the existing Theorem 7, Appendix B.1, based on the PMOO principle, to take into account the specificities of wormhole NoCs, in comparison to classic switched networks.

The wormhole NoCs allow the flit-level preemption during transmission, which modifies the lower priorities impact on the *foi* in comparison to the non-preemptive mode in classic switched networks. Hence, a lower priority flow that is being transmitted at any node can delay the *foi* $f$ by at most the maximum transmission time of one flit. Consequently, the term $\max_{i \ni k, i \in slp(f)} L_i$ in Eq. (B.1) must be modified for each node on $\mathbb{P}_f$ as follows:

- if there is one or more same-VC contending flow(s), this term becomes the maximum packet size of the contending, same priority flow(s) ;
- if there is one or more lower-VC flow(s), it equals the size of one flit $S_{flit}$ ;
- if there is no same or lower-VC flow, it equals zero.

Therefore, the flit-level preemption property of NoCs infers that a *foi* $f$ will suffer from lower priority flows within any crossed node $r \in \mathbb{P}_f$ during the maximum transmission time of $L^r_{slp(f)}$, which is defined as follows:

$$L^r_{slp(f)} = \max\left(\max_{j \in sp(f)}\left(L_j \cdot \mathbf{1}_{\{sp(f) \supset r\}}\right),\ S_{flit} \cdot \mathbf{1}_{\{lp(f) \supset r\}}\right)$$

Afterwards, we apply Theorem 7, Appendix B.1, while taking into account such modification (the impact of lower priority flows due to flit-level preemption). In doing this, we obtain the end-to-end service curve of flow $f$ along its path $\mathbb{P}_f$, which integrates only the impact of (i) the base latency (or technological latency) and (ii) the direct blocking set of $f$, $DB_f$, as follows:

$$R_f = \min_{r \in \mathbb{P}_f}\left\{R^r - \sum_{j \ni r, j \in shp(f)} \rho_j\right\} \tag{4.6a}$$

$$T_f = \sum_{r \in \mathbb{P}_f}\left(T^r + \frac{L^r_{slp(f)}}{R^r}\right) + \sum_{i \in DB_f \cap shp(f)} \frac{\sigma_i^{\mathrm{cv}(i,f)} + \rho_i \cdot \sum\limits_{r \in \mathbb{P}_f \cap \mathbb{P}_i}\left(T^r + \frac{L^r_{slp(f)}}{R^r}\right)}{R_f} \tag{4.6b}$$

We can split the first sum in the expression of $T_f$ (Eq. 4.6b) into the following contributions:

$$\begin{aligned} T_{\mathbb{P}_f} &= \sum_{r \in \mathbb{P}_f} T^r \\ T_{lp} &= \sum_{r \in \mathbb{P}_f} \frac{L^r_{slp(f)}}{R^r} \end{aligned}$$

From this point on, the computation of the different parts of the direct blocking latency defined in Theorem 2 is straightforward. □

### 4.5.2 Indirect Blocking Latency

Knowing the IB set, the Indirect Blocking Latency $T_{IB}$ is computed using the following Theorem:

**Theorem 3.** *(Maximum Indirect Blocking Latency)*
*The maximum indirect blocking latency for a* foi *$f$ along its path $\mathbb{P}_f$, in a NoC under flit-level preemptive FP multiplexing with strict service curve nodes of the*

*rate-latency type $\beta_{R,T}$ and leaky bucket constrained arrival curves $\alpha_{\sigma,\rho}$, is as follows:*

$$T_{IB} = \sum_{(k,subP)\in IB_f} \frac{\sigma_k^{subP[0]}}{\widetilde{R}_k^{subP}} + \widetilde{T}_k^{subP} \tag{4.7}$$

*where:*

$$\widetilde{R}_k^{subP} = \min_{r\in subP} \left\{ R^r - \sum_{j\ni r, j\in hp(f)} \rho_j \right\} \tag{4.8a}$$

$$\widetilde{T}_k^{subP} = \sum_{r\in subP} \left( T^r + \frac{S_{flit}\mathbf{1}_{\{lp(k)\supset r\}}}{R^r} \right) + \sum_{i\in DB_k^{|subP}\cap hp(k)} \frac{\sigma_i^{\mathrm{cv}(i,k)} + \rho_i \sum_{r\in subP\cap\mathbb{P}_i} \left( T^r + \frac{S_{flit}\mathbf{1}_{\{lp(k)\supset r\}}}{R^r} \right)}{\widetilde{R}_k^{subP}} \tag{4.8b}$$

*Proof.* The maximum indirect blocking latency for a *foi* $f$ is induced by the flows set $IB_f$, computed in the previous section using Algorithm 1. Any flow $j \in IB_f$ will impact the *foi* $f$ during the maximum time it occupies the associated subpath $subP_j$, $\Delta t_j^{max}$. Hence, a safe upper bound on the indirect blocking latency is as follows:

$$T_{IB} \leq \sum_{j\in IB_f} \Delta t_j^{max}$$

On the other hand, for any flow $j \in IB_f$, $\Delta t_j^{max}$ is upper bounded by the end-to-end delay bound of flow $j$ along its associated subpath $subP_j$, $D_j^{subP_j}$, which infers the following:

$$T_{IB} \leq \sum_{(j,subP_j)\in IB_f} D_j^{subP_j} \tag{4.9}$$

Based on Theorem 6, Appendix B.1, the delay bound of flow $j$, $D_j^{subP_j}$, is computed as the maximum horizontal distance between:

- the maximum arrival curve of flow $j$ at the input of the subpath $subP_j$, $\alpha_j^{subP_j[0]}$, which takes into account the impact of all the interferences suffered by flow $j$ upstream the node $subP_j[0]$, *i.e.*, the propagated arrival curve of flow $j$ until the input of $subP_j[0]$ using Theorem 6;
- the granted service curve to flow $j$ by its VC along $subP_j$, $\widetilde{\beta}_j^{subP_j}$, called VC-service curve, when ignoring the same-priority flows (which are already included in $IB_f$). The latter condition is due to the pipelined behavior of the network, where the same-priority flows sharing $subP_j$ are served one after another if they need shared resources. Hence, the impact of flows with the

same priority as flow $j$ is already integrated within the sum expressed in Eq. (4.9).

To compute the granted service curve $\widetilde{\beta}_j^{subP_j}$ for each flow $j \in IB_f$ along $subP_j$, we follow similar approach than in the proof of Theorem 3 through applying the existing Theorem 7, Appendix B.1, when:

- ignoring the same-priority flows in $sp(j)$, thus all $shp(j)$ will become $hp(j)$ and $slp(j)$ will become $lp(j)$ in Eqs. (B.1a) and (B.1b);
- considering the flit-level preemption, thus the impact of lower-priority flows in Eq. (B.1a) is bounded by the maximum transmission time of $S_{flit} \cdot \mathbf{1}_{\{lp(k) \supset r\}}$ within each crossed node $r \in subP_j$;
- considering only the direct blocking flows of $j$ intersecting $\mathbb{P}_j$ on $subP_j$, thus considering $DB_j^{|subP_j} \cap hp(j)$ in Eq. (B.1b).

Hence, we obtain $\widetilde{R}_j^{subP_j}$ and $\widetilde{T}_j^{subP_j}$ described in Eqs. (4.8a) and (4.8b), respectively. Consequently, the maximum indirect blocking latency in Eq. (4.9) can be re-written as follows:

$$T_{IB} \leq \sum_{(j,subP_j) \in IB_f} \frac{\sigma_j^{subP_j[0]}}{\widetilde{R}_j^{subP_j}} + \widetilde{T}_j^{subP_j} \tag{4.10}$$

□

### 4.5.3 Computation Algorithm

The computation of the end-to-end service curve is recursive. To understand this, we present Algorithm 2 that details the computation.

There are two cases in which the function calls itself:

1. when computing $T_{DB}$, we need to know the burst of the contending flow at the convergence point with the *foi*. Thus, we compute the service curve for the contending flow from its source to the convergence point with the *foi* (Algorithm 2, line 6);
2. when computing $T_{IB}$, we need to know the arrival curve of each flow in the IB set at the beginning of its subpath. Thus, we compute the service curve of this flow from its source to the beginning of the appropriate subpath (Algorithm 2, Line 15).

Because of these two recursive calls and the fact that they depend a lot on the flow pattern configuration, it may be hard to analytically bound the complexity of the algorithm. Therefore, we will estimate it during the computational analysis of Section 4.7.3. However, we can conjecture that these two recursive calls may be the main contribution to the algorithm computational complexity.

**Algorithm 2** Computing the end-to-end service curve for a flow $f$

endToEndServiceCurve$(f, \mathbb{P}_f)$

**Input:** $f$, the flow of interest, $\mathbb{P}_f$ the associated path
**Output:** $\beta_f(t)$, the end-to-end service curve granted to flow $f$ on $\mathbb{P}_f$

Compute $R_f$
Compute $T_{\mathbb{P}_f}$
*// Compute $T_{DB}$:*
$T_{DB} \leftarrow 0$
**for** $k \in DB_f$ **do**
  $r_0 \leftarrow \text{cv}(k, f)$ *// Get convergence point of $f$ and $k$*
  $\beta_k \leftarrow \text{endToEndServiceCurve}(k, [\mathbb{P}_k[0], \cdots, r_0])$
  $\alpha_k^0 \leftarrow$ initial arrival curve of $k$
  $\alpha_k^{r_0} \leftarrow \text{computeArrivalCurve}(\alpha_k^0, \beta_k)$
  $T_{DB} \leftarrow \text{directBlocking}(\alpha_k^{r_0})$
**end for**
*// Compute $T_{IB}$:*
$IB_f \leftarrow \text{indirectBlockingSet}(f)$
$T_{IB} \leftarrow 0$
**for** $\{k, S\} \in IB_f$ **do**
  $\widetilde{\beta}_k \leftarrow$ VC-service curve of $k$ on $S$
  *// Compute the service curve of $k$ from its first node to the beginning of $S$:*
  $\beta_k \leftarrow \text{endToEndServiceCurve}(k, [\mathbb{P}_k[0], \cdots, S[0]])$
  $\alpha_k^{S[0]} \leftarrow \text{computeArrivalCurve}(\alpha_k^0, \beta_k)$
  *// Now add the delay over the subpath to $T_{IB}$ :*
  $T_{IB} \leftarrow T_{IB} + \text{delayBound}(\alpha_k, \widetilde{\beta}_k^{S[0]})$
**end for**
**return** $R_f(t - (T_{\mathbb{P}_f} + T_{DB} + T_{IB}))^+$

## 4.6 Illustrative Example

We propose an example to illustrate the method for computing the worst-case delay bound of a flow. We consider the configuration of Figure 4.3 and assume that:

- all routers have a service curve $\beta(t) = R(t - T)^+$;
- flow $i$ has a packet length $L_i = L$ and the initial arrival curve $\alpha_i(t) = \sigma + \rho t$;
- all flows are mapped to the same VC;
- all packets are 3 flit-long and all buffers hold one flit, so that the spread index for any flow is 3;
- The flow of interest is flow 1.

**Step 1: IB analysis**

We start with flow 1 and notice it is directly blocked by flow 2. The divergence point of these two paths is R3 (last shared output). Thus, we compute the subpath

of flow 2 relatively to flow 1 and initialize $S$ as follows:

$$S = \{\{2, [\text{R4}, \text{R5}, \text{R6}]\}\}$$

We then enter the while loop:

1. we pop the pair {2, [R4,R5,R6]} from $S$. The subpath intersects the path of flow 3 and their divergence point is R6.
2. we compute the subpath of flow 3 relatively to [R4,R5,R6], and add the corresponding pair to $S$ and $IB_1$. Now, we have:

$$S = \{\{3, [\text{R7}, \text{R8}, \text{R9}]\}\}$$

3. we pop {3, [R7,R8,R9]} from $S$. It does not intersect any path that we haven't crossed yet, so there is nothing to be done. Now $S = \varnothing$ and we exit the loop.

Finally, we have:

$$IB_1 = \{\{3, [\text{R7}, \text{R8}, \text{R9}]\}\}$$

**Step 2: End-to-end service curve computation**

**DB latency computation:**

All flows are mapped to the same VC, thus $T_{hp} = T_{lp} = 0$. We then have:

$$\begin{aligned} T_{\mathbb{P}_1} &= 4T \\ T_{sp} &= \frac{\sigma_2^{R3} + \rho(\cdot T + \frac{L_i}{R})}{R - \rho} \\ &= \frac{\sigma + \rho \cdot (T + \frac{L}{R})}{R - \rho} \end{aligned}$$

Hence :

$$\begin{aligned} T_{DB} &= 4T + \frac{\sigma}{R - \rho} + \rho \frac{T + \frac{L}{R}}{R - \rho} \\ &= 7,368421053 \text{ cycles} \end{aligned}$$

with $R = 1$ flit/cycle, $T = 1$ cycle, $\rho = 0.05$ flits/cycle, $\sigma = 3$ flits and $L = 3$ flits.

**IB latency computation:**

To compute the indirect blocking latency for flow 1 on the configuration of Figure 4.3, we need to compute the arrival curve of flow 3 at R7 (more precisely we are interested in the burst of this flow at the input of R7). This leads to recursive calls.

The service curve granted to flow 1 on [R1, R2] is:

$$\beta_1^{[R1,R2]} = R(t-2T)^+$$

The burst of flow 1 at R3 is thus

$$\sigma_1^{R3} = \sigma + 2\rho T$$

The service curve granted to flow 2 on [R3, R4, R5] is:

$$\beta_2^{[R3,R4,R5]} = (R-\rho)\left(t - 3T - \frac{\sigma_1^{R3} + \rho(T+\frac{L}{R})}{R-\rho}\right)^+$$

The burst of flow 2 at R6 is thus:

$$\sigma_2^{R6} = \sigma + \rho\left(3T + \frac{\sigma_1^{R3} + \rho(T+\frac{L}{R})}{R-\rho}\right)$$

The service curve granted to flow 3 on [R6] is:

$$\beta_3^{[R6]} = (R-\rho)\left(t - T - \frac{\sigma_2^{R6} + \rho(T+\frac{L}{R})}{R-\rho}\right)^+$$

Finally, the burst of flow 3 at R7 is:

$$\sigma_3^{R7} = \sigma + \rho\left(T + \frac{\sigma_2^{R6} + \rho(T+\frac{L}{R})}{R-\rho}\right)$$

Numerically, with $R = 1$ flit/cycle, $T = 1$ cycle, $\rho = 0.05$ flits/cycle, $\sigma = 3$ flits and $L = 3$ flits, we have :

$$\begin{aligned}
\sigma_1^{R3} &= 3 + 2 \times 0.05 \times 1 \\
&= 3.1 \\
\sigma_2^{R6} &= 3 + 0.05 \times \left(3 \times 1 + \frac{3.1 + 0.05(1+3/1)}{0.95}\right) \\
&= 3 + 0.05 \times (3 + 3.3/0.95) \\
&= 3.323684211 \\
\sigma_3^{R7} &= 3 + 0.05 \times \left(1 + \frac{3.323684211 + 0.05(1+3/1)}{0.95}\right) \\
&= 3 + 0.05 \times (1 + 3.709141275) \\
&= 3.235457064
\end{aligned}$$

Then we compute $\widetilde{T_3}^{[R7,R8,R9]}$. Since there are no higher or lower priority flows on the configuration, we have :

$$\begin{aligned}\widetilde{T_3}^{[R7,R8,R9]} &= 3T \\ \widetilde{R_3}^{[R7,R8,R9]} &= R\end{aligned}$$

Finally:

$$\begin{aligned}T_{IB} &= 3T + \sigma_3^{R7}/R \\ &= 6.235457064 \text{ cycles}\end{aligned}$$

We had to perform 3 additional calls to the function computing the end-to-end service curve.

**Step 3: End-to-end delay bound:**

Once we computed all latencies, we can infer the end-to-end delay bound using Equation A.1:

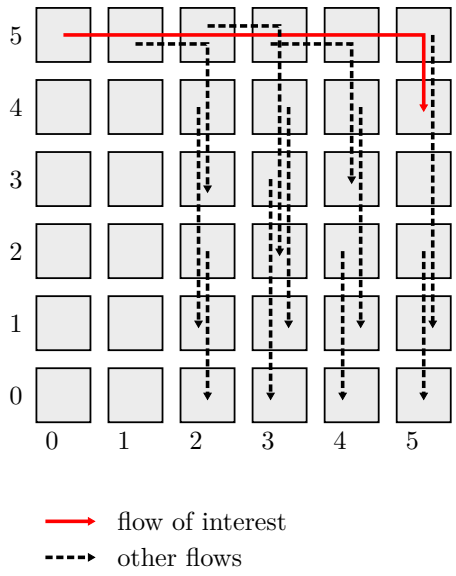
$$\begin{aligned}D_1^{\mathbb{P}_1} &= i\frac{\sigma_1}{R_1} + T_{\mathbb{P}_1} + T_{DB} + T_{IB} \\ &= \left\lceil \frac{\sigma}{R-\rho} + 4T + \frac{\sigma}{R-\rho} + \rho\frac{T+\frac{L}{R}}{R-\rho} + 3T + \frac{\sigma_3^{R7}}{R} \right\rceil \\ &= 17 \text{ cycles}\end{aligned}$$

The end-to-end delay bound for flow 1 is thus 17 cycles.

## 4.7 Performance Evaluation

In this section, we conduct a performance evaluation of our model. First, we conduct a sensitivity analysis of BATA (Section 4.7.1) to identify the configuration parameters that have the highest impact on the delay bounds. In Section 4.7.2, we assess the tightness of the delay bounds yielded by BATA, for different values of the identified parameters. Then, we evaluate computational aspects of the approach in Section 4.7.3. Finally, we provide a comparative analysis with a CPA-based state-of-the-art approach in Section 4.7.4.

### 4.7.1 Sensitivity Analysis




| Flow | Source | Destination |
|---|---|---|
| 1 | (0, 5) | (5, 4) |
| 2 | (1, 5) | (2, 3) |
| 3 | (2, 5) | (3, 2) |
| 4 | (3, 5) | (4, 3) |
| 5 | (5, 5) | (5, 1) |
| 6 | (2, 4) | (2, 1) |
| 7 | (2, 2) | (2, 0) |
| 8 | (3, 4) | (3, 1) |
| 9 | (3, 3) | (3, 0) |
| 10 | (4, 4) | (4, 1) |
| 11 | (4, 2) | (4, 0) |
| 12 | (5, 2) | (5, 0) |

Figure 4.6 – Flow configuration on a 6×6 mesh NoC

For the sensitivity analysis, we will analyze the end-to-end delay bounds when varying the following parameters:

- buffer size for values 1, 2, 3, 4, 6, 8, 12, 16, 32, 48, 64 flits;
- total packet length (including header) for values 2, 4, 8, 16, 64, 96, 128 flits;
- flow rate for values between 1% and 40% of the total link capacity (so that the total utilization rate on any link remains below 100%).

To achieve this aim, we consider the configuration described on Figure 4.6. This configuration remains quite simple but exhibits sophisticated indirect blocking patterns. We assume flows are periodic with no jitter, and have the same period and packet length. We also assume each router can handle one flit per cycle and it takes one cycle for one flit to be forwarded from the input of a router to the input of the next router, *i.e.*, for any node $r$, $T^r = 1$ cycle and $R^r = 1$ flit/cycle. Finally, to maximize indirect blocking, we consider that all the flows are mapped on the same VC. Our flow of interest is flow 1, because it is one of the flows that is most likely to undergo indirect blocking.

Figure 4.7 illustrates the end-to-end delay bounds of the *foi* when varying buffer size. For the left graph, we keep each flow rate constant at 4% of the total bandwidth; whereas for the right graph, we keep each flow packet length at 16 flits.

We notice on both graphs that end-to-end delay bounds decrease with buffer size, with occasional stalling from one value to another. This is something we could

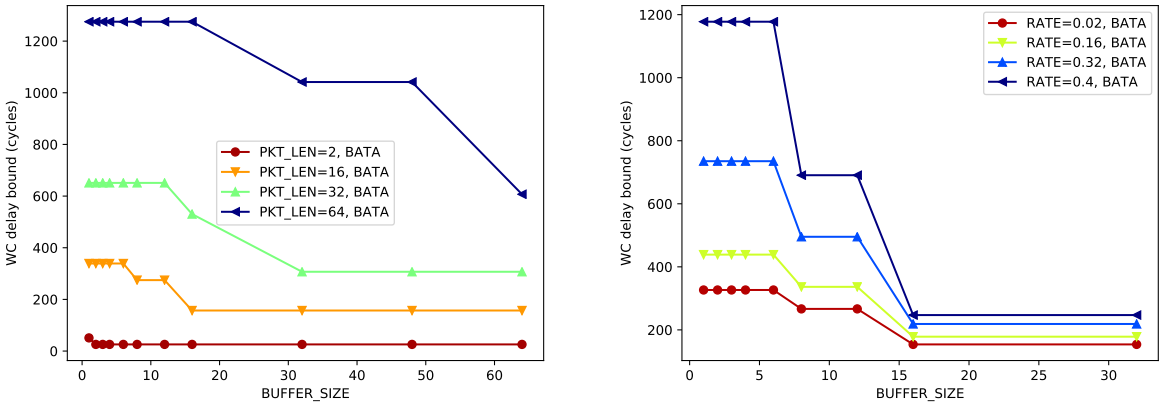



Figure 4.7 – Buffer size impact on BATA end-to-end delay bounds

expect because at a given packet length, the greater the buffer, the less a packet can spread in the network. Consequently, the IB set tends to be smaller, or to contain smaller subpaths. We notice that past a certain buffer size, the end-to-end delay bounds stay constant with larger buffers. This results from the fact that the IB set remains the same once buffers are big enough to hold an entire packet. Therefore, adding buffer space after a certain point does not improve end-to-end delay bounds. Hence, over-dimensioning the buffers within routers is not efficient to enhance NoC performance.

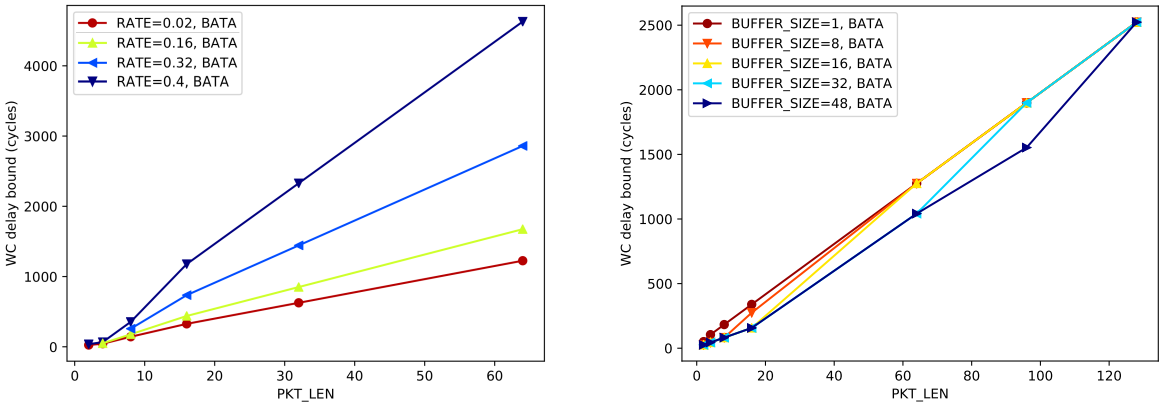



Figure 4.8 – Packet length impact on BATA end-to-end delay bounds

Next, we focus on the packet length impact on the end-to-end delay bound, as illustrated in Figure 4.8. The left graph presents results when the buffer size is

constant (4 flits) and the right one when the rate of each flow is constant (4% of the link capacity).

The first observation we can make from both graphs is that the delay bounds evolve in an almost linear manner with the packet length. For instance, on the left graph, with 8 flits of buffer size and packet length equal to 16, 64, 96 and 128 flits, the ratio of packet length and end-to-end delay bound is 17.2, 19.9, 19.8, 19.7.

On the right graph, we observe further interesting aspects of sensitivity at a relatively low rate (4%):

- For a given packet length, the buffer size has a limited impact on the end-to-end delay bounds. For instance, for packet length 64 flits, the delay bounds decrease by less than 25% when the buffer size increases by 480%;
- For packet lengths that are significantly larger than buffer size, the delay bound remains constant regardless of the buffer size, *e.g.*, it is the case for packet length 128 flits.

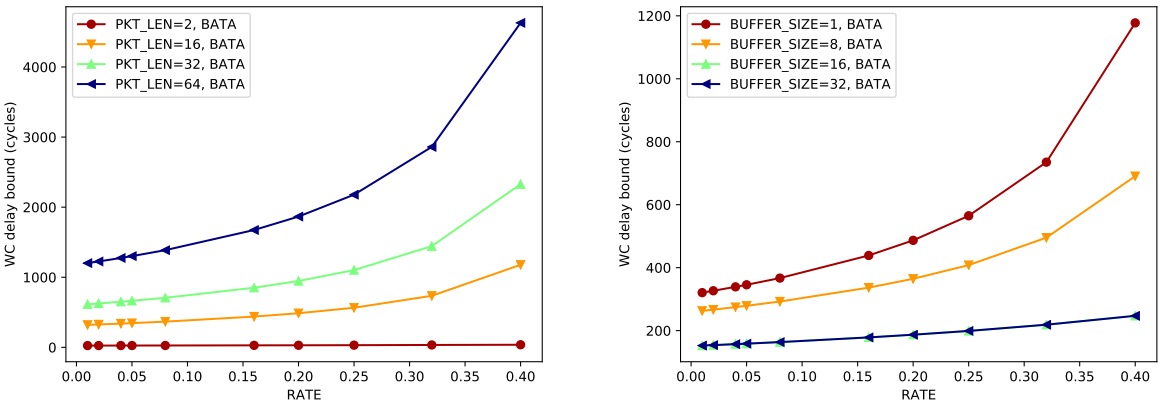



Figure 4.9 – Rate impact on BATA end-to-end delay bounds

Finally, we study the impact of the flow rate on the end-to-end delay bounds, as illustrated on Figure 4.9. The buffer size is fixed to 4 flits on the left graph and the packet length to 16 flits on the right one. Both graphs show that the delay bounds increase with the rate. Based on the left graph, one can notice that at a constant rate, increasing the packet length usually causes the end-to-end delay bound increase. The right graph confirms the conclusion drawn from Figure 4.7: increasing buffer size does not improve delay bounds after a certain value. Moreover, for small buffer sizes, delay bounds seem more sensitive to the rate variation. For instance, for buffer size equal to 1 flit, delay bounds are 321 cycles and 1178 cycles (×3.7) for rates equal to 1% and 40% (×40), respectively. Whereas, for buffer size

equal to 32 flits, the delay bounds are multiplied by only 1.6 when considering the same rate values.
The conducted sensitivity analysis reveals two main interesting conclusions:

- The configuration parameters having the highest impact on the derived delay bounds are the buffer size and the flow rate; Thus, both parameters will be considered for the tightness analysis;
- Increasing the buffer size within routers after a certain point does not improve the NoC performance; Thus, over-dimensioning the buffers is not considered as an efficient solution to decrease the delay bounds.

### 4.7.2 Tightness Analysis

To assess the tightness of the delay bounds yielded by BATA, we consider herein a worst-case simulation using Noxim simulator engine [50]. Knowing no method to compute the exact worst-case for wormhole NoCs, we derive an achievable worst-case delay through simulation, that we compare to the analytical end-to-end delay bounds.
In order to approach the worst-case scenario, we run each flow configuration many times while varying the flows offsets and we consider the maximum worst-case delay over all the simulated configurations. Afterwards, we compute the "tightness ratio" for each flow $f$, denoted $\tau_f$, that is the ratio of the achievable worst-case delay $D_{\text{WC}}$ and the worst-case delay bound $D_f$:

$$\tau_f = \frac{D_{\text{WC}}}{D_f}$$

A tightness ratio of 100% means the worst-case delay bound is the exact worst-case delay. However, it is worth noticing that a tightness ratio below 100% does not necessarily mean that the worst-case delay bound is inaccurate, but it can simply reveal that the worst-case scenario has not been reached by the simulation. Therefore, the determined tightness ratio is a lower bound on the exact tightness ratio.
To perform worst-case simulations, we have configured Noxim simulator engine [50] to control the traffic pattern using the provided traffic pattern file option. For each flow, we have specified:

- the source and destination cores;
- *pir*, packet injection rate, *i.e.* the rate at which packets are sent when the flow is active;

| Rate | 8% | | | 32% | | |
|---|---|---|---|---|---|---|
| Buffer | 4 | 8 | 16 | 4 | 8 | 16 |
| Tightness Statistics | | | | | | |
| **Average** | **70.1%** | **72.1%** | **80.8%** | **49.7%** | **64.2%** | **79.8%** |
| **Max** | **91.7%** | **92.0%** | **88.3%** | **95.6%** | **88.9%** | **97.3%** |
| **Min** | **40.6%** | **38.1%** | **48.9%** | **20.8%** | **33.3%** | **43.8%** |
| Per flow tightness ratios | | | | | | |
| 1 | 44% | 46% | 79% | 44% | 66% | 87% |
| 2 | 41% | 38% | 79% | 24% | 47% | 97% |
| 3 | 64% | 69% | 85% | 51% | 59% | 54% |
| 4 | 68% | 71% | 86% | 64% | 81% | 87% |
| 5 | 77% | 80% | 85% | 46% | 53% | 64% |
| 6 | 75% | 79% | 86% | 21% | 46% | 86% |
| 7 | 89% | 90% | 87% | 44% | 69% | 97% |
| 8 | 68% | 70% | 70% | 46% | 65% | 71% |
| 9 | 47% | 49% | 49% | 24% | 33% | 44% |
| 10 | 88% | 90% | 88% | 70% | 88% | 88% |
| 11 | 92% | 92% | 88% | 96% | 79% | 96% |
| 12 | 89% | 90% | 87% | 66% | 85% | 88% |

Table 4.1 – Tightness ratio results for the tested configuration

- *por*, probability of retransmission, *i.e.* the probability one packet will be retransmitted (in our context, this parameter is always 0);
- $t_{\text{on}}$, the time the flow wakes up, *i.e.* starts transmitting packets with the packet injection rate;
- $t_{\text{off}}$, the time the flow goes to sleep, *i.e.* stops transmitting;
- $P$, the period of the flow.

Moreover, since we want to simulate a deterministic flow behavior to approach the worst-case scenario, we use the following parameters for each flow:

- Maximal packet injection rate : 1.0;
- Minimal probability of retransmission : 0.0;

To create different contention scenarios and try approaching the worst-case of end-to-end delays, we randomly chose the offset of each flow and perform simulations with uniformly distributed values of offsets for each flow. We generate 40000 different traffic configurations for each set of parameters and simulate each of them for an amount of time that allows at least 5 packets to be transmitted.

We simulate the configuration of Figure 4.6, when varying buffer sizes to 4, 8 and 16 flits, and flow rates to 8% and 32% of the total available bandwidth. We extract the worst-case end-to-end delay found by the simulator and compute the tightness

ratio for each flow. The obtained results are gathered in Table 4.1. As we can see, tightness ratio is up to 80%. We also notice the average tightness ratio improves when the buffer size increases. For 8% rate, the average tightness ratio varies between 70.1% and 80.8%. For 32% rate, the average tightness ratio varies between 49.7% and 79.8%.

According to our sensitivity analysis, the indirect blocking patterns covered by our model tend to become simpler when the buffer size increases, making the IB latency smaller. Moreover, we can expect a correlation between the tightness ratio and the IB set size:

- first, for each {flow index, subpath} pair in the IB set, the analysis may introduce a slight pessimism in the IB latency computation;
- second, the more complex the potential blocking scenarios are, the harder it is to reach or approach the worst-case delay by simulations: it requires a precise synchronization between flows to achieve those scenarios. Moreover, the greater the IB set, the less such a synchronization statistically happens over random offsets.

Therefore, we can infer that the greater the buffer size, the easier it is to approach the worst-case delay by simulating the configuration. This is confirmed by the general trend of the average tightness ratio. It is also backed up by the following fact: in our analysis, flow 1 and 2 have the largest IB sets and are the most likely to undergo indirect blocking. We notice that at 8% rate (resp. 32% rate), their delay bounds tightness rises from 44% to 79% (resp. 44% to 87%) and 41% to 79% (resp. 24% to 97%) when the buffer size increases from 4 flits to 16 flits.

### 4.7.3 Computational Analysis

We now assess how well BATA scales on larger configurations by evaluating the computation time. To achieve this aim, we consider a NoC larger than the one considered for tightness analysis, while varying the number of flows. We particularly consider a $8 \times 8$ NoC with 4, 8, 16, 32, 48, 64, 80, 96 and 128 flows and generate 20 configurations for each fixed number of flows $N$. To do so, we randomly pick $2N$ ($x$-coordinate, $y$-coordinate)-couples, where each coordinate is uniformly chosen in the specified range (here, from 0 to 7). We use $N$ of these couples for source cores and the other $N$ for destination cores. All other parameters (flow rate, packet length, buffer size, router latencies) are kept constant.

For each considered configuration, we focus on the following metrics:

- $\Delta t$, the total analysis runtime (computation time);

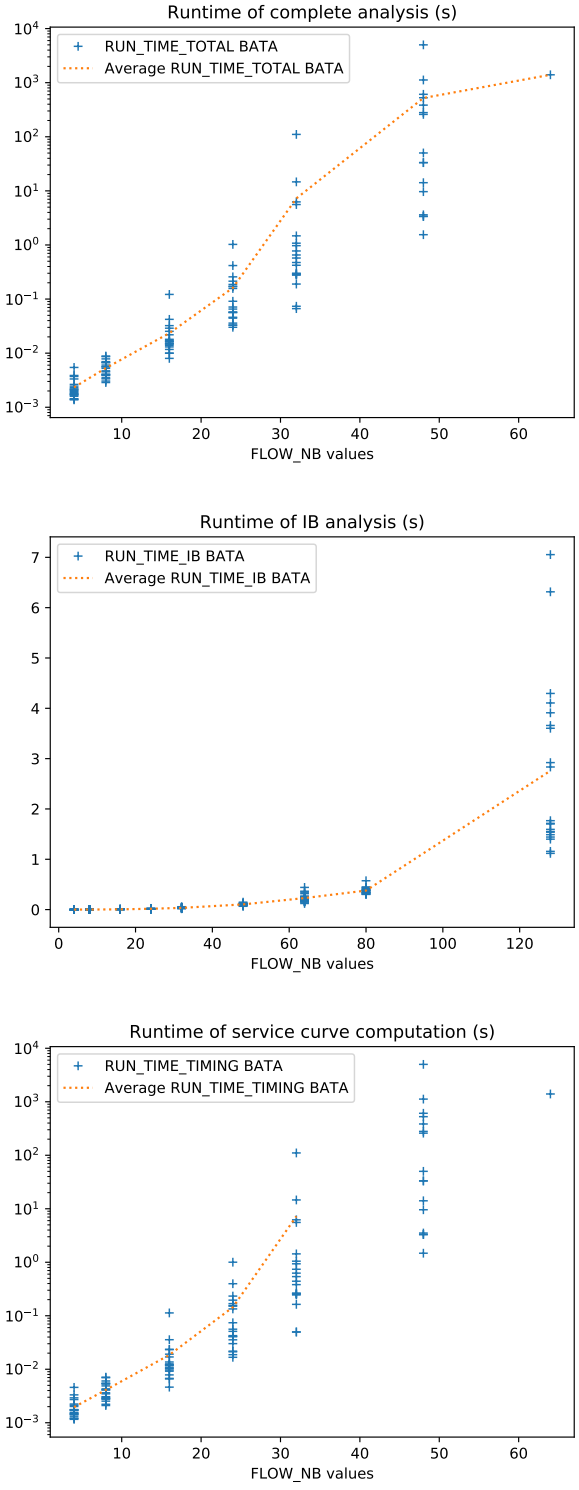



Figure 4.10 – Results of the BATA computational analysis

- $\Delta t_{IB}$, the duration of the IB set analysis;
- $\Delta t_{e2e}$, the duration of all end-to-end service curves computations.

The derived results are illustrated in Figure 4.10: the top graph for $\Delta t$, the middle one for $\Delta t_{IB}$ and the bottom one for $\Delta t_{e2e}$. For each flow number, we have plotted the average runtime for all the configurations with this number of flows, as well as the computed metric for each configuration (one dot per configuration). Only configurations with runtime up to $10^4$ s have been considered.

The left graph shows that the runtime grows rapidly with the number of flows (we're using a logarithmic scale on the Y-axis). Moreover, we notice that runtimes may vary a lot for the same number of flows. For instance, for 32 flows, they range between 67ms and 110s. For 48 flows, they go from 1.5s to more than 1h10min.

To further assess what impacts the complexity of BATA, we plot the contributions to the total runtime of the IB set analysis and the end-to-end service curve computation. We notice that the IB set analysis alone (middle graph) runs in less than 8 seconds for all tested configurations. This shows that BATA approach complexity is mostly due to the end-to-end service curve computation as shown in the right graph. This fact is mainly due to the recursive call to end-to-end service curve function in Algorithm 2, as discussed in Section 4.5.3.

To highlight this aspect, we measured the number of calls to the function computing a service curve during the analysis. The results displayed on Figure 4.11 clearly show the expected trend.

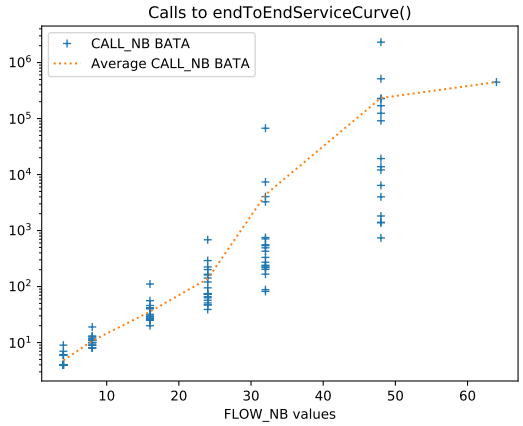



Figure 4.11 – Number of calls to the function endToEndServiceCurve()

We can also notice that when all flows are mapped to a different VC, no indirect blocking is possible, *i.e.* $IB_f = \varnothing$ for any flow $f$. Hence, there is no computation to be done for $T_{IB}$, which drastically reduces the complexity of the approach in this

case. In Algorithm 2, this means that lines 12 to 18 are not executed.
The derived results show that BATA gives accurate delay bounds for medium-scale configurations in less than one hour. However, the complexity of BATA increases with the number of flows due to the recursive calls to end-to-end service curve function. This fact is inherent to the large panel of NoCs, *i.e.*, priority-sharing, VC-sharing and buffer backpressure, covered by BATA.

### 4.7.4 Comparative Study

We performed a comparison based on the configuration presented on Figure 4.12 taken from [1] using CPA approach, where all flows have a packet length of 4 flits. We have reproduced different scenarios of [1] to compute the delay bounds with our proposal with respect to the flow rate and buffer size.

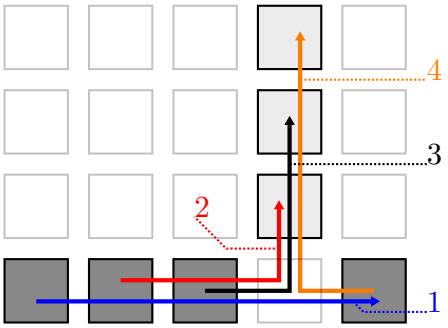



Figure 4.12 – A simple configuration from [1].

First, we vary the requested bandwidth per sender (*i.e.* the rate of each flow relatively to the maximal rate). Since the packet length is constant, we adjust the flow period to get different values of rate. The full bandwidth corresponds to a rate of one flit per cycle.
For each bandwidth value, we compute the corresponding delay bound predicted by our model for all 4 flows for a buffer size equal to 4 flits and the derived results are in Figure 4.13. For each flow, we also plotted a vertical line representing the saturation point of [1] CPA model : when the CPA-predicted latency is greater than $10^3$ cycles[3], we consider that the model diverges.
We first notice that the curves of our predictions are smoother than [1]. Moreover, for low bandwidths (below 10%), our predictions are similar to [1], or even tighter. They also grow smoother for higher bandwidths and do not present any saturation point like in [1]. Specifically, for flow 1, which may suffer from buffer backpressure,

[3]This value is similar in this case to infinity since it is very high in comparison to the flows deadline

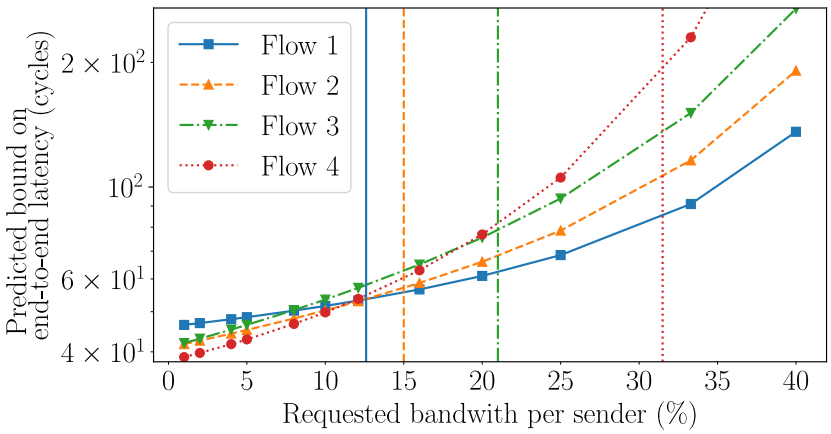



Figure 4.13 – Predicted bounds for different values of bandwidth

the CPA approach predicted upper bound reaches $10^3$ cycles shortly after 12.5% bandwidth. Our bound, on the other hand, is 54 cycles for 12.1% and 57 cycles for 16% bandwidth, very tight in comparison to the simulation results in [1].

Next, we study the impact of buffer size with a constant requested bandwidth per sender (12.5%) for flow 1. We compute the predictions of our model for the same buffer sizes in the experiment by [1] and for additional values, especially for all buffer sizes lower than packet size which are not handled in the CPA model [1]. The derived results of both approaches are illustrated in Figure 4.14.

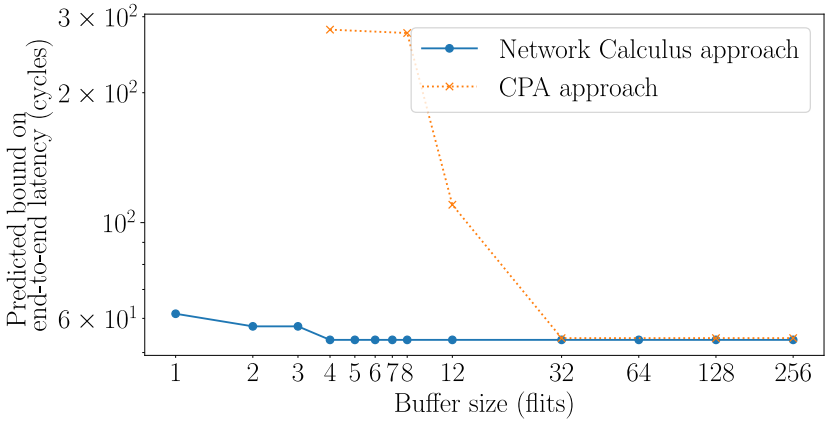



Figure 4.14 – Delay bounds of flow 1 vs buffer size under CPA and NC approaches

As we can notice, our approach allows much tighter delay bounds for small buffer sizes. For buffer sizes lower than 12 flits (3 packets), we obtain a bound tightness improvement of 45.5% with our model in comparison to the CPA one, and it is more than 80% for the lowest buffer size (4 flits). We also notice that increasing buffer size does not improve the delay bound past a certain point under our approach. For

instance, the end-to-end delay upper bound remains constant for buffer sizes above 4 flits (the size of one packet for any flow in the configuration).
This is due to the fact that the spread index of a flow remains constant when the buffer size exceeds the length of one packet. With a constant spread index, the indirect blocking set analysis remains the same; thus the indirect blocking latency (and the end-to-end delay bound).

## 4.8 Conclusions

We presented BATA, a worst-case timing analysis approach for wormhole NoCs integrating the impact of buffer size and flow serialization. It is applicable to any wormhole NoC with direct topology, using deterministic routing and such that flows sharing resources on part of their path do not interfere again after their divergence point. It works with input-buffered routers with VCs and can also model output-buffered routers with no major changes. It covers FP arbitration between traffic classes and any arbitration policy between inputs, and supports both priority sharing and VC-sharing.

To evaluate the tightness, we first studied how the various system parameters impact the computed end-to-end delay bounds. We found our model to be most sensitive to rate and buffer size. Consequently, we proceeded the tightness analysis with a set of different buffer sizes and flow rates. We were able to achieve a tightness ratio up to 80% on average, with reference to worst-case simulation.

We then estimated the scalability of our approach in terms of computation time when increasing the number of flows. We found that the main complexity of the analysis lies in the indirect blocking latency computation, as it leads to additional recursive calls to the function computing the end-to-end service curve. While this complexity comes from the fact that the model is able to cover configurations with shared priority and shared VCs, it causes the method to be hardly scalable past 64 flows. Most configurations we tested with 64 flows or more exceeded the two-hour limit we set on the computation time. Therefore, such a limitation needs to be addressed.
Our next focus will be to generalize BATA to cover a larger panel of NoC configurations (heterogeneous platforms with links of different transmission capacities and different buffer sizes within routers), in addition to considering a more general traffic model covering bursty traffic. Furthermore, we will cope with the complexity

issue to improve the approach scalability.

*Croyant que repartir calmerait mes sanglots,*
*Au futur j'ai promis de faire mes baggages.*
*Mais quand vient le moment de traverser les flots,*
*Il n'y a que des adieux pour celui qui voyage.*

*
* *

Chapter 5

# Extending Buffer-Aware Worst-Case Timing Analysis: Interference Graph Approach

## Contents


**5.1 Problem Statement . . . . . 88**
5.1.1 Illustrative Example . . . . . 88
5.1.2 Main Extensions . . . . . 90
**5.2 Extended System Model . . . . . 91**
5.2.1 Traffic Model . . . . . 91
5.2.2 Network Model . . . . . 91
**5.3 Interference Graph Approach for Indirect Blocking Set . . 93**
**5.4 Refining Indirect Blocking Latency . . . . . 96**
**5.5 G-BATA: Illustrative Example . . . . . 98**
**5.6 Performance Evaluation . . . . . 100**
5.6.1 Computational Analysis . . . . . 101
5.6.2 Sensitivity Analysis . . . . . 107
5.6.3 Tightness Analysis . . . . . 112
**5.7 Conclusions . . . . . 115**


In this chapter, we present a series of extensions of BATA approach to cover heterogeneous platforms and general traffic model, while improving BATA scalability (computational complexity). The proposed approach stems from BATA and uses an interference graph structure in the indirect blocking analysis; thus we called it G-BATA, that can either stand for Graph-BATA or Generalized-BATA.

The remainder of this chapter is organized as follows. We first present the problem statement in Section 5.1, with an illustrative example to highlight a situation in which BATA cannot be applied anymore (Section 5.1.1), and we outline the main

extensions proposed by the new approach (Section 5.1.2). In Section 5.2, we detail how we extend the system model to cope with the new assumptions on the traffic (5.2.1) and the network (5.2.2). Afterwards, Section 5.3 presents our new method for indirect blocking analysis. In Section 5.4, we explain the associated computation of the indirect bocking latency. Section 5.5 illustrates an application of the new method on an example. Finally, we proceed to a performance evaluation of G-BATA in Section 5.6 and conclude the chapter in Section 5.7.

## 5.1 Problem Statement

### 5.1.1 Illustrative Example

To exhibit the impact of considering general traffic model, *e.g.* bursty traffic, when applying BATA, we propose the configuration detailed on Figure 5.1 (left). We assume each buffer can hold one flit and all flows have 3-flit packets, so that each packet can be stored in three buffers. Furthermore, we consider all flows are mapped to the same VC and take flow 1 as the *foi*. When performing the IB analysis as described in Algorithm 1, Section 4.4, we first compute the subpath of flow 2 relatively to flow 1, denoted $\mathbb{S}_a$ on Figure 5.1 (right). We then notice that $\mathbb{S}_a$ does not intersect any path of another flow, and in particular it does not intersect the path of flow 3, $\mathbb{P}_3$. Therefore, the algorithm terminates and we have:

$$IB_1 = \{\}$$

which means flow 3 cannot directly block flow 1.
However, this conclusion relies on the fact that there can be at most one packet of flow 2 in the network. Consider the scenario depicted in Figure 5.2. We suppose a packet A of flow 3 is being transmitted. It requested the North output port of R7 and was granted access. Now its header flit has reached the input port of R8. At the same time, there is a packet B of flow 2 which header flit has reached node R7 input. It also requested the North output of R7, but as A is using it, it has to wait. B is immediately followed by another packet of flow 2, denoted C, which header flit is in R4. C previously requested R3 East output port and was granted it. Here, C has not been completely injected into the network. In fact, B has its last flit in R5 buffer, and is blocked. Consequently, C has to wait that B moves forward to be able to move too. Finally, flow 1 has injected a packet D into the NoC. D has its header flit in R3 and is requesting R3 East output port. However, as C already requested and got the use of that same port, D has to wait.

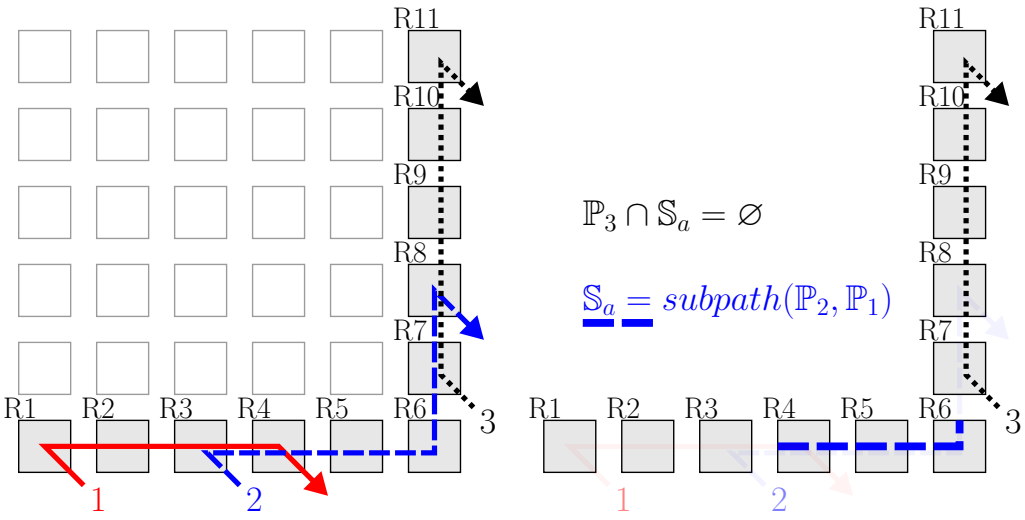



Figure 5.1 – Example configuration (left) and subpaths computation with BATA (right)

In that case, D is indirectly blocked by A. This means that if we assume it is possible to have two consecutive packets of flow 2 in the network, flow 3 may impact flow 1, which was not detected when computing $IB_1$ with BATA.

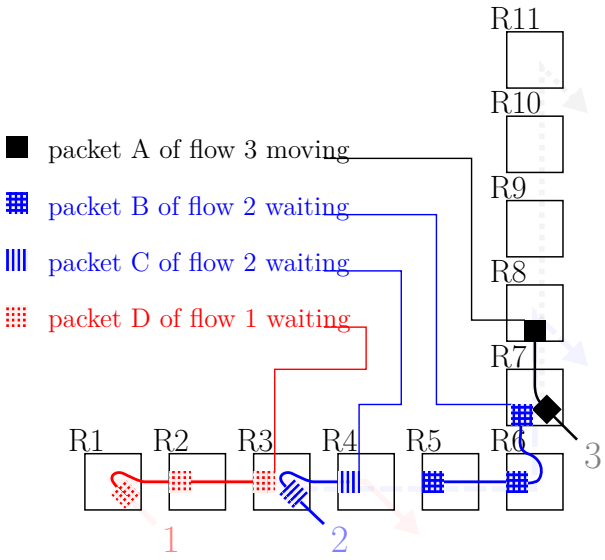



Figure 5.2 – Packet configuration with two instances of flow 2

We refer to this assumption as “Consecutive Packet Queueing” (CPQ). CPQ can happen when considering bursty traffic, *i.e.* flows that can generate and inject a burst of several packets one after the other. Example of such flows include real-time audio and video streams. It can also occur when a packet of a periodic CBR flow experiences enough congestion for the next packet to “catch up” on it. CPQ is not

taken into account by the BATA model.

### 5.1.2 Main Extensions

In the light of the example above, we propose the main following extensions:

- First, we extend the traffic model of BATA to integrate possible CPQ and model bursty traffic flows. We also generalize notions such as spread index to cover heterogeneous architectures, that is NoCs buffer size, link capacities and processing latencies may differ from one router to another. This extended system model will be detailed in Section 5.2.
- Then, we will update the indirect blocking analysis to take into account the changes in the traffic model and integrate CPQ. In that respect, we propose a new method based on a graph structure to model indirect interference. This method is called Interference Graph Approach and will be detailed in Section 5.3.
- Finally, we will adapt the computation of the indirect blocking latency and consequently the end-to-end delay bound. Compared to the original BATA approach, the indirect blocking analysis and the indirect latency computation will change, slightly impacting the end-to-end service curve computation. However, the direct blocking latency computation remains identical. This point will be presented in Section 5.4.

We summarized the steps of G-BATA on Figure 5.3 and highlighted the ones that differ from BATA.

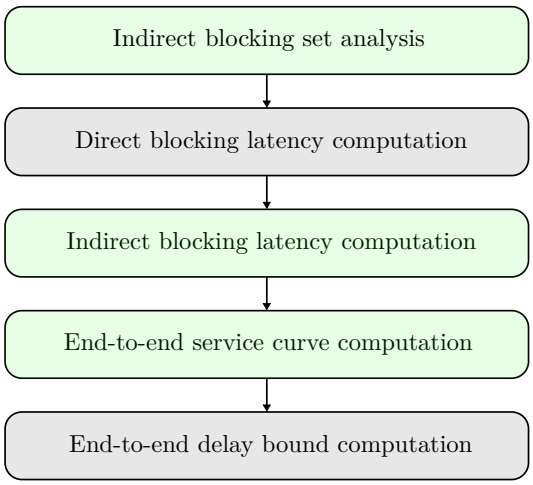



Figure 5.3 – Main steps of G-BATA. Highlighted steps are the differences with BATA approach

## 5.2 Extended System Model

In this section, we present the extended system model that is the base of G-BATA. We first detail the traffic model, then present the network model.

### 5.2.1 Traffic Model

As in BATA, the characteristics of each traffic flow $f \in \mathcal{F}$ are modeled with the following leaky bucket arrival curve:

$$\alpha_f(t) = \sigma_f + \rho_f \cdot t$$

This arrival curve can model a bursty traffic flow. It integrates the maximal packet length $L_f$ (payload and header in flits), the period or minimal inter-arrival time $P_f$ (in cycles), the burst (number of packets the flow may release consecutively) $b_f$ and the release jitter $J_f$ (in cycles) in the following way :

$$\begin{aligned} \rho_f &= \frac{L_f}{P_f} \\ \sigma_f &= b_f \cdot L_f + J_f \cdot \rho_f \end{aligned}$$

If $f$ is CBR flow, we have $b_f = 1$.
We keep other notations identical to BATA. For each flow $f$, its path $\mathbb{P}_f$ is the list of nodes (router-outputs) crossed by $f$ from source to destination. The end-to-end service curve granted to flow $f$ on its whole path is still denoted:

$$\beta_f(t) = R_f \, (t - T_f \,)^+$$

### 5.2.2 Network Model

To account for heterogeneous architectures, we generalize the model of the router. More specifically, instead of considering buffer size, processing capacity and technological latency to be homogeneous, we denote, for any node $r$ on the path of a flow, $B^r$ the available buffer size, $R^r$ the processing capacity, and $T^r$ the technological latency.
From there on, to model the way a packet can spread in the network, we propose an extended definition for the spread index:

**Definition 11.** Extended Spread Index
*Consider a flow $f$ of maximum packet length $L_f$ flits. The spread index of $f$ at*

*node $i$, denoted $N_f^i$, is defined as follows:*

$$N_f^i = \min_{l \geq 0} \left\{ l, L_f \leq \sum_{j=0}^{l-1} B^{\mathbb{P}_f[i+j]} \right\}$$

*where $B^r$ the buffer size at node $r$ in flits.*

$N_f^i$ is the number of buffers needed to store one packet of flow $f$ from node $\mathbb{P}_f[i]$ onwards on the path of $f$.
Using this notion, we extend the definition of the "subpath of $k$ relatively to $l$" to refer to the section of the path of flow $k$ from $\mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$ through $N_k^{\mathrm{dv}(\mathbb{P}_k,\mathbb{P}_l)}$ nodes (at most). We keep the same name and notation as in Definition 7, because the following definition of the subpath is backward-compatible with the previous one.

**Definition 12.** *The subpath of a flow $k$ relatively to a flow $l$ is:*

$$\begin{aligned} subpath(\mathbb{P}_k, \mathbb{P}_l) \quad = \quad & \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l) + 1], \ldots, \\ & \mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l) + N_k^{Last(\mathbb{P}_k,\mathbb{P}_l)+1}]\Big] \end{aligned}$$

*where $Last(\mathbb{P}_k, \mathbb{P}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{P}_l\}$ is the index of the last node shared by $k$ and $l$ along $\mathbb{P}_k$, i.e $\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l)] = \mathrm{dv}(\mathbb{P}_k, \mathbb{P}_l)$.*

As previously, we extend this notion and define the subpath of any flow $k$ relatively to a subpath $\mathbb{S}_l \subset \mathbb{P}_l$ of any flow $l$ (with $l \neq k$ or $l = k$):

**Definition 13.** *The subpath of a flow $k$ relatively to any subpath $\mathbb{S}_l$ of any flow $l$ is:*

$$\begin{aligned} subpath(\mathbb{P}_k, \mathbb{S}_l) \quad = \quad & \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + 1], \ldots, \\ & \mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + N_k^{Last(\mathbb{P}_k,\mathbb{S}_l)+1}]\Big] \end{aligned}$$

*where $Last(\mathbb{P}_k, \mathbb{S}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{S}_l\}$ is the index along $\mathbb{P}_k$ of the last node shared by $k$ and $l$ within $\mathbb{S}_l$. In a similar fashion as in Chapter 4, by abuse of notation, we denote $subpath(k, l)$ to refer to $subpath(\mathbb{P}_k, \mathbb{P}_l)$, and similarly $subpath(k, \mathbb{S}_l)$ to refer to $subpath(\mathbb{P}_k, \mathbb{S}_l)$.*

## 5.3 Interference Graph Approach for Indirect Blocking Set

To handle CPQ assumption, we start from two modifications. First, we allow to compute the subpath of any flow $f$ relatively to a subpath $\mathbb{S}_f \subset \mathbb{P}_f$ of $f$ to model several packets of the same flow queuing in the network. Second, we use a graph structure to maintain the dependency information between the subpaths. By doing so, we are able to know how each subpath was computed, and we also can explore all possible interference patterns more easily.

We use a directed graph where each vertex corresponds to a subpath of a flow and holds the following information :

- **fkey** : the flow identifier;
- **path** : the subpath;
- **dependencies** : the list of all edges $(v, u)$ where $v$ is the current vertex and $u$ is such that $v.path$ is the subpath of flow $v.fkey$ relatively to subpath $u.path$;
- **dependents** : the list of all edges $(w, v)$ where $v$ is the current vertex and $w$ is such that $w.path$ is the subpath of flow $w.fkey$ relatively to subpath $v.path$.

The two functions to construct the graph are detailed in Algorithms 3 and 4. The main steps of Algorithm 3 are as follows:

1. We create a graph with one vertex corresponding to the *foi* (Line 1);
2. We compute all subpaths relatively to the *foi* and create a vertex depending on the *foi*'s vertex for each non-empty subpath (Lines 2 and 7);
3. We add these vertices to the graph, making sure there are no duplicates and merging the dependencies of the new vertex with the existing one if needed (Line 5);
4. We iterate these steps on each new vertex, in a breath-first manner, until no new vertex is created (loop on Line 3).

Once the graph is created, we extract the pairs $(k, subk)$ from all vertices such that $k \notin DB_f \cup \{f\}$, that is from vertices that do not correspond to flows directly interfering with the *foi* $f$.

To construct the graph, we rely on an auxiliary function detailed in Algorithm 4. Its role is to construct new vertices from the previously constructed vertices of the graph. Essentially, it loops over a list of vertices (Line 1. For each vertex, it computes the possible subpaths relatively to the subpath of the current vertex, creates the corresponding vertex (Line 5) and appends it to a list (Line 6, that is returned at the end.

**Algorithm 3** Computing the indirect blocking graph for flow $f$

constructIBGraph($f, \mathbb{P}_f, \mathcal{F}$)

**Input:** $f$, the flow of interest, $\mathbb{P}_f$ the associated path, $\mathcal{F}$ the set of flows
**Output:** $\mathcal{G}_f$, a graph of all subpaths involved in indirect blocking patterns impacting $f$

1: $v_0 \leftarrow$ vertex($f, \mathbb{P}_f, [], []$)
2: $\mathcal{L}_0 \leftarrow$ getNextVertices($[v_0], \mathcal{F}$) *// Initialize a list*
3: **while** $\mathcal{L}_0 \neq []$ **do**
4: **for** $v \in \mathcal{L}_0$ **do**
5: addVertex($\mathcal{G}_f, v$)
6: **end for**
7: $\mathcal{L}_0 \leftarrow$ getNextVertices($\mathcal{L}_0, \mathcal{F}$)
8: **end while**
9: **return** $\mathcal{G}_f$

**Algorithm 4** Computing vertices and adding them to the graph

getNextVertices($\mathcal{L}_{in}$, $\mathcal{F}$)

**Input:** $\mathcal{L}_{in}$, a list of vertices, $\mathcal{F}$ the flow set
**Output:** $\mathcal{L}_{out}$, a list of the vertices depending on the vertices of $\mathcal{L}_{in}$

1: **for** $v \in \mathcal{L}_{in}$ **do**
2: **for** $k \in \mathcal{F}$ **do**
3: $\mathbb{S} \leftarrow subpath(k, v.path)$
4: **if** $\mathbb{S} \neq \varnothing$ **then**
5: $w \leftarrow$ vertex($k, \mathbb{S}, [v], []$)
6: append $w$ to $\mathcal{L}_{out}$
7: **end if**
8: **end for**
9: **end for**
10: **return** $\mathcal{L}_{out}$

addVertex($\mathcal{G}, v$)

**Input:** $\mathcal{G}$, a graph, $v$, a vertex to add
**Output:** void, graph $\mathcal{G}$ is updated

1: **if** $\exists w \in \mathcal{G}$ such that $w.path = v.path$ and $w.fkey = v.fkey$ **then**
2: merge $v$ with $w$
3: **else**
4: add $v$ to $\mathcal{G}$
5: **end if**

The computational complexity of Algorithm 3, when considering a flow set $\mathcal{F}$ on the NoC, is denoted as $\mathcal{C}(|\mathcal{F}|)$ and is defined in the following property.

**Property 2.** *Consider a flow set $\mathcal{F}$, the computational complexity of Algorithm 3 is as follows:*

$$\mathcal{C}(|\mathcal{F}|) = \mathcal{O}\left(\max_{f\in\mathcal{F}} |\mathbb{P}_f| \cdot \sum_{f\in\mathcal{F}} |\mathbb{P}_f|\right) \tag{5.1}$$

*and can be roughly bounded as follows :*

$$\mathcal{C}(|\mathcal{F}|) = \mathcal{O}\left((\max_{f\in\mathcal{F}} |\mathbb{P}_f|)^2 \cdot |\mathcal{F}|\right) \tag{5.2}$$

*Proof.* We first notice that vertices of the graphs are defined only by their flow index and subpath. For a flow $f$, there are $|\mathbb{P}_f|$ possible subpaths (each of them starting at a different node of the path of $f$). Therefore, there are at most $\sum_{f\in\mathcal{F}} |\mathbb{P}_f|$ distinct subpaths for the flow set $\mathcal{F}$.
We can thus bound the number of vertices of the computed graph. For each of these vertices, the algorithm computes all possible subpaths relatively to the current vertex' subpath (in `getNextVertices()` main loop).
Assume this subpath is $S$ and that we have a preprocessed dictionary listing, for every node, the indexes of flows using this node.[1] Although we wrote the secondary loop of `getNextVertices()` as a loop over all flows in $\mathcal{F}$ for clarity reasons, all we have to do to get all possible subpaths relatively to $S$ is run through the nodes of $S$ and check for intersection with another flow's path. Comparing the indexes of the current node with those of the previous node, we can find divergence nodes of contending flows relatively to $S$. We assume that, knowing the divergence point of a contending flow relatively to $S$, it takes a constant time to find its subpath (we only need to compute the spread index).
Thus, the complexity of finding all subpaths relatively to any subpath is $\mathcal{O}(\max_{f\in\mathcal{F}} |\mathbb{P}_f|)$, hence the final result. The last bound is found bounding each path length of the sum by the maximal path length in the whole flow set. □

The reason we can account for more than one packet of the same flow stalling in the network is because we allow to compute the subpath of a flow relatively to a subpath of that very same flow.

[1] We do run such a preprocessing on the configuration.

## 5.4 Refining Indirect Blocking Latency

When using the G-BATA approach, we take into account the possible queueing of several packets of each flow through the consideration of multiple consecutive subpaths for one flow. Therefore, when computing $T_{IB}$, the main difference compared to the BATA approach is that, for each {flow index, subpath} pair of the derived IB set, we do not need to compute the arrival curve at the beginning of the subpath and instead use the initial arrival curve of one packet of the corresponding flow. Having several consecutive subpaths for the same indirectly interfering flow allows to take into account a burst of more than one packet.
Computation of the indirect blocking latency $T_{IB}$ is done using the following Theorem :

**Theorem 4.** *(Maximum Indirect Blocking Latency)*
*The maximum indirect blocking latency for a* foi *$f$ along its path $\mathbb{P}_f$, in a NoC under flit-level preemptive FP multiplexing with strict service curve nodes of the rate-latency type $\beta_{R,T}$ and leaky bucket constrained arrival curves $\alpha_{\sigma,\rho}$, is as follows:*

$$T_{IB} = \sum_{\{k,subP\}\in IB_f} \frac{L_k + J_k\rho_k}{\widetilde{R}_k^{subP}} + \widetilde{T}_k^{subP} \tag{5.3}$$

*where:*

$$\widetilde{R}_k^{subP} = \min_{r\in subP}\left\{R^r - \sum_{j\ni r, j\in hp(f)} \rho_j\right\} \tag{5.4a}$$

$$\widetilde{T}_k^{subP} = \sum_{r\in subP}\left(T^r + \frac{S_{flit}\mathbf{1}_{\{lp(k)\supset r\}}}{R^r}\right) + \sum_{i\in DB_k^{|subP}\cap hp(k)} \frac{\sigma_i^{\mathrm{cv}(i,k)} + \rho_i \sum\limits_{r\in subP\cap\mathbb{P}_i}\left(T^r + \frac{S_{flit}\mathbf{1}_{\{lp(k)\supset r\}}}{R^r}\right)}{\widetilde{R}_k^{subP}} \tag{5.4b}$$

*Proof.* For any pair $\{j, subP_j\} \in IB_f$, a packet of flow $j$ will impact the *foi* $f$ during the maximum time it occupies the associated subpath $subP_j$, $\Delta t_j^{max}$. Hence, a safe upper bound on the indirect blocking latency is as follows:

$$T_{IB} \leq \sum_{\{j,subP_j\}\in IB_f} \Delta t_j^{max}$$

On the other hand, for any pair $\{j, subP_j\} \in IB_f$, $\Delta t_j^{max}$ is upper bounded by the end-to-end delay bound of one packet of flow $j$ along its associated subpath $subP_j$,

$D_j^{subP_j}$, which infers the following:

$$T_{IB} \leq \sum_{\{j,subP_j\} \in IB_f} D_j^{subP_j} \tag{5.5}$$

Based on Theorem 6, Appendix B.1, the delay bound of flow $j$, $D_j^{subP_j}$, is computed as the maximum horizontal distance between:

- the maximum arrival curve for a single packet of flow $j$ at the input of the subpath $subP_j$, $\alpha_j^{subP_j[0]}$. We consider one packet per subpath. This is due to the fact that each subpath holds one packet (from the definition of the spread index). The multiple number of packets is taken into account through the multiple consecutive subpaths of the same flow. Thus, the considered arrival curve is the initial arrival curve of flow $j$ with $b_j$ equal to one, that is with a burst equal to $L_j + J_j\rho_j$;
- the granted service curve to flow $j$ by its VC along $subP_j$, $\widetilde{\beta}_j^{subP_j}$, called VC-service curve, when ignoring the same-priority flows (which are already included in $IB_f$). The latter condition is due to the pipelined behavior of the network, where the same-priority flows sharing $subP_j$ are served one after another if they need shared resources. Hence, the impact of the same-priority flows than flow $j$ is already integrated within the sum expressed in Eq. (5.5).

To compute the granted service curve $\widetilde{\beta}_j^{subP_j}$ for each flow $j \in IB_f$ along $subP_j$, we follow similar approach than in the proof of Theorem 2 through applying the existing Theorem 7, when:

- ignoring the same-priority flows in $sp(j)$, thus all $shp(j)$ will become $hp(j)$ and $slp(j)$ will become $lp(j)$ in Eqs. (B.1a) and (B.1b);
- considering the flit-level preemption, thus the impact of lower-priority flows in Eq. (B.1a) is bounded by the maximum transmission time of $S_{flit} \cdot \mathbf{1}_{\{lp(k) \supset r\}}$ within each crossed node $r \in subP_j$;
- considering only the direct blocking flows of $j$ along $subP_j$, thus considering $DB_j^{|subP_j} \cap hp(j)$ in Eq. (B.1b).

Hence, we obtain $\widetilde{R}_j^{subP_j}$ and $\widetilde{T}_j^{subP_j}$ described in Eqs. (5.4a) and (5.4b), respectively. Consequently, the maximum indirect blocking latency in Eq. (5.5) can be re-written as follows:

$$T_{IB} \leq \sum_{\{j,subP_j\} \in IB_f} \frac{L_j + J_j\rho_j}{\widetilde{R}_j^{subP_j}} + \widetilde{T}_j^{subP_j} \tag{5.6}$$

□

What is interesting, compared to the BATA approach, is that we do not need to

**Algorithm 5** Computing the end-to-end service curve for a flow $f$

endToEndServiceCurve$(f, \mathbb{P}_f)$

1: Compute $R_f$
2: Compute $T_{\mathbb{P}_f}$
*// Compute $T_{DB}$:*
3: $T_{DB} \leftarrow 0$
4: **for** $k \in DB_f$ **do**
5: $r_0 \leftarrow \text{cv}(k, f)$ *// Get convergence point of $f$ and $k$*
6: $\beta_k \leftarrow \text{endToEndServiceCurve}(k, [\mathbb{P}_k[0], \cdots, r_0])$
7: $\alpha_k^0 \leftarrow$ initial arrival curve of $k$
8: $\alpha_k \leftarrow \alpha_k^0 \oslash \beta_k$
9: $T_{DB} \leftarrow \text{directBlocking}(\alpha_k^{r_0})$
10: **end for**
*// Compute $T_{IB}$:*
11: Compute $IB_f$ according to G-BATA
12: $T_{IB} \leftarrow 0$
13: **for** $\{k, S\} \in IB_f$ **do**
14: $\widetilde{\beta}_k \leftarrow$ VC-service curve of $k$ on $S$
15: $\alpha_k \leftarrow$ initial arrival curve of $k$ for $b_k$=1
*// Now add the latency over the subpath to $T_{IB}$ :*
16: $T_{IB} \leftarrow T_{IB} + h(\alpha_k, \widetilde{\beta}_k)$
17: **end for**
18: **return** $\beta = R_f(t - (T_{\mathbb{P}_f} + T_{DB} + T_{IB}))^+$

propagate the arrival curves of flows in $IB_f$ at the beginning of the subpaths when computing $T_{IB}$. Consequently, our new approach does not need to compute service curves upstream the subpaths, which decreases the number of recursive calls to `endToEndServiceCurve()` in Algorithm 2. For that reason, we expect a complexity gain, that we will show and estimate in Section 5.6.1.

To illustrate this last aspect, we rewrite the algorithm for service curve computation, but this time assuming G-BATA approach is used (Algorithm 5). The differences with Algorithm 2 are in Line 11, when computing the indirect blocking set using the interference graph approach, and in Line 15, when we consider the initial arrival curve instead of the propagated one.

## 5.5 G-BATA: Illustrative Example

In this section, we detail the complete G-BATA approach on the example configuration displayed on Figure 5.1.

We assume all routers have a service curve $\beta = R(t - T)^+$ and flow $i$ has a packet length $L_i = L$ and the initial arrival curve $\alpha_i = \sigma + \rho t$. All flows are mapped to

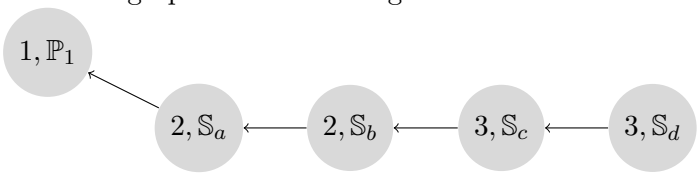
the same VC, thus $T_{hp} = T_{lp} = 0$. We also assume all jitters equal zero, and the burst of flows is 2. Furthermore, although G-BATA is able to handle buffers with different sizes, we assume all buffers can hold one flit.

**Step 1: Extended buffer-aware analysis of the indirect blocking set**

We first apply Algorithm 3. The subpaths corresponding to the computed vertices are represented on Figure 5.4:

1. starting from flow 1, we create vertex $v_1$ with index 1 and path $\mathbb{P}_1$ and we call `getNextVertices()` on $[v_1]$. We get $v_2 = \text{vertex}(2, \mathbb{S}_a)$. Since $v_2$ was computed from $v_1$, we add $v_2$ in $v_1.dependents$ and $v_1$ in $v_2.dependencies$.
2. we call `getNextVertices()` on $[v_2]$. We get $v_2' = \text{vertex}(2, \mathbb{S}_b)$, add $v_2'$ in $v_2.dependents$ and $v_2$ in $v_2'.dependencies$;
3. we call `getNextVertices()` on $[v_2]$. We get $v_3 = \text{vertex}(3, \mathbb{S}_c)$;
4. we call `getNextVertices()` on $[v_3]$. We get $v_3' = \text{vertex}(3, \mathbb{S}_d)$.
5. we call `getNextVertices()` on $[v_3']$. It returns the empty list [] and the algorithm terminates.

The final graph is the following:




and the associated IB set :

$$IB_1 = \{\{3, \mathbb{S}_c\}, \{3, \mathbb{S}_d\}\}$$

**Step 2: End-to-end service curve computation**

**Direct blocking latency computation**

As this step is almost identical to BATA, except for the fact that $b_1 = 2$, which implies $\sigma_1 = 2L_1$, we do not detail it and instead simply give the final result. The astute reader is more than welcome to check that:

$$\begin{aligned} T_{DB} &= 4T + \frac{\sigma_1}{R-\rho} + \rho\frac{T + \frac{L}{R}}{R-\rho} \\ &= 10,526315789 \text{ cycles} \end{aligned}$$

**Indirect blocking latency computation**

We now use Theorem 4 and the computed IB set $IB_1$ to derive the indirect blocking

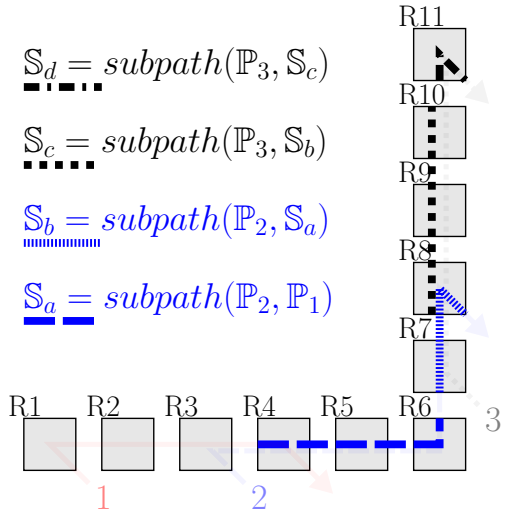



Figure 5.4 – Subpaths computation with G-BATA approach

latency.

$$\begin{aligned} T_{IB} &= \frac{L_3 + J_3\rho_3}{R_3^{\mathbb{S}_c}} + T^{\mathbb{S}_c} + \frac{L_3 + J_3\rho_3}{R_3^{\mathbb{S}_d}} + T^{\mathbb{S}_d} \\ &= \frac{L_3}{R_3^{\mathbb{S}_c}} + T^{\mathbb{S}_c} + \frac{L_3}{R_3^{\mathbb{S}_d}} + T^{\mathbb{S}_d} \\ &= 2\frac{L}{R} + 6T \quad \text{since there are no higher priority flows} \\ &= 12 \text{ cycles} \end{aligned}$$

**Step 3: End-to-end delay bound computation**

We now compute the end-to-end delay bound for flow 1 with Equation A.1:

$$\begin{aligned} D_1^{\mathbb{P}_1} &= \left\lceil \frac{\sigma}{R-\rho} + 4T + \frac{\sigma}{R-\rho} + \rho\frac{T + \frac{L}{R}}{R-\rho} + 2\frac{L}{R} + 6T \right\rceil \\ &= 29 \text{ cycles} \end{aligned}$$

## 5.6 Performance Evaluation

In this section, we first analyse the computational effort of G-BATA and particularly on heavy configurations, with reference to BATA. Afterwards, we conduct a sensitivity analysis of the proposed approach when varying the system parameters and analyze their effect on the end-to-end delay bound. Finally, we assess the tightness of the derived bounds, using the insight we got thanks to the sensitivity analysis.

Unlike the performance evaluation phase of the previous chapter, we chose to start with the computational analysis to assess the enhancement in terms of complexity with reference to BATA. Moreover, this analysis highlights some new parameters which have a great impact on the tightness analysis.

### 5.6.1 Computational Analysis

In this section, we study the computational aspect of G-BATA. We will first run G-BATA on the same randomly-generated configurations as in Section 4.7.3, with 4, 8, 16, 32, 48, 64, 80, 96 and 128 flows on a $8 \times 8$ NoC, to compare it with BATA. There are 20 configurations for each flow number, and we set a time limit of two hours for the analysis.

For each configuration, we will focus on the following *complexity metrics*, that give an idea of the cost of analyzing a configuration:

- $\Delta t$, the total analysis runtime;
- $\Delta t_{IB}$, the duration of the IB analysis (for BATA, determining IB set; for G-BATA, constructing the interference graph);
- $\Delta t_{e2e}$, the duration of all end-to-end delay bounds computation;
- $N_{e2e}$, the number of calls to the function `endToEndServiceCurve()`;
- $N_{iter}$, the number of calls to a representative IB analysis function:
  - for BATA, the number of *while* iterations;
  - for G-BATA, the number of calls to the function `getNextVertices()`;

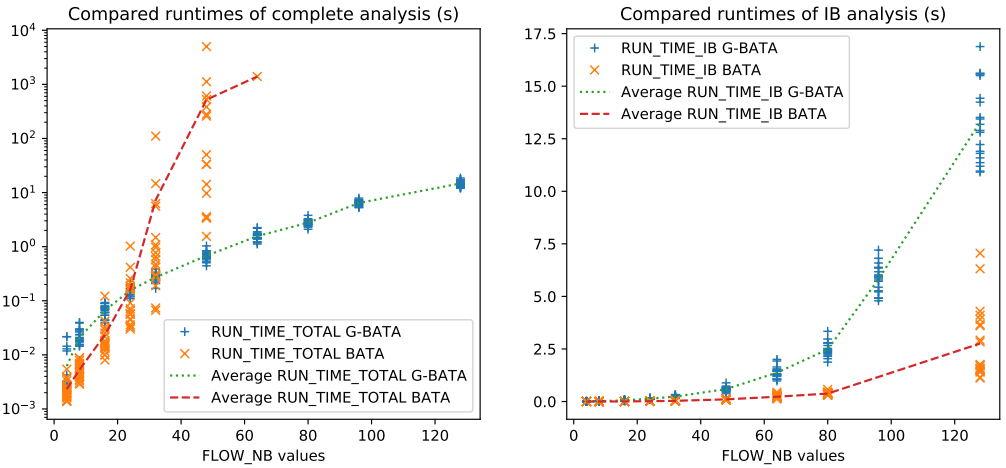



Figure 5.5 – Compared runtimes of both approaches: total runtimes (left) and IB analysis runtimes (right)

We begin the comparative study by plotting the total analysis runtime $\Delta t$ as well

as the duration of IB analysis $\Delta t_{IB}$ as a function of the number of flows in the configuration (Figure 5.5). The first thing we can notice, on the left graph, is that BATA takes more time than G-BATA, especially for flow sets of more than 32 flows. For instance, the total analysis of 48-flow configurations is on average 766 times faster with G-BATA than with BATA. There were no timeouts for G-BATA, whereas BATA timed out for most configurations with 64 flows or more.
However, we expect the IB analysis part of BATA approach to be computationally less expensive than G-BATA. Since the IB analysis is independent from the end-to-end service curve and delay bound computation, we were able to do it with no time-outs. We have plotted the runtimes of IB analysis part *vs* flow number for the two approaches to check this intuition (right graph of Figure 5.5). The result is very explicit: IB analysis of BATA is faster than G-BATA. For instance, on 48-flow configurations, BATA is on average 5.7 times faster than G-BATA.
In an attempt to be more platform-independent, we have used other metrics than runtimes to estimate the complexity of analyses. To do so, we counted the number of calls of relevant functions. For the end-to-end delay bounds computations, we counted the total number of calls to the function `endToEndServiceCurve()`, which is used in both approaches. For the IB analysis part, the two approaches are significantly different; thus, we counted the number of iterations of the *while* loop for BATA and the number of calls to the function `addVertex()` for G-BATA when this function creates a new vertex.[2] The number of calls to `addVertex()` in G-BATA is roughly the equivalent of the number of *while* iterations of BATA.
We gathered the results in Figure 5.6. We plotted two graphs: one for the service curve computation (left), the other one for the IB analysis (right). The results match what the runtime graphs showed: G-BATA is way faster on the end-to-end service curve computation, while BATA is faster on IB analysis. More precisely, for the total analysis of 48-flow configurations, BATA performs on average 1883 times as many calls to `endToEndServiceCurve()` as G-BATA does. For the IB analysis, G-BATA performs on average 1.5 times as many IB analysis iterations as BATA does.
We then performed additional experiments on randomly generated configurations for G-BATA approach, on a $8 \times 8$ NoC, with a number of flows from 20 to 800, to study how well the new method scales on large flow sets. As before, we perform the analysis and measure total runtime, runtime of the IB analysis and runtime of the service curve computation. We plot the results on Figure 5.7. What comes out of

[2] It can also merge the vertex passed in argument with an existing one; in that case we do not count this call as significant.

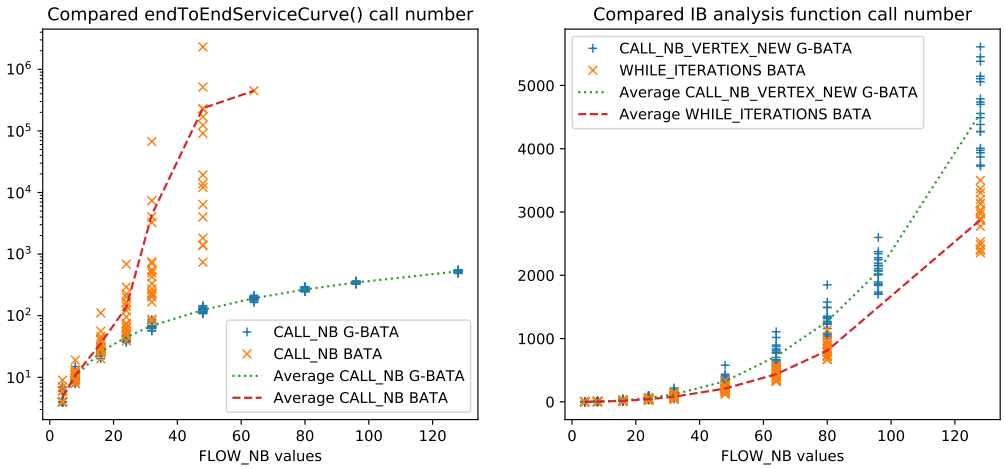



Figure 5.6 – Comparative study of the algorithmic complexity

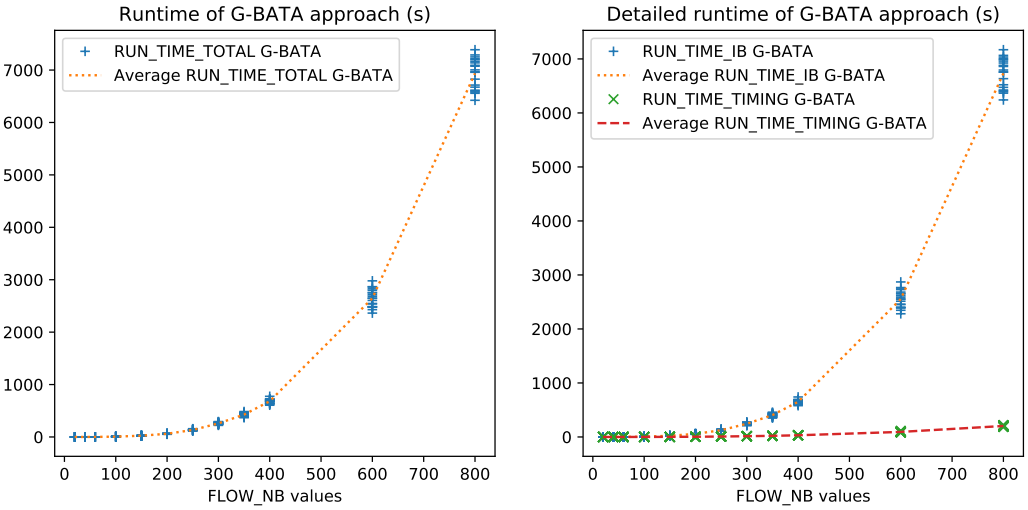



Figure 5.7 – Studying the scalability of G-BATA on large flow sets. RUN_TIME_IB denotes the duration of the IB analysis, while RUN_TIME_TIMING denotes the duration of the service curve computation.

this additional study is that G-BATA analysis scales well: without parallelization, on a laptop powered by an Intel core i7 processor, computing end-to-end delay bounds for each of the 800 flows takes around 7200 seconds in the worst case (2 hours), *i.e.* around 9 seconds per flow, as shown on the left graph of Figure 5.7. Moreover, the IB analysis runtime is the more computationally expensive phase: for the 800-flow configurations, it represents on average 97.1% of the total runtime,

as illustrated on the right graph of Figure 5.7.

**Key points:** G-BATA approach scales way better than BATA approach. The difference is especially visible for flow sets of 32 and 48 flows, where the average runtime of the total analysis for BATA is 10 to 100 times higher than G-BATA. For bigger configurations, we have not been able to get much comparative information as running one analysis with BATA takes more than two hours. Moreover, G-BATA approach performs well on heavy configurations (600 and 800 flows) with an average total runtime of 2647 and 6935 seconds, respectively. Finally, we notice that depending on the approach, the more computationally expensive step is either the indirect blocking analysis (G-BATA) or the service curve computation (BATA). For the latter case, it is what limits BATA approach scalability for large flow sets.

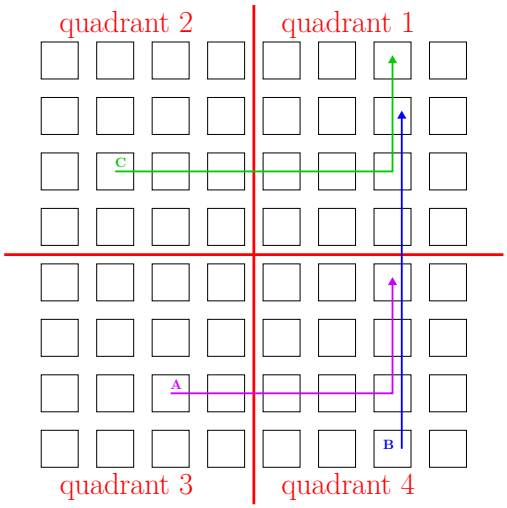



Figure 5.8 – Quadrants of the NoC and illustration of flows from families A, B and C

From these illustrated results, we can notice that for a given number of flows, runtimes can vary significantly from one configuration to another. For instance, on the left graph of Figure 5.5, for 48-flow configurations, runtimes differ by up to 57% and up to 99% for G-BATA and BATA, respectively. Hence, the configuration complexity seems to not only depend on the number of flows, but also on at least another hidden parameter.

In an attempt to better understand what are the configuration parameters impacting the approach complexity, we define two congestion indexes.

**Definition 14.** *Given a configuration $\mathcal{F}$ and a* foi *$f$, the direct blocking index (DB*

*index) of $f$, denoted $I_{DB}(f)$, is the number of flows in the direct blocking set of $f$:*

$$I_{DB}(f) = |DB_f|$$

**Definition 15.** *Given a configuration $\mathcal{F}$ and a* foi *$f$, the indirect blocking index (IB index) of $f$, denoted $I_{IB}(f)$, is the number of {flow index, subpath} pairs in the indirect blocking set of $f$:*

$$I_{IB}(f) = |IB_f|$$

The value of one such index is specific to one flow. Hence, to quantify how complex a configuration is, we introduce the following average indexes:

- $|\mathcal{F}|$, the number of flows of the configuration;
- $\overline{I_{IB}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} I_{IB}(f)$, the average IB index of flow set $\mathcal{F}$;
- $\overline{I_{DB}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} I_{DB}(f)$, the average DB index of flow set $\mathcal{F}$.

To evaluate the impact of these introduced indicators on the runtime of BATA and G-BATA, we randomly generated another series of 4-, 8-, 16- and 32-flow configurations (20 configurations per number of flows), but this time following a different paradigm. We split the NoC into 4 quadrants (Figure 5.8). Then, we randomly choose flows according to 3 different sets, A, B and C:

- flows in A have their source in the 3$^{\text{rd}}$ quadrant and their destination in the 4$^{\text{th}}$ quadrant;
- flows in B have their source in the 4$^{\text{th}}$ quadrant and their destination in the 1$^{\text{st}}$ quadrant;
- flows in C have their source in the 2$^{\text{nd}}$ quadrant and their destination in the 1$^{\text{st}}$ quadrant.

It is worth noticing that these communication patterns favor direct and indirect blocking, which impact the introduced direct and indirect blocking indexes.

We perform the same analysis as before and compare the results we get for both approaches, on these constrained configurations (referred to as “constrained”) and the previous 4-, 8-, 16- and 32-flow configurations.

We first plot total runtime as a function of flow number, and the average curve (Figure 5.9). We notice that for both G-BATA and BATA approaches, there is a noticeable difference between the constrained and the uniformly distributed configurations. For a given number of flows, constrained sets generally require greater

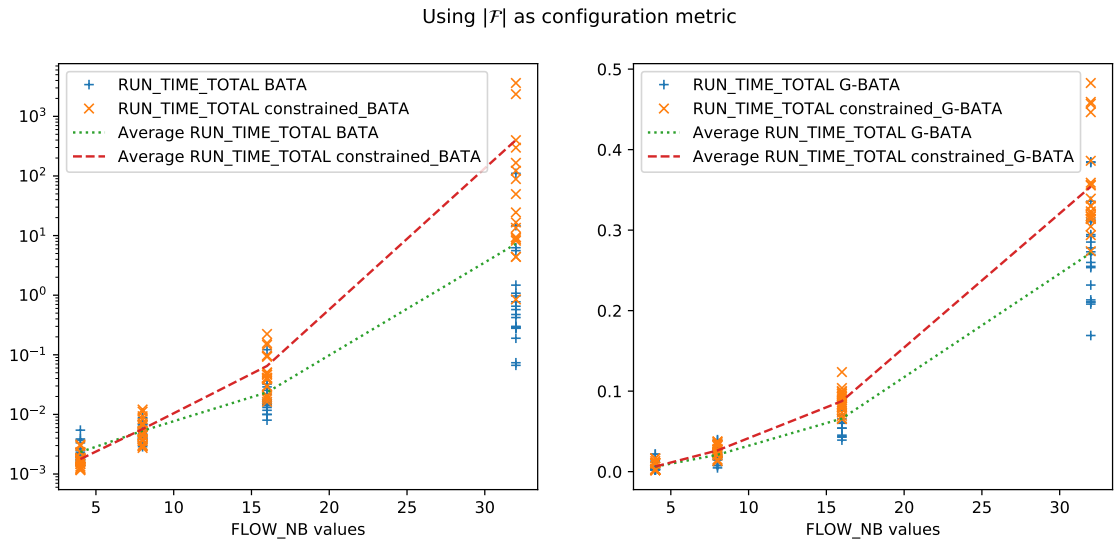



Figure 5.9 – Runtimes *vs* flow number for both configuration types for BATA (left) and G-BATA (right)

runtimes than the previous sets. We did not include the plots of other runtimes (IB analysis and service curve computation) *vs* flow number for the two configuration types, but they exhibit the same trend as total runtimes.

Hence, to better understand the correlation between the runtime and the congestion pattern, we focus on the 32-flow configurations and we plot, for both approaches, all points $(x, y)$ where:

- $x$ is the average DB index (resp. IB index) of the configuration, $\overline{I_{DB}}$ (resp. $\overline{I_{IB}}$);
- $y$ is the total analysis runtime.

The results are gathered in Figure 5.10. For both approaches, we notice that the runtime tends to increase with the average congestion index (direct or indirect). We conclude that a higher average congestion index (direct or indirect) tends to characterize configurations that require a higher computation time.

Moreover, the average IB index does not bring more insights than the average DB index on how computationally expensive the analysis of a configuration may be. So, given that it is computationally more expensive to compute the average IB index than the average DB index, especially for G-BATA approach, we conclude that average DB index is a good configuration indicator to quantify the complexity of a configuration in addition to the number of flows.

**Key points:** Although there is a correlation between the number of flows of one set and the runtime needed to perform its analysis, we find that it is not sufficient

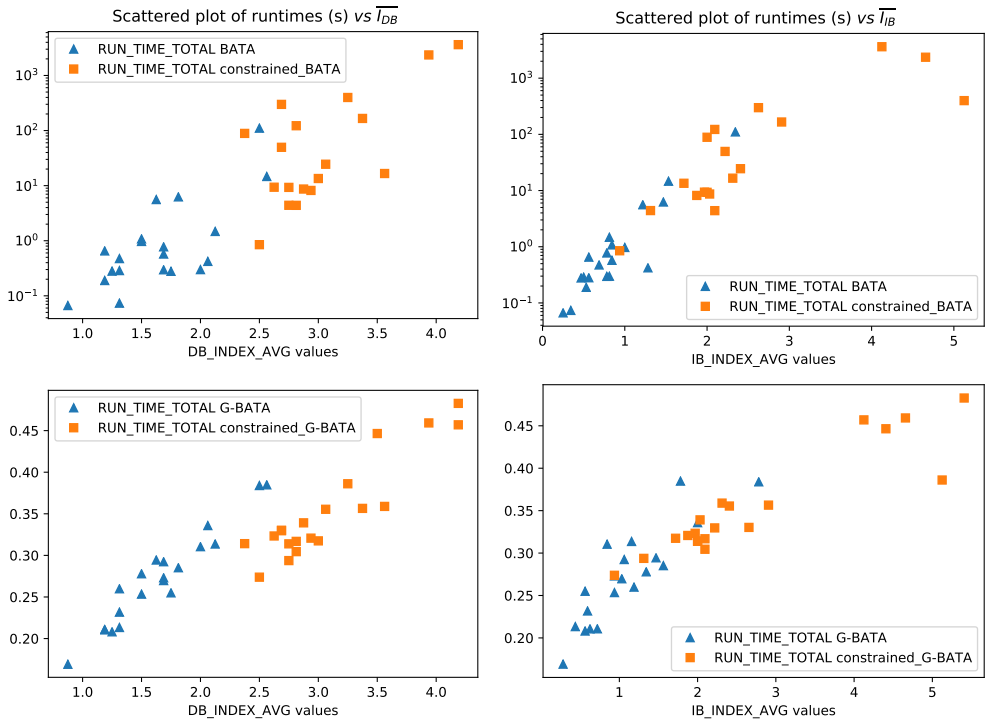



Figure 5.10 – Studying the correlation between average DB index (resp. average IB index) and total runtime for 32-flow configurations, for BATA (top graphs) and G-BATA (bottom graphs)

to characterise how long the timing analysis may take. In that respect, we propose two configuration indicators to refine the quantitative aspect of the complexity of a flow set: the average DB and IB indexes. We show that both are adequate complementary configuration parameters. Nonetheless, the average IB index is computationally more expensive while not bringing much more information. Hence, the DB index and the size of the flow set are considered as sufficient to represent a configuration complexity.

### 5.6.2 Sensitivity Analysis

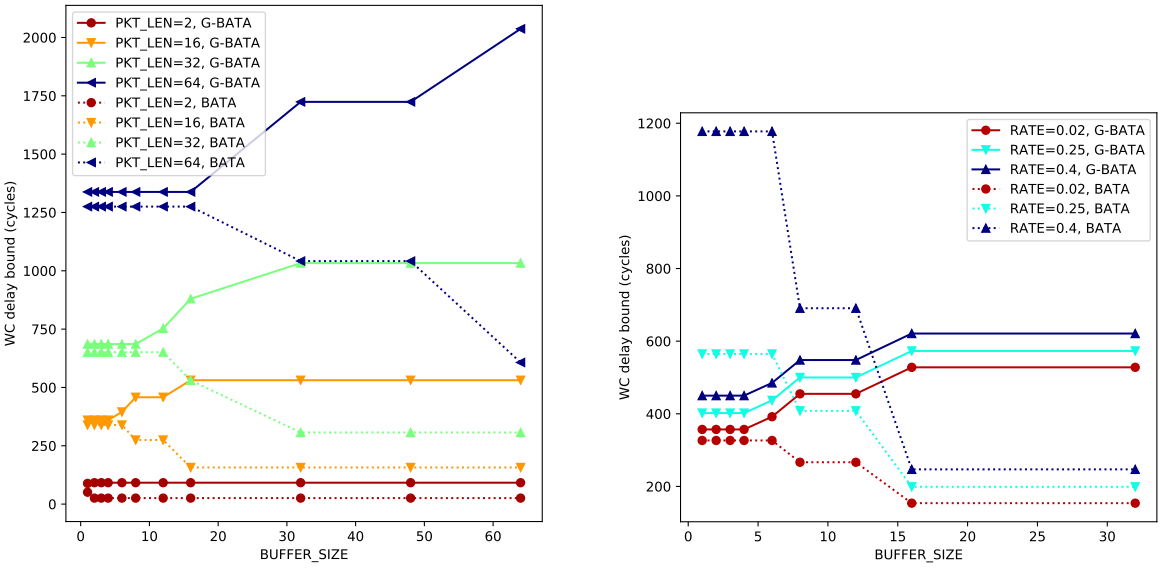
In this section, we study the impact of different parameters on the end-to-end delay bounds yielded by G-BATA. We will process as we did with BATA in Section 4.7.1, and plot the same graphs. To better highlight the impact of the various parameters on G-BATA in reference to BATA, we display the results of G-BATA along with the existing results obtained with BATA.

We still consider the configuration described on Figure 4.6, with 12 flows and an

average DB index of 2. We recall our assumptions:

- each router can handle one flit per cycle and it takes one cycle for one flit to be forwarded from the input of a router to the input of the next router, *i.e.*, for any node $r$, $T^r = 1$ cycle and $R^r = 1$ flit/cycle;
- all the flows are mapped on the same VC;
- our flow of interest is flow 1.

Figure 5.11 illustrates the end-to-end delay bounds of the *foi* when varying buffer size. For the left graph, we keep each flow rate constant at 4% of the total bandwidth; whereas for the right graph, we keep each flow packet length at 16 flits.




(a) Constant rate

(b) Constant packet length

Figure 5.11 – Compared buffer size impact on end-to-end delay bounds

First, on both graphs, we notice an opposite trend between G-BATA and BATA approaches. The former predicts that delay bounds increase when buffer size increases, whereas the latter predicts that delay bounds decrease. This is mainly due to the variation of the spread index of flows and its impact on each approach. When buffer size increases, spread indexes of flows decrease, causing the subpaths to be smaller.

For BATA, this generally makes the IB set smaller: as this approach does not consider CPQ, reducing the length of a subpath reduces the possibility that this subpath intersects with the paths of other flows. Consequently, the derived IB latency tends to decrease, as well as the end-to-end delay bound.

For G-BATA approach, however, the interference graph takes CPQ into account, and in that respect, the number of consecutive packets is not bounded. Therefore, reducing the size of the subpaths increases their number. The extracted IB set thus contains more subpaths of smaller size. Consequently, there are more terms in the indirect blocking delay sum (Equation 5.3), which may increase the end-to-end delay bound.

Second, we notice that with both approaches, the end-to-end delay bounds increase with the packet length and rate. Moreover, we observe that past a certain value of buffer size, the end-to-end delay bounds remain constant. This corresponds to the IB set remaining constant once buffers are large enough to hold one packet (spread index of 1 for all flows).
Finally, on the right graph, we notice that BATA is more sensitive to rate than G-BATA: for buffer sizes below 6 flits, BATA predicts delay bounds between 327 and 1178 cycles, while G-BATA gives delay bounds between 357 and 486 cycles.

**Key points:** Although increasing buffer size may improve end-to-end delay bounds when no CPQ happens (under BATA), we find that it does not impact favorably the end-to-end delay bound when CPQ can occur and the number of consecutive packets queueing is not limited. Moreover, G-BATA is less sensitive to rate variations than BATA for small buffer sizes.

Next, we focus on the packet length impact on the end-to-end delay bound for G-BATA and BATA, as illustrated on Figure 5.12 and 5.13, respectively. For clarity reason, we plotted separate graphs for the two approaches. On each figure, the left graphs present results when the buffer size is constant (4 flits) and the right ones when the rate of each flow is constant (4% of the link capacity).
The first observation we can make from all graphs is that the delay bounds evolve in an almost linear manner with the packet length. For instance, on the right G-BATA graph, with 8 flits of buffer size and packet length equal to 64, 96 and 128 flits, the ratios of packet length and end-to-end delay bound are 20.9, 20.7 and 20.6, respectively.
Still on the same right graph, we observe further interesting aspects:

- At a given packet length, the buffer size has a limited impact on the end-to-end delay bounds. For instance, for a packet length of 64 flits, the delay bounds increase with less than 30% when the buffer size increases with 480%;
- For packet lengths that are significantly larger than buffer size, the delay

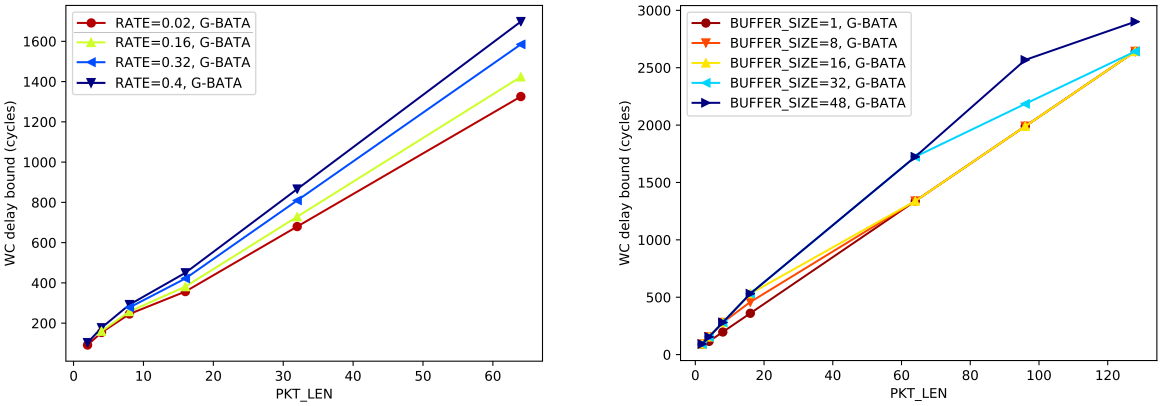



Figure 5.12 – Packet length impact on G-BATA end-to-end delay bounds

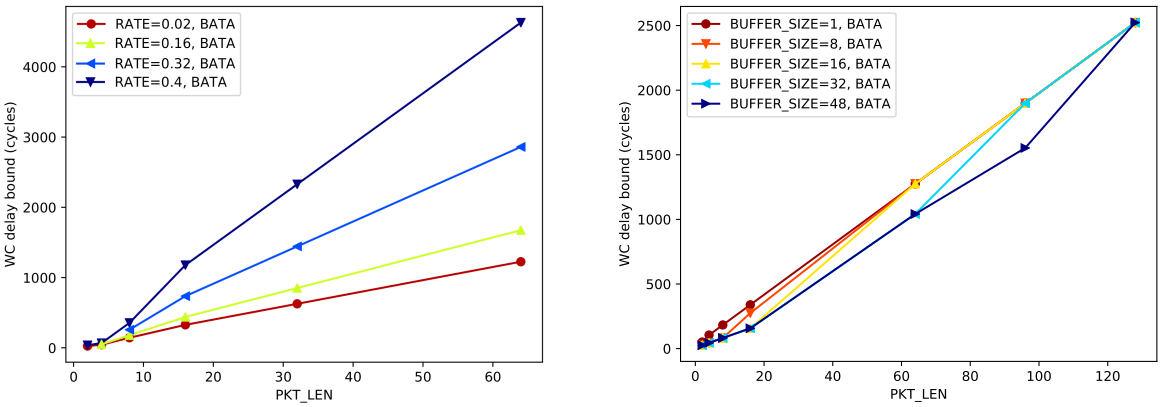



Figure 5.13 – Packet length impact on BATA end-to-end delay bounds

bound remains constant regardless of the buffer size, *e.g.*, it is the case for a packet length of 128 flits.

Similar observations can be made for BATA approach.

However, looking at the left graphs for BATA and G-BATA, we notice that BATA is more sensitive to rate variations than G-BATA: for a packet of 64 flits, when the rate increases from 2% to 40%, the end-to-end delay bound yielded by BATA increases from 1226 cycles to 4630 cycles (+278%) while the delay bound predicted by G-BATA increases only from 1326 cycles to 1698 cycles (+28%).

**Key points:** at a given rate and packet length, we observe that buffer size has a limited impact on the end-to-end delay bound, and this observation is valid

for both G-BATA and BATA approaches. We also notice that the evolution of the delay bound with the packet length follows an almost linear trend, for both approaches as well. Finally, we further confirm that BATA is more sensitive to rate variations than G-BATA, especially for large packet lengths.

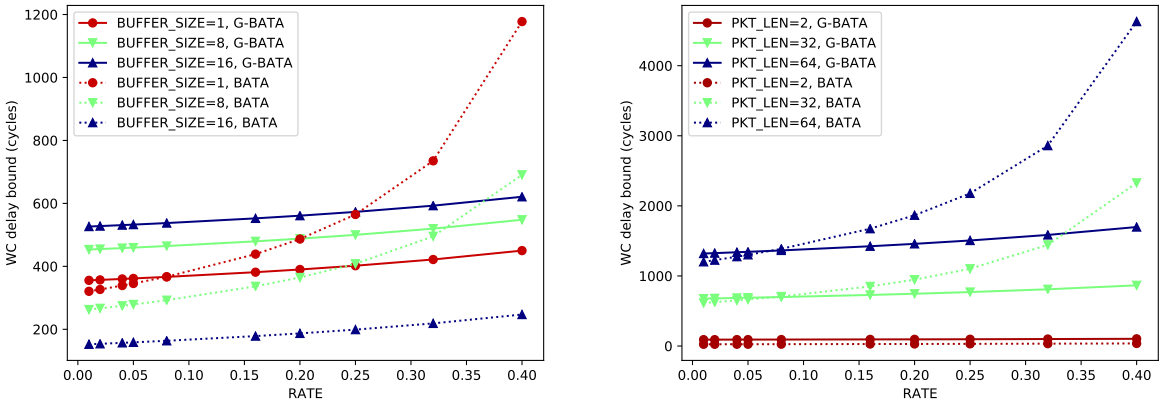



Figure 5.14 – Compared flow rate impact on end-to-end delay bounds

We now focus on the impact of the flow rate on end-to-end delay bounds (Figure 5.14). The left graph represents the evolution of delay bounds when packet length is fixed (16 flits) for different values of buffer size, and the right graph shows the evolution of delay bounds with a fixed buffer size (4 flits) and values of packet length from 2 to 64 flits. As expected, with both approaches, the end-to-end delay bound increases with the rate. What is more interesting is that delay bounds with G-BATA approach increase much less rapidly than with BATA approach for buffers of 1 and 8 flits: at a 40% flow rate, BATA gives bounds that are 26% to 162% greater than bounds given by G-BATA approach (left graph on Figure 5.14). Therefore, we can confirm one more time that BATA is more sensitive to rate variations than G-BATA.

Although there is generally no strict order between the bounds given by the two approaches, for instance for $B = 8$ flits, we can notice a trend regarding the relative position of the bounds: BATA predicts smaller bounds than G-BATA for large buffer sizes and small rates, and the trend is opposite for small buffer sizes, especially as the rate increases. When the rate of flow $\rho$ increases with all other parameters constant, the propagated burst of an arrival curve increases by $\rho \cdot T$ per node with a service curve latency of $T$. Results obtained with BATA are especially impacted by this burst propagation since the burst is propagated at the beginning of the

subpaths when computing $T_{IB}$ (Algorithm 2, Line 16). This explains why BATA-predicted bounds increase faster than graph-predicted bounds when increasing the rate.

**Key points:** Both approaches predict an increase of the end-to-end delay bound with the rate, however this increase is significantly different depending on the approach. Burst propagation at the beginning of subpaths in BATA approach leads to important bound increase when the flow rate is high. For instance, the computed bounds are up to 275% higher with BATA than with G-BATA at 40% flow rate.

### 5.6.3 Tightness Analysis

To assess the tightness of the delay bounds yielded by G-BATA, we consider herein a worst-case simulation using Noxim simulator engine [50]. We proceed as in Section 4.7.2. We run each flow configuration many times while varying the flows offsets and we consider the maximum worst-case delay over all the simulated configurations and compute the tightness ratio for each flow.

We simulate the configuration of Figure 4.6, when varying buffer sizes in 4, 8 and 16 flits, and flow rates in 8% and 32% of the total available bandwidth. We extract the worst-case end-to-end delay found by the simulator and compute the tightness ratio for each flow. The obtained results are gathered in Table 5.1.

We recall the computed tightness ratios obtained with BATA. Additionally, we computed and included the congestion indexes associated with G-BATA approach. We only displayed results for buffer sizes 4 and 16.

We notice that the lower the congestion indexes are, the greater the tightness is. Low congestion indexes mean that the contending possibilities are reduced. Hence the worst case is simpler to find and thus more likely to be achieved or approached with randomly chosen offsets. We stress out the fact that there are many possibilities for the wake up time of each flow, and that our series of simulations may not have been able to approach or achieve the worst-case for every flow.

For a buffer size of 4 flits and a flow rate of 8%, G-BATA and BATA give similar results (with a slightly better average tightness for BATA). However, for a 32% rate, G-BATA gives tighter bounds. For 16 flits of buffer size, BATA gives tighter results for both rates. However, we want to stress out that in this case, for 32% rate, we might not be able to verify that no CPQ can occur. Thus the results

| | $B = 4$ | | | | | |
|---|---|---|---|---|---|---|
| | rate = 8% | | rate = 32% | | | |
| Flow | G-BATA | BATA | G-BATA | BATA | $I_{DB}$ | $I_{IB}$ |
| 1 | 40% | 44% | 74% | 44% | 4 | 11 |
| 2 | 42% | 41% | 87% | 24% | 2 | 12 |
| 3 | 63% | 64% | 75% | 51% | 3 | 5 |
| 4 | 64% | 68% | 93% | 64% | 2 | 3 |
| 5 | 72% | 77% | 68% | 46% | 2 | 0 |
| 6 | 73% | 75% | 57% | 21% | 2 | 0 |
| 7 | 85% | 89% | 100% | 43% | 1 | 0 |
| 8 | 66% | 68% | 67% | 46% | 2 | 0 |
| 9 | 45% | 47% | 37% | 24% | 2 | 0 |
| 10 | 84% | 88% | 88% | 70% | 2 | 0 |
| 11 | 87% | 92% | 81% | 69% | 1 | 0 |
| 12 | 85% | 89% | 91% | 66% | 1 | 0 |
| avg | 67.36% | 70.11% | 76.52% | 47.41% | - | - |
| min | 40.20% | 40.61% | 36.51% | 20.77% | - | - |
| max | 87.19% | 91.71% | 99.74% | 70.46% | - | - |

| | $B = 16$ | | | | | |
|---|---|---|---|---|---|---|
| | rate = 8% | | rate = 32% | | | |
| Flow | G-BATA | BATA | G-BATA | BATA | $I_{DB}$ | $I_{IB}$ |
| 1 | 24% | 79% | 44% | 87% | 4 | 22 |
| 2 | 18% | 79% | 43% | 97% | 2 | 27 |
| 3 | 35% | 85% | 48% | 75% | 3 | 14 |
| 4 | 34% | 86% | 51% | 87% | 2 | 8 |
| 5 | 66% | 85% | 47% | 64% | 2 | 0 |
| 6 | 63% | 86% | 45% | 86% | 2 | 0 |
| 7 | 83% | 87% | 78% | 97% | 1 | 0 |
| 8 | 60% | 70% | 46% | 71% | 2 | 0 |
| 9 | 40% | 49% | 27% | 44% | 2 | 0 |
| 10 | 80% | 88% | 76% | 88% | 2 | 0 |
| 11 | 86% | 88% | 100% | 96% | 1 | 0 |
| 12 | 83% | 87% | 76% | 88% | 1 | 0 |
| avg | 56.08% | 80.76% | 56.73% | 81.60% | - | - |
| min | 18.07% | 48.94% | 26.96% | 43.83% | - | - |
| max | 86.50% | 88.32% | 100.00% | 97.34% | - | - |

Table 5.1 – Tightness Summary for both approaches, buffer size 4 flits (top) and 16 flits (bottom)

yielded by BATA should be taken with caution.

**Key points:** On the tested configuration, with 4-flit-large buffers and at 8% flow rate, both models give similar results. With the same buffer size and a higher rate (32%), G-BATA gives tighter results than BATA, showing that BATA tends to be pessimistic for high flow rates. With larger buffer sizes, BATA performs better, but when flow rates are high, BATA might not be applicable. Overall, the tightness is good. G-BATA averages at 72% when the buffer size is 4 flits and 56% for 16 flits, whereas BATA averages at 59% and 81%, respectively. For flows subject to the more complex congestion patterns, the worst-case may not have been approached as closely as for flows undergoing little to no interference, hence the derived tightness ratio is smaller. This conjecture is supported by the fact that the measured tightness is lower for flows with higher congestion indexes.

In order to determine whether BATA or G-BATA should be used, we propose a decision-making graph (Figure 5.15). Essentially, G-BATA should be used whenever the model hypothesis do not allow to use BATA, or when analysis results should be obtained as quickly as possible. In the absence of such limiting factors, one can use BATA under the assumption that CPQ cannot occur.

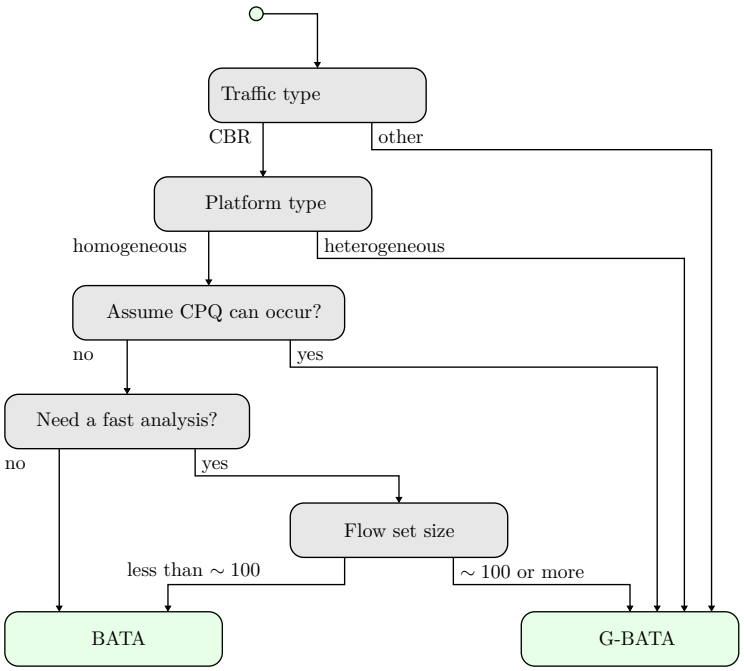



Figure 5.15 – Determining which approach should be used

## 5.7 Conclusions

To improve the scalability and the applicability of BATA approach, we proposed G-BATA. G-BATA extends the flow model to support bursty traffic and extends the network model to cover heterogeneous architectures. Furthermore, we re-engineered the indirect blocking analysis to support the bursty traffic assumption and the consecutive packet queueing scenarios that can result, using a graph structure to capture interference patterns. We consequently adapt the service curve computation method. The graph structure allows us to reduce the number of recursive calls.

Then, we study the computational aspect of our new approach. We find that G-BATA exhibits a much better scalability than BATA, with average computation times 10 to 100 times lower on 32- and 48-flow configurations. Moreover, G-BATA scales well and is able to compute all end-to-end delay bounds of 800-flow configurations in less than two hours (around 9 seconds per flow).
We then seek to estimate the complexity of flow sets by introducing and evaluating two *congestion indexes*, namely the DB index and the IB index. These two indicators aim at quantifying the congestion a flow is likely to undergo. We show that for a given number of flows, the average DB and IB indexes of a flow set are correlated with the duration of G-BATA and BATA analysis on the flow set, and therefore can give further insight on how complex the configuration is.

Afterwards, we perform a sensitivity analysis of G-BATA, and compare it with BATA. We find that G-BATA is less sensitive than BATA to flow rate. Contrarily to BATA, the bounds yielded by G-BATA increase when buffer size increases. This stems from the fact that G-BATA considers that consecutive packet queueing (CPQ) can happen and does not bound the number of packets that can stall in the network. Consequently, increasing buffer size also increases the potential CPQ, hence the end-to-end delay bound.

Finally, tightness study results on the test configuration show an average tightness ratio up to 72% for G-BATA. For flows with a high rate, G-BATA yields tighter end-to-end delay bounds than BATA. We also find that tightness ratio is higher for flows with lower congestion indexes, for both approaches.

*Aux sourdes percussions mes doigts se sont levés,*
*Nourris de rythmes fous et de nappes criardes.*
*Tantôt les cliquetis sur les touches blafardes*
*Se muent en mots, en nombre, en matière achevée.*

*
* *

Chapter 6

# Hybrid Methodology for Design Space Exploration: Simulation and Network Calculus

Contents


**6.1 Introduction . . . . . 117**
**6.2 Overview and Extended Workflow . . . . . 118**
**6.3 System Modeling: Adding a NoC Component in TTool . . 119**
6.3.1 Implementation . . . . . 119
6.3.2 Functional View . . . . . 120
6.3.3 Architecture . . . . . 124
**6.4 Verification, NoC Generation and Simulation . . . . . 124**
**6.5 Performance Evaluation . . . . . 126**
6.5.1 Example Modeling . . . . . 126
6.5.2 Analysis and Results . . . . . 128
**6.6 Conclusion . . . . . 131**


## 6.1 Introduction

In this chapter, we tackle design space exploration. As previously noticed, few of the existing methods integrate the verification of real-time constraints in the design space exploration workflow. Besides, most of the timing requirements are checked using simulation results instead of a formal approach. Therefore, our aim is to provide a methodology allowing to verify the compliance with the deadlines early in the design process.

We present an extended workflow compatible with the Y-chart approach, in which real-time constraints can be verified during the design process. We present the high-level view of our approach and the updated workflow in Section 6.2.
Then, we implement this approach. To this end, we use an existing toolkit for design space exploration, TTool [9, 90, 91], to which we add certain features, and a tool for worst-case timing analysis of networks called WoPANets [10, 11], in which we integrate our model for NoC analysis G-BATA.
We add a NoC model in TTool's architecture description tool to allow simulation of NoC-based systems. We detail the system modeling aspect in Section 6.3. In Section 6.4, we present the NoC generation step and the verification capabilities of the approach. In Section 6.5, we demonstrate the use of the approach and assess its performance in terms of computation time and scalability. Finally, we conclude on our approach (Section 6.6).

## 6.2 Overview and Extended Workflow

The typical workflow for NoC-based system design using our methodology is shown on Figure 6.1. The gray boxes refer to steps of the regular Y-chart approach. The colored ones refer to our extensions: the green one refer to system modeling, while the blue one denotes steps related to timing analysis.

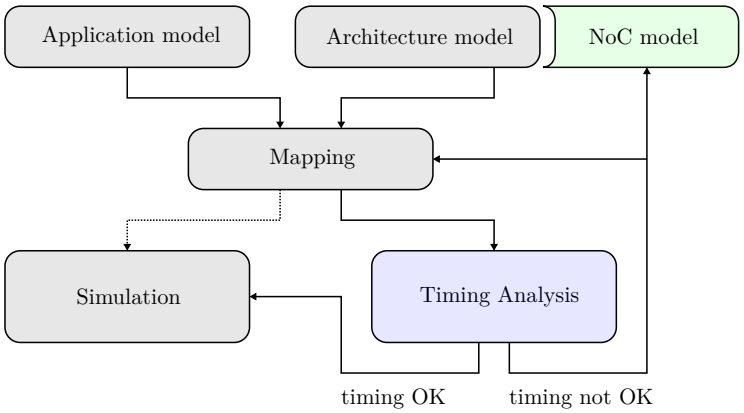



Figure 6.1 – Workflow of the hybrid approach

The functional and architectural description of the system constitute the first step, and they are done independently. The functional structure defines the different tasks of the system, their relationships between each other, and their behavior. The architecture model consists of processing elements, interconnects and communication media that will execute the functional model. The mapping step assigns

each task of the functional description to an architecture module.
Once a mapping is generated, the timing analysis step can be performed, in order to discard infeasible mappings. If the mapping complies with the timing requirements, simulation will allow to get additional insight on the system behavior and further refine the design. Otherwise, either the mapping or the architecture should be changed, and checked again, until a configuration compatible with the timing constraints is reached.
We expect that performing the timing analysis early in the design flow will speed up the design space exploration process, because it avoids to simulate configurations that do not meet the timing requirements. In the next section, we detail the practical implementation of our approach.

## 6.3 System Modeling: Adding a NoC Component in TTool

### 6.3.1 Implementation

We base our implementation on a set of existing tools, to which we add certain functionalities: (i) TTool [9], a toolkit for design space exploration, and its profile for system-level modeling, DIPLODOCUS [90]; (ii) WoPANets [10, 11], a software for worst case performance analysis of networks, and more specifically a plugin we added for NoC worst case analysis, implementing our approach G-BATA.
As TTool already complies with the Y-chart approach, we use it to describe the functional behavior, the architecture and the mapping. At this point, the one or several NoC component(s) used in the architecture description phase are considered to be black boxes. We also use TTool for the simulation phase.
The formal real-time analysis relies on WoPANets, and it can be done once a mapping is generated. We will detail it later on, in Section 6.4.
If the timing analysis validates the mapping, the designer can proceed to the simulation to gain more insight on the system. This step uses the TTool simulator engine. We add the NoC-removal step before the compilation step to generate all the components and tasks that constitute the NoC. Then, we use TTool to generate, compile and run the simulation code. Additional metrics and informations obtained using simulation can be used to propose modifications of the architecture or mapping.
The implemented approach workflow is presented on Figure 6.2. Green boxes still refer to our extensions in TTool, and blue ones to the timing analysis step.

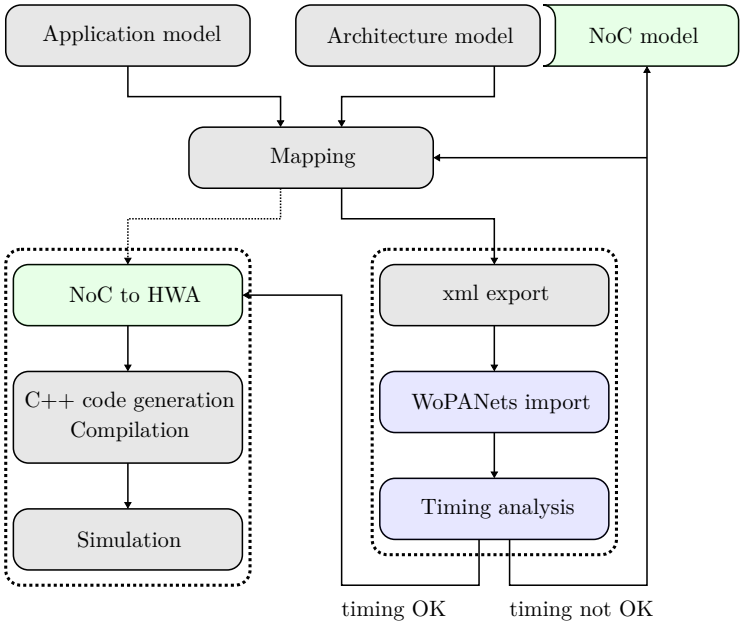



Figure 6.2 – Extended workflow with implementation details

To include a NoC component in TTool, we also follow the Y-chart approach and separate the NoC functional description from its architecture. The next sections detail the NoC model we implemented.

### 6.3.2 Functional View

We distinct two basic elements that constitute a NoC: the router and the network interface. Routers are interconnected to one another to create the NoC topology, and they handle packets in a flit-per-flit manner. Network interfaces allow to link flit-unaware senders and receivers, that only deal with packets, to routers of the NoC. The input network interface is in charge of injecting packets into the NoC and perform arbitration between different senders, while the output network interface will consume incoming flits and notify the appropriate receiver when a complete packet is available. This way, the final NoC component can be connected to other components available in TTool without any changes. We synthesize the general functional architecture on Figure 6.3.
The router model is an input-buffered router with VC support. Such a router must be able to:

- dispatch incoming traffic to appropriate VC queues;
- make the routing decision for each packet;
- arbitrate between different inputs;

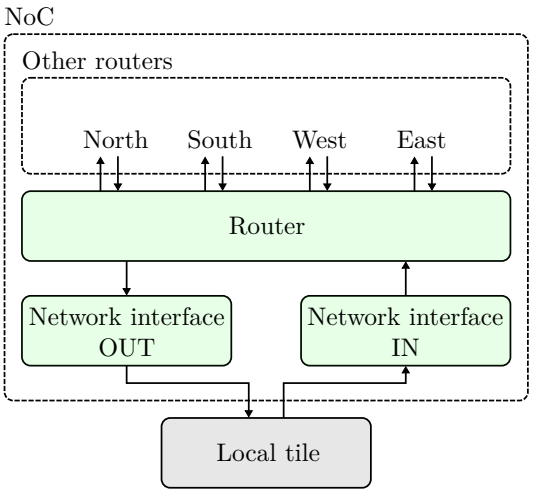



Figure 6.3 – NoC functional view

- arbitrate between different VCs;

To perform these actions, we propose the functional structure displayed on Figure 6.4, for a simplified router with two VCs, two inputs and two outputs.

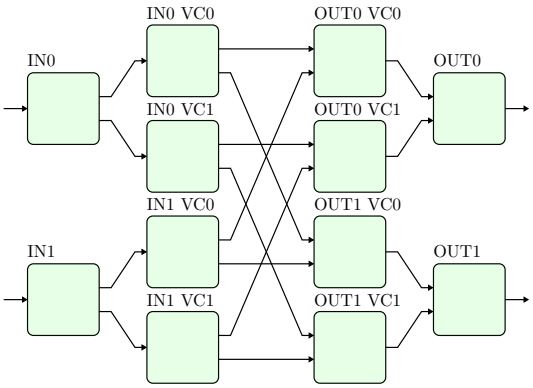



Figure 6.4 – Functional structure of a router

This view consists of several stages in charge of a particular aspect of flit forwarding. Such an organization simplifies each stage and allows for more flexibility in the design. It also makes modification of arbitration policies easier, as most of them require alteration to only one block. Note that each block has a behavior specified using *activity diagrams*. We do not show them here and summarize the behavior of the blocks instead.

First, the `IN` functions receive incoming traffic and dispatch it to the appropriate VC. The `INVC` functions compute the routing decision and request for the appropriate output port. Note that, at this stage, we perform XY routing. We haven't implemented other routing policies yet and leave this aspect as an improvement

perspective. However, due to the modularity of the functional architecture, implementing a different routing algorithm can be easily done modifying only the `IN_VC` function.
The next stage is the `OUT_VC` function. It receives output requests from different inputs of one VC and arbitrates in a FCFS manner. It sends output requests to the `OUT` module that performs the multiplexing of traffic from different VCs requesting the same output. We assume Fixed-Priority arbitration between VCs, as it allows to provide different QoS to different traffic types. We leave other arbitration policies as a future work and note that our architecture makes it possible to alter the VC arbitration policy by only modifying the `OUT` function.

We now detail the way the different functional modules interact with each other, in relationship to the expected behavior of a router. TTool offers two types of ports that we will use: channels and events. Channels, on one hand, correspond to data transfers between tasks (data being read/written). They have to be mapped on a medium and they consume bandwidth during the transfer. In TTool, however, the value of the data is abstracted, and only its size is relevant. Events, on the other hand, represent the control aspect of the application. They do not consume any bandwidth and do not need to be mapped.
A router performs reads and writes when it handles a flit, as shown on Figure 6.5. In our model, there is a first read/write operation at the `IN` function when an incoming flit is dispatched to the appropriate queue. Then, the flit is read from the buffer by the `IN_VC` module, and will be later written by the `OUT` module to the input of the downstream router. These stages also use events for feedback control and to transmit information about the current packet to downstream modules, as detailed hereafter.
We use events for the control functions of the router, as illustrated on Figure 6.6. This is especially the case for the modules `IN_VC` and `OUT_VC`, that use events to notify downstream modules that a flit is available and transmit information about the packet (length, destination, VC, channel ID). We also use events to implement a feedback control mechanism and arbitration (input selection and VC selection). Initially, `IN_VC` modules send as many feedback control credits as the number of flits their buffer can hold. Afterwards, all modules consume one credit before forwarding a flit downstream. Once the flit is forwarded, they generate a credit for the upstream module.

Within routers, each event corresponds to either a flit or a feedback control token.

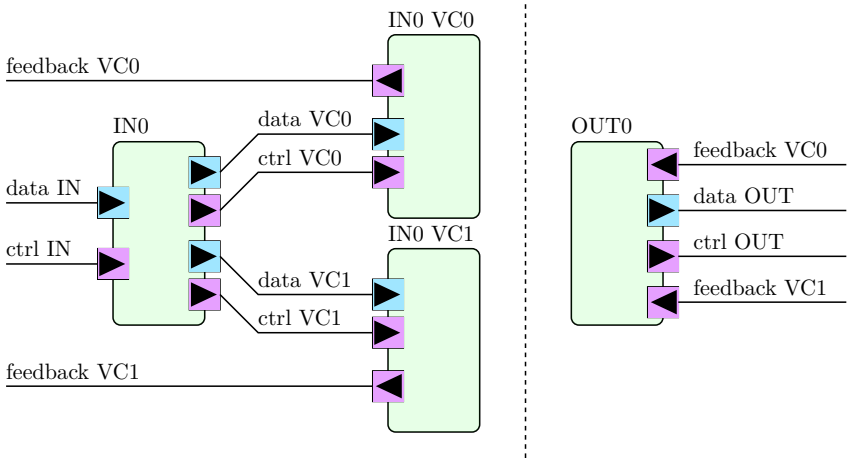



Figure 6.5 – Read and Write operations in the router model. Blue ports correspond to channels, purple ports are events.

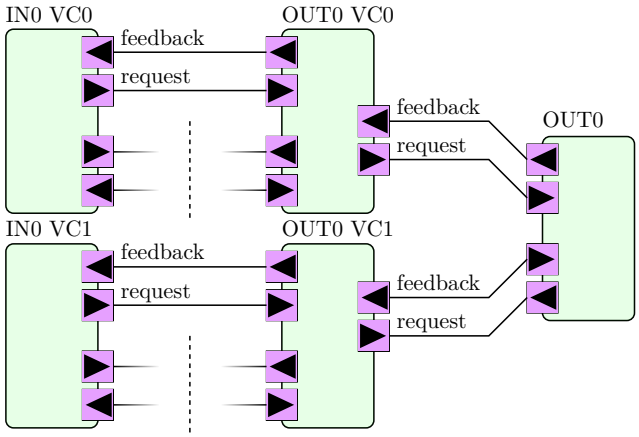



Figure 6.6 – Control events in the router model

Outside of the NoC, each event corresponds to a packet. One of the roles of the network interface modules is to perform the appropriate conversion between packet-view and flit-view.

The input network interface receives one event for each packet. This event contains information about the packet length, its destination, the VC it is mapped on and a channel identification corresponding to the identity of the flow. This last information is used to separate flows with the same destination core but that are handled by different tasks. The input network interface sends one event before each flit. This event contains all the information of the packet, and a field `eop` (end of packet) to indicate whether the corresponding flit is the last one.

Conversely, the output network interface receives packets flit after flit. Once it has read a whole packet, it notifies the receiving task with an event containing all the

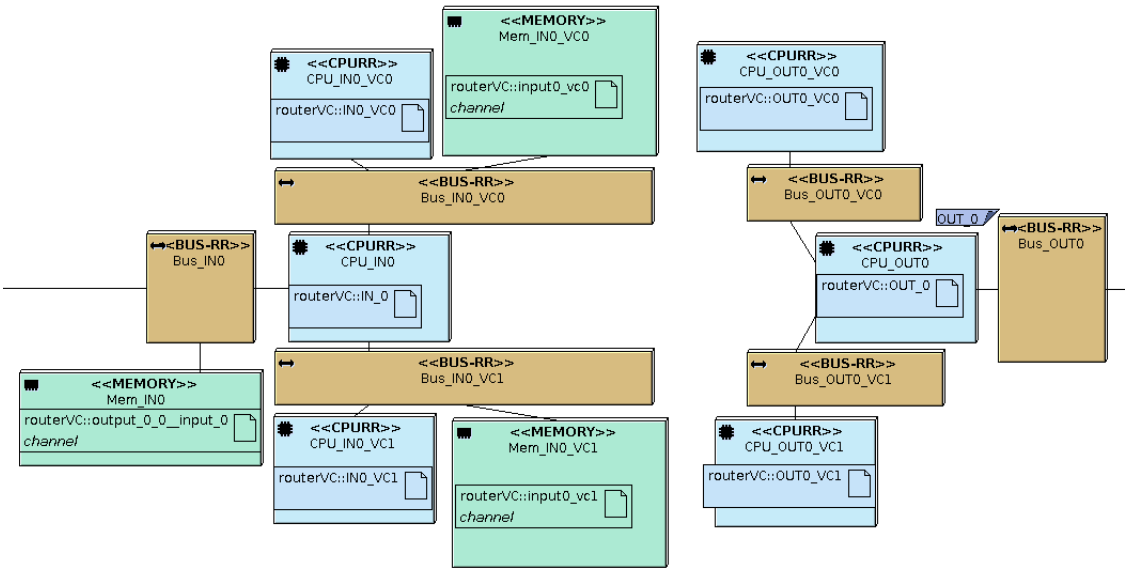



Figure 6.7 – Architectural view of the presented router (with task mapping)

information of the packet (length, destination, VC and channel ID).

### 6.3.3 Architecture

The architecture we designed for the NoC components must ensure each task of the routers and network interfaces can run without concurrency. A partial view of the architecture and mapping is shown on Figure 6.7. For a router with two VCs, there are as many such parts as there are input/output pairs. We map each task on one processing element, thus the architectural view has a structure very similar to the functional diagram. We use memory for the input buffers, the network interface buffers, and for the `IN` module (that must be able to read and write one flit).
Notice that the architecture model is an abstraction of the real architecture at a higher level, hence the processing elements used are CPUs here, but it does not matter much. In a real router, these would be implemented on lighter, dedicated hardware.

## 6.4 Verification, NoC Generation and Simulation

The first verification is the worst-case timing analysis based on G-BATA (Figure 6.2). It is done after the mapping step, once the model is syntactically checked, in order to ensure the mapping is compliant with the real-time constraints. In practice, the model is exported to a XML file that can be read and processed by

WoPANets. Afterwards, the worst-case timing analysis is performed to determine whether or not the mapping can satisfy all real-time constraints. At this point, unfeasible mappings are discarded. For now, our integration work does not allow WoPANets to propose alternative mappings or modifications of the architecture when the system does not satisfy real-time constraints. However, this could be implemented and used in the near future.
The main challenge of this part is to compute the flows arrival curves from the functional description of the system. There is no explicit description of the flows *per se* in the functional model, therefore, we consider each channel corresponds to a flow. The length of the packets can be extracted from the XML file by looking at the `Write` instructions on the corresponding channel. Henceforth, the computation of the maximum packet length is straightforward.
However, timing characteristics (period or maximum inter-packet time, and jitter) of the flow model we use may not be explicitly specified, and hard to derive from the activity diagram of the task generating the flow.
To cope with this difficulty, we require that attributes specifying at least the period and packet length of each flow are present in the functional description; either in the task generating the flow, or in one or several control tasks handling the flow generation. We may develop tools to drop this requirement in the future, but for now, this simplifies the import of the XML file in WoPANets.

The second verification technique is based on transactional simulation. Transactional simulation allows to get a detailed, cycle-accurate behavior of the system running on the target platform. While it does not provide guarantees on the real-time constraints as such, it can be used to further refine some design choices, beyond the compliance with deadlines.
To handle the NoC component, we add features to the TTool entity in charge of generating the simulation code. At the NoC removal step, it generates all tasks of the different components of the NoC (routers tasks, network interfaces) and the corresponding architecture and mapping of the NoC component, according to the pattern of Figure 6.7.
Then, it generates C++ code for the simulator engine. From this point onwards, the simulator can be run from TTool graphical interface or from a terminal.
The main challenge at this point is to interpret simulation traces, that can be very large due to the number of devices and tasks a NoC represents. For instance, a $4 \times 4$ square NoC with 2 VCs represents 448 tasks and as many devices to run them.

## 6.5 Performance Evaluation

In this section, we detail the modeling, analysis and simulation of a small example to demonstrate the capabilities of our approach (Section 6.5.1). Then, in Section 6.5.2, we use the results to evaluate the computational cost gain from integrating the timing analysis step in the workflow.
We model a target configuration, shown on Figure 6.8. It consists of three flows on a small $2 \times 2$ tiled platform with a NoC.

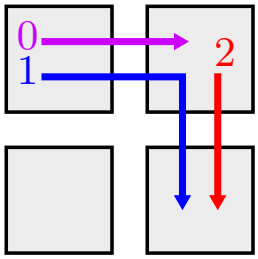



Figure 6.8 – Example configuration

### 6.5.1 Example Modeling

Figure 6.9 shows the functional view of one flow, consisting in a TX task, an RX task and a control task. The displayed model is replicated two times to obtain three similar schemes in total, corresponding to three flows. We present the activity diagrams of the `ctrl`, `src` and `dst` tasks on Figure 6.10. The `src` tasks generate traffic, *i.e.* they send one packet when instructed to do so by an event. The `dst` task consume traffic, *i.e.* they wait for a packet to be read when instructed to do so by an event. Events are generated by three "control tasks" (`ctrl`) so that each flow transmits exactly two packets with given period and offset.
Notice that `src` tasks write 3 flits while `dst` tasks read 4 flits. This is due to a choice we made to represent the additional flit needed for the header of the packet to be sent.
We map the functions according to figure 6.11. Each CPU has enough cores for each task to run without concurrency. Note that the control tasks are mapped on a dedicated CPU so that they do not interfere with the `src` and `dst` tasks. We set the buffer size to 2 flits.
To get the NoC end-to-end latency of a flow, we focus on two particular instants: (i) the time at which the first flit of the packet is read by the `IN` module of the first router local input, called *injection time* and denoted $t_i$; and (ii) the time at

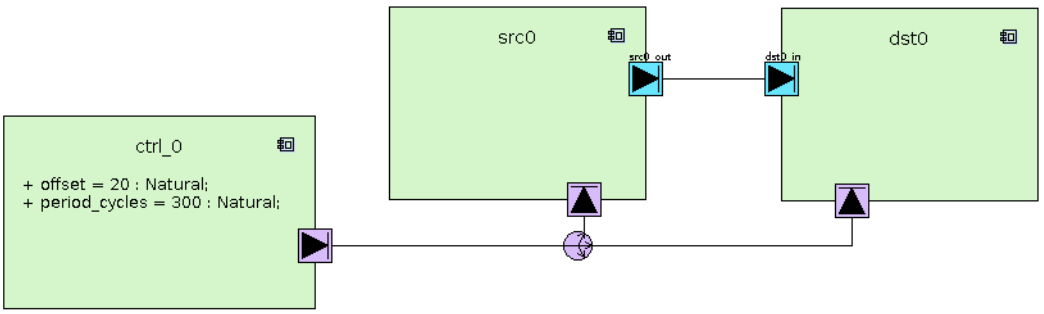
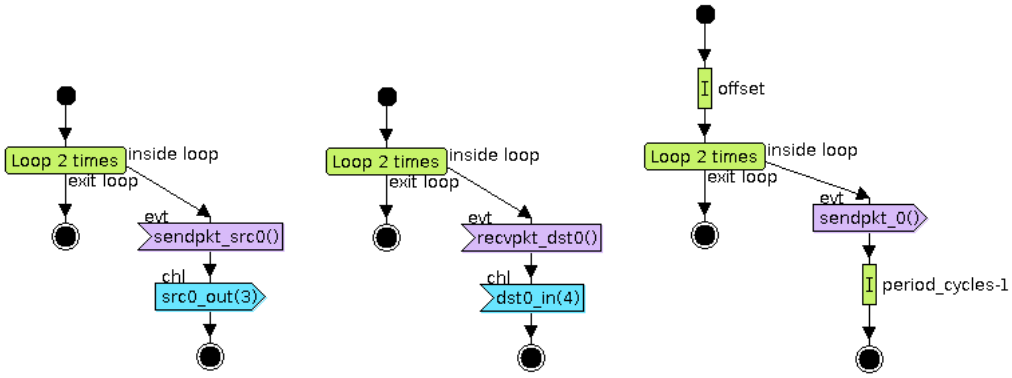



Figure 6.9 – Functional view of the example




(a) TX task (b) RX task (c) Control task

Figure 6.10 – Activity diagrams

which the last flit of the packet is written by the `OUT` module of the last router local output, called *ejection time* and denoted $t_e$. The remaining part of the delay is not taken into account by the timing analysis model since it is due to the traversal of network interfaces. The end-to-end delay can thus be expressed as:

$$D_m = t_e - t_i$$

However, when two packets are originating from the same tile at the same time, one will be delayed by the other before the injection time. Hence, the delay $D_m$ measured in this case may be lower than the actual delay experienced by the blocked packet. To take this particularity into account, we first get the time when the packet requests the use of the network interface, denoted $t_x$, and referred to as *request time* hereafter. This time corresponds to an event sent by the source task after it performed the "Write" of the packet to the NoC, so it is relatively easy to extract. Then, we measure the time taken by the first flit to cross the network interface

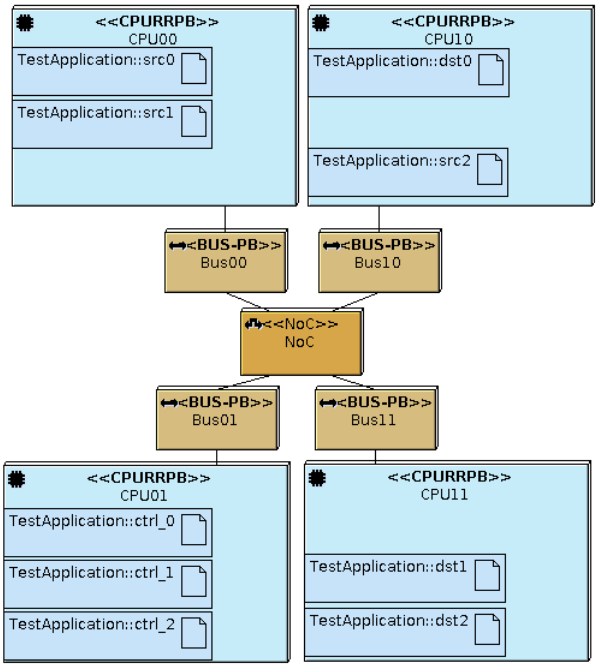



Figure 6.11 – Mapping example

when there is no congestion, denoted $\Delta_x$. We finally obtain the measured delay with the following expression:

$$D'_m = t_e - (t_x + \Delta_x)$$

Notice that for flows not experiencing a delay of their injection time, we have $D_m = D'_m$.

### 6.5.2 Analysis and Results

With the mapping shown of Figure 6.11, we proceed to the timing analysis step. First, we determine the characteristics of the nodes of our router model using examples with one flow and no congestion. We find that the service curve for one router is:

$$\beta(t) = \frac{1}{13}(t-5)^+$$

We stress out that these characteristics depend on the model of the router we designed. We can vary them by adapting the clock frequency of the NoC relatively to the clock frequency of the platform CPUs. Besides, it is possible to propose another router model for the NoC to obtain different characteristics, but this is a

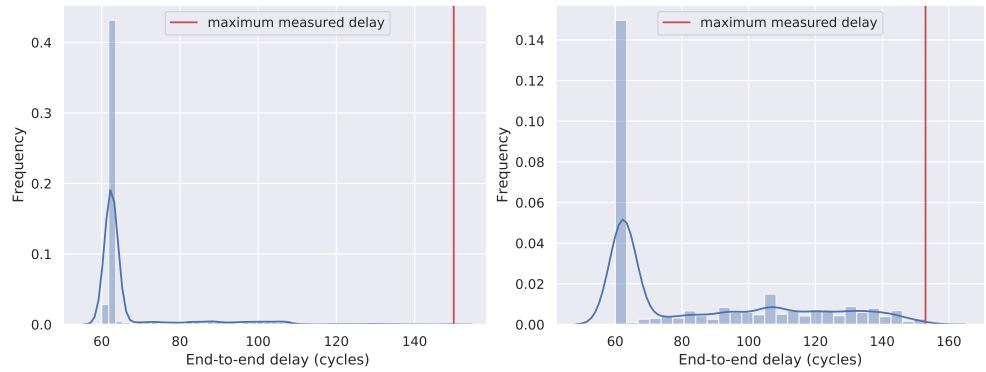
future development perspective. We run G-BATA on the configuration and derive the end-to-end delay bounds.
Afterwards, we proceed to the simulation, first setting all offsets at zero. We extract injection and ejection times from simulation traces for the two first packets (Table 6.1), compute the corresponding delays and compare them to the worst-case bound.

| | Flow 0 | | Flow 1 | | Flow 2 | |
|---|---|---|---|---|---|---|
| $t_x$ (cycles) | 8 | 208 | 11 | 211 | 8 | 208 |
| $t_i$ (cycles) | 13 | 213 | 43 | 243 | 13 | 213 |
| $t_e$ (cycles) | 75 | 275 | 139 | 339 | 75 | 275 |
| $D_m$ (cycles) | 62 | 62 | 96 | 96 | 62 | 62 |
| $D'_m$ (cycles) | 62 | 62 | 123 | 123 | 62 | 62 |
| Delay bound (cycles) | 199 | 199 | 455 | 455 | 288 | 288 |

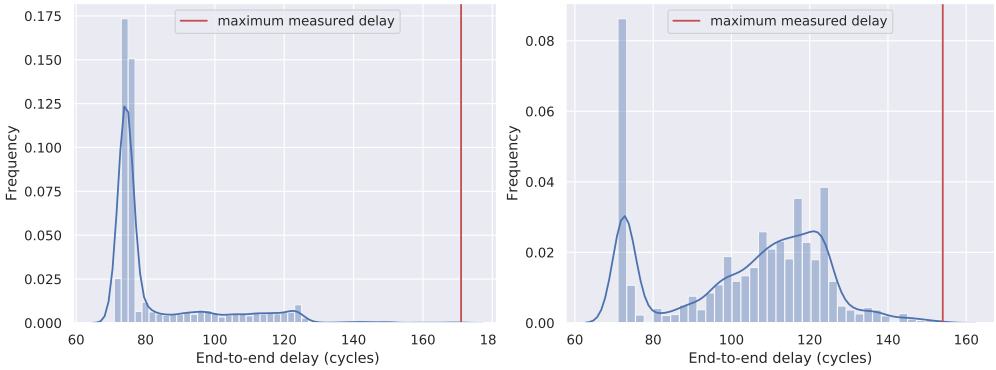
Table 6.1 – Request times, injection times, ejection times and delays for all flows

We first notice that flow 1 delay bound is greater than its deadline (455 cycles *vs* 300 cycles). The analysis of the traces also reveals that flow 0 and 2 do not undergo any congestion in the simulated scenario. This is further confirmed by three elements. First, as the offsets are zero, flow 2, injected from tile (1,0), is initially the only one requesting the use of the link from (1,0) to (1,1), and therefore can proceed to its destination without congestion.
Second, packets of flow 0 and 1, released at the same time, compete for the use of the link between (0,0) and (1,0). One of them will not experience any delay, while the other will wait. As both flows originate from the same tile, this blocking scenario occurs before the injection time. Since $D_m$ and $D'_m$ are the same for the packets of flow 2, we infer that no congestion occurs for this packet right before injection time.
Third, we compute the base latency for flows 0 and 2, and find it is the exact same value as the measured delay, which denotes a transmission without congestion.

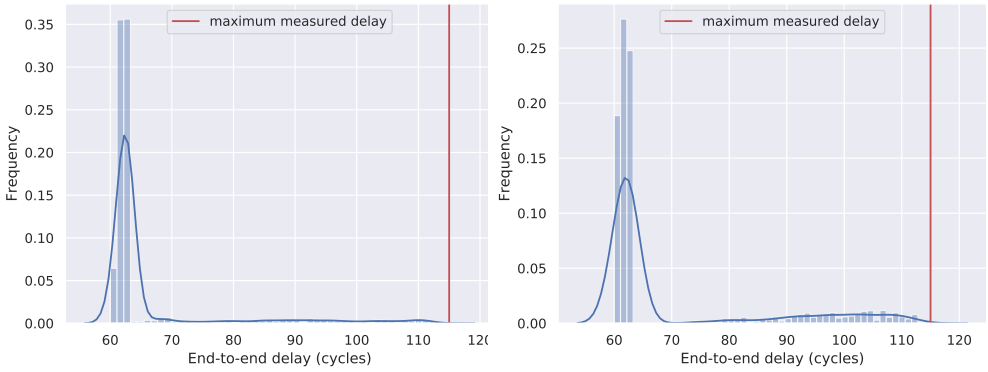
To vary the transmission scenarios, we perform a series of additional simulations with random offsets. To that end, we randomly chose an offset for each flow, according to a uniform distribution, then simulate the configuration so that 10 packets of each flow are transmitted. We repeat this process 3000 times with offsets between 0 and 299, and 3000 times with an offset between 0 and 50. We plot the distribution of the delays for each flow on Figure 6.12.
On the machine used, the NoC removal, code generation and compilation were done




(a) Flow 0




(b) Flow 1




(c) Flow 2

Figure 6.12 – Distribution of end-to-end delays with random 0- to 299-cycle offsets (left) and 0- to 50-cycle offsets (right)

in less than two minutes. The 3000 simulations and associated trace processing took approximately 44 minutes, and were not able to exhibit a deadline miss for flow 1.

The timing analysis of the system took 602 $\mu$s.
We can make two observations about this experiment:

- If the flow offsets are known, controlled or constrained in a certain way, simulation may be used to refine the timing analysis results, identify blocking scenarios that actually happen and help mitigate them. We exhibit a case of controlled offsets where simulation is able to refine the bound given by G-BATA. Moreover, simulation shows that most delays are distributed around the minimal latency. It also shows that blocking scenarios often impact flow 1 for small offsets (left graph of Figure 6.12b).
- However, in the case where flow offsets are unknown, for instance if flows are not synchronized, even this simple configuration becomes complicated to analyze on the basis of simulation, because all three flows injection times may impact the blocking scenarios. In that case, relying on the analytical approach avoids costly simulations.

We conclude that the benefits of the hybrid approach are already visible on a simple example with a small number of flows. Therefore, we can expect even more benefits on realistic case studies. Simultaneously, simulation results provide more insight on transmission scenarios and can help refine the results from the timing analysis.

## 6.6 Conclusion

We presented a general hybrid approach, using simulation and Network Calculus, for design space exploration. The workflow we propose complies with the Y-chart approach and integrates worst-case timing analysis in the design process. To the best of our knowledge, this has not been done before.
Then, we implemented our methodology on the base of existing tools, WoPANets and TTool. To this end, we added a NoC component in TTool. It may be used in architectural descriptions as a black box for the timing analysis, and turned into a functional NoC for the transactional simulation. We interfaced TTool with WoPANets via an XML export/import, and performed simulation trace filtering and processing to derive information such as end-to-end latencies of flows from the trace data. We developed the tools to be as modular as possible; as such, our work can be used as a base for various developments.

In the next chapter, we will validate our methodology by applying it to the study of an autonomous vehicle control application. We will show that our implementation of the hybrid design space exploration approach can successfully model a complex

application running on a manycore platform, validate the mapping of the tasks, and gain more detailed insight on the system behavior once the timing constraints are satisfied.

*Bien assis, le dos droit, et la tête*
*Évoquant constamment l'élégance,*
*Nous maintenons l'allure, et la bête*
*Équine se déplace en cadence.*

*
* *

Chapter 7

# Practical Applications

## Contents


**7.1 Experiments on TILE-Gx8036 . . . . . 134**

7.1.1 Platform Characteristics . . . . . 134

7.1.2 Traffic Generation . . . . . 134

7.1.3 Latency Measurements . . . . . 136

7.1.4 Results and Discussion . . . . . 137

**7.2 Control of an Autonomous Vehicle: Timing Analysis and Comparative Study . . . . . 138**

**7.3 Control of an Autonomous Vehicle: Modeling and DSE . . 143**

7.3.1 Functional Description . . . . . 144

7.3.2 Architecture Modeling . . . . . 145

7.3.3 Simulation . . . . . 145

**7.4 Results and Conclusion . . . . . 146**


In this chapter, we validate our methodology using a real platform, as well as a realistic case study. First, we present the results of our experiments on a Tilera TILE-Gx8036 36-core chip in Section 7.1. These experiments aim at proving that our model can be practically used on a real platform, and confronting the predicted bounds to physically measured delays. We then confront our methodology to a realistic case study, the control application of an autonomous vehicle, running on a $4 \times 4$ manycore chip. In Section 7.2, we first perform the timing analysis and compare our results with a state-of-the-art approach based on scheduling theory and detailed in [42]. Section 7.3 presents the modeling phase using our hybrid approach and the simulation results based on the same case study. Finally, Section 7.4 concludes the chapter.

## 7.1 Experiments on TILE-Gx8036

### 7.1.1 Platform Characteristics

To gain deeper insights into our approach G-BATA, we performed experiments on a manycore chip. The idea is to generate traffic on the NoC of a TILE-Gx8036 chip according to a known configuration, measure the latency for a sample of packets, and confront the theoretical bound derived using G-BATA to actual latency measurements.
We used a Tilera TILE-Gx8036. It is a 36-core chip with a NoC that has several subnetworks, all decoupled from one another. We will use the User Dynamic Network (UDN). Other subnetworks include the I/O Dynamic Network (IDN) and those used by the memory system to handle memory requests (QDN, RDN, SDN). Thanks to this, the data flows we monitor in our experiments will not suffer from memory-related interference.
The UDN has 3-flit-deep buffers and no VCs. There is a centralized arbiter for each router, meaning two flows crossing the same router without sharing any inputs or outputs may cause a one-cycle interference to one another. This will be taken into account in our model by adding this extra-cycle latency to the technological latency whenever needed.

### 7.1.2 Traffic Generation

To generate traffic, we define two process types, TX and RX. TX processes generate packets and send them on the NoC, while RX processes consume packets at the destination core.
To assign these processes to the appropriate cores and generate the defined traffic, we follow the steps illustrated on Figure 7.1. We use a configuration file containing all the traffic pattern information. We load the file onto the platform before running the code. This file is then read and its information is stored in a shared memory zone.
Then, the application forks as many time as there are TX and RX processes, and each process assigns itself to the appropriate core according to the information loaded from the configuration file.
We want the behavior of the application to be as predictable as possible. Therefore, when we generate and receive traffic, we want to make sure that nothing will interrupt the execution of the corresponding processes. Especially, we don't want any interference from the operating system. Thus, each process switches to "dataplane

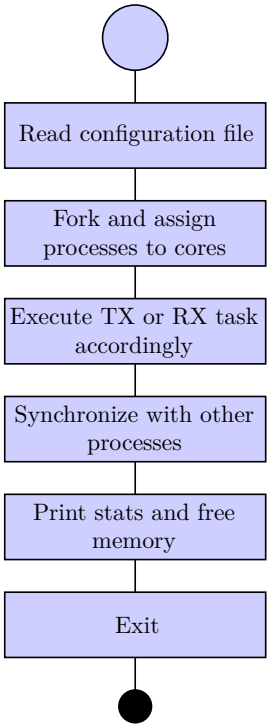



Figure 7.1 – Main application algorithmic view

mode" once it is assigned to the appropriate core. This way, it will not suffer from OS interruptions and will be granted the exclusive use of the tile resources.
We add a synchronization step between each RX/TX pair to ensure the RX process is ready to consume packets before the TX process starts transmitting.
Moreover, to test our model, we want to maximize the possibility of interference between the flows of our configuration. Therefore, we add a synchronization barrier between TX processes to make sure that they will not start transmitting packets before they are all ready to do so.
The behavior of the TX and RX processes is shown on Figure 7.2. The green background boxes correspond to the code executed in dataplane mode. The TX process first waits until its associated RX process is ready to receive packets. Then, it synchronizes with the other TX processes so that all of them start to transmit packets simultaneously. After a warm-up time, it measures the sending time of each packet before executing the send instruction. Once the appropriate number of measurements has been made, it keeps sending packets until the end. Finally, the process switches out of dataplane mode to print the sending times of the packets.

The RX process notifies its TX process when it is ready to receive, and consumes the

exact number of packets corresponding to the warm-up period. Then, it measures the receive time of each packet after the receive instruction is executed. It keeps consuming incoming packets until the end, and switches out of dataplane mode to print the receive times of the packets.

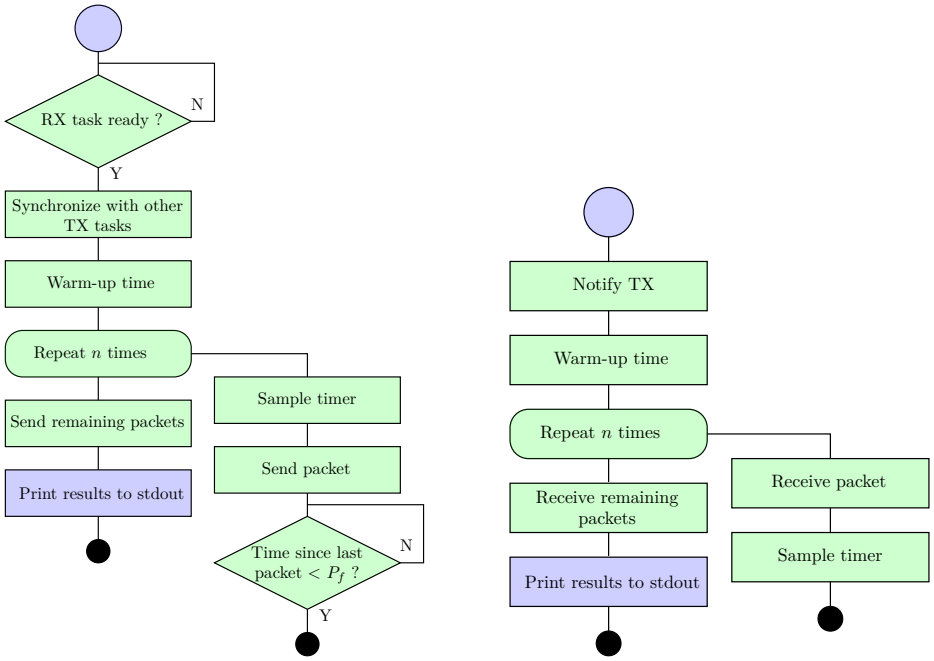



Figure 7.2 – TX and RX processes

### 7.1.3 Latency Measurements

To measure packet end-to-end latency for each flow, we proceed as follows. Each TX (or RX) process samples the cycle counter right before sending a packet on the UDN (or right after receiving a packet from the UDN). It stores the value in an array at the position corresponding to the packet number. We sample only a certain number of packets for each flow, allow a warm-up period before measuring latency, and ensure all flows continue transmitting for a while after all the measurements have been made for all flows. Consequently, there is always a possibility of interference when we measure an end-to-end latency.

We print the measured values to the standard output. Since the `printf()` function is a system call, we have to switch out dataplane mode first. We do this after the flow has finished transmitting to avoid interference with the current process. After the execution, we process these values to get the measured end-to-end latency for each packet of each flow.

However, the measured latency includes the time needed by the application to access the UDN, at TX and RX ends. This latency is not taken into account in our model. To determine it, we measure end-to-end latency on a 15-flow configuration where the UDN latency is known (no congestion). We ran the experiment 100 times, and we found that the minimal UDN access latency is a piecewise affine function of the packet length, and that 99.99% of the packets are at most 5 cycles above the minimal UDN access latency.[1]
Therefore, we consider the 99.99%-accurate measured bound, *i.e.* the latency such that 99.99% of the flows have a latency below this value.

### 7.1.4 Results and Discussion

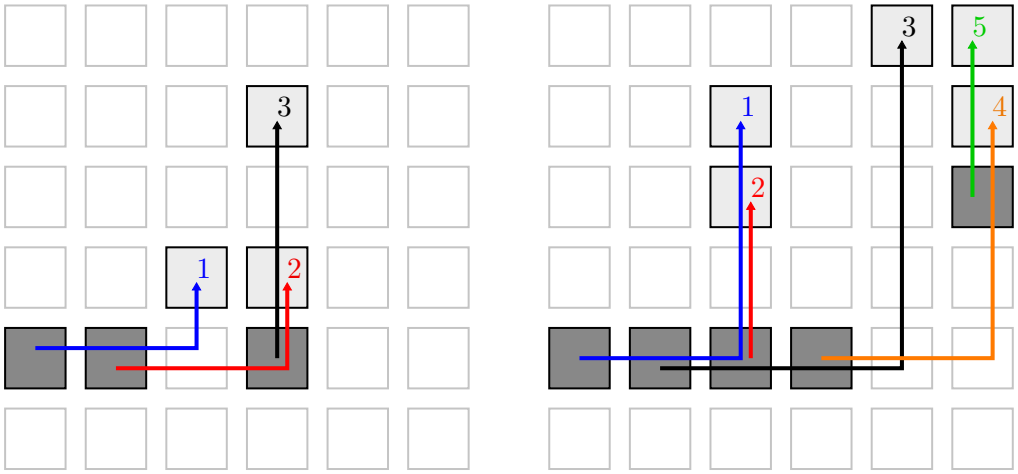



Figure 7.3 – Experiment configurations 1 (left) and 2 (right)

We test two configurations, shown in Figure 7.3. The first configuration is a simple one with only 3 flows, with a period of 200 cycles and packets of 8 flits. In this configuration, flow 1 can undergo indirect blocking from flow 3. The second one has 5 flows. Flow 1 can experience indirect blocking from flow 4 via flow 3. Flow 3 can experience indirect blocking from flow 2 via flow 1. All flows are 100-cycle-periodic and their packets are 8-flit-long.
For each flow, we compute the theoretical bound on end-to-end latency and the 99.99%-accurate measured bound. The derived results are in Figure 7.4.
We notice that for the simple configuration of 3 flows, our delay bounds are tight, in comparison to the measured ones. For instance, the tightness bound for flow 1 is 97.6%. However, for the more complex configuration (5 flows), our analytical delay

[1] We were not able to get a deterministic value for this additional latency, hence the 99.99% bound.

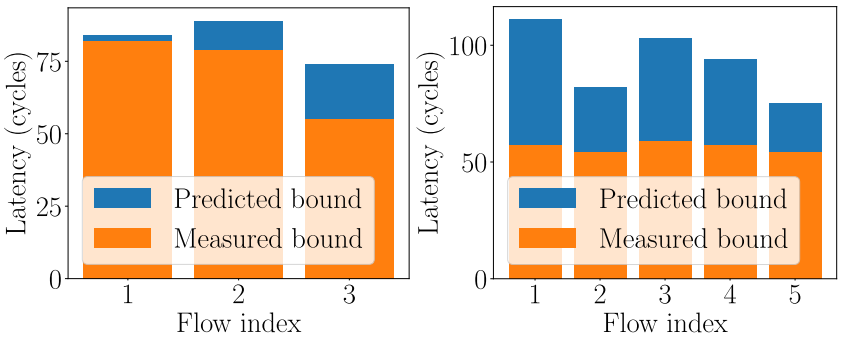



Figure 7.4 – Experiment results for configuration 1 (left) and 2 (right)

bounds are less tight, especially for flows 1 and 3, that are subject to more complex indirect blocking. This fact is mainly due to the difficulty of catching the theoretical worst-case scenario, which requires all interfering flows to be synchronized in an unfavorable way along the shared paths. Although this may seem counter-intuitive, the more flows are involved in one congestion pattern, the more difficult it will be to reach a proper synchronization between the interfering flows at least once during the experiments.

## 7.2 Control of an Autonomous Vehicle: Timing Analysis and Comparative Study

This case study was presented in [14] and used in [42]. The application controls an autonomous vehicle. It features several tasks in charge of processing data from the sensors, managing the obstacle data base, controlling the actuators and stabilizing the vehicle. Various data flows are exchanged between these tasks. Further description of the application can be found in [14], but we recall the details in Appendix C, Tables C.1 and C.2. We took the same 33 tasks mapped on a $4 \times 4$ 2D-mesh NoC, and the same mapping of the 38 data flows between tasks, routed in a XY fashion.

The parameters used are the following:

- The duration of a cycle is 0.5 ns;
- All routers have a technological latency of 3 cycles;
- The link capacity is one flit per cycle;
- Flows' priority assignment follows a rate monotonic policy;
- Each router supports 4 Virtual Channels with no priority-sharing and no VC-sharing, *i.e.*, one flow per VC;
- To compare our results to the ones in [42], we performed the analysis for

| | $B = 2$ | $B = 100$ | $B = \infty$ |
|---|---|---|---|
| Average tightness | 64% | 67% | 71% |
| Average tightness difference | +0.07% | +0.08% | -0.03% |
| Maximum tightness difference | +3.70% | +3.49% | +0.01% |
| Minimum tightness difference | -0.10% | -0.10% | -0.10% |

Table 7.1 – Average tightness and tightness differences for various buffer sizes

different buffer sizes (2, 100 and 1000000 flits, the latest being large enough to assume buffer size is infinite).

All flows have a different priority. As they are mapped to VCs in such a way that at each router, all VCs are non-shared, there is no indirect blocking. Thus, we expect BATA and G-BATA to give the exact same results for the worst-case delay bounds, which we checked was the case. We then plotted comparative graphs on Figure 7.5 and computed the average tightness of our approach (Table 7.1), using results from simulations performed by Nikolić et al. [42]. All the computed tightness ratios are shown on Table C.3, in Appendix C. The average tightness ratio for G-BATA approach with buffer size 2, 100 and infinite are 64%, 67% and 71% respectively.

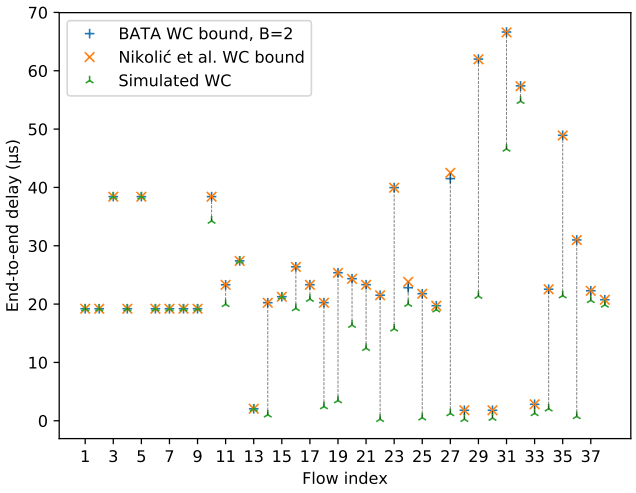
We first notice that our approach gives similar results to [42]. To further quantify the similarity of the results, we subtracted the tightness ratio obtained by the two approaches on each bound to obtain what we call "tightness difference", denoted $\Delta\tau$. For a given flow:

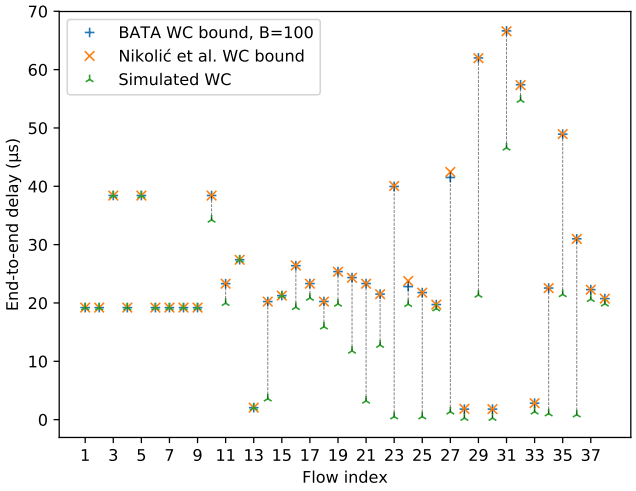
$$\Delta\tau = \tau_{\text{G-BATA}} - \tau_{\text{ST}} \quad ,$$

where $\tau_{G-BATA}$ is the tightness ratio of the bound yielded by G-BATA, and $\tau_{ST}$ is the tightness ratio of the bound yielded by the method of [42]. The tightness difference $\Delta\tau$ is positive when G-BATA gives the tighter bound and negative otherwise. We synthesized the differences in Table 7.1. We computed the minimum, maximum and average tightness difference.

Even though they are based on fundamentally different theories, we can notice both approaches yield very close results, giving credit to both models.

Authors in [42] have shown that only 4 VCs are sufficient to find a mapping of flows to VCs that ensures each flow has exclusive use of the VC within each router, which greatly simplifies the computation. However, having only one flow per VC at each node can raise scalability problems: with larger and/or less favorable configurations, ensuring each flow has the exclusive use of a VC within each router would require



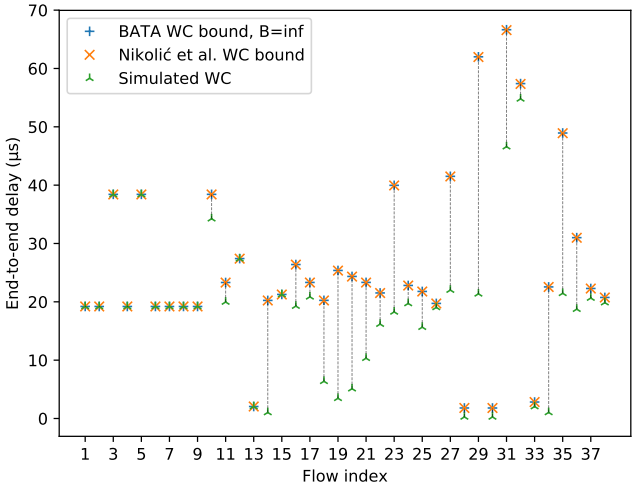




Figure 7.5 – Worst-case end-to-end delay bounds comparative

a number of different VCs that is not reasonable any more.

In that respect, we want to stress out that our model allows priority sharing and VC sharing (several flows sharing priority levels and VCs). Therefore, we have performed another analysis on the same configuration using only 2 VCs, with the following priority mapping:

- Flows 1 to 19 have the higher priority and are mapped to VC0;
- Flows 20 to 38 have the lower priority and are mapped to VC1.

We also analyze a configuration with only 1 shared VC.

We have plotted the results with the different VC configurations on Figure 7.6. We only displayed the results for a buffer size of 2 flits, but the trend is similar with other sizes. To get an insight into the impact of reducing the number of VCs on delay bounds, we also computed, for each flow and for each $n$ VC configuration, the relative increase of the worst-case delay bounds compared to the delay bound with 4 VCs, as follows:

$$inc = \frac{\text{delay with } n \text{ VCs} - \text{delay with 4 VCs}}{\text{delay with 4VCs}}$$

The results are on Table 7.2.

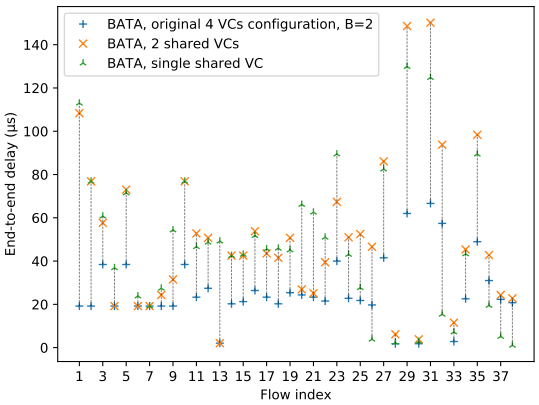



Figure 7.6 – Delay bounds with 4, 2 and 1 VC with buffer size = 2 flits

First, as we can notice from Fig. 7.6, all flows have delay bounds less than their periods (the shortest period is 40 ms); thus remain schedulable. More particularly, when we reduce the number of VCs, the computed delay bound for each flow either increases (34 and 32 times out of 38 under 2 VCs and 1 VC, respectively) or remains the same (4 and 1 time out of 38 under 2VCs and 1VC, respectively). However, for 5 flows cases with one VC, the computed bound decreases compared to the

| | 2 VCs | 1 VC |
|---|---|---|
| Average bound increase | 101.11% | 145.13% |
| Minimal bound increase | 0.00% | -95.05% |
| Maximal bound increase | 464.16% | 2293.01% |

Table 7.2 – Relative increase of the worst-case end-to-end delay bounds for $B = 2$

original configuration with no shared VC (4 VCs). The concerned flows are among the flows that have the lowest priorities in the original configuration. Mapping all flows to the same priority allows more fairness. This means it tends to increase the delay bounds of the flows that had the highest priorities in the original mapping, and conversely to decrease the waiting time of lower priority flows. However, for 5 flows cases with one VC, the computed bound decreases compared to the original configuration with no shared VC (4 VCs). The concerned flows are among the flows that have the lowest priorities in the original configuration. Mapping all flows to the same priority allows more fairness. This means it tends to increase the delay bounds of the flows that had the highest priorities in the original mappin
Moreover, as shown in Table 7.2, the average bound increase stays reasonable (up to 150 %) when the number of available VCs is divided by up to 4. Hence, BATA and G-BATA yield noticeable improvements to decrease the platform complexity (less Virtual Channels) while guaranteeing schedulability, in comparison to the state-of the-art method in [42].
Finally, we provide some insights into the runtime of our methods. For each buffer size and number of VCs, we measured the runtime of our analysis and summarized our results in Table 7.3. We notice that runtimes with non-shared VCs are in the order of 10 times lower than runtimes with 1 and 2 VCs for BATA. This confirms our conclusions regarding the inherent complexity of BATA to handle the priority-sharing and VC-sharing assumptions.
With G-BATA, we observe a similar trend. Runtimes when VCs are shared face a significant increase and are even higher than those of BATA for the maximal value of buffer size. This may be due to the inherent additional complexity of the interference graph construction with G-BATA, shown in Section 5.6.1, Figure 5.5 (especially for large buffer sizes), and the fact that the configuration used is not large enough to fully witness the computational gain from using G-BATA instead of BATA. In the studied cases, G-BATA still performs in an acceptable duration.
When no VC is shared between several flows, the IB latency is zero, and conse-

| | | 4 VCs | 2 VCs | 1 VC |
|---|---|---|---|---|
| Runtime of BATA (ms) | $B = 2$ | 6.36 | 55.4 | 84.6 |
| | $B = 100$ | 6.30 | 125.0 | 144.4 |
| | $B = \infty$ | 6.20 | 62.4 | 84.4 |
| Runtime of G-BATA (ms) | $B = 2$ | 4.77 | 62.2 | 316 |
| | $B = 100$ | 4.95 | 55.8 | 306 |
| | $B = \infty$ | 4.78 | 217 | 1346 |

Table 7.3 – Runtimes of BATA and G-BATA for different NoC configurations

quently, computing the end-to-end service curve is faster. This also explains the similar runtimes between both approaches in that case. On the contrary, when VCs are shared, there are (i) additional recursive calls to end-to-end service curve function needed to compute the IB latency with BATA; and (ii) a more complex interference graph to construct with G-BATA. Therefore, we can expect an increase in the analysis duration, for both approaches, but due to different reasons. Although G-BATA is generally faster than BATA, we observe the opposite in the configuration with 2 VCs and 1 VC. We stress out that in this particular case, the duration of the end-to-end service curve computation does not take much longer with BATA compared to G-BATA. Therefore, most of the analysis duration with G-BATA corresponds to the IB analysis and takes longer than the IB analysis BATA.

## 7.3 Control of an Autonomous Vehicle: Modeling and DSE

We now model the case study, assuming buffers can hold 2 flits. We proceed to a few alterations of the system according to our results from the previous section. First, we consider a platform with one single shared VC. It still allows flows to meet their deadlines and it reduces the complexity of the platform, therefore we want to further explore this option.

Second, we reduce the packet lengths and periods of the flows so that the rates of all flows remain constant. This way, we can observe more packet transmissions within the same amount of simulated cycles. For instance, a flow with a packet length of 1024 flits and a period of 40ms has the same rate as a flow with a packet length of 4 flits and a period of 156,25$\mu$s.

Third, we consider the tasks run on CPUs clocked at 200MHz while the NoC is clocked at 2GHz. We compute the delays in cycles of 0.5ns.

### 7.3.1 Functional Description

We first define the different tasks and create channels between them to model the data flow they exchange. Then, we specify the behavior of each task. Figure 7.7 shows the task graph. Each edge represents a flow.

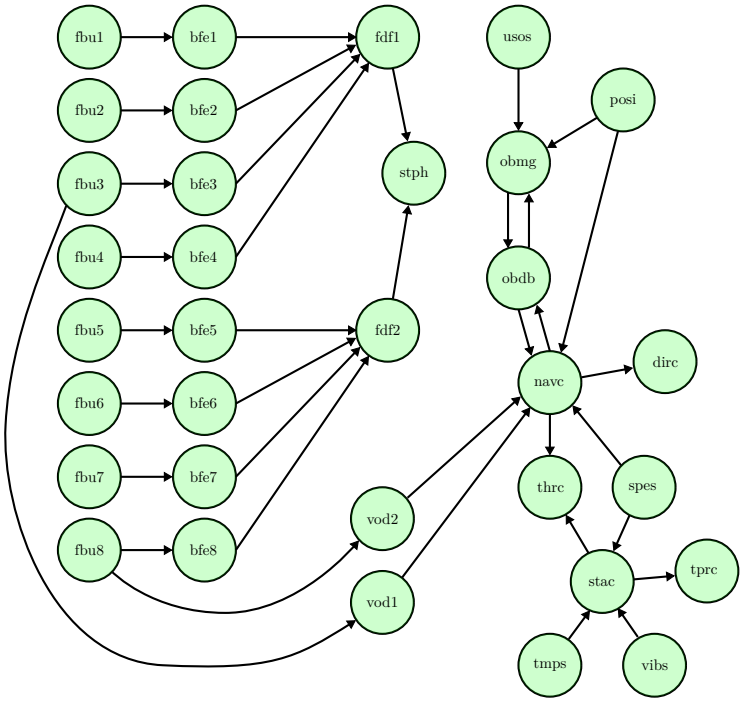



Figure 7.7 – Task graph of the case study

As we did with the example in Section 6.5.1, and to abstract the exact behavior of the tasks in between data exchanges, we trigger periodic packet emissions and receptions with dedicated control tasks. We sometimes used control events to several Read or Write operations, depending on the nature of the task(s) involved. We did so in a way that was reasonable considering the task description (Table C.1) and the graph, but we will not detail it here. As such, there are only 13 control tasks for the 38 flows, and consequently flow offsets are not all independent from each other. For instance (Figure 7.8), the control task 12 triggers two flows on task obdb: obdb to obmg and obdb to navc. It also triggers packet reception on task obmg, for the flow obdb to obmg.

Besides, some tasks will wait for a packet before sending one, which is consistent with the idea that the data sent by a task depends on the data it receives.

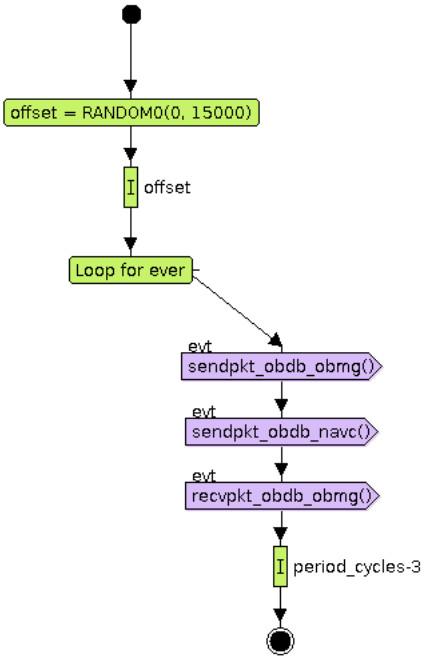



Figure 7.8 – Activity diagram of control task 12

### 7.3.2 Architecture Modeling

As the previously-conducted timing analysis showed the original mapping of the tasks complies with the deadlines on a single-VC platform, we will use the TTool based-approach to explore this option further. The architecture of the platform is shown on Figure 7.9.

To prevent control tasks from interfering with the real tasks, we map them on dedicated CPUs, or CPUs with enough cores to run all tasks independently. We use the same mapping as the original case study.

### 7.3.3 Simulation

Before entering the main loop of each control task, we randomly choose an offset and wait the corresponding duration. This allows us to simulate different scenarios of flow release each time we run a simulation from $t = 0$. We perform 300 simulations for 500000 cycles.

We derive the end-to-end delay of the flows as we did for the approach assessment (Section 6.5), by extracting packets request, injection and ejection times.

We present the results on Figure 7.10. Since there are 38 flows with end-to-end delays of different orders of magnitude, we do not show the distribution of the

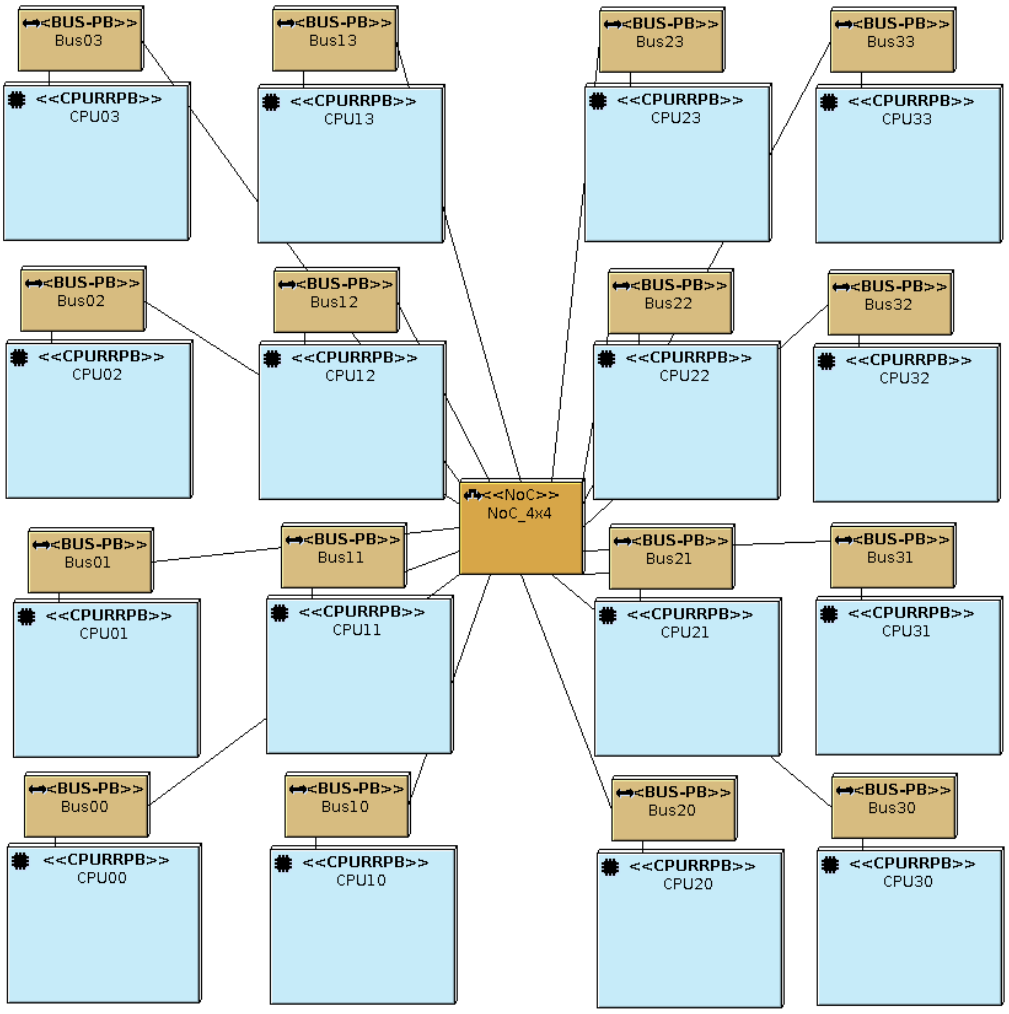



Figure 7.9 – Architecture of the platform

measured delays for all flows. Instead, we first "normalize" all delay measurements to a value between 0 and 1 using an affine transformation (0 corresponds to the base latency, and 1 to the delay bound). Then, we plot all "normalized" delays of all flows on the same graph. We can see that most values are close to 0, which denotes an end to end delay close to the base latency. This means that statistically, most flows experience little congestion, if any. There are values near 1 as well. They correspond to a flow that has its base latency equal to its worst-case delay bound.

## 7.4 Results and Conclusion

We were able to use our model to accurately bound the transmission time of flows on a physical manycore chip, with a reasonably good tightness, which proves both

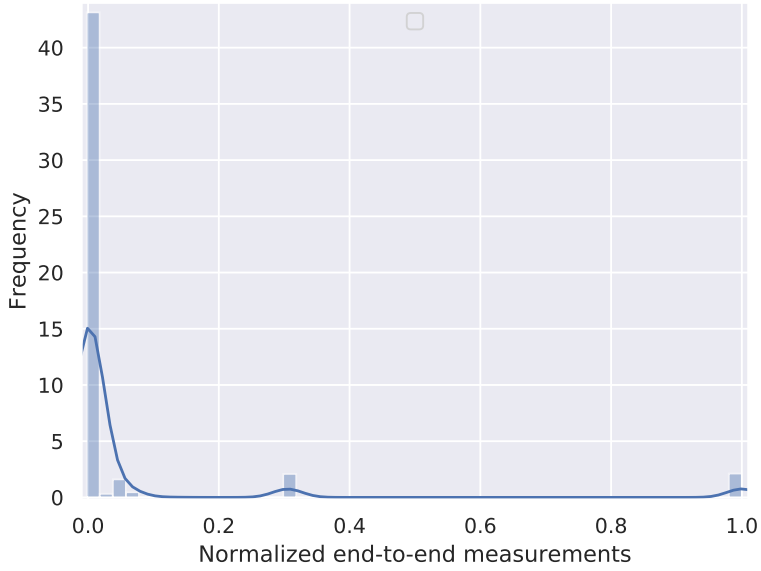



Figure 7.10 – End-to-end delay measurements on the case study

BATA and G-BATA are able to model real-world configurations.

On the autonomous vehicle case study, G-BATA allowed us to improve the state-of-the-art approach by extending the analysis applicability to shared virtual channels, proving that a single-VC platform can be used while satisfying the timing constraints. Even then, the worst-case delay bounds are at least 280 times smaller than the deadline. This suggests that the platform network capabilities could handle a heavier traffic load.

Following this analysis, we performed additional modeling and simulation with TTool, varying the flow offsets. We showed that end-to-end delays are statistically likely to be close to the base latency of the flows. Considering the flow rates are very low (0.00048 flits per cycle at most), this result is not surprising, and suggests it would be reasonable, on the presented case study, to use a way of synchronizing flows to avoid congestion when possible. Doing so would reduce the worst-case delay bounds of the flows and improve predictability of the application. It would also allow to better use the residual service offered by the NoC.

In terms of performance, the code generation and compilation for the model took around 8 minutes, while simulating 2 million cycles was done in approximately 53

minutes, that is more than 3000 times the duration of the timing analysis (under a second at most).
This confirms the conclusions of Chapter 6 and stresses out the relevance of integrating formal worst-case timing analysis at early design stages.

*Heure par heure, alors que décroît la lumière,*
*Assis parmi les murs de livres, nous plissons*
*Nos yeux, jusqu'à ce que l'horloge nous libère*
*À l'étreinte furtive où se meurt un frisson.*

*
* *

# Part III

# Conclusion

*In three words: Think before deploying.*
*In two words: Think first.*
*In one word: Don't.*

—James Mickens

*All we have to decide is what to do with the time that is given us.*

—J.R.R. Tolkien, *The Fellowship of the Ring*

Chapter 8

# Conclusion

Contents


**8.1 Summary of Contributions . . . . . 151**
8.1.1 BATA . . . . . 151
8.1.2 G-BATA . . . . . 152
8.1.3 Hybrid Design Space Exploration . . . . . 152
8.1.4 Validation . . . . . 153
**8.2 Perspectives . . . . . 153**
8.2.1 Models and Approaches . . . . . 153
8.2.2 Tools . . . . . 155


## 8.1 Summary of Contributions

### 8.1.1 BATA

Our first works addressed worst-case timing analysis of flit-preemptive wormhole Networks-on-Chip. We introduced BATA, an approach to derive worst-case delay bounds of flows on a mesh NoC with shared virtual channels implementing priority classes. This approach takes into account serialization phenomena and arbitrary buffer sizes. It uses Network Calculus results to model traffic flows and network elements, combined with a formalism to analyze "indirect blocking" scenarios, when packets of different flows may queue one after the other and block several links of the network.

We then evaluated the tightness of the approach by performing simulations of a test configuration and comparing the delays with the worst-case delay bounds computed using BATA. On the tested configuration, the average tightness of the end-to-end

delay bounds reaches 80%. However, we found that BATA computational complexity limits its scalability. For instance, computing bounds for configurations with 48 flows on a $8 \times 8$ 2D mesh NoC usually takes two hours or more.

### 8.1.2 G-BATA

Therefore, we presented G-BATA approach to tackle the complexity issue of BATA. We also improved the applicability of the approach: we used an extended traffic model and we adapted our formalism to cover bursty traffic flows and heterogeneous architectures. Subsequently, we modified the indirect blocking analysis to integrate these extensions. We rely on a graph structure to model interference between flows.

Our performance evaluation of G-BATA found that it yields similar tightness results as BATA. The indirect blocking analysis based on interference graph takes longer than the one in BATA but allows to speed up the computation of the end-to-end delay bound. The total analysis with G-BATA on 48 flows configurations exhibits a computation time 10 to 100 times lower than BATA. Moreover, G-BATA performs well on large configurations, with computation times around 9 seconds per flow for scenarios with 800 flows on an $8 \times 8$ NoC.

### 8.1.3 Hybrid Design Space Exploration

To improve the techniques of design space exploration for real-time applications on NoC-based platforms, we presented a workflow complying with the Y-chart approach integrating our work on timing analysis. The aim is to trim down the design space at an early design stage and avoid to simulate mappings that do not satisfy timing constraints.

We implemented our proposal on the base of existing tools, TTool and WoPANets. We were able to show that on a simple configuration, simulation alone turned out to be more costly than timing analysis by several orders of magnitude. It also was unable to detect a potential deadline miss for one flow.

However, simulation allows to gain a deeper insight on the distribution of delays among the tested scenarios. It can also be used to refine timing analysis results. For instance, if some flows are synchronized in such a way that they do not interfere, simulations will show it and help the designer orientate his effort to formally prove

it.
This exhibits the interest of combining a formal approach with simulation at system-level when performing design space exploration.

### 8.1.4 Validation

Finally, we confronted our approaches to practical applications. We successfully generated and analyzed traffic patterns on a Tilera TILE-Gx8036 manycore chip and showed that G-BATA and BATA provided safe delay bounds in regards to the latencies we measured. This proves our timing analysis methods can be practically applied to real-word systems.

Then, we focused on a case study, the control application of an autonomous vehicle. We compared our delay bounds results to those obtained by a state-of-the art approach based on scheduling theory. Our results are similar although the underlying theory is fundamentally different. This further confirms our bounds are safe. Moreover, our approach can also analyze configurations with shared virtual channels, which is not possible with the scheduling theory based approach.

Following this, we were able to model and simulate a configuration based on the same case study, but with shared VCs. To this end, we used part of our implementation of the hybrid design space exploration workflow we presented. We detailed the modeling choices we made and estimated the distribution of the delays of the different flows.

## 8.2 Perspectives

We now present some development perspectives that stem from our contributions. These either regard the analysis and models we use, or the tools that implement our approaches.

### 8.2.1 Models and Approaches

On the modeling and analysis aspect of our contributions, we see three axes of improvement regarding: (i) the indirect blocking computation in the timing analysis approach; (ii) the timing analysis step of the hybrid DSE methodology; and (iii) the mapping selection strategy and the system modeling paradigm of the

DSE approach.

In our G-BATA analysis, subpaths are computed relatively to any flows. This means that given a subpath $\mathbb{S}$ of a flow $f$, the algorithm will compute the subpath of $f$ relatively to $\mathbb{S}$, and iterate this computation of consecutive subpaths until the computed subpath is empty. The consequences of this were exhibited during the sensitivity analysis, when we noticed that in the IB set of a given flow, the shorter the subpaths (*i.e.* the larger the buffer size), the greater their number.
The number of consecutive subpaths of the same flow is not bounded in the algorithm in its current state. This is what allows us to cover the bursty traffic assumption and account for consecutive packet queueing. However, we could try to find a way of bounding the maximum number of consecutive packets in a given flow. One way to do this would be to use the end-to-end service curve to derive the burst of all flows at the end of their path using the output arrival curve expression in Theorem 6. We can use the value of the burst to derive the maximum number of consecutive packets of each flow that can queue in the network at all times. Then, we inject that value it in the interference graph construction algorithm to limit the number of consecutive subpaths it computes for each flow, and recompute the delay bounds according to the new updated graph.
Iterating this step would ultimately converge to a new value of the end-to-end delay that should improve G-BATA tightness, particularly for short subpaths (large buffers and/or small packets).

For now, our DSE approach uses G-BATA only to get a binary answer on whether or not the considered mapping satisfies the timing constraints. A possible improvement of the approach would be to consider the insight we gained with the G-BATA sensitivity analysis to perform optimizations related to the architecture characteristics. We expect this to orientate the design space exploration and save additional time by converging more rapidly towards a system satisfying the timing requirements.

Our workflow does not specify a strategy to generate mappings. We could work on that aspect as well, for instance by proposing different strategies to select the next mapping to be evaluated. Combined with optimization approaches, this could also help speed up the exploration process.
Besides, in our DSE approach, the code generation and compilation steps represent a major computational cost relatively to the timing analysis step. Since our

methodology targets NoC-based systems, we could consider using Repetitive Structure Modeling or a similar paradigm in the workflow to simplify the system model.

### 8.2.2 Tools

Our contributions lead us to use, extend or implement tools. There is still room for improving our work and make it easier to practically use the approaches we presented in this thesis. These improvements regard either TTool or WoPANets.

In WoPANets, there are perspectives in the technologies supported. Although our timing analysis approaches mostly target NoCs with input-buffered routers, we mentioned that they can be applied to NoCs with output-buffered routers because this paradigm does not change the number of links and buffers that a given flow will cross along its path. The only difference is a change of node indexing in the computation of the subpaths. We have not implemented this feature, but it would be possible and interesting to integrate it in the NoC worst-case timing analysis plugin.

Still on the WoPANets end, the timing analysis step of the DSE workflow is based on the extraction of certain characteristics needed for the timing analysis from the functional view of the system provided by TTool. For now, it requires to manually compute the period or inter-packet arrival time of the flows. We could work on this aspect as well, and design a piece of software (either integrated in WoPANets or as a standalone project) to process the system functional description and activity diagrams in a more advanced manner. This is related to interoperability issues and connecting different representations of the same realities. Addressing this aspect actually outgrows the sole scope of a software interfacing problem. It would contribute to bridge the gap between system-level view and timing analysis view of NoC-based platforms and more generally networked systems.

On the system modeling aspect, our NoC model can be extended. For instance, it is possible to code additional router models that can be used in place of the input-buffered router with VCs we designed. Similarly, we can add other possibilities for NoC topology, network interface behavior and routing algorithm.
With the experience we gained when adding the NoC model in TTool, we could also try to improve the NoC model we designed. This could offer more flexibility to the designer in terms of parameter choices, in particular regarding the technological latency and processing capacity.

In its current state, the simulation engine of TTool does not offer trace processing capabilities adapted to NoC-based systems. Therefore, we wrote a series of Python and bash scripts to extract relevant data from the simulation traces and derive values of metrics such as end-to-end delays. We also pipelined simulation steps and trace processing steps to avoid using too much disk space when performing simulations. However, since they were developed as they were needed, these tools are not always user friendly and do not have a unified usage syntax. It would be worth improving their usability and integrating them into TTool to help designers getting insight on the NoC-based system they are working on.

*Rien ne laissait penser que j'irai jusqu'à faire*
*Huit ans pour un diplôme.*
*Il m'a fallu le soutien d'une quasie-frère,*
*Transcendant le génome !*

*
* *

# Part IV

# Appendix

*La théorie et la pratique c'est la même chose,*
*sauf qu'en pratique ce n'est pas vrai*

—Étienne Klein

*They died from terminal stupidity.*

—Frank Castle, *The Punisher*

Annexe A

# Plateformes pluri-cœurs avec réseau sur puce pour les applications temps réel : Analyse de performance et exploration d'architectures

---

**Contents**


**A.1 Introduction . . . . . 160**

**A.2 Contexte de la thèse . . . . . 162**

A.2.1 Systèmes temps-réel . . . . . 162

A.2.2 Architectures pluri-cœurs . . . . . 163

**A.3 État de l'art . . . . . 166**

A.3.1 Analyse temps réel des réseaux sur puce . . . . . 166

A.3.2 Exploration d'architectures et mapping logiciel/matériel sur architectures pluri-cœurs . . . . . 170

**A.4 Analyse temporelle pire cas des réseaux sur puce wormhole intégrant l'impact des mémoires tampon . . . . . 173**

A.4.1 Modélisation du réseau et des flux . . . . . 173

A.4.2 Illustration du problème . . . . . 175

A.4.3 Formalisme et calculs . . . . . 176

A.4.4 Résumé de l'analyse de performance . . . . . 178

A.4.5 Conclusion . . . . . 182

**A.5 Analyse temps réel des NoCs wormhole hétérogènes par graphe d'interférences . . . . . 182**

A.5.1 Formalisme étendu . . . . . 182

A.5.2 Définition et construction du graphe d'interférence . . . . . 183

A.5.3 Analyse de performance . . . . . 185
A.5.4 Conclusion . . . . . 186
**A.6 Approche hybride pour l'exploration d'architectures . . . . 186**
A.6.1 Workflow étendu . . . . . 187
A.6.2 Modélisation système du NoC . . . . . 187
A.6.3 Performance . . . . . 189
A.6.4 Conclusion . . . . . 191
**A.7 Validation des contributions . . . . . 193**
A.7.1 Analyse pire cas . . . . . 193
A.7.2 Modélisation sous TTool . . . . . 195
**A.8 Conclusion . . . . . 196**
A.8.1 Résumé des contributions . . . . . 196
A.8.2 Perspectives . . . . . 198

## A.1 Introduction

Les architectures des processeurs ont longtemps été organisées autour d'un unique composant capable d'exécuter des opérations. Désormais, ce paradigme montre ses limites devant l'augmentation ininterrompue des besoins des systèmes en matière de puissance de calcul, et ce malgré les diverses améliorations (pipeline, exécution spéculative, prédiction de branchement, augmentation de la fréquence d'horloge).
Les architectures dites multi-cœurs (intégrant plusieurs cœurs de calcul) ne résolvent que très partiellement le problème puisque l'interconnexion des processeurs repose sur un bus, intrinsèquement incapable de passer à l'échelle.
Les architectures de type pluri-cœurs [4] ont été proposées pour pallier ces limites. Elles consistent en un grand nombre de processeurs sur la même puce, souvent organisés selon une matrice de dalles regroupant un ou plusieurs processeurs. Les dalles sont interconnectées par un réseau sur puce, ou *Network-on-Chip* (NoC) afin d'assurer les mécanismes de cohérence de cache, et plus généralement les requêtes mémoire et l'échange de messages d'un cœur à l'autre, tout en garantissant un maintien des performances pour plusieurs dizaines ou centaines de cœurs. De telles puces sont désormais proposées par plusieurs fabricants [5, 6, 7, 8].

Les systèmes critiques (dans l'automobile ou l'avionique notamment) assurent leur sûreté de fonctionnement par des processus de certification coûteux qui

favorisent la réutilisation de composants éprouvés et certifiés. Il y a, au contraire, peu d'incitations à utiliser des architectures plus performantes. Néanmoins, la transition vers des architectures pluri-cœurs paraît inévitable en raison de (i) l'abandon progressif des architectures mono-processeurs devenues obsolètes, qui risque de limiter leur disponibilité ; (ii) les limites déjà apparentes de ces mêmes architectures, considérant les besoins présents et futurs des systèmes critiques.

Il est donc crucial, pour assurer la transition vers des architectures pluri-cœurs, d'être en mesure de garantir la bonne exécution d'applications critiques sur de telles plateformes. Il existe plusieurs moyens de tendre à ce but, tous devant prendre en compte les spécificités des architectures pluri-cœurs quant au partage des ressources (mémoire, média de communication) et d'exécution concurrente.
Le présent rapport propose des contributions visant à faciliter la transition vers des architectures pluri-cœurs dans un contexte temps-réel. Nous nous efforçons de proposer des solutions adaptables à de nombreuses plateformes différentes, en travaillant dans un premier temps sur l'analyse temporelle pire cas des communications sur un réseau sur puce. Une telle analyse permet, connaissant le trafic généré et/ou consommé par les tâches s'exécutant sur une architecture pluri-cœurs, de borner les délais de transmission des messages sur le réseau sur puce et de déterminer si la configuration donnée permet de respecter les contraintes temps-réel associées. Nous proposons deux modèles à cette fin (A.4, A.5). Nous évaluons leur sensibilité aux paramètres d'entrée, leur finesse dans l'estimation du pire cas, et leur capacité à passer à l'échelle.
Dans un second temps, nous nous intéressons à l'exploration d'architecture pour les systèmes basés sur des réseaux sur puce. Nous proposons (A.6) une méthodologie pour intégrer le respect des contraintes temps-réel dans le processus de conception d'un système.
Enfin, nous procédons à des expériences sur une plateforme pluri-cœurs pour prouver l'applicabilité de nos modèles d'analyse temporelle et valider les résultats qu'ils génèrent, et nous appliquons nos contributions à deux études de cas réalistes. Nous détaillons une partie de la validation de nos travaux en section A.7.
La section A.8 conclut ce rapport et présente les perspectives futures découlant de nos contributions.

## A.2 Contexte de la thèse

Face à cette nécessité de fournir des outils pour l'analyse et le design des systèmes temps-réel basés sur des architectures pluri-cœurs, nous définissons dans un premier temps le cadre propre aux systèmes temps-réel. Puis, nous nous intéressons aux différents paradigmes existants dans les réseaux sur puce et à leur pertinence dans un cadre temps réel, afin de justifier le contexte restreint que nous considérons dans les sections suivantes.

### A.2.1 Systèmes temps-réel

Un système est dit *temps-réel* si la validité du résultat fourni est conditionnée non seulement par sa validité logique mais aussi par l'obtention dudit résultat selon une contrainte temporelle [12].
Contrairement aux systèmes dits *best effort*, qui cherchent à optimiser un temps de réponse moyen, le principal enjeu des systèmes temps-réel est de garantir le respect des échéances dans le pire des cas possibles, même si cela doit impacter le cas moyen.

On distingue, parmi les systèmes temps-réel, différents niveaux de *criticité* qui estiment la gravité des conséquences d'un non-respect des contraintes temporelles. Les systèmes temps-réel *durs* sont les plus critiques. Une défaillance d'un tel système est grave et peut entraîner d'importants dommages matériels ou la mort d'êtres vivants. A contrario, les systèmes temps-réel *mous* peuvent tolérer un échec occasionnel à respecter les contraintes temporelles sans que cela compromette le fonctionnement du système, ou sans que cela ait de conséquences sérieuses. On trouve parfois la notion de système temps-réel *ferme* pour désigner les systèmes dont le fonctionnement est grandement compromis en cas de non-respect des contraintes temporelles, mais ce sans conséquence grave sur le reste du monde.
En aéronautique, par exemple, l'application FADEC (Full Authority Digital Engine Controller) contrôle les turbofans d'un aéronef. C'est un système temps-réel dur, car sa défaillance peut entraîner la perte d'un moteur ainsi que des dégâts sur l'appareil ou les humains qu'il transporte.
Le système de vidéo à la demande de Netflix est un système temps-réel mou. Les conséquences d'une partie des photogrammes (trames vidéo) n'arrivant pas à destination à temps peuvent être imperceptibles, ou au pire provoquer la frustration de l'utilisateur et quelques jurons.

### A.2.2 Architectures pluri-cœurs

Pour éviter les goulots d'étranglement, les architectures pluri-cœurs reposent sur un paradigme d'accès mémoire non uniforme (NUMA, Non-Uniform Memory Access). Chaque dalle possède son propre cache L1 et/ou sa propre mémoire. Ainsi, un processeur peut adresser des requêtes localement ou vers une dalle distante, que ce soit pour les opérations mémoire, les mécanismes de cohérence des mémoires caches, l'envoi de message d'un cœur à l'autre, ou encore l'accès aux périphériques externes et aux entrées/sorties (I/O). C'est pourquoi le choix du moyen de faire communiquer les cœurs entre eux est crucial pour les performances de la puce.
Utiliser un bus devient assez vite problématique puisque la bande passante offerte à chaque composant diminue linéairement avec le nombre de composants partageant le bus. Une interconnexion point-à-point, elle, nécessite un nombre de câbles qui croît de façon quadratique avec le nombre d'entités connectées. Contrairement à ces deux paradigmes, les réseaux sur puce permettent un certain maintien de la bande passante et du débit offerts lorsque le nombre de nœuds augmente, tout en gardant une complexité de câblage raisonnable. Ainsi, ils constituent une solution privilégiée pour l'interconnexion des dalles d'une puce pluri-cœurs.
Ces réseaux sont similaires aux réseaux commutés classiques, puisqu'ils sont composés de routeurs connectés aux dalles de calculs et à d'autres routeurs, grâce auxquels sont transmis des paquets. Cependant, leur nature « sur puce » les rend particulièrement sensibles à des paramètres comme la consommation électrique et la dissipation énergétique, la quantité de mémoire tampon nécessaire dans chaque routeur, la surface occupée sur la puce ou la complexité de câblage. Les paramètres possibles pour un réseau sur puce concernent :

- la topologie du réseau ;
- le(s) protocole(s) utilisé(s) pour la transmission des données d'un nœud à l'autre, et la gestion du trafic ;
- les politiques d'arbitrage ;
- l'utilisation ou non de canaux virtuels (VCs, Virtual Channels) ;
- le(s) algorithme(s) de routage.

Nous restreignons notre étude aux réseaux sur puce utilisant des paradigmes :

- permettant de maintenir une certaine prédictabilité de fonctionnement ;
- prenant en compte les contraintes liées à la nature sur puce ;
- garantissant la scalabilité du système ;
- utilisés, voire courants, dans les composants sur étagère.

Les sections suivantes résument et justifient nos choix.

#### A.2.2.1 Topologie

La topologie désigne la façon dont les routeurs sont connectés entre eux et aux dalles de calcul ou autres composants. Dans les réseaux sur puce, celle-ci est statique et souvent régulière. La grille, ou *mesh*, et le tore [20, 21, 22, 23, 24, 25] sont les paradigmes les plus courants (Figure A.1). Tous deux constituent un bon compromis entre simplicité, scalabilité et performances. On les trouve principalement en 2D, et occasionnellement en 3D [26, 27].

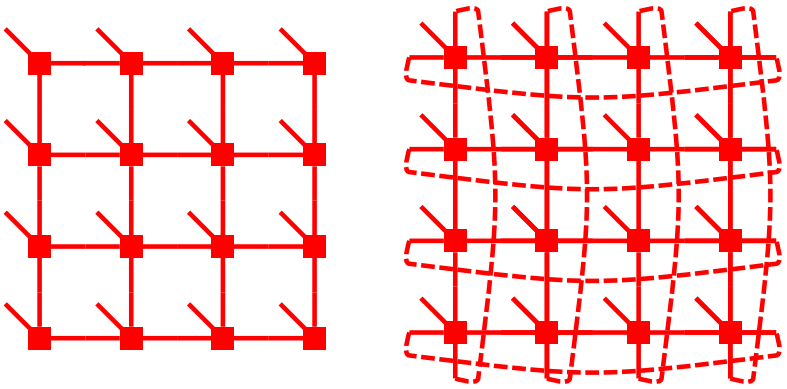


FIGURE A.1 – Topologies en grille et tore 2D

#### A.2.2.2 Transmission et contrôle de flux

La technique de transmission des données d'un routeur à l'autre impacte la latence d'un paquet transmis et la quantité de mémoire dite *tampon* nécessaire dans chaque routeur. Nous nous intéressons ici au *routage chenille* (en anglais *wormhole switching*) [20]. Le principe est de découper les paquets en mots de taille fixe, appelés *flits*, qui sont transmis d'un routeur à l'autre, à la queue leu-leu. Contrairement à la technique du *Store-and-Forward* (S&F) utilisée dans les réseaux classiques, il n'est pas nécessaire de mettre en mémoire tampon la totalité du paquet à chaque routeur avant de commencer la transmission vers le routeur suivant.

Cette technique est couramment utilisée car elle ne nécessite que de quoi stocker un flit par routeur (utiliser le S&F nécessite de pouvoir stocker au moins un paquet entier) et permet une transmission plus rapide que le S&F.

La technique de transmission est associée à un mécanisme de contrôle de flux, qui peut être un contrôle de flux :

**Par crédit, ou bloquant** – le router en aval autorise l'envoi d'un flit par un système de crédits. Lorsque tous les crédits sont consommés (*i.e.* que la mémoire tampon est pleine en aval), les flits doivent attendre.

**Par perte** – si un router ne peut pas accueillir de flit, il peut choisir de l'ignorer. La retransmission des flits perdus est gérée par une couche supérieure du réseau.
**Par déflection** – si un flit ne peut être envoyé vers le lien qu'il demande, il est détourné de sa route initiale vers une route disponible.
Les deux dernières techniques rendent le pire cas difficile à borner (techniquement, un paquet peut être jeté ou dévié un nombre arbitraire de fois consécutives). Ainsi, pour favoriser un comportement déterminisme, on privilégiera ci-après le contrôle de flux bloquant. Il est à noter que le contrôle de flux bloquant entraîne, en cas de congestion, une propagation du blocage à d'autres flux. En effet, un paquet bloqué occupe de l'espace en mémoire tampon parfois réparti sur plusieurs routeurs. Ce phénomène est appelé *propagation du blocage en amont* dans la suite de ce rapport.

#### A.2.2.3 Politique d'arbitrage et canaux virtuels

Lorsque deux paquets demandent l'utilisation de la même ressource, un routeur doit décider lequel des paquets sera le premier à utiliser la ressource. On parle alors d'arbitrage. Il peut y avoir plusieurs niveaux d'arbitrage au sein d'un routeur, par exemple entre les différentes entrées, ou entre différents types de paquets. Les politiques possibles, néanmoins, restent les mêmes. On distingue en particulier le premier arrivé premier servi, le tourniquet et sa variante le tourniquet pondéré, ou l'arbitrage par priorité fixe.
Cette dernière politique peut être implémentée en utilisant des canaux virtuels [38, 39]. Les canaux virtuels consistent à partager un lien physique en plusieurs liens logiques, en introduisant des files d'attentes séparées dans la mémoire tampon d'un routeur. Les flits de canaux différents empruntent donc le même lien entre deux routeurs, mais sont stockés et font la queue dans des files distinctes.

#### A.2.2.4 Algorithmes de routage

Pour qu'un paquet arrive à destination, il est nécessaire de calculer le chemin qu'il doit emprunter. On appelle *algorithme de routage* un algorithme permettant de réaliser cette tâche. Il doit avant tout assurer qu'aucun blocage irréversible (*deadlock*) n'ait lieu et que tout paquet arrive à destination dans un temps fini, et peut faire partie de différentes familles ([33]) :
**Centralisé *vs* distribué** – dans le premier cas, le calcul est fait par une entité centrale pour tous les flux. Dans le second cas, tout ou partie du calcul de route est réalisé par des entités différentes.
**À la source *vs* par saut** – dans le premier cas, l'entité émettrice d'un paquet cal-

| | Prédictabilité | Complexité | Scalabilité | Restriction |
|---|---|---|---|---|
| **Topologie** | | ✓ | ✓ | mesh, tore |
| **Transmission** | ✓ | ✓ | | wormhole |
| **Contrôle de flux** | ✓ | | | par crédit |
| **Arbitrage** | ✓ | ✓ | | RR, WRR, FP |
| **VCs** | | ✓ | | implémentation FP |
| **Routage** | ✓ | | ✓ | déterministe, distribué |

Tableau A.1 – Résumé des choix de conception et leur impact

cule la totalité du chemin qu'il doit emprunter. Dans le second cas, chaque routeur décide du lien vers lequel il doit envoyer un paquet reçu sans se préoccuper de la suite. Ces deux techniques font partie du routage distribué.

**Déterministe *vs* adaptatif** – pour une source et une destination données, un algorithme déterministe donnera toujours la même route, tandis qu'un algorithme adaptatif pourra réagir dynamiquement à d'autres paramètres comme la congestion ou la défaillance d'un lien.

**Minimal *vs* non-minimal** : le chemin calculé peut être ou non l'un des chemins les plus courts, en nombre de sauts.

#### A.2.2.5 Résumé

Pour faciliter l'analyse et l'obtention de garanties temps-réel, nous sommes amenés à restreindre les types d'architectures considérés. Le tableau A.1 résume les principaux impacts de chacun des choix de conception et les restrictions dans la suite de ce rapport.

## A.3 État de l'art

Deux domaines principaux sont au cœur de notre étude : l'analyse temporelle des réseaux sur puce, et l'exploration d'architectures. Dans les sections suivantes nous résumons les principales contributions dans chacun de ces domaines.

### A.3.1 Analyse temps réel des réseaux sur puce

Étant donnée une configuration de flux de données transitant par un réseau sur puce, l'analyse temps réel doit permettre, pour chaque flux, de déterminer une borne supérieure du pire temps de transmission d'un paquet. À ce titre, les méthodes

stochastiques ou basées sur la simulation seule ne sont pas adaptées à l'obtention de telles garanties.
Ce sujet a été abordé dans la littérature, en utilisant des méthodes comme la théorie de l'ordonnancement, le calcul récursif, la *Compositional Performance Analysis*, et le calcul réseau. Nous proposons une vue d'ensemble des contributions les plus pertinentes par approche dans les sections suivantes.

#### A.3.1.1 Théorie de l'ordonnancement

La théorie de l'ordonnancement traite à l'origine de problématiques liées à l'exécution de tâches sur des ressources de calcul. On peut l'appliquer à l'analyse temporelle des réseaux sur puce en faisant un parallèle entre les ressources d'exécution et les liens, mémoires et routeurs d'un réseau d'une part, et les tâches à exécuter et les messages à transmettre d'autre part. Le principal enjeu est alors de raffiner le modèle d'interférences pour capturer au mieux l'impact des flux les uns sur les autres.
Shi et Burns proposent [56] une analyse des réseaux sur puce avec routage chenille prenant en compte des niveaux de priorité implémentés grâce à des canaux virtuels, mais sans possibilité d'attribuer un même niveau de priorité à plus d'un flux, puis une extension [41] couvrant le partage d'un canal virtuel par plusieurs flux. Ces travaux prennent en compte les interférences dues à la propagation du blocage en amont, existant entre des flux qui ne partagent pas de ressources, et désignées par le terme « blocage indirect ».
Néanmoins, ce modèle entraîne une sous-estimation des délais dans certains cas particuliers mis en évidence dans [57], travail lui-même revu et corrigé [58], mais sans prendre en compte le partage des canaux virtuels pour les classes de priorité.
Des améliorations des travaux de [41] ont été proposées dans [59] à travers une prise en compte plus fine du domaine physique de contention, mais l'analyse se limite à des mémoires tampon pouvant contenir un flit.
Finalement, les travaux les plus récents sont ceux de [42], qui raffinent ceux de [58], mais qui toutefois ne prennent toujours pas en compte les canaux virtuels partagés. Les auteurs considèrent que cette limite peut être compensée par le fait que les futures architectures disposeront d'assez de canaux virtuels pour qu'il soit possible, à chaque nœud, d'assigner un unique flux à chaque file d'attente. Cependant, rien ne garantit qu'une telle répartition soit toujours possible, en particulier pour un grand nombre de flux. De plus, cette politique nécessite d'attribuer chaque flux à un canal virtuel hors ligne et n'offre donc peu ou pas de marge pour l'ajout dynamique de

flux non critiques ou non temps réel.

#### A.3.1.2 Compositional Performance Analysis

La *Compositional Performance Analysis* (CPA) a été proposée comme un cadre permettant de combiner différentes approches pour l'analyse de systèmes temps-réel embarqués complexes [61]. Elle utilise un formalisme voisin de la théorie de l'ordonnancement mais permet en sus l'utilisation de modèles existants pour l'analyse de parties spécifiques du système, et lie les résultats obtenus par différentes approches pour obtenir un modèle global.
Les travaux de [63] développent un modèle CPA pour l'analyse des réseaux sur puce avec routage chenille, basé sur l'approche de [55]. Ils distinguent différents types de blocage selon que le flux d'intérêt partage une entrée, une sortie, ou les deux avec un autre flux, et introduisent une notion de blocage indirect.
Cette approche prend en compte le partage des classes de priorité et des canaux virtuels, mais ignore le phénomène de propagation du blocage en amont lors des phénomènes de congestion.
Une extension de ces travaux modélise l'impact du contrôle de flux bloquant, mais en considérant un seul canal virtuel et des mémoires tampon de taille supérieure à un paquet.

#### A.3.1.3 Calcul récursif

Utilisé notamment pour l'analyse du réseau SpaceWire [65], cette technique repose sur une analyse de la contention subie par le flux d'intérêt sur chaque lien de son chemin, en prenant récursivement en compte la contention subie par les flux qui peuvent le bloquer.
La méthode telle qu'elle était présentée sur ces premiers travaux ne modélisait pas certains paramètres des flux comme le débit, et ne permettait pas de prendre en compte les mécanismes de priorité.
Des travaux plus récents [66] ont proposé une révision du calcul récursif pour couvrir des configurations avec des tailles de mémoire tampon supérieures à un flit (qui donnaient lieu à des résultats optimistes avec les approches basiques).
Plus récemment, les travaux de [67] étendent le calcul récursif en prenant en compte la sérialisation de la transmission dans les réseaux sur puce avec routage chenille et proposent un algorithme récursif intégrant leur approche, mais sans modéliser de canaux virtuels. Le modèle a néanmoins été appliqué à l'analyse temporelle des flux entre cœurs et entrées/sorties [13], sur des architectures pluri-cœurs du type

des puces Tilera [5].

#### A.3.1.4 Calcul réseau

Le calcul réseau est une théorie basée sur l'algèbre $(\min, +)$ introduite par [68] et développée par Le Boudec et Thiran [69]. Les principaux résultats sur lesquels nos travaux reposent sont présentés en annexe B.1. La théorie sous-jacente utilise des courbes d'arrivée pour borner le trafic cumulé en un point en fonction du temps, et des courbes de service pour modéliser le service minimal cumulé garanti par un élément de réseau en fonction du temps. À partir de ces deux types de courbes, pour un flux donné, des théorèmes (*cf* Annexe B.1) permettent d'obtenir une borne supérieure du délai de bout en bout ainsi que de la quantité de données présente dans le réseau à tout instant.
Le calcul réseau est utilisé dans [71, 72] pour l'analyse temporelle des réseaux SpaceWire. Dans [73], les auteurs proposent une analyse basée sur le calcul réseau en appliquant aux réseaux sur puce les travaux de [74] sur les réseaux en arbre. Leur approche distingue trois schémas de contention possible entre le flux d'intérêt et deux flux concurrents (en inclusion, parallèle, croisé) et se base sur la construction d'un arbre de contention. Elle ne prend pas en compte la taille finie des mémoires tampon et la propagation du blocage en amont qui en résulte.
Les auteurs prolongent leurs travaux dans [75] en analysant le blocage dans les réseaux sur puce avec routage chenille de façon récursive. Le système de contrôle de flux est modélisé comme un composant du réseau dont le comportement dépend du routeur aval. Cette méthode nécessite la résolution de problèmes de point fixe et sa capacité à passer à l'échelle n'est pas claire. De plus, elle ne considère que des réseaux sur puce sans canaux virtuels.
Les travaux de [76] raffinent les courbes classiquement utilisées (seau percé et débit latence), et prennent en compte la sérialisation des flux, mais sans considérer de routage chenille, évitant ainsi de devoir modéliser la propagation du blocage en amont.
Dans [77], les auteurs proposent une méthode de calcul des délais de bout en bout sur une puce Kalray MPPA2 [7]. Cette méthode reste néanmoins spécifique à une architecture, d'autant qu'elle suppose d'utiliser les composants de mise en forme du trafic offerts par la puce Kalray pour assurer qu'il n'y ait pas de propagation du blocage en amont.

#### A.3.1.5 Discussion

La plupart des approches connues s'appuient sur des hypothèses limitant leur applicabilité, restreignant notamment la technique de transmission (routage chenille ou autre), l'utilisation de canaux virtuels ou leur partage par plusieurs flux. De plus, beaucoup de modèles ne prennent pas en compte les phénomènes de sérialisation des flux ou la taille des mémoires tampon.
Nous proposons donc une approche aussi générale que possible pour pallier les limites des travaux existants. Un premier modèle permettra d'analyser : (i) les réseaux sur puce wormhole homogènes, de topologie quelconque fixe et munie d'un algorithme de routage déterministe (typiquement, un mesh 2D muni d'un routage XY) ; (ii) des routeurs avec des mémoires tampon de taille quelconque et des canaux virtuels implémentant des classes de priorité partagées par un nombre arbitraire de flux, et permettant la préemption au niveau flit ; (iii) des flux de données périodiques ou sporadiques.
Un second modèle permettra, en outre, d'étendre le premier pour couvrir : (i) les réseaux hétérogènes ; (ii) les flux de données dits *bursty* , ou *en rafale*, *i.e.* pouvant injecter plusieurs paquets consécutivement.
Nous choisissons d'utiliser le calcul réseau pour les raisons suivantes :

- la théorie sous-jacente offre de puissants résultats théoriques pour la modélisation des réseaux ;
- le calcul réseau est une méthode éprouvée, il a notamment servi à certifier le réseau AFDX utilisé dans les plus récents appareils produits par Airbus comme l'A380 ;
- sa modularité permet de mettre à jour ou améliorer les modèles sans devoir repenser toute l'organisation de l'analyse.

### A.3.2 Exploration d'architectures et mapping logiciel/matériel sur architectures pluri-cœurs

Concevoir un système temps réel basé sur une architecture contenant un réseau sur puce présente principalement deux aspects distincts. D'une part, à haut niveau, il est nécessaire de pouvoir restreindre les choix d'architecture à une ou quelques plateformes qui conviennent le mieux au système envisagé.
D'autre part, étant donné une architecture matérielle et un ensemble de fonctions implémentées logiciellement (applications et tâches), un autre problème est d'associer chaque fonction à un élément matériel chargé de son exécution. On appelle cette étape *mapping* logiciel/matériel.

Ces deux aspects peuvent être liés dans le processus de conception d'un système. Néanmoins, nous nous intéressons dans un premier temps aux méthodologies générales d'exploration d'architectures (section A.3.2.1), puis nous résumons les principales techniques de mapping dans la section A.3.2.2.

### A.3.2.1 Exploration d'architectures

L'exploration d'architectures suppose une modélisation du système en cours de conception. Différents facteurs sont à considérer dans la modélisation d'un système : le(s) niveau(x) d'abstraction offert(s) – system level, transaction level modeling (TLM), register transfer level (RTL), cycle-accurate bit-accurate (CABA) –, l'indépendance des modélisations fonctionnelles et architecturales, et les techniques d'exploration et de mapping disponibles (si applicable). D'autre part, il est important de considérer les aspects pratiques des méthodes choisies (format d'entrée, modèles disponibles pour les composants, possibilité de générer du code).
La conception de systèmes basés sur des réseaux sur puce a été abordée dans plusieurs travaux. Dans le domaine des systèmes de télécommunications, les travaux de [79] présentent un processus de conception permettant en outre la génération de code SystemC pour la validation du système. Dans [80], les auteurs proposent un outil reposant sur Arteris, permettant d'identifier les limites d'une architecture au regard des contraintes de performances d'une application, et d'itérer le processus pour converger vers une architecture qui satisfait les contraintes. Les travaux présentés dans [81] détaillent une extension du simulateur MPSoCSim intégrant un modèle de NoC étendu et plus flexible que son prédécesseur.

D'autres travaux se concentrent sur des outils plus génériques pour la conception de systèmes et leur modélisation à différents niveaux d'abstraction, comme Ptolemy [82, 83] pour les systèmes très hétérogènes ; Artemis [84], qui permet de détailler ou d'abstraire plus ou moins le système selon le stade de conception, et qui est compatible avec l'approche Y-chart [85] ; MoPCoM, une méthode de co-design avec trois niveaux d'abstraction pour les systèmes embarqués temps réel permettant de générer du code VHDL ; ou encore TTool [9], qui supporte divers environnements SysML/UML, comme DIPLODOCUS [90], permettant notamment la modélisation au niveau système suivant l'approche Y-chart.

Enfin, certaines approches intègrent des modèles de réseau dans les processus de conception et d'exploration d'architectures, permettant de distinguer les fonctions

relevant du réseau de celles relevant du système, d'analyser les interactions réseau/système, ou de proposer des abstractions pour les canaux de communication [92, 93, 94].

Essentiellement, aucune de ces approches ne prend en compte la vérification des contraintes temps réel autrement que par simulation.

#### A.3.2.2 Mapping logiciel/matériel

Étant donné un ensemble d'applications composées de tâches éventuellement assorties de contraintes (temps réel, utilisation de composants externes, précédence), et un ensemble de ressources, le problème du mapping est d'attribuer chaque tâche à un élément de calcul pour garantir la bonne exécution de chaque tâche tout en respectant les contraintes. Le mapping peut être fait en ligne ou hors ligne [95], et cette dernière solution est la plus adaptée au contexte temps réel pour des questions de déterminisme.

Le mapping sur architectures pluri-cœurs est un problème NP-hard [96, 95]. En raison de la complexité des systèmes basés sur des NoCs, trouver un mapping optimal devient rapidement déraisonnable du point de vue du temps de calcul.
Une première famille d'approches s'attèle à proposer des méthodes basées sur des algorithmes d'exploration approximatifs, qui fournissent une solution en un temps raisonnable sans garantir son optimalité, comme des algorithmes génétiques [97], par séparation et évaluation [99] ou par optimisation linéaire [100].
Certaines approches sont construites sur des heuristiques prenant en compte les contraintes de bande passante [98] ou la contention [100], favorisant la prédictabilité [101, 102], ou les contraintes d'entrées-sorties [13].
Une autre famille de travaux s'attache à offrir des garanties de prédictabilité sur l'exécution des tâches. Ces techniques reposent parfois sur l'utilisation d'extensions matérielles [103], ce qui limite l'applicabilité d'une telle approche et empêche largement l'utilisation de composants sur étagères. D'autres travaux formalisent les contraintes de mapping et proposent un modèle d'exécution qui génère un mapping spatial et un ordonnancement permettant d'exécuter un ensemble de tâches temps réel sur trois architectures sur étagère, avec les garanties temporelles associées [46]. Des travaux similaires dans la thèse de Quentin Perret [30] ont permis de proposer et d'implémenter un modèle d'exécution prédictible pour les applications temps réel sur une puce pluri-cœurs Kalray MPPA 256.

#### A.3.2.3 Discussion

Le principal manque des approches d'exploration d'architectures existantes concerne l'analyse pire cas des contraintes temporelles, en particulier dans le cas des architectures basées sur des réseaux sur puce. La vérification des contraintes temporelles, lorsqu'elle est abordée, est faite par simulation. À notre connaissance, aucune contribution n'intègre la vérification formelle des contraintes temporelles dans son processus de design et d'exploration d'architectures.
Ainsi, nous proposons comme troisième contribution (section A.6) une approche hybride d'exploration d'architectures, combinant la modélisation système et le calcul réseau. Son principal objectif est de réduire la durée du processus en éliminant au plus vite les configurations ne satisfaisant pas les contraintes temporelles.

## A.4 Analyse temporelle pire cas des réseaux sur puce wormhole intégrant l'impact des mémoires tampon

Notre première contribution est un modèle d'analyse temps-réel pour les réseaux sur puce wormhole, appelé BATA (**B**uffer-**A**ware worst-case **T**iming **A**nalysis). Nous présentons dans un premier temps les principaux éléments de notre modélisation basée sur le Calcul Réseau (section A.4.1).
L'idée centrale de ce modèle est de prendre en compte l'étalement des paquets dans le réseau afin de prendre en compte l'impact d'un paquet en attente dans le réseau sur le chemin des autres flux, et de prévoir les possibles blocages en cascade. La taille du paquet et des mémoires tampons des routeurs impacte directement les possibles scenarii de blocage. Nous montrons cet aspect sur un exemple, section A.4.2.
Une fois les configurations de blocage déterminées, on peut déduire une estimation haute du délai subi par un paquet du flux d'intérêt le long de son chemin. Ce calcul comprend 3 étapes, que nous présentons section A.4.3.
Enfin, nous procédons à une évaluation du modèle, section A.4.4.

### A.4.1 Modélisation du réseau et des flux

Nous considérons un réseau sur puce de type wormhole, avec contrôle de flux bloquant, à topologie statique. Le réseau est constitué de routeurs avec mémoires en entrée, supportant des canaux virtuels qui implémentent des classes de priorité. L'arbitrage entre les entrées est supposé quelconque, et l'arbitrage entre canaux virtuels suit une politique de priorité fixe. Nous ne spécifions pas d'algorithme de

routage, mais celui-ci doit être déterministe. On suppose de plus que deux flux dont les chemins divergent ne se rencontrent pas de nouveau après leur point de divergence. Pour fixer les idées, nos exemples considèreront des réseaux de type mesh 2D, routés selon un algorithme XY. L'architecture typique d'un routeur, similaire à celle présentée dans [24], est illustrée Figure A.2.

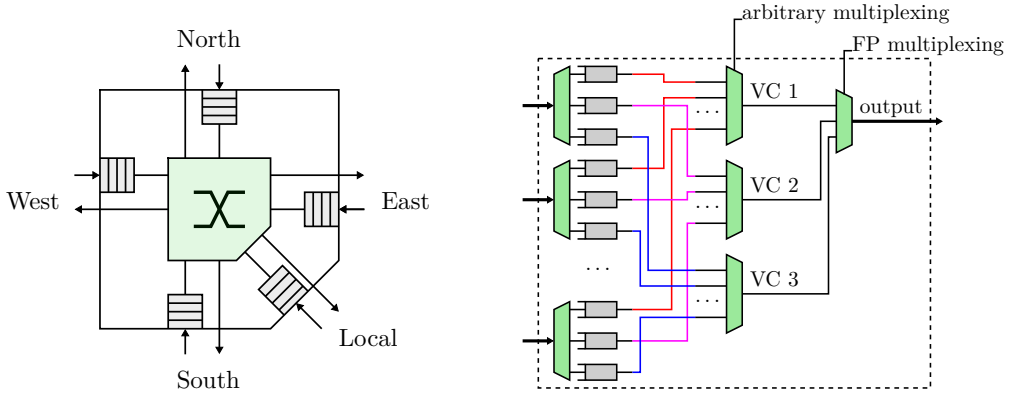



FIGURE A.2 – Architecture d'un router de mesh 2D et niveaux d'arbitrage

Pour modéliser les deux niveaux d'arbitrage, on considère que chaque sortie du routeur se comporte comme une multiplexeur avec deux niveaux d'arbitrage. Dans le reste de ce rapport, nous appelons *nœud* tout couple routeur,sortie. Ainsi, on modélise un nœud $r$, avec le formalisme du calcul réseau, par une courbe de service de type *rate latency*, ou *débit-latence* :

$$\beta^r(t) = R^r \left(t - T\right)^+$$

$R^r$ représente le débit minimum de données que peut traiter le nœud $r$ (typiquement exprimé en nombre de flits par cycle d'horloge) et $T^r$ représente le retard infligé par le nœud $r$ à un flit qui le traverse (typiquement, un ou quelques cycles).
Nous considérons ici que l'architecture est homogène, c'est-à-dire que tous les liens, mémoires tampon et routeurs du réseau ont les mêmes caractéristiques.

Le trafic considéré est périodique ou sporadique. On modélise un flux $f$ par une courbe d'arrivée de type seau percé (*leaky bucket*) :

$$\alpha_f(t) = \sigma_f + \rho_f t$$

dont les paramètres s'expriment en fonction de la période du flux $P_f$, de sa jigue

$J_f$ et de la taille maximale d'un paquet $L_f$ :

$$\rho_f = \frac{L_f}{P_f}$$
$$\sigma_f = L_f + J_f \cdot \rho_f$$

Étant donné un flux $f$, on désigne son chemin, *i.e.* la liste des nœuds qu'il traverse entre sa source et sa destination, par la notation $\mathbb{P}_f$. Pour un indice $k$ approprié, $\mathbb{P}_f[k]$ désigne le $k-1^{\text{e}}$ nœud sur le chemin de $f$. De plus, pour tout $r \in \mathbb{P}_f$, la courbe d'arrivée de $f$ propagée jusqu'au nœud $r$ conformément au théorème 6 (B.1) est notée :

$$\alpha_f^r(t) = \sigma_f^r + \rho_f^r \cdot t$$

La courbe de service offerte au flux $f$ sur son chemin complet est notée :

$$\beta_f(t) = R_f \left(t - T_f\right)^+$$

### A.4.2 Illustration du problème

Pour illustrer le problème, nous considérons la configuration décrite figure A.3, à gauche. Elle comprend trois flux. Nous supposons que chaque paquet contient 3 flits et chaque mémoire tapon peut contenir un flit, de telle sorte qu'il faut 3 mémoires tampon pour stocker un paquet. En outre, nous considérons que tous les flux utilisent le même canal virtuel.

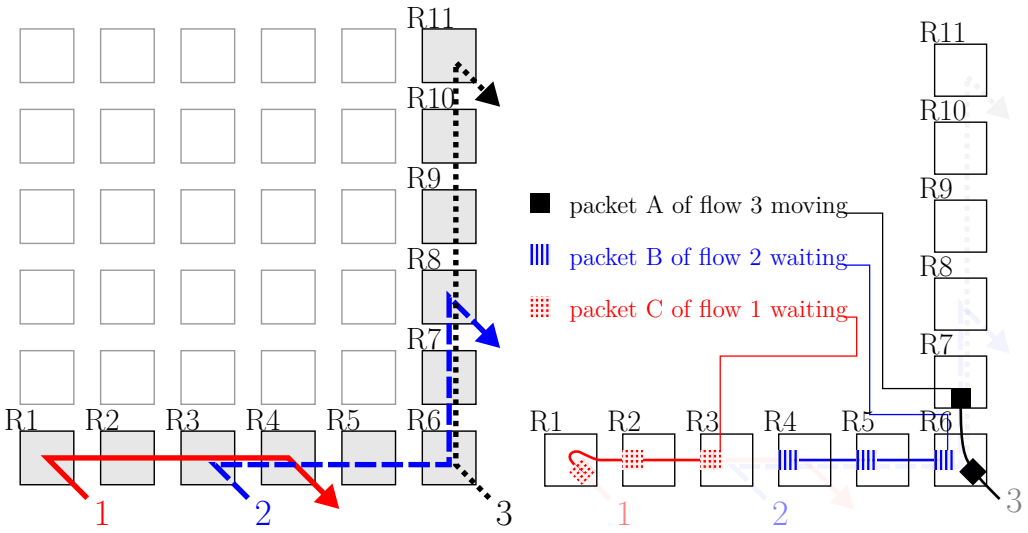



FIGURE A.3 – Configuration typique (gauche) et étalement des paquets (droite)

Supposons qu'un paquet A, du flux 3, vient d'être injecté dans le réseau, et qu'il a

été autorisé à utiliser le port de sortie Nord du routeur R6. Au même moment, un paquet B, du flux 2, demande l'utilisation de la même sortie. Comme le paquet A l'utilise, le paquet B doit attendre. Enfin, un paquet C du flux 1 vient d'attendre R3 et demande l'utilisation du port de sortie Est de R3. Cependant, comme la mémoire d'entrée du port Ouest de R4 est occupée par un flit du paquet B, le paquet C doit attendre.
Dans ce cas, A bloque indirectement C, alors même que les flux correspondants, 1 et 3, ne partagent pas de ressources. Nous appelons ce phénomène *blocage indirect*.

### A.4.3 Formalisme et calculs

Le calcul de la borne sur le délai de bout en bout pour un flux d'intérêt $f \in \mathcal{F}$ s'effectue en 3 étapes : l'analyse du blocage indirect subi par $f$, le calcul de la courbe de service, et le calcul du délai de bout en bout.
**Étape 1 – analyse du blocage indirect** : Cette étape détermine quels flux impactent le flux d'intérêt sans partager de ressources avec celui-ci.

**Definition 16.** *Le set de blocage indirect d'un flux $f$, appelé également IB set, est l'ensemble des flux ne partageant aucune ressource avec $f$ mais qui peuvent impacter son temps de transmission parce qu'ils impactent (directement ou non) un flux partageant des ressources avec $f$. On le note $IB_f$. Il contient des paires {indice de flux, sous-chemin} afin de spécifier, pour chaque indice de flux, la portion de chemin sur laquelle un paquet de ce flux peut impacter le flux d'intérêt $f$, par le biais du phénomène de propagation du blocage en amont.*

Pour quantifier l'étalement d'un paquet dans le réseau, nous introduisons la notion d'indice d'étalement, qui représente le nombre de mémoires tampons nécessaires pour contenir un paquet du flux considéré.

**Definition 17.** *L'indice d'étalement du flux $f$ est noté $N_f$ et défini par :*

$$N_f = \left\lceil \frac{L_f}{B} \right\rceil$$

Puis, nous définissions le *subpath*, ou *sous-chemin*, d'un flux par rapport à un autre.

**Definition 18.** *Étant donnés deux flux $k$ et $l$ dont les chemins s'intersectent, le sous-chemin de $k$ relativement à $l$ est la portion de $\mathbb{P}_k$, de longueur au plus $N_k$, située après le point de divergence de $\mathbb{P}_k$ et $\mathbb{P}_l$ :*

$$subpath(\mathbb{P}_k, \mathbb{P}_l) \quad = \quad \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{P}_l) + 1], \ldots, \mathbb{P}_l[Last(\mathbb{P}_k, \mathbb{P}_l) + N_k]\Big]$$

*où* $Last(\mathbb{P}_k, \mathbb{P}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{P}_l\}$ *est l'indice du dernier nœud partagé par* $k$ *et* $l$ *sur le chemin de* $k$*, soit l'indice tel que* $\mathbb{P}_k[Last(k,l)] = \mathrm{dv}(k,l)$*. Par convention, si* $k$ *et* $l$ *ne s'intersectent pas, les sous-chemins* $subpath(k,l)$ *et* $subpath(l,k)$ *sont vides.*

Nous étendons cette notion en définissant de la même manière le sous-chemin d'un flux $k$ quelconque relativement à n'importe quel sous-chemin $\mathbb{S}_l \subset \mathbb{P}_l$ d'un flux $l$ quelconque (avec $l \neq k$ ou $l = k$). On a alors :

$$subpath(\mathbb{P}_k, \mathbb{S}_l) \quad = \quad \Big[\mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + 1], \ldots, \mathbb{P}_k[Last(\mathbb{P}_k, \mathbb{S}_l) + N_k]\Big]$$

où $Last(\mathbb{P}_k, \mathbb{S}_l) = \max\{n,\ \mathbb{P}_k[n] \in \mathbb{S}_l\}$ est l'indice (dans $\mathbb{P}_k$) du dernier nœud partagé par $k$ et $l$ sur le sous-chemin $\mathbb{S}_l$. Par abus de notation, on pourra utiliser $subpath(k,l)$ pour désigner $subpath(\mathbb{P}_k, \mathbb{P}_l)$ et $subpath(k, \mathbb{S}_l)$ pour désigner $subpath(\mathbb{P}_k, \mathbb{S}_l)$.
Nous illustrons la notion de sous-chemin sur la figure A.4, en considérant le flux d'intérêt $k$ et un indice d'étalement égal à 3 pour le flux $l$.

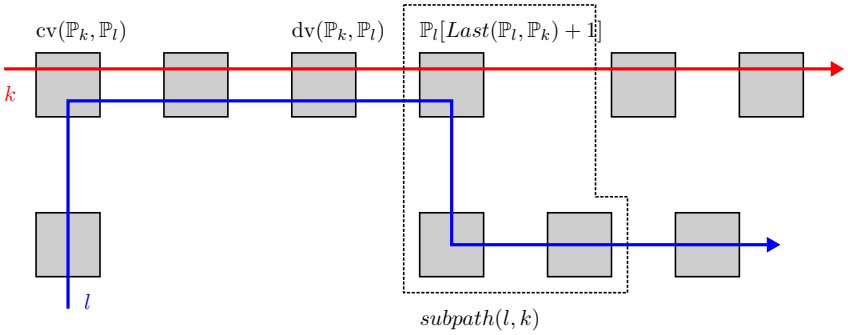



FIGURE A.4 – Calcul d'un sous-chemin relativement au flux $k$

Pour déterminer le set de blocage indirect d'un flux $f$, nous calculons les sous-chemins de chaque flux $k \neq f$ par rapport à $f$. Puis, nous itérons ce calcul relativement aux sous-chemins nouvellement calculés, jusqu'à n'obtenir plus aucun sous-chemin non vide. Le calcul est détaillé section 4.4, algorithme 1. L'on y trouvera également une borne supérieure de la complexité de l'algorithme.
**Étape 2 – courbe de service de bout en bout** : La deuxième étape consiste en deux calculs : le calcul de la latence de base et de la latence de blocage direct, puis le calcul de la latence de blocage indirect.
Le premier calcul prend en compte l'impact de la sérialisation des flux. Il s'appuie

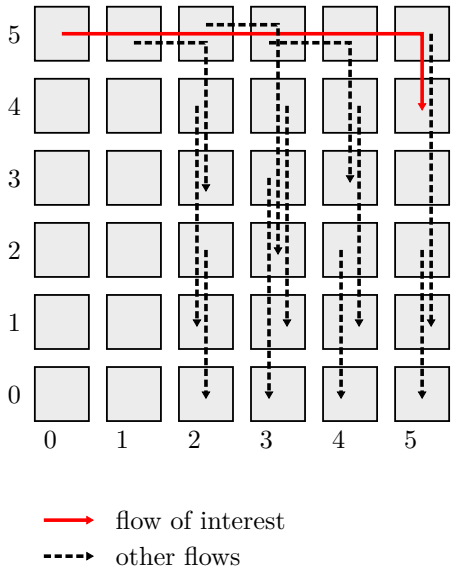
sur un résultat de [70] pour une politique d'arbitrage par priorité fixe, que nous rappelons Annexe B.1. Le résultat est quelque peu lourd à écrire et fait appel à des définitions supplémentaires, aussi nous ne le rappelons pas ici et renvoyons la sagace lectrice à la section 4.5.1.
Ensuite, nous calculons la latence de blocage indirect $T_{IB}$, en utilisant $IB_f$ calculé à l'étape 1. L'idée est d'additionner des délais de bout en bout sur chacun des sous-chemins de $IB_f$. Là encore, la lectrice pourra se référer à la section correspondante (4.5.2) pour les théorèmes et preuves détaillés.
Il est à noter que le calcul de la courbe de service est doublement récursif : il nécessite le calcul de courbes de services intermédiaires à la fois lors du calcul de $T_{DB}$ et de $T_{IB}$ . Cet aspect est détaillé section 4.5.3.
**Étape 3 – calcul du délai de bout en bout** : Connaissant la courbe de service offert au flux $f$ de bout en bout, nous utilisons le théorème 6, annexe B.1, pour calculer une borne supérieure du délai de bout en bout :

$$D_f^{\mathbb{P}_f} = \left\lceil \frac{\sigma^{\mathbb{P}_f}[0]}{R_f} + T_{\mathbb{P}_f} + T_{DB} + T_{IB} \right\rceil \tag{A.1}$$

Nous utilisons la fonction partie entière supérieure pour obtenir un nombre entier de cycles (un demi cycle d'horloge n'a ici pas de réalité physique).

### A.4.4 Résumé de l'analyse de performance

Nous procédons à une évaluation de notre modèle selon trois axes : (i) la sensibilité aux paramètres d'entrée ; (ii) l'évaluation de la finesse, c'est à dire de l'écart entre la borne calculée et le pire cas réel ; (iii) et enfin, l'aspect calculatoire, en particulier la scalabilité du modèle. Pour les deux premiers axes, nous utilisons une même configuration de 12 flux sur un NoC $6 \times 6$, détaillée figure A.5, sur laquelle nous varierons certains paramètres. Cette configuration, relativement simple, présente néanmoins des possibilités de blocage direct et indirect. Pour le troisième axe, notre étude est basée sur des configurations d'un nombre de flux variable, sur un NoC $8 \times 8$.
**Sensibilité :** Nous étudions dans un premier temps la sensibilité du modèle à trois paramètres d'entrée : la taille des mémoires tampon, le débit des flux et la taille des paquets. Chaque flux a la même période et la même taille de paquet, ainsi qu'une jigue nulle. Nous varions chacun des paramètres pour tous les flux et étudions l'effet de cette variation sur le délai de bout en bout d'un flux représentatif, le flux 1, car celui-ci subit du blocage indirect suffisamment complexe pour mettre en évidence




| Flux | Source | Destination |
|---|---|---|
| 1 | (0, 5) | (5, 4) |
| 2 | (1, 5) | (2, 3) |
| 3 | (2, 5) | (3, 2) |
| 4 | (3, 5) | (4, 3) |
| 5 | (5, 5) | (5, 1) |
| 6 | (2, 4) | (2, 1) |
| 7 | (2, 2) | (2, 0) |
| 8 | (3, 4) | (3, 1) |
| 9 | (3, 3) | (3, 0) |
| 10 | (4, 4) | (4, 1) |
| 11 | (4, 2) | (4, 0) |
| 12 | (5, 2) | (5, 0) |

FIGURE A.5 – Configuration test sur un réseau sur puce en grille 6×6

les particularités de notre modèle.

Nous faisons varier la taille des mémoires tampons entre 1 et 64 flits, la taille de paquet entre 2 et 128 flits, et le débit des flux entre 1% et 40%. Les résultats obtenus montrent en particulier les aspects suivants :

- Augmenter la taille des mémoires tampons n'est bénéfique pour les délais de bout en bout que jusqu'à un certain point. Une telle augmentation diminue l'indice d'étalement des flux, entraînant une diminution de la taille du set de blocage indirect et de la latence correspondante, et donc du délai de bout en bout. Cependant, pour des mémoires tampons plus grandes que la taille d'un paquet, cette diminution n'a plus lieu car l'indice d'étalement devient alors stationnaire ;
- Le débit des flux et la taille des mémoires tampons sont les paramètres impactant le plus significativement les délais de bout en bout.

Pour plus de détails, nous renvoyons la sagace lectrice vers la section 4.7.1.

**Finesse :** La finesse du modèle correspond à l'écart entre la borne pire cas calculée et le pire cas réel. Comme le pire cas réel est difficile voire impossible à déterminer, nous évaluons la finesse des bornes en utilisant la simulation. Nous simulons une configuration de nombreuses fois, en variant aléatoirement l'offset des flux, *i.e.* l'instant relatif où chaque flux commence à transmettre. Ainsi, nous modifions les configurations de blocage. Pour chaque flux, nous prenons ensuite le pire cas obtenu

| Rate | 8% | | | 32% | | |
|---|---|---|---|---|---|---|
| Buffer | 4 | 8 | 16 | 4 | 8 | 16 |
| Statistiques sur l'indice de finesse | | | | | | |
| **Moyen** | **70.1%** | **72.1%** | **80.8%** | **49.7%** | **64.2%** | **79.8%** |
| **Max** | **91.7%** | **92.0%** | **88.3%** | **95.6%** | **88.9%** | **97.3%** |
| **Min** | **40.6%** | **38.1%** | **48.9%** | **20.8%** | **33.3%** | **43.8%** |

Tableau A.2 – Indices de finesse pour les configurations testées

par simulation et calculons son *indice de finesse* défini comme le ratio du pire cas obtenu sur la borne calculée. Plus cet indice est proche de 100%, plus la borne est fine. L'indice de finesse calculé est donc une borne inférieure de la finesse réelle, puisque le pire cas obtenu par simulation peut être inférieur au pire cas réel.
L'étude de sensibilité conduite a mis en évidence que la taille des mémoires tampon et le débit des flux ont un impact déterminant sur la borne calculée. Ainsi, nous considérons des valeurs différentes pour ces deux paramètres lors de l'étude de finesse : mémoires tampon de 4, 8 et 16 flits, débits de 8% et 32% pour une taille de paquet de 16 flits.
Nous utilisons Noxim [50] pour effectuer les simulations et présentons les résultats obtenus dans le tableau A.2. Sur les différentes simulations, l'indice de finesse moyen mesuré atteint 80%. Pour un débit de 8%, l'indice de finesse moyen se situe entre 70.1% et 80.8%, tandis que pour un débit de 32%, il varie entre 49.7% et 79.8%.
Nous observons également que la valeur moyenne de l'indice de finesse augmente avec la taille des mémoires tampons. Ce dernier point est cohérent avec les résultats de l'étude de sensibilité. En effet, les configurations de blocage indirect prédites par le modèle tendent à devenir plus simples à mesure que l'indice d'étalement diminue (donc que la taille des mémoires tampon augmente). Ainsi, le léger pessimisme introduit par le calcul de la latence de blocage indirect est d'autant moindre que le set de blocage indirect est petit. De plus, les scenarii de blocage indirect les plus complexes et défavorables sont également les moins susceptibles de se produire sur un ensemble d'offsets aléatoires, car ils supposent une synchronisation fine des différents flux entre eux.

**Scalabilité :** Nous mesurons le temps nécessaire pour l'analyse complète sur diverses configurations comprenant un nombre de flux variable. Pour ce faire, nous considérons un réseau sur puce $8 \times 8$, et un nombre de flux prenant les valeurs 4, 8, 16, 32, 48, 64, 80, 96 et 128. Nous générons au hasard 20 configurations par valeur du nombre de flux $N$, en choisissant $2N$ couples d'entiers $(x, y)$ avec

les coordonnées tirées uniformément dans l'intervalle idoine (ici $[0, 7]$). $N$ couples sont utilisés pour les sources des flux, et les $N$ autres pour les destinations. Tous les autres paramètres (mémoire tampon, taille de paquet, débit des flux, latences technologiques des routeurs) sont fixés.

Pour chaque configuration, nous nous intéressons à la durée totale de l'analyse, $\Delta t$, ainsi qu'à la durée de l'analyse de blocage indirect, $\Delta t_{IB}$, et à la durée du calcul des courbes de service, $\Delta t_{e2e}$.

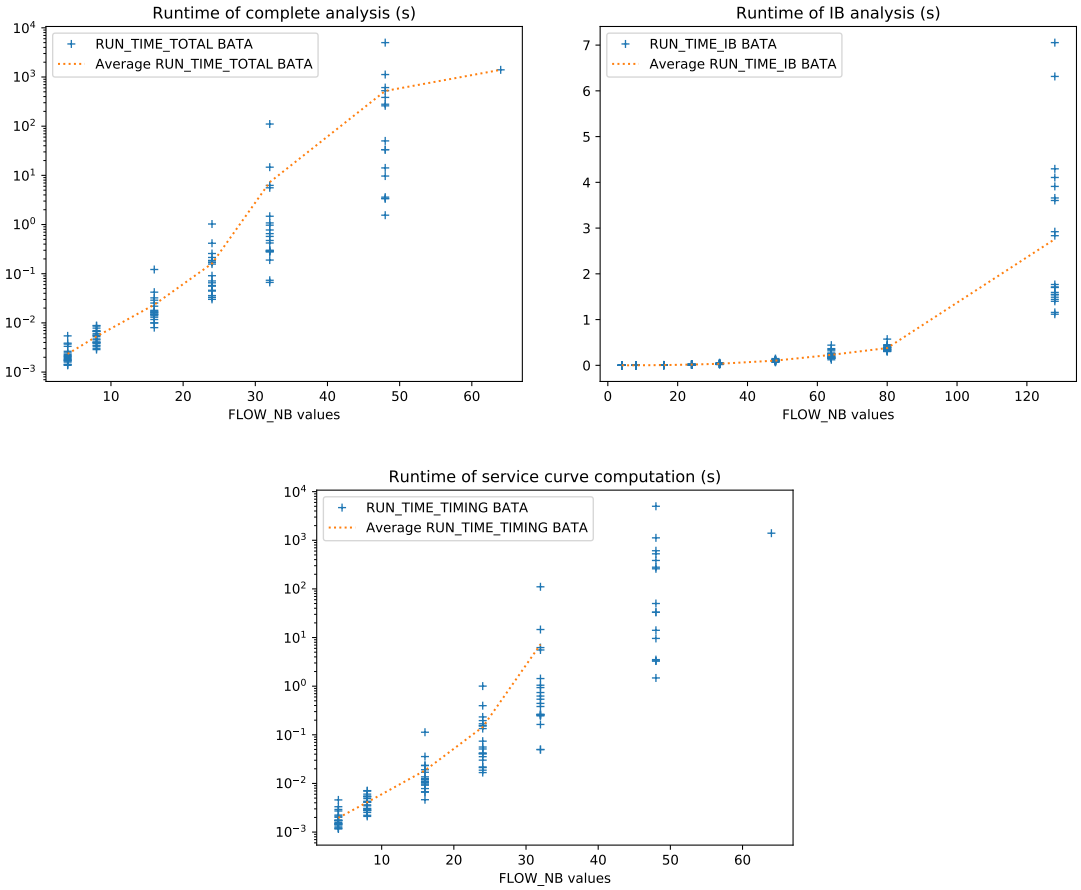



FIGURE A.6 – Résultats de l'analyse de scalabilité

Nous obtenons les résultats présentés en figure A.6. Le graphique en haut à gauche représente la durée de l'analyse totale en fonction du nombre de flux, celui en haut à droite concerne l'analyse de blocage indirect uniquement, et le dernier isole la durée du calcul des courbes de service. Chaque point représente la durée de l'analyse pour une configuration. Nous calculons aussi les valeurs moyennes de l'analyse pour un nombre de flux donné et relions les valeurs par une courbe en pointillés.

Cette étude montre dans un premier temps que la principale source de complexité est le calcul des courbes de service. Pour des configurations de 64 flux et plus, l'analyse totale nécessite plus de deux heures. Au vu de l'évolution du temps total d'analyse avec le nombre de flux, on peut conjecturer que le calcul a une complexité exponentielle. A contrario, l'analyse de blocage indirect est relativement rapide par rapport à l'analyse totale (au maximum 7 secondes pour des configurations de 128 flux), et son évolution confirme la borne calculée section 4.4.

### A.4.5 Conclusion

Ce premier modèle permet de déterminer des bornes supérieures du délai pire cas de transmission de flux CBR sur un réseau sur puce homogène, en prenant en compte la taille des mémoires tampon des routeurs. L'indice de finesse du modèle atteint jusqu'à 80% en moyenne, mais la méthode montre une certaine limite dans sa capacité à passer à l'échelle. Ainsi, pour les configurations testées comprenant plus de 64 flux, la durée nécessaire à l'analyse dépasse les deux heures. De plus, le seul trafic supporté est de type CBR, et les architectures sont supposées homogènes. Par la suite, nous cherchons donc à étendre ce modèle pour prendre en compte plus de types de trafic différents, et améliorer ses performances calculatoires afin d'analyser des configurations plus complexes en une durée moindre.

## A.5 Analyse temps réel des NoCs wormhole hétérogènes par graphe d'interférences

La deuxième contribution de cette thèse est de proposer une extension du modèle précédent. Cette extension concerne (i) le modèle de trafic pris en compte dans l'analyse ; (ii) les architectures sur lesquelles est applicable le modèle ; et (iii) le calcul de la latence de blocage indirect. Ce dernier point repose sur l'utilisation sur des graphes d'interférence ; aussi nous baptisons l'approche G-BATA (**G**raph-based BATA). Nous soulignons les principales extensions dans la modélisation du système (section A.5.1). La section A.5.2 présente notre nouvelle approche pour l'analyse de blocage indirect. Enfin, nous donnons un aperçu des résultats de l'analyse de performance du modèle.

### A.5.1 Formalisme étendu

Cette partie diffère légèrement du formalisme de BATA. Nous étendons le modèle de trafic pour couvrir les flux *bursty*. En plus des caractéristiques précédemment

mentionnées (taille de paquet $L_f$, période $P_f$, jigue $J_f$), chaque flux $f$ est caractérisé par un *burst* noté $b_f$, correspondant au nombre maximal de paquets qu'il peut générer consécutivement. Sa courbe d'arrivée s'exprime alors ainsi :

$$\begin{aligned} \alpha_f &= \sigma_f + \rho_f t \\ \text{avec : } \rho_f &= \frac{L_f}{P_f} \\ \sigma_f &= b_f \cdot L_f + J_f \cdot \rho_f \end{aligned}$$

Pour la partie réseau, la différence majeure est que la courbe de service d'un élément de réseau dépend du nœud considéré et les tailles des mémoires tampon ne sont plus supposées uniformes. Ainsi, on notera $R^r$ la capacité de traitement du nœud $r$, $T^r$ la latence technologique du nœud $r$, et $B^r$ la taille de la mémoire tampon.

Puisque le réseau n'est plus considéré comme homogène, l'indice d'étalement d'un flux dépend du nœud à partir duquel il est calculé, et sa définition doit donc être revue.

**Definition 19.** Indice d'étalement étendu
*Considérons un flux $f$ de taille de paquet maximale $L_f$. L'indice d'étalement étendu de $f$ au nœud d'indice $i$, noté $N_f^i$ est défini comme suit :*

$$N_f^i = \min_{l \geq 0} \left\{ l, L_f \leq \sum_{j=0}^{l-1} B^{\mathbb{P}_f[i+j]} \right\}$$

*où $B^r$ désigne la taille de la mémoire tampon au nœud $r$.*

Nous étendons ensuite la définition du sous-chemin de $k$ relativement à $l$ pour intégrer l'indice d'étalement étendu. La définition détaillée se trouve section 5.2.2, définition 12 et est très similaire à celle présentée dans la section précédente.

### A.5.2 Définition et construction du graphe d'interférence

Afin de capturer la particularité du trafic bursty, nous définissons une structure de graphe orienté, telle que chaque sommet correspond à un sous-chemin représentant l'étalement d'un paquet d'un certain flux dans le réseau. Les sommets sont connectés entre eux par des arêtes orientées représentant les dépendances d'un sous-chemin par rapport à un autre. Ainsi, une arrête d'un sommet $a$ vers un sommet $b$ signifie que le sous-chemin du sommet $a$ a été calculé relativement au sous-chemin de $b$.

Comme le trafic bursty autorise que plusieurs paquets consécutifs d'un même flux se suivent dans le réseau, il est possible de calculer le sous-chemin d'un flux $k$ par rapport à un autre sous-chemin de ce même flux $k$. Notons que ce dernier point permet également de modéliser un scénario dans lequel un paquet d'un flux CBR est retardé suffisamment pour que le paquet suivant le « rattrape », voire se retrouve bloqué derrière le premier.
Formellement, un sommet $v$ possède donc les attributs suivants :

- **fkey** : l'indice du flux ;
- **path** : le sous-chemin de ce flux ;
- **dependencies** : la liste des arêtes $(v, u)$ où $u$ est le sommet tel que $v.path$ est le sous-chemin de $v.fkey$ relativement à $u.path$ ;
- **dependents** : la liste des arêtes $(w, v)$, où $w$ est le somment tel que $w.path$ est le sous-chemin de $w.fkey$ relativement au sous-chemin $v.path$.

Pour construire le graphe (*cf* section 5.3, algorithmes 3 et 4 pour les détails), l'on construit d'abord un sommet dit « racine » correspondant au flux d'intérêt sur son chemin total, duquel ne part aucune arête. Puis, pour chaque flux $k \in \mathcal{F}$, on calcule le sous-chemin de $k$ par rapport à $f$. Si ce sous-chemin est non vide, on crée un sommet qui contient le sous-chemin calculé et une arête vers le sommet de $f$. On ajoute ce sommet au graphe (s'il n'y est pas déjà). On itère ensuite cette approche sur les nouveaux sommets obtenus. L'algorithme termine lorsqu'aucun nouveau sommet n'est créé.
Nous déterminons, section 5.3, propriété 2, une borne supérieure de la complexité de ce nouvel algorithme.

Une fois le graphe construit, on extrait le set de blocage indirect correspondant en faisant la liste des sommets du graphe, et en éliminant le sommet racine et les sommets correspondant à des flux qui partagent des ressources avec le flux d'intérêt.

Le reste de l'analyse (courbe de service et délai de bout en bout) est similaire, à une légère différence près pour le calcul de $T_{IB}$. Au lieu de calculer la courbe d'arrivée propagée au début de chaque sous-chemin du set de blocage indirect, nous prenons à cet endroit la courbe d'arrivée initiale pour un burst d'un paquet ($b = 1$). Cette modification entraîne en particulier une réduction du nombre de calculs de courbes de service intermédiaires, que nous ne détaillons pas ici. Cet aspect est développé section 5.4 (voir en particulier l'algorithme 5) et ses conséquences seront observées dans l'analyse de performance.

### A.5.3 Analyse de performance

L'analyse de performance de G-BATA suit un processus similaire à celui de BATA. Nous proposons ici un résumé des principaux résultats de cette analyse, selon les trois axes présentés section A.4.4. Pour l'analyse de sensibilité et l'évaluation de la finesse, nous utilisons les mêmes configurations et valeurs de paramètres.

**Sensibilité :** La principale différence observée concerne l'évolution des délais de bout en bout avec les tailles des mémoires tampon. En raison de l'hypothèse supplémentaire sur le trafic, le nombre de sous-chemins consécutifs du même flux n'est pas limité. La diminution des longueurs des sous-chemins entraîne une augmentation de leur nombre et, en conséquence, $T_{IB}$ et les délais de bout en bout augmentent également. En revanche, on assiste toujours à un phénomène de palier au-delà d'une taille de mémoire tampon permettant de contenir en entier un paquet de n'importe quel flux.

L'autre aspect intéressant est que G-BATA est moins sensible au débit des flux que BATA. Cela est dû au fait qu'avec l'approche BATA, pour un flux d'intérêt $f$, et pour $\{k, \mathbb{S}\} \in IB_f$, on calcule la courbe d'arrivée de $k$ propagée au début du sous-chemin $\mathbb{S}$, alors qu'avec G-BATA, on utilise la courbe d'arrivée initiale de $k$ pour un burst égal à un paquet. Or, pour les types de courbes utilisés, l'expression du burst de la courbe d'arrivée propagée contient un terme supplémentaire $\rho T$ (*cf* théorème 6, annexe B.1), qui se retrouvera dans la borne sur le délai de bout en bout. Pour plus de détails, nous renvoyons la lectrice vers la section 5.6.2.

**Finesse :** Nous évaluons la finesse de G-BATA sur les mêmes configurations que BATA. Les résultats sont résumés dans le tableau 5.1. En moyenne, l'indice de finesse de G-BATA est de 72% pour des mémoires tampon de 4 flits et de 56% pour des mémoires tampon de 16 flits. Pour des flux de débit 8% et des mémoires tampon de 4 flits, les deux modèles donnent des résultats similaires. Pour un débit de 32%, G-BATA donne des bornes plus fines. Au contraire, lorsque la mémoire tampon est plus grande, BATA donne des bornes plus fines.

**Scalabilité :** Nous évaluons les performances de G-BATA d'abord avec les mêmes configurations utilisées pour BATA dans la section précédente. Les courbes comparatives montrent que G-BATA est bien plus performant : par exemple, la durée moyenne de l'analyse totale de configurations contenant 48 flux est 766 fois moindre avec G-BATA. Le calcul des courbes de services est l'étape où G-BATA montre pleinement son avantage sur BATA. En revanche, on constate que l'analyse du blocage

indirect est en moyenne 5.7 fois plus rapide avec BATA qu'avec G-BATA.
Puis, nous évaluons les performances de G-BATA sur des configurations plus grosses (de 20 à 800 flux). L'approche reste performante à grande échelle, puisque sur un ordinateur portable équipé d'un processeur Intel core i7, le calcul de toutes les bornes des configurations contenant 800 flux prend autour de deux heures au maximum (soit environ 9 secondes par flux). Nous observons également que l'analyse du blocage indirect représente la majorité du coût de l'analyse totale (97.1% de la durée totale en moyenne pour les configurations de 800 flux).

### A.5.4 Conclusion

Notre modèle d'analyse temporelle pire cas basé sur des graphes d'interférences permet de traiter des architectures hétérogènes et de prendre en compte un modèle de trafic plus général que le précédent, tout en gardant une bonne finesse sur les bornes calculées. En outre, il permet d'obtenir des résultats en 10 à 100 fois moins de temps que BATA pour des configurations de 32 et 48 flux. Le calcul des bornes pire cas sur des configurations de 800 flux prend moins de 2h sur un ordinateur portable, soit 9 secondes par flux, ce qui valide la possibilité d'utiliser le modèle à grande échelle.
Comme résumé en figure 5.15, il convient d'utiliser G-BATA lorsque les hypothèses nécessaires à l'application de BATA ne sont pas vérifiées ou lorsqu'on cherche à obtenir des résultats rapidement. En l'absence de telles contraintes, l'on peut utiliser BATA à condition qu'il n'y ait pas plusieurs paquets consécutifs d'un même flux dans le réseau.

## A.6 Approche hybride pour l'exploration d'architectures

Nous nous intéressons maintenant aux problèmes liés à l'exploration d'architectures. Comme mentionné précédemment, la plupart des approches considérées n'intègrent pas la vérification de contraintes temps réel dans leur processus de conception. L'objet de cette section est de proposer une méthodologie prenant en compte cet aspect.
À cette fin, nous proposons un processus hybride respectant l'approche Y-chart, bâti autour d'un outil existant, TTool [9]. Ce processus de conception et d'exploration étendu nécessite l'ajout d'un modèle de réseau sur puce dans l'outil TTool, ainsi qu'une liaison avec notre modèle d'analyse G-BATA, que nous intégrons à l'outil

WoPANets, destiné à la modélisation et l'analyse temporelle pire cas de réseaux. Nous évaluons ensuite l'intérêt de notre méthode sur un exemple simple montrant le gain de temps par rapport à une étude du système basée uniquement sur la simulation transactionnelle.

### A.6.1 Workflow étendu

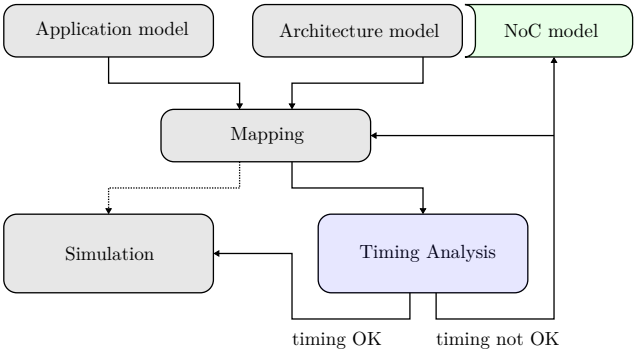



FIGURE A.7 – Processus de l'approche hybride

La figure A.7 montre le processus de conception étendu pour intégrer la vérification des contraintes temps réel.
La modélisation du système sépare l'aspect architectural de l'aspect fonctionnel. À ce stade, le composant réseau sur puce est utilisé comme une boîte noire pour la description de l'architecture. Une fois un mapping choisi, on peut exporter le modèle sous format XML vers WoPANets pour calculer les bornes sur le délai de bout en bout de chacun des flux.
Si un mapping ne permet pas de satisfaire les contraintes temps réel, il est ignoré et on peut alors agir sur l'architecture ou le mapping pour proposer une autre possibilité. Si, au contraire, le mapping satisfait les contraintes, l'on peut simuler le système.
Pour cela, on transforme d'abord le NoC en ses sous-composants et tâches, avant de générer et compiler le code de simulation. Ces fonctionnalités sont implémentées dans TTool. Le reste du processus est semblable à l'approche Y-chart classique.

### A.6.2 Modélisation système du NoC

Pour ajouter un composant NoC dans TTool, nous définissons ses deux sous-composants : le routeur et l'interface réseau, selon l'architecture résumée en figure A.8. Les routeurs gèrent les paquets flit par flit et s'interconnectent entre eux pour former la topologie du NoC. Les interfaces réseau permettent de lier des émetteurs

et récepteurs de trafic qui ne manipulent que des paquets avec les routeurs du NoC, et ainsi d'autoriser la connexion d'un NoC à des composants existants sous TTool par l'intermédiaire d'un bus. L'interface réseau d'entrée se charge d'injecter les paquets dans le NoC et d'arbitrer entre les différents émetteur. L'interface réseau de sortie lit les flits qu'elle reçoit et notifie le récepteur lorsqu'un paquet complet est disponible.

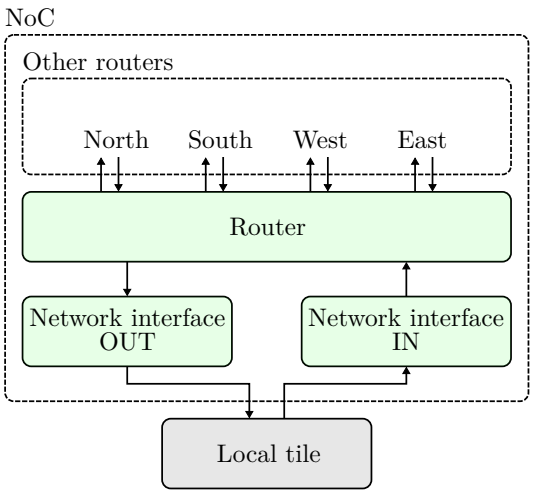



FIGURE A.8 – Description fonctionnelle du réseau sur puce

Le modèle de routeur choisi est un routeur avec mémoire tampon en entrée supportant les canaux virtuels. Nous le scindons en plusieurs blocs fonctionnels (figure A.9), chargés en particulier de répartir les flits reçus dans les différentes files d'attente, d'arbitrer entre les différentes entrées et canaux virtuels, et de router le flit vers sa destination. Nous spécifions le fonctionnement de chaque bloc fonctionnel grâce à un *diagramme d'activité* (non détaillé ici). Une telle organisation modulaire permet de proposer une modification d'un aspect du router (*e.g.* politique d'arbitrage ou algorithme de routage) sans devoir repenser l'intégralité de la structure.
Comme le modèle de données de TTool ne permet pas de spécifier leur valeur, nous réalisons le contrôle de flux grâce à des événements. Au sein d'un routeur et entre ceux-ci, un événement correspond à un flit ou un crédit de contrôle de flux, tandis qu'en dehors du réseau sur puce, un événement correspond à un paquet. Les événements correspondant à des paquets contiennent les informations relatives à sa destination, sa taille, ainsi qu'un identifiant unique servant à distinguer les flux.

L'architecture matérielle que nous proposons est très similaire à l'architecture fonctionnelle, puisque chaque tâche au sein du routeur doit s'exécuter sans concurrence.

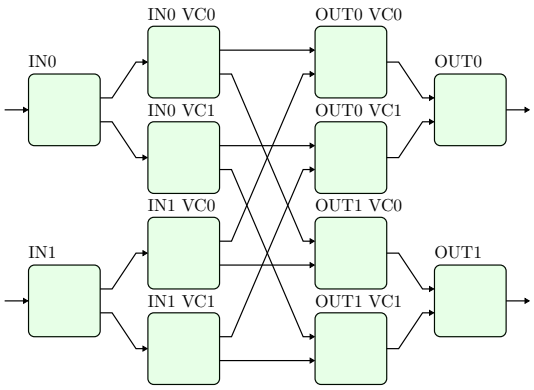



FIGURE A.9 – Description fonctionnelle d'un routeur 2 entrées, 2 sorties, 2 canaux virtuels

### A.6.3 Performance

Nous proposons un exemple simple pour mettre en évidence les capacités de modélisation de notre approche et évaluer le gain qu'elle offre. La configuration considérée est schématisée en figure A.10. Elle comprend trois flux sur une plateforme $2 \times 2$.

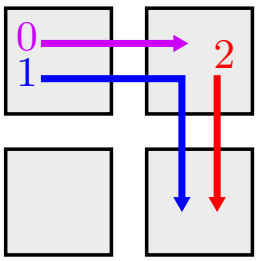



FIGURE A.10 – Configuration considérée

La vue fonctionnelle d'un flux, décrite en figure A.11, est constituée d'une tâche émettrice, d'une tâche réceptrice, et d'une tâche de contrôle chargée de déclencher l'envoi et la réception périodiques d'un paquet. Les diagrammes d'activité correspondant sont représentés en figure A.12.
Le modèle d'architecture et le mapping des tâches est présenté en figure A.13.
La courbe de service d'un routeur du réseau sur puce considéré est la suivante :

$$\beta(t) = \frac{1}{13}(t-5)^{+}$$

Pour mesurer le délai obtenu par simulation pour chaque flux, nous traitons automatiquement les traces de simulation pour extraire l'instant où le premier flit d'un paquet est injecté dans le réseau et l'instant ou le dernier flit du même paquet quitte le réseau.

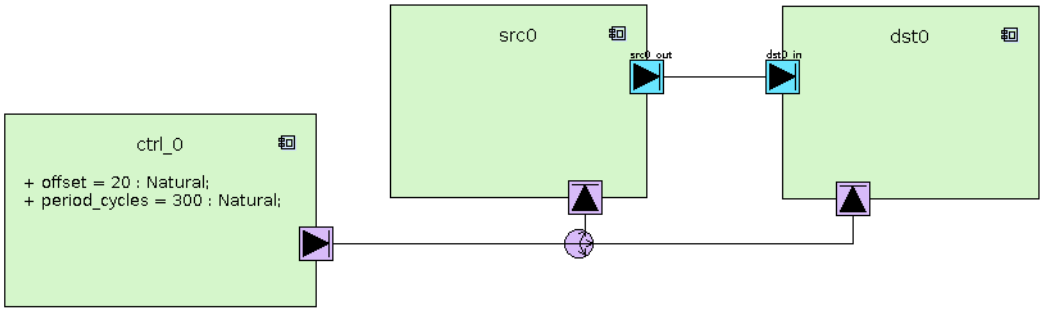
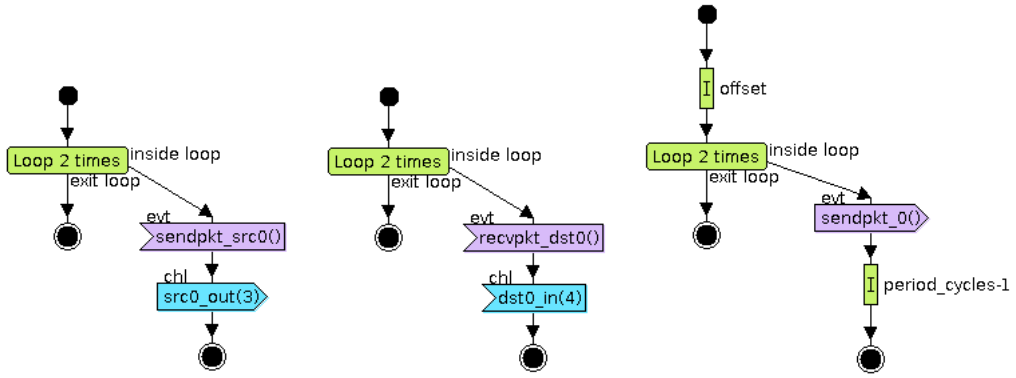



FIGURE A.11 – Vue fonctionnelle




(a) TX task (b) RX task (c) Control task

FIGURE A.12 – Diagrammes d'activité

Nous simulons l'envoi et la réception de 2 paquets par flux et comparons les résultats obtenus aux bornes pire cas calculées (tableau A.3).

| | Flux 0 | | Flux 1 | | Flux 2 | |
|---|---|---|---|---|---|---|
| Délai de bout en bout mesuré (cycles) | 62 | 62 | 123 | 123 | 62 | 62 |
| Borne sur le délai pire cas (cycles) | 199 | 199 | 455 | 455 | 288 | 288 |

Tableau A.3 – Résultats de simulation

A priori le flux 1 ne respecte pas sa contrainte temporelle (borne pire cas supérieure à sa période). On constate de plus que le délai mesuré pour les flux 0 et 2 est égal au temps de transmission d'un paquet sans congestion, ce qui indique que lors du scénario simulé, ces flux n'ont pas subi de congestion.
Pour varier les scenarii de transmission, nous effectuons une autre série de simulations en décalant aléatoirement l'instant d'envoi du premier événement de contrôle de chaque flux par rapport à l'instant 0. Nous présentons la distribution des délais

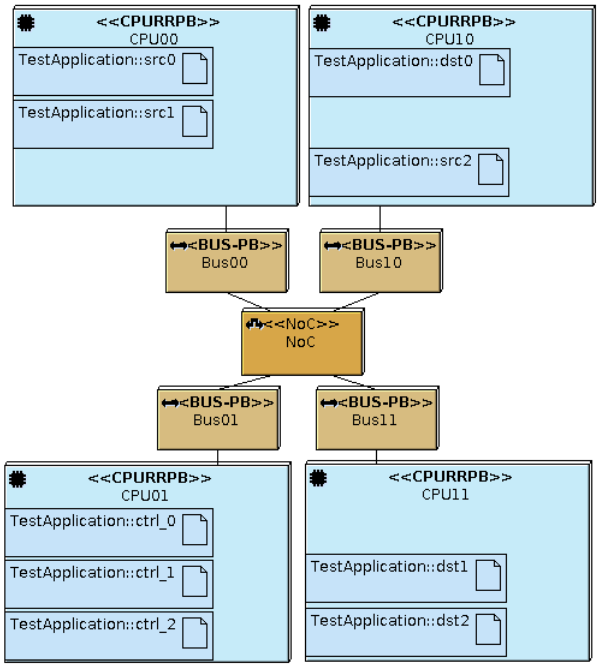


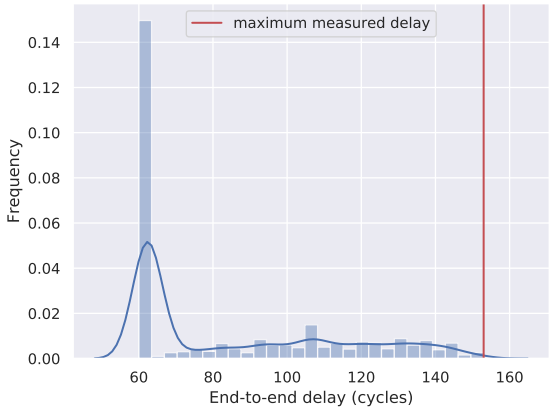

FIGURE A.13 – Architecture et mapping

mesurés sur les graphiques de la figure A.14. On constate que statistiquement, les flux 0 et 2 sont moins susceptibles de subir de la congestion que le flux 1. En revanche, la simulation n'a pas été en mesure d'exhiber un scénario où le flux 1 ne respecte pas sa contrainte temporelle. À titre indicatif, la génération, compilation et exécution du code de simulation pour les 3000 scenarii testés a duré environ 44 minutes, alors que l'analyse pire cas n'a nécessité que 602 $\mu$s. On peut donc faire deux remarques sur cet exemple :

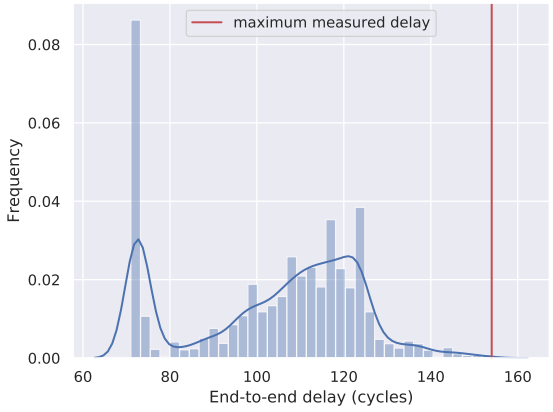
- Si les offsets des flux sont connus ou contraints d'une certaine façon, la simulation permet de raffiner les résultats de l'analyse temporelle et peut aider le concepteur a identifier les scenarii de blocage qui se produisent le plus pour tenter de les éviter.
- Si au contraire les offsets ne sont pas connus ou ne peuvent pas être contrôlés, simuler même une configuration aussi simple peut prendre du temps et ne pas être en mesure de prouver que les contraintes temporelles sont respectées.

### A.6.4 Conclusion

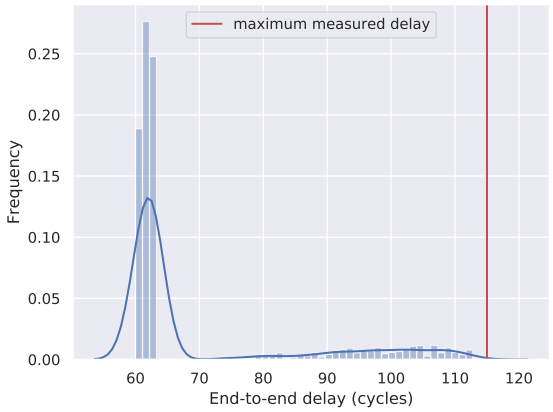
L'approche hybride que nous présentons combine le calcul réseau et la simulation dans un processus d'exploration d'architecture compatible avec le Y-chart.




(a) Flux 0




(b) Flux 1




(c) Flux 2

FIGURE A.14 – Distribution des délais mesurés

Nous proposons une implémentation de cette méthodologie sur la base des outils TTool et WoPANets. Une telle implémentation s'attache à rester aussi modulaire que possible afin de permettre de futures améliorations.
Enfin, nous mettons en évidence l'intérêt d'intégrer une méthode formelle d'analyse pire cas dans l'exploration d'architecture sur un exemple simple.

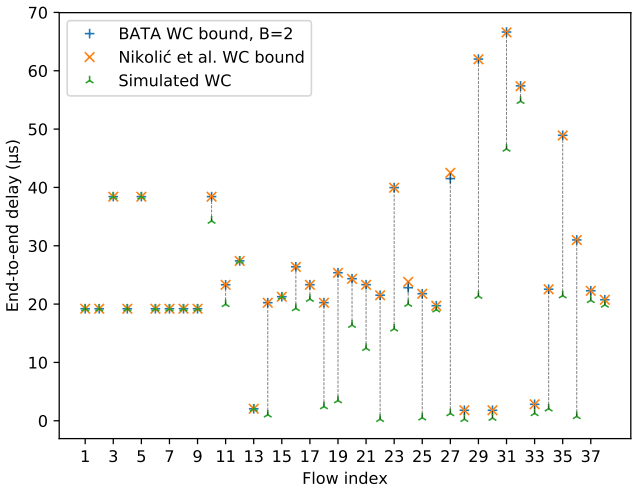
## A.7 Validation des contributions

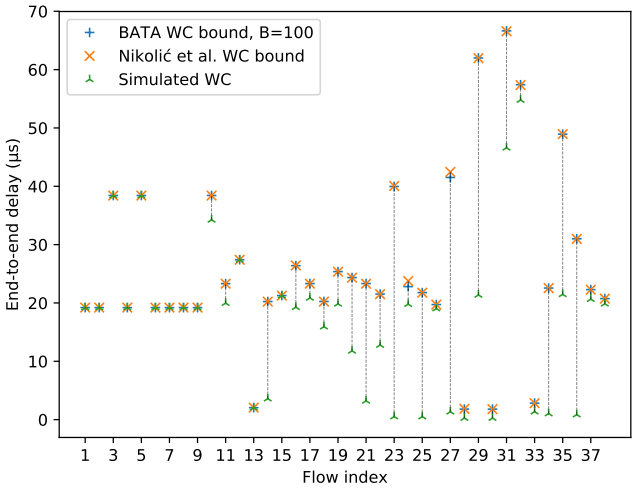
Pour démontrer l'applicabilité de nos contributions, nous avons d'abord effectué des expériences sur une puce pluri-cœurs pour mesurer les délais de bout en bout de flux de données et les comparer aux prédictions du modèle. Puis nous avons appliqué notre approche d'exploration d'architecture à un cas d'étude. Nous proposons ici de détailler uniquement l'étude de cas portant sur l'application de contrôle d'un véhicule autonome, utilisée en particulier dans [14, 42].

### A.7.1 Analyse pire cas

Nous analysons dans un premier temps la configuration avec les même caractéristiques que dans [42] à des fins de comparaison. Le mapping des 33 tâches ainsi que les caractéristiques des 38 flux sont détaillés Annexe C. L'architecture contient un réseau sur puce $4 \times 4$ et l'on suppose que les canaux virtuels ne sont pas partagés. La latence technologique des routeurs est de 3 cycles, la capacité des liens est d'un flit par cycle, et un cycle dure 0.5 ns. Nous calculons les bornes sur les délais pire cas pour une taille de mémoire tampon de 2, 100 et 1000000 de flits (cette dernière étant suffisamment grande pour considérer la mémoire tampon comme infinie).

Comme les flux ne partagent pas de canaux virtuels, il n'y a pas de blocage indirect. Ainsi, BATA et G-BATA doivent donner les mêmes résultats, ce qui est bien le cas. Nous traçons les graphes comparatifs entre les résultats de notre approche et ceux de [42] figure A.15. L'indice de finesse moyen des bornes calculées par G-BATA pour des mémoires tampon de 2 flits (resp. 100, infinie) est de 64% (resp. 67%, 71%). Nous renvoyons au tableau C.3, annexe C, pour le détail des indices de finesse de tous les flux.
Notre approche donne des résultats très similaires à ceux de [42]. En moyenne, la différence entre l'indice de finesse de notre approche et de celle de [42] se situe entre -0.03% et +0.08%.



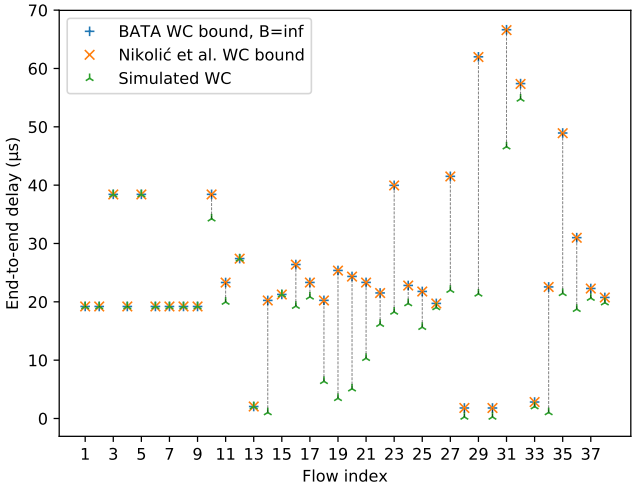




FIGURE A.15 – Comparatif des bornes sur le délai de bout en bout

Toutefois, notre approche permet en sus d'analyser la même configuration sans l'hypothèse de non-partage des canaux virtuels. Nous avons donc refait les calculs pour une plateforme avec 2, puis un VC. Les résultats, détaillés section 7.2, montrent que les contraintes temporelles sont toujours respectées dans ces deux cas.

### A.7.2 Modélisation sous TTool

Nous modélisons maintenant le système pour des mémoires tampon de 2 flits avec notre approche. Nous considérons une plateforme avec un seul canal virtuel et réduisons les tailles de paquets et les périodes des flux de sorte à conserver le même débit pour chaque flux.
Comme en A.6.3, nous utilisons un modèle d'application composé d'émetteurs, de récepteurs et de tâches de contrôle, que nous ne détaillons pas ici.
Notons simplement que certaines tâches sont à la fois émettrices et réceptrices d'un ou plusieurs flux. Ainsi, certains événements de contrôle sont mutualisés, et il n'y a que 13 tâches de contrôle pour 38 flux.

Le modèle d'architecture de la plateforme est présenté en figure A.16. Le mapping est le même que celui de l'étude de cas originale. Pour éviter l'interférence des tâches de contrôle avec les autres tâches, nous les exécutons sur des CPUs dédiés.
Pour la simulation, nous introduisons un offset aléatoire dans chaque tâche de contrôle, afin d'obtenir des scenarii de transmission différents. Nous effectuons 300 simulations de 500000 cycles chacune.
Pour chaque flux, nous normalisons les valeurs mesurées pour le délai de bout en bout par transformation affine, pour obtenir une valeur entre 0 (correspondant à la latence de transmission minimale, sans congestion) et 1 (correspondant à la borne pire cas). Nous pouvons ainsi représenter la distribution des délais de bout en bout pour tous les flux. Le graphique correspondant est présenté en figure A.17.
Statistiquement, on constate que les délais sont proches des latences de transmission minimales. Cela suggère que la plupart du temps, les paquets ne subissent pas de congestion. Cette conclusion n'est pas surprenante puisque les flux considérés ont des débits très faibles (0.00048 flits par cycle au plus).
Une telle observation est également cohérente avec le fait que les bornes sur les délais pire cas sont de l'ordre de 280 fois moindres que les périodes des flux (si ce n'est plus).

Notons également qu'en termes de performance, 2 millions de cycles de simulation

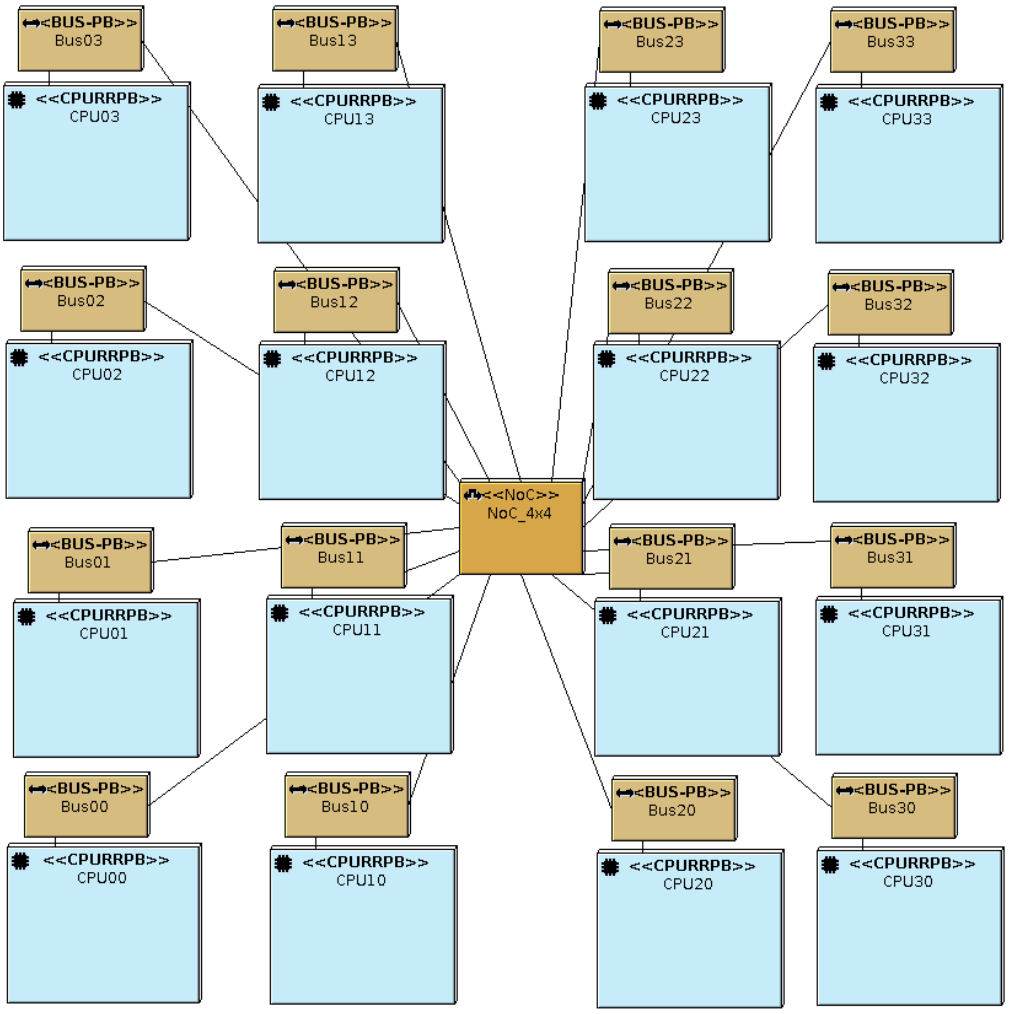



FIGURE A.16 – Architecture de la plateforme

nécessitent environs 53 minutes de calcul, contre moins d'une seconde pour l'analyse formelle correspondante.
Ce dernier point conforte l'intérêt d'utiliser une approche hybride pour l'exploration d'architecture.

## A.8 Conclusion

### A.8.1 Résumé des contributions

Nos contributions concernent deux domaines : l'analyse temporelle pire cas des réseaux sur puce avec routage chenille et canaux virtuels implémentant des classes de priorité fixe, et l'exploration d'architectures avec réseau sur puce pour les

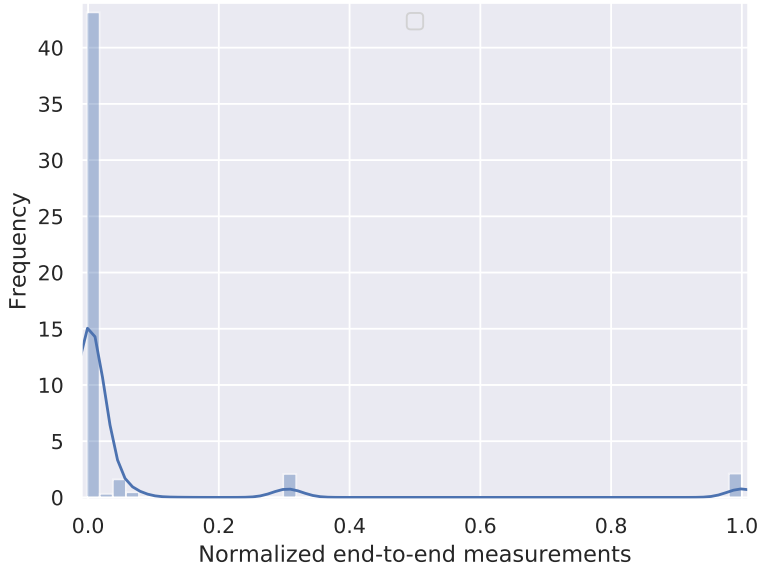



FIGURE A.17 – Distribution des délais de bout en bout normalisés

applications temps réel.

Dans la première partie, nous proposons une première approche, BATA, prenant en compte l'impact de la taille des mémoires tampon et la sérialisation des flux de données, et permettant de calculer des bornes sur le délai de bout en bout de chaque flux dans le pire cas, pour des flux CBR. Cette approche permet d'obtenir des bornes avec une bonne finesse mais son point faible est sa capacité à passer à l'échelle.
Nous étendons donc cette approche en proposant G-BATA, pour modéliser en sus les architectures hétérogènes et couvrir un type de trafic plus général (*bursty*). G-BATA utilise des graphes pour modéliser les interférences entre flux, permettant d'accélérer le calcul des bornes pire cas d'un facteur pouvant atteindre $10^2$ dès quelques dizaines de flux. Il offre également une finesse similaire à celle de son prédécesseur.

Dans une seconde partie, nous présentons un processus de conception et d'exploration d'architecture hybride, intégrant notre approche d'analyse temporelle pire cas. Cette méthodologie est compatible avec le Y-chart et vise à réduire l'espace à explorer en éliminant dès que possible les configurations qui ne satisfont pas les contraintes temporelles.

Nous implémentons notre approche sur la base d'outils existants, TTool et WoPANets, auxquels nous ajoutons les fonctionnalités nécessaires.
Nos exemples mettent en évidence le gain de temps de calcul résultant de l'intégration d'une méthode formelle dans le processus de conception.

Enfin, nous proposons une validation de nos contributions à travers une série d'expériences sur une puce pluri-cœurs et l'étude d'une application de contrôle d'un véhicule autonome.

### A.8.2 Perspectives

Les perspectives de nos travaux concernent deux aspects : les modèles et approches présentés, et l'outillage développé. Nous présentons ici une perspective de chaque aspect et renvoyons la sagace lectrice à la conclusion générale du chapitre 8 pour plus de détails.

Notre méthodologie d'exploration d'architectures utilise l'analyse temporelle pore cas pour obtenir une réponse binaire sur le respect des contraintes temporelles. Il serait possible d'utiliser les résultats de l'analyse de sensibilité pour proposer des modifications de l'architecture proposée en cas d'impossibilité de satisfaire les contraintes temporelles.

En matière d'outillage, WoPANets utilise un import basique des fichiers XML issus de TTool pour générer un modèle et l'analyser. En particulier, il est nécessaire d'avoir préalablement calculé les périodes des flux et de les avoir intégrées dans le modèle du système réalisé avec TTool sous la forme d'attributs dans les composants de la vue fonctionnelle. Pour améliorer l'interopérabilité des logiciels utilisés, il serait intéressant de proposer un outil plus puissant, intégré à WoPANets, permettant de faire le lien entre la vue « système » d'une application et la vue « réseau » des flux de données générés par l'application. Le problème sous-jacent concerne le lien entre deux représentations d'une même réalité physique et dépasse largement le développement d'un outil logiciel.

*Lemons and limes on a poster*
*Are not enough to impress you*
*Nevermind – can you look closer*
*And tell me if you see "you" through?*

*
* *

Appendix B

# Network Calculus Memo

---

*Il fallait, c'est compliqué,*
*Maîtriser les seaux percés*
—J.-Y. Le Boudec

## B.1 Basics

Network Calculus describes data flows by means of cumulative functions, defined as the number of transmitted bits during the time interval $[0, t]$. Consider a system $S$ receiving input data flow with a Cumulative Arrival Function (CAF), $A(t)$, and putting out the same data flow with a Cumulative Departure Function (CDF), $D(t)$. To compute upper bounds on the worst-case delay and backlog, we need to introduce the maximum arrival curve, which provides an upper bound on the number of events, e.g., bits or packets, observed during any interval of time.

**Definition 20.** *(Arrival Curve)[69] A function $\alpha$ is an arrival curve for a data flow with the CAF A, iff:*

$$\forall t, s \geq 0, s \leq t, A(t) - A(s) \leq \alpha(t - s)$$

A widely used curve is the leaky bucket curve, which guarantees a maximum burst $\sigma$ and a maximum rate $\rho$, *i.e.*, the traffic flow is $(\sigma, \rho)$-constrained. In this case, the arrival curve is defined as $\gamma_{\sigma,\rho}(t) = \sigma + \rho \cdot t$ for $t > 0$. Furthermore, we need to guarantee a minimum offered service within crossed nodes through the concept of minimum service curve.

**Definition 21.** *(Simple Minimum Service Curve)[69] The function $\beta$ is the simple service curve for a data flow with the CAF A and the CDF D, iff:*

$$\forall t \geq 0, D(t) \geq \inf_{s \leq t}(A(s) + \beta(t - s))$$

A very useful and common model of service curve is the rate-latency curve $\beta_{R,T}$, with $R$ the minimum guaranteed rate and $T$ the maximum latency before starting the service. This rate-latency function is defined as $\beta_{R,T}(t) = [R \cdot (t-T)]^+$, where $[x]^+$ is the maximum between $x$ and 0.
This service curve is easy to define in the case of *one input/output node* serving one or many traffic flows coming from the same source and going to the same destination. Moreover, to model a node implementing aggregate scheduling, *i.e.*, multiplexes the crossing flows at the input and demultiplexes them at the output, we need to define the *left-over service curve* guaranteed to each flow within the crossed node, considering the impact of contention with other traffic flows. The computation of such a left-over service curve depends on the implemented scheduling policy, and its derivation needs strict service curve property in the general case.

**Definition 22.** *(Strict service curve)[69] The function $\beta$ is a strict service curve for a data flow with the CDF $D(t)$, if for any backlogged period* [1] *$]s,t]$, $D(t)-D(s) \geq \beta(t-s)$.*

The main results concerning the left-over service curves computation are as follows:

**Theorem 5.** *(Left-over service curve - Arbitrary Multiplex)[104] let $f_1$ and $f_2$ be two flows crossing a server that offers a strict service curve $\beta$ such that $f_1$ is $\alpha_1$-constrained, then the left-over service curve offered to $f_2$ is:*

$$\beta_2 = (\beta - \alpha_1)_\uparrow$$

*where* $f_\uparrow(t) = \max\{0, \sup_{0\leq s\leq t} f(s)\}$

**Corollary 1.** *(Left-over service curve - FP Multiplex)[105] Consider a system with the strict service $\beta$ and $m$ flows crossing it, $f_1$,$f_2$,..,$f_m$. The maximum packet length of $f_i$ is $l_{i,max}$ and $f_i$ is $\alpha_i$-constrained. The flows are scheduled by the non-preemptive fixed priority (NP-FP) policy, where $f_i \succ f_j \Leftrightarrow i < j$. For each $i \in \{2,..,m\}$, the strict service curve of $f_i$ is given by:*

$$(\beta - \sum_{j<i} \alpha_j - \max_{k\geq i} l_{k,max})_\uparrow$$

Knowing the arrival and service curves, one can compute the upper bounds on performance metrics for a data flow, according to the following theorem.

[1] A backlogged period $]s,t]$ is an interval of time during which the backlog is non null, *i.e.*, $A(s) = D(s)$ and $\forall u \in ]s,t]$, $A(u) - D(u) > 0$

**Theorem 6.** *(Performance Bounds) Consider a flow constrained by an arrival curve $\alpha$ crossing a system $\mathcal{S}$ that offers a service curve $\beta$, then:*
*Delay* [2]*:* $\forall\ t:\ d(t) \leq h(\alpha, \beta)$
*Backlog* [3]*:* $\forall\ t:\ q(t) \leq v(\alpha, \beta)$
*Output arrival curve* [4]*:* $\alpha^*(t) = \alpha \oslash \beta(t)$

In the case of a leaky bucket arrival curve and a rate-latency service curve, the calculus of these bounds is greatly simplified. The delay and backlog are bounded by $\frac{\sigma}{R} + T$ and $\sigma + \rho \cdot T$, respectively; and the output arrival curve is $\sigma + \rho \cdot (T + t)$. Finally, we need the following results concerning the end-to-end service curve of a flow of interest (*foi*) accounting for flows serialization effects in feed-forward networks, based on the Pay Multiplexing Only Once (PMOO) principle [70], under non-preemptive Fixed Priority (FP) multiplexing.

**Theorem 7.** *The service curve offered to a flow of interest $f$ along its path $\mathbb{P}_f$, in a network under non-preemptive FP multiplexing with strict service curve nodes of the rate-latency type $\beta_{R,T}$ and leaky bucket constrained arrival curves $\alpha_{\sigma,\rho}$, is a rate-latency curve, with a rate $R^{\mathbb{P}_f}$ and a latency $T^{\mathbb{P}_f}$, as follows :*

$$R^{\mathbb{P}_f} = \min_{k \in \mathbb{P}_f} \left( R^k - \sum_{i \exists k, i \in shp(f)} \rho_i \right) \tag{B.1a}$$

$$T^{\mathbb{P}_f} = \sum_{k \in \mathbb{P}_f} \left( T^k + \frac{\max\limits_{i \exists k, i \in slp(f)} L_i}{R^k} \right) + \sum_{i \in DB_f \cap shp(f)} \frac{\sigma_i^{\mathrm{cv}(\mathbb{P}_i, \mathbb{P}_f)} + \rho_i \cdot \sum\limits_{k \in \mathbb{P}_f \cap \mathbb{P}_i} \left( T^k + \frac{\max\limits_{i \exists k, i \in slp(f)} L_i}{R^k} \right)}{R^{\mathbb{P}_f}} \tag{B.1b}$$

*where the required notations are defined in Table B.1.*

## B.2 Notations

Hereafter are gathered all notations used throughout this report. As a general rule, upper indexes of a notation $X$ refer to a node or a subset of nodes, while lower indexes refer to a flow. $X_f^r$ means "$X$ at node $r$ for flow $f$".

---

[2] $h(f, g)$: the maximum horizontal distance between $f$ and $g$
[3] $v(f, g)$: the maximum vertical distance between $f$ and $g$
[4] $f \oslash g(t) = \sup_{\forall u \geq 0}\{f(t + u) - g(u)\}$

| Notation | Definition |
|---|---|
| $\mathcal{F}$ | The set of flows on the NoC |
| $S_{flit}$ | The size of one flit |
| $B^r$ | The buffer size at node $r$ |
| $\mathbb{P}_f$ | The list of nodes crossed by $f$ from source to destination |
| $\mathbb{P}_f[k]$ | The $k+1^{th}$ node of $f$ path |
| $subpath(\mathbb{P}_k, \mathbb{P}_l)$ | The subpath of flow $k$ relatively to flow $l$ after $\text{dv}(\mathbb{P}_k, \mathbb{P}_l)$ |
| $Last(\mathbb{P}_k, \mathbb{P}_l)$ | The index of $\text{dv}(\mathbb{P}_k, \mathbb{P}_l)$ in $\mathbb{P}_k$ |
| $\text{cv}(\mathbb{P}_k, \mathbb{P}_l)$ | The convergence node of $\mathbb{P}_k$ and $\mathbb{P}_l$ |
| $\text{dv}(\mathbb{P}_k, \mathbb{P}_l)$ | The divergence node of $\mathbb{P}_k$ and $\mathbb{P}_l$ |
| $f \ni r$ | Flow $f$ crosses node $r$ |
| $F \supset r$ | There is a flow $f \in F$ such that $f \ni r$ |
| $\mathbf{1}_{\{cdt\}}$ | equals 1 if $cdt$ is true and zero otherwise |
| $L_f$ | The maximal packet length of $f$ |
| $J_f$ | The release jitter of $f$ |
| $P_f$ | The period of $f$ |
| $b_f$ | The number of packets in a burst of flow $f$ |
| $\alpha_f(t)$ | The initial arrival curve of $f$ |
| $\alpha_f^r(t)$ | The arrival curve of $f$ at the input of node $r$ |
| $\sigma_f^r$ | The burst of $\alpha_f^r$ |
| $\rho_f^r$ | The rate of $\alpha_f^r$ |
| $\beta_f(t)$ | The end-to-end service curve of $f$ |
| $\widetilde{\beta}_f^{subP}(t)$ | The VC-service curve of $f$ on $subP$ |
| $\widetilde{R}_f^{subP}$ | The rate of $\widetilde{\beta}_k^{subP}$ |
| $\widetilde{T}_f^{subP}$ | The latency of $\widetilde{\beta}_k^{subP}$ |
| $DB_f$ | The set of all flows directly interfering with $f$ |
| $DB_f^{\|path}$ | Flows $i \in DB_f$ such that $\mathbb{P}_i \cap path \neq \varnothing$ |
| $hp(f)$ | Flows mapped to a VC of strict higher priority than $f$ |

| | |
|---|---|
| $sp(f)$ | Flows mapped to the same VC as $f$ |
| $lp(f)$ | Flows mapped to a VC of strict lower priority than $f$ |
| $slp(f)$ | All flows with a priority lower or equal than $f$ |
| $shp(f)$ | All flows with a priority higher or equal than $f$ |
| $IB_f$ | Indirect blocking set of flow $f$ |
| $N_f^i$ | Number of buffers needed to store a packet of $f$ from node $i$ on $\mathbb{P}_f$ |
| $D_f^{\mathbb{P}_f}$ | End-to-end delay bound of $f$ |

Table B.1 – Summary of notations

*Compliqué de te mettre en quatre,*
*Hormis si j'ôte quelques lettres.*

*Remplis les trous si tu veux être*
*Le poète que j'idolâtre.*

*
* *

Appendix C

# Case Study Data

This section details the parameters and characteristics of the case study used in chapter 7, as well as per-flow results of the tightness analysis conducted on the same configuration.

| Task | Core | RX | TX | Description |
|---|---|---|---|---|
| fbu1 | 0,0 | | ✓ | Frame buffer left camera |
| fbu2 | 0,1 | | ✓ | Frame buffer left camera |
| fbu3 | 0,2 | | ✓ | Frame buffer left camera |
| fbu4 | 0,3 | | ✓ | Frame buffer left camera |
| fbu5 | 3,0 | | ✓ | Frame buffer right camera |
| fbu6 | 3,1 | | ✓ | Frame buffer right camera |
| fbu7 | 3,2 | | ✓ | Frame buffer right camera |
| fbu8 | 3,3 | | ✓ | Frame buffer right camera |
| fdf1 | 2,1 | ✓ | ✓ | Feature data fusion 1 |
| fdf2 | 1,2 | ✓ | ✓ | Feature data fusion 2 |
| bfe1 | 1,0 | ✓ | ✓ | Background estimation and feature extraction 1 |
| bfe2 | 1,1 | ✓ | ✓ | Background estimation and feature extraction 2 |
| bfe3 | 1,2 | ✓ | ✓ | Background estimation and feature extraction 3 |
| bfe4 | 1,3 | ✓ | ✓ | Background estimation and feature extraction 4 |
| bfe5 | 2,0 | ✓ | ✓ | Background estimation and feature extraction 5 |
| bfe6 | 2,1 | ✓ | ✓ | Background estimation and feature extraction 6 |
| bfe7 | 2,2 | ✓ | ✓ | Background estimation and feature extraction 7 |
| bfe8 | 2,3 | ✓ | ✓ | Background estimation and feature extraction 8 |
| vod1 | 1,0 | ✓ | ✓ | Visual odometry 1 |
| vod2 | 2,3 | ✓ | ✓ | Visual odometry 2 |
| navc | 1,1 | ✓ | ✓ | Navigation control |
| thrc | 1,3 | ✓ | | Throttle control |
| stph | 2,2 | ✓ | ✓ | Stereo photogrammetry |
| usos | 0,3 | | ✓ | Ultrasonic sensor |
| obmg | 0,1 | ✓ | ✓ | Obstacle database manager |
| spes | 3,2 | | ✓ | Speed sensor |
| stac | 3,1 | ✓ | ✓ | Stability control |
| dirc | 3,3 | ✓ | | Direction control |
| vibs | 2,0 | | ✓ | Vibration sensor |
| obdb | 0,2 | ✓ | ✓ | Obstacle database |
| tpms | 0,3 | | ✓ | Tyre pressure monitoring |
| posi | 0,0 | | ✓ | Position sensor interface |
| tprc | 3,0 | ✓ | | Tyre pressure control |

Table C.1 – Original task-core mapping

| Flow | Denomination | SRC | DST | $L_f$ (flits) | $P_f$ (ms) |
|---|---|---|---|---|---|
| 1 | fbu3 → vod1 | 0,2 | 1,0 | 38400 | 40 |
| 2 | fbu8 → vod2 | 3,3 | 2,3 | 38400 | 40 |
| 3 | fbu1 → bfe1 | 0,0 | 1,0 | 38400 | 40 |
| 4 | fbu2 → bfe2 | 0,1 | 1,1 | 38400 | 40 |
| 5 | fbu3 → bfe3 | 0,2 | 1,2 | 38400 | 40 |
| 6 | fbu4 → bfe4 | 0,3 | 1,3 | 38400 | 40 |
| 7 | fbu5 → bfe5 | 3,0 | 2,0 | 38400 | 40 |
| 8 | fbu6 → bfe6 | 3,1 | 2,1 | 38400 | 40 |
| 9 | fbu7 → bfe7 | 3,2 | 2,2 | 38400 | 40 |
| 10 | fbu8 → bfe8 | 3,3 | 2,3 | 38400 | 40 |
| 11 | fdf1 → stph | 2,1 | 2,2 | 8192 | 40 |
| 12 | fdf2 → stph | 1,2 | 2,2 | 8192 | 40 |
| 13 | stph → obmg | 2,2 | 0,1 | 4096 | 40 |
| 14 | bfe1 → fdf1 | 1,0 | 2,1 | 2048 | 40 |
| 15 | bfe2 → fdf1 | 1,1 | 2,1 | 2048 | 40 |
| 16 | bfe3 → fdf1 | 1,2 | 2,1 | 2048 | 40 |
| 17 | bfe4 → fdf1 | 1,3 | 2,1 | 2048 | 40 |
| 18 | bfe5 → fdf2 | 2,0 | 1,2 | 2048 | 40 |
| 19 | bfe6 → fdf2 | 2,1 | 1,2 | 2048 | 40 |
| 20 | bfe7 → fdf2 | 2,2 | 1,2 | 2048 | 40 |
| 21 | bfe8 → fdf2 | 2,3 | 1,2 | 2048 | 40 |
| 22 | vod1 → navc | 1,0 | 1,1 | 512 | 40 |
| 23 | vod2 → navc | 2,3 | 1,1 | 512 | 40 |
| 24 | navc → thrc | 1,1 | 1,3 | 1024 | 100 |
| 25 | usos → obmg | 0,3 | 0,1 | 1024 | 100 |
| 26 | spes → stac | 3,2 | 3,1 | 1024 | 100 |
| 27 | stac → thrc | 3,1 | 1,3 | 1024 | 100 |
| 28 | navc → dirc | 1,1 | 3,3 | 512 | 100 |
| 29 | spes → navc | 3,2 | 1,1 | 512 | 100 |
| 30 | vibs → stac | 2,0 | 3,1 | 512 | 100 |
| 31 | obdb → navc | 0,2 | 1,1 | 16384 | 500 |
| 32 | obdb → obmg | 0,2 | 0,1 | 16384 | 500 |
| 33 | navc → obdb | 1,1 | 0,2 | 2048 | 500 |
| 34 | tpms → stac | 0,3 | 3,1 | 2048 | 500 |
| 35 | posi → navc | 0,0 | 1,1 | 1024 | 500 |
| 36 | posi → obmg | 0,0 | 0,1 | 1024 | 500 |
| 37 | obmg → obdb | 0,1 | 0,2 | 4096 | 1000 |
| 38 | stac → tprc | 3,1 | 3,0 | 2048 | 1000 |

Table C.2 – Flow set characteristics

| | $B = 2$ | | $B = 100$ | | $B = \infty$ | |
|---|---|---|---|---|---|---|
| **Flow** | **G-BATA** | **[42]** | **G-BATA** | **[42]** | **G-BATA** | **[42]** |
| 1 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 2 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 3 | 99.95% | 100.00% | 99.95% | 100.00% | 99.95% | 100.00% |
| 4 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 5 | 99.95% | 100.00% | 99.94% | 99.99% | 99.94% | 99.99% |
| 6 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 7 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 8 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 9 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 10 | 89.13% | 89.17% | 89.12% | 89.16% | 89.12% | 89.16% |
| 11 | 85.64% | 85.69% | 85.64% | 85.69% | 85.64% | 85.69% |
| 12 | 99.88% | 99.95% | 99.88% | 99.95% | 99.88% | 99.95% |
| 13 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 14 | 5.09% | 5.09% | 17.71% | 17.72% | 5.09% | 5.09% |
| 15 | 99.93% | 100.00% | 99.93% | 100.00% | 99.93% | 100.00% |
| 16 | 73.00% | 73.05% | 73.00% | 72.91% | 73.00% | 73.05% |
| 17 | 89.42% | 89.49% | 89.42% | 89.49% | 89.42% | 89.49% |
| 18 | 12.07% | 12.08% | 78.66% | 78.74% | 31.49% | 31.53% |
| 19 | 13.67% | 13.68% | 78.18% | 78.26% | 13.65% | 13.66% |
| 20 | 67.33% | 67.41% | 48.38% | 48.43% | 20.96% | 20.98% |
| 21 | 53.31% | 53.38% | 13.79% | 13.81% | 44.30% | 44.35% |
| 22 | 1.21% | 1.21% | 59.38% | 59.42% | 75.17% | 75.22% |
| 23 | 39.45% | 39.47% | 1.30% | 1.30% | 45.71% | 45.73% |
| 24 | 87.69% | 83.99% | 86.77% | 83.28% | 86.46% | 86.52% |
| 25 | 2.38% | 2.38% | 2.38% | 2.38% | 71.86% | 71.91% |
| 26 | 96.53% | 96.58% | 96.52% | 96.57% | 96.52% | 96.57% |
| 27 | 3.02% | 2.95% | 3.26% | 3.18% | 53.02% | 53.05% |
| 28 | 14.76% | 14.75% | 14.76% | 14.35% | 14.76% | 14.75% |
| 29 | 34.47% | 34.49% | 34.46% | 34.48% | 34.46% | 34.48% |
| 30 | 25.52% | 25.53% | 14.57% | 14.57% | 14.57% | 14.57% |
| 31 | 69.92% | 70.00% | 69.92% | 69.99% | 69.92% | 69.99% |
| 32 | 95.43% | 95.53% | 95.43% | 95.53% | 95.43% | 95.53% |
| 33 | 44.90% | 44.90% | 48.22% | 48.21% | 72.64% | 72.63% |
| 34 | 9.13% | 9.14% | 4.59% | 4.60% | 4.59% | 4.60% |
| 35 | 43.85% | 43.88% | 43.88% | 43.87% | 43.85% | 43.88% |
| 36 | 2.44% | 2.45% | 2.75% | 2.75% | 60.45% | 60.49% |
| 37 | 92.64% | 92.69% | 92.63% | 92.67% | 92.63% | 92.67% |
| 38 | 95.73% | 95.77% | 95.72% | 95.76% | 95.72% | 95.76% |

Table C.3 – Computed tightness ratios for buffer size values 2, 100 and $\infty$

Appendix D

# List of Publications

---

## Published

- F. Giroudot, A. Mifdaoui, "Buffer-Aware Worst-case Timing Analysis of Wormhole NoCs Using Network Calculus", in *IEEE 24th Real-Time and Embedded Technology and Applications Symposium*, April 2018
- F. Giroudot, A. Mifdaoui, "Work-in-Progress: Extending Buffer-Aware Worst-Case Timing Analysis of Wormhole NoCs", in *IEEE 39th Real-Time Systems Symposium*, December 2018
- F. Giroudot, A. Mifdaoui, "Tightness and Computation Assessment of Worst-Case Delay Bounds in Wormhole Networks-On-Chip", in *27th International Conference on Real-Time Networks and Systems (RTNS)*, November 2019

## Under review

- F. Giroudot, A. Mifdaoui, "Extending Buffer-Aware Worst-Case Timing Analysis of Wormhole NoCs with Interference Graph Approach", in *IEEE Access*

## To be submitted

- F. Giroudot, L. Apvrille, A. Mifdaoui, "An Hybrid Methodology Using Simulation and Network Calculus for Design Space Exploration of NoC-based Architectures Under Real-Time Constraints", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*

Appendix E

# List of Abbreviations

| | |
|---|---|
| **ASIC** | Application Specific Integrated Circuit |
| **BE** | Best-effort |
| **CABA** | Cycle-Accurate Bit-Accurate |
| **CAF** | Cumulative Arrival Function |
| **CDF** | Cumulative Departure Function |
| **COTS** | Commercial Off-The-Shelf |
| **CPA** | Compositional Performance Analysis |
| **CPQ** | Consecutive Packet Queueing |
| **DSE** | Design Space Exploration |
| **FCFS** | First-Come First-Served |
| **FIFO** | First-In First-Out |
| ***foi*** | flow of interest |
| **FP** | Fixed-Priority |
| **FPGA** | Field-Programmable Gate Array |
| **HW** | Hardware |
| **ISAE** | Institut Supérieur de l'Aéronautique et de l'Espace |
| **MDE** | Model-Driven Engineering |
| **NC** | Network Calculus |
| **NI** | Network Interface |
| **NoC** | Network-on-Chip |
| **NUMA** | Non-Uniform Memory Access |
| **QoS** | Quality of Service |
| **RC** | Recursive Calculus |

| | |
|---|---|
| **RR** | Round-Robin |
| **RTL** | Register Transfer Level |
| **S&F** | Store and Forward |
| **SoC** | System-on-Chip |
| **ST** | Scheduling Theory |
| **SW** | Software |
| **TCP** | Transmission Control Protocol |
| **TDMA** | Time-Division Multiple-Access |
| **TLM** | Transaction Level Modeling |
| **UML** | Unified Modeling Language |
| **VC** | Virtual Channel |
| **VCT** | Virtual Cut-Through |
| **WCET** | Worst-Case Execution Time |
| **WCRT** | Worst-Case Response Time |
| **WRR** | Weighted Round-Robin |

# Bibliography


[1] S. Tobuschat and R. Ernst, "Real-time communication analysis for networks-on-chip with backpressure," in *Design, Automation Test in Europe Conference Exhibition*, 2017. (Cited in pages xi, 36, 41, 82 et 83.)

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 973–990, USENIX Association, 2018. (Cited in page 3.)

[3] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. (Cited in page 3.)

[4] M. B. Taylor, *Tiled microprocessors*. PhD thesis, Massachussets Institute of Technology, February 2007. (Cited in pages 3 et 160.)

[5] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, Sept 2007. (Cited in pages 4, 23, 28, 55, 160 et 169.)

[6] Mellanox Technologies, "TILE-Gx72 processor." `http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf`. (Cited in pages 4, 24, 28 et 160.)

[7] Kalray Corporation, "The MPPA hardware architecture," 2012. (Cited in pages 4, 14, 15, 24, 28, 160 et 169.)

[8] Intel, "The SCC programmer's guide." `https://www.intel.cn/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-program-guide.pdf`, May 2010. (Cited in pages 4, 15, 23, 28 et 160.)

[9] *TTool GitLab*. (Cited in pages 5, 43, 118, 119, 171 et 186.)

[10] "Wopanets: Worst case performance analysis of embedded networks tool." `https://websites.isae-supaero.fr/wopanets/`. (Cited in pages 5, 118 et 119.)

[11] A. Mifdaoui and H. Ayed, "Wopanets: A tool for worst case performance analysis of embedded networks." (Cited in pages 5, 118 et 119.)

[12] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, pp. 10–19, Oct 1988. (Cited in pages 8 et 162.)

[13] L. Abdallah, *Worst-case delay analysis of core-to-IO flows over many-cores architectures*. PhD thesis, 2017. Thèse de doctorat dirigée par Fraboul, Christian et Jan, Mathieu Réseaux, Télécommunications, Systèmes et Architecture Toulouse, INPT 2017. (Cited in pages 8, 38, 46, 168 et 172.)

[14] A. Burns, L. S. Indrusiak, and Z. Shi, "Schedulability analysis for real time on-chip communication with wormhole switching," *Int. J. Embed. Real-Time Commun. Syst.*, vol. 1, pp. 1–22, Apr. 2010. (Cited in pages 8, 138 et 193.)

[15] Australian Government Department of Home Affairs, *Subclass 407 Training Visa*. `https://immi.homeaffairs.gov.au/visas/getting-a-visa/visa-listing/training-407`. (Cited in page 9.)

[16] G. Buttazzo, *Hard Real-Time Computing Systems*. Springer US, 2004. (Cited in page 9.)

[17] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 250–256, March 2000. (Cited in page 11.)

[18] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pp. 684–689, June 2001. (Cited in page 11.)

[19] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, pp. 70–78, Jan 2002. (Cited in page 11.)

[20] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, pp. 62–76, Feb 1993. (Cited in pages 13, 16, 22 et 164.)

[21] J. D. Owens, W. J. Dally, R. Ho, D. N. Jayasimha, S. W. Keckler, and L. S. Peh, "Research challenges for on-chip interconnection networks," *IEEE Micro*, vol. 27, pp. 96–108, Sept 2007. (Cited in pages 13 et 164.)

[22] N. Ni, M. Pirvu, and L. N. Bhuyan, "Circular buffered switch design with wormhole routing and virtual channels," in *ICCD*, pp. 466–473, 1998. (Cited in pages 13 et 164.)

[23] *NOCS 2010, Fourth ACM/IEEE International Symposium on Networks-on-Chip, Grenoble, France, May 3-6, 2010*, IEEE Computer Society, 2010. (Cited in pages 13 et 164.)

[24] N. Kavaldjiev, G. J. M. Smit, and P. G. Jansen, "A virtual channel router for on-chip networks," in *Proceedings 2004 IEEE International SOC Conference, September 12-15, 2004, Hilton Santa Clara, CA, USA*, pp. 289–293, IEEE, 2004. (Cited in pages 13, 55, 164 et 174.)

[25] G. P. Nychis, C. Fallin, T. Moscibroda, O. Mutlu, and S. Seshan, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," *SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 407–418, Aug. 2012. (Cited in pages 13 et 164.)

[26] M. B. Healy, K. Athikulwongse, R. Goel, M. M. Hossain, D. H. Kim, D. L. Lewis, B. Ouellette, M. Pathak, H. Sane, , D. H. Woo, G. H. Loh, and H. S. Lee, "Design and analysis of 3d-maps: A many-core 3d processor with stacked memory," in *IEEE Custom Integrated Circuits Conference 2010*, pp. 1–4, Sep. 2010. (Cited in pages 13 et 164.)

[27] G. Sun, C. Chang, B. Lin, and L. Zeng, "An oblivious routing algorithm for 3d mesh networks to achieve a new worst-case throughput bound," *IEEE Embedded Systems Letters*, vol. 4, pp. 98–101, Dec 2012. (Cited in pages 13 et 164.)

[28] W. J. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Transactions on Computers*, vol. 39, pp. 775–785, June 1990. (Cited in page 13.)

[29] J. Kim, J. D. Balfour, and W. J. Dally, "Flattened butterfly topology for on-chip networks," *Computer Architecture Letters*, vol. 6, no. 2, pp. 37–40, 2007. (Cited in page 13.)

[30] Q. Perret, *Predictable execution on many-core processors.* Thèses, Institut Superieur de l'Aéronautique et de l'Espace (ISAE) ; Université de Toulouse, Apr. 2017. (Cited in pages 14, 48 et 172.)

[31] R. Das, S. Eachempati, A. K. Mishra, N. Vijaykrishnan, and C. R. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps," in *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pp. 175–186, IEEE Computer Society, 2009. (Cited in page 14.)

[32] LIP6, "Tsar architecture overview." `https://www-soc.lip6.fr/trac/tsar/wiki/Specification`. (Cited in page 15.)

[33] A. Agarwal and R. Shankar, "Survey of network on chip (noc) architectures and contributions," *Journal of Engineering, Computing and Architecture*, vol. 3, 01 2009. (Cited in pages 16, 17, 21 et 165.)

[34] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks (1976)*, vol. 3, no. 4, pp. 267 – 286, 1979. (Cited in page 16.)

[35] G. Michelogiannakis, D. Sanchez, W. J. Dally, and C. Kozyrakis, "Evaluating bufferless flow control for on-chip networks," in *NOCS 2010, Fourth ACM/IEEE International Symposium on Networks-on-Chip, Grenoble, France, May 3-6, 2010*, pp. 9–16, IEEE Computer Society, 2010. (Cited in page 17.)

[36] B. Daya, L. S. Peh, and A. Chandrakasan, "Towards high-performance bufferless nocs with scepter," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015. (Cited in page 17.)

[37] C. Busch, M. Herlihy, and R. Wattenhofer, "Routing without flow control," in *SPAA*, pp. 11–20, 2001. (Cited in pages 17 et 22.)

[38] W. J. Dally, "Virtual-channel flow control," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), pp. 60–68, ACM, 1990. (Cited in pages 19 et 165.)

[39] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, June 2006. (Cited in pages 19 et 165.)

[40] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. 36, pp. 547–553, May 1987. (Cited in pages 20 et 22.)

[41] Z. Shi and A. Burns, "Real-time communication analysis with a priority share policy in on-chip networks," in *21st Euromicro Conference on Real-Time Systems*, pp. 3–12, July 2009. (Cited in pages 20, 35, 40, 41 et 167.)

[42] B. Nikolic, S. Tobuschat, L. Soares Indrusiak, R. Ernst, and A. Burns, "Real-time analysis of priority-preemptive nocs with arbitrary buffer sizes and router delays," *Real-Time Systems*, 06 2018. (Cited in pages 20, 35, 40, 41, 133, 138, 139, 142, 167, 193 et 210.)

[43] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, (New York, NY, USA), pp. 278–287, ACM, 1992. (Cited in page 22.)

[44] M. Gries, U. Hoffmann, M. Konow, and M. Riepen, "Scc: A flexible architecture for many-core platform research," *Computing in Science Engineering*, vol. 13, pp. 79–83, Nov 2011. (Cited in page 23.)

[45] W. Puffitsch, E. Noulard, and C. Pagetti, "Mapping a multi-rate synchronous language to a many-core processor," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 293–302, April 2013. (Cited in page 23.)

[46] W. Puffitsch, E. Noulard, and C. Pagetti, "Off-line mapping of multi-rate dependent task sets to many-core platforms," *Real-Time Syst.*, vol. 51, pp. 526–565, Sept. 2015. (Cited in pages 23, 47 et 172.)

[47] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," *IEEE Design Test of Computers*, vol. 22, pp. 414–421, Sep. 2005. (Cited in page 28.)

[48] D. Wiklund and D. Liu, "Socbus: Switched network on chip for hard real time embedded systems," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, (Washington, DC, USA), pp. 78.1–, IEEE Computer Society, 2003. (Cited in page 28.)

[49] D. Bui, A. Pinto, and E. A. Lee, "On-time network on-chip: Analysis and architecture," Tech. Rep. UCB/EECS-2009-59, University of California, Berkeley, May 2009. (Cited in page 28.)

[50] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-accurate network on chip simulation with noxim," *ACM Trans. Model. Comput. Simul.*, vol. 27, pp. 4:1–4:25, Aug. 2016. (Cited in pages 32, 77, 112 et 180.)

[51] J. Hu and R. Marculescu, "Application-specific buffer space allocation for networks-on-chip router design," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '04, (Washington, DC, USA), pp. 354–361, IEEE Computer Society, 2004. (Cited in page 32.)

[52] A. E. Kiasari, Z. Lu, and A. Jantsch, "An analytical latency model for networks-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 113–123, Jan 2013. (Cited in page 32.)

[53] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973. (Cited in page 33.)

[54] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, Sep. 1993. (Cited in page 33.)

[55] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, pp. 117–134, Apr. 1994. (Cited in pages 34 et 168.)

[56] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in *Networks-on-Chip, Second ACM/IEEE International Symposium on*, April 2008. (Cited in pages 34, 35 et 167.)

[57] Q. Xiong, Z. Lu, F. Wu, and C. Xie, "Real-time analysis for wormhole noc: Revisited and revised," *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 75–80, 2016. (Cited in pages 35, 40 et 167.)

[58] Q. Xiong, F. Wu, Z. Lu, and C. Xie, "Extending real-time analysis for wormhole nocs," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2017. (Cited in pages 35, 40, 41 et 167.)

[59] M. Liu, M. Becker, M. Behnam, and T. Nolte, "Tighter time analysis for real-time traffic in on-chip networks with shared priorities," in *10th IEEE/ACM International Symposium on Networks-on-Chip*, 2016. (Cited in pages 35, 41 et 167.)

[60] L. S. Indrusiak, A. Burns, and B. Nikolić, "Buffer-aware bounds to multi-point progressive blocking in priority-preemptive nocs," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 219–224, March 2018. (Cited in page 35.)

[61] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, pp. 148–166, March 2005. (Cited in pages 36 et 168.)

[62] R. Hofmann, L. Ahrendts, and R. Ernst, *CPA: Compositional Performance Analysis*, pp. 721–751. Dordrecht: Springer Netherlands, 2017. (Cited in page 36.)

[63] E. A. Rambo and R. Ernst, "Worst-case communication time analysis of networks-on-chip with shared virtual channels," in *Proceedings of Design, Automation Test in Europe Conference Exhibition*, 2015. (Cited in pages 36, 41 et 168.)

[64] K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, pp. 133–151, Mar 1994. (Cited in page 36.)

[65] T. Ferrandiz, F. Frances, and C. Fraboul, "A method of computation for worst-case delay analysis on spacewire networks." (Cited in pages 37, 41 et 168.)

[66] M. Liu, M. Becker, M. Behnam, and T. Nolte, "Buffer-aware analysis for worst-case traversal time of real-time traffic over rra-based nocs," in *27th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2017. (Cited in pages 37, 41, 55 et 168.)

[67] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul, "Wormhole networks properties and their use for optimizing worst case delay analysis of many-cores," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10, June 2015. (Cited in pages 37, 38, 41 et 168.)

[68] R. L. Cruz, "A calculus for network delay. i. network elements in isolation," *IEEE Transactions on Information Theory*, vol. 37, pp. 114–131, Jan 1991. (Cited in pages 38, 56 et 169.)

[69] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001. (Cited in pages 38, 56, 169, 201 et 202.)

[70] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *14th*

*GI/ITG Conference - Measurement, Modelling and Evalutation of Computer and Communication Systems*, pp. 1–15, March 2008. (Cited in pages 38, 60, 65, 178 et 203.)

[71] T. Ferrandiz, F. Frances, and C. Fraboul, "Modeling spacewire networks with network calculus," in *Proceedings of the 1st International Workshop on Worst-Case Traversal Time*, WCTT '11, (New York, NY, USA), pp. 51–57, ACM, 2011. (Cited in pages 39 et 169.)

[72] T. Ferrandiz, F. Frances, and C. Fraboul, "A network calculus model for spacewire networks," in *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2011, Toyama, Japan, August 28-31, 2011, Volume 1*, pp. 295–299, IEEE Computer Society, 2011. (Cited in pages 39 et 169.)

[73] Yue Qian, Zhonghai Lu, and Wenhua Dou, "Analysis of communication delay bounds for network on chips," in *2009 Asia and South Pacific Design Automation Conference*, pp. 7–12, Jan 2009. (Cited in pages 39 et 169.)

[74] L. Lenzini, L. Martorini, E. Mingozzi, and G. Stea, "Tight end-to-end per-flow delay bounds in {FIFO} multiplexing sink-tree networks," *Performance Evaluation*, vol. 63, no. 9-10, pp. 956 – 987, 2006. (Cited in pages 39 et 169.)

[75] Y. Qian, Z. Lu, and W. Dou, "Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip," in *Networks-on-Chip, 3rd ACM/IEEE International Symposium on*, May 2009. (Cited in pages 39, 40, 41 et 169.)

[76] F. Jafari, Z. Lu, and A. Jantsch, "Least upper delay bound for vbr flows in networks-on-chip with virtual channels," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, pp. 35:1–35:33, June 2015. (Cited in pages 39, 41 et 169.)

[77] M. Boyer, B. Dupont De Dinechin, A. Graillat, and L. Havet, "Computing Routes and Delay Bounds for the Network-on-Chip of the Kalray MPPA2 Processor," in *ERTS 2018 - 9th European Congress on Embedded Real Time Software and Systems*, (Toulouse, France), Jan. 2018. (Cited in pages 39, 40, 41 et 169.)

[78] A. Mifdaoui and H. Ayed, "Buffer-aware worst case timing analysis of wormhole network on chip," *arXiv*, vol. abs/1602.01732, 2016. (Cited in pages 39 et 40.)

[79] J. Delorme, *Methodology of modeling and architectural exploration of Network on Chip applied to telecommunications*. Theses, INSA de Rennes, Feb. 2007. (Cited in pages 42 et 171.)

[80] J.-J. Lecler and G. Baillieu, "Application driven network-on-chip architecture exploration & refinement for a complex soc," *Design Automation for Embedded Systems*, vol. 15, pp. 133–158, Jun 2011. (Cited in pages 42 et 171.)

[81] M. M. Real, P. Wehner, J. Rettkowski, V. Migliore, V. Lapotre, D. Göhringer, and G. Gogniat, "Mpsocsim extension: An ovp simulator for the evaluation of cluster-based multi and many-core architectures," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 342–347, July 2016. (Cited in pages 43 et 171.)

[82] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Readings in hardware/-software co-design," ch. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pp. 527–543, Norwell, MA, USA: Kluwer Academic Publishers, 2002. (Cited in pages 43 et 171.)

[83] C. Ptolemaeus, ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. (Cited in pages 43 et 171.)

[84] A. D. Pimentel, L. O. Hertzbetger, P. Lieverse, P. van der Wolf, and E. E. Deprettere, "Exploring embedded-systems architectures with artemis," *Computer*, vol. 34, pp. 57–63, Nov 2001. (Cited in pages 43 et 171.)

[85] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers, *A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach*, vol. 2268, pp. 321–324. 04 2002. (Cited in pages 43 et 171.)

[86] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J. Diguet, "A co-design approach for embedded system modeling and code generation with uml and marte," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 226–231, April 2009. (Cited in page 43.)

[87] "UML profile for MARTE specification." `https://www.omg.org/spec/MARTE/`. (Cited in page 43.)

[88] A. Gamatié, V. Rusu, and E. Rutten, "Operational semantics of the marte repetitive structure modeling concepts for data-parallel applications design," in *2010 Ninth International Symposium on Parallel and Distributed Computing*, pp. 25–32, July 2010. (Cited in page 43.)

[89] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, Nov. 2011. (Cited in page 43.)

[90] D. Knorreck, L. Apvrille, and R. Pacalet, "Formal system-level design space exploration," vol. 25, pp. 1 – 8, 07 2010. (Cited in pages 43, 118, 119 et 171.)

[91] L. W. Li, D. Genius, and L. Apvrille, "Formal and virtual multi-level design space exploration," in *Model-Driven Engineering and Software Development* (L. F. Pires, S. Hammoudi, and B. Selic, eds.), (Cham), pp. 47–71, Springer International Publishing, 2018. (Cited in pages 44 et 118.)

[92] N. Bombieri, F. Fummi, and D. Quaglia, "System/network design-space exploration based on tlm for networked embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 9, 03 2010. (Cited in pages 44 et 172.)

[93] F. Fummi, D. Quaglia, F. Stefanni, and G. Lovato, "Modeling of communication infrastructure for design-space exploration.," vol. 2010, pp. 92–97, 01 2010. (Cited in pages 44 et 172.)

[94] E. S. M. Ebeid, F. Fummi, D. Quaglia, H. Posadash, and E. Villar, "A framework for design space exploration and performance analysis of networked embedded systems," 01 2014. (Cited in pages 44 et 172.)

[95] A. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," 05 2013. (Cited in pages 44, 45 et 172.)

[96] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1979. (Cited in pages 45 et 172.)

[97] Tang Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Euromicro Symposium on Digital System Design, 2003. Proceedings.*, pp. 180–187, Sep. 2003. (Cited in pages 45 et 172.)

[98] S. Murali, S. Murali, G. De Micheli, G. De Micheli, and G. De Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '04, (Washington, DC, USA), pp. 20896–, IEEE Computer Society, 2004. (Cited in pages 45, 46 et 172.)

[99] Jingcao Hu and R. Marculescu, "Energy- and performance-aware mapping for regular noc architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 551–562, April 2005. (Cited in pages 45, 46 et 172.)

[100] C.-L. Chou and R. Marculescu, "Contention-aware application mapping for network-on-chip communication architectures," pp. 164 – 169, 11 2008. (Cited in pages 46 et 172.)

[101] A. Kanduri, A. Rahmani, P. Liljeberg, and H. Tenhunen, "Predictable application mapping for manycore real-time and cyber-physical systems," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pp. 135–142, Sep. 2015. (Cited in pages 46 et 172.)

[102] C. Zimmer and F. Mueller, "Low contention mapping of real-time tasks onto tilepro 64 core processors," in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, p. 2012. (Cited in pages 47 et 172.)

[103] T. Carle, M. Djemal, D. Potop-Butucaru, and R. De Simone, "Static mapping of real-time applications onto massively parallel processor arrays," in *14th International Conference on Application of Concurrency to System Design*, Proceedings ACSD 2014, (Hammamet, Tunisia), June 2014. (Cited in pages 47 et 172.)

[104] A. Bouillard, L. Jouhet, and E. Thierry, "Service curves in Network Calculus: dos and don'ts," Research Report RR-7094, INRIA, 2009. (Cited in page 202.)

[105] A. Bouillard, N. Farhi, and B. Gaujal, "Packetization and aggregate scheduling," *Research Report - INRIA*, 07 2011. (Cited in page 202.)