THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

Mme LAURE ABDALLAH

le mercredi 5 avril 2017

Titre :

Worst-case delay analysis of core-to-IO flows over many-cores architectures

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. CHRISTIAN FRABOUL

M. MATHIEU JAN

Rapporteurs :

M. LAURENT GEORGE, ESIEE NOISY LE GRAND

Mme ISABELLE PUAUT, UNIVERSITE RENNES 1

Membre(s) du jury :

M. LAURENT GEORGE, ESIEE NOISY LE GRAND, Président

M. CHRISTIAN FRABOUL, INP TOULOUSE, Membre

M. JEROME ERMONT, INP TOULOUSE, Membre

M. MARC GATTI, THALES AVIONICS, Membre

M. MATHIEU JAN, CEA SACLAY, Membre

Acknowledgements

I would like to extend my sincere gratitude to all the reviewers for accepting to evaluate this work: to Mr. Laurent Georges, Professor at the University of Paris-Est, and Mrs Isabelle Puaut, Professor at the University of Rennes 1, who very kindly agreed to be the scientific evaluators of this thesis. My earnest thanks are also due to Mr. Marc Gatti, Research Director at Avionics Thales, for accepting to be a member of the jury.

I have an immense respect, appreciation and gratitude towards my advisors: Mr. Mathieu Jan, Research Engineer at CEA LIST, Mr. Jérôme Ermont, Associate Professor at INP-ENSEEIHT, and Mr. Christian Fraboul, Professor at INP-ENSEEIHT, for their consistent support and for sharing their invaluable knowledge, experience and advice which have been greatly beneficial to the successful completion of this thesis.

Thanks to all L3S team members for the occasional coffees and discussions especially during our "PhD Days".

I would also like to thank my friends Damien, Briag, Sarah, Jad, Hadi and Marcelino for their support and all our happy get-togethers and for uplifting my spirit during the challenging times faced during this thesis.

I am particularly grateful to my friends Ola and Mohammad for their priceless friendship and precious support, and for continuously giving me the courage to make this thesis better.

Last but definitely not least, I am forever indebted to my awesomely supportive parents, my wonderful sisters, Layal, Reem and Ghina, and my beloved brother Wassim, for inspiring and guiding me throughout the course of my life, and hence this thesis.

Abstract

Many-core architectures are more promising hardware to design real-time systems than multi-core systems as they should enable an easier mastered integration of a higher number of applications, potentially of different level of criticalities. In embedded real-time systems, these architectures will be integrated within backbone Ethernet networks, as they mostly provide Ethernet controllers as Input/Output(I/O) interfaces. Thus, a number of applications of different level of criticalities could be allocated on the Network-on-Chip (NoC) and required to communicate with sensors and actuators. However, the worst-case behavior of NoC for both inter-core and core-to-I/O communications must be established. Several NoCs targeting hard real-time systems, made of specific hardware extensions, have been designed. However, none of these extensions are currently available in commercially NoC-based many-core architectures, that instead rely on wormhole switching with round-robin arbitration. Using this switching strategy, interference patterns can occur between direct and indirect flows on many-cores. Besides, the mapping over the NoC of both critical and non-critical applications has an impact on the network contention these core-to-I/O communications exhibit.

These core-to-I/O flows (coming from the Ethernet interface of the NoC) cross two networks of different speeds: NoC and Ethernet. On the NoC, the size of allowed packets is much smaller than the size of Ethernet frames. Thus, once an Ethernet frame is transmitted over the NoC, it will be divided into many packets. When all the data corresponding to this frame are received by the DDR-SDRAM memory on the NoC, the frame is removed from the buffer of the Ethernet interface. In addition, the congestion on the NoC, due to wormhole switching, can delay these flows. Besides, the buffer in the Ethernet interface has a limited capacity. Then, this behavior may lead to a problem of dropping Ethernet frames. The idea is therefore to analyze the worst case transmission delays on the NoC and reduce the delays of the core-to-I/O flows.

In this thesis, we show that the pessimism of the existing Worst-Case Traversal Time (WCTT) computing methods and the existing mapping strategies lead to drop Ethernet frames due to an internal congestion in the NoC. Thus, we demonstrate properties of such NoC-based wormhole networks to reduce the pessimism when modeling flows in contentions. Then, we propose a mapping strategy that minimizes the contention of core-to-I/O flows in order to solve this problem.

We show that the WCTT values can be reduced up to 50% compared to current state-of-the-art real-time packet schedulability analysis. These results are due to the modeling of the real impact of the flows in contention in our proposed computing method. Besides, experimental results on real avionics applications show significant improvements of core-to-I/O flows transmission delays, up to 94%, without significantly impacting transmission delays of core-to-core flows. These improvements are due to our mapping strategy that allocates the applications in such a way to reduce the impact of non-critical flows on critical flows. These reductions on the WCTT of the core-to-I/O flows avoid the drop of Ethernet frames.

Résumé

Les architectures pluri-coeurs sont plus intéressantes pour concevoir des systèmes en temps réel que les systèmes multi-coeurs car il est possible de les maîtriser plus facilement et d'intégrer un plus grand nombre d'applications, potentiellement de différents niveau de criticité. Dans les systèmes temps réel embarqués, ces architectures peuvent être utilisées comme des éléments de traitement au sein d'un réseau fédérateur car ils fournissent un grand nombre d'interfaces Entrées/Sorties telles que les contrôleurs Ethernet et les interfaces de la mémoire DDR-SDRAM. Aussi, il est possible d'y allouer des applications ayant différents niveaux de criticités. Ces applications communiquent entre elles à travers le réseau sur puce (NoC) du pluri-coeur et avec des capteurs et des actionneurs via l'interface Ethernet. Afin de garantir les contraintes temps réel de ces applications, les délais de transmission pire cas (WCTT) doivent être calculés pour les flux entre les coeurs ("inter-core") et les flux entre les coeurs et les interfaces entrées/sorties ("core-to-I/O"). Plusieurs réseaux sur puce (NoCs) ciblant les systèmes en temps réel dur ont été conçus en s'appuyant sur des extensions matérielles spécifiques. Cependant, aucune de ces extensions ne sont actuellement disponibles dans les architectures de réseaux sur puce commercialisés, qui se basent sur la commutation wormhole avec la stratégie d'arbitrage par tourniquet. En utilisant cette stratégie de commutation, différents types d'interférences peuvent se produire sur le réseau sur puce entre les flux. De plus, le placement de tâches des applications critiques et non critiques a un impact sur les contentions que peut subir les flux "core-to-I/O". Ces flux "core-to-I/O" parcourent deux réseaux de vitesses différentes: le NoC et Ethernet. Sur le NoC, la taille des paquets autorisés est beaucoup plus petite que la taille des trames Ethernet. Ainsi, lorsque la trame Ethernet est transmise sur le NoC, elle est divisée en plusieurs paquets. La trame sera supprimée de la mémoire tampon de l'interface Ethernet uniquement lorsque la totalité des données aura été transmise. Malheureusement, la congestion du NoC ajoute des délais supplémentaires à la transmission des paquets et la taille de la mémoire tampon de l'interface Ethernet est limitée. En conséquence, ce comportement peut aboutir au rejet des trames Ethernet. L'idée donc est de pouvoir analyser les délais de transmission pire cas sur les NoC et de réduire leurs délais afin d'éviter ce problème de rejet.

Dans cette thèse, nous montrons que le pessimisme de méthodes existantes de calcul de WCTT et les stratégies de placements existantes conduisent à rejeter des trames Ethernet en raison d'une congestion interne sur le NoC. Des propriétés des réseaux utilisant la commutation "wormhole" ont été définies et validées afin de mieux prendre en compte les conflits entre les flux. Une stratégie de placement de tâches qui prend en compte les communications avec les I/O a été ensuite proposée. Cette stratégie vise à diminuer les contentions des flux qui proviennent de l'I/O et donc de réduire leurs WCTTs.

Les résultats obtenus par la méthode de calcul définie au cours de cette thèse montrent que les valeurs du WCTT des flux peuvent être réduites jusqu'à 50% par rapport aux valeurs de WCTT obtenues par les méthodes de calcul existantes. En outre, les résultats expérimentaux sur des applications avioniques réelles montrent des améliorations significatives des délais de transmission des flux "core-to-I/O", jusqu'à 94%, sans impact significatif sur ceux des flux "inter-core". Ces améliorations sont dues à la stratégie d'allocation définie qui place les applications de manière à réduire l'impact des flux non critiques sur les flux critiques. Ces réductions de WCTT des flux "core-to-I/O" évitent le rejet des trames Ethernet.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivation

The use of real-time systems continues to spread in many diverse application areas including, engine management, process control, medical electronics, telecommunications, robotics, multimedia and avionics. The underlying idea of real-time systems is not the speed concept but the reactivity: a real-time system operates in a dynamic environment and must constantly adapt to the changes in this environment. This implicates that the response to these changes must be adapted (functional correction), but must also respect the time constraints (time correction). These time constraints vary considerably in their degree of severity. In soft real-time systems, the late response may be tolerated. In contrast, in hard real-time systems, a failure to respond within the time constraints may constitute a catastrophic failure of the system.

In this thesis, we focus on hard real-time systems.

Real-time systems range in complexity from simple controllers implementing a single function, to complex sets of communicating sub-systems, each of which is responsible for performing a number of critical functions. These systems are becoming increasingly complex with the complexity of applications, whether in architectures (number of processors, the presence of networks), or in requirements (criticality of tasks, computing power, energy consumption, etc..). The need to meet these requirements lead to computer systems that comprised millions of transistors on a single chip, commonly called Systems on Chip or SoC. The most common communication backbone used in SoCs is the shared medium arbitrated bus. However, a bus based SoC does not scale with the number of cores attached, and so the performance is reduced. Thus, a search for the communication backbone of next generation many-core based SoCs supporting new inter-core communication demands started, and lead to the design of what is called Network on Chip (NoC). NoC has emerged as a viable alternative as it consists of various cores connected to a router-based network. Such communication architecture is described as modular and scalable [BDM02b, DT01, KJS+02].

In this thesis, we focus on the NoC-based many-core architectures.

Many-core architectures are indeed promising hardware to support the design of hard real-time systems [NYP+14a]. They are based on simpler cores, without complex hardware mechanisms that can be found in multi-core architectures. The timing predictability of cores within many-core architectures are thus easier to analyze. In order to support hard real-time traffic, the guarantees in the worst-case scenarios must be established. Then, the Worst-Case Traversal Time (WCTT) of all the packets generated by a flow must be lower than a predetermined deadline. A deadline marks the latest time that a packet should arrive to its destination. Thus, Quality-of-Service (QoS) has been a major concern for NoC [BM06]. In fact, packets are routed in a network where resources are shared, thus bringing unpredictable performance. NoCs can thus be used in hard real-time systems using two approaches. A first approach is to use analytic methods to analyze the WCTTs of flows on existing many-cores. A second one is to modify the hardware architecture in such a way no contentions can occur by design, leading to straightforward WCTT for flows. However, the problem in the second approach is that it does not exist a commercial NoC architecture implementing it.

In this thesis, we focus on the use of NoC architectures in hard real-time systems using the first approach.

In embedded real-time systems, a number of applications of different level of criticalities could be allocated on a NoC and required to communicate with sensors and actuators. As many-cores architectures present an important number of Ethernet interfaces, thus, it could be used as processing elements within a backbone Ethernet network. The challenge is then to analyze the worst-case behavior of the underlying NoC [BDM02a] used for both inter-core and core to external memories or peripherals communications. Such real-time packet schedulability analysis have been done for various types of networks by taking into account the type of contentions that can occur between flows. However, none of these works consider in addition the core-to-I/O communications.

In this thesis, we focus on the integration of I/O constraints within NoC communications.

A core-to-I/O flow experiences a change in its speed as it crosses two networks of different types (NoC and Ethernet). However, in NoCs, the congestion is possible especially when using a wormhole switching. A NoC congestion delays the core-to-I/O flow leading to an overflow of the buffer of the Ethernet interface which is of limited capacity. Therefore, incoming Ethernet frames holding critical data could be dropped. Real-time packet schedulability analysis must then be done by taking into account all types of contentions that can occur between all types of flows, which severely complicate its timing analysis.

The objective of this thesis is to analyze the WCTT of all flows on the NoC and to reduce the WCTT of the core-to-I/O flow in order to avoid the loss of critical payloads.

1.2 Contributions

In the aim to reach the objectives mentioned in the previous section, this thesis presents three main contributions:

- An analysis showing that existing computing methods and congestion-aware mapping strategies are not sufficient to integrate the I/O constraints within the NoC communications. Besides, an explanation of the I/O interfaces and an illustration of the communications between cores and I/O interfaces are presented. To the best of our knowledge, this is the first work that integrates I/O communications within the NoC communications.
- An analytical method to compute the WCTT by including three properties of the wormhole switching when analyzing contentions between flows. This method reduces the pessimism of the existing methods, thus tightest bounds of the delay can be obtained.
- A congestion-aware mapping strategy of critical and non-critical applications that consider not only the core-to-core communications, but also the core-to-I/O communications. This strategy aims to reduce the contention on the core-to-I/O flows and then their WCTTs.

The thesis is organized as follows:

- **Chapter 2** presents the main NoC concepts that impact the WCTT of flows over different architectures. Besides, we present some of the existing NoC architectures in order to argue the architecture we assume in this work
- **Chapter 3** describes the existing methods to compute the WCTT of the flows on the wormhole networks. It illustrates also the existing mapping strategies that reduce the congestion on the NoC and thus the WCTT of these flows.
- **Chapter 4** presents our first contribution: an illustration of the the problems when using NoCs in real-time systems interconnected to sensors and actuators via Ethernet. Thus, first, we present the model of our NoC architecture and the I/O model. Then, we show

in a motivating case study made of real-time applications what are the limitations of the existing WCTT analysis and mapping methods that lead to these problems.

- **Chapter 5** explains our second contribution: an analysis method that reduces the pessimism when computing WCTT of flows. This chapter shows the improvements of our method compared to the existing methods.
- **Chapter 6** describes our third contribution: a mapping algorithm that takes into account the core-to-I/O flows and that aims to reduce the contention on these flows.
- **Chapter 7** presents an evaluation of our contributions compared to the current state-of-the-art methods on several case studies. The objective of this chapter is to illustrate the impacts of our proposed approaches on the WCTT of the core-to-I/O and core-to-core flows.
- **Chapter 8** concludes the manuscript by summarizing the major contributions of the thesis and proposing interesting research directions as future work.

1.3 Publications

This is a list of papers and publications that reflects the results achieved during the developement of the research work presented in this dissertation. A significant part of this thesis is compiled from these papers and publications.

- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian Fraboul. "Reducing the contention experienced by real-time core-to-I/O flows over a Network on Chip", In *28th Euromicro Conference on Real-Time Systems (ECRTS)*, Toulouse, France, July 2016.
- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian 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 2015)*, pages 59–68, Siegen, Germany, June 2015.

- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian Fraboul. “I/O contention aware mapping of multi-crticalities real-time applications over many-core architectures”, In *Proc. of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Session Work-In-Progress*, Vienna, Austria, April 2016.

- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian Fraboul. “Optimizing worst case delay analysis on wormhole networks by modeling the pipeline behavior”, In *Proc. of the 13th International Workshop on Real-Time Networks (RTN)*, Madrid, Spain, July 2014.

- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian Fraboul. “Why and how to map real-time core-to-I/O flows over a Tilera-like Network on Chip?”, In *FNRS Seminar-Real-time networks*, Brussels, Belgium, May 2016.

- Laure Abdallah, Mathieu Jan, Jérôme Ermont and Christian Fraboul. “Propriétés des réseaux wormhole pour optimiser l’analyse de délai pire cas dans les many-coeurs”, In *Real-time summer school*, Rennes, France, August 2016.

Chapter 2

Network-on-Chip for real-time service

Contents

Network-on-chip has been a very active research field since their emergence in the early 2000s, as it offers various opportunities in terms of performance and computing capabilities. At the same time, they pose many challenges to be used in real-time systems and ensure the temporal predictability of flows. Indeed, the flows on a NoC are routed in a network where resources are shared, thus bringing unpredictable performance. Different parameters impact the upper bounds delay of a flow on a NoC. This chapter aims to analyze the different implementations of the NoC concepts and their impact on delivering real-time requirements. In this chapter, we first present the NoC components and we define the real-time systems. Then, we show how the

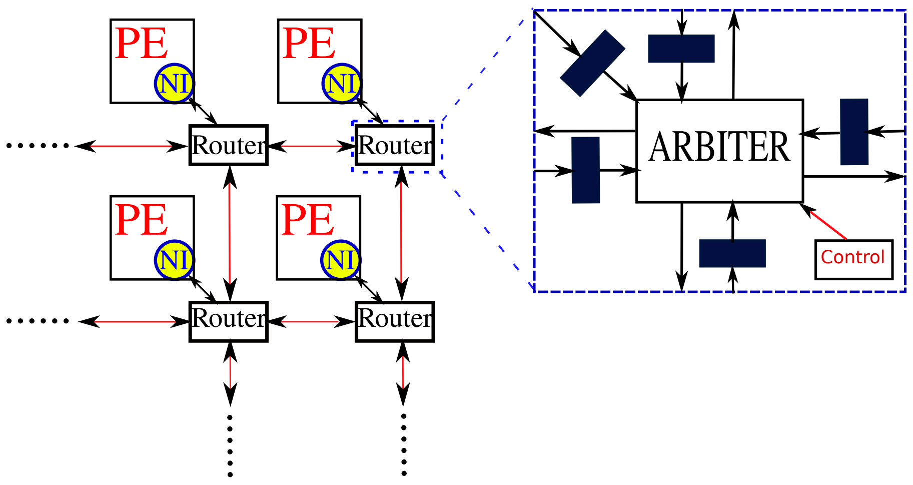


Figure 2.1: The NoC components.

implementation of NoC concepts impacts the needed Quality-of-Service (QoS) and the design complexity of the NoC. Finally, this chapter shows the NoC architecture on which our work is based.

2.1 NoC components

NoC was introduced in [DT01] as a better alternative to global wiring structures, used to interconnect different Intellectual Property (IP) cores, which are also called Processing Elements (PE). A NoC, being on a single chip, is composed of a number of interconnected tiles. A tile, which is also called a node, contains one or many IP cores and a network router. A NoC also interconnects the tiles to the I/O interfaces such as the external memory, the Ethernet interfaces, etc. Figure 2.1 illustrates the structure of a NoC which is composed of routers, network interfaces and links.

Router

A router is used to send messages from one tile to another. It consists of a set-up of input and output ports that can (or not) contain buffers, an interconnection matrix, and an arbiter. Routers can have buffers either at input port or at output port [DT04]. Figure 2.1 illustrates an example of a router with input buffers. These inputs and outputs ports are connected to each other via the interconnection matrix which is controlled by the arbiter [DMB06].

Links

Routers communicate to each other by one or more physical channels, i.e. a link [DMB06, LRV06]. This connection could be uni- or bi-directional.

Network interface

Network Interface (NI) provides an interface between the core and the network. It makes NoC transparent for the PEs. It is used by PEs to access the interconnect medium. The data produced by the core have to be encapsulated by the source NI and decapsulated by the destination NI. NI converts the messages into packets for the transmission of the NoC.

A packet is divided into flow control units (flits). A flit is the minimum unit of information that can be transferred across a link [CSG99]. The first flit is called the header. The packet header contains informations about the destination NoC node. These informations are needed by NI to determine the path of the packet. The header is followed by one or several flits which compose the data payload. The data payload is the data transmitted by the IP core across the NoC.

2.2 Real-time systems

In a real-time system, the accuracy of the application depends not only on the result but also on the time at which this result is produced. A data packet received by a destination too late could be useless or even cause a severe consequence. Typical examples of real-time systems include control systems for cars, aircraft and space vehicles. In an Antilock Braking System (ABS), not only the breaking pressure must be calculated, but also the time of application is critical to gain a functioning ABS. ABS braking system should not put more than 150ms to receive the information and 1s to react.

Thus, time is an important characteristic for real-time systems. This characteristic distinguishes these systems from other types of computing systems. For on-chip communications, the packet transmission duration is the time which interests us. Assigning a deadline to each packet distinguishes a real-time communication from a non real-time one. A deadline marks the latest time that a packet should arrive to its destination. Thus, a real-time communication means meeting the deadlines, i.e. satisfying the timing constraints. For hard real-time systems, a missed deadline is not tolerated. Thus, communications in these systems must imperatively meet deadlines, otherwise the system might fail and the penalty incurred is catastrophic. It could lead to loss of life, serious damage to the environment or threats to business interests. As an example of hard real-time applications, Full Authority Digital Engine Controller (FADEC) that controls the activities of an aircraft jet engine. The FADEC design requires particular timing requirements. For example, if the FADEC senses that a turbine drive shaft has broken, then the FADEC must respond with a damage mitigating action in a predetermined time. In this thesis, we are interested in such hard-real time communications, where we focus on the response time communications. Now, let us see how the implementation of NoC concepts impact the delivery of such real-time requirements.

2.3 NoC concepts and real-time requirements

A number of metrics are needed to measure and quantifying the performance of a NoC [KPN$^+$05, DYN03]. In general, it is desirable that a NoC architecture exhibits low latency, high throughput, energy efficiency, low cost, low area overhead and high scalability. In hard real-time systems, the delay bounds of each packet must be guaranteed. As we deal with these systems, we focus on the latency metric where the maximal latency, called Worst-case Traversal Time (WCTT), of each packet must be lower than its deadline. The goal of this section is to introduce a number of NoC concepts that impact the latency and the design complexity. First, let us see what is the latency metric?

2.3.1 Latency and Quality-of-Service

The measurement of the end-to-end network latency is either at the packet or the message level. Thus, the latency of a packet is the time between sending the first flit at the source and receiving the last flit of this packet at the destination. Similarly, the latency of a message is the time between the transmission of the first packet and the reception of the last one.

However, the latency of a packet is the sum of two components [LRV06]:

1. The conflict-free delay;
2. The blocking delay.

The conflict-free delay includes the router delay, the wire delay and the distance between the source and the destination. The blocking delay results from the possible contention between packets transmitted on the NoC. However, these delays depend on the choices made when designing the elements of a NoC. But, the degree of freedom is large. For this reason, the key NoCs concepts have to be carefully considered by the designer as they affect all the performance metrics. The latency is an important performance metric as it is often associated with a need for Quality-of-Service (QoS). Actually, a QoS defines a certain level of guarantees that is given for packet transfers. [GDvM^{+}03] identifies two basic QoS classes:

1. **Best-Effort (BE):**

   BE services do not reserve any resources, and hence, provide no guarantees on latency and throughput. Thus, it optimizes the average network resources usage.

2. **Guaranteed services (GS):**

   In this service, a number of mechanisms are used to allocate the network resources to ensure fixed throughput and/or latency, regardless of network load.

In the context of real-time systems over NoC, the WCTT must be computed. Thus, in this context, either we use different analytic methods to compute the WCTTs of packets in BE NoCs,

or we use GS NoCs where the analysis of WCTT is more simpler as it leads to straightforward WCTT values.
The way in which a NoC concept is implemented can determine the QoS class offered for the applications on the NoC. Besides, it impacts the performance of a NoC on several metrics, such as the latency, the throughput, the area, etc.

2.3.2 Main concepts impacting the latency

The need for a QoS affects the way in which NoC concepts are implemented. Let us first see what are the essential NoC concepts that are used to provide a QoS class.

Switching techniques

The switching strategy defines how the flits are transmitted and stored by the routers. There are some points that affect the choice of the strategy, such as the cost, the granularity of data to be transmitted, the complexity of the router and the need of the Quality-of-Service (QoS). The switching techniques are divided into two categories: the circuit-switching and the packet switching [MMM$^+$03].

1. **Circuit switching:** this strategy is usually used to provide guaranteed services, i.e. guaranteed latency, as it reduces the blocking delays. In fact, in this strategy, the connection between two communicating nodes is reserved before the message is sent by the source node. This reservation is only released when the transfer of data is complete and so received by the destination. Thus, the resources (links, buffers) are only used by this communication and could not be used by another pair of nodes willing to communicate via the same path. This strategy is more adapted to transfer large payload in order to compensate the negotiation time to establish the connexion. Here, we can mention SoCBus [WL03] that aims to achieve the real-time guarantees by implementing the circuit switching.
   The advantages of this strategy is that the available bandwidth and the latency of the

message are known. However, one of the disadvantages of circuit switching is the inefficient bandwidth usage.

2. **Packet switching:** In this strategy, packets are injected into the network as soon as the network can accept them, i.e. based on local availability information, without waiting for path setting before sending packets. Thus, in this case there is no guarantee that the path will be entirely available from the source to the destination. Each router takes the decision if a packet should be forwarded to the corresponding output port or whether it should wait in case of unavailability of the port. Thus, a control flow mechanism is required. In case of the unavailability of the port, packets are stored in buffers at intermediate routers.
   An advantage of this strategy is the shared network between the communications compared to the circuit-switching where a communication is blocked until the release of the active connexion. Besides, there is no delay to establish a connexion as opposed to circuit-switching. However, this strategy leads to greater delays than on the circuit-switching due to the possible contentions between packets sharing the same resources. Therefore, the packet switching is rather suitable for traffic with low requirements of QoS, i.e. BE. However, it could provide different QoS classes by implementing with specific arbitration mechanisms as we explain later. There are three basic packet switching schemes for forwarding data: store-and-forward (SF), virtual-cut through (VCT) and wormhole [DYN03]. These schemes are differentiated by their packet transmission granularity, i.e. at the flit level or packet level. SF and VCT schemes are not widely used in NoC architectures as they induce a high area cost of a router. Actually, a router should include at each input port a buffer able to store entire packets. Wormhole scheme is the most widely used scheme by NoC [SKH08] and especially to provide a BE service. For this purpose, we are interested by this scheme of packet switching. Let us see the functionality of wormhole switching.

Wormhole switching:

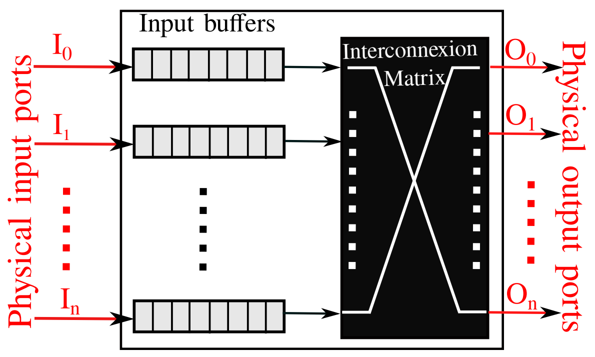
Wormhole switching [Moh98] reduces the buffer requirement at the flit level. Thus, the buffer capacity of the router is a multiple of flits, and so their size is smaller. The packets are transmitted between routers in units of flits, where a flit is transmitted as soon as there is space for one flit in the buffer of the next router. The header flit contains the routing information and the next flits containing data follow contiguously this header in a pipeline fashion. If a packet is blocked, flits of the packet may be stalled on a sequence of routers.

This technique provides a low router latency because they do not have to store the full packets. In addition, the area cost is reduced because the queues are much smaller. However, a packet may occupy several routers at the same time. In this case, it is possible to block the transmission of other packets, leading to a high level of congestion and inefficient use of channel bandwidth due to chained blocking. This chained blocking could lead to deadlock where messages wait for each other and no one can advance any further. To avoid deadlock, specific buffering and/or routing schemes are combined with the wormhole switching. Wormhole switching can provide more efficient network channel utilization. For instance, Tilera [Til11], Mango [BS05], Teraflops [HVS+07], Aethereal [GDR05] and Kalray [dDvAPL14] NoCs implement this mode.

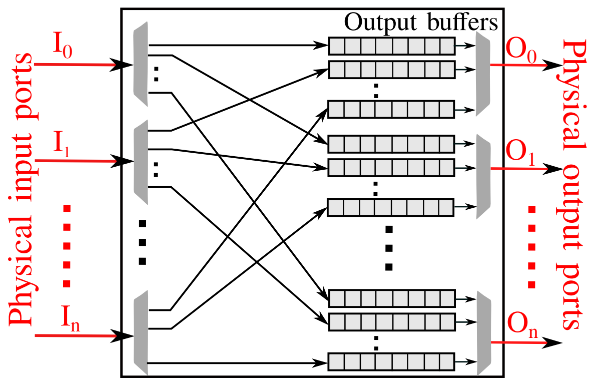
Buffering

The buffering determines the capacity of storage at the input or output ports of a router. Increasing buffering capacities can improve the latency and the throughput at the price of the cost of area and power. The relative positioning of the buffers at input and output ports of the router is performed using various strategies. In this section, we distinguish three main strategies used in NoC design: input queuing, output queuing and virtual output queuing which are also called by virtual channels [RGR+03].

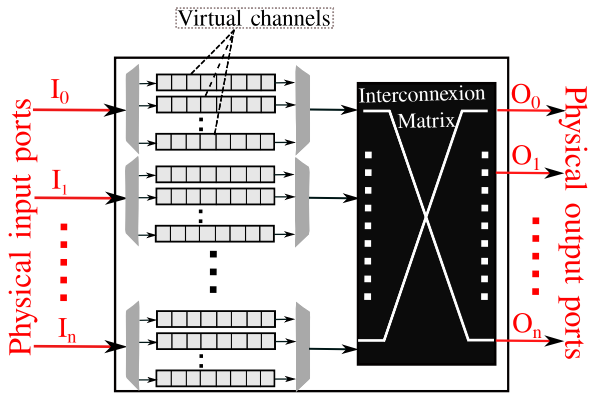
1. **Input queuing:** In this strategy, N queues are placed at the input ports of the router as illustrated in Figure 2.2a. We note that the router presents N input and N output ports. An arbiter determines when an input queue is connected to an output port so that no conflict occurs. Although this technique is the least expensive on the surface, it



(a) Input queuing



(b) Output queuing



(c) Virtual output queuing

Figure 2.2: The different buffering strategies.

can induce to the Head-of-line blocking problem [KHM87]. This happens when a given data in the head of queue cannot access the associated output port, thus blocking other packets in the queue, even though their output ports are free.

2. **Output queuing:** When the queues are placed at the output of the router as shown in Figure 2.2b, each output port has a number of output queues equals to the number of input ports. This technique presents higher performance than input queuing but it increases the area cost: for a router of N input ports and N output ports, we need N^2 queues.

   These two previous schemes are usually used to provide BE services. However, in order to reduce the waiting delay from which a flow can suffer by waiting a place in the next queues, virtual channels are used.

3. **Virtual output queuing or virtual channels:** Figure 2.2c illustrates this strategy which combines the advantages of the input and output queuing. Thus, for each input port, there are a number of input queues that help to buffer the incoming packets in function of their destination or of a level of priority. This strategy is also called virtual channels (VCs) [Dal90, MTCM05] as for each physical channel, i.e. the link between two nodes, there are a number of VCs that share the bandwidth of the link. Thus, in this way, a blocked packet can be doubled by another packet sharing the same physical channel, but stored in another VC. It is used to solve the Head-of-Line blocking problem and to increase the router throughput. Although it implies an increase of the area, VCs have several advantages. They are deadlock free and support guaranteed traffic. Thus, this buffering is either coupled with circuit-switching to provide guaranteed latency, as in MANGO NoC, or with the packet switching. However, in the latter case, it should be implemented with the priority arbitration (which is explained in the next paragraph) by using a priority level for each VC to ensure a low latency for the high priority traffic.

Arbitration mechanisms

An arbitration mechanism is a way to address the contention problem. At each router, when more than one packet on different input ports, compete for the same output port, the arbiter chooses which input port will transfer the packet. Besides, the arbitration mechanisms are required when VCs are implemented in order to select which VC transfers a flit on a link. In the following, we describe the arbitration mechanisms used in NoCs.

1. **Time Division Multiple Access (TDMA):** This policy is usually coupled with the circuit-switching mode in order to ensure a guaranteed latency (GS) where it leads to straightforward WCTT. This policy divides the time into time-slots. Each time-slot is assigned to a connexion between an input and an output port. Thus, during a time-slot, an input port can have the full access to an output port. However, when these time slots are not carefully aligned, higher latencies cannot be avoided [LRV06]. The Aethereal NoC uses the TDMA with the circuit-switching to ensure guaranteed services.

2. **Priority-based:** This policy assigns to each packet an individual priority during its transmission from the source to the destination. Thus, when multiple packets on different input ports contend to the same output port, the packet with the highest priority is granted first. This technique is usually used in real time NoCs where the packets with the highest priority must meet their deadlines. It is especially used with virtual channels where to each VC is assigned a priority level. For example, QNOC [BCGK04] chooses this strategy with virtual channels to provide real-time requirements. However, this policy could lead to a starvation for low priority packets.

3. **Round-Robin (RR):** The RR policy assigns priorities to each input port. In each round of arbitration, the requested input port with the highest priority is served first. But, the request that is served has then the lowest priority in the next round of arbitration [SRM13]. In this way, this policy ensures a fairness between the requested communication from input ports and gives equal chance to the competitors. Besides, it ensures a faster access to an output port than the TDMA policy. Actually, a source port

has to wait its time slot to send packets even when there are any competitors to this output port. The drawback of this policy is that does not differentiate the quality of services demanded by the different packets. Thus, it is usually used in BE NoCs. However, the RR policy is widely used in NoC architectures [TSSJ14] because it is fair and prevents starvation. For example Teraflops [HVS$^+$07], Aetheral [GDR05], Tilera [Til11], Kalray [dDvAPL14] and Mango [BS05] NoCs use this strategy.

To summarize, a GS is provided either by the use of a circuit-switching usually coupled with TDMA arbitration and VCs or the implementation of wormhole switching coupled with a priority arbitration and VCs. A BE NoC usually uses the wormhole routing, the RR arbitration and input buffering.

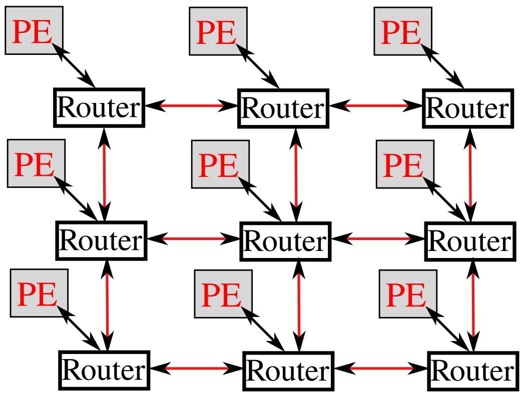
2.3.3 Main concepts impacting the design complexity

There are other concepts that have an impact on the design complexity of the NoC, such as the topology, the routing algorithms and the flow control. However, The choice of an implementation of these concepts is not inclusive to a certain QoS class. Let us see what are the different possibilities of these concepts implementations and which one is the most used in the NoCs architecture.

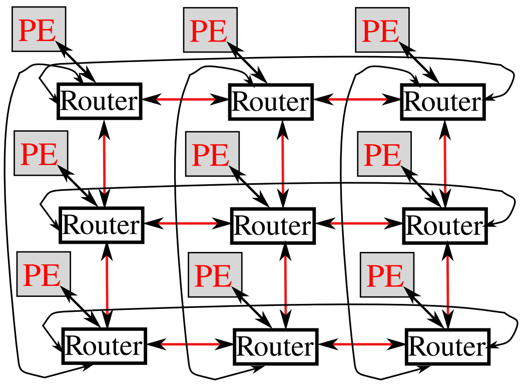
Topology

The Topology of a network defines how the routers are interconnected using network links. The choice of the topology directly impacts the timing performance and the area. It is the first step in the design of the network as it defines the routing strategy used [BC06].

There are a wide variety of network topologies [DMB06] that are used in NoCs. 2-D mesh and torus topologies, illustrated in 2.3, constitute over 60% of the NoC architectures [SKH08]. They indeed allow to define simple routing rules and provide good electrical properties. However, the most common topology is the 2D-Mesh [IG13], because of the following reasons:



(a) 2D Mesh



(b) 2D Torus

Figure 2.3: Example of topologies that are most used in NoCs.

1. It has an acceptable wire cost;
2. It presents a reasonably high bandwidth;
3. It is easy to group IPs that communicate a lot so that they do not consume any unnecessary high amount of resources in the network.

Routing algorithms

The routing algorithm, implemented at each router, determines which path the packet should take through the network, i.e. the output port from which the packet should be delivered. The selection of the routing algorithm depends on several factors such as the implementation complexity and the performance requirements [OHM05]. Thus, a compromise is needed between an optimal use of the network communication channels and a simple implementation of an algorithm that does not require excessive hardware resources.

A deterministic routing is preferred by NoC designers: according to [SKH08], it is used by more than 70% of the NoCs. It is indeed a simple routing algorithm where the path taken by the packet is completely known in advance for a given source/destination pair [DA93]. Then, it is simple to implement and it is inexpensive. Besides, it presents a low latency when the network is not congested. It is deadlock free when wormhole switching is used. As an example of deterministic routing is the XY routing, which is the popular routing algorithm

considered on 2D mesh or torus NoCs [NM93]. An adaptive routing is more complicated to implement [BDM02b] as it provides multiple paths to route a packet from its source to its destination. It may lead to network deadlock or livelock or both. Livelock occurs when a packet does not arrive at its destination and rotates round in the network.
These algorithms can also be determined either at the source (source routing) or constructed sequentially in routers (distributed routing). In the source routing, the source determines the whole path of the packet which will be indicated in the header. In the distributed routing, each router chooses the next destination in function of the final destination. The source routing is simpler than distributed routing but cannot adapt paths in case of traffic congestion as the paths are pre-computed offline. As an example of NoCs that uses the source deterministic routing, we can cite Tilera [Til11] and Mango [BS05] NoCs.

Flow control

The choice of the flow control strategy depends on which switching and buffering schemes/strategies are used. Because of the limited capacity of buffers, a flow control is necessary to avoid buffers to overflow and thus to reject some data. Thus, flow control ensures that a router cannot send any data to the next router if there is not enough space available to buffer it. The common control flow mechanisms proposed in the literature are the followings: ON/OFF [CPC08], credit-based protocol [BS04, DRGR03, RDG$^+$04] and ACK/NACK [PABB05]. The ON/OFF strategy minimizes the amount of the backpressure signaling by sending only a single control bit that indicates to the upstream router whether it is allowed to send (on) or not (off). In the credit-based mechanism, there is a counter in each upstream router to keep track of the number of free flits within the downstream router. Thus, a router can send a number of flits corresponding to the number of credits. Finally, in the ACK/NACK protocol, the logic flow control must acknowledge each flit received by the receiver port.
In the literature, the credit-based scheme is the most widely used in NoCs because of its high performance with limited buffering [Gai15].

2.4 Assumed NoC architecture

This chapter has introduced the different concepts of NoCs and how they can be implemented to provide different classes of QoS. Thus, designing a NoC is difficult as it depends on a large number of parameters. This design is affected by the QoS needed and by a trade-off between the complexity/cost and ensuring a minimum throughput/latency. In order to give a general view on the state-of-the-art on the NoCs, we present a table summarizing a set of NoCs architectures (Table 2.1). For a more exhaustive list, [SKH08] lists sixty architectures of NoCs. We classify these works into the following types: BE (Best-Effort) NoCs providing a good average performance, GS (Gauaranteed-Services) NoCs providing hard real-time requirements or GS and BE NoCs where the routers are more complex as they include two switching mode.

The majority of NoCs uses the mesh topology which is explained by the fact that it is the simplest and most flexible solution to implement. Some NoCs rely on torus topology in the aim to reduce the network diameter which could reduce the latency. Also, we can identify that the round-robin scheme is predominating other arbitration policies. Mostly about 70% of NoCs [SKH08] use the deterministic routing due to its advantages over the adaptive routing as mentioned before. Besides, in order to reduce the cost area of routers and especially buffers, virtual channels and output queuing are avoided.

However, the parameters that are directly related to the QoS are the switching mode and the arbitration mechanism. We can notice that wormhole switching with the round-robin arbitration are the most common mechanisms used in BE NoCs, as it provides lower latency, smaller and faster routers than other techniques [LRV06]. In GS NoCs, the virtual or pure circuit switching is more used and it is coupled with TDMA to ensure a higher predictability for the network as Aethereal [GDR05] and Nostrum [PJ06, MNTJ04]. Other NoCs use the packet switching mode with the priority-based arbitration to ensure guaranteed services such as QNoC [BCGK04] and Faust [DBL05] NoCs. Thus, in order to provide guarantees to hard real-time traffic, GS NoCs present a complex hardware architecture. Thus, no contentions can occur by design, leading to straightforward WCTT for packets but with penalizing the average performance. Besides, none of these NoCs targeting hard real-time constraints are available in commercially existing

		Topology	Switching mode	Buffering	Routing	Flow control	Arbitration
BE	**SCC [HDV$^+$11]**	Mesh	Virtual-cut through	VOQ (8 VCs)	Source deterministic	-	WWFA
	Teraflops[HVS$^+$07]	Mesh	Wormhole	VOQ (2 VCs)	deterministic adaptive	On/OFF	RRA
	Tilera [Til11]	Mesh	Wormhole	Input	Source deterministic	Credit-based	RRA
	Kalray [dDvAPL14]	Torus modified	Wormhole	Output	Source deterministic	source-based	RRA
	Proteo [STAN04]	Ring	virtual cut-through	-	Source deterministic	-	-
	Xpipes [BB04]	-	wormhole	-	Source deterministic	-	-
	Spin [AG03]	Tree	wormhole	-	Distributed adaptive	-	-
	Octagon [KND02]	Ring	wormhole	-	Source deterministic	-	-
GS	**SoCBUS [WL03]**	Mesh	Circuit	Input	Didtributed Deterministic	ACK/NACK	RRA
	Faust [DBL05]	Mesh	wormhole	VOQ (2 VCs)	source deterministic	ACK/NACK	Priority-based
	QNoC [BCGK04]	Mesh	wormhole	VOQ	Source deterministic	credit-based	Priority-based
GS + BE	**Aethereal [GDR05]**	Mesh	Wormhole + Circuit	Input	Source deterministic	Credit-based	RRA + TDMA
	MANGO [BS05]	Mesh	Wormhole + Circuit	VOQ (8 VCs)	Source deterministic	Credit-based	RRA
	Nostrum [PJ06]	Mesh	Store&forward + Circuit	VOQ (8 VCs)	Adaptive	-	TDMA

Table 2.1: Table reporting some examples of NoC architectures.

many-core architectures.

In hard real-time systems, analytic methods are used to compute the WCTT in BE NoCs. For instance, Tilera [Til11] and Kalray [dDvAPL14] are commercially BE NoCs. Besides, this class of QoS optimizes the average network resources usage. For this reason, we base our work on an existing commercial NoC architecture and we choose a Tilera like NoC. Tilera is a commercial BE NoC implementing the most common used options (Mesh topology, Wormhole switching, Round-Robin arbitration, input buffers). Besides, the documentation on this architecture was available when this study was made.

In this work, we are interested in real-time packet schedulability analysis for many-cores networks. Thus, we need to analyze the worst-case behavior for both inter-core and core to external memories or peripherals communications by taking into account the different contentions that could occur between packets. The next chapter presents a state-of-the-art of the existing methods that are used to analyze, compute and reduce the WCTT.

Chapter 3

Related works around Worst Case Traversal Time (WCTT)

Contents

Chapter 2 has shown different NoCs architectures that provide different classes of QoS. Regardless of the NoC architecture, in distributed hard real-time systems, the WCTT of all the packets generated by a flow must be lower than a predetermined deadline. Such real-time packet schedulability analysis have been done for various types of networks by taking into account the type of contentions that can occur between flows. The challenge lies in the ability to define analysis techniques that have a limited complexity, in order to give results in a reasonable amount of time, and at the same time compute WCTT values that are not too pessimistic. The complexity of the analysis depends on the NoC concepts implemented. Virtual circuit switching relying on a TDMA leads to WCTT that can be easily computed. In fact, no contention can therefore occur. However, in wormhole networks, the analysis of the system behavior becomes more complex due to different types of interferences. Thus, in this chapter, we focus on approaches used to compute the WCTT over wormhole networks. Even we have assumed a BE NoC architecture in the remainder of the work, we focus in this chapter not only on BE wormhole networks but also on priority-based wormhole networks. Actually, we are interested to present how the existing approaches model the behavior of the wormhole switching. Besides, we show how they consider the different types of interferences in the computation of the WCTT. On the other hand, the WCTT depends on the mapping of the application. Thus, another challenge lies in the ability to reduce the congestion incurred by the flows, whether belonging to the same application or not. At the end of the chapter, we present a state-of-the-art on the mapping strategies used to reduce the contentions between flows and thus their WCTT.

3.1 WCTT over wormhole networks

In this section, we present a state-of-the art on the approaches used to analyze the WCTT on priority-based wormhole networks and BE wormhole networks.

3.1.1 Priority-based wormhole networks

Real-time communication over wormhole networks were studied before NoCs were introduced. The most important mechanism in a wormhole real-time network is the use of virtual channels. Moreover, coupled with a mechanism for preemptive priorities, it allows a message to temporarily stop the transmission of another lower priority message.

The first approaches to upper-bound the latency of transmitted packets over a priority-based wormhole switching with virtual channels were proposed in [Mut94, HO97]. Using these approaches, the priority of flows is mapped over the priority bits of VCs. At each router, links are assigned to VCs of the highest priority that have a flit ready to be transmitted. [HO97] presents a feasibility test that determines whether a set of messages can be transmitted in a given network respecting the deadlines of all messages. Each message is periodic and has a different priority level. Each physical link includes a virtual channel for each message that uses it. The principle to compute the worst-case delay of a message m_i is to compute the delays of transmission of the messages m_j having higher priorities than m_i and sharing at least one link with m_i. Moreover, as the messages are periodic, multiple instances of the same message can block m_i. The authors propose to consider the entire path of a given message as a single shared resource, i.e. as one link which is must be free of any message of highest priority than m_i. Kim et al. [KKHL98] noticed that neither of those approaches consider the impact of indirect interference. An indirect interference happens when a message shares at least one link with m_j but does not share any link with m_i, and thus could have an impact on the latency bounds of m_i.

[KKHL98] proposes an algorithm to compute WCTT of flows over priority-based wormhole networks. Compared to [Mut94, HO97], the authors use a dependency graph between flows to identify both the direct and indirect contentions a flow may suffer. When an indirect flow is identified, the accuracy of WCTT values is improved, as the blocking effect of indirect flows can thus be modeled to occur only when other (periodic) blocking flows are released simultaneously. Otherwise, this indirect message can not interfere and its impact is not counted in the computation of the upper bound. However, [Shi09] shows that this method still introduces excessive interference to the studied flow. Actually, they show on an example that possible parallel communication will take place on disjoint links which reduces the possible interference from higher priority flows.

[LJS05] proposed another improved algorithm. The defect in the approaches in [Mut94, HO97] is that they aggregate all the links used by the studied flow and the flows that interfere with it in a single resource. This resource must be fully reserved for the studied flow in order to be transmitted. Thus, if two flows in conflict with the studied flow do not intersect, they can be transferred simultaneously. Therefore, the authors in [LJS05] distinguish direct from indirect contentions but by utilizing a contention tree. This tree is further used to identify when parallel transmissions of flows on disjoint concurrent paths can occur, reducing the contention interferences.

Priority-based wormhole was also applied in the context of NoC. [SB08] is based on previous works and provides another method for verifying the schedulability of a set of messages. It presents a method for computing the WCTT of flows which integrates both direct and indirect interferences of higher-priority flows. The computation proposed by [SB08] is based on the assumption that a message is undergoing its worst-case delay if it is transmitted at the same time with all higher priority messages. The authors then provide an iterative method and separately compute the delays due to direct and indirect interference. Indirect interferences are treated as additional release jitter on flows that directly interfere with the analyzed flow. They also show that their method provides tighter bounds than [HO97], but it is not optimal when simultaneous interference exists. Besides, they show that values obtained in [LJS05] can be optimistic, as the worst case scenario do not occur when flows are released simultaneously. However, the authors imply that it is not possible to obtain an exact bound in this case.
Both algorithms proposed in [SB08] and [LJS05] assume that all flows that are in indirect contentions have an impact.

[Shi09] is an evolution of [SB08] in the aim to reduce the number of priority levels necessary to the proper functioning of this type of network. Indeed, the virtual channels are expensive in terms of memory and power consumption. So the authors propose to share a given priority level between several messages. They also provide a method to compute the transmission delay of a flow in this model. The principle of the computation is as follows: at each priority level, they treat all flows having this level as a single message that must reserve all links used by the messages of the same level in order to be transmitted. The WCTT for this "aggregate message" is then calculated according to the method presented in [SB08]. The delay for a message is given by the sum of the delay for the "aggregate message" and the delay for the message from the source node.

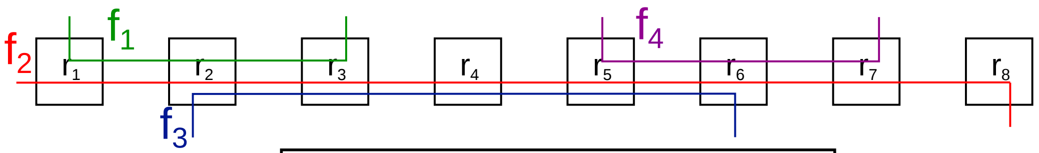
More recently, [NYP14b] analyzes WCTT of flows of applications designed following a partitioning approach but where runtime migrations are decided by the applications themselves. As paths of flows cannot be known at design time, the size of the search space of the worst-case scenario is higher. Compared to [SB08], a lower complexity but more pessimistic method is thus used.

3.1.2 BE wormhole networks and recursive methods

In contrast to a wormhole real-time network, a wormhole BE network does not provide mechanisms dedicated to respect the real-time constraints, such as priority mechanisms. In the context of BE NoC, [QLD10] extends the contention tree used in [LJS05] to identify three basic contention patterns. Indeed, they consider that the shared resources in the network are of three types: the credit flow control, the links and the routers input buffers. Thus, complex contentions are then decomposed into these patterns and Network Calculus (NC) [LBT01] is used to compute WCTT of flows. However, [FFF11] clearly showed that this method leads to over-dimensioning the resources.

[Lee03] presents a recursive algorithm to compute the WCTT of flows induced by contentions on the path of the analyzed flow. It presents an algorithm called WCFC (Wormhole Channel Feasibility Checking) which computes a bound on the worst-case delay of a flow and checks that its deadline is fulfilled. This computation is based on a recursive formula that takes into account the contentions in routers traversed successively by a packet and the contentions incurred by the downstream packets that block it. Indeed, WCFC does not assume the existence of a round-robin mechanism as used in most BE NoC routers and thus gives more pessimistic values of the WCTT bound.

In [RMB$^+$09, RMB$^+$13] and [FFF09b], the tightness of the WCFC method [Lee03] is improved as the round-robin arbitration is modeled. Therefore, only a single flow per port in contention can use the link before the analyzed flow. Indeed, the authors in [RMB$^+$09, RMB$^+$13] present two methods to compute the worst-case delay in a BE wormhole network. The first, called LL-RTB (**R**eal-**T**ime **B**ound for **L**ow **L**atency traffic), assumes that the interval between two successive packets of the same flow is important enough that packets do not interfere with them.



$$\boxed{\begin{aligned}&delay(f_1) = d(f_1) + delay(f_2) + delay(f_3)\\&delay(f_2) = d(f_2) + delay(f_3) + delay(f_4)\\&delay(f_3) = d(f_3) + delay(f_4)\\&delay(f_4) = d(f_4)\end{aligned}}$$

Figure 3.1: An example illustrating Recursive Calculus method.

The second method, named HB-RTB for **R**eal-**T**ime **Bound** for **H**igh **B**andwidth traffic, is similar in principle to the first one, but is based on a different traffic hypothesis. Here, it is assumed that successive packets of a flow are transmitted continuously, as long as the network can support them. Furthermore, it assumes that, initially, the network is already fully saturated by packets. This method leads to higher bound values than RTB-LL, but it is more general because it does not require any assumptions about the frequency of packets or a minimum interval. This method was introduced simultaneously with the Recursive Calculus (RC) method presented in [FFF09b]. They are both based on computing the worst-case delay of a packet recursively. They analyze the contention incurred by the analyzed flow, by considering that all the intermediate buffers in the routers between the source and the destination are filled to their capacities. Besides, they consider also that the packets can be injected into the network continuously. A packet delivery is assumed to be divided into two phases. In the first phase, the header of the packet is routed to its destination and creates a virtual circuit between the source and the destination. In the second phase, the whole packet is then transferred. The method recursively analyzes the contention in the path of the analyzed flow f. At each router, the direct flows are identified and their delays are computed by thus taking into account the indirect flows of f. As a round-robin arbitration is assumed, at each input port, the maximum delay from the set of flows in direct or indirect contention with f is added.

Figure 3.1 illustrates an example of the computation of the WCTT using RC method. In this example, we consider f_1 the analyzed flow, f_2 and f_3 are direct flows to f_1, while f_4 is an indirect flow. f_1 is blocked by f_2 at r_1 and by f_3 at r_2. Thus, the delays of f_2 and f_3, noted by $delay(f_2)$ and $delay(f_3)$, are computed recursively and added to the transmission delay of

f_1, i.e. $d(f_1)$. Actually, f_2 is blocked by f_3 at r_2 and by f_4 at r_5. Then, to compute $delay(f_2)$, the delays of f_3 and f_4 are computed and added to $d(f_2)$. Similarly, $delay(f_3)$ and $delay(f_4)$ are computed. Then, $delay(f_1) = d(f_1) + d(f_2) + 2d(f_3) + 2d(f_4)$.

[FFF12] shows that, in general, NC gives tighter WCTT values than RC. However, RC is less pessimistic than NC when the network is saturated. As in [Lee03, RMB$^+$09, RMB$^+$13], this method supposes that packets must reach their destinations before the next one can progress.

Recently, [DNNP14] introduces a branch-and-prune approach and identifies two main sources of pessimism of the initial RC method over best effort NoC. Indeed, this approach is based on the RC method but it gives tighter WCCT values than RC by reducing the occurrence of a flow to block the analyzed flow. Indeed, an analyzed flow could be blocked directly by two flows f_i and f_j respectively at routers r_i and r_j. However, as a recursive method is applied, thus, if f_j blocks also f_i, then the delay of f_j is added twice. To overcome this pessimism, the method in [DNNP14], introduces for each flow, constraints on the inter-release arrival of packets and the number of packets that can be emitted for a given interval.

All the approaches introduced in this section assume that flows in direct contention must reach their destinations before the analyzed flows can progress, which is pessimistic. Actually, these methods do not model the wormhole switching at the flit granularity.

3.2 Contention-aware mapping

Several methods are used in NoCs to allocate hard real-time applications. However, how hard real-time tasks that generate the flows (critical and non-critical) are mapped within cores, is of utmost importance to control the contention over the NoC and thus the WCTT of flows.
Several contention aware mapping strategies are proposed in the literature. We classify these strategies into two groups: task mapping and application mapping. The task mapping presents the strategies used to allocate a single application. It aims to reduce the congestion of the inter-core communications belonging to this application, called internal congestion. The application

mapping presents the strategies to allocate several applications. It has a goal to reduce not only the internal congestion but also the congestion between applications, called external congestion.

3.2.1 Task mapping

Different strategies exist to reduce the number of contentions that a flow can experience on its path, when allocating an application on the NoC, using minimization functions. A link contention aware mapping problem has appeared in [CM08], where an **I**nteger **L**inear **P**rogramming (ILP) is proposed. It aims to minimize the contention that flows exhibit at each intermediate cores between their sources and their destinations. The authors first analyze the factors that produce a network contention. These factors may be a source-based, a destination-based or a path-based contention. The source-based is due to the flows sharing the same source, while the destination-based are those going to the same destination. The analysis shows that minimizing the path-based contention due to the flows sharing only some links, leads to minimize the network contention. Thus, the authors propose an ILP formulation that minimizes the cost function of flows that share a subset of their paths and have different source and destination cores. The objective function has two parts: 1) a weighted communication distance term, 2) a term emphasizing the link contention of on-chip communication. As this is an NP-complete problem, they proposed to keep only the first term and then reducing only the distance between communicating tasks, which will decrease the number of links shared by the inter-cores communications. The authors claim that by minimizing this distance, the packet latency will eventually decrease.

The authors in [ZM12] propose a low-contention mapping algorithm for real-time applications. They developed a TDMA-like approach to ensure a separation of communications on NoC. Thus, the communications are allocated within several time frames, which minimize the number of interference communications. In order to minimize the contention due to inter-process communications (IPC) sharing the same time frame, an ILP approach based on the number of links shared by flows is used, similarly to [CM08].

[RI12] uses a genetic algorithm to explore the search space of contention aware mapping of tasks. A genetic algorithm is a stochastic search algorithm based on operations of natural genetics. Here, fixed-sized population of chromosomes evolves over a number of generations following the principle of natural selection. Natural operations, such as crossovers and mutations, are represented by operators that presents this behavior. A fitness measure is associated to each chromosome which identifies a potential solution. In [RI12], the authors investigate the effectiveness of genetic algorithms for static task scheduling in wormhole Network-on-Chip-based systems. They explore the mapping of tasks as well as the priority ordering of the task set. They define a fitness function which select the mapping where all communications of the application meet their deadlines. Indeed, they select randomly two or five mappings (populations). Each mapping is presented as a chromosome: two branches presenting the index of the task and the index or coordinates of the core. Then, crossover and mutation operators are applied on these chromosomes. The chromosomes are selected to participate in the crossover operation. Their parts are exchanged to create new offspring. The mutation operator is implemented by selecting first a parent chromosome, then changing randomly some of its portions. Then, they use flows response times analysis of each mapping that model contentions as fitness function. This function measures the number of unschedulable flows, and thus the mapping which presents the minimum number is chosen.

A tree-model based contention-aware mapping which takes the bandwidth constraints into consideration is proposed in [YGSP12]. It abstracts the NoC into an extend tree by traversing the tiles from the central one. The most communicating task is mapped to the root. Thereafter, at each step, the task having the largest communication volume with the mapped tasks is selected and mapped to one node from the next level of the tree. This node is chosen in such way that the bandwidth constraints are respected. Thus, as [ZM12] and [CM08], this method minimizes the distance of the paths between communicating tasks in order to reduce potential contentions.

All these approaches consider the mapping of a single application. However, a NoC presents a large number of computing cores, and a hardware that permits to allocate a mix of real-time applications. Then, these strategies cannot be used to allocate several applications as they do not consider the external congestion. For instance, the underlying genetic algorithms used in [RI12] should be extended to enable the coexistence of different fitness functions, so that mapping several applications of potentially different levels of criticality can be supported.

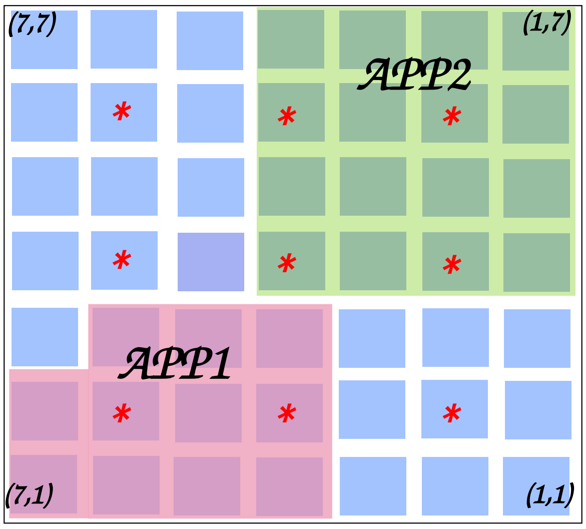
3.2.2 Application mapping

Several mapping strategies deal with the problem of reducing the internal and external congestion. The mapping area selection of each application is of the utmost importance. It impacts the mapping of the other applications as we will present in this section.
These strategies divide the mapping into two phases: the application mapping and the task mapping. The application mapping consists on finding the region where to allocate the application. Thus, a first core is selected to define this region. The task mapping uses different heuristics to allocate the tasks of each application. Actually, it maps a first task on the first selected core, and different mapping heuristics are used to allocate the remaining tasks around the first one.

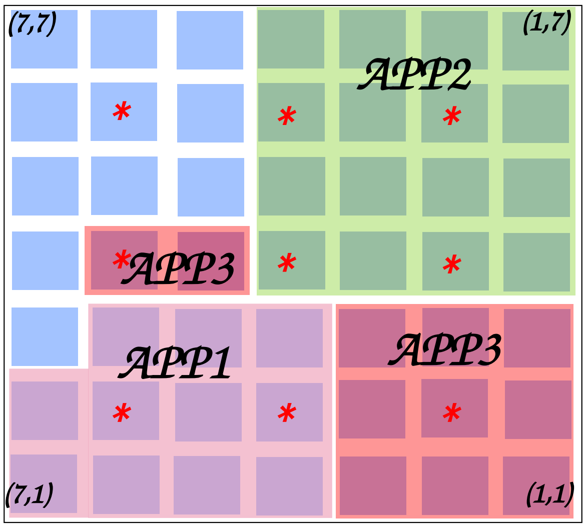
[dSCCM10], [CCM07] divide the NoC arbitrarily into clusters for the application mapping. Each cluster is dedicated to an application. An initial task in each application is selected and is placed inside a cluster. [COM08] enhance the definition of clusters for applications by making them near convex regions. Thus, generating non-contiguous regions is avoided, which reduces the external congestion. Then, within each cluster a congestion-aware mapping heuristic, similar to one in [ZM12], minimizes the bandwidth utilization of NoC links and the distance between communicating tasks. [FRD+12] shows a problem in these methods: the first core selection policy when building regions, leads to map an application over a fragmented region.

In order to solve this problem, [FRD+12] proposes a solution, called CoNA, to select the core having the most available neighbors (up to 4) as the first core in the allocation. It aims to avoid region fragmentation and thus decrease both internal and external congestions. The task mapped onto the first core is the one with the largest number of communications. Then, the task graph is traversed in breadth-first order from this first task allocated. The subsequent tasks are mapped to the cores that fit into the smallest square centered in the first core. However, CoNA cannot guarantee that the first core is always surrounded by enough free cores as it only considers direct neighbors when selecting the first core and then still leads to fragmentation of areas.

A mapping performance analysis has been made in [FDLP13] to show the impact of the first core selection. The authors illustrate the drawbacks of the previous mentioned strategies of mapping. They show that these methods lead to allocate applications into fragmented regions on the NoC, which increases the external congestion. Figure 3.2 shows an example of such a fragmentation when applying previous methods. Figure 3.2a illustrates the mapping of two applications, noted App_1 and App_2 using the previous methods. The asterisks present the



(a)



(b)

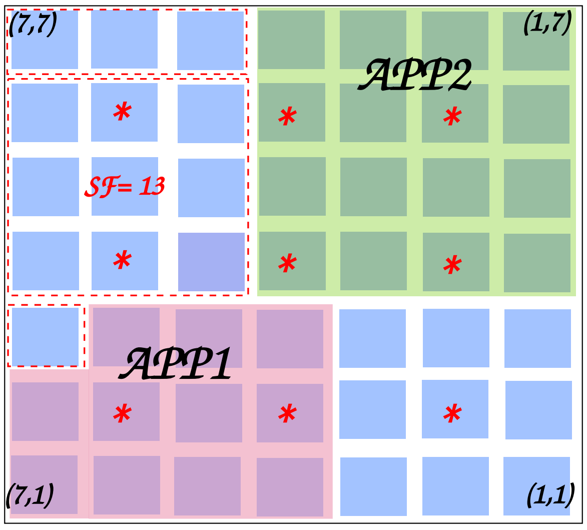
Figure 3.2: Allocating App_3 using the mapping strategies presented in [dSCCM10], [FRD$^+$12] and [COM08].

starting core within the different clusters defined by the method in [dSCCM10]. These previous methods lead to an allocation of the application App_3 over a fragmented region as shown in Figure 3.2b. Actually, the method in [dSCCM10] and CoNA chooses the core located at (2,2) to be the first core to define the region where App_3 will be allocated. This core has 4 direct neighbors which is sufficient to be selected as first core in CoNA method. The core (1,1) is the first core selected by the method presented in [COM08]. This core is inside a near-convex region. However, in all cases, the first core selection policy does not take into consideration the size of the application to be allocated. Here, App_3 has a size of 11 tasks. Thus, whatever the heuristic used to allocate the tasks within this region, App_3 is fragmented on the NoC. The first contiguous region selected has an area of 9 which is lower than the size of the application.

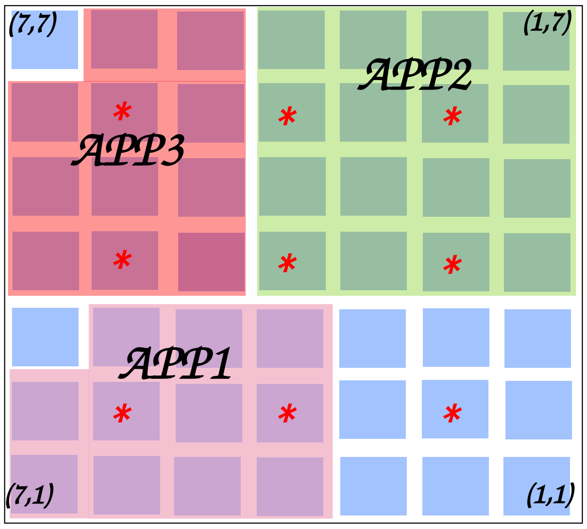
Smart Hill Climbing (SHiC) approach [FDLP13] finds a contiguous near square region having a number of free cores at least equal to the number of tasks of the application. It considers a new metric called square-factor (SF) for selecting the first core when building region to approximate the contiguous available cores around the selected one. Towards the SF calculation of each core, each application already allocated on the NoC is defined by a rectangle that may include all the tasks of an application or not. Thus, the SF of a core is the maximal size of the square area in which that core can be put in, to which the number of free cores around this square is added. This square must not have any conflicts with the other rectangles, i.e. applications already mapped. The first core is then the one having a SF greater or equal to the size of the application to be mapped, i.e. the number of cores that are needed assuming a core can only execute a single task. The remaining tasks are then allocated in this region as in CoNA method.

Figure 3.3a shows on the example introduced previously, the first core selection policy of SHiC method. The core located at (6,5) has an SF equal to 13. In fact, this core is centered on a square of area 9 and presents 4 cores around this square. Figure 3.3b shows how SHiC method leads to allocate App_3 in a contiguous region without been fragmented.

[FRX$^+$14] adapts SHiC so that contiguous regions are used to map critical applications, in order to reduce contentions, while non-critical applications are mapped over non-contiguous



(a)



(b)

Figure 3.3: Allocating App_3 using SHiC method presented in[FDLP13].

regions to increase the system throughput. This leads to allocate two tasks repectively from a critical and non-critical applications on the same core. Thus, it needs to schedule the critical tasks with a higher priority; i.e. the non critical tasks are suspended as soon as critical tasks demand for the system resources.

3.3 WCTT and mapping baseline references

The challenges facing the NoCs are mainly in the ability to analyze the behavior of network in order to fulfill hard real-time requirements. Thus, analytic methods are needed to compute the WCTT of the critical flows exchanged on the NoC. In this chapter, we have presented the existing methods to compute the worst-case delays over wormhole networks. As we have chosen a Tilera-like architecture in chapter 2, we focus then on the methods used over Best-Effort networks. The method presented in [DNNP14] consider all direct and indirect contentions from which a flow suffers. It reduces the pessimism of the existing recursive methods by adding assumptions at the application level. However, we focus first on transmission properties at the network level to reduce the pessimism in the existing methods. Thus, in this work, we refer to recursive calculus method (RC), presented in [FFF09b], as it is the best method that could be used in our assumed architecture.

Besides, the mapping of the tasks of the different applications on the NoC impacts the delays of the flows. Thus, several strategies of mapping are proposed in the literature to reduce the congestion on the NoC which reduces the WCTT of the flows generated by these tasks. These strategies either consider the allocation of an application or several applications. A contention-aware mapping should reduce both the internal and external congestions when allocating different applications on the NoC. The SHiC method, introduced in [FDLP13], is the best multi-application congestion-aware mapping approach that reduces both the internal and external congestions of the inter-core communications. Thus, in this work, we refer to SHiC method as a contention-aware mapping.

The next chapter illustrates the problems when integrating I/O constraints within the NoC communications. We detail the drawbacks of existing mapping strategies and computing methods.

Chapter 4

Managing I/O in many-cores: problems and approach

Contents

In embedded real-time systems, many-cores architectures could be used as processing elements within a backbone Ethernet network, as these architectures present an important number of I/Ointerfaces such as the Ethernet interfaces and the memory controllers. Besides, it is possible to allocate a number of applications of different level of criticalities on the NoC-based many-cores architectures. These applications communicate with sensors and actuators via the

Ethernet interfaces. Therefore, in this context, a NoC supports different types of communications either between cores, or between cores and I/Ointerfaces. The real-time constraints concern not only the inter-cores communications but also the I/Ocommunications. In chapter 3, we have presented the analytical methods to compute the WCTT of flows on different NoC architectures and the strategies to reduce the congestion on the NoC. However, these existing works only focus on the inter-core communications and do not consider the I/Opart. The objective of this chapter is to present the problems from which suffer the core-to-I/Oflows when using NoCs in real-time systems connected to a number of sensors and actuators via Ethernet. Thus, in this chapter, we first present an explanation about the I/Ointerfaces in Tilera NoC. We illustrate how the communications are done between cores and I/Ointerfaces. Besides, we present the model we assume in this work. Finally, we illustrate the problems on a case study made of several real-time applications and show the limitations of existing works.

4.1 System architecture and assumptions

In this section, we first present a description of the I/Ointerfaces in Tilera NoC. Besides, we illustrate the communications between cores and I/Ointerfaces. Then, we show our architectural model based on Tilera NoC, and the different assumptions considered in this work.

4.1.1 Description of I/O in Tilera

A Tilera NoC interconnects cores, Ethernet and DDR-SDRAM memory interfaces that are located on its edges. The term DDR in the remainder of this thesis refers to the DDR-SDRAM memory. In a Tilera NoC, a tile contains only one core. A set of identical controllers are in general spread around the NoC in opposite cardinal directions. For instance, two memory controllers are located north and two south of the NoC in Tilera Tile64, as illustrated in Figure 4.1. However, it only provides Ethernet controllers on the east of its NoC. Finally, each I/O interface can be accessed from the core adjacent to this interface through specific ports. For instance, each memory controller of the Tile64 has 3 ports. The 6 central columns (out of

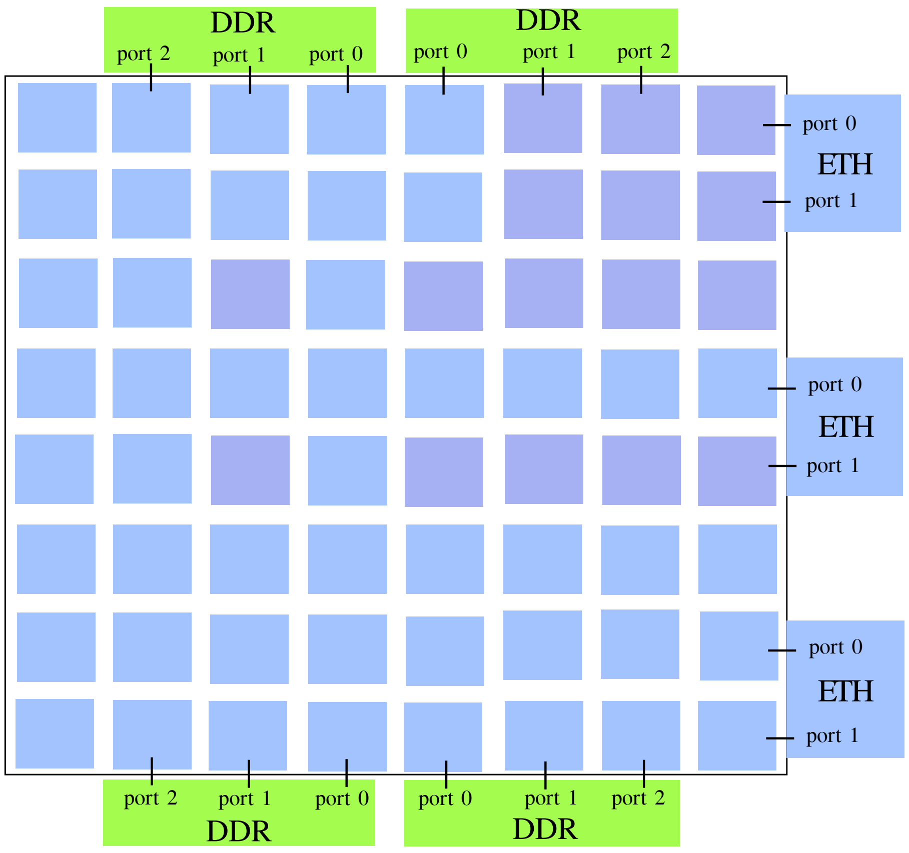


Figure 4.1: An overview of a Tilera architecture.

8) of the Tile64 are connected from both their north and south edges to a port of a memory controller. Each Ethernet controller of the Tile64 is connected to 2 ports respectively.

Now, let us see how an Ethernet frame is transmitted on the NoC. Figure 4.2 illustrates the different steps of the transmission of the ingress data flows. This transmission proceeds as follows:

1. The header and the payload of an Ethernet frame are separated into two buffers.

2. The header is transmitted to a specific tile which determines where the payload will be sent.

3. There are two possibilities to send a payload to a destination core. These possibilities depend on where the data will be stored:

   (a) ***First possibility***

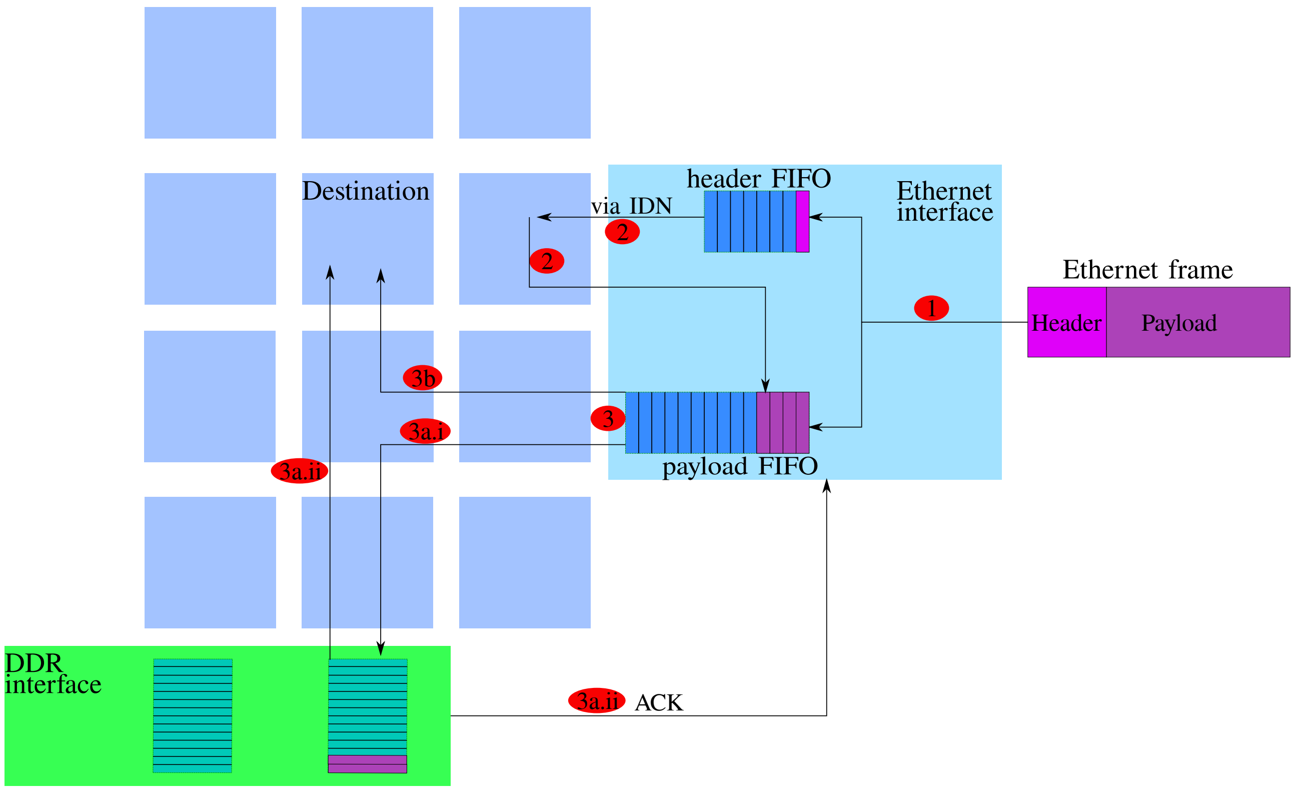


Figure 4.2: The different steps of the ingress data flows.

i. The data is sent to the DDR memory.

ii. Then, the DDR memory sends the data to the destination tile while it sends back an ACK to the I/Ointerface.

(b) ***Second possibility***

i. The data is sent directly to the destination tile.

This transmission is done under different constraints. First, the buffer on the Ethernet interface which stores the payload, has a limited capacity: 2KB for a giga-Ethernet interface. Second, the maximal size of NoC packets is limited and it depends on the network used by these packets. Actually, Tilera provides five dynamic networks to support different type of flows. The transmission of the ingress data flows could use two networks: IDN (I/ODynamic Network) and MDN (Memory Dynamic Network). The IDN is primarily used to exchange data between I/O themselves, and between I/Oand tiles. It supports NoC packets made of 128 flits at maximum. The MDN network supports data transfer between tiles, I/O interfaces and between tiles and DDR memory. It supports packets made of 2 flits at minimum and 19 flits at maximum. Thus,

a core can receive data directly from the Ethernet interface via IDN (step 3b) or through an intermediate memory controller via MDN (step 3a). Besides, the header is transmitted to a specific tile via IDN (step 2). On the other hand, the flits have a size of 32 bits. Then, the size of an Ethernet frame is generally several time higher than the size of a NoC packet. Thus, several NoC packets are needed to transmit an Ethernet payload to a tile.
A similar process is used for the egress data flows where a data is sent directly to the Ethernet interface via the IDN or through the DDR memory via MDN.

4.1.2 Model of NoC architecture and assumptions

NoC Model

We model a Tilera-like NoC as a mesh network that interconnects $L \times W$ routers as illustrated in Figure 4.3. A core of the NoC is identified by its (x,y) coordinates and we assume that the core (1,1) is located on the bottom right of the NoC. The NoC interconnects cores, Ethernet and DDR interfaces.

We assume that a set of tasks is statically allocated over a NoC. These tasks exchange various payloads. A payload is modeled as a flow and we note F the set of m flows of an application. Each flow f_i ($i = 1..m$) is made of successive packets that are transmitted on the NoC.
We consider the following assumptions related to the model of the NoC architecture, the transmission mode and and the mechanisms used in this NoC:

A1. A mesh NoC which tiles are made of a single core. We illustrate such a NoC on a 7×7 grid.

A2. A core can execute only one task.

A3. The packets are divided into a set of flits of fixed size of 32 bits.

A4. The transfers are done only on the MDN network, thus the packets are made between 2 and 19 flits.

A5. The packets are injected at a given inter-arrival time to prevent contentions with previous

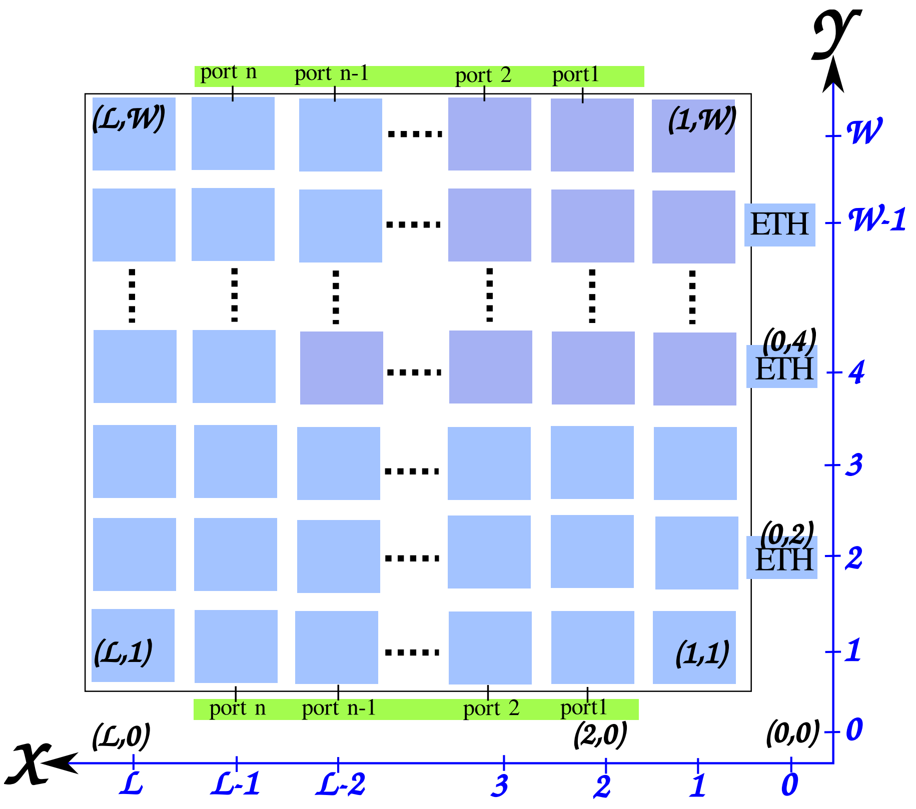


Figure 4.3: A Tilera-like NoC architecture.

packets of the same flow to occur.

A6. The wormhole switching, the Round-Robin arbitration, the credit-based flow control and the XY routing are the mechanisms used on this NoC.

A7. The destination router immediately consumes any flit arriving, as well as forwards back credit to the previous router.

A8. The latency for a flit to be read from an input buffer, traverse the crossbar, and reach the storage at the input of a neighboring switch is a single cycle.

A9. A worst-case scenario occurs when each of the NoC packet is blocked at each router by all NoC flows that can be encountered.

A10. Since we consider worst-scenarios and in order to ensure an upper bound of the WCTT of flows, we assume that the input buffer of each router has a size of a single flit. Actually, when the size of the buffer is reduced, the flits of each packet are spread on many routers, which could increase the contentions on the network.

I/OModel

In this work, we assume the NoC to be connected to two memory controllers. One is located to the north and the other one to the south of the NoC, as shown by Figure 4.3. As in Tilera, we assume that each controller can be accessed from the edges of columns 2 to $L-1$ of the NoC, through the ports 1 to n. We therefore have $L \geq 3$. On the Ethernet side, we consider that a set of at least two Ethernet controllers are connected to the east of the NoC, as in Tilera. Similar to the localization of memory ports, we assume that these Ethernet controllers can be accessed from the east of lines 2 to $W-1$ of the NoC. We therefore have $W \geq 4$ and the maximum number of Ethernet controller equals to $W-2$. Ethernet interfaces can therefore be shared between several applications to be mapped on the NoC. Note that if $W-2$ is higher than the actual number of Ethernet controllers, the Ethernet controllers are spaced by some lines of cores. Each Ethernet controller can be accessed from the core adjacent to it through one port.

Hence, the assumptions to model the I/Ointerfaces and the transmission of the I/Oflows are the followings:

A11. 3 Ethernet controllers are located to the east, respectively at the second, fourth and sixth rows of the 7×7 grid, i.e. $y = 2, 4$ and 6. The DDR memory located to the north and to the south are accessed directly from the second till the sixth column, i.e. $x = 2$ till $x = 6$.

A12. Single giga-Ethernet interfaces are considered, where the buffer in each interface has a size of 2 KB.

A13. The transfer of data between I/Ointerfaces and tiles are done following the first possibility mentioned in section 4.1.1, i.e. first, from the Ethernet interface to the DDR memory and then from the DDR to the destination core. Thus, the WCTT of a core-to-I/Oflow is the WCTT of this flow to reach the DDR memory added to the WCTT of the flow from the DDR to the core destination. In the remainder of this work, we focus on the WCTT of the core-to-I/Oflow to reach the DDR memory, since this WCTT impacts directly the buffer state on the Ethernet interface.

A14. The Ethernet controller sends the data to the nearest memory and to the first DDR port that might not be used by another Ethernet controller. However, the DDR sends the data to the destination core from the port belonging to the same column of this core. For example, in Figure 4.3, if the south Ethernet controller located at (0,2) sends data to the core $(L, 1)$, it first sends data to port 1 of the south memory and then the data are transmitted by port n to the core (steps 3a.i and 3a.ii of figure 4.2). However, if the Ethernet controller located at (0,4) want to send this data, it sends it first to port 2 of the south memory.

A15. A payload stored in the buffer of the Ethernet interface is removed only when all its corresponding NoC packets are received by a port of the DDR memory.

A16. The Ethernet frames have a Maximum Transmit Unit (MTU) of 1500 bytes.

4.2 An avionic case study

As said previously, a NoC integrates a mix of real-time applications with different levels of criticalities, which communicate to a number of sensors and actuators via Ethernet interfaces. In the remainder of this work, we consider only two levels of criticalities: critical and non-critical. Flows can have various requirements depending on the applications. For instance, critical applications have critical latency requirements and often a low payload, as it consists of sensors and actuators command. Larger payloads can be exchanged in less critical applications which are received by a larger number of sensors. However, the latency requirement of these flows is less strict than those of critical applications.

Critical applications

The considered critical applications in this work are based on: (1) Full Authority Digital Engine (FADEC) application, (2) Research Open-Source Avionics and Control engineering (ROSACE) application.

FADEC is an application that controls all aspects of aircraft engine performance. It receives low payloads of sensors data from an engine. Thus, we consider that it receives 1500 bytes

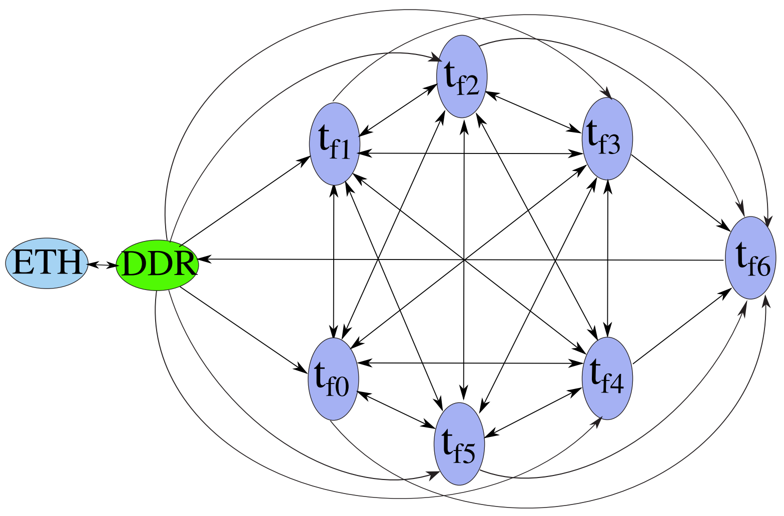


Figure 4.4: Task graph of core-to-core and core-to-I/O communications of the FADEC application assuming 7 tasks.

of data from the Ethernet interface. These data are then divided and distributed to n tasks, noted t_{f0} to t_{fn}. These tasks, except t_{f6}, exchange 250 bytes of data between them. All these tasks also send 250 bytes of data to the task noted t_{f6}. Then, t_{f6} stores 130 bytes within a DDR interface and sends back 76 bytes of actuators data through the same Ethernet interface. Figure 4.4 shows a graph of the core-to-core and core-to-I/O communications between the tasks of the FADEC application, assuming 7 tasks. As the control of aircraft engine is complex, the number of tasks composing this application could be increased. Thus, different instances for this application are considered and noted by $FADEC_n$ where n corresponds to the number of tasks composing the application.

ROSACE is the second critical application taken from the case study introduced in [PSG+14]. It manages the longitudinal motion of a medium-range civil aircraft in *en-route* phase. ROSACE is composed of 10 tasks. Figure 4.5 presents the longitudinal flight controller architecture, which is divided into two parts: the environment simulation and the controller. The environment simulation presents the aircraft as well as the engines and elevators that are to be controlled. The controller is composed of 5 filters and 3 controllers. We modify this application in such a way that it communicates with the I/Ointerfaces: Figure 4.6 shows the task graph that we thus assume. In the original application, data exchanged between the tasks are of low payload (about tens of Bytes). In this work, as we consider data coming from an Ethernet interface

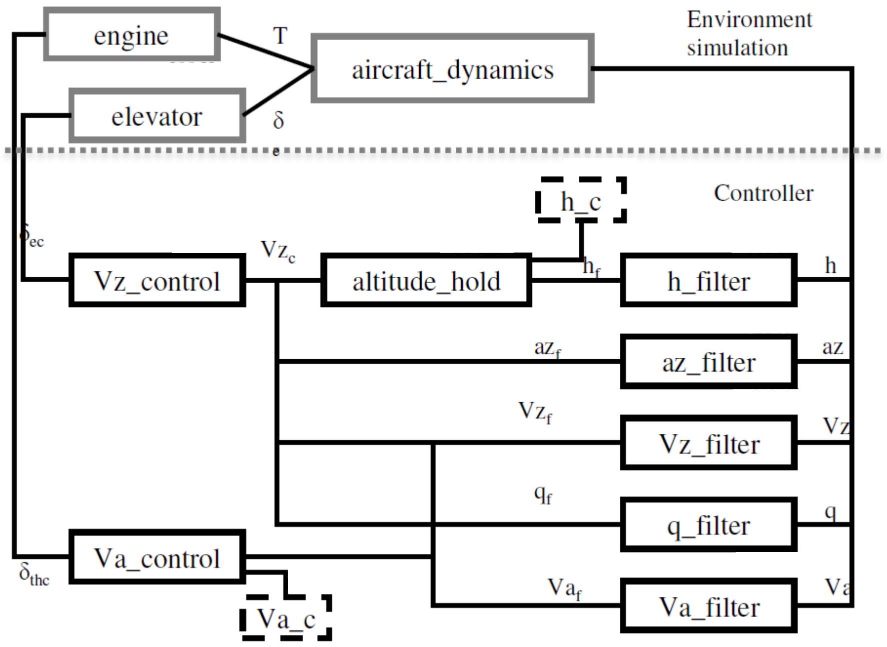


Figure 4.5: ROSACE case study, extracted from [PSG+14].

where the buffer could store larger payloads (2 KB), we thus increase the size of data exchanged in such a way to be aligned with FADEC application. Hence, we consider that a 600 bytes of payload are transmitted via Ethernet frame. These data are divided and distributed to 5 tasks: hf, azf, vzf, qf and vaf. These tasks then send these data of 120 bytes to vzc or to vac or to both of them. Finally, vzc and vac send these data to respectively eng and $elev$, which then transfer them to the DDR in order to be sent back to the Ethernet interface. This critical application is considered because it provides different requirements from FADEC. It indeed presents less-constrained communications.

Non-critical applications

The considered non-critical applications are based on: (1) a Health Monitoring (HM) application, (2) a Fast-Fourier Transform (FFT) application.

HM are signal processing applications used to recognize incipient failure conditions of engines. An HM application continuously receives from a large number of sensors through an Ethernet interface, a set of frames representing data to be processed in order to anticipate engine failures. The size of a frame is 130 KBytes and a set is made of 30 frames. When a set of frames is received, every two frames are assigned to a different task amongst n tasks, noted t_{h0}

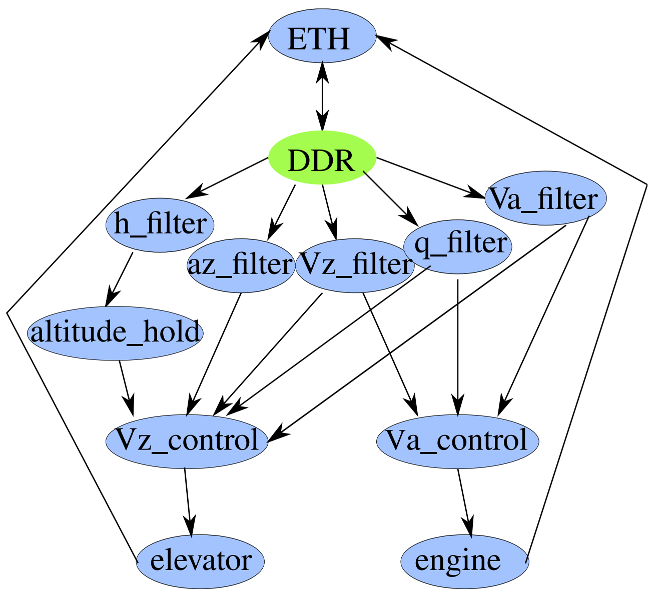


Figure 4.6: Task graph illustrating ROSACE case study.

to t_{hn}. When the processing takes place, task t_{hi} also sends 112 bytes of data to t_{hi+1}, with $i \in [0, n]$. Finally, all these tasks finish their processing by storing their frames into the memory. Figure 4.7 shows the task graph of the application HM_6 which is composed of 6 tasks. It is

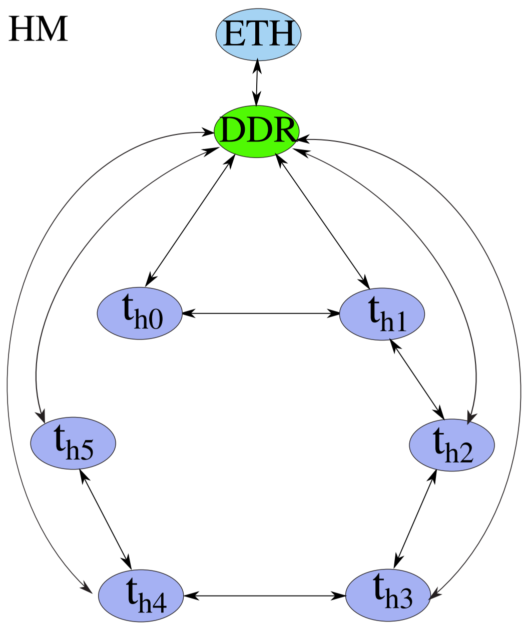


Figure 4.7: Task graph illustrating HM application assuming 5 tasks.

characterized by its high number of communications with the DDR where its large payloads are stored. HM is a signal processing application that can be easily parallelized on an arbitrary number of cores. For this reason, we consider different instances of the HM application.

FFT application is of widespread use in signal processing and embedded control [DPPB+12]. This application consists of different stages in the computation of FFT, as shown by Figure 4.8. Each core exchanges data with another one, and performs some local computation. In stage 5,

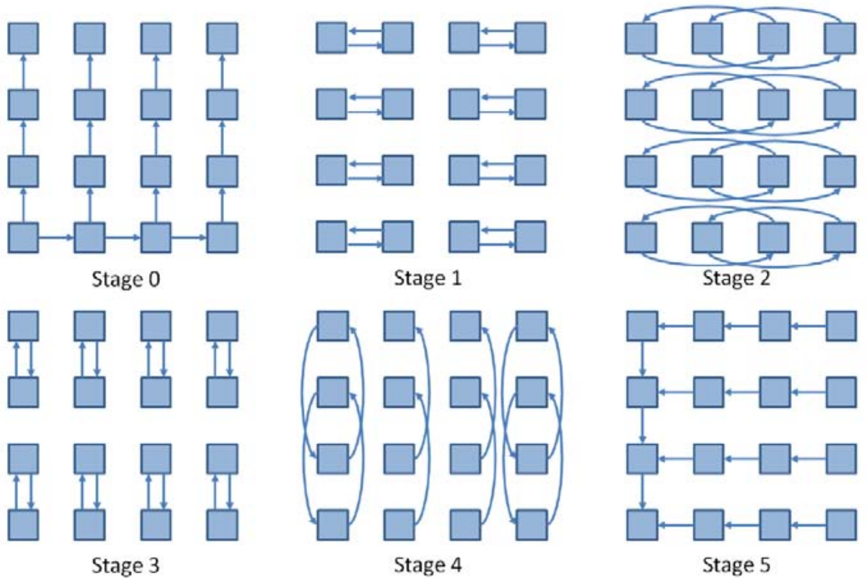


Figure 4.8: The 6 FFT communication stages.

the result computed by each core is gathered on one tile. In this work, we focus only on this stage as it presents a large number of communications as illustrated in Figure 4.9. Different size of data could be exchanged by tasks of this application. Thus, in this work, we consider arbitrary size of data where we suppose FFT receives 750 bytes of data from the Ethernet interface in order to be aligned with the applications introduced previously. This data is distributed on 15 tasks, noted t_{ff0} to t_{ff14}. Each of this task performs some computing operations, and sends the results to the task t_{ff15}. We vary the size of data sent from these tasks from 8 bytes to 76 bytes, which correspond to the minimal and maximal size of the NoC packets (refer to $\boldsymbol{A4}$).

Considered case study A

A mix of these realistic applications is considered to illustrate the problem when the NoC is connected to sensors and actuators via Ethernet interfaces. The considered case study, noted A, is made of the following applications: $FADEC_9$, FFT_{16}, and two instances of HM noted by HM_{11} and HM_{12}. Figure 4.10 shows an arbitrary mapping of this case study. The square 3×3, whose left corner is located at (7,1) defines the regions where $FADEC_9$ is mapped. The

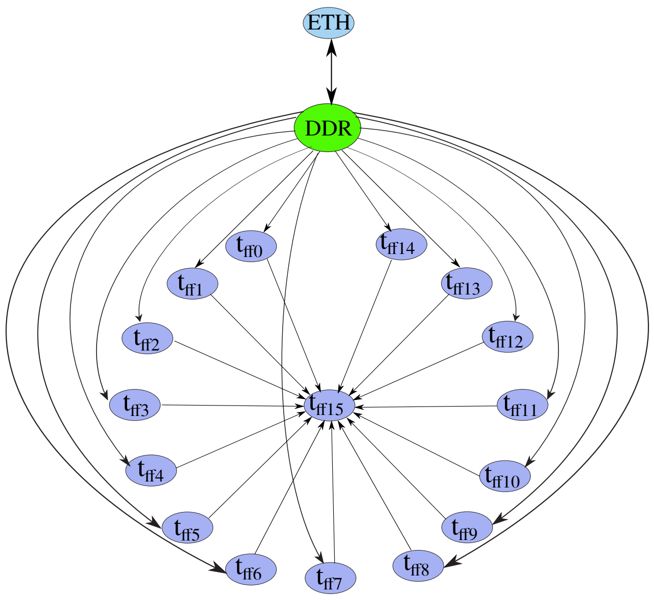


Figure 4.9: Task graph illustaring FFT application assuming 16 tasks.

rectangle 4 × 3, whose right corner located at (1,1), defines the region where HM_{12} is mapped. HM_{12} and $FADEC_9$ therefore use the same Ethernet interface, located at $(0, 2)$ as shown on the Figure 4.10. The rectangle 3 × 4, defined by its upper left corner located at (7,7) and the square 4 × 4, whose upper right corner is located at (1,7), define respectively the regions where HM_{11} and FFT_{16} are mapped. Thus, they use the same Ethernet interface located at (0,6). In this case study, we consider that the Ethernet interface at (0,4) is not used.

4.3 Problem illustration

Different types of communications are exchanged on the NoC: core-to-core and core-to-IO. However, the core-to-I/Oflows experience a change in their speeds as they traverse two networks of different types: Ethernet and NoC. An Ethernet frame coming to the NoC will first be buffered in a buffer of limited capacity (***A12***). It is then divided into a number of NoC packets in order to be transmitted on the NoC, and this is due to the difference of the maximum size of packets allowed on each network (***A4, A13 and A16***). However, if a next frame comes to the same interface and given the assumption ***A15***, the question will then be: ***would the buffer overflow and thus lead to drop the frame?*** To answer the question, the WCTT

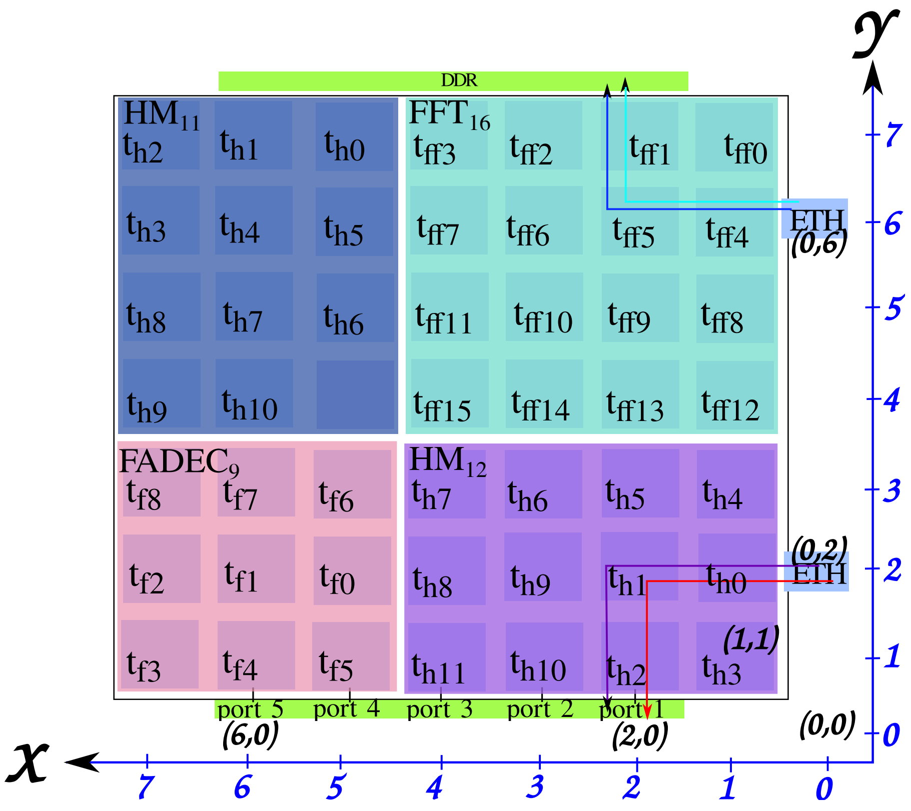


Figure 4.10: Arbitrary mapping of a case study made of one FADEC, one FFT and two HM applications.

of the core-to-I/Oflows on the NoC must be analyzed. Let us first illustrate this problem on the considered case study A.

Problem identification

Let us first focus on the steps and the timing of the core-to-I/Oflow coming from the Ethernet interface (0,6) for HM_{11}. We suppose that an Ethernet frame of HM_{11} is transmitted before a frame of FFT_{16}. When an Ethernet frame for HM_{11} arrives at the Ethernet interface, it is stored into the Ethernet buffer in 12.336 μs (transmission of 1500 bytes of payload at 1Gb/s). Since the size of the payload of the Ethernet frame for HM_{11} is 1500 bytes and the Ethernet buffer size is 2 KB (***A12***), the Ethernet buffer can then store an additional Ethernet frame of only 500 bytes. The size of a FFT Ethernet frame is however 750 bytes. The HM frame must therefore have been transmitted, through the NoC, to the memory before the FFT frame can be stored (***A13***). This means that the WCTT of the HM core-to-I/Oflow to reach the memory must be less than the arrival delay of FFT frame to the Ethernet interface. We recall that the HM core-to-I/Oflow is composed from a number of NoC packets corresponding to the HM frame. Due to the maximum packet size over the NoC (***A13***), the HM frame is divided into 20 packets of 19 flits followed by one packet of 15 flits. Each one of these packets is blocked by the FFT_{16} flows that can be encountered (***A9***). For instance, we consider that core-to-core communications in the FFT application have a size of 8 bytes which is equivalent to 2 flits. The WCTT of an individual HM packet on the NoC is computed by considering the recursive calculus (RC) method presented in section 3.1.2 of chapter 3. Thus, by using the RC method, an HM packet made of 19 flits takes $t_p = 400.775\ ns$ to reach the memory. The analyzed flow noted f_a, i.e. a packet of an HM core-to-I/O flow, can indeed be blocked by direct and indirect flows. Figure 4.11a shows the possible blocking flows of the analyzed flow f_a. This figure presents one direct blocking flow: f_1, and seven indirect flows: f_2 till f_8. At each router, RC thus identifies the direct flows and add their delays. Therefore, the value t_p returned by RC takes into account indirect flows. For instance, f_1 is the direct flow blocking f_a. The flows that block f_1 on its path are identified: f_2, f_3, f_4, f_5, f_6, f_7 and f_8. The delays of these flows are also computed and added to the delay of f_1. Recursively, for each of these flows, blocking flows are identified and their delays are also added. The blocking delay of f_1 is therefore: $delay(f_1) = d(f_2) + 2d(f_3) + 4d(f_4) + 4d(f_5) + 12d(f_6) + 12d(f_7) + 36d(f_8)$, where $d(f_i)$ is the transmission delay of a flow to reach its destination. Therefore, $t_p = d(f_1) + d(f_a) + delay(f_1)$. Thus, if all the packets of HM core-to-I/Oflow experience their WCTT on the NoC, the global WCTT for the HM core-to-I/Oflow is $t_1 = 8.407\ \mu s$. However, the transmission of the FFT frame on Ethernet takes $t_2 = 6\ \mu s$. As we have $t_1 > t_2$, then this WCTT leads to drop the FFT frame. Let us see why we have this negative result.

4.3.1 Is improving the computation of the WCTT sufficient?

The second part of Figure 4.11a shows the flits position of the different flows at the moment where the flow f_3 is no more blocked by the flows f_4 and f_5 at the router (4,6). We note that f_3 blocks directly f_1 and indirectly f_a. Thus starting from this state, figure 4.11b shows the timeline of the pipeline transmission of the different flits of these flows. Each line represents a router, so once a flit is transmitted to the next router, a credit is also transmitted to the previous router. These transmissions are presented on the timeline by oblique lines. At each router, we show the time of passage of each flit and its delay, which is reprenseted by a small rectangle. A flow is blocked at a router by different flows that share the same output by respecting the RR arbitration (***A6***). A blocked flow is presented on the timeline by a line from a router to the next one, without being followed by its flit delay, i.e. the small rectangle. At t_1, the second flit of the direct blocking flow f_1 leaves the router located at (2,6) to be blocked at the router (3,6). At this moment, its input buffer is thus free and so f_a can progress since it is no longer blocked. f_a reaches its destination at $t_2 = 277ns$ without being affected by the transmission of the indirect blocking flows, while f_1 progresses slowly waiting for its blocking flows to progress. Thus, f_1 reaches its destination at $t_3 = 314ns$. However, in the recursive methods, f_a could not progress until f_1 reaches its destination located at the router (4,4).

Thus, recursive methods do not consider the pipeline transmission leading to an over-approximation of WCTT. The delay of f_a is made dependent on the remaining distance between its destination and the destination of the flows in contention. Also, these methods always consider all the indirect flows as influent.

Let us now compute the WCTT of the HM_{12} core-to-I/O flow where its frame is transmitted before $FADEC_9$ frame. We recall that these 2 applications share the second Ethernet interface located at (0,2). The Ethernet frame for HM_{12} arrives at the Ethernet interface and is stored into the Ethernet buffer in 12.336 μs. Each packet corresponding to the HM payload is blocked by the HM flows. The WCTT of an individual HM packet on the NoC takes 807.3 ns by using the RC method. The global WCTT of the HM_{12} core-to-I/Oflow is therefore 16.944 μs. However, the transmission of the FADEC frame on Ethernet takes also 12.336 μs as its payload size is 1500 bytes. As FADEC frame reaches the Ethernet interface before the removal of the HM frame and its size is greater than the free size in the Ethernet buffer, then the FADEC frame is dropped.

However, by modeling the real transmission of the pipeline behavior, similar to what is done for the FFT application, the WCTT for the HM frame (16.5 μs) still leads to drop the FADEC frame. Therefore, we need another strategy to reduce this WCTT.

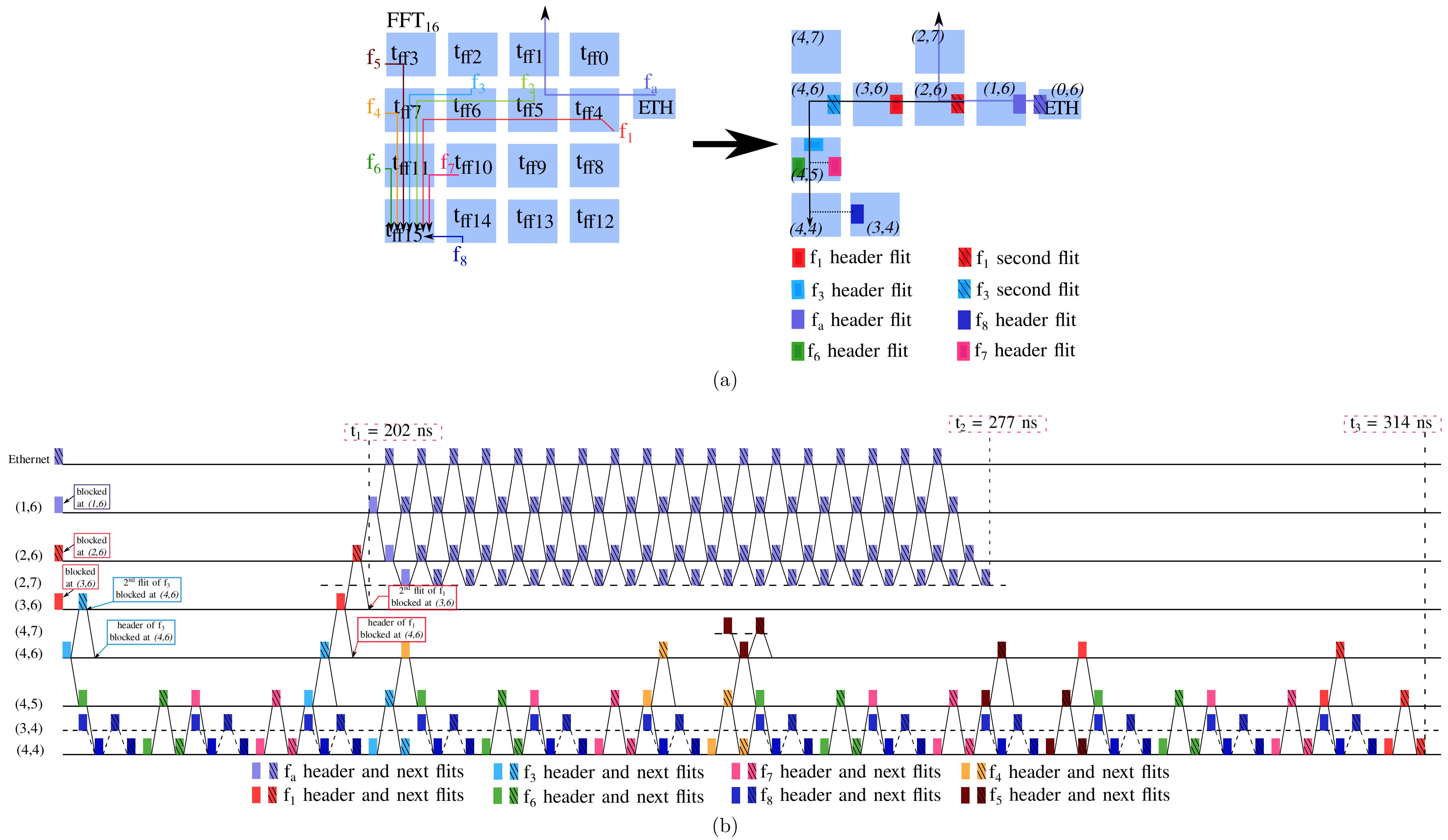

FFT$_{16}$
t$_{ff}$0 t$_{ff}$1 t$_{ff}$2 t$_{ff}$3 t$_{ff}$4 t$_{ff}$5 t$_{ff}$6 t$_{ff}$7 t$_{ff}$8 t$_{ff}$9 t$_{ff}$10 t$_{ff}$11 t$_{ff}$12 t$_{ff}$13 t$_{ff}$14 t$_{ff}$15
ETH
f$_a$ f$_1$ f$_3$ f$_3$ f$_4$ f$_5$ f$_6$ f$_7$ f$_8$
(4,7) (2,7) (4,6) (3,6) (2,6) (1,6) (0,6) (4,5) (4,4) (3,4)
f$_1$ header flit
f$_1$ second flit
f$_3$ header flit
f$_3$ second flit
f$_a$ header flit
f$_8$ header flit
f$_6$ header flit
f$_7$ header flit
(a)
t$_1$ = 202 ns
t$_2$ = 277 ns
t$_3$ = 314 ns
Ethernet
(1,6) (2,6) (2,7) (3,6) (4,7) (4,6) (4,5) (3,4) (4,4)
blocked at (1,6)
blocked at (2,6)
blocked at (3,6)
2nd flit of f$_3$ blocked at (4,6)
header of f$_3$ blocked at (4,6)
2nd flit of f$_1$ blocked at (3,6)
header of f$_1$ blocked at (4,6)
f$_a$ header and next flits
f$_1$ header and next flits
f$_3$ header and next flits
f$_6$ header and next flits
f$_7$ header and next flits
f$_8$ header and next flits
f$_4$ header and next flits
f$_5$ header and next flits
(b)

Figure 4.11: (a) Core-to-core blocking flows of HM core-to-I/Oflow and their flits position, (b) Transmission timeline of core-to-core and core-to-I/Ocommunications.

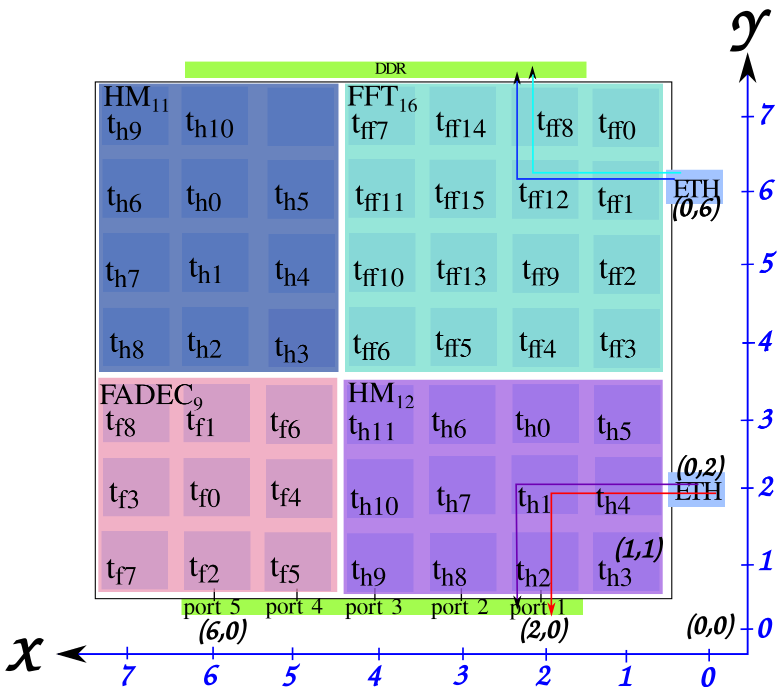


Figure 4.12: SHiC mapping of the case study A made of one FADEC, one FFT and two HM applications.

4.3.2 Is a contention-aware mapping strategy the solution?

We know that changing the mapping of tasks of applications, modifies the values of the WCTT of the different flows. Thus, let us see what happens when we modify the mapping of the FFT and HM applications by considering the SHiC method, a congestion-aware mapping presented in section 3.2.2 of chapter 3. Figure 4.12 shows the mapping of our considered case study by applying the SHiC method. We note that in this mapping the regions, where the applications are allocated, are not modified compared to the arbitrary mapping proposed before. Only the mapping of the tasks in each region changes. Actually, SHiC first chooses the core located at (6,2) having a Square Factor (SF) equal to 16, as this core is centered into a square of area 9 and having 7 cores available at its borders. On this core, SHiC allocates the task of $FADEC_9$ that has the maximum number of communications, i.e. t_{f0}. The other tasks communicating with t_{f0} are allocated to the neighboring cores by forming the smallest square including t_{f0}. This same process is applied for the next applications where each of these cores on (2,2), (3,6) and (6,5) presents the first core to start the allocation for respectively HM_{12}, FFT_{16}, HM_{11} with SF = 16, 20 and 12.

We now compute the WCTT of the HM_{11} core-to-I/O flow coming from the Ethernet interface at (0,6). The global WCTT of the HM core-to-I/O flow blocked by the FFT communications using recursive calculus method is $t_1 = 1.681$ μs. Thus, after that HM frame is removed from the Ethernet buffer, the next FFT frame is stored at the buffer and is not dropped as it takes $t_2 = 6$ μs to reach the Ethernet interface. However, if we consider that packets of 15 flits are exchanged between the FFT tasks, the global WCTT of the HM core-to-I/O flow increases and it takes $t_1 = 6.762\mu$s to reach the memory. t_1 is now greater than t_2, thus this WCTT leads to drop the FFT frame. The problem of dropping the FFT frame is then solved partially.

Besides, this problem remains when analyzing the WCTT of core-to-I/O flow for the HM_{12} whose frame is transmitted before $FADEC_9$. The values are the same of those obtained by considering the arbitrary mapping.

Need of a congestion-aware mapping considering the core-to-I/Oflows

The only goal of SHiC and related works mapping strategies is to reduce the congestion on the core-to-core flows. Thus, they do not consider the locations of I/O interfaces within the NoC during applications mapping. But, we saw that these interfaces could be shared between several applications. In this case study, the critical application FADEC is allocated far from Ethernet and DDR controllers. So, a core-to-I/Oflow experiences a delay due to the non-critical application HM, which presents a high number of communications of large payloads, especially with the DDR memory. Besides, the internal mapping of the applications also influences the WCTT of this core-to-I/O flow. In fact, SHiC, similar to most existing strategies, allocates the task with the highest number of communications, approximate at the center of the square where the application is mapped. Thus, for the FFT application, as illustrated in Figure 4.12, it allocates the task t_{ff15} at the core located at (3,6) having $SF = 16$. As this task receives data from all other tasks, the core-to-I/Oflow is then blocked directly by the flow generated by t_{ff1} having t_{ff15} as destination. However, this flow is also blocked by other flows, such as the flows coming from t_{ff11}, t_{ff12}, t_{ff13} and t_{ff14}, which induce indirect contentions with the core-to-I/Oflow. Figure 4.13 illustrates these flows blocking directly and indirectly the core-to-I/Oflow. We note that the flows generated by t_{ff11}, t_{ff13} and t_{ff14} block directly the one

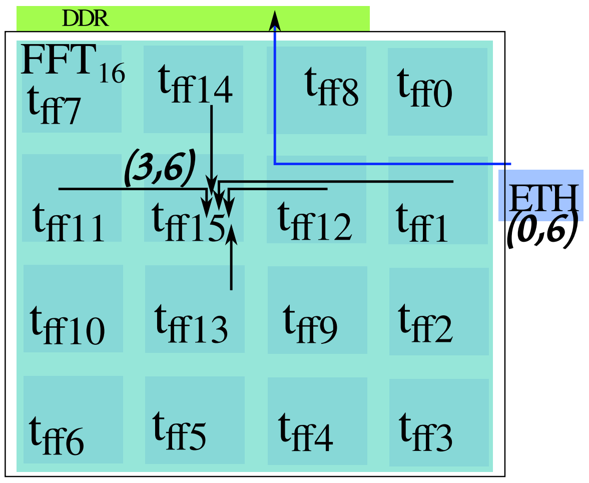


Figure 4.13: FFT flows blocking the HM core-to-I/Oflow by considering SHiC mapping.

generated by t_{ff1} at the router (3,6) as they share the same destination. The flow coming from t_{ff12} blocks the one generated by t_{ff1} as they share the same link. Thus, these direct and indirect contentions with the core-to-I/Oflow lead to an increase in its WCTT. Thus, this internal mapping strategy is not appropriate to reduce the contention on the core-to-I/Oflows as it does not consider these flows when mapping the tasks. In order to reduce the WCTT of a core-to-I/Oflow and avoid the dropping of incoming I/O packet, the number of contentions a core-to-I/O flow experiences should instead be reduced.

Thus, we need to define a mapping application strategy that considers core-to-I/O flows on a Tilera-like NoC as first-class citizen, so that the contentions they exhibit and thus their WCTT are reduced to avoid the aforementioned problem.

4.4 Proposal

NoC could be used as processing elements within a backbone Ethernet network as they provide different I/Ointerfaces. Thus, different types of communications can exist in the NoC: core-to-core and core-to-IO. The core-to-I/Oflows experience a change in their speed when crossing two types of network: Ethernet and NoC. However, when a NoC congestion occurs on paths taken by core-to-I/O flows, their estimated WCTTs can be higher than the arrival delay of the next incoming Ethernet frame. In this case, this next Ethernet frame may be dropped due to the lack of space in the Ethernet buffer, which is of limited capacity. Previous Ethernet frames could indeed be stored in this buffer while waiting associated NoC packets can progress towards the DDR interface. For this reason, the WCTT of core-to-I/Oflows should be analyzed on the NoC. In this chapter, we illustrated this problem using a case study made from critical and non-critical applications of different requirements. We then showed the need to reduce the pessimism that exists in the current state-of-the-art method, i.e. Recursive Calculus, to compute the WCTT of this core-to-I/Oflow. In fact, this analysis method do not take advantage of the pipeline transmission of wormhole-switching. However, adding such capabilities in RC, is not sufficient to avoid dropping Ethernet frames. For this end, we need to change the mapping to reduce the WCTT of the core-to-I/O flows. Nevertheless, existing mapping strategies consider only the objective to reduce the contention of core-to-core communications. We illustrated that such a mapping, i.e. SHiC mapping, does not consider the contention on the path of core-to-I/O flows which lead to drop critical and non-critical Ethernet frames. Thus, a static mapping strategy of critical and non critical real-time flows that reduces the WCTT of core-to-I/O communications over a Tilera-like NoC is needed. The next chapters present our approaches to avoid the aforementioned problem: dropping Ethernet frame. Thus, in Chapter 5, we describe our analytical method to reduce the pessimism when compting the WCTT of flows. In chapter 6, we present our mapping strategy that takes into account both core-to-I/O flows and core-to-core communications in order to reduce the WCTT of the core-to-I/Oflows. The evaluation of these approaches is illustrated in Chapter 7.

Chapter 5

RC_{NoC}: an optimized WCTT analysis for NoC

Contents

Chapter 4 has shown that the recursive methods do not model the pipeline behavior of the wormhole switching. This is a source of pessimism when computing the WCTT of flows. To reduce this pessimism, the analysis should be done at the granularity of a flit. To compute the WCTT of an analyzed flow f_a, we have to explore all the possibilities of transmission of different flows. However, the number of possibilities can be too high, and then they cannot be explored in a reasonable amount of time. In this chapter, we propose to define an improved recursive method, called RC_{NoC}, to analyze the behavior of the network. This method is based on three properties which allow to reduce the number of scenarios and the pessimism of the classical recursive methods. The proposed method is then evaluated on a synthetic benchmark. First, we propose some notations which will be used to describe our method.

5.1 Notations and definitions

The following notations and definitions are used in this chapter in order to explain the properties on which are based our proposed method.

- ***Notation 1:*** d_s is the switching delay of a flit, i.e. the time required for the router to grant an output port for the packet. In Tilera Tile64 NoC, the switching delay of the header and that of the remaining flits are identical.
- ***Notation 2:*** f_{size} is the size of a flit.
- ***Notation 3:*** d_t is the traversal delay of a link of capacity C. Thus,

  $$d_t = \frac{f_{size}}{C}$$

  This delay corresponds to the traversal delay for regular data or credits due to the flow control policy.
- ***Notation 4:*** d_{flit} is the delay taken by a flit to be transmitted from a router r_k to r_{k+1}. Then,

  $$d_{flit} = d_s + d_t$$

  In this work, we assume that $d_s = d_t$ and $d_{flit} = 1$ cycle as in the Tilera NoC.

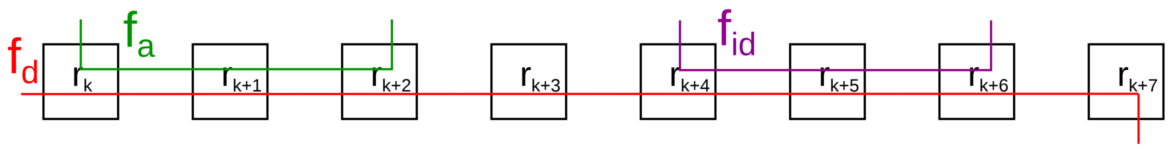


Figure 5.1: Example illustrating the number of routers separating an analyzed flow f_a from its indirect flow f_{id}.

- ***Notation 5:*** n_{f_i} is the size of a packet of a flow f_i in numbers of flits.
- ***Notation 6:*** a flow f_i is described by a set of routers defining its path from the source router, noted $r_{isource}$, to its destination router, noted $r_{idestination}$. The path of this flow is noted by $path(f_i)$. Then,

$$path(f_i) = r_{isource}, r_j, r_k, ..., r_{idestination}$$

The example in Figure 5.1 is used to illustrate the following notations from 7 till 9.

- ***Notation 7:*** f_d is a direct flow that blocks f_a. The last common router between f_a and f_d is r_d. In the example of Figure 5.1, r_d corresponds to r_{k+2}.
- ***Notation 8:*** f_{id} is an indirect flow of f_a and which blocks f_d at r_{id}. In the example, r_{id} corresponds to r_{k+4}.
- ***Notation 9:*** E_r is the number of routers separating f_a from f_{id}, i.e. between r_d and r_{id}. Thus, in the example, $E_r = 1$, which corresponds to the router r_{k+3}.
- ***Definition 1:*** F is the set of flows in an application.
- ***Definition 2:*** $F_p \subset F$ is the set of flows to be considered when analyzing f_a, should they be in direct or indirect contention with f_a.
- ***Definition 3:*** $F_{i,k}$ is the set of flows blocking f_i on a router r_k of its path. Due to the wormhole behavior, F_p includes not only the flows blocking directly f_a in its path, but also the indirect flows that blocks the direct blocking flows. This means that $\cup_{\forall r_k \in path(f_a)} F_{a,k} \subseteq F_p$.

5.2 Wormhole network properties

In the following sections, we describe three properties that permit to compute the WCTT and reduce the pessimism in this computation. The first one reduces the number of possible scenarios by studying the arrivals of the flows on each router. The two following ones concern the pipeline behavior of the transmission and its utilization in the WCTT computation.

5.2.1 Local worst-case scenario

Computing a WCTT requires to identify the worst-case scenario. To identify such a worst-case scenario, the possible sequences of flows from F_p over routers must be explored, even though all these scenarios do not lead to the WCTT. To reduce the number of scenarios to explore when performing a WCTT analysis, we thus define a first property that identifies the worst-case scenario. The local worst-case scenario is identified when computing the maximal blocking delay of f_a at each router.

Property 1: Identifying the worst-case scenario.
For a flow f_a, the worst-case scenario, on each router $r_k \in path(f_a)$, occurs when:

1. *The port on which f_a arrives is the last port served by the round-robin arbitration (RRA).*
2. *The headers of f_a and a flow $f_b \in F_{a,k}$ arrive synchronously on r_k.*
3. *The other flows $f_j \in F_{a,k}$ $(f_j \neq f_b)$ arrive on r_k either synchronously with f_a and f_b or no later than the release of the output port by the served flow (i.e. f_b or another f_j), consequently before the next round of arbitration.*

The first condition is obvious. Actually, when f_a is the last one served, then it waits for the progression of all flows contending to the same output port. Now, let us prove the condition 2.

Proof. Let us assume that f_a is blocked at router r_k by two flows f_i and f_j. These flows have therefore in common a next router r_{k+1} in their paths towards their destinations. At least one

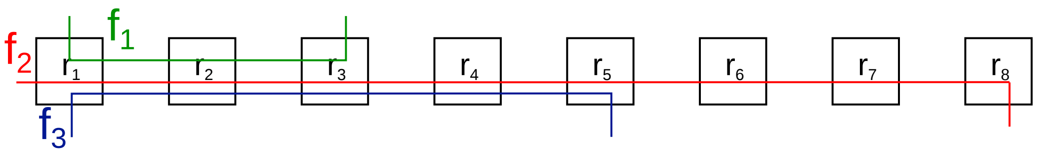


Figure 5.2: Example illustrating the second condition of Property 1.

link must be shared between two flows so that they are in (direct) contention. When the headers of f_a and f_i arrive synchronously on r_k, f_a must wait till all the flits of f_i leave r_{k+1} before being able to progress, as we consider buffers of capacity of a single flit (**A11**). This leads to a blocking delay D_1 equals to $n_{f_i} \times 2d_{flit}$, since the two routers r_k and r_{k+1} must have transmitted all the flits of f_i before being able to proceed with another flow. Now, let us assume that n_{cf_i} flits of f_i have already been transmitted by r_k before the header of f_a arrives at r_k. Then, the blocking delay of f_a before being able to progress is $D_2 = (n_{f_i} - n_{cf_i}) \times 2d_{flit}$. Therefore, the arrival of the header of f_a with any flit of f_i, except the header, leads to a lower delay of f_a, where $D_2 < D_1$. Thus, the synchronous arrival of headers is a worst-case scenario. □

To illustrate this behavior, let us consider the example in Figure 5.2. This example uses 3 flows. f_1 is the analyzed flow f_a, f_2 and f_3 are direct flows blocking f_1 at r_1. We assume that all packets are made of 4 flits. We focus, first, on the transmission of f_1 and f_2.

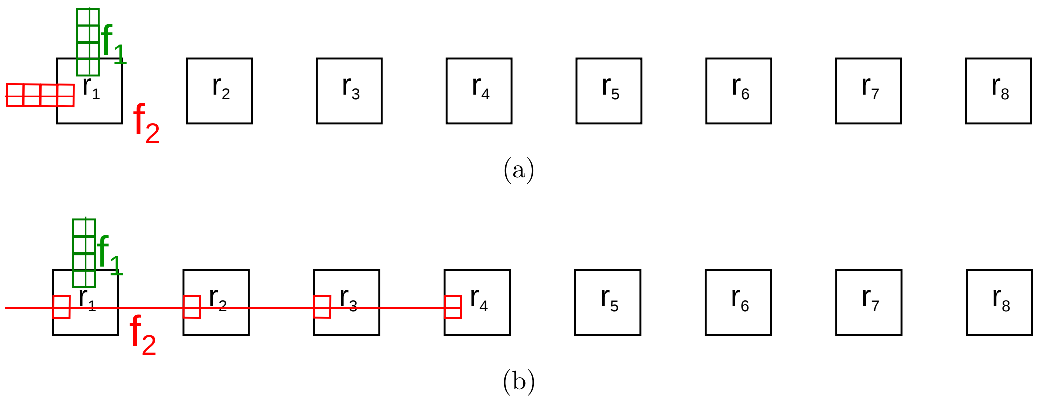


Figure 5.3: Different scenarios to illustrate the second condition of property 1.

The first possible scenario is proposed in Figure 5.3a. f_1 and f_2 arrive synchronously at r_1. As f_1 and f_2 share the same link between r_1 and r_2, f_1 waits for the transmission of all flits of f_2 by the router r_2. This leads to a blocking delay $D_1 = 4 \times 2d_{flit}$ as illustrated in the scenario (a) on the timeline of Figure 5.4. Thus, $delay(f_1) = d(f_1) + D_1 = 17 cycles$, where $d(f_1)$ corresponds to the transmission delay of f_1 to reach its destination. In the second scenario shown in Figure 5.3b, f_1 arrives synchronously with the fourth and last flit of f_2. The scenario (b) on the timeline of Figure 5.4 illustrates the waiting delay of f_1. f_1 only waits for the progression of the last flit of f_2 from the router r_2 to r_3, leading to $D_2 = 2d_{flit}$. Then, $delay(f_1) = d(f_1) + D_2 = 11cycles$.

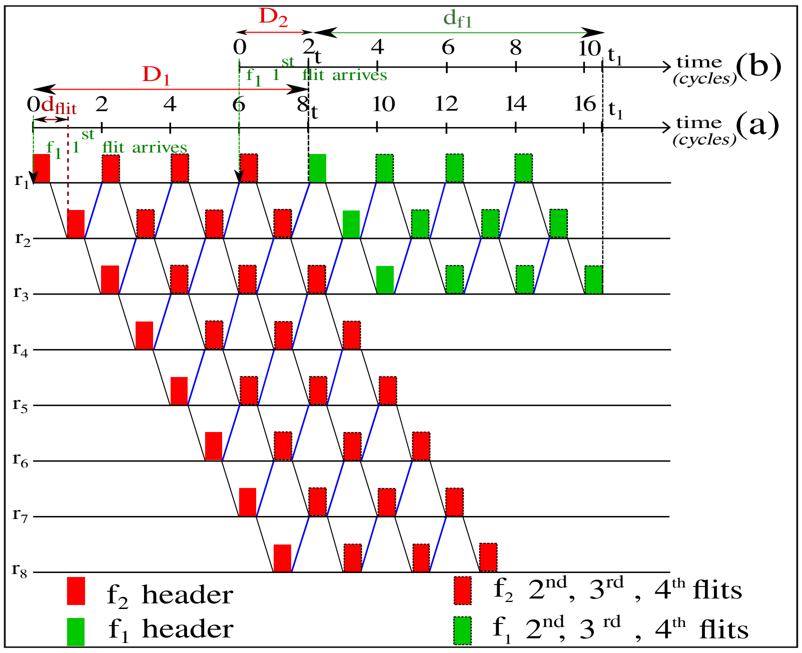


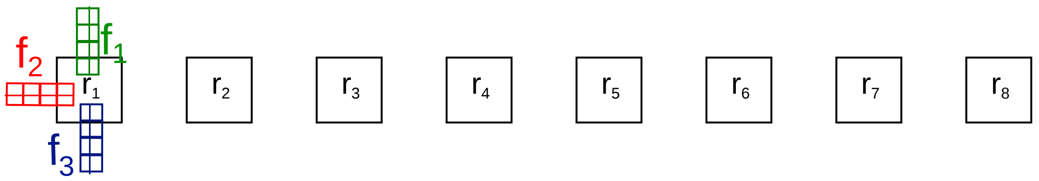
Figure 5.4: Timeline of the transmission of flows for the two different scenarios of the example shown in Figure 5.3.

Now, we have to prove the third condition of Property 1: the worst-case blocking delay of f_a is obtained when f_j arrives either synchronously with f_a and f_b or before the next round of arbitration.

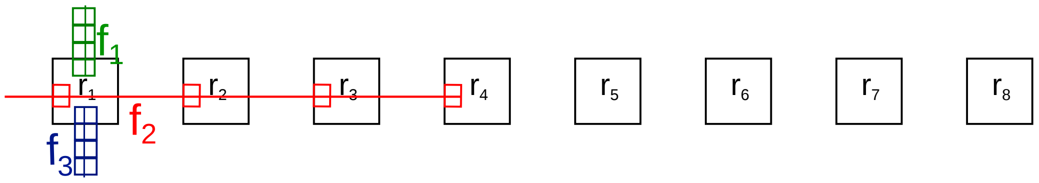
Proof. If f_j arrives after the current round of arbitration, it is clear that f_a is going to progress before f_j. In this case, the worst-case blocking delay that f_a suffers (due to f_b) is equal to D_1, previously introduced. Let us assume that f_j arrives before the next round of arbitration and when n_{cf_b} flits of f_b have been transmitted by r_k. Meanwhile, f_a was blocked for an amount of time equal to $n_{cf_b} \times 2d_{flit}$. However, the next round of arbitration only occurs when all flits of f_b are transmitted by r_k. The blocking delay that f_a suffers is thus equal to $n_{f_b} \times 2d_{flit}$, independently from the number of flits that was transmitted before f_j arrives. This blocking delay corresponds to: $n_{cf_b} \times 2d_{flit} + (n_{f_b} - n_{cf_b}) \times 2d_{flit}$, where the second term $(n_{f_b} - n_{cf_b}) \times 2d_{flit}$ is the blocking delay that f_a suffers since the arrival of f_j at r_k. The next round of arbitration is won by f_j due to the first condition of Property 1. Hence, f_a is again blocked till all flits of f_j are transmitted by r_k. In total, f_a is thus blocked for an amount of time $D_3 = (n_{f_b} + n_{f_j}) \times 2d_{flit}$. Now, let us assume that f_j arrives before the synchronous arrival of f_a and f_b. When f_a arrives at r_k, it is thus blocked for an amount of time corresponding to the transmission of the remaining flits by r_k. The same rationale leads to a total worst-case blocking delay for f_a equal to $D_4 = (n_{f_b} + n_{cf_j}) \times 2d_{flit}$. As $D_4 < D_3$, the worst-case blocking delay of f_a is obtained when f_j arrives either synchronously with f_a and f_b or before the next round of arbitration. $\square$

Note that if f_j arrives synchronously with the arrival of f_a and f_b (i.e. $n_{cf_b} = 0$), the RRA and the first condition of Property 1 ensure that f_a will be the last flow whose flits will be transmitted by r_k. r_k will either transmit f_b or f_j first, however the blocking delay that f_a will suffer is still equal to D_3.

Figure 5.5 illustrates the third condition of Property 1 when the 3 flows of our example are considered. Figure 5.5a presents a first scenario where f_3 arrives synchronously with f_2 and f_1. The second scenario, shown in Figure 5.5b, illustrates the case where f_3 arrives when r_1 has transmitted a number of flits of f_2. Here, we assume that f_1 and f_2 arrive synchronously, as previously, and that f_3 arrives with the fourth flit of f_2. At this time, r_1 has transmitted



(a)



(b)

Figure 5.5: Different scenarios illustrating the third condition of property 1

three flits of f_2. f_1 has thus been waiting for $3 \times 2d_{flit}$ and has to wait that r_1 transmits the last flit of f_2. This total additional delay is thus equal to $4 \times 2d_{flit}$. The timeline of figure 5.6

illustrates this scenario noted by (b). We can see that f_1 will only be able to progress after all flits of f_3 have been transmitted by r_1, i.e. at t_1 since f_1 has arrived on the last port served by the RRA. This corresponds to an additional delay of $4 \times 2d_{flit}$. Thus, the blocking delay of f_a is equal to $8 \times 2d_{flit}$ and $delay(f_1) = 8 \times 2d_{flit} + d(f_1) = 24 cycles$. However, this delay is the same if f_3 arrives synchronously with f_1 and f_2, as shown in Figure 5.6 in the scenario noted by (a). In this case, RRA serves f_2, f_3 and then f_1. Thus, f_1 waits until the progression of all flits of f_2 and f_3.

We claim that the global worst-case scenario is obtained when Property 1 is recursively applied to each flow that affects f_a. This is illustrated by an example in Figure 5.7, where f_3 blocks f_2 and delays f_1 and $f_a = f_1$.

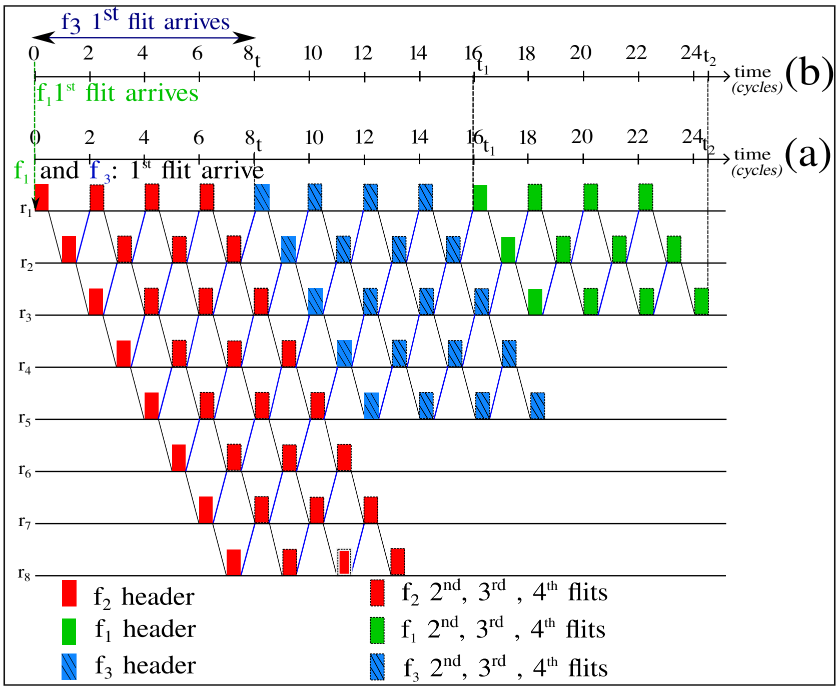


Figure 5.6: Timeline of the transmission of flows of the example shown in Figure 5.5.

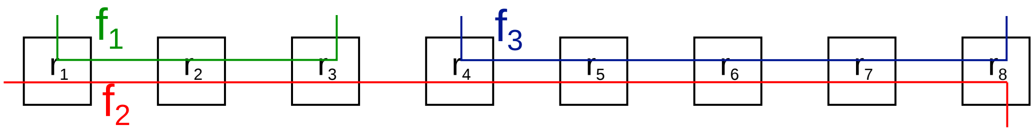
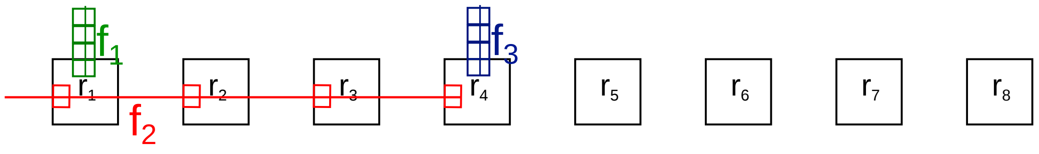
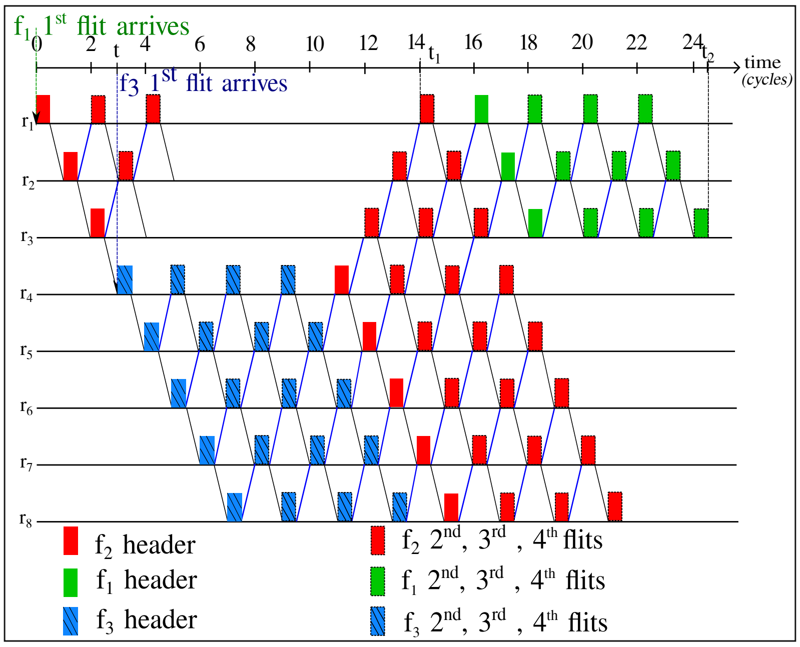


Figure 5.7: Example illustrating that Property 1 should be applied recursively.

As mentioned before, f_1 has to wait that the second common router between f_1 and $f2$, i.e. r_2, transmits all the flits of f_2. However, f_2 will be blocked by f_3 at r_4. Again, two scenarios can be defined at router r_4, as shown in Figures 5.8 and 5.9. The scenario leading to the worst-



(a)



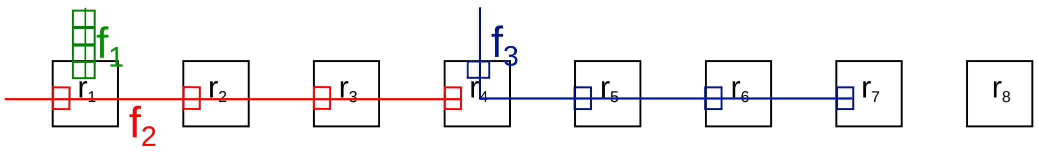
(b)

Figure 5.8: A worst-case scenario of the example in Figure 5.7 illustrating the recursion of Property 1.

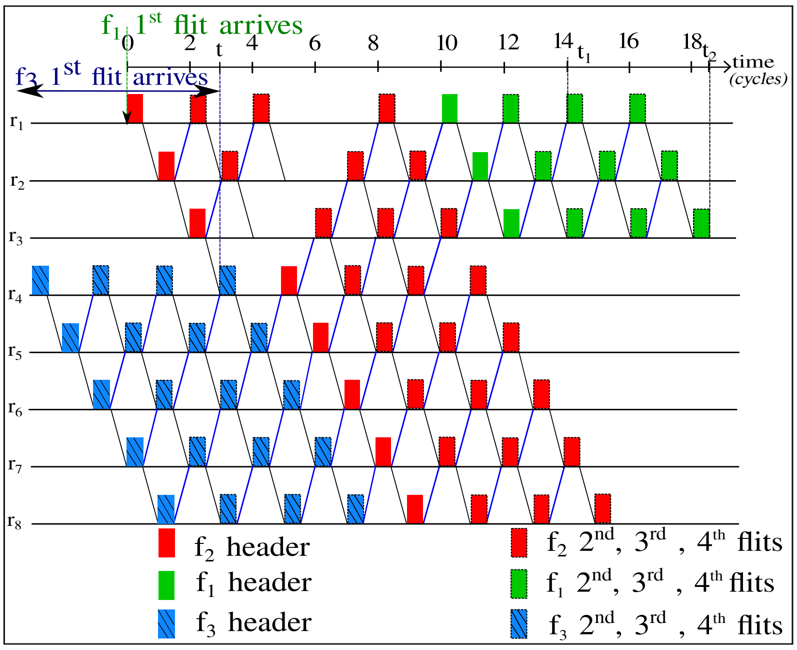
case delay corresponds to the first scenario, i.e. Figure 5.8a, illustrated in the timeline of the Figure 5.8b. In this scenario, f_2 has indeed to wait that r_4 transmit all flits of f_3 before being able to progress. f_1 is thus blocked for a delay D_5 equals to $(n_{f_3} + n_{f_2}) \times 2d_{flit}$ before being able to progress, leading to a delay of f_1 equal to 25 *cycles*. In the second scenario, plotted in Figure 5.9b, n_{cf_3} flits of f_3 have already been transmitted by r_4 before f_2 arrives at this router. f_1 thus waits for a delay equal to $(n_{f_3} - n_{cf_3} + n_{f_2}) \times 2d_{flit}$, that is lower than D_5. This scenario leads to a delay of f_1 equal to 19 *cycles*.

5.2.2 Direct and indirect contentions

To reduce the pessimism when computing WCTT, two properties are defined. The first property computes the maximal delay from which a flow can suffer. The second property eliminates some



(a)



(b)

Figure 5.9: A second configuration of the example in Figure 5.7 which does not lead to a worst-case scenario.

scenarios by identifying flows in indirect contention with f_a as non influent. Note that Property 1 is applied each time a flow is blocked by another one.

Property 2: Computing the maximal blocking delays.

If f_a is blocked by an unblocked flow f_j, its maximal blocking delay is bounded by $n_{f_j} \times 2d_{flit}$. This delay corresponds to the time needed by the last flit of f_j to leave the blocked router.

As seen previously, if f_a is blocked by f_j at router r_k, then these flows must have in common a next router r_{k+1} in their paths towards their destinations. Due to the pipeline transmission, f_a can progress only when the last flit of f_j leaves their next common router r_{k+1}. f_a is thus blocked for an amount of time equals to $n_{f_j} \times 2d_{flit}$. The scenario (a) on the timeline of Figure 5.4 is an example of this behavior. f_1 is blocked by the unblocked flow f_2. We can see that f_1 waits for a delay t that is equal to $n_{f_2} \times 2d_{flit}$, i.e. 8 cycles, before it can progress. Now, let us prove by contradiction that this delay **t** is the maximal blocking delay f_a can suffer when it is blocked by an unblocked flow.

Proof. Let us assume that a bigger blocking delay t_1 for f_a exists. Since $t_1 > n_{f_j} \times 2d_{flit}$, the header of f_j has moved from r_k to $r_{k+2n_{f_j}}$. Actually, when a flit moves from a router r_{k+1} to a router r_{k+2}, r_{k+1} transmits a credit back to r_k allowing the transmission of the next flit. This explains why flits of a flow are separated by one router when these flits follow the header in a pipeline way. To explain this behavior, let us go back to the scenario (a) of Figure 5.4, where we consider $r_1 = r_k$. We can see that when the header of f_2 is at the router r_{k+7}, i.e. at r_8, then the 2^{nd} flit is at r_{k+5}, i.e. at r_6. The 3^{rd} and 4^{th} flits are respectively at r_4 and r_2. Therefore, after t_1, the header of f_j should have been transmitted by router $r_{k+2n_{f_j}}$. The last flit of f_j is then located within router r_{k+2}. f_a has therefore progressed at router r_{k+1} which is contradictory with the initial assumption.

We note that when n unblocked flows block f_a at r_k, then the blocking delay of f_a is equal to $\sum^n n_{f_j} \times 2d_{flit}$. This is the case presented in the timeline of Figure 5.6. f_1 waits first for all flits of f_2 to quit the router r_2, i.e. $n_{f_2} \times 2d_{flit}$. Then, it waits the progression of all flits of f_3 to leave the router r_2, i.e. $n_{f_3} \times 2d_{flit}$.

However, a flow f_i could block f_j at router r_l. f_j should be unblocked first, in order to unblock f_a. This property must thus be applied recursively on all blocking flows. For this reason, we consider the property 2-bis.

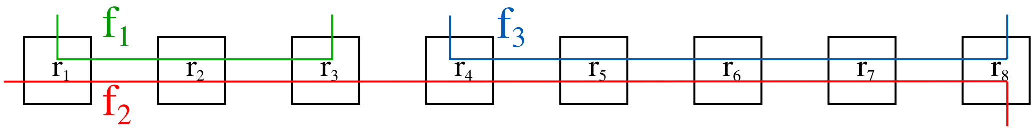
***Property 2-bis**: If f_a is blocked by a blocked flow f_j, the maximal blocking delay of f_a incurred by f_j, when this one is unblocked, depends on the number of hops separating r_k from r_l.*

When f_j is blocked by f_i at r_l, f_j has progressed from r_k before being blocked at r_l. Once blocked, f_j waits for an amount of time equal to $\mathbf{t}$, i.e. $n_{f_i} \times 2d_{flit}$, to receive the credit from r_{l+1}. Meanwhile, f_a is blocked at r_k and waits for the credit from r_{k+1} to progress: this blocking delay depends on the number of hops, noted n_{hops}, separating r_k and r_l. Three cases exists:

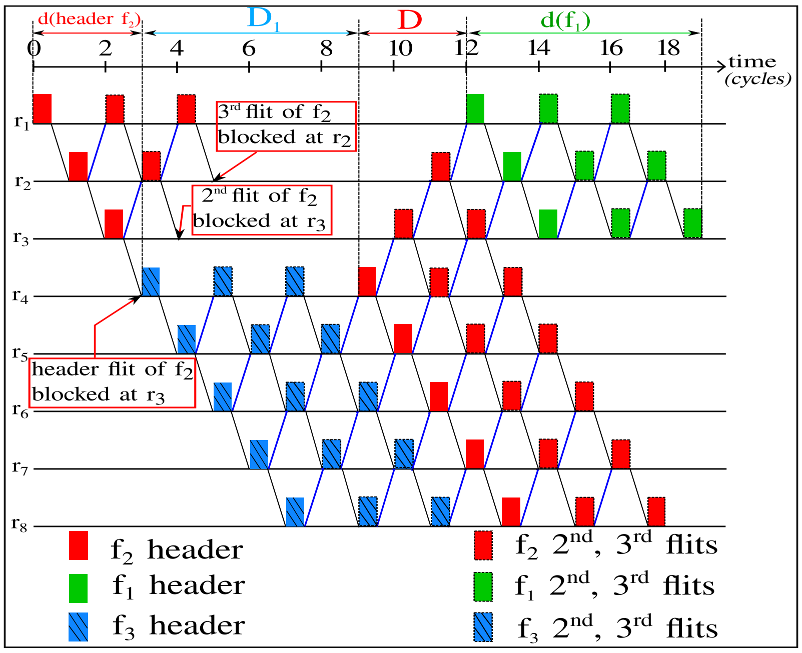
1. $n_{hops} = n_{f_j}$. In this case, the last flit of f_j is located within r_{k+1} when f_j has reached r_l, where it is blocked by f_i. Actually, r_l can be noted as $r_{k+n_{hops}}$ which is equal in this case to $r_{k+n_{f_j}}$. This means that the header of f_j is blocked at $r_{k+n_{f_j}}$, and so the last flit is located

at r_{k+1}. f_a can immediately progress when each flit of f_j progresses one router, i.e. the last flits leaves r_{k+1}. This is possible when f_j is no more blocked by f_i. Then, from this instant, the blocking delay that f_1 suffers, noted D is thus equal to: $n_{f_j} \times d_{flit}$.

We illustrate this case of this property on the example of Figure 5.10a where we consider three flows, each one has a size of three flits: f_1, f_2 and f_3. In this case, f_1 is blocked at r_1 by f_2, while f_2 is blocked by f_3 at r_4. Then, here $n_{hops} = n_{f_2} = 3$. The timeline



(a)

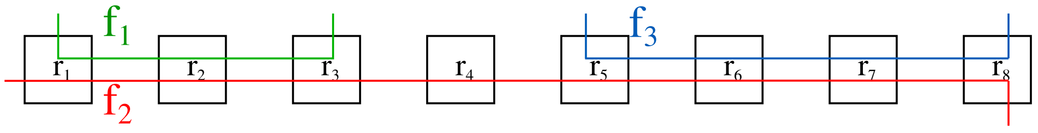


(b)

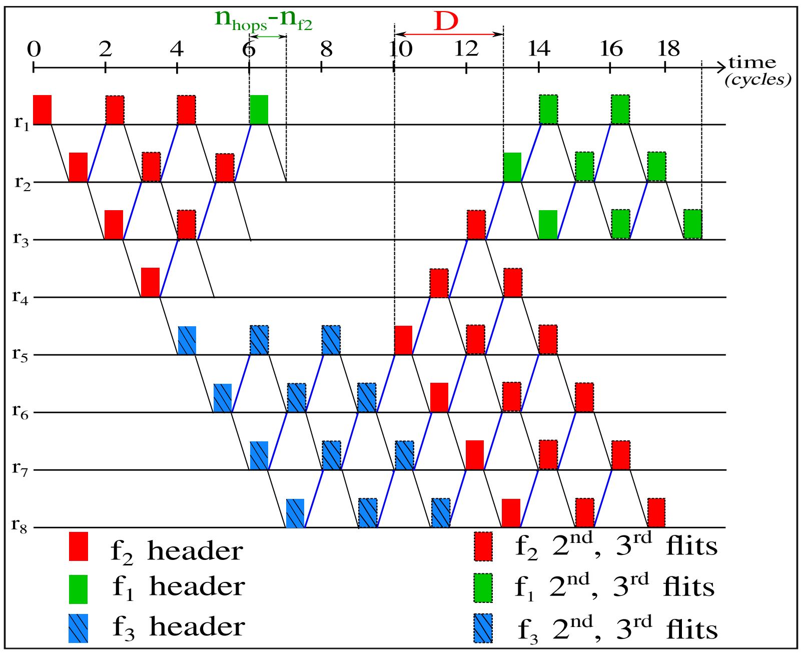
Figure 5.10: An example illustrating the case where $n_{hops} = n_{f_j}$.

in Figure 5.10b illustrates this case. We can see that the last flit of f_2 is blocked at r_2, waiting for the progression of the header. But, f_2 is blocked by the unblocked flow f_3, thus its blocking delay is equal to $n_{f_3} \times 2d_{flit} = 6$ cycles, noted by D_1. Now, f_1 waits for that each flit of f_2 moves to its next router, i.e. the last flit moves from the router r_2 to the router r_3. This blocking delay D is equal to $n_{f_2} \times d_{flit} = 3$ cycles. Therefore, $delay(f_1) = d(f_1) + d(header_{f_2}) + D + D_1 = 19$ *cycles*, where $d(header_{f_2})$ corresponds to the transmission delay of the header of f_2 from r_1 to r_4.

2. $n_{hops} > n_{f_j}$. Compared to the previous case, some flits of f_a must have already been transmitted by r_k when f_j reaches r_l. The previous blocking delay D of f_a at r_i is thus lowered by the number of routers the header of f_a has crossed. This number is equal to $n_{hops} - n_{f_j}$, as when f_j is blocked at $r_{k+n_{hops}}$, the header of f_a has reached $r_{k+n_{hops}-n_{f_j}}$. Then, the blocking delay of f_a at r_k when f_j is no more blocked corresponds to: $D - (n_{hops} - n_{f_j}) \times d_{flit}$.



(a)



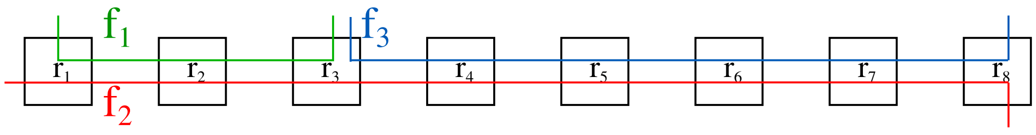
(b)

Figure 5.11: An example illustrating the case where $n_{hops} > n_{f_j}$

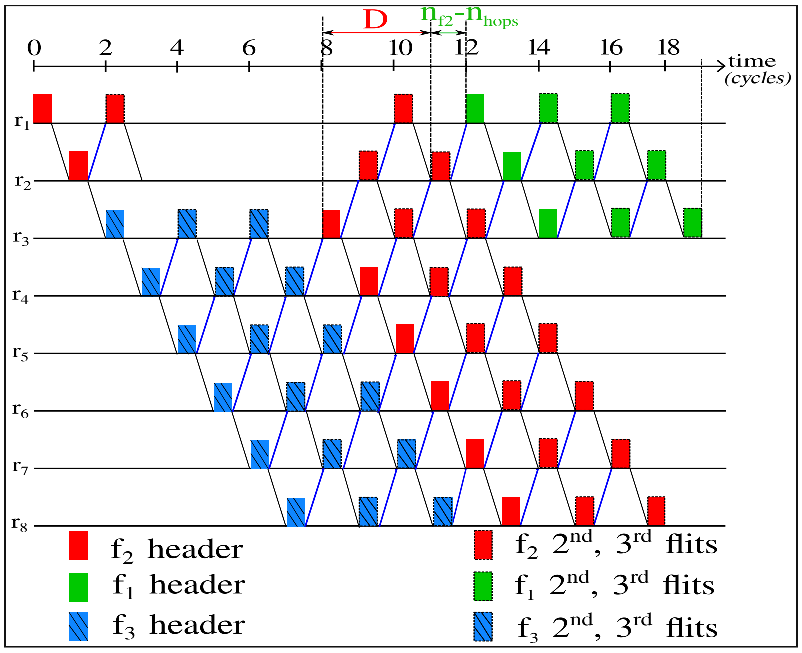
To illustrate this case, we modify the previous example, where f_2 is now blocked at r_5. We illustrate this example in Figure 5.11a, where $n_{hops} = 4$. The delay D, computed in the previous case, is lowered by $(n_{hops} - n_{f_j}) = 1$ as shown in Figure 5.11b. In fact, the header of f_1 has progressed $n_{hops} - n_{f_j}$, i.e. one router, when f_2 is blocked at r_2.

3. $n_{hops} < n_{f_j}$. In this case, a number of flits of f_j must be located on r_k and possibly on previous routers. f_a is thus blocked till these flits of f_j are transmitted by r_k. This number is equal to $n_{f_j} - n_{hops}$. The blocking delay of f_a of the first case therefore increases to: $D + (n_{f_j} - n_{hops}) \times d_{flit}$.

This case is illustrated in Figure 5.12a, where $n_{hops} = 2$ as f_2 is blocked at r_3. As illustrated in Figure 5.12b, when $n_{hops} < n_{f_2}$, only 2 flits of f_2 among 3 flits can progress before being blocked by f_3 at the router r_3. Thus f_1, blocked at r_1, should wait for the last flit of f_2, i.e. the $(n_{f2} - n_{hops})$ flits, to progress from r_1 to r_3. So, when f_2 is no more blocked, the blocking delay of f_1 at r_1 is equal to $D + (n_{f_j} - n_{hops}) \times d_{flit} = 4$ cycles.



(a)



(b)

Figure 5.12: An example illustrating the case where $n_{hops} < n_{f_j}$.

Finally, we recall that to obtain the total blocking delay of f_a, the blocking delay computed in each case must be added to the transmission delay of the header of f_j to reach r_l and its blocking delay at this router.

□

Property 3: Reducing the delays by identifying the non-influencing indirect flows. *f_d is a direct flow blocking f_a, while f_{id} is an indirect flow that blocks f_d (**Notations 7 and***

8). *f_{id} does not impact the progression of f_a when $E_r \geq n_{f_d} - 1$. We recall that E_r corresponds to the number of routers separating f_a and f_{id}* (***Notation 9)***.

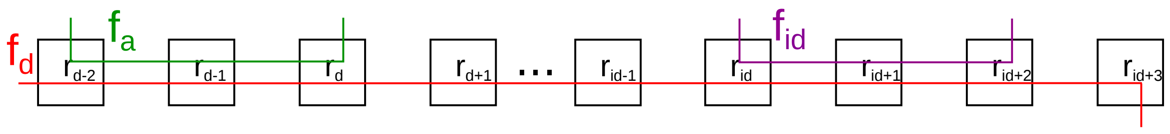


Figure 5.13: Example illustrating the analyzed flow f_a, the direct flow f_d and the indirect flow f_{id}.

Proof. Let us first consider Figure 5.13 in order to illustrate f_a, f_d and f_{id}. If $E_r \geq n_{f_d} - 1$, the header of f_d must therefore be blocked on r_{id} and the remaining flits are located on routers r_{d+1} up to r_{id-1}. r_d can therefore transmit the first flit of f_a which will then be able to progress in a pipeline way. Consequently, f_a is not affected by f_{id}, and thus its delay is independent from f_{id}.

To illustrate this property , we consider the example in Figure 5.14 where $E_r = 2$, as r_4 and r_5 separates f_1 from f_3. Let us first suppose that f_2 has a size of 3 flits, i.e. $E_r = n_{f_2} - 1$. This

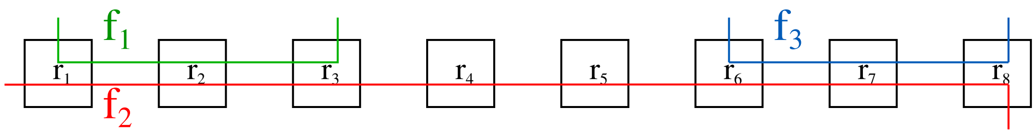
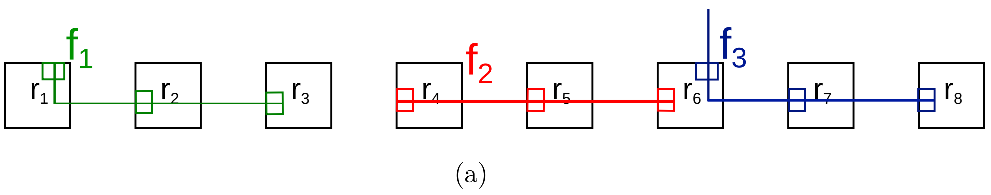


Figure 5.14: Example illustrating Property 3.

case is illustrated in Figure 5.15. In this case, the header of f_2 is blocked at r_6. The 2^{nd} and 3^{rd} flits are located at r_5 and r_4 as shown in Figure 5.15a. Hence, the path of f_1 is no more blocked. The timeline in Figure 5.15b illustrates the transmission of these flows where we can see that f_3 does not affect the transmission of f_1. Here, $delay(f_1) = n_{f_2} \times 2d_{flit} + d_{f_1} = 13\ cycles$.

However, if $E_r < n_{f_d} - 1$, a flit of f_d must be located on r_d since f_d is blocked at r_{id} by f_{id}. f_a is thus blocked on its path by f_d and thus indirectly delayed by f_{id}.

This is the case when f_2, in this example, has a size of 4 flits. This case is illustrated in Figure 5.16. In this case, E_r is less than $n_{f_2} - 1$, thus the 4^{th} flit of f_2 is located at r_3, blocking the progression of f_1 as shown in Figure 5.16a. f_1 should wait for the progression of f_3. The timeline, illustrating this transmission in Figure 5.16b, shows that f_3 affects indirectly f_1. f_1



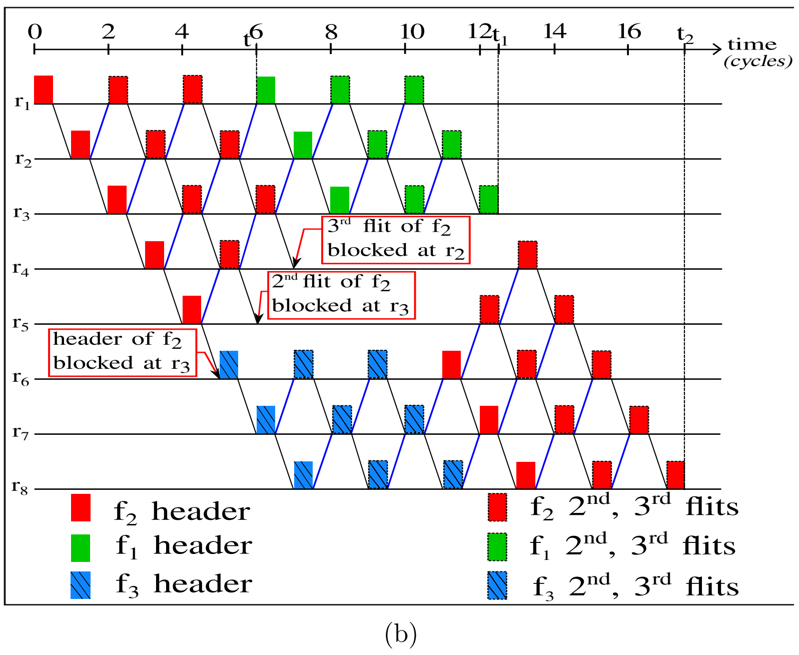
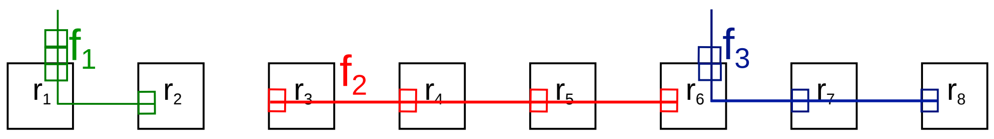


Figure 5.15: Transmission of flows f_1, f_2 and f_3 of the case in Figure 5.14 where we consider a size of 3 flits for each packet.

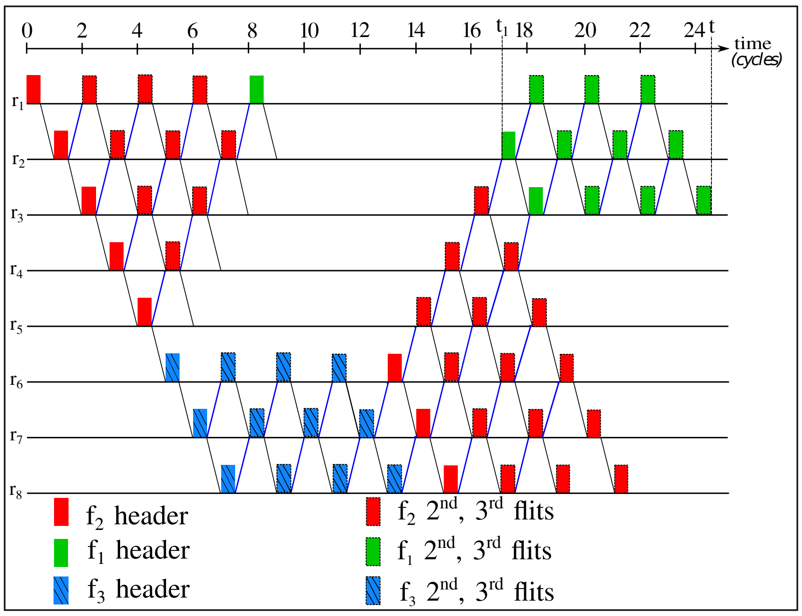
waits for f_2 to become unblocked by f_3, in order that the last flit of f_2 leaves the router r_3 at t_1. This case leads to a delay of f_1 that is equal to 25 *cycles*. $\square$

5.3 Description of the proposed pipeline-based algorithm

The pessimism when computing the WCTT can be reduced by modeling the pipeline behavior. Thus, the analysis should be performed at the granularity of flits, as presented in Properties 2 and 3. In this section, we illustrate on an example our algorithm RC_{NoC} to compute WCTT of flows. We then describe how RC_{NoC} implements the properties introduced at the previous section.



(a)



(b)

Figure 5.16: Timelines of transmission of flows f_1, f_2 and f_3 of the case in Figure 5.14 where we consider a size of 4 flits for each packet.

5.3.1 An illustrating example

In order to understand the behavior of our proposed algorithm, we propose to study the example in Figure 5.17. This example is made of 5 flows: f_1, f_2, f_3, f_4 and f_5. f_1 is the studied flow,

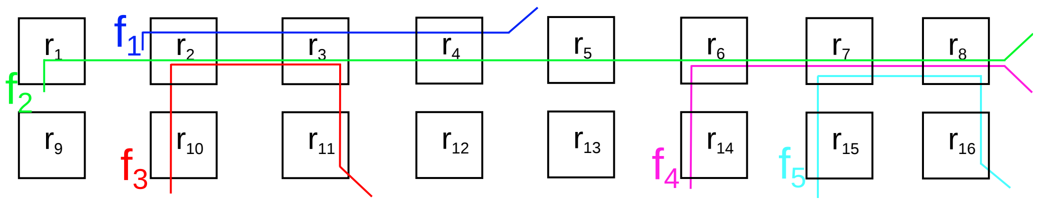


Figure 5.17: Example illustrating the method of computation.

i.e. the flow that we want to compute its WCTT. f_2 and f_3 are in direct contention with f_1. f_4 and f_5 are in indirect contention with f_1 as they are in direct contention with f_2 and share no common routers with f_1.

In order to compute the WCTT of f_1, we have to determine the list of flows that block directly and indirectly f_1. Thus first, we have to build, for each flow f_i, the list of flows that are in direct contention with f_i, and impacts f_1, by using Property 1. We note L_{f_i} this list and it is made of a set of couples $< f_j, r_k >$ where f_j is a flow in direct contention with f_i and r_k is the first router where f_j blocks f_i. The flows are ordered by the round robin arbitration, assuming Property 1. To identify when f_i can be transmitted by a router r_k and then progress, i.e. it is no longer blocked at r_k, f_i is also added in L_{f_i} when passing from a router to another.
For instance, in our example, $L_{f_1} = \{< f_2, r_2 > < f_3, r_2 > < f_1, r_2 >\}$. This means that f_2 and f_3 block f_1 at the same router r_2 as they share with f_1 a common link between r_2 and r_3. Considering the RRA, f_2 is sent before f_3. As f_2 and f_3 are two blocking flows of f_1 at r_2, then their blocking flows may also have an impact on the transmission of f_1. This requires to build their list of blocking flows starting from r_3. Thus, $L_{f_2} = \{< f_4, r_6 > < f_2, r_6 > < f_5, r_7 > < f_2, r_7 >\}$ and $L_{f_3} = \{< f_3, r_2 >\}$. Actually, f_2 blocks f_1 at r_2 and it is transmitted before f_3 due to RRA. This RRA at r_2 explains the reason why we do not consider f_3 as a blocking flow to f_2 and vice versa. f_2 can progress without being blocked till the router r_6. The recursion of Property 1 applies, and thus, f_2 is blocked by f_4 at r_6. When f_2 is no more blocked by f_4, it can progress to be blocked by f_5 at r_7. Now, f_2 is no more blocked on its path. For f_3, it progresses after f_2. On its path, f_3 is not blocked.
Similar to f_2 and f_3, as f_4 and f_5 block f_2 at r_6 and r_7 respectively, then their blocking flows could also have an impact on f_1. Thus, we build their list of blocking flows starting respectively from r_7 and r_8. Then, $L_{f_4} = \{< f_5, r_7 > < f_4, r_7 >\}$ and $L_{f_5} = \{< f_5, r_7 >\}$.

Now, we have finished to build the list of flows that have an impact, directly and indirectly, on the analysis of f_1. Let us now see how to analyze the WCTT of f_1 by stepping through these lists by starting from L_{f_1}, as f_1 is the studied flow. L_{f_1} indicates that f_2 is the first direct flow blocking f_1. Therefore, we have to analyze L_{f_2} in order to compute the maximal blocking delay after which the next flows after f_2 in L_{f_1}, i.e. f_3 and f_1, could progress. Since both f_4 and f_5 in L_{f_2} are indirect flows of f_3 and f_1, i.e the next flows that progress after f_2 in L_{f_1}, then two scenarios can occur:

- **First scenario: $n_{f_2} = 2$.**

  Property 3 is applied: f_4 has no indirect influence over f_3 and f_1, since one router at minimum separates f_4 from these flows, i.e. $E_r = n_{f_2} - 1 = 1$. Consequently, f_2 becomes an unblocked flow for the flow after f_2 in L_{f_1}, i.e. f_3. This means that f_3 can progress directly after f_2 when its last flit leaves the router r_3. Thus, the blocking of f_4 to f_2 does not impact the transmission of f_3. Property 2 is therefore applied: the maximal blocking delay of f_3 before it can progress is computed and is equal to $n_{f_2} \times 2d_{flit}$. Now, f_3 can progress. Thus, we step to L_{f_3} to compute the maximal blocking delay from which suffer the next flow after f_3 in L_{f_1}, i.e. f_1, before it can progress. L_{f_3} indicates that f_3 is an unblocked flow. Thus, the next flow f_1 waits only the transmission of f_3. Property 2 applies that f_1 waits $n_{f_3} \times 2d_{flit}$ before it can progress. Now, f_1 can progress and thus the transmission time of f_1 till its destination must be added to compute its WCTT. This WCTT is thus equal to: $d(f_1) + (n_{f_2} + n_{f_3}) \times 2d_{flit}$. The timeline in Figure 5.18 illustrates the transmission of these flows where we can see that $WCTT(f_1) \approx 13cycles$.

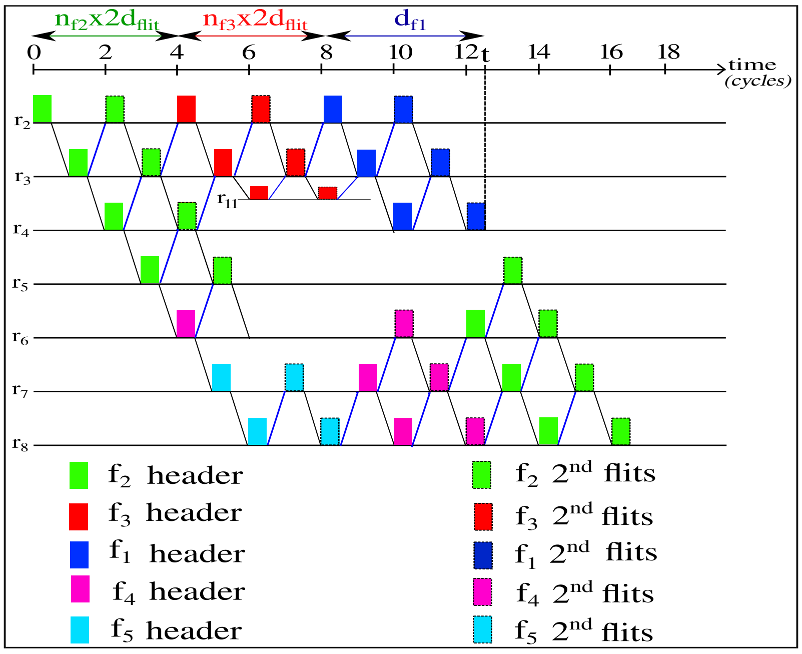


Figure 5.18: Timeline illustrating the transmission of flows of the example in Figure 5.17 and considering the first scenario.

- **Second scenario: $n_{f_2} > 2$.**

  Two cases could be considered in this scenario.

- **First case: $n_{f_2} = 3$.** In this case, f_4 does not impact the transmission of the flow f_3 but impacts f_1. Actually, there are one router separating f_4 from f_1 and thus $E_r < n_{f_2} - 1$. On the other hand, there are two routers separating f_4 from f_3, which means that $E_r = n_{f_2} - 1$.
- **Second case: $n_{f_2} > 3$.** In this case, f_4 has an indirect influence on both f_3 and f_1, as the number of routers separating f_4 from f_1 or f_3 is less than $n_{f_2} - 1$.

In this example, we show the analysis by considering the second case. As f_4 has an influence on both of these flows, therefore we need to analyze L_{f_4}. L_{f_4} tells us that f_4 is blocked by f_5, which requires the analysis of L_{f_5} in order to compute the maximal blocking delay of f_4. However, f_5 is not blocked, as presented in L_{f_5}, so Property 2 computes the maximal delay of f_4, which is now no more blocked as there are no more flows in L_{f_4}. Now, as f_4 is an unblocked flow, the maximal blocking delay that the flow after f_4 in L_{f_2} suffers from it is computed by Property 2. Going back to L_{f_2}, f_2 can now progress but this list indicates that f_2 progress only one router, i.e. r_6, before being blocked by f_5 at r_7. Here, f_5 has an influence on f_3 and f_1, i.e. the flows after f_2 in L_{f_1}. Thus, the analysis of both L_{f_3} and L_{f_5} is required.

This behavior has been implemented into an algorithm that is described in the next section.

5.3.2 Identifying the global worst-case scenario

RC_{NoC} is divided into two parts. The goal of the first part, illustrated in Algorithm 1, is to identify the set of blocking flows of f_a, denoted F_p (**Definition 2**). The second part described in Algorithm 3 computes the WCTT of f_a by adding the delays of the blocking flows by considering the properties defined in section 5.2.

Computing the blocking flows F_p

This first part consists to build F_p from the set of all flows exchanged in a given configuration, i.e. F (**Definition 1**). This part is composed from four steps as described in Algorithm 1.

Algorithm 1 Compute_Blocking(F)

```
1: $F_p.insert(f_a)$
2: $f_i \leftarrow F_p.head()$
3: do
4:     $F.remove(f_i)$
   #Step 1: Compute the direct blocking flows of $f_a$
5:     if $((f_i = f_a)||(f_i.same\_source(f_a))$ then
6:         for all $r_k \in [r_{isource}, r_{idestination}]$ in $f_i$ do
7:             $Compute\_direct\_blocking(F_p, F, f_i, L_{f_i}, r_k)$
8:         end for
   #Step 2: Compute the indirect blocking flows of $f_a$
9:     else
10:        for all $r_k \in ]r_{iblocked}, r_{idestination}]$ in $f_i$ do
11:            $Compute\_direct\_blocking(F_p, F, f_i, L_{f_i}, r_k)$
12:        end for
13:    end if
   #Step 3: Applying the RRA
14:    for all $f_j \in L_{f_i}$ do
15:        if $((!f_j.samesource(f_a))$ & $(f_j.sameport(\forall f_l \in L_{f_i}, r_{jblocked})))$ then
16:            $F.remove(f_l)$
17:            $F_p.remove(< f_l, r_{jblocked} >)$
18:            $L_{f_i}.remove(< f_l, r_{jblocked} >)$
19:        end if
20:    end for
   #Step 4: Following the progression of $f_i$ in $L_{f_i}$
21:    for all $f_j \in L_{f_i}$ do
22:        if $(r_{jblocked}$ in $< f_j, r_{jblocked} > \neq r_{kblocked}$ in $next(< f_j, r_{jblocked} >))$ then
23:            $L_{f_i}.insert(< f_i, r_{jblocked} >)$
24:        end if
25:    end for
26:    $f_i = F_p.next()$
27: while $!Fp.last\_reached()$
```

In the first step, we compute the direct blocking flows of f_a that block its path (**Notation 6**) from the source router $r_{asource}$ to the destination router $r_{adestination}$ (lines 5 to 8). Thus, here we consider the flows blocking f_a at the source, i.e. sharing the same source, and on its path, i.e. sharing the same links.

The second step consists to compute the indirect blocking flows of f_a. As seen previously, it is necessary to consider the blocking flows of each flow f_i in F_p (initially from L_{f_a}). Their L_{f_i} must thus also be computed and any flow $f_j \in L_{f_i}$ $(f_j \notin F_P)$ must be added in F_p (lines 10 to 12).

In the third step, i.e. after computing the different blocking flows, we must consider the RRA

to choose one blocking flow from each port of each router $r_{jblocked}$, i.e. the router where f_j blocks a flow f_i in L_{f_i} (lines 14 to 20).
The final step consists on following the progression of each flow on their blocked routers. Thus, the flow f_i is also added to its list of blocking flows L_{f_i} each time that f_i can progress some routers before it is blocked by other flows at the following routers on its path (lines 21 to 25).

How to compute the blocking flows?

The function *Compute_direct_blocking*(), described in Algorithm 2, identifies the direct blocking flows of a flow f_i. Thus, for the analyzed flow f_a, we must consider two types of direct blocking flows. The first type are the flows that block f_a at its source (lines 2 to 4). The second type are the direct flows sharing a part of the path or the same destination with the analyzed flow. Thus, in this type, it is sufficient to identify two consecutive common routers between the analyzed flow and a flow f_j to consider f_j as a direct blocking flow (lines 6 to 10). Therefore, each flow of L_{f_a}, identified as blocking flow, is added in F_p. The blocking flows of f_i are computed by determining the flows sharing a part of its path. But, we must consider this path from the next router of $r_{iblocked}$ where f_i blocks f_a (line 10 in Algorithm 1). We note that the flows blocking f_i at the router $r_{iblocked}$ are not added to L_{f_i} as they blocked f_i once due to the RRA.

Algorithm 2 Compute_direct_blocking($F_p, F, f_i, L_{f_i}, r_k$)

```
1:  for all (($f_j \in F$) & ($f_j \notin L_{f_i}$)) do
    #Step 1: Compute the direct blocking flows sharing the same source of $f_a$
2:      if ($f_j.same\_source(f_a)$ & ($f_i = f_a$)) then
3:          $F_p.insert\_head(< f_j, r_{isource} >)$
4:          $L_{f_i}.insert\_head(< f_j, r_{isource} >)$
    #Step 2: Compute the direct blocking flows sharing the same path of $f_a$ or $f_i$
5:      else if ($!f_j.same\_port(f_i, r_k)$) then
6:          if (($r_k.common\_router(f_i, f_j)$) & ($r_k.next().common\_router(f_i, f_j)$)) then
7:              $r_{jblocked} = r_k$
8:              $F_p.push(< f_j, r_{jblocked} >)$
9:              $L_{f_i}.push(< f_j, r_{jblocked} >)$
10:         end if
11:     end if
12: end for
```

Example. We go back to the example in Figure 5.17, to describe how to build the list of blocking flows using Algorithms 1 and 2. In Algorithm 1, we thus begin by the first step (lines from 5 to 8), where we have to consider the path of the studied flow f_1 from r_2 to r_4 in order to determine its direct blocking flows. f_1 is not blocked at source but it is only blocked at its path. Thus, the second step in Algorithm 2 (lines from 6 to 10) applies that r_2 and r_3 are two common consecutive routers between f_2 and f_1. Then, f_2 is added to F_p and $< f_2, r_2 >$ is added to L_{f_1}. Similar to f_2, f_3 is also a direct blocking flow of f_1 and so f_3 is added to F_p and $< f_3, r_2 >$ is added to L_{f_1}.
As now there are no more direct blocking flows for f_1 on its path, we move to the next step (Step 2 from line 10 to 12), where we build the list of blocking flows for the flows in F_p, i.e. f_2 and f_3. Actually, f_3 is not considered a direct blocking flow to f_2 as both f_2 and f_3 block f_1 at the same router. f_4 and f_5 are identified by the second step of Algorithm 2 as direct blocking flows of f_2. Thus, F_p contains now f_3, f_4 and f_5. This recursion of this step is applied on the different flows in F_p and ends when no new flows are added to F_p.
Finally, the step 4 of Algorithm 1 is applied on L_{f_1}, L_{f_2}, L_{f_3}, L_{f_4} and L_{f_5}. For example, it adds $< f_1, r_2 >$ at the end of L_{f_1} to indicate that f_1 can progress.

5.3.3 Implementation of direct and indirect contention analysis

This part consists to compute the WCTT of f_a by stepping through the list of blocking flows, L_f, built in the previous algorithm. Each direct blocking flow is identified and its blocking delay, after which it leaves the blocked router, is computed.
Algorithm 3 describes this computation of the WCTT of f_a. For each analyzed flow f, we compute (lines 3 to 6): its first direct blocking flow in L_f (noted f_b), its next direct blocking flow in L_f (f_{bnext}), the blocking flow of f_b in L_{f_b} (f_{bb}) and the next blocking flow of f_b in L_{f_b} (f_{bbnext}). These variables are used to compute the delay incurred by f_b, potentially blocked by f_{bb}, in order to give the credit to the router allowing the progression of f_{bnext}. A list L is defined to add at each time the flow that is blocked and so it helps to indicate which flow can progress after the release of the path of the blocking flows. Initially, L contains the analyzed flow f_a.

Algorithm 3 Compute_Delay(f, L_f, r_{leave})

```
1: $delay = 0$
2: $influent = false$
   **#Step 1: Compute the first direct blocking flows of $f_i$ and their blocking flows**
3: $< f_b, r_b > = L_f$.get_$f_b()$
4: $< f_{bb}, r_{bb} > = L_{f_b}$.get_$f_b()$
5: $< f.f_{bnext}, f.r_{bnext} >= L_f$.get_$f_b().next()$
6: $< f_{bbnext}, r_{bbnext} > = L_{f_b}$.get_$f_b().next()$
   **#Case 1: $f_b$ can progress some routers, we verify if it will be blocked or not?**
7: **if** $(f_b == f_{bb})$ **then**
8:     $delay+ = check\_Next(L_f, f_b, f_{bbnext}, L)$
9: **end if**
   **#Case 2: $f_b$ is blocked directly by $f_{bb}$, and $f_{bb}$ is in direct contention with $f_{bnext}$**
10: **if** $(f_{bb}.is\_direct(f.f_{bnext}))$ **then**
11:     $delay+ = f_b.progress\_up(r_{bb})$
12:     $L.add(f_b)$                                    ▷ $f_b \notin L$
13:     **return** $delay + compute\_Delay(f_b, L_{f_b}, r_{bnext})$
14: **end if**
   **#Case 3: $f_b$ is blocked directly by $f_{bb}$, and $f_{bb}$ is in indirect contention with $f_{bnext}$**
15: **if** $(f_{bb}.is\_indirect(f_{bnext}))$ **then**
16:     $delay+ = check\_influence(L_f, f_b, f_{bb}, f_{bnext}, L)$
17: **end if**
18: **return** $delay$
```

Implementing the Property 2: Check whether blocking flows are blocked or not By definition on how lists of blocking flows are built, the case $f_{bb} = f_b$ means that f_b can progress some routers before it is blocked by a flow f_{bbnext} or it is not or no more blocked. At line 8, the function *check_Next* deals with the cases where f_b is blocked or not. Algorithm 4 describes this function. In the case where f_b is blocked, f_{bbnext} is not NULL (lines 1 to 6). This means that f_b can be blocked by several f_{bbnext} at r_{bbnext} before no longer blocking f at r_b. f_b can thus first progress from its current location r_{bb} to the first r_{bbnext}. This delay is computed by the function *progress_up*() and is equal to: $d_{flit} \times (r_{bbnext} - r_{bb})$.

However, as f_b can progress in its path, this progression can release the path of the flow f blocked by f_b. Thus, if a flow blocked by f exists, its analysis started in a previous iteration must next be resumed, in order to check whether f_{bbnext} is now non influent over f_{bnext} and can also be discarded (Property 3). The function *Compute_delay*() is iterated on the last flow blocked by f in the list L (which is expressed at line 4). If not, the analysis of f must be continued on the next flow blocking f, i.e. calling *compute_Delay* with f_{bb} set to f_{bbnext} (lines

Algorithm 4 check_Next(L_f, f_b, f_{bbnext}, L)

```
    #Case 1: $f_b$ is blocked
 1: if ($f_{bbnext} \neq NULL$) then
 2:     $delay+ = f_b.progress\_up(r_{bbnext})$
 3:     $f_b.setTo(f_{bbnext})$
 4:     $f = L.previous\_tail()$
 5:
 6:     return $delay + Compute\_Delay(f, L_f, r_{bnext})$
    #Case 2: $f_b$ is no more blocked
 7: else
 8:     if ($f_b == f_a$) then
 9:         return $delay + f_a.progress\_leave(r_{adestination})$
10:     else
11:         $delay+ = f_b.give\_credit(r_{bnext})$
12:         $f.setTo(f_{bnext})$
13:         $L.remove(f_b)$
14:         $f = L.previous\_tail()$
15:
16:         return $delay + Compute\_Delay(f, L_f, r_{bnext})$
17:     end if
18: end if
```

4 and 6).

In the other case (lines 7 to 17), when f_b is no longer blocked, f_b can therefore progress to give the credit to f_{bnext} at router r_{bnext}, i.e. f_b leaves the router r_{bnext}. This corresponds to the use of the Property 2 to compute the maximal delay before that f_{bnext} can progress. This delay is computed by the function *give_credit*(). Also, here the same iteration is done as in the previous case to identify the state of progression of the flow previously blocked. However, in the case where f_b is the flow f_a, so the analyzed flow is no more blocked and can progress to its destination router $r_{adestination}$ from the last router r_b where it was blocked. This delay is equal to $d_{flit} \times (r_{adestination} - r_b + 1) + 2d_{flit} \times (n_{f_a} - 1)$, and it is computed by the function *progress_leave*() at line 9. We note that $d_{flit} \times (r_{adestination} - r_b + 1)$ computes the transmission delay of f_a to reach its destination.

Implementing Properties 2 and 3: Check whether blocking flows are direct or not

In the other case where $f_b \neq f_{bb}$, f_b is blocked by f_{bb}. However, f_{bb} can be either in direct or indirect contention with the flow f_{bnext}. If f_{bb} is in direct contention, the time needed by

f_b to progress to r_{bb}, i.e. its blocking router, must first be accounted by using the function *progress_up*() (line 11). Then, f_b must be analyzed to compute its blocking delay until it can progress and so allowing the progression of f_{bnext}. Thus, f_b is added to the list L (lines 12 and 13).

Otherwise, when f_{bb} is an indirect contention with f_{bnext}, the function *check_influence* (line 15) establishes whether f_{bb} is influent over f_{bnext} or not by applying Property 3. Algorithm 5 describes this function. The computation to be performed when f_{bb} is influent is identical to the case where f_{bb} is in direct contention with f_{bnext} (lines 4 to 7 which are similar to lines 10 to 13 in Algorithm 3). However, two cases must be considered when f_{bb} is non influent.

The first case is when f_{bb} has no influence over any remaining flow f_x in L_f ($f_x \neq f_b$ and $f_x \neq f_{bnext}$)(line 11 and lines 17 to 22). Thus, at r_{bnext}, f_b gives the credit to f_{bnext}, i.e. f_b can progress and leaves the router r_{bnext}, allowing the transmission of f_{bnext}. This blocking delay of f_{bnext}, by waiting the progression of f_b, is computed by applying Property 2. If a flow blocked by f exists, its analysis must next be resumed. If not, the analysis of f must be continued with f_{bb} set to f_{bbnext}. The second case is when f_{bb} has an influence on a flow f_x (line 13 and lines 23 to 37). In this case, the maximal delay is obtained by exploring two scenarios as the impact of future contentions are unknown at this point. In the first scenario (lines 24 to 29), all the flows in L_f before f_x are skipped when further analyzing f. Before, the analysis continues similarly to the direct case (lines 11 and 12 in Algorithm 3) but with $f_{bnext} = f_x$. In the second scenario (lines 30 to 35), the computation skips f_{bb} and f_b gives the credit to f_{bnext}. And so the analysis is similar to the first case.

Example. Let us illustrate how this algorithm is applied on the first scenario of the proposed example in section 5.3.1 ($n_{f_2} = 2$).

First, lets recall the list of blocking flows:

- $L_{f_1} = \{< f_2, r_2 > < f_3, r_2 > < f_1, r_2 >\}$
- $L_{f_2} = \{< f_4, r_6 > < f_2, r_6 > < f_5, r_7 > < f_2, r_7 >\}$

Algorithm 5 check_influence($L_f, f_b, f_{bb}, f_{bnext}, L$)

```
1: $delay\_ind = 0$
2: $delay\_d = 0$
3: $L_{ind} = L$
   #Case 1: Property 3 indicates that $f_{bb}$ is influent to $f_{bnext}$
4: if $(f_{bb}.is\_influent(f_{bnext}))$ then
5:     $delay+ = f_b.progress\_up(r_{bb})$
6:     $L.add(f_b)$
7:     return $delay\ +\ Compute\_Delay(f_b, L_{f_b}, r_{bnext})$
   #Case 2: Property 3 indicates that $f_{bb}$ is not influent to $f_{bnext}$
8: else
9:     for all $f_x \in ]f_{bnext}, f]$ in $L_f$ do
10:        if $((f_{bb}.is\_non\_influent(f_x))$ then
11:            $influent = False$
12:        else
13:            $influent = True$
14:            $break$
15:        end if
16:    end for
   #Case 3: Property 3 indicates that $f_{bb}$ is not influent to $f_x$
17:    if $influent = false$ then
18:        $delay+ = f_b.give\_credit(r_{bnext})$
19:        $f.setTo(f_{bnext})$
20:        $L.remove(f_b)$
21:        $f = L.previous\_tail()$
22:        return $delay\ +\ Compute\_delay(f, L_f, r_{bnext})$
   #Case 3: Property 3 indicates that $f_{bb}$ is influent to $f_x$
23:    else
24:        $f_{bnext\_init} = f_{bnext}$
25:        $f_{bnext} = f_x$                         ▷ $f_{bnext}$ in $L_f$ is set to $f_x$
26:        $delay_d = f_b.progress\_up(r_{bb})$
27:        $L.add(f_b)$
28:        $delay_d+ = Compute\_delay(f_b, L_{f_b}, r_{bnext})$
29:        $f_{bnext} = f_{bnext\_init}$
30:        $delay\_ind = f_b.give\_credit(r_{bnext})$
31:        $f.setTo(f_{bnext}$
32:        $L.remove(f_b)$
33:        $L.previous\_tail()$
34:        $delay\_ind+ = Compute\_delay(f, L_f, r_{bnext})$
35:    end if
36:    return $delay+ = max(delay\_ind, delay_d)$
37: end if
```

- $L_{f_3} = \{< f_3, r_2 >\}$
- $L_{f_4} = \{< f_5, r_7 > < f_4, r_7 >\}$
- $L_{f_5} = \{< f_5, r_7 >\}$

We start by L_{f_1} as f_1 is the studied flow. Initially, L contains f_1. Let us compute f_b, f_{bb}, f_{bnext} and f_{bbnext} (lines from 3 to 6 in Algorithm 3):

- $< f_b, r_b >=< f_2, r_2 >$: f_2 blocks directly f_1.
- $< f_{bb}, r_{bb} >=< f_4, r_6 >$: f_4 blocks directly f_2.
- $< f_{bnext}, r_{bnext} >=< f_3, r_2 >$: f_3 is the next flow that progress after f_2.
- $< f_{bbnext}, r_{bbnext} >=< f_2, r_6 >$: f_2 is the next flow that progress after f_4.

As $f_b \neq f_{bb}$, then we have to check the influence of f_{bb}, i.e. f_4, on f_{bnext}, i.e. f_3. f_4 is an indirect flow over f_3 (line 5 in Algorithm 3), thus the function $check\_influence()$ is used to apply Property 3 and identifies the influence of f_4 over f_3. By considering the first scenario of this example, Property 3 applies that f_4 is a non influent flow to f_3. This implies to check this influence over $f_x \in]f_{bnext}, f]$ in L_f (line 9 in Algorithm 5), i.e. $f_x \in]f_3, f_1]$ in L_{f_1}. In this case, f_x corresponds only to f_1. f_4, in the same scenario, does not have an influence over f_1 ($influent = false$). Then, f_2 can progress and leaves the router r_3, allowing the transmission of f_3 by applying Property 2. f_3 waits only $n_{f_2} \times 2d_{flit}$.
f_b is now equal to f_3 in L_{f_1}. We iterate the function $Compute\_blocking()$ on f_1 as it is the last flow blocked in L. Here:

- $< f_b, r_b >=< f_3, r_2 >$
- $< f_{bb}, r_{bb} >=< f_1, r_2 >$
- $< f_{bnext}, r_{bnext} >=< f_3, r_2 >$
- $< f_{bbnext}, r_{bbnext} >= NULL$

$f_b = f_{bb}$ means that f_b, i.e. f_3, can progress. This is the case described in lines from 7 to 9 in Algorithm 3. Thus, the function *check_Next*() is applied to verify if f_3 is blocked or not. f_{bbnext} is $NULL$ in this example. As $f_3 \neq f_a$, the flow f_3 can progress to leave the router r_2 allowing the transmission of f_1 (lines 11 to 16 in Algorithm 4). This means that Property 2 is applied, where f_1 waits only $n_{f_3} \times 2d_{flit}$, which is computed by the function $give\_credit(r_2)$. As f_1 still the last blocked flow in L so the function $Compute\_blocking()$ is iterated on L_{f_1} and f_b is set to f_{bnext}, i.e. f_1. Now:

- $< f_b, r_b >=< f_1, r_2 >$
- $< f_{bb}, r_{bb} >= NULL$
- $< f_{bnext}, r_{bnext} >=< f_1, r_2 >$
- $< f_{bbnext}, r_{bbnext} >= NULL$

In this iteration, $f_b = f_{bb}$, $f_{bnext} = NULL$ and $f_b = f_a$ (line 8 in Algorithm 4). This indicates that f_1 can progress to its destination r_4. This delay is computed by the function $progress\_leave(r_2)$ and it is equal to:

$$(r_4 - r_2 + 1) \times d_{flit} + 2d_{flit} \times (n_{f_a} - 1) = 3d_{flit} + 2d_{flit} = 5d_{flit}.$$

We note that $(r_4 - r_2 + 1) \times d_{flit}$ corresponds to the transmission delay of the header from r_2 to r_4. If we consider that the packets are made of 2 flits, thus:

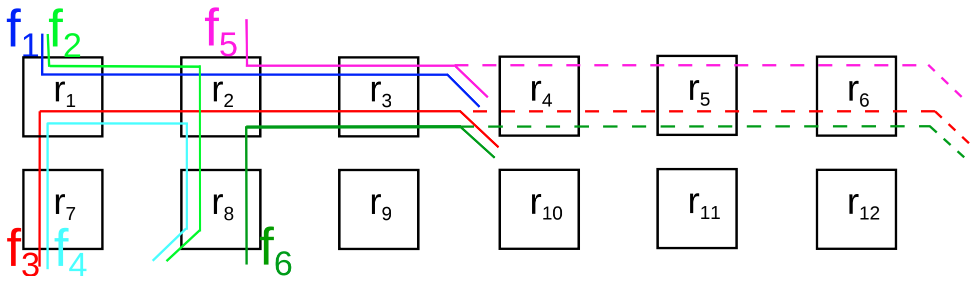
$$WCTT(f_1) = 5d_{flit} + (n_{f_2} + n_{f_3}) \times 2d_{flit} = 13\ cycles.$$

This value corresponds to the timeline obtained in Figure 5.18.

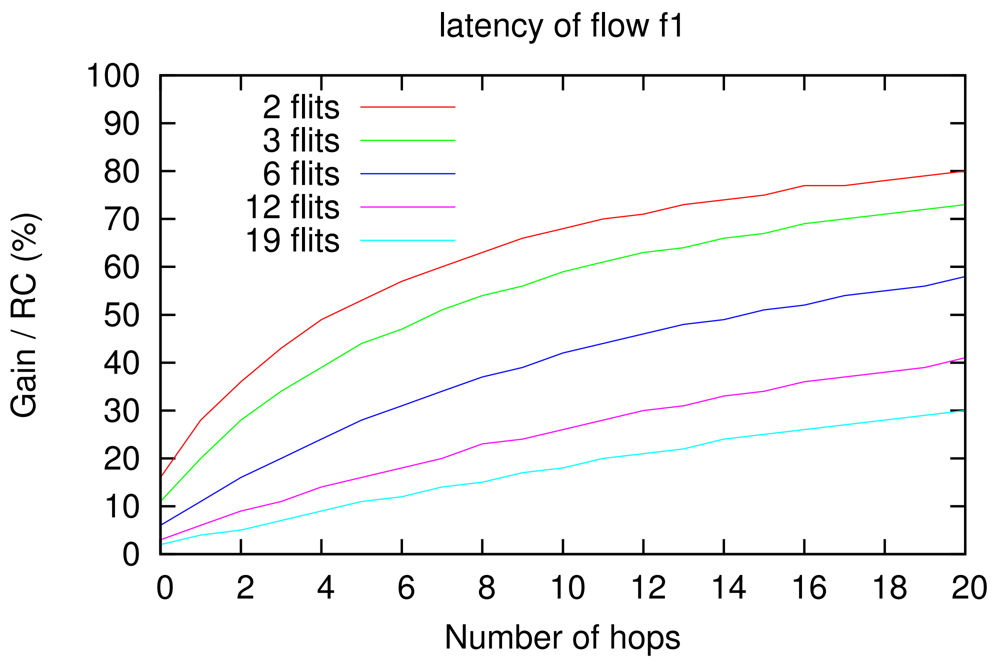
5.4 Unitary evaluation of the properties

This section presents an evaluation of our algorithm made over a synthetic benchmark. The evaluation over the synthetic benchmark reports how Properties 2 and 3 can reduce the pessimism of the WCTT by comparing to the classical RC method. To quantify the improvements against the RC method, we compute the following metric: $\frac{(d_{RC}-d_A)\times 100}{d_{RC}}$, where d_{RC} is the WCTT value returned by the RC method and d_A the value returned by RC_{NoC}. We also report on how the modifications of some parameters, such as the number of flits or the size of the NoC, affect our results.

We used a SpaceWire application described in [FFF09a]. Since NoCs used within many-cores are larger than a SpaceWire network, we vary the destination of flows of the initial SpaceWire application. We consider the mapping shown in Figure 5.19a, extracted from the SpaceWire application. We compute the WCTT of f_1, i.e. $f_a = f_1$, and present several configurations to explain the gain due to the properties 2 and 3.



(a)



(b)

Figure 5.19: Mapping of flows in the first configuration and normalized gain compared to the RC method.

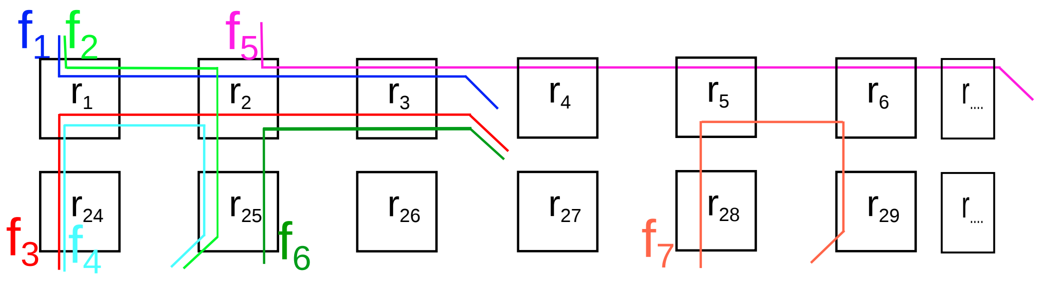
Impact of Property 2. The goal of this first configuration is to quantify, over specific mapping of flows over a NoC, how Property 2 improves the computation of the WCTT. As shown in Figure 5.19a, we generate different mapping of flows by modifying the destination of some of the direct flows represented in dotted lines. For each mapping, we compute the WCTT in order to study how the number of hops, they make after the destination of f_1, affects the results. For instance, a number of hops equals to 0 means that these direct flows have the same destination as f_1, while a number of hops 3 corresponds to the dotted lines.

Figure 5.19b shows the gain obtained for various number of hops and different length of packets. In the classical RC method, a flow cannot progress until its direct blocking flows reach their destinations. The WCTT computed by RC therefore increases with the number of hops. Conversely, Property 2 states that this delay only depends on the number of flits of the direct flow. Then, by using RC_{NoC}, for each packet of a given number of flits, the WCTT remains constant when increasing the number of hops. The blocked flow indeed always waits for the same amount of time the credit from the blocking direct flow, independently of its destination. This difference between the classical RC and RC_{NoC} explains the shape of each curve.

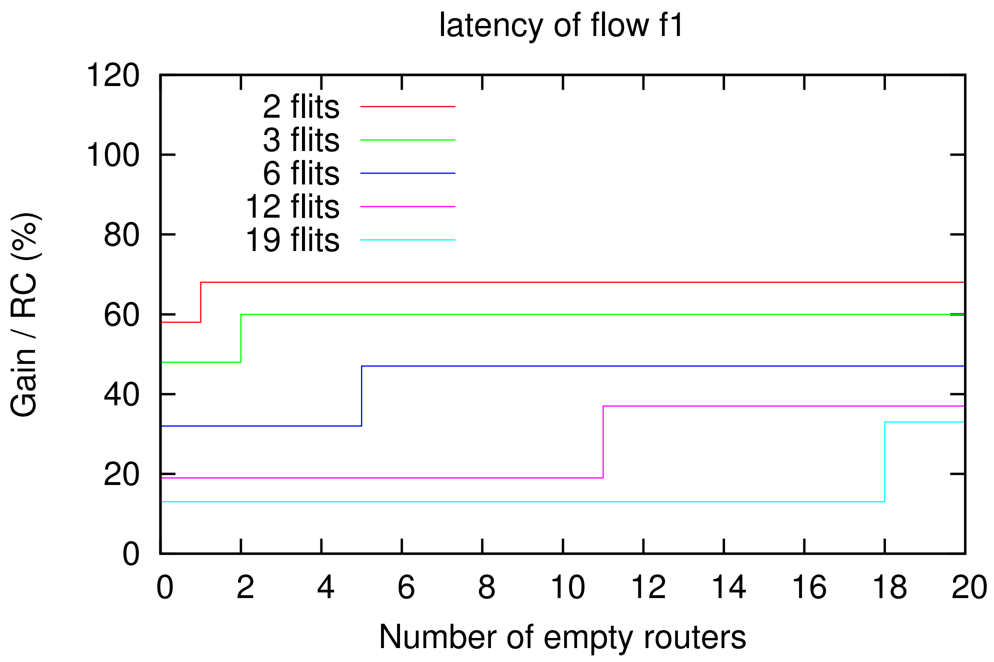
For a given number of hops, the gain decreases when increasing in the number of flits of the packet. The blocking delay in both methods indeed increases as more flits of the direct blocking flows have to be transmitted by routers. Also, the larger the packet is, the closer is the maximum blocking delay of a flow blocked by f_i to the waiting delay for f_i to reach its destination.

Impact of Property 3. The goal of this second configuration is to quantify the gain, over specific mapping of flows, due to Property 3. Compared to the previous configuration, we therefore add an indirect flow f_7, that blocks f_5 a direct flow of f_1, as shown in Figure 5.20a. We also set the destination of f_5 21 hops after the destination of f_1. The value of E_r is one parameter that determines whether an indirect flow is influent or not. The router where f_7 blocks f_5 ranges from r_5 to r_{23}.

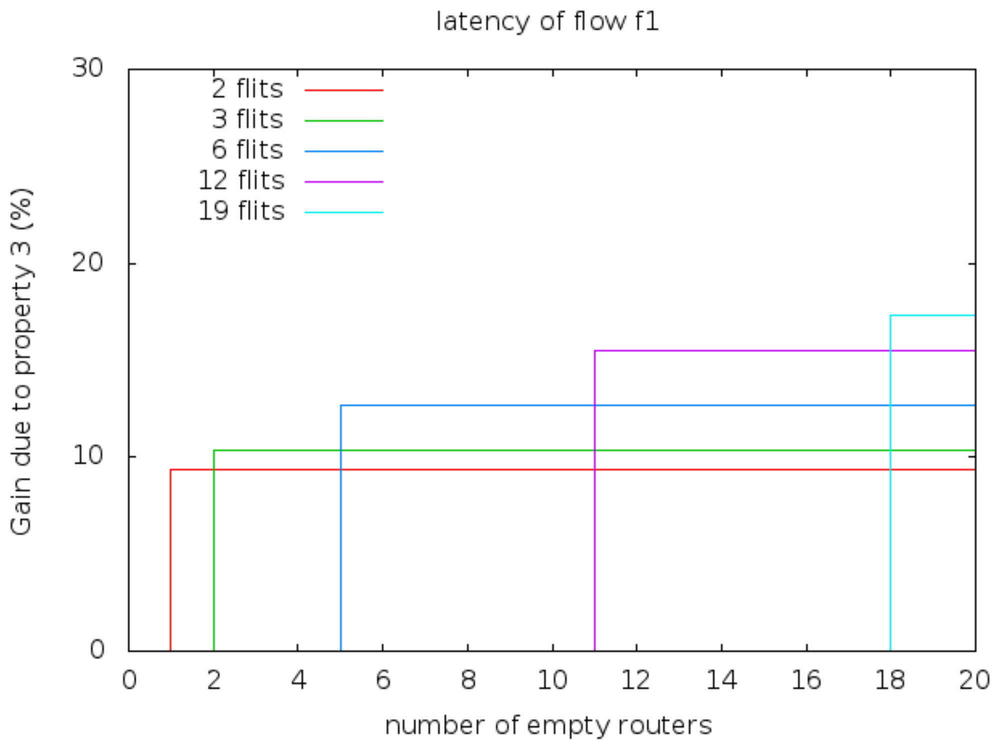
Figure 5.20b shows the gain for various values of E_r and for different length of packets. For a given length of packets, the two constant obtained gains are explained by the fixed 21 hops



(a)



(b)



(c)

Figure 5.20: Mapping of flows in the second configuration and normalized gain compared to the RC method.

that f_5 performs to reach its destination which are accounted in the WCTT computed by the RC method. The existence of the two constants gains comes from whether Property 3 applies or not, i.e. either f_7 is influent or not. This graph does not show only the gain due to Property 3, but it includes the constant gain due to the Property 2. In order to show only the gain due to Property 3, we deduced from the initial gain, the one when the empty routers are equal or less than the number of flits. The graph 3 in the figure 5.20c, presents the gain due only to property 3.

When $E_r < n_{f_i}$ (left part of the curves), f_7 indirectly blocks f_1 and thus influences its WCTT. However, when $E_r \geq n_{f_i}$ the RC method still assumes that f_7 influences f_1 and thus adds it in the computation of the WCTT of f_1. RC_{NoC} instead identifies that f_7 is not influent as Property 3 applies. This difference leads to an increase in the obtained gain and the second plateau in the curves shown in Figure 5.20b and so the curves shown in Figure 5.20c.

Discussion

Considering the pipeline transmission of flits in the WCTT computation reduces the pessimism introduced by the classical RC method. This improvement depends on the type of contentions, the size of the NoC and the size of packets. Property 2 improves WCTT values when the size of NoC is large and the size of packets is small or when the number of hops performed by the flows is greater than the size of packets in flits. Property 3 helps when the size of NoC is sufficient for indirect contentions to occur with a minimum spacing between analyzed flows and indirect ones. Let us stress that while our examples assume that all packets have the same size, the properties we stated do not. Besides, these properties could also be generalized allowing to overcome the assumption **A10**.

5.5 Conclusion

In this chapter, we have shown that the classical RC method leads to pessimistic values for the WCTT of the flows. It is explained by the fact that the pipeline way of transmitting flits in the wormhole networks is not taken into account. The current existing methods consider that an analyzed flow can be blocked by others flows till they have reach their destinations. Besides, it considers all indirect flows as blocking flows. Thus the transmission delays of all blocking flows, i.e. direct and indirect, are added to the transmission delay of the analyzed flow f_a, leading to an over-approximation of the WCTT of f_a. However, we have shown that the pipeline transmission of flits over NoC can be used to reduce the maximal blocking delay of flows, thus tightening computed WCTTs.

In this way, we introduce three properties to either reduce the number of scenarios to be explored when performing a WCTT analysis or to reduce the pessimism when computing WCTTs:

1. **Reducing the number of scenarios:** The first property states the worst-case scenario when computing the maximal blocking delay a flow can suffer when a round-robin arbiter is used.
2. **Computing maximal delays:** The second property states that the maximal blocking delay a flow can suffer is bounded. This eliminates the need to wait till the blocking flow reaches its destination.
3. **Reducing the delays:** A third property allows to check whether an indirect flow influences the transmission delay of an analyzed flow.

These properties are implemented in an algorithm based on a recursive method in order to improve the WCTT, as shown in the evaluation we report. We compare our algorithm noted RC_{NoC} to the RC method by considering a synthetic benchmark. We showed how some parameters affect the gain we obtained related to RC thanks to the use of these properties. The evaluation shows a significant reduction of the WCTT due to the properties 2 and 3, especially with small packets. The reduction of the WCTT is of utmost importance as it permits to flows to respect their constraints. Besides, as shown in Chapter 4, the computation of WCTT has an impact on the behavior of the core-to-I/O flows. The evaluation of RC_{NoC} on the problem stated in Chapter 4, is illustrated in Chapter 7.

Chapter 6

Map_{IO}: an I/O contention-aware mapping technique

Contents

Chapter 4 has shown that even if we reduce the pessimism in the values computed for the

WCTT, the problem of rejecting critical frames remains. A strategy of mapping that reduces the WCTT of the core-to-I/O flows is thus needed. However, as mentioned in Chapter 4, the existing mapping methods consider only core-to-core and core-to-memory communications with the objective to minimize their contentions. Thus, these methods do not reduce the contention on the core-to-I/O flows, leading to drop Ethernet frames. In order to avoid this problem, we propose a new mapping strategy, called Map_{IO}. This strategy takes into account the position of the I/O interfaces and aims to reduce the contention on the path of the core-to-I/O flows.

6.1 How can the WCTT of core-to-I/O flows be reduced?

In Chapter 4, we have shown on the case study A, in section 4.3.2, that existing mapping strategies do not consider where I/O interfaces are located within a NoC architecture when mapping applications. We recall that this case study is made of 1 critical application and 3 non-critical applications. Let us explain how the allocation of a critical application, taking into account the location of the I/O interfaces, may impact the WCTT of the core-to-I/O flows.
When allocating a critical application far from Ethernet, a core-to-I/O flow may therefore suffer from the large payload of the non-critical flows. Besides, the distance, in number of hops, taken by the path of the core-to-I/O flow from DDR to Ethernet increases. Thus, the probability that the core-to-I/O flow is blocked on its path increases. Then, the WCTT of the core-to-I/O flow may increase. However, if a critical application is allocated far away from the DDR, the distance taken by the path of the core-to-I/O flow going from/to DDR to/from the core destination increases. This allocation leads also to increase the distance taken by the path of the core-to-I/O flow coming from Ethernet to the DDR. Therefore, the WCTT of the core-to-I/O flow may increase due to this allocation as the probability that this core-to-I/O flow is on contention on different routers with the flows of non-critical applications increases.
However, we have shown that even we map an application near to both DDR and Ethernet controllers, the internal mapping of the applications may no longer be appropriate to the objective of reducing the contention on the path of the core-to-I/O flow. Therefore, a static mapping strategy of critical and non critical real-time flows that reduces the WCTT of core-

to-I/O communications is needed.

This mapping strategy presents two objectives:

- **Objective 1:** Minimizing the distance separating the placement of critical applications from both of the DDR and Ethernet controllers.

If we note d the total distance, in number of hops, of the path taken by a core to I/O flow. We have $d = d_1 + 2d_2 + d_3$ with d_1 the distance from the Ethernet to the memory controller, d_2 the distance between the DDR controller and the destination core of the core-to-I/O flow, and d_3 the distance of the outgoing I/O flow from the DDR to the Ethernet controller. The first goal is then: $min(d_1 + d_2 + d_3)$ as minimizing the distance of a flow reduces the number of times it can be in contention with other flows. The values of d_1 depends on which Ethernet interface is coming the core-to-I/O flow. We note that it depends on the region where the applications are mapped on the NoC (***A14***). Both d_2 and d_3 depend on where the destination core is mapped on the NoC, as well as on the memory port that is used.

- **Objective 2:** Reducing the contention on the core-to-I/O flow.

Once the distance of a core-to-I/O flow is minimized, the second goal is to minimize the contention that this flow can experience on its path. As the path taken by a core-to-I/O flow is induced by the XY routing, the allocation of specific tasks on and around this path can reduce its contention.

6.2 Overview of our strategy of mapping: Map_{IO}

Figure 6.1 illustrates the different steps of Map_{IO} which allocates the different applications of the case study A presented in Chapter 4. This mapping is divided into two phases following the two objectives introduced previously.

In the first phase illustrated in steps from a to d, we allocate the applications on the different regions on the NoC. The idea is to allocate critical applications in the corners of the NoC where

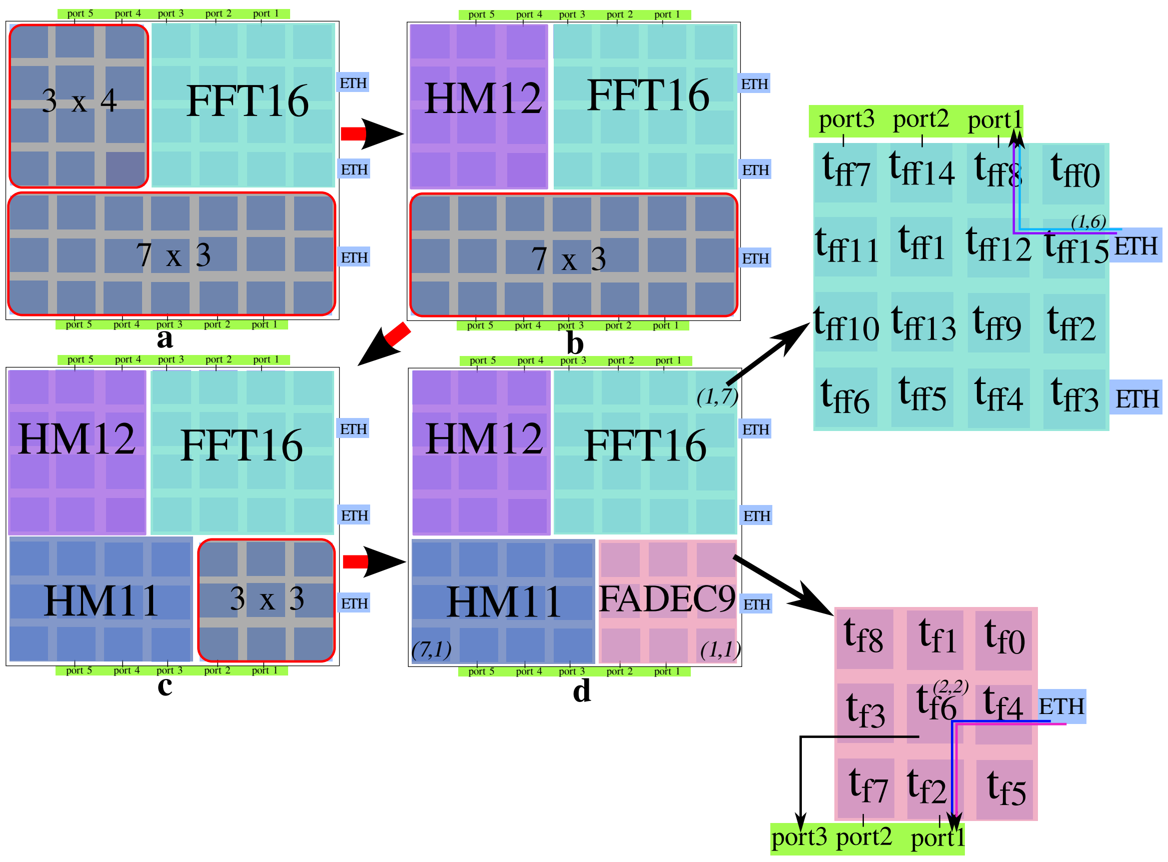


Figure 6.1: The different steps of our approach Map_{IO} for mapping FADEC, HM and FFT applications of case study A.

both Ethernet and DDR controllers are available. This placement reduces the distance d as it especially reduces d_2 and d_3. Besides, it reduces the contention with flows of non-critical applications. However, allocating arbitrarily critical applications at the corner could fragment the remaining regions on the NoC. For this reason, we have to follow a unique direction in the allocation. Thus, we begin from a corner of the NoC on the side of DDR and Ethernet and we follow one circular direction in order to not fragment the regions on the NoC. The applications are ordered in such a way to allocate the critical applications near of both Ethernet and DDR controllers to minimize the distance d. In order to avoid the contention between the applications, we use rectangular shapes when mapping the applications in the regions, due to the XY routing. The shapes are chosen in such a way to correspond to the size of the region where they are allocated.

In the example of Figure 6.1, we begin by the non-critical application FFT_{16} by choosing a shape 4×4, as shown in the step a. On the next region 3×4, we choose a non-critical application whose size is closest to the size of this region, i.e. HM_{12}. Similarly, in the step c, on the region 7×3, HM_{11} is allocated by a rectangular shape 4×3. And finally, the critical application $FADEC_9$ is then allocated near to Ethernet and DDR controllers.

After allocating the applications in the different regions on the NoC, we are interested now in reducing the contention on the core-to-I/O flows. These core-to-I/O flows follow a fixed path (***A13 and A14***). Thus, the second phase consists in allocating the tasks of the applications, placed near to the Ethernet, controllers in such a way to reduce this contention.
As illustrated in the task mapping of FFT application, the task t_{ff15} is allocated to the core located at (1,6). Actually, this task does not send any flow to the other tasks, thus it does not block directly the path of the core-to-I/O flow. Also, $FADEC_9$ present a task t_{f6} which only receives data from the other tasks. Then, if we allocate this task at the core (2,2), this task will not generate a flow to the task located at (2,1), i.e. t_{f_2}, and thus not blocking the Y path of the core-to-I/O flow at the router (2,2).

The following sections illustrates the details of each phase.

6.3 Phase 1: Core-to-I/O flows distance minimization

This section focuses on the first phase of Map_{IO} strategy over a Tilera-like architecture. This first phase allocates in priority critical applications in a dedicated region close to memory and Ethernet controllers. To this end, we have first to order the applications in a list, noted L_{app}, in order to be followed when allocating the applications. This phase is divided into three steps to reach our first objective. The first step consists on filling the applications in the different regions on the NoC. It matches each region to a corresponding application, by following both L_{app} and a circular direction. Then, the second step assigns a rectangular shape to each application. However, when this is not possible we authorize the definition of arbitrary shape, possibly leading to inter-application contention due to the XY routing policy used within NoC. Finally, the third step reclaims cores unused within assigned shapes to the regions where critical applications are allocated in order to be used by the second phase of this strategy. These unused cores can be allocated on the path of the core-to-I/O flow in order to reduce its contention. These steps are described in Algorithm 6, which will be explained progressively in the following sections.

In this section, we consider another case study, noted B, made of two instances of FADEC: $FADEC_8$, $FADEC_9$ and 3 instances of HM: HM_{14}, HM_9, HM_8. This case study is used to explain the different steps of the first phase of Map_{IO}.

6.3.1 Filling NoC regions with applications

We note L_{region} the list of rectangular subsets of the NoC, called *regions*. L_{region} initially contains a single region of size equals to the size of the NoC, i.e. $L \times W$. L_{app} is the list of applications that remain to be mapped. As mentioned in the previous sections, the objective is to allocate critical applications in the corner close to memory and Ethernet controllers. Besides, we have to follow a circular direction in order to avoid the fragmentation of the different regions on the NoC. The organization of the applications within L_{app} is essential to reach this objective.

How the applications in L_{app} are organized?

Non-critical applications are first inserted in L_{app}, while critical applications are then inserted between the head and the tail. All applications are inserted in decreasing order of their size, noted SA_i. SA_i is the number of cores an application uses. Then, the larger a critical application is, the closer to the memory and Ethernet controllers it is located to. Therefore, starting the allocation from a corner on the NoC (side of Ethernet interfaces) and following both L_{app} and the circular direction on the NoC, ensure the placement of the critical applications near both DDR and Ethernet controllers.

However, another objective of our mapping is to reclaim the unused cores from regions assigned to non-critical applications in order to give them to the regions that will later on be assigned to critical applications. We note that UNused cores value (UN) corresponds to the number of cores remaining free if we allocate all the applications on the NoC and thus it is equal to: $UN = (L \times W) - \sum_{i}^{n} SA_i$. These unused cores will be useful to reach our "objective 2": reducing the contention on the path of the core-to-I/O flow, and so, for the second phase. However, this can not be done if we do not choose correctly which application should be placed in head or in tail of the list L_{app}. For example, in the case of one critical application and $UN \geq 0$, when inserting the critical application at the head and beginning by mapping this application, it avoids to give the unused cores to the critical application. Actually, if we choose to reclaim the unused cores at the beginning, we can not ensure a regular shape for the remaining applications. For this reason, we choose to allocate the critical application at the end, and insert it at the tail, and in this way, when we can reclaim unused cores from non-critical applications, it could be used within the region of the critical application.

How critical applications are inserted between the head and the tail of L_{app} in order to reclaim unused cores?

How critical applications are allocated between the head and the tail of L_{app} depends on the value of UN. If $UN = 0$, the critical applications are thus inserted at the head of L_{app} except one which is inserted at the tail when there are more than one critical application, since no free cores can be reclaimed. Thus, in this case, the critical applications are allocated first which ensures rectangular shapes to these applications. If however $UN \geq 0$, then it is the opposite: critical applications are inserted at the tail except one at the head. In this case, critical applications are allocated at the end while retrying to reclaim unused nodes from the non-critical applications. In both cases, we ensure that at least two critical applications are allocated in the two corners near to DDR and Ethernet interfaces.

How to fill the applications into the regions?

Once L_{app} is built, we can thus fill the applications into the different regions on the NoC. We start to fill regions from one corner of the NoC. We then proceed following a clockwise or counterclockwise direction, depending on whether the initial corner is located respectively on the bottom or the top of the NoC. The function $Search\_App\_to\_Map()$ (line 9 of Algorithm 6) searches for the corresponding application to allocate to the current region.

Algorithm 6 $Map\_IO(L_{app})$

```
1: $Region = L_{region}.head()$
2: while $(Region \neq NULL)$ do
3:     $App_i = head(L_{app})$
   #There is only one application in $L_{app}$ which is allocated on the remaining regions on the NoC
4:     if $(L_{app}.Size == 1)$ then
5:         $App_i.Accepted\_Shape.Insert(L_{region})$
6:         $L_{app}.Delete(App)$
7:         $L_{region} = NULL$
8:     end if
   #$L_{app}$ contains more than one application
   #Step 1: Search the corresponding application to the current region on the NoC
9:     $App_i = Search\_App\_to\_Map(L_{app})$
   #There is no application that has a size less than the size of the current region, then aggregate regions
10:    if $(App_i = NULL)$ then
11:        $Aggregate\_Regions(L_{region})$
   #An application is found to be allocated in the current region
12:    else
   #Step 2: Choose the corresponding rectangular shape for the selected $App_i$
13:        $min\_xy = min\_X\_Y(App_i, L_{app})$
14:        $App_i.Selected\_shape = Choose\_Shape(min\_xy, Region, App_i)$
   #Step 3: Reclaim unused cores within the selected shape
15:        $Map\_Update(Region, L_{region}, App_i, min\_xy, min\_app\_size, UN)$
   #The application $App_i$ is allocated
16:        $App_i.Remove(L_{app})$
17:        $Region.Remove(L_{region})$
18:    end if
   #Update $L_{app}$ and $L_{region}$ depending on the $UN$ value
19:    if $((UN == 0)\&\&(Initial\_UN! = 0))$ then
20:        $L_{app}.Inverse()$
21:        $L_{region}.Inverse()$
22:    end if
23: end while
```

The current head of L_{app} defines the criticality level that is considered when performing this lookup process to select an application. The application verifying this criticality level and has the highest SA_i value that is equal or less than the size of the considered region is selected to fill this region. Whenever a part of a region is filled by an application, after performing the second and third steps (lines 13 to 18 which are explained in the next sections), we say that this application is mapped and it is removed from L_{app}. L_{region} is thus updated with the remaining rectangular regions of the NoC. If the size of the considered region is strictly inferior to the size of any application (lines 10 to 12), the head of L_{region} and the next region are associated by the function $Aggregate\_Regions()$. The size of this aggregated region is computed and the lookup process is repeated until a region of sufficient size is build. Nevertheless, when it remains only one application to allocate in L_{app} (lines 4 to 8), this application will be assigned to the remaining regions in L_{region}.

Finally, as rectangular shapes are used to avoid inter-application contention, an application may use additional unused cores ($UN \geq 0$) to define such a rectangular area. Thus, the value of UN dynamically changes when mapping an application. However, if these cores are used by non-critical applications and so UN moves to 0, they can prevent the mapping of critical applications. To avoid this situation when $UN = 0$ (lines 19 to 22), we reverse the lists L_{region} and L_{app} and therefore continue from the other corner. We also inverse the direction followed to fill regions.

Example. In our new case study B, we have $UN = 1$ and we have more than one critical application. $FADEC_9$ is thus inserted at the head of L_{app} while $FADEC_8$ is inserted at the tail. Therefore, L_{app} contains $\{FADEC_9, HM_{14}, HM_9, HM_8, FADEC_8\}$. However, in the case study A illustrated in Figure 6.1, there is one critical application $FADEC_9$ which is inserted at the tail of L_{app} as $UN = 1$. Now, let us see in the next section in which shapes these applications are allocated on the NoC.

6.3.2 Application shape and its assignment to regions

Filling a part of a region with an application requires to define the actual rectangular shape taken by this application. This choice of the rectangular is not arbitrary as we will see in this second step of the first phase of Map_{IO}.

Each application App_i is described by its size SA_i, assuming each core is dedicated to a task. For each shape, we note UN_i the number of UNused cores by App_i and we therefore have $UN_i = (X_i \times Y_i) - SA_i$. We arbitrarily compute three possible rectangular $X_i \times Y_i$ shapes of minimal UN_i values and whose sizes are equal or higher than SA_i. These computed shapes are as close as possible to squares to avoid inter-application contentions. We thus forbid X_i and its Y_i to be equal 1 if $SA_i > 3$.

How to choose the appropriate rectangular shape for each application? When an application is mapped on the NoC, its most appropriate shape is selected. However, this selection depends on the possible shapes of other applications to avoid generating a remaining region unable to map any other application. The goal is to optimize the shape of these newly created regions so that next applications can be mapped without being fragmented, i.e. assigned on two regions. To this end, the minimum value of the X_i and Y_i of all applications, noted by min_{xy} is computed by the function $min\_X\_Y()$ (line 13). min_{xy} represents the minimum size for either the row or the column of an application. Thus, the function $Choose\_Shape()$ (line 14) includes the following constraints on min_{xy} and therefore selects the shape that avoids an application to be fragmented on several regions.

$$((L - X_i \geq min_{xy}) \vee (L - X_i = 0)) \wedge \tag{6.1}$$

$$((W - Y_i \geq min_{xy}) \vee (W - Y_i = 0)) \tag{6.2}$$

Example. Figure 6.2 illustrates the different steps of the application mapping for this case study. Table 6.1 shows the computed rectangular shapes for each application of the case study B as well as the value of UN_i.

$FADEC_9$		HM_{14}		HM_9		HM_8		$FADEC_8$	
$X \times Y$	UN	$X \times Y$	UN	$X \times Y$	UN	$X \times Y$	UN	$X \times Y$	UN
3×3	0	2×7	0	3×3	0	2×4	0	2×4	0
2×5	1	3×5	1	5×2	1	3×3	1	3×3	1
3×4	3	4×4	2	4×3	3	2×5	2	2×5	2

Table 6.1: Different shapes and their free cores for each application.

The application at the head of L_{app} is a critical application. We thus have to consider only critical applications. The first matching critical application which has the highest SA_i value

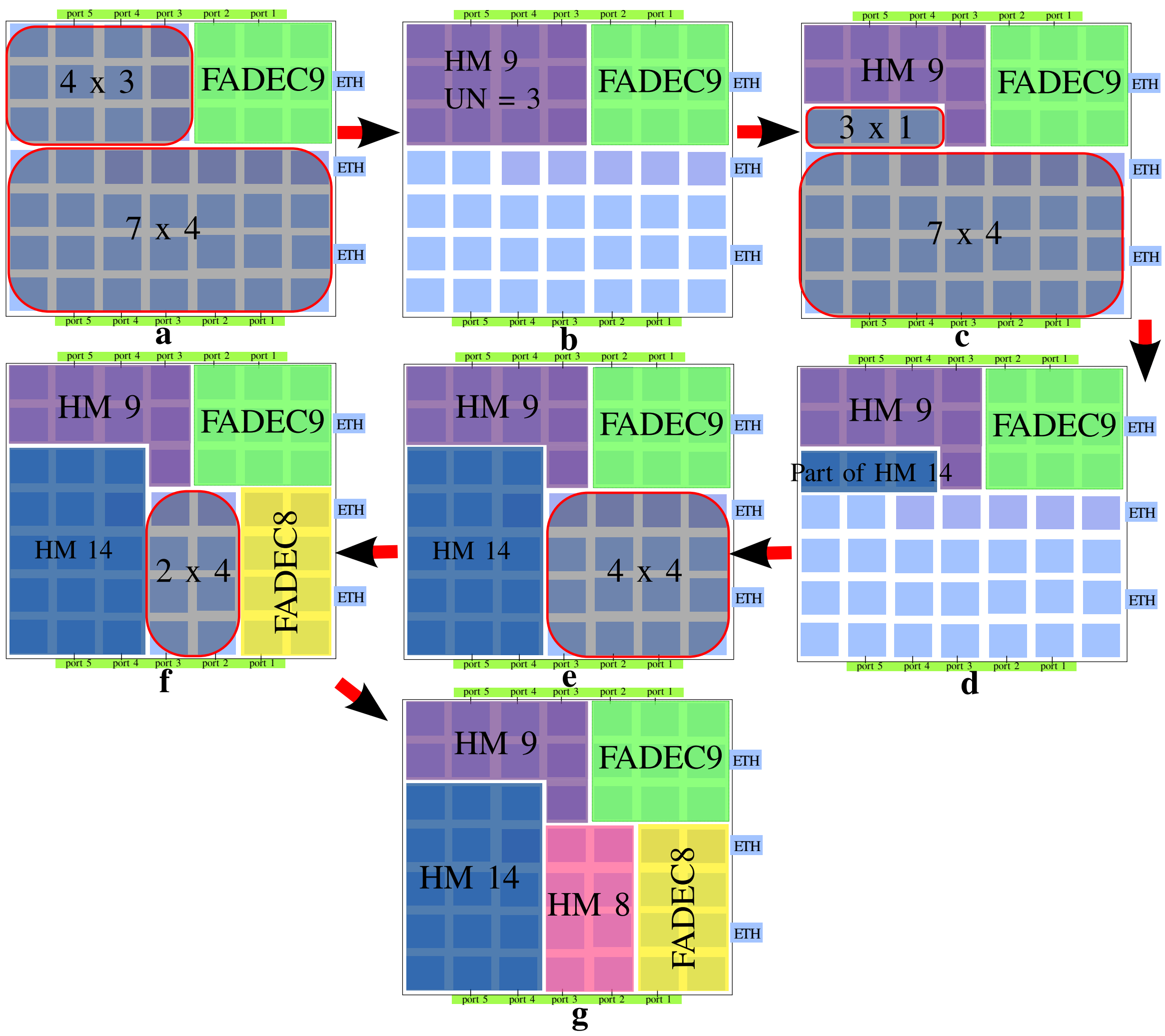


Figure 6.2: Mapping of FADEC and HM applications of the case study B using our approach Map_{IO}.

that is equal or less than the size of the NoC is $FADEC_9$ ($SA_{FADEC_9} = 9$). To select the shape of $FADEC_9$, we compute min_{xy} which is equal to 2. Since the first shape 3×3 for $FADEC_9$ keeps free more than 2 columns and 2 rows, then this shape is selected. We start by filling the NoC from its top right corner and therefore map $FADEC_9$ on this corner, as shown in Figure 6.2a. We then follow a counterclockwise direction as explained in the step 1, as shown by the other steps of Figure 6.2. The next region that is considered is of size 4×3. HM_{14} is now at the head of the list L_{app}, thus a non-critical application should be selected. $SA_{HM_{14}}$ is greater than the size of this next region, then HM_{14} does not correspond to this region. Therefore, HM_9 is selected to fill this region as its SA_i value is less than the size of this region. At this step, min_{xy} is still equal to 2. The first shape that is considered for HM_9 is

3×3. However, it does not keep free min_{xy}, i.e. 2 columns, in order to be able to map a next application with a rectangular shape, i.e. it does not verify the equation 6.1 ($(7-3)-3 \ngeq 2$). In fact, the remaining region 1×3 has a size that is less than the minimum size of the remaining applications. Then, it is impossible to allocate an application within this region without being fragmented. On the other hand, the shape 2×5 does not fit in the selected region 4×3. Thus, the selected shape is thus 4×3 which verifies the equation 6.1, as show in Figure 6.2b. However, this shape has a size greater than the size of the application, and thus, it generates an number of unused cores. Let us see, in the next section, the next step (step 3) to be performed when selecting a shape having $UN_i > 0$.

6.3.3 Reclaiming unused cores within shapes

A selected shape can generate a number of unused cores UN_i which indicates whether these cores could be reclaimed or not. The function $Map\_update()$ of the third step (line 15) determines which case is applied for the selected shape in the second step. Algorithm 7 describes this function. There are two cases:

- **Case 1:** The unused cores are eliminated from the shape of an application when $UN_i > UN$ or $UN_i \geq min_{xy}$.

When $UN_i \geq min_{xy}$, these unused cores could in fact form a contiguous region with the one of the critical application. Thus, we reclaim these cores to allocate them to a critical application. On the other hand, when $UN_i > UN$, these unused cores must be eliminated in order to allocate all the applications on the NoC. The shape must be modified by eliminating a number of unused cores from either the column or the row. This choice depends on whether the remaining region, after the removal of cores from the column, corresponds to the size of one existing application. For this reason, we compute first the remaining region when eliminating the unused cores from the column (lines 2 and 3). The size of the remaining region is thus $col\_remaining\_region.col \times col\_remaining\_region.row$. If this computed region could correspond to the size of an application having the same level of criticality, then the unused cores are eliminated from the columns (lines 4 to 7). The new shape, noted *Accepted_Shape*, is recomputed using the function *Compute_Shape_col*. Otherwise, we remove these cores from the row and the new shape is recomputed using the function *Compute_Shape_row* (lines 9 to 12). This choice is in fact due to the clockwise or counterclockwise direction chosen to progress between the regions. This direction increases the probability to have the maximum number of cores presented next to the remaining region when eliminating the cores from row. Thus, it avoids to allocate the remaining applications into fragmented regions.

- **Case 2:** The unused cores cannot be eliminated from the shape of an application when $UN_i \leq UN$ and $UN_i < min_{xy}$.

When $UN_i < min_{xy}$, there are no applications that can benefit from these unused cores as they cannot form a contiguous region. Then, there is no need to eliminate these unused cores. However, we have to verify that $UN_i \leq UN$ in order to guarantee a sufficient number of remaining cores to allocate all applications. Then, the definitive shape is the one that was selected in the previous part, i.e. the second step (line 14).
Finally, L_{region} is updated by computing the remaining regions (line 16).

Example. In our example, the shape selected for HM_9, i.e. 4×3, presents 3 unused cores. This number exceeds the allowed number of unused cores which is $UN = 1$. We therefore have to removed these 3 unused cores from the shape. This removal is done from the row. Actually, the elimination from the column generates a region of size 1×3, which does not corresponds to the size of any remaining application. Figure 6.2b and 6.2c present these steps to select the definitive shape for HM_9 which is $4 \times 2 + 1 \times 1$. L_{region} therefore contains two remaining regions which are $\{< 3 \times 1 >; < 7 \times 4 >\}$. These two remaining regions are then aggregated to cope with the size of the remaining non-critical application HM_{14}. The region 3×1 handles a part of HM_{14}. The size of the remaining part of HM_{14} is equal to 11 which fits in the region 7×4. A new computation of candidate shapes is done at this step by considering the size 11, in the same way as described in section 6.3.2. The selected shape is 3×4 since it verifies the equations 6.1 and 6.2. These steps are shown by the Figures 6.2d and 6.2e. This last shape does not need to be redefined as $UN_i < min_{xy}$. The definitive shape of HM_{14} is an aggregated shape of $< 3 \times 1 >$ and $< 3 \times 4 >$. L_{region} contains a single region which is $\{< 4 \times 4 >\}$ and we now have $UN = 0$. We therefore have to reverse both L_{region} and L_{app}, leading to $L_{app} = \{FADEC_8, HM_8\}$. We then continue from the bottom right corner of the NoC and follow a clockwise direction. This ensures to map the critical $FADEC_8$ application near to both the Ethernet and DDR controllers. Its first candidate shape 2×4 is the definitive shape as it verifies the equations 6.1 and 6.2. Besides, it does not generate any unused cores. Finally, the remaining region of 2×4 is left to the remaining application HM_8. These steps are illustrated on the part f and g of Figure 6.2.

Algorithm 7 $Map\_Update(Region, L_{region}, App_i, min\_xy, min\_app\_size, UN)$

```
    #Case 1: $UN_i$ can be reclaimed
 1: if $((App.Selected\_shape.UN_i \geq min_{xy}) \| (App.Selected\_shape.UN_i > UN))$ then
    #Compute the remaining regions if we eliminate $UN_i$ from the column of the selected shape
 2:     $col\_remaining\_region.row = region.row - shape.row + 1$
 3:     $col\_reamining\_region.col = App.shape.UN$
    #Eliminate $UN_i$ from the column of the selected shape
 4:     if $(col\_reamining\_region.Size \geq min\_app\_size)$ then
 5:         $Eliminate\_unused\_cores\_col(region, App, L_{region})$
 6:         $App.Accepted\_Shape = Compute\_Shape\_Col(App)$
 7:         $App.Selected\_shape.UN_i = 0$
    #Eliminate $UN_i$ from the row of the selected shape
 8:     else
 9:         $Eliminate\_unused\_cores\_row(region, App, L_{region})$
10:         $App.Accepted\_Shape = Compute\_Shape\_row(App)$
11:         $App.Selected\_shape.UN_i = 0$
12:     end if
13: else
    #Case 2: $UN_i$ cannot be reclaimed
14:     $App.Accepted\_shape.Insert(App.shape)$
15: end if
    #Update $L_{region}$
16: $Compute\_remaining\_regions(region, App, L_{region})$
```

6.4 Phase 2: Core-to-I/O flows contention minimization

The second phase of Map_{IO} maps the tasks of each application within its assigned subset of the Tilera-like NoC. It reduces the WCTT of the core-to-I/O flows by applying specific rules for mapping tasks on and around the paths taken by these flows. These rules favor the perpendicular communications to the core-to-I/O flow as the XY routing is used. In this section, we present these rules where first we begin the allocation on the path of the core-to-I/O flow, noted by the critical path. Then, we allocate the tasks around this path and finally, we apply some rules to reduce the contention on the I/O outgoing flow, i.e. from the DDR to the Ethernet interface. We note that the following rules are described by considering the core-to-I/O flows going to the DDR memory located south of the NoC. However, to apply these rules on the core-to-I/O flows using the DDR memory located north of the NoC, we have just to inverse the Y axis.
On the other hand, we apply the SHiC method (described in Chapter 3 in section 3.2.2) to allocate the tasks within the applications mapped far away from Ethernet interfaces and which region is not crossed by core-to-I/Oflows.
Before explaining the rules used in the second phase, let us first describe how to determine the critical path.

6.4.1 I/O critical paths within applications

Let us assume an application App_i whose bottom right edge is located at (1,1) and the associated region, obtained from the first phase, is of size $m \times n$. A core-to-I/O flow, traversing its region, follows a path from $(0, y_i)$, where is located the used Ethernet controller, to $(x_i, 0)$, where is located the used DDR port. This path may thus be located entirely or partially within the region of App_i depending on the values of m and n. The part of the path located within this region is called the *critical path* of the core-to-I/O flow. When $x_i < m$ and $y_i < n$, the path from $(0, y_i)$ to $(x_i, 0)$ belongs to App_i as shown by Figure 6.3a. It is therefore the critical path of this flow. In the other cases, only either a subset of the X part ($x_i > m$ and $y_i < n$) or the

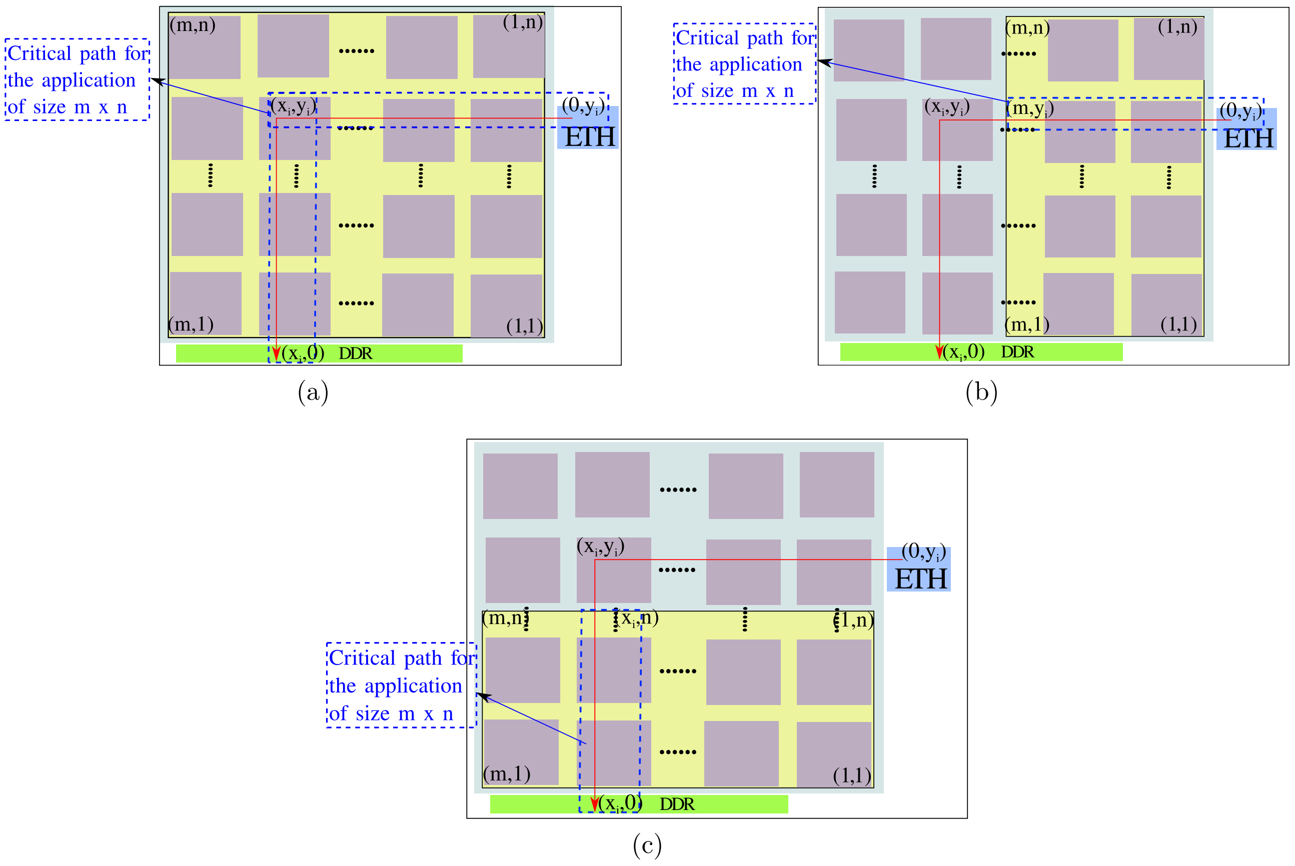


Figure 6.3: Different cases of a possible critical path, shown in blue, for a core-to-I/O flow (path shown in red) of an application (whose mapping is shown in yellow).

Y part ($x_i < m$ and $y_i > n$) of the path of the core-to-I/O flow is located within the region of App_i. These other cases are illustrated by the Figure 6.3b and 6.3c. In these latter cases, the remaining parts of the critical path belongs to another application, and so the same principle will be applied for this application. Note also that an application can thus be crossed by several core-to-I/O flows and can therefore have several critical paths associated to it.

Example. Let us focus, in the case study B illustrated in Figure 6.2, on the Ethernet controllers located at:

- $(0, 2)$ which is shared between $FADEC_8$ and HM_8,
- $(0, 4)$ which is used by HM_{14}.

Let us first consider the mapping of $FADEC_8$ whose bottom right edge is located at (1,1) and

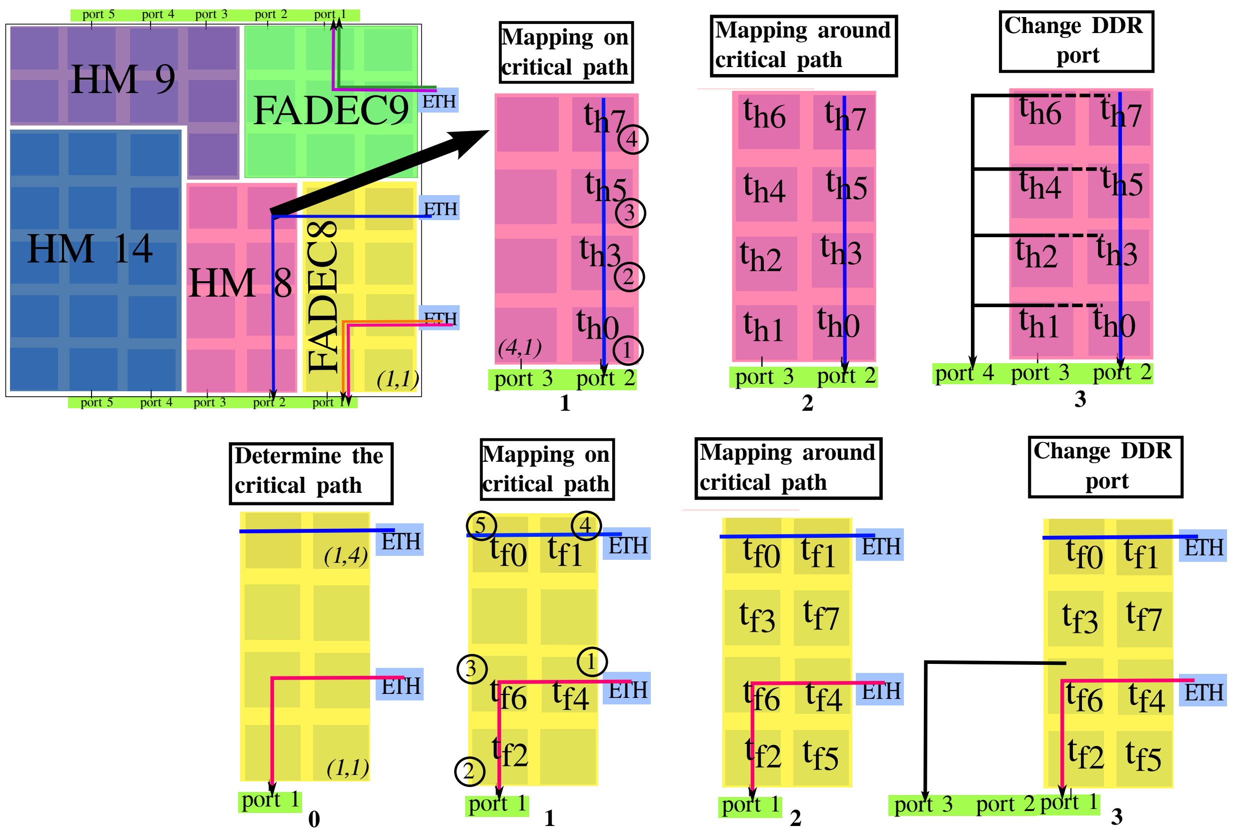


Figure 6.4: The different steps of internal mapping for $FADEC_8$ and HM_8.

the size of its shape is 2×4. There are two critical paths, as shown by the step 0 at bottom of the Figure 6.4. The first one is its core-to-I/O flow or the core-to-I/O flow of HM_8 (as they follow the same path), which comes from the Ethernet controller located at $(0, 2)$ and goes to the DDR port located at $(2, 0)$. The second critical path is a subset of the core-to-I/O flow of HM_8, which comes from $(0, 4)$ and goes to $(3, 0)$. This critical path is defined as from $(0, 4)$ to $(2, 4)$. The remaining part, from $(3, 4)$ to $(3, 0)$, is the critical path of HM_8.

Impact of the critical path on the WCTT

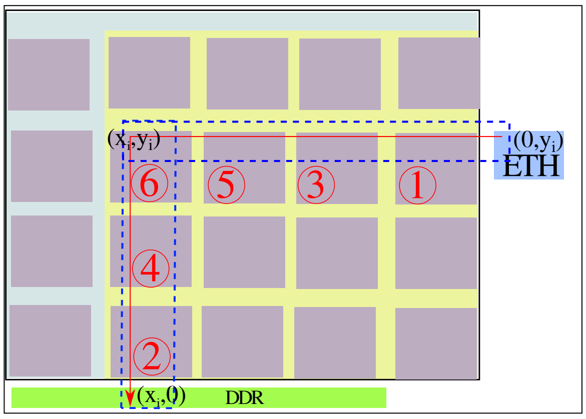
The critical path is the path taken by the core-to-I/O flow going from Ethernet to DDR controller. A strategy to avoid the initial problem of dropping Ethernet frames, stated in chapter 4, is to reduce the WCTT of the core-to-I/O flows. A solution consists of reducing the contention experienced by the core-to-I/O flows. Thus, it is important to allocate the tasks in such a way that they do not generate flows in direct contention with the core-to-I/O flow. For this reason, we have to divide the solution into two parts. First, we have to choose the corresponding tasks to be allocated on the critical path. Second, as these tasks on the critical path could generate different communications with other tasks, thus it is important to choose also the placement of these remaining tasks. Let us see how to allocate the tasks on and around the critical path.

6.4.2 Mapping tasks on the critical paths

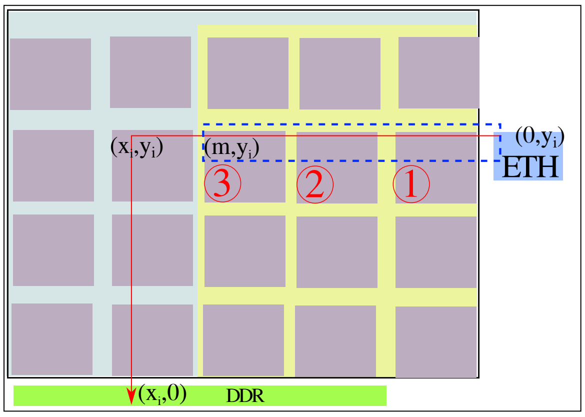
Let us consider an application App_i with several critical paths. We sort the critical paths by increasing length. Let us consider a critical path from $(0, y_i)$ to $(x_i, 0)$. The contention level on a critical path can be reduced by limiting the number of flows in direct contention with the corresponding core-to-I/O flow. Note that the minimum value is simply obtained by not mapping tasks on the critical path due to the XY routing policy. We thus alternatively select a core from the row y_i and from the column x_i in order to map tasks. Figure 6.5 illustrates the order to follow in the different cases of the critical path of App_i. For the row y_i and the column x_i, we start respectively from the cores located at $(1, y_i)$ and $(x_i, 1)$ and end up when the limit of the critical path within App_i is reached. When the end of the row or the column is reached, and $x_i \neq y_i$, we continue in sequence with the cores on the remaining direction of the critical path as illustrated in Figure 6.5a. This alternation returns to the fact that this order of cores present the most blocking effect on the path of core-to-I/O flow. To explain this alternation, we have to show first how much it is interesting to leave the cores unused by following this order to reduce the contention when $UN_i \neq 0$.

Specific mapping rules when unused cores exist

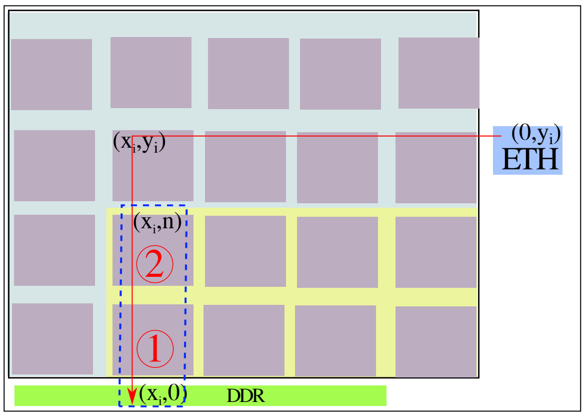
When $UN_i > 0$ for an application App_i, leaving cores on one of its critical path unused, i.e. without tasks being mapped on them, reduces the contention the core-to-I/O flow experiences. Let us first focus on the X part of a critical path. The first reason is that the number of flows in direct contention with the core-to-I/O flow is indeed reduced. This is illustrated by Figures 6.6a and 6.6b that shows that if we do not map a task on $(1, y_i)$, the number of Egress communications (EC) blocking the critical path is reduced. This example can be easily adapted



(a)



(b)



(c)

Figure 6.5: The order of mapping the tasks by considering the different cases of a critical path.

for the Y part of a critical path.

When allocating a task on $(1, y_i)$ but keeping unused the core $(2, y_i)$, all flows sent from this task to any task on the NoC, except those on its first column, block directly the critical path. They indeed share the same links. We therefore define a mapping rule for leaving on the row y_i cores unused: the closest one to $(0, y_i)$ are leaved unused if possible ($U_i > 0$). This task mapping rule also impacts the outgoing core-to-I/O flows, i.e. from DDR to Ethernet controllers. These flows are indeed also directly blocked by flows received from the tasks located on the first row till y_{i-1}. The number of blocking flows is however reduced by simply inversing the mapping, as shown by Figure 6.6b. The task on $(2, y_i)$ can indeed: 1) send to the columns 1 and 2 and 2) receives from any task, except those located on the first row with $x > 2$, without blocking the outgoing core-to-I/O flow in both cases.

A similar strategy applies for the Y part of the critical path. Figure 6.6c and 6.6d justifies the

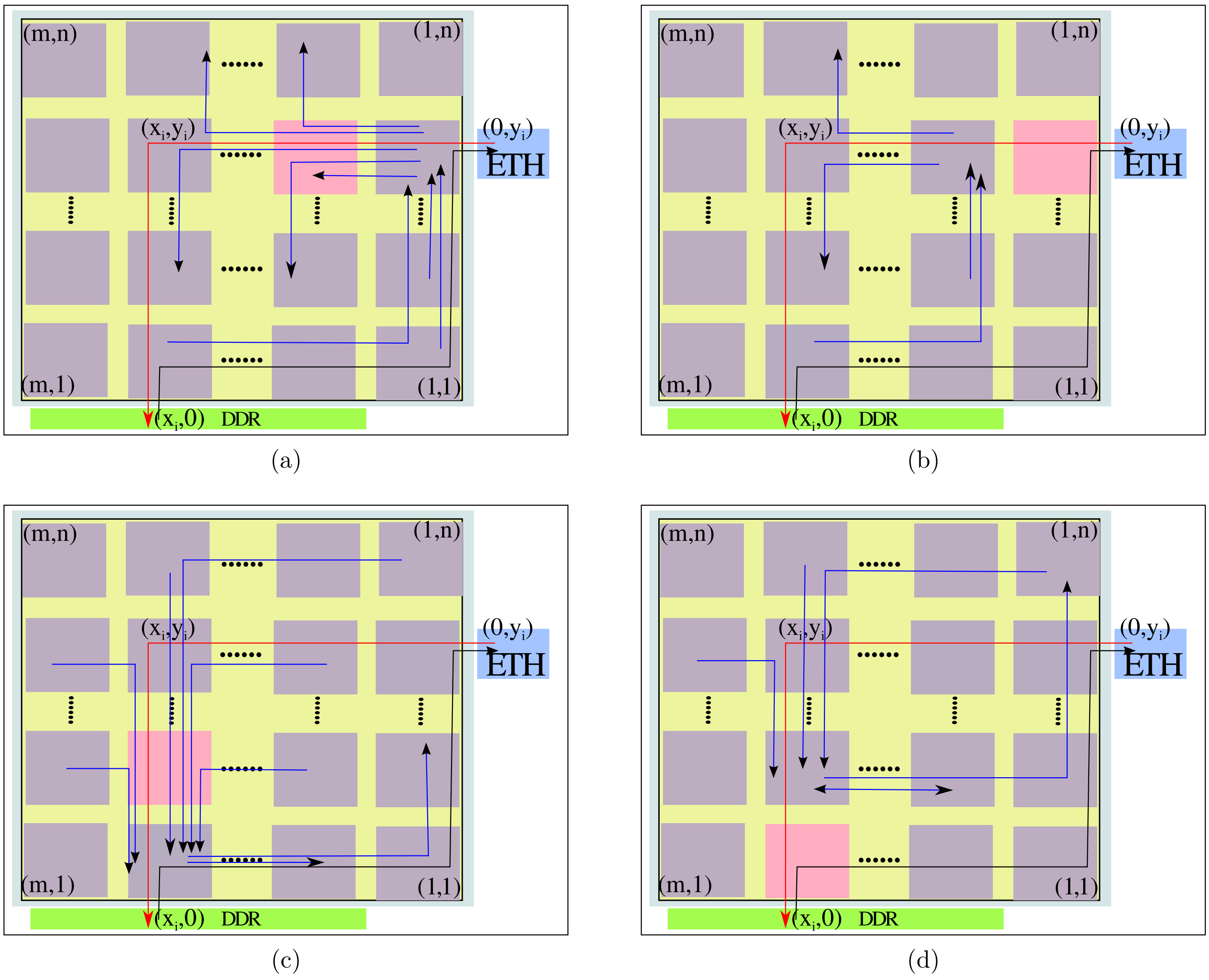


Figure 6.6: The different communications that could block the X and Y part of a critical path depending on where free cores are localized.

order where to choose the unused cores on the column of the critical path by reasoning now on the flows received by the tasks. Thus, when we allocate a task on $(x_i, 1)$, all flows received from the tasks allocated on the NoC, except the first row, block the critical path, even in its Y part or in its XY part. Furthermore, the flows sent from this task to all columns having $x < x_i$ block the outgoing I/O flow. However, these number of blocking flows is reduced when we allocate a task on $(x_i, 2)$ and we keep free the core $(x_i, 1)$ as illustrated in Figure 6.6d.

This explains the order to follow when allocating the tasks on the cores on the critical path. This is the same order when we choose to keep a core unused on this path. Note that the number of cores of the critical path (both the X and Y parts) with no tasks mapped on them depends on the value of UN_i. If UN_i is greater than the number of cores on the critical path, then all cores on the critical path are unused. Besides, we keep unused the cores in the region where $x < x_i$ and $y < y_i$ as this region could affect directly the path of the I/O outgoing flow. Let us see now, which tasks are chosen to be allocated on the critical path.

Mapping rules when no unused cores exist

When $UN_i = 0$, all the cores on the critical path should be allocated by following the order illustrated in Figure 6.5a. Once a core is selected, we distinguish whether it is located in the X part or the Y part of the critical path when we apply the mapping rules.

Allocation on the X part. On the X part of the critical path, we allocate tasks from $(1, y_i)$ to $(x_i - 1, y_i)$. We then map a task on (j, y_i) if it does not directly block the core-to-I/O flow, i.e. follows the following set of constraints:

- when $j = 1$, the task $(1, y_i)$ must not send data to DDR controllers: A task on $(1, y_i)$ that communicates with DDR, generates a flow in contention with the X part of the critical. This condition is specific to Tilera architecture as the first column of the NoC is not connected directly to the first DDR port.
- must not send any data to the tasks allocated on (x_i, k), with $k \in [1, j-1]$: A task sending data to (x_i, k) generates a flow that is in direct contention with the X and Y path of the core-to-I/O flow.
- must not receive any data from the tasks allocated on $(j-1, y_i)$ (when $j \in [2, x_i]$): A task on $(2, y_i)$ sending data to $(3, y_i)$ generates a flow in direct contention with the X path of the core-to-I/O flow. For this reason, the task that should be allocated on $(3, y_i)$ must not receive data from the tasks allocated on $(2, y_i)$ and $(1, y_i)$.

We order the list of tasks that fulfill these constraints in an increasing number of EC. Let us suppose that the task of highest EC is allocated on $(1, y_i)$. The probability that it sends data to the tasks allocated on other columns than the one to which it belongs increases. Only the flows sent to the tasks belonging to its column, i.e. the first one, do not block the X part of the critical path. For this reason, we have to allocate on $(1, y_i)$ the task that verifies these rules and has the minimum number of EC. However, a task on $(2, y_i)$ could sent to two columns, i.e $x = 1$ and $x = 2$ without blocking this X path. Thus, this constraint decreases while increasing x_i.

Allocation on the Y part. On the Y part of the critical flow, similar rules can be defined when we allocate tasks from $(x_i, 1)$ to $(x_i, y_i - 1)$. Thus, a task is allocated to (x_i, j) must verify the following rules:

- must not send data to the tasks allocated on (x_i, k), with $k \in [1, j-1]$ to avoid the Y contention with the critical path;
- must not receive data from tasks allocated on (l, y_i), with $l \in [1, j]$ to avoid the contention on the XY part of the critical path.

The list of tasks that fulfill these constraints is ordered in an increasing number of Ingress Communication (IC). Let us suppose that the task of highest number of IC is allocated on $(x_i, 1)$. The probability that it receives data from the tasks allocated on other rows than the one to which it belongs increases and so could block directly the Y part of the critical path. Therefore, we have to allocate on $(x_i, 1)$ the task that verifies these rules and has the minimum number of IC, as it can only receives from tasks belonging to its row without blocking the critical path.

Finally, note that if other critical paths exist within an application, coming from other Ethernet interfaces $(0, y_m)$, we must take into consideration all the critical paths. Thus, the tasks for example on the cores from $(1, y_m)$ to (x_i, y_m), with $y_m > y_i$, must not send to the tasks allocated on cores from $(x_i, 1)$ to $(x_i, y_i - 1)$. This avoids the contention with the Y path of the core-to-I/O flow.

Allocation on the XY part. The tasks mapped on (x_i, y_i) must not receive data from tasks mapped on their rows and not send data to those on their columns to avoid the XY contention with the critical path.

Example. The two steps 1 of Figure 6.4 show for both HM_8 and $FADEC_8$ the task mapping on their critical paths. We start by the shortest critical path in $FADEC_8$, i.e. the one of $FADEC_8$ or HM_8 that comes from $(0, 2)$. We first have to select a task to be mapped on $(1, 2)$. The task must have the minimum EC value but also it must not send data to DDR. t_{f6} is the task of minimum EC value, however it sends data to the DDR. We therefore have to choose another task. As the remaining tasks have the same characteristics, t_{f4} is arbitrarily chosen. We must map on $(2, 1)$ the task with the minimum IC value and that not receive data from t_{f4}. The matching task is t_{f2}. Finally, t_{f6} is mapped on $(2, 2)$ as it is the sole task that do not send to t_{f2}. Let us now consider the second critical path from $(0, 4)$ to $(2, 4)$, i.e. the one of HM_8. The task on $(1, 4)$ must have the minimum EC value and must not send data to the tasks on the column of the first critical path. Moreover, the task on $(2, 4)$ must also not receive from the one allocated on $(1, 4)$. Arbitrary tasks are however chosen since all the tasks have the same characteristics. The mapping on the critical path in HM_8 is shown by the upper part of Figure 6.4. t_{h0} is mapped on $(3, 1)$ as it is the task having the minimum IC value. On the cores above, we have to ensure to allocate the tasks that not send data to the allocated tasks below.

6.4.3 Mapping tasks around the critical path

Once the task mapping is performed on the critical path, we have to map the remaining tasks around the critical path. Actually, the objective is that the remaining tasks, i.e. are not yet allocated, communicating with the tasks allocated on the critical path, do not generate flows in direct contention with the core-to-I/O flow. To reduce the number of contentions in the X or Y part of the critical paths, flows from these tasks should cross the critical path perpendicularly. For this reason, we consider different regions around the critical path, as shown in Figure 6.7, where in each region, we allocate the tasks that generate perpendicular flows with the critical path.

For each configuration of a critical path, Figure 6.8 illustrates the different areas that must then be considered. In the general case (Figure 6.8a), i.e. the entire critical path belongs to the

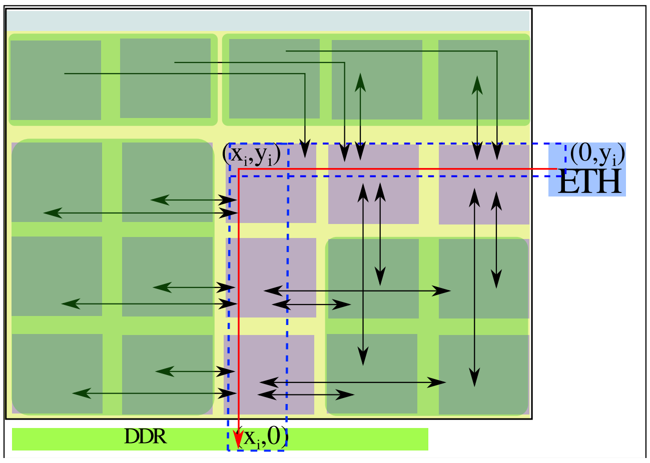


Figure 6.7: Perpendicular communications with the critical path avoids the contention with the core-to-I/O flow.

current application, 4 areas must be considered: I) $x < x_i$ and $y < y_i$, II) $x > x_i$ and $y \leq y_i$, III) $x < x_i$ and $y > y_i$ and IV) $x > x_i$ and $y > y_i$. However, the number of regions can be reduced to two (Figures 6.8b and 6.8c). Note that if several critical paths exist, the one with the highest values for both x_i and y_i is used to define the areas to consider.

Mapping in region I ($x < x_i$ and $y < y_i$)

Let us first focus on the case of Figure 6.8a where the cores in this region share the same rows and columns with the critical path. We have to take into consideration the characteristics of the tasks allocated on the critical path. We recall that the tasks allocated on the row y_i present a minimum EC, while those on the column x_i have a minimum IC. Therefore, each task that might be mapped on (x, y) in this region, should send data to the tasks on its row and belonging to the critical path, i.e. on (x_i, y). These tasks must also receive data from the tasks on their column and belonging to the critical path, i.e. on (x, y_i). However, these tasks should not send to all tasks mapped on the rows of critical path, starting from $(x_i, 1)$ till $(x_i, y - 1)$. This last condition avoid a Y contention with the critical path. In the other cases, i.e. shown by Figures 6.8b and 6.8c, one of these conditions is applied depending on the part of critical path that belongs to the current application. Note that if we can not find any tasks verifying at least one condition, for the moment we allocate unused cores and then we apply the rules to

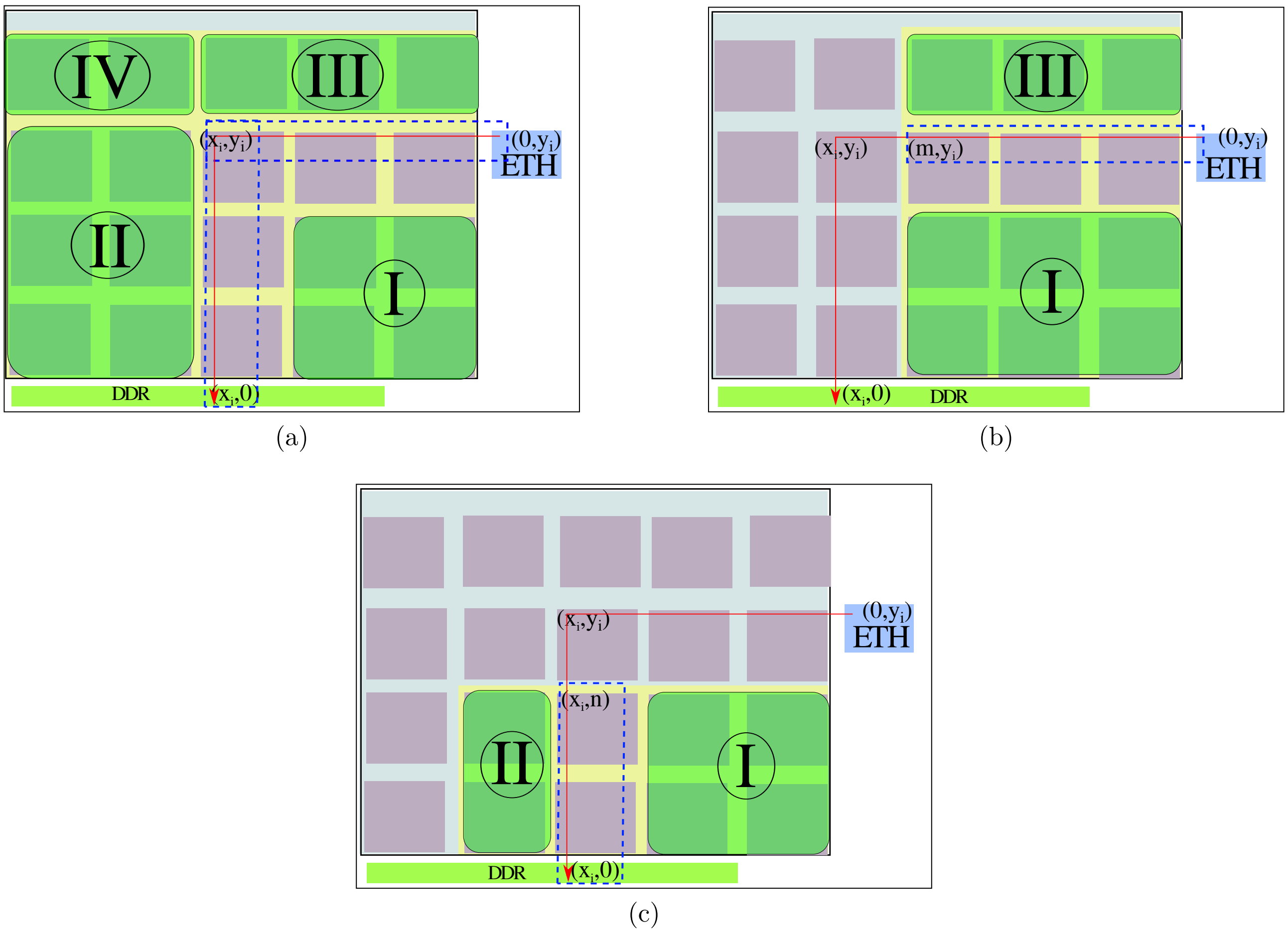


Figure 6.8: For each possible configuration of a critical path, the defined areas and their order in the tasks mapping.

reduce the contention on the I/O outgoing flow, which are explained in the next section.

Example. The step 2 for $FADEC_8$, shown on the bottom part of Figure 6.4, puts in this region I the task t_{f7} on $(1, 3)$ that receives from t_{f1} and t_{f4}. A similar condition is applied for the task mapped on $(1, 1)$. The tasks on the third row must also not send to t_{f6} and t_{f2} while the one on $(1, 1)$ have to send to t_{f2}.

Mapping in region II ($x > x_i$ and $y \leq y_i$)

As in this region the cores share only the same rows with the critical path (Figures 6.8a and 6.8b), then we have to apply only the conditions related to the row of the critical path. Therefore, in this region, each task mapped on the core (x, y) should send data to the tasks on (x_i, y). Besides, it should not send data to all tasks mapped on the row of critical path, starting from $(x_i, 1)$ till $(x_i, y-1)$.

Example. For instance, the step 2 for HM_8, shown by Figure 6.4, puts in this region II a task on $(4, 4)$ that send to t_{h7} and not send to t_{h5}, t_{h3} and t_{h0}, i.e. t_{h6}. Similarly, t_{h4} is mapped on $(4, 3)$ as it sends to t_{h5} and not send to t_{h3} and t_{h0}. The same rules are applied to allocate the tasks on the remaining cores, i.e. $(4, 2)$ and $(4, 1)$.

Mapping in region III ($x < x_i$ and $y > y_i$)

Since $x < x_i$, each task on (x, y) must receive data from the tasks on the column of its critical path, i.e. (x, y_i). However, in order to reduce the contention on the Y part with the critical path, it must also not send data to tasks on the column of the critical path starting from $(x_i, 1)$ till $(x_i, y-1)$, as in the case of Figure 6.8a.

Mapping in region IV ($x > x_i$ and $y > y_i$)

The cores in this region do not share any row or column of the critical path. However, we have to ensure that the tasks in this region do not send any data to the tasks on the row of the critical path starting from $(x_i, 1)$ till $(x_i, y-1)$.

6.4.4 Minimizing the contention of outgoing flows

The rules detailed before minimize the WCTT of the flow incoming from Ethernet. We also define tasks mapping rules that reduce the latency of outgoing core-to-I/O flows, i.e. from the DDR to the Ethernet controller. These rules are applied when tasks remain to be mapped.

The first rule is to allocate the tasks sending to Ethernet on the nearest cores to both the DDR and Ethernet controllers, as this minimize d_2 and d_3 and thus the delay from the task to the DDR and from the DDR to the Ethernet. In fact, the tasks sending to Ethernet send first

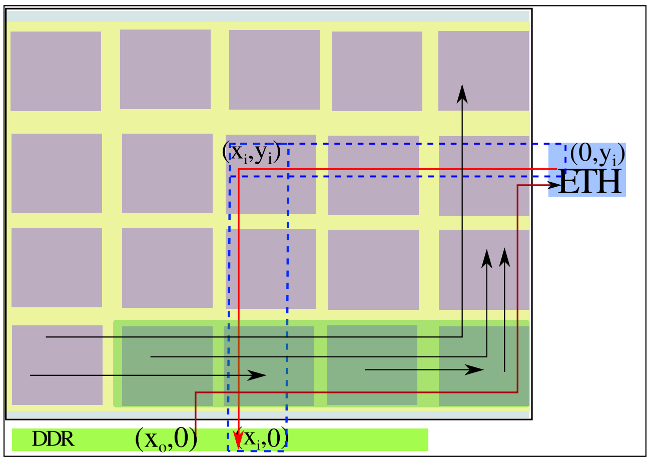
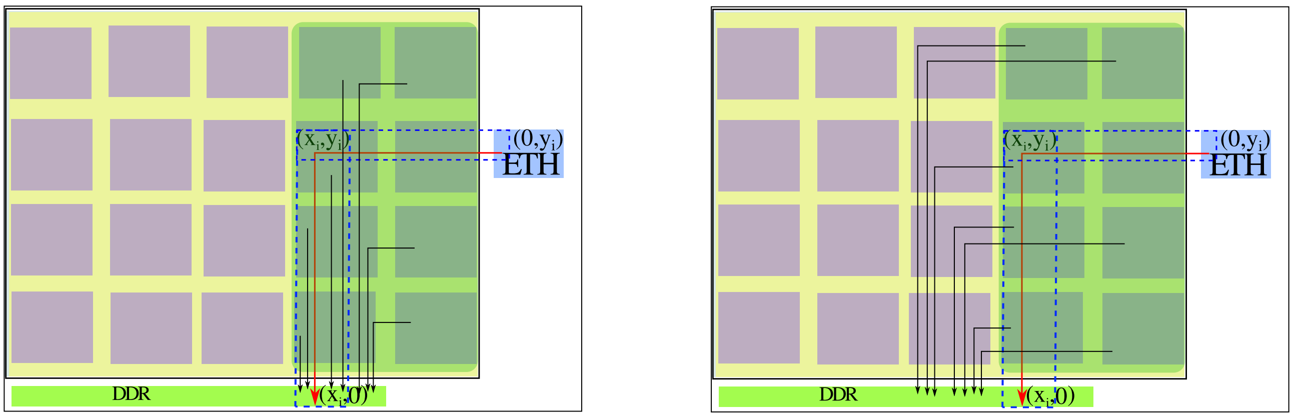


Figure 6.9: Flows blocking directly the path of the outgoing flow.

the data to the DDR controller and then this last one transfer data to Ethernet. Thus, we identify some conditions for mapping the remaining tasks. Where these currently unmapped tasks are going to be allocated can indeed affect the outgoing flows on the NoC, going from the port x_o, and which uses the first row then the first column to reach the Ethernet controller. Remaining tasks that might be allocated on $(x, 1)$ must not send data to the tasks allocated on $(x_j, 1)$, with $x_j < min(x, x_o)$. Actually, when we allocate a task on $(x, 1)$ with $x < x_o$, then this task must not send to the tasks allocated on $(x_j, 1)$ with $x_j < x$. This condition can avoid the contention with the X part of the path of the I/O outgoing flow. However, it can send to the tasks on $(x_j, 1)$ with $x_j > x$, due to the use of the bidirectional links, which eliminate the possible contention with the I/O outgoing flow. On the other hand, when $x > x_o$, the task on $(x, 1)$ should not send to any task allocated on $(x_j, 1)$ with $x_j < x_o$, in order to avoid this X contention with the I/O outgoing flow.

A task on $(x, 1)$ must also not send to the tasks allocated on the first column, in order to avoid the contention with the Y part of the path of the path of the I/O outgoing flow. These conditions can minimize the number of flows that could block the flows outgoing from DDR to Ethernet controller. As show in Figure 6.9, when considering a task on $(x, 1)$ that sends data to tasks allocated on the first row and/or first column, it generates flows blocking directly the path of the I/O outgoing flow.

After applying all rules and allocating the tasks verifying these rules, the remaining tasks are mapped arbitrarily on the cores. Actually, in this work, we do not consider any conditions to



(a) (b)

Figure 6.10: The configuration in (a) shows how the communications with DDR interfere the critical path while this interference is avoided by our mapping as shown in (b).

map the remaining tasks on the remaining cores.

Choosing the DDR port to which tasks should communicate

Once all tasks have been mapped, we have to check the tasks communicating with the DDR. In fact, in this work we consider the communications with DDR controllers even as a final destination, or an intermediate destination. The flows coming from the tasks communicating with DDR and located at the column x_i or at the first column when $x_i = 2$, interfere with the critical path as shown in Figure 6.10a. For this reason, we have to choose for each task communicating with DDR, the corresponding port to which it should send data without interfering with the critical path. Therefore, we consider virtual tasks that will be allocated on the $x_i + 1$ port as a destination port for the tasks located at the column x_i and at the first column when $x_i = 2$, as illustrated in Figure 6.10b.

We note that when it exists more than one critical path in the application, we consider the one with the highest x_i value. However, we also modify the destination of tasks from other applications: we modify the port from $x = x_i + 1$ to $x_i + 2$.

Example. For $FADEC_8$ of our case study B, the task t_{f6} should send data to the port 1 of the DDR. However, this port is used by the critical path coming from the Ethernet interface located at (0,2). Thus, we modify the DDR port destination for this task. The port 2 is used

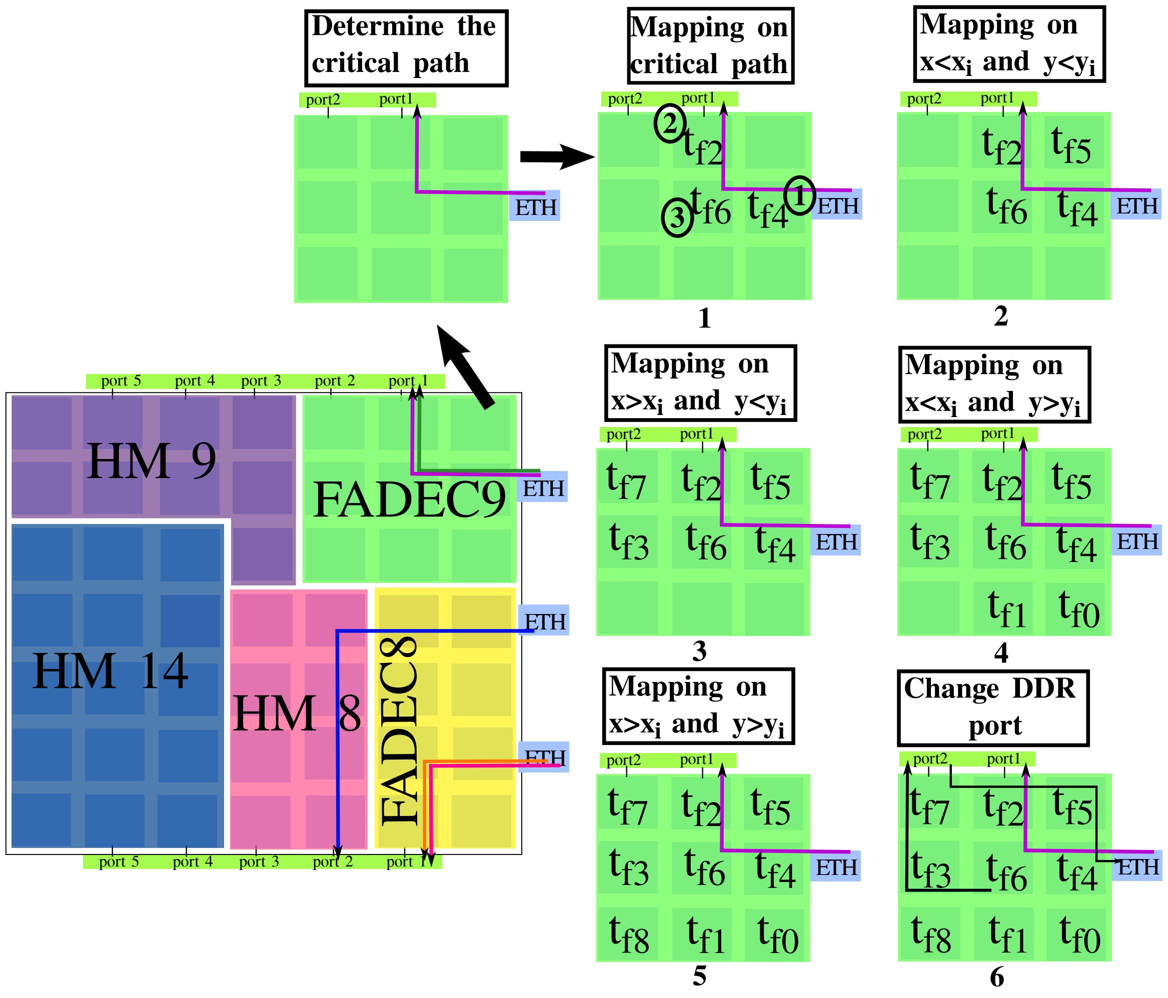


Figure 6.11: Mapping of $FADEC_9$ of the case study B.

by the critical path coming from the Ethernet interface at (0,6). To avoid the contention with these critical paths, the port 3 is chosen as the destination port for the task t_{f_6}, as illustrated on the step 3 in Figure 6.4.

Figure 6.11 illustrates the internal mapping of $FADEC_9$ sharing the Ethernet $(0, 6)$ with HM_9. This mapping applies the same rules as in the mapping of $FADEC_8$, where t_{f6} is allocated on $(2, 6)$. This task sends to DDR, and thus could not be allocated at $(1, 6)$ even it presents a minimum number of EC. However, as it is the sole task that does not receive data from t_{f4} allocated at $(1, 6)$, t_{f6} is thus allocated at $(2, 6)$ as illustrated at the step 1. Besides, the step 6 shows that t_{f6} sends data to the port 2 to avoid the direct contention with the critical path.

Example: Case study A. Now let us go back to the case study illustrated in Figure 6.1 to explain how the tasks are allocated within FFT. t_{ff15} is a task that does not send to DDR and also presents the minimum number of EC. Thus, this task is allocated on the core at $(1, 6)$. Besides, on the core located at $(2, 7)$, we choose a task with a minimum number of IC and does not receive from t_{ff15}, i.e. t_{ff8}. Finally, a task that does not receive from t_{ff15} and does not send to t_{ff8} is chosen to be allocated at $(2, 6)$, i.e. t_{ff12}. However, the remaining tasks are allocated arbitrarily on the remaining regions as they present the same characteristics.

6.5 Conclusion

Chapter 4 has shown that reducing the pessimism when computing the WCTT is a strategy to avoid Ethernet frames to be dropped. However, this strategy is not always sufficient to solve the problem as illustrated in the considered case study A. Existing contention-aware mapping strategies aim at minimizing the inter-core congestion without taking into account requirements of I/O communications of applications. However, the WCTT of core-to-I/O flows depends on the congestions generated by the mapping of both critical and non-critical applications.

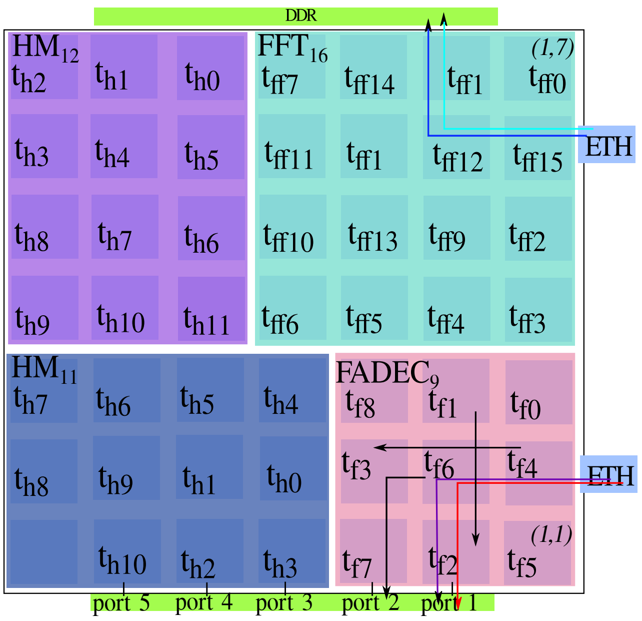
This chapter presents a description of our static mapping strategy of critical and non critical real-time flows that reduces the WCTT of core-to-I/O communications over Tilera-like NoC, called Map_{IO}. Map_{IO} splits the NoC into regions and then allocates in priority critical applications in a dedicated region close to memory and Ethernet controllers. The path of core-to-I/O and the outgoing I/O flows of critical applications are thus reduced. These regions are contiguous and non-fragmented which avoids the contention between applications. Besides, it permits us to allocate a number of applications whose size is equal to the size of the NoC.
Then, the second part of Map_{IO} is the mapping of the tasks within the regions in order to reduce the contentions that core-to-I/O flows can experience over their paths. The evaluation of Map_{IO} method compared to the SHiC method, a method of the state-of-the-art, is presented in the next chapter.

Chapter 7

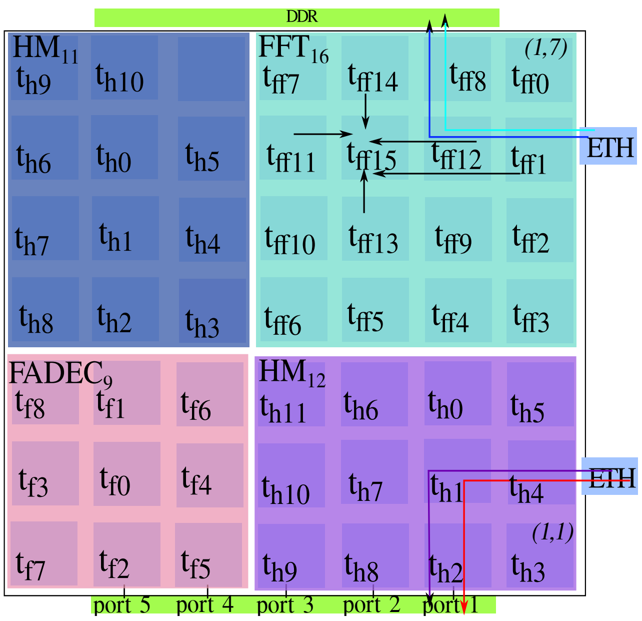
Evaluation of RC_{NoC} and Map_{IO} on case studies

Contents

Chapter 5 has presented RC_{NoC}, a method that allows to reduce the pessimism of the computed WCTT. Its evaluation on a synthetic benchmark shows a significant reduction of the WCTT values compared to the RC method. Besides, chapter 6 has described Map_{IO}, an application placement strategy, that takes into account the communications between cores and I/O interfaces. In this chapter, we propose to evaluate these two methods. Thus, we first evaluate

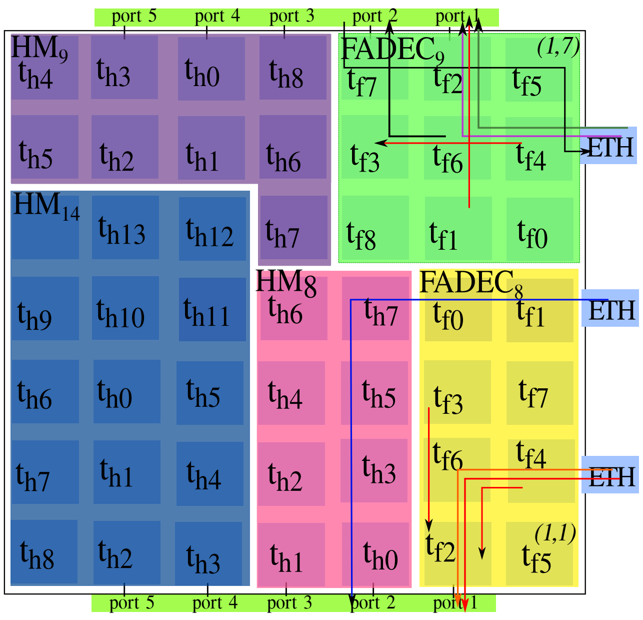


(i) Our mapping

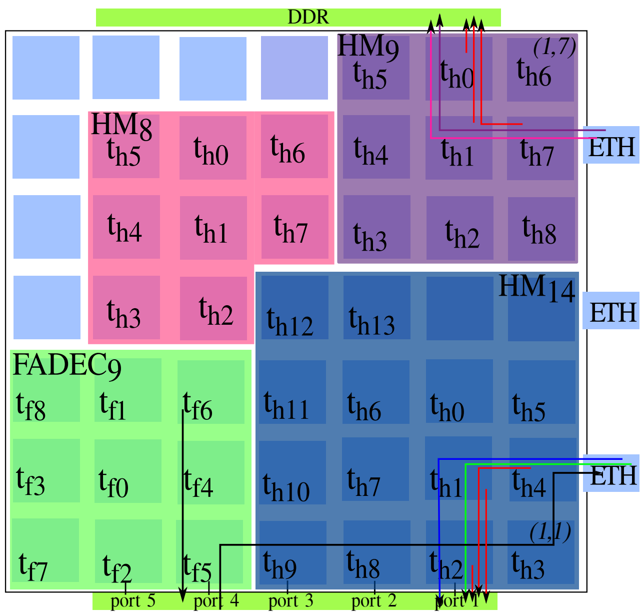


(ii) SHiC mapping

(a) Case study A



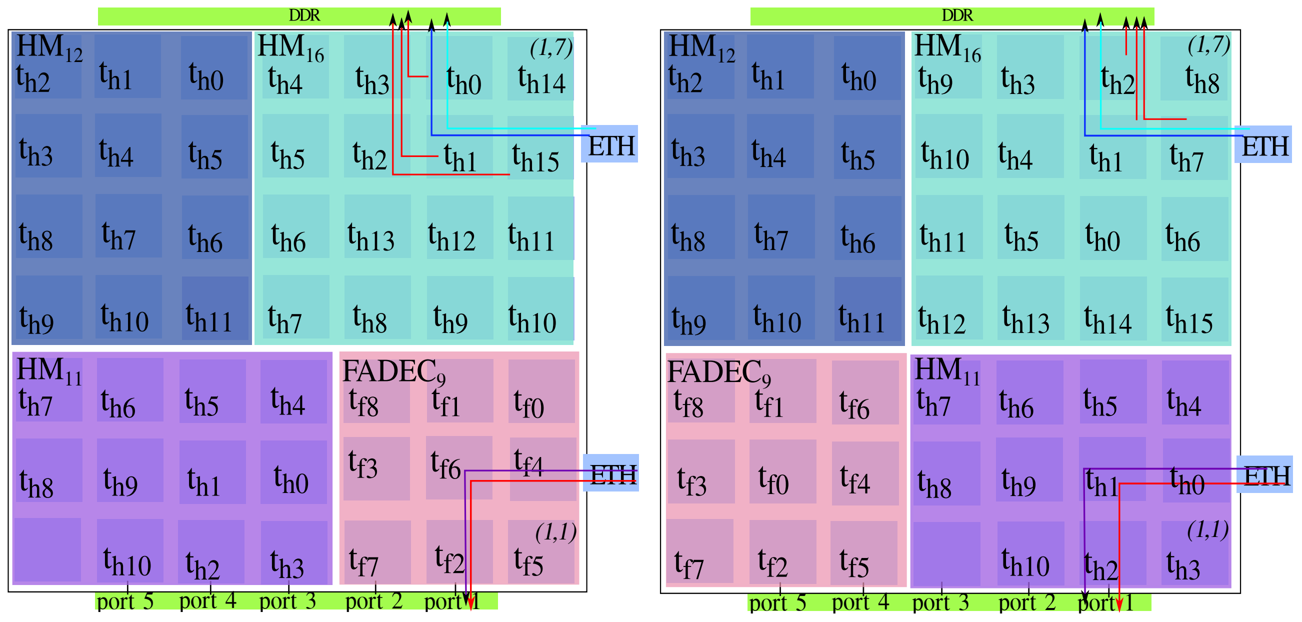
(i) Our mapping



(ii) SHiC mapping

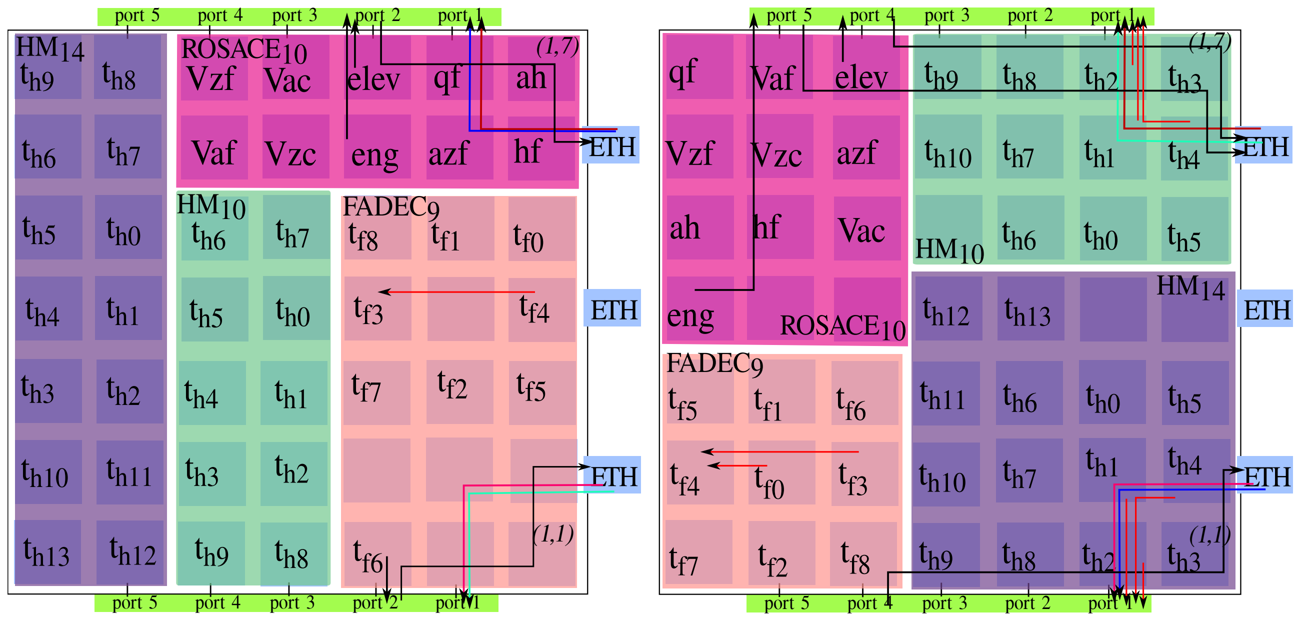
(b) Case study B

Figure 7.1: mapping of flows of case studies A and B by applying Map_{IO} mapping and SHiC mapping.



(i) Our mapping (ii) SHiC mapping

(a) Case study C



(i) Our mapping (ii) SHiC mapping

(b) Case study D

Figure 7.2: mapping of flows of case studies C and D by applying Map_{IO} mapping and SHiC mapping.

the impact of these methods, applied independently and then together, on the initial problem: dropping of Ethernet frames for critical or non-critical applications. Later, we evaluate the impact of these methods, applied together, on the WCTT of both core-to-I/O and core-to-core flows. We thus explain the impact of Map_{IO} rules and the number of unused cores, generated by this strategy, on these WCTT values.
For these evaluations, we use different case studies: A and B (introduced in the previous chapter), C and D (new case studies which are introduced in this chapter). In order to facilitate the access to these case studies, we illustrate, in the beginning of this chapter, the mapping of the case studies A and B, in Figure 7.1, and the one of C and D in Figure 7.2. In these figures, we show the mapping obtained by Map_{IO} and SHiC methods.

7.1 Impact of RC_{NoC} on the core-to-I/O flows

In order to evaluate the impact of RC_{NoC} proposed in Chapter 5 on the core-to-I/O flows, we refer to the case study A introduced at Chapter 4 and illustrated in Figure 4.10. We recall that FFT and HM_{11} share the same Ethernet interface at (0,6), while FADEC and HM_{12} share the Ethernet interface at (0,2). As shown in the section 4.3, the RC method computes the WCTT of a packet of the HM core-to-I/O flow by adding the WCTT of all direct and indirect flows blocking this packet. This computation leads to drop the FFT frame. We recall that the WCTT of a packet of the HM core-to-I/O flow is equal to 400.775 ns by considering the RC method. However, by applying RC_{NoC}, we can note a reduction of 31% which corresponds to a WCTT of 277.725 ns. This delay is equal to the delay obtained by the timeline illustrated in Figure 4.11b modeling the real pipeline behavior of flits transmission.
Actually, this result shows that RC_{NoC} models this pipeline behavior, where Property 3 is applied to remove the impact of the indirect flows as presented in the timeline. In fact, when f_1 is no more blocked by f_2, f_3 is the sole indirect flow that impact the analyzed flow f_a, i.e. the HM core-to-I/O flow. Indeed, Property 3 indicates a separation of $n_{f1} - 1$, i.e. one router corresponding to (3,6), between f_a and f_{id}, to classify f_{id} as non-influent. As f_3 is not separated from the HM core-to-I/O flow by any router, then f_3 is an influent indirect

flow (see Figure 4.11a). Thus, when f_1 arrives to the router located at (3,6), an HM packet progresses without being affected by the flows blocking f_1, i.e. indirect flows: f_4, f_5, f_6, f_7 and f_8. Besides, Property 2 reduces the pessimism by computing the progression of flows without waiting the blocking flows to reach their destination. For example, f_1 does not wait f_3 to reach its destination at (4,4) before it progresses, and thus its maximal blocking delay is computed. Therefore, the WCTT of the 21 packets corresponding to the HM core-to-I/O flow is equal to 5.8 μs which is less than the transmission delay of the FFT frame, i.e. 6 μs. In this case, the FFT frame is no more dropped.

Reducing the pessimism in the computation not sufficient

Let us increase the number of flits composing the FFT core-to-core flows, for example to 3 flits, then the WCTT of a packet of HM core-to-I/O flow increases to 490.47 ns. In this case, additional indirect flows are considered as influent on the core-to-I/O flow. Thus, this core-to-I/O flow takes 10.29 μs to send all the payload to the DDR. This delay is greater than the transmission delay of FFT frame on Ethernet leading to drop it.
On the other hand, a packet of the HM_{11} core-to-I/O coming from the Ethernet interface at (0,2) takes 790.05 ns to reach the DDR memory. Comparing to the RC method, we reduce the WCTT value of only 2%. Actually, Property 3 can not be applied due to the absence of indirect blocking flows. Besides, the flows are made of 19 flits that crosses at maximum four routers. The maximal blocking delay is therefore close to the waiting delay for the blocking flow to reach its destination. Thus, Property 2 does not provide a significant gain against the RC method. This gain of 2% is not sufficient to avoid to drop the FADEC frame as the global WCTT of the HM core-to-I/O flow is equal to 16.5 μs and thus greater than the transmission delay of the FADEC frame.

7.2 Impact of Map_{IO} on the core-to-I/O flows

Problem solved: packets are no longer dropped

As seen previously, reducing the pessimism when computing the WCTT is not sufficient in some cases to avoid dropping Ethernet frames for the case study A. In this section, we compare the mapping generated by Map_{IO}, called Map_{IO} mapping, against the mapping generated by the SHiC method, called SHiC mapping. We also show that the problem mentioned above is solved.

We recall that Figure 7.1a.i illustrates Map_{IO} mapping, while Figure 7.1a.ii shows SHiC mapping. Table 7.1 summarizes the WCTT of the core-to-I/O flows obtained when applying Map_{IO} and SHiC method for the considered case study A. The second line of this table reports the

WCTT of flows	Map_{IO}	SHiC approach
HM_{11} in SHiC and HM_{12} in Map_{IO} blocking FFT_{16}: ETH $\rightarrow$ DDR	0.98 μs $\Longrightarrow$ FFT_{16} received	6.704 μs $\Longrightarrow$ FFT_{16} dropped
HM_{12} in SHiC and HM_{11} in Map_{IO} blocking $FADEC_9$: ETH $\rightarrow$ DDR	11.8 μs $\Longrightarrow$ $FADEC_9$ received	16.5 μs $\Longrightarrow$ $FADEC_9$ dropped

Table 7.1: Table illustrating the WCTT of the core-to-I/O flows, of the case study A, sharing the Ethernet (0,6) and (0,2) using Map_{IO} and SHiC approach.

WCTT values of the HM core-to-I/O flow blocking FFT_{16} and the third one reports these of the HM core-to-I/O flow blocking $FADEC_9$. In both mapping strategies, we indicate whether the problem of dropping Ethernet frame is solved or not.

As mentioned in the section 4.3, when we increase the size of packets of the FFT core-to-core flows to 15 flits, then the FFT frame is dropped when allocating the FFT tasks using the SHiC mapping. Actually, the WCTT of the HM_{11} core-to-I/O flow takes 6.704μs to send all the payload to the DDR by considering RC_{NoC} and 6.762μs by applying RC method. In both cases, the delay of the HM core-to-I/O flow is greater than the arrival delay of the FFT frame, i.e 6μs. However, in Map_{IO} mapping, FFT is still near to Ethernet but we apply Map_{IO} rules to change the mapping of the tasks into FFT. In this mapping, FFT share the Ethernet interface with HM_{12}. The HM_{12} core-to-I/O flow is not blocked. Its global WCTT is equal to 0.98μs and thus the FFT frame is not dropped. We note that in this case, this WCTT is independent from the number of flits making the FFT core-to-core flows, unlike when considering the SHiC mapping. In the SHiC mapping, the HM_{11} core-to-I/O flow is blocked by the FFT core-to-core flows, and so depends from the size of their packets. Thus, when increasing the size of these packets, the indirect flows become influent, and the blocking delay of the HM_{11} core-to-I/O flow increases.

On the other hand, for the Ethernet interface at (0,2), when considering the SHiC mapping, the HM_{12} core-to-I/O flow takes 16.5 μs (16.944 μs by applying RC method) to reach the DDR controller. This delay is greater than the arrival delay of the FADEC frame, i.e. 12.336 μs, and leads to drop FADEC frame. Compared to the SHiC mapping, let us simply permute the HM_{12} and $FADEC_9$ applications. The SHiC mapping of tasks on $FADEC_9$ leads to a WCTT for the HM_{12} core-to-I/O flow of 15.5 μs. This delay is reduced compared to the initial one. However, it is still higher than the transmission delay of the $FADEC_9$ frame on Ethernet, i.e. 12.336 μs, which leads to drop FADEC frame. This demonstrates the need for a task mapping within applications that further reduces the WCTT of core-to-I/O flows.

WCTT of flows	Map_{IO}	SHiC approach
HM_{12} in SHiC and HM_{11} in Map_{IO} blocking $FADEC_9$:	0.98 μs $\Longrightarrow$ FFT_{16} received	6.704 μs $\Longrightarrow$ FFT_{16} dropped
HM_8 in Map_{IO} blocking $FADEC_8$: ETH $\rightarrow$ DDR	6.48 μs $\Longrightarrow$ $FADEC_8$ received	

Table 7.2: Table illustrating the WCTT of the core-to-I/O flows, of the case study B, sharing the Ethernet (0,6) and (0,2) using Map_{IO} and SHiC approach.

Map_{IO} mapping of $FADEC_9$, shown by Figure 7.1a, leads to a WCTT for the HM_{11} core-to-I/O flow of 11.8 μs. We note that this core-to-I/O flow uses the same Ethernet controller as $FADEC_9$, i.e. the one located $(0, 2)$. This delay is lower than the transmission delay of the $FADEC_9$ frame on Ethernet. Therefore, the $FADEC_9$ frame reaches the Ethernet interface after the removal of the HM_{11} frame from the Ethernet buffer.

Besides, we evaluate Map_{IO} on the case study B introduced in the previous chapter in order to compare it to SHiC mapping illustrated in Figure 7.1b. Table 7.2 reports the WCTT values of the HM core-to-I/O flows blocking respectively $FADEC_9$ in the second line, and $FADEC_8$ in the third line. The results show that the WCTT values obtained in Map_{IO} lead to avoid the dropping of $FADEC_8$ and $FADEC_9$ Ethernet frames. Also, we can notice that the WCTT values of the HM core-to-I/O flow in the second line are not reported for SHiC mapping. Actually, SHiC mapping cannot allocate the critical application $FADEC_8$ as illustrated in Figure 7.1b.ii.

7.3 Combining our Map_{IO} and RC_{NoC} is necessary

The previous section has shown that applying Map_{IO} on the case studies A and B solve the problem of dropping Ethernet frames. But, is it sufficient to apply only the mapping strategy without considering the computing method? To answer this question, we consider the following case study, noted C, where figures 7.2a.i and 7.2a.ii illustrate respectively Map_{IO} mapping and SHiC mapping.

This case study is made of the following applications: FFT_{12}, $FADEC_9$ and two instances of HM: HM_{16} and HM_{12}. We consider that the HM_{12} Ethernet frame, transmitted before the HM_{16} frame, holds a payload of 684 Bytes. Besides, we suppose that the core-to-core flows exchanged in the HM applications are made of 16 flits.

In this section, we focus on the Ethernet interface (0,6) which is shared between HM_{12} and HM_{16} in both mapping. Table 7.3 shows the WCTT of the HM_{12} core-to-I/O flow computed when applying RC_{NoC} and the RC method in both mapping illustrated in figures 7.2a.
The results show that by considering the SHiC mapping and whatever is the computing method, the HM_{16} Ethernet frame is dropped. In fact, the WCTT of a packet of the HM_{12} takes 672.75 ns (RC_{NoC}) to reach the DDR. However, the HM_{12} frame is divided into 9 NoC packets. Thus, the global WCTT of HM_{12} core-to-I/O flow to reach the DDR is 6.05 μs which is greater than the transmission delay of HM_{16} on the Ethernet link (5.808 μs). This delay is due to the fact that HM_{12} core-to-I/O flow in SHiC mapping is blocked by 3 direct flows coming from t_{h7}, t_{h1} and t_{h2} (see Figure 7.2a.ii).

On the other hand, when considering Map_{IO}, the WCTT of the core-to-I/O flow computed by RC method leads to drop the HM_{16} frame. The WCTT of a packet of HM_{12} computed by the

	Map_{IO}		SHiC approach	
WCTT of flows	RC_{NoC}	RC method	RC_{NoC}	RC method
HM_{12} blocking HM_{16}: ETH $\rightarrow$ DDR	5.7 μs $\Longrightarrow$ HM_{16} received	5.86 μs $\Longrightarrow$ HM_{16} dropped	6.05 μs $\Longrightarrow$ HM_{16} dropped	6.21 μs $\Longrightarrow$ HM_{16} dropped

Table 7.3: Table illustrating the WCTT of HM_{12} blocking HM_{16} of the case study C and sharing the Ethernet (0,6) using the different mapping strategies and computing methods.

RC method is 651.47 ns and thus its global WCTT to reach DDR is 5.86 μs. This WCTT is greater than 5.808 μs. However, RC_{NoC} presents a reduction of 2.55% of the WCTT of the HM_{12} packet, where the global WCTT of the HM_{12} core-to-I/O flow is 5.7 μs. This WCTT is less than 5.808 μs where the HM_{12} payload is removed from the Ethernet interface buffer before the arriving of HM_{16} Ethernet frame.
Actually, this difference between the values obtained by RC_{NoC} and the RC method is explained by the applicability of Property 2. This property computes the maximal blocking delay of each blocking flow (either direct or indirect flow) without waiting it to reach its destination. We note that in Map_{IO} mapping, the HM_{12} core-to-I/O flow is blocked directly by one flow generated by t_{h15} which explains the values obtained by Map_{IO} mapping compared to SHiC mapping (see Figure 7.2a.i).

As this case study shows the need to combine RC_{NoC} and Map_{IO}, thus in the following evaluations, we use RC_{NoC} to compute the WCTT values of the different flows.

7.4 Impact of Map_{IO} on both core-to-I/O and core-to-core flows

The previous evaluations show the positive impact of Map_{IO}, combined with RC_{NoC}, on the WCTT of the core-to-I/O flows leading to solve the problem of dropping Ethernet frames. In this section, we illustrate on different case studies the impact of the internal mapping rules and the presence of unused cores generated by Map_{IO} not only on the core-to-I/O flows but also on the core-to-core flows.

7.4.1 Impact of Map_{IO} rules

We consider the case studies A and B and provide the WCTT values of the core-to-core flows for the critical applications. Besides, we explain how Map_{IO} rules lead to the different WCTT values for the core-to-I/O and core-to-core flows. We compare these values to those obtained when applying SHiC method. The second column of Tables 7.4 and 7.5 report WCTT values when the mapping is build using Map_{IO}, while the third column reports values assuming the SHiC mapping.

WCTT of flows	Map_{IO}	SHiC approach
HM_{11} and HM_{12}: ETH $\rightarrow$ DDR	0.98 μs	6.704 μs
FFT_{16}: core-to-cores	1136.81 ns	1616.9 ns

Table 7.4: Table illustrating the different results for applications, of case study A, sharing the Ethernet (0,6) using Map_{IO} and SHiC approach.

Case study A: FFT_{16} and HM applications

Table 7.4 refers only to the case study A, and reports the WCTT value of the HM core-to-I/O flow (second line) and the average WCTT value of the core-to-core flows into FFT_{16} (third line). We recall that in SHiC mapping, HM_{11} shares the same Ethernet interface, i.e. located at (0,6), with FFT_{16}, while in Map_{IO} mapping, HM_{12} shares this interface with FFT application.

Impact on the core-to-I/O flows. Map_{IO} reduces the WCTT of the HM core-to-I/O flow by 85 %. This gain is only due to the internal mapping of the tasks into FFT application as FFT in both mapping, i.e. Map_{IO} and SHiC mapping, is allocated in the same region. In fact, the characteristic of the FFT application is that all tasks from t_{ff0} to t_{ff14} send data to one destination, i.e. t_{ff15}. Map_{IO} rules put the task t_{ff15} which presents the minimum EC at the core (1,6). This placement ensures that there are no flow in direct contention with the core-to-I/O flow, as t_{ff15} does not send data. However, the SHiC mapping allocates this task at (3,6) as it is the task with the maximum number of communications. Thus, there is one

WCTT of flows	Map_{IO}	SHiC approach
HM_9, HM_{14}, HM_{11} and HM_{12}: ETH $\rightarrow$ DDR	11.8 μs	16.58 μs
$FADEC_9$: $t_{f6} \rightarrow$ ETH	3341.9 ns	7028.8 ns
$FADEC_9$: core-to-cores	3344.522 ns	2872.4 ns

Table 7.5: Table illustrating the different results for HM and FADEC in both case studies A and B using Map_{IO} and SHiC approach.

flow in direct contention with the core-to-I/O flow generated by the task allocated at (1,6) and having t_{ff15} as destination, i.e. the flow coming from t_{ff1} to t_{ff15} (see Figure 7.1a.ii). This direct blocking flow is also blocked at the core (2,6) by the flow coming from t_{ff12}, as it shares the same link between (2,6) and (3,6). Besides, it is also blocked at (3,6) by other flows coming to the destination of t_{ff15}, as they share the same destination. This explains the increased value of the WCTT of HM core-to-I/O flow when considering the SHiC mapping.

Impact on the core-to-core flows. On the other hand, Map_{IO} has a positive effect on the internal congestion as it reduces the average WCTT by 29.7 %. Although the number of routers traversed by the flows increase by allocating the task t_{ff15} at (1,6), this placement reduces for each flow the blocking delay at the destination. Actually, when t_{ff15} is located at (3,6), i.e. in the SHiC mapping, a flow coming to this core will be blocked by 3 flows having t_{ff15} as destination and coming from 3 different ports. This blocking is due to the Round-Robin arbitration where for example a flow coming to the core (3,6) from the east port, will be blocked before reaching the destination core by flows coming from the north, south and west ports. However, when this task is allocated at (1,6), a flow can only be blocked at the destination by 2 flows coming from 2 ports. For example, a flow coming to the core (1,6) from the west is blocked by 2 flows entering respectively from the north and the south ports. Therefore, each of these 15 flows having t_{ff15} as destination, has their blocking delay reduced at the destination core.

Case studies A and B: $FADEC_9$ and HM applications

Table 7.5 refers to both case studies. For the case study B, HM_{14} shares with $FADEC_9$ the Ethernet interface (0,2) when considering the SHiC mapping. In Map_{IO} mapping, it is HM_9

that shares this interface with $FADEC_9$. However, for the case study A, HM_{11} and HM_{12} are respectively the applications that share the Ethernet interface with FADEC in Map_{IO} mapping and in SHiC mapping (second line of Table 7.5). The third line of this table reports the outgoing flow from the task t_{f6} of $FADEC_9$ to Ethernet via the DDR controller in both case studies. The internal mapping of $FADEC_9$ is the same in both case studies, thus the last line of Table 7.5 reports the average WCTT value of the $FADEC_9$ core-to-core communications. As seen in Figure 7.1b.ii, SHiC mapping is unable to map $FADEC_8$, so we do not report any WCTT result for this application.

Impact on the core-to-I/O flows. Map_{IO} reduces the WCTT of the Ethernet to DDR flow by 28% and by 52% for the flow from task to Ethernet. These gains are mainly due the mapping of the critical applications near both the Ethernet and DDR controllers. Besides, the task mapping of $FADEC_9$, using Map_{IO}, maps t_{f6} in the center of its 3×3 region. Actually, this reduces the Y contention on the critical path, as t_{f6} does not send any data and especially to the task located on the Y part of the critical path (i.e. t_{f2}). Thus, in this mapping illustrated in Figure 7.1b.i, the core-to-I/O flow is blocked directly two flows generated by t_{f4} and t_{f1}. Besides, the outgoing I/O flow crosses 4 routers to reach the Ethernet interface as shown in Figure 7.1b.i. However, the SHiC mapping, shown in 7.1b.ii allocates FADEC application far from Ethernet controller and so the outgoing I/O flow crosses more routers before reaching this interface. Furthermore, the core-to-I/O flow is blocked directly by the HM core-to-memory communications at three routers, i.e. at (1,2), (2,2) and (2,1).

Impact on the core-to-core flows. Map_{IO} increases the internal congestion of $FADEC_9$ by 16% compared to the congestion obtained when applying the SHiC method. Mapping the task with the maximum number of communications at the center reduces the internal congestion, as this core will be the nearest to all cores around it. The center of $FADEC_9$ region is, in Map_{IO} mapping, the core located at $(2, 6)$ (see Figure 7.1b.i). Map_{IO} mapping allocates t_{f6} on this core. However, t_{f6} has the lowest number of communications link with other tasks, as it only receives data from other tasks. SHiC maps t_{f0}, instead of t_{f6}, at the center of the region of

$FADEC_9$, as illustrated in Figure 7.1b.ii, explaining the reduced internal congestion.

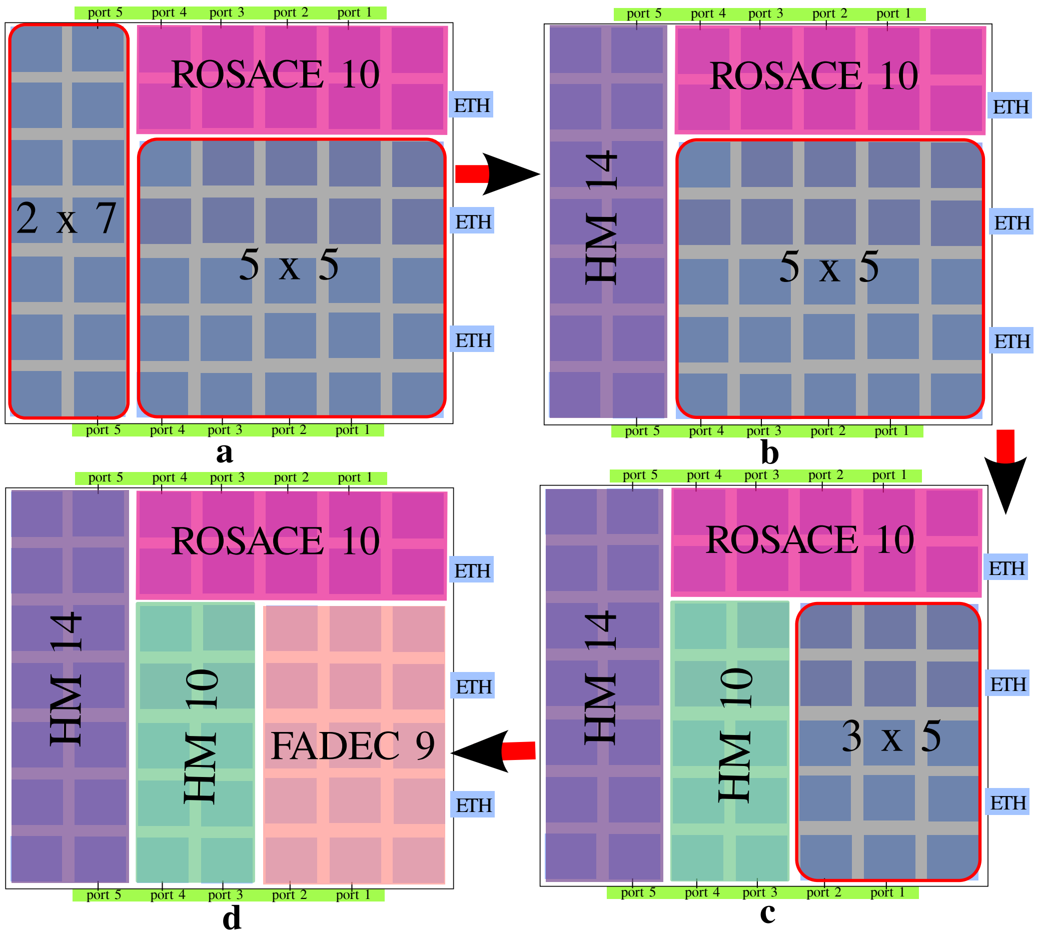


Figure 7.3: The different steps for mapping the applications of the case study D using Map_{IO} .

7.4.2 Impact of unused cores

The goal of this variant of the case studies introduced before is to show the impact of unused cores on the WCTT values, *i.e.* when $UN > 0$. Besides, we show the impact of Map_{IO} on a less-constrained critical application. The modified case study noted D is made of two critical and two non-critical applications. The critical applications are now $ROSACE_{10}$ in addition to $FADEC_9$, while the non critical applications are HM_{14} and HM_{10}.

Figures 7.3 and 7.4 illustrate the application and task mapping steps of Map_{IO}. We assume that only two Ethernet interfaces are used. The applications are allocated in this order: $ROSACE_{10}$, HM_{14}, HM_{10} and $FADEC_9$, as shown in Figure 7.3, due to the presence of two critical applications and of 6 unused cores.

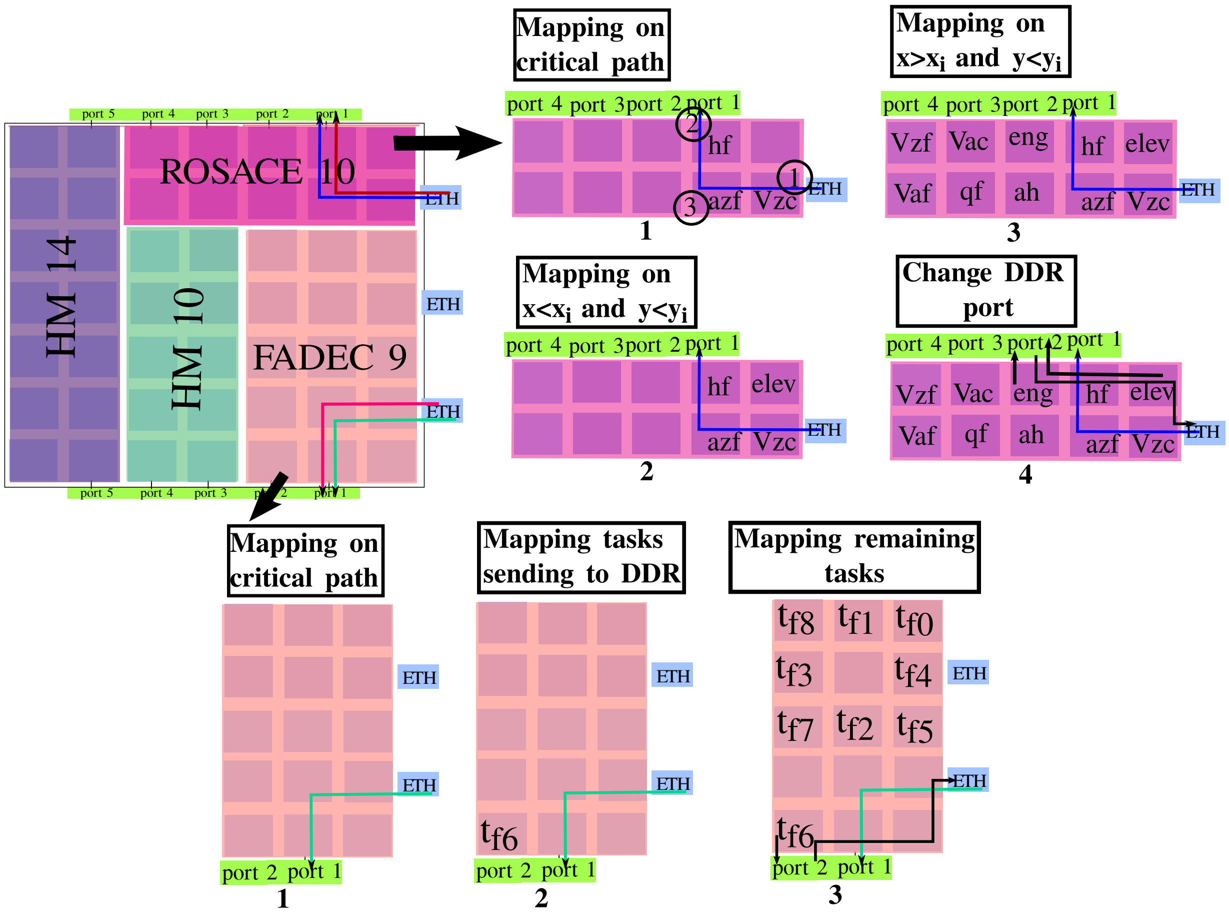


Figure 7.4: The different steps of Map_{IO} task mapping for ROSACE and FADEC applications of the case study D.

Impact on the core-to-I/O flows. Table 7.6 presents the WCTT of core-to-I/O flows from Ethernet controllers for both Map_{IO} and SHiC mapping. Note that SHiC mapping allocates the HM applications near to the Ethernet controllers. The second line of Table 7.6 reports the

WCTT of flows	Map_{IO}	SHiC
Both HM_{14} and HM_{10}: ETH $\rightarrow$ DDR	0.98 μs	16.58 μs
$FADEC_9$: t_{f6} $\rightarrow$ ETH	1262.125 ns	7028.8 ns
$ROSACE_{10}$: $elev$ $\rightarrow$ ETH	399 ns	621 ns
$ROSACE_{10}$: eng $\rightarrow$ ETH	335 ns	1620.925 ns

Table 7.6: Table reporting the WCTT of core-to-I/O flows for the applications of the case study D using Map_{IO} and the SHiC method.

WCTT of both the core-to-I/O flows of: 1) HM_{10} that shares the Ethernet controller located

at $(0, 2)$ with $FADEC_9$, and 2) HM_{14} which shares the Ethernet controller located at $(0, 6)$ with $ROSACE_{10}$. Using Map_{IO}, the non-blocked core-to-I/O flow of HM_{10} takes advantage of the unused cores that are put in priority on its critical path. This explains the decrease from a WCTT of 16.58 μs in case studies A and B to 0.98 μs. However, thanks to Map_{IO} task mapping rules, the WCTT value of HM_{14} is also equal to 0.98 μs, even though no unused cores are available into the region of the ROSACE application. Actually, as shown in the step 1 of Figure 7.4, and by referring to the ROSACE task graph in Figure 4.6, we allocate first on the core (1,6) the task with the minimum EC, i.e Vzc. As this task generates only one flow to the task *elev*, this one is allocated at the same column with Vzc (see step 2 of Figure 7.4), avoiding the contention with the X path of the core-to-I/O flow. Besides, on the core (2,7), we allocate the task with the minimum IC, i.e. hf. Thus, there are no flows coming to this task and so it avoids the Y contention with the critical path. Finally, we choose a task that does not send to hf and does not receive from Vzc to be allocated at the core (2,6), i.e. azf. Therefore, there are no flows in direct contention with the core-to-I/O flow. This explains why the HM_{14} core-to-I/O flow is not blocked in this case. However, in SHiC mapping illustrated in Figure 7.2b.ii, the core-to-I/O flow is blocked directly by the HM_{10} core-to-memory communications at 3 routers as in the previous case study.
We note that the WCTT of the outgoing I/O flows are also reduced thanks to Map_{IO} application mapping that allocates the critical applications near to both the Ethernet and DDR controllers. Besides, we allocate the tasks sending to Ethernet via the DDR controller, i.e. *elev* and *eng*, at the available cores near to the both interfaces, i.e. (1,7) and (3,6). Thus, the outgoing I/O flow will cross less routers to reach the Ethernet interface compared to the SHiC mapping (see Figure 7.2b.i).

Impact on ROSACE core-to-core flows. The average WCTT of core-to-core flows for $ROSACE_{10}$ increases by 28.8%, compared to this obtained when using SHiC mapping. The arbitrary mapping of the tasks of $ROSACE_{10}$ in the 3×2 region (see step 2 of Figure 7.4), whose right corner is located at $(3, 7)$, explains this increased value. In the 6th row, Vaf, qf and *ah* send data to the same task Vzc. Each flow generated by these tasks are thus blocked at each router before reaching Vzc. However, in SHiC mapping, Vzc that presents a maximum number of communications is allocated at the center, thus reducing the number of cores crossed by the flows to reach Vzc and so decreases the number of contention in their path.

Impact on FADEC core-to-core flows. On the other hand, the average WCTT of core-to-core flows for $FADEC_9$ is however decreased by 7.4%, compared to SHiC. Our task mapping, also shown by step 3 of Figure 7.4, indeed leaves unused the core located at $(2, 4)$. This thus reduces the number of contentions for the other flows. Let us take as an example a flow coming from t_{f4} to t_{f3}. This flow in Map_{IO} mapping is not blocked at the core $(2, 4)$ by any flow as it is an unused core. However, in the SHiC mapping illustrated in Figure 7.2b.ii, this flow is blocked at the core $(6, 2)$ by a flow generated from t_{f0} to t_{f3}. Besides, t_{f6} is mapped at $(3, 1)$ when using Map_{IO}. Due to the XY routing policy, all incoming flows towards this task can only come from the north port of core located at $(3, 1)$. There is thus no more contention at the destination core when a flow arrives to the router located at $(3, 1)$. Then, the arbitration for accessing to the next link has fewer input requests to consider, as they are not coming from all directions (we assume a RRA strategy).

What happens when modifying the number of unused cores?

Let us now show the influence of UN_i in the FADEC application on the WCTT of the core-to-I/O flow of HM_{10}. We increase the size of the FADEC application and vary it from 11 up to 15 and with therefore a value of UN_i ranging respectively from 4 to 0. The second line of Table 7.7 reports the computed WCTT in each configuration. We recall that Map_{IO} task mapping rules put the unused cores in priority on the critical path by the following order of priority: $\{(1, 2), (2, 1), (2, 2)$ and $(1, 1)\}$. Figure 7.5 illustrates the internal mapping of FADEC tasks when varying the number of unused cores from 4 to 0.

$UN_i(Map_{IO})$	4	3	2	1	0
WCTT of ETH $\rightarrow$ DDR flow	0.98μs	0.98μs	0.98μs	6.45μs	23.7μs
Average core-to-core impact	+11.8%	+16.4%	+7.4%	−14.5%	−20.8%

Table 7.7: Table reporting the WCTT of the core-to-I/O flow of HM when varying the number of unused cores (U_i) by $FADEC_9$.

Impact on HM core-to-I/O flow. When the flow is not blocked by any other flows, its WCTT is equal to 0.98μs (cases where $UN_i = 4$ up to $UN_i = 2$, i.e. the unused cores are located at $(1, 2)$ and $(2, 1)$). When we allocate two tasks on cores ($(2, 2)$ and $(1, 1)$), i.e. cases where $UN_i = 3$ and $UN_i = 2$, the core-to-I/O flow is not in direct contention with other flows on its critical path, even if these tasks generate flows or receive flows. These flows do not impact directly the path of the core-to-I/O flow. However, if we allocate a task on core $(2, 1)$, as in the case where $UN_i = 1$, i.e. t_{f13}, then this task generates a flow that is in direct contention as it receives data from the tasks allocated above the first row. Thus, the core-to-I/O flow is blocked by one flow at the core $(2, 2)$, i.e. the one coming from t_{f2} to t_{f13}. We note that when $UN_i = 1$, we allocate t_{f6} at the core $(2, 2)$ as it does not send data to t_{f13} and thus reduces the number of flows blocking the core-to-I/O flow. However, the WCTT further increases significantly when $U_i = 0$ and reaches 23 μs. In this case, the core-to-I/O flow is blocked by a flow generated from the task allocated at $(1, 2)$, i.e. t_{f14}. Besides, it is blocked by the flows coming to t_{f13}. Indeed, these blocking flows are assumed to be blocked at each router of their path, which increases the blocking delay of the core-to-I/O flow. To summarize, with a few number of unused cores (2 here) the WCTT of the core-to-I/O is significantly reduced.

Impact on FADEC core-to-core flows. Let us now evaluate how the value of UN_i, in the various instances of the FADEC application, influences the average value of the WCTT of its core-to-core flows, i.e. its internal congestion. We note $AWCTT_{SHiC}$ and $AWCTT_O$ the average value of the WCTT of core-to-core flows computed using respectively the SHiC method and our approach, Map_{IO}. The last line of Table 7.7 thus reports the impact of Map_{IO} compared to the SHiC method using the following equation: $\frac{AWCTT_{SHiC} - AWCTT_O}{AWCTT_{SHiC}}$.

When reducing the value of UN_i: 1) the absolute values of the internal congestion obviously increases, as the number of tasks is increased and 2) the impact of Map_{IO} on the internal congestion changes from a positive effect to a negative one compared to SHiC (except for $UN_i = 3$). For instance, for $FADEC_{11}$ having $UN_i = 4$, Map_{IO} has a positive effect of 11.8% on the internal congestion compared to SHiC. For $FADEC_{15}$ having $UN_i = 0$, Map_{IO} has however a negative effect of 20.8%. Reducing the number of unused cores indeed simply

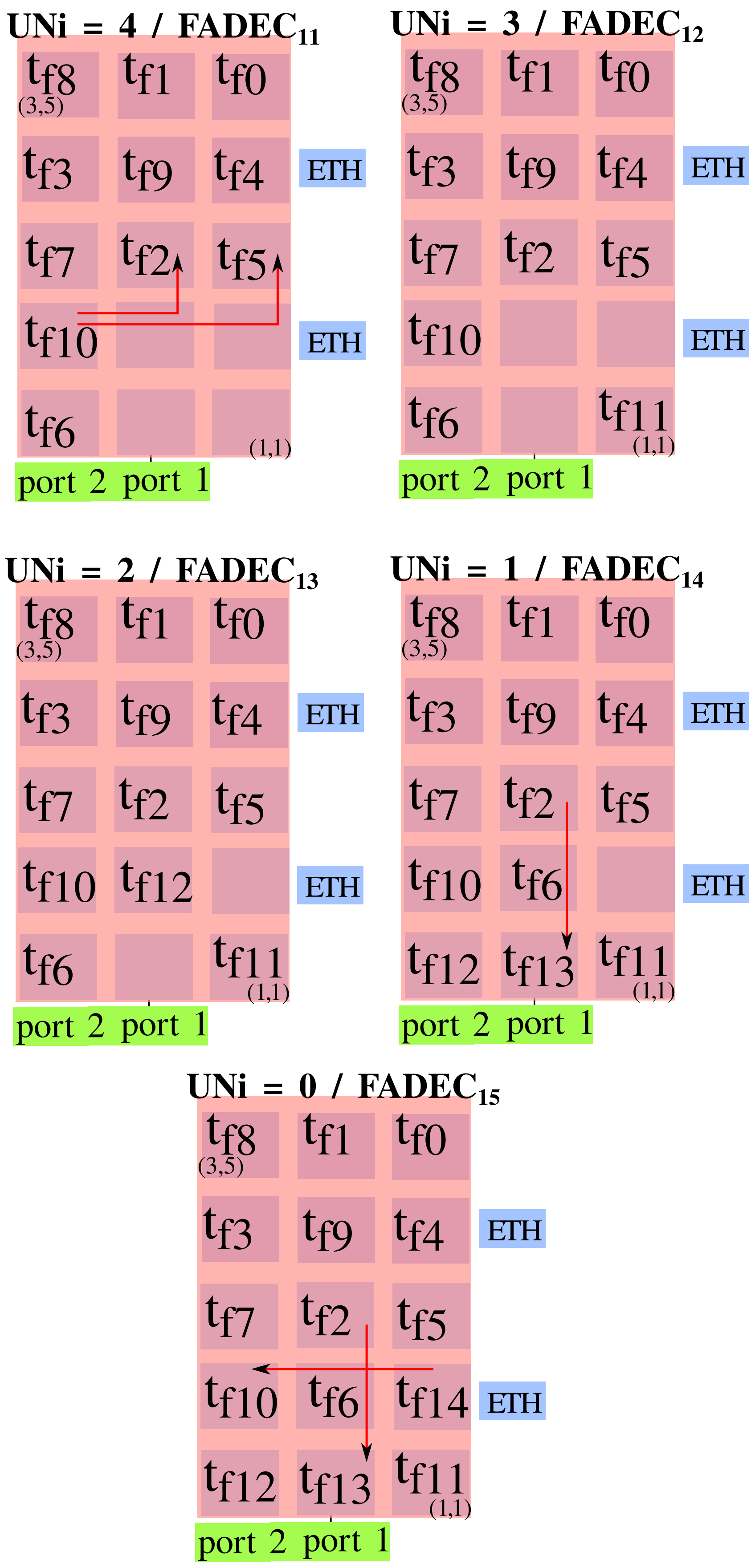


Figure 7.5: Map_{IO} mapping of FADEC when varying the number of unused cores UN_i.

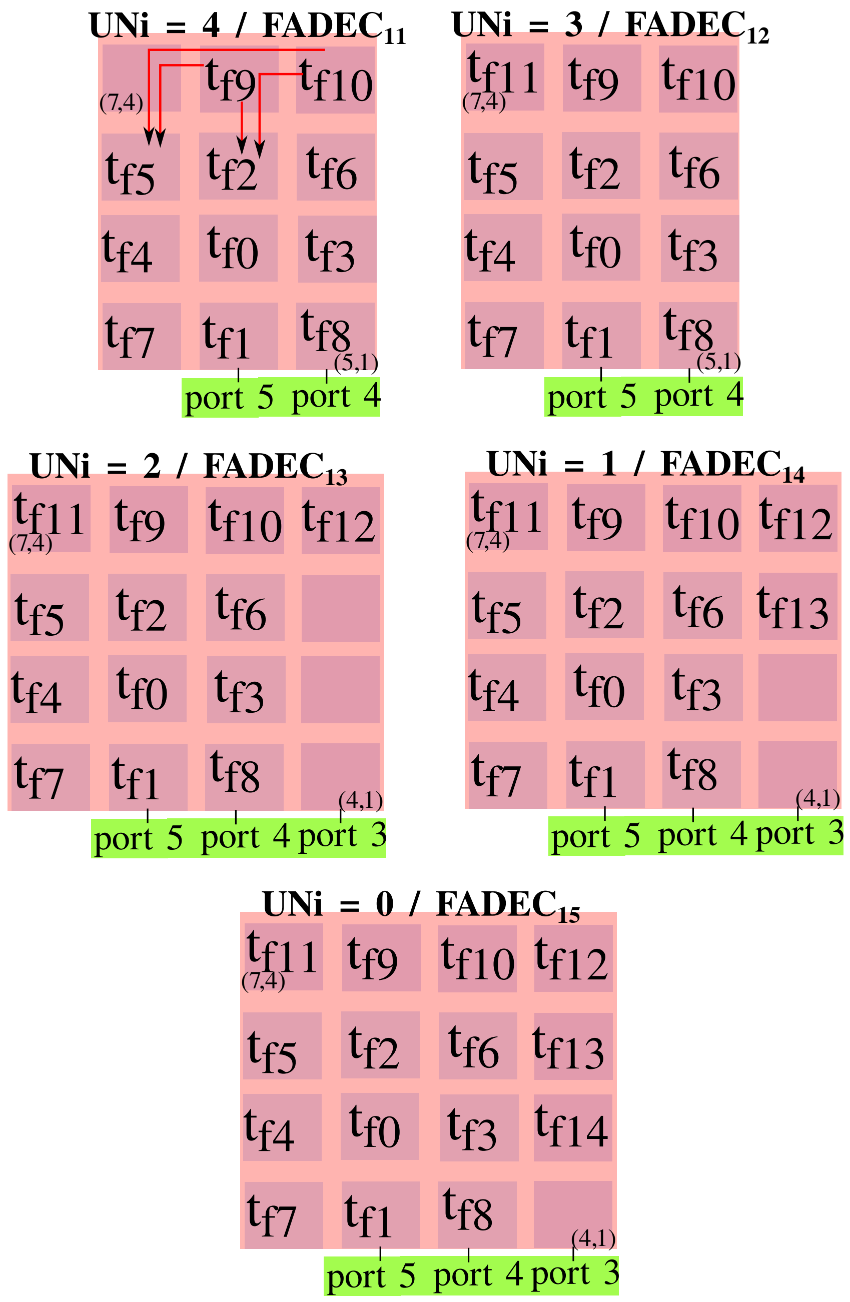


Figure 7.6: SHiC mapping of FADEC when varying the number of unused cores UN_i referring to Map_{IO} mapping.

increases the number of contentions both core-to-I/O and core-to-core flows can experience.

Figure 7.6 shows the internal mapping tasks of FADEC by considering the SHiC mapping, when varying the unused cores from $UN_i = 4$ to $UN_i = 0$ referring to Map_{IO} mapping in terms of UN_i. Thus, when mapping $FADEC_{11}$, i.e. $UN_i = 4$, we can see that Map_{IO} mapping in Figure 7.5 allocates FADEC within a region of 3×5, thus including these 4 unused cores in its region, while the SHiC mapping allocates FADEC within a region of 3×4, and thus including only one unused core.
In Map_{IO} mapping, the 2 unused cores located at $(2, 2)$, $(1, 2)$ reduces the contention between flows. If we take the flows coming from t_{f10} respectively to t_{f2} and t_{f5}, then these flows are not blocked at these cores by any flow. But, in the SHiC mapping, these flows will be blocked at $(6, 4)$ by the flows generated by t_{f9}, and the contention is only reduced by one unused core, i.e. $(7, 4)$, for the flow going to t_{f5}. However, when moving from $FADEC_{11}$ to $FADEC_{12}$, the positive effect of Map_{IO} increases, while the value of UN_i decreases. We explain this discrepancy by the inability of the SHiC to include unused cores in the region allocated to $FADEC_{12}$. As seen in the Figure 7.6 when $UN_i = 3$, SHiC indeed allocates a 3×4 region, with thus no unused cores. However, Map_{IO} still uses a 3×5 region, with thus 3 unused cores as for the $FADEC_{11}$ case. This positive impact reduces for $FADEC_{13}$ but still higher than the SHiC mapping. Even that the SHiC mapping presents more unused cores than Map_{IO} mapping, the placement of t_{f6} at the corner in Map_{IO} mapping, i.e. at $(3, 1)$, reduces the number of blocking flows at this core. Actually, each flow arriving to the core $(3, 1)$ will be blocked at this core by two flows coming respectively from the east and the north port. In the case of the SHiC mapping, t_{f6} is allocated at the core $(5, 3)$ and each flow received by this task, will be blocked at this core by 3 flows coming respectively from the north, south and west ports.

When decreasing the number of unused nodes to $UN_i = 1$ and $UN_i = 0$, t_{f6} is now allocated at the core $(2, 2)$ as we have to apply Map_{IO} mapping rules, i.e. t_{f6} does not send any flow to the t_{f13} allocated at $(2, 1)$. Thus, each flow received by t_{f6} will incur a contention with two to three flows at the core $(2, 2)$. These flows are coming from the different ports of the router due to the RRA. This contention at t_{f6} is the same in the case of the SHiC mapping as t_{f6} is also allocated approximatively at the center of the region, i.e. at $(5,3)$ as illustrated in Figure 7.6. However, in these cases, the SHiC mapping provides respectively 2 and 3 unused cores when allocating $FADEC_{14}$ and $FADEC_{15}$, while Map_{IO} mapping includes respectively 1 and 0 unused cores. This explains the negative impact on the core-to-core internal congestion.

We have to note that in SHiC mapping, $FADEC_{14}$ and $FADEC_{15}$ could not be allocated on the NoC when considering all the applications of the case study D. Thus, these unused cores presented in SHiC mapping are considered in the case where FADEC is allocated sole on the NoC.

7.4.3 Discussion

The results in this chapter show the positive impact of Map_{IO}, combined with RC_{NoC}, on the WCTT values of the core-to-I/O flow leading to solve the problem of dropping Ethernet frames. We have seen that Map_{IO} leads to a significant reduction of 94% for the WCTT values, in two cases: 1) when including unused cores in the region of the critical applications, and 2) when allocating a less-constrained critical application. However, this mapping has a reasonably limited impact on the core-to-core congestions. Actually, we have shown that there is a positive impact of the presence of unused cores on the core-to-core flows congestions. However, in the case of $UN_i = 0$, this impact could be negative.

7.5 Conclusion

This chapter presents an evaluation of both of RC_{NoC} and Map_{IO}. We have first shown on the case study A how RC_{NoC} impacts the WCTT of the core-to-I/O flows, avoiding in some cases to drop Ethernet frames. However, the reduction of the WCTT values is not sufficient to avoid this problem. Thus, applying Map_{IO} on this case study is a solution to solve this problem and this by reducing the contention on the core-to-I/O flows.

However, the mapping strategy applied without reducing the pessimism in the computation of the WCTT could not be the best solution. We have illustrated on the case study C the need to combine RC_{NoC} with Map_{IO} to avoid dropping Ethernet frames. The results show on this case study that even applying Map_{IO}, the WCTT values computed by the recursive method leads to drop the Ethernet frame. Despite the small reduction of this WCTT when applying RC_{NoC} (2.5%), this is sufficient to solve the problem.

Finally, we evaluate the impact of the rules of Map_{IO} and the presence of the unused cores generated by this strategy on both core-to-I/O and core-to-core flows. Our results show on realistic avionics case studies that the core-to-I/O transmission delays are significantly reduced, up to 94%. Meanwhile, the internal congestion for the core-to-core flows can increase up to 28.8%, but slightly impacting this congestion in the other cases we have considered. Besides, we have noticed on some case studies the inability of existing mapping strategy to allocate all applications whose size does not exceed the size of NoC, unlike Map_{IO}.

Chapter 8

Conclusion

Contents

8.1 Summary of Thesis Contributions

In this thesis, we are interested in the use of NoCs in real-time systems interconnected to sensors and actuators via Ethernet. In this context, a congestion in the NoC, due to wormhole switching, delays the core-to-I/O flow (coming from Ethernet) leading to an overflow of the buffer of the Ethernet interface which is of limited capacity. Therefore, incoming Ethernet frames could be dropped. Real-time packet schedulability analysis must then be done, taking into account all the types of flows. The objective of this thesis was to analyze the WCTT of the different types of flows and to reduce the WCTT of the core-to-I/O flow in order to avoid the drop of Ethernet frames. We have illustrated two main problems in the existing methods when applying them in this context over a Tilera-like architecture:

1. Existing WCTT computing methods do not model the pipeline transmission of flits over wormhole NoCs, thus leading to an over-approximation of the WCTT values. Actually,

these methods consider that an analyzed flow can be blocked by others flows while these last flows have not reach their destinations. Besides, they add the transmission delays of all direct and indirect blocking flows to the transmission delay of the analyzed flow. The pessimism values of the WCTT, especially for the core-to-I/O flows, leads to drop Ethernet frames.

2. Existing contention-aware mapping strategies aim to minimize only the inter-core congestion without taking into account the requirements of I/O communications of applications. Then, the WCTT of core-to-I/O flows depends on the congestions generated by applications allocated next to the Ethernet interfaces. These mapping strategies cannot reduce the congestion on the core-to-I/O flows, leading to drop Ethernet frames.

In order to reach our objective, i.e. to avoid the drop of Ethernet frames, two approaches have been proposed:

1. A WCTT computing method, noted RC_{NoC}, that models the pipeline transmission of flits. For this purpose, we have defined three properties to reduce both the number of scenarios to be explored when performing a WCTT analysis and the computed WCTTs values. Using these properties, we compute the maximal blocking delay a flow can suffer from the blocking flow. This leads to eliminate the need to wait till the blocking flow reaches its destination. Besides, we identify the indirect flows that do not impact the transmission of an analyzed flow. Then, the delay of these flows are not added to the transmission delay of the analyzed flow. We have implemented these properties in an algorithm based on a recursive method to compute the WCTT of the flows.

2. A static mapping strategy of critical and non critical real-time flows, noted Map_{IO} that reduces the WCTT of core-to-I/O communications over Tilera-like NoC. This mapping is divided into two phases. In the first phase, the NoC is split into regions where critical applications are allocated in priority in dedicated regions close to memory and Ethernet controllers. Thus, the lengths of core-to-I/O communications of critical and non-critical applications are thus reduced. This phase ensures a mapping of all applications without

being fragmented over the NoC. In the second phase, the tasks of each application is mapped within its region in such a way that the contentions core-to-I/O communications experience are reduced.

We have first evaluated the impact of RC_{NoC} on the WCTT of core-to-core flows. We compared the results against an existing recursive method (RC) on a synthetic benchmark. These results shown a significant improvement of this WCTT. The last chapter has shown the impact of RC_{NoC} and Map_{IO} on the core-to-I/O flows over several realistic case studies. We have evaluated their behavior facing the initial problem compared to current state-of-the-art method. The results have shown the need to combine both methods in order to avoid the drop of Ethernet frames. Besides, these methods lead to a significant improvement of the WCTT of the core-to-I/O flows. This WCTT is reduced by up to 29% when a critical application presenting an all-to-all communicate is allocated in the region near to I/O interfaces. However, this reduction reaches 94% when unused cores are allocated in the region of this critical application. Besides, we find a similar result when a less-constrained critical application is allocated in this region without the presence of unused cores.
The impact of our approach on the core-to-core flows is also evaluated. The internal congestion for the core-to-core flows can increase up to 28.8%, but slightly impacting this congestion in most the other cases we have considered.

In the end, the work in this thesis addresses a new problem where the I/O constraints are integrated within NoC communications. This work has led to find an approach to avoid the problem of the drop of Ethernet frames and computing tightness values of the WCTT. The study done during this thesis research opens several perspectives that we detail in the next section.

8.2 Future Work

The analysis on the case study D, which shows a negative impact of Map_{IO} on the internal congestion on the NoC, leads to the **first perspective**: consider a mapping strategy where a trade-off between reducing the WCTT of the core-to-core and the core-to-I/O flows is done. Actually, in real-time applications, the WCTT of the different flows must be lower then a predetermined deadline. Thus, we have to merge our mapping method, Map_{IO} with the congestion-aware strategies for core-to-core flows. We then allocate a number of tasks using Map_{IO} in the region impacting the core-to-I/O flows in such a way to ensure that the Ethernet frame will not be dropped. The remaining tasks are allocated in the remaining region by applying the congestion-aware strategies for core-to-core flows. Besides, we have shown the positive impact of the unused cores not only on the core-to-I/O flows but also on the core-to-core flows. Thus, we can exploit the presence of the unused cores in order to reach the objective of this perspective.

The evaluation on the case study D, has shown an increasing value of the internal congestion of ROSACE application. The average WCTT of core-to-core flows for ROSACE increases by 28.8%, compared to the mapping generated by a current state-of-the-art method (SHiC). Our mapping method, Map_{IO}, has allocated the tasks on the square 2×2 near to Ethernet interface in order to reduce the contention on the path of the core-to-I/O flow (see steps 1 and 2 of Figure 7.4). These tasks are chosen arbitrary among a number of tasks verifying the Map_{IO} rules. Besides, as explained in section 7.4.2, the remaining tasks are allocated arbitrary in the remaining region 3×2 (see steps 2 and 3 of Figure 7.4). Then, Map_{IO} method could generate different possibilities of mapping by changing the tasks chosen with applying always the Map_{IO} rules. Thus, during this work, we have considered another mapping, generated by Map_{IO}, for ROSACE tasks as illustrated in Figure 8.1. This mapping leads to the same average WCTT of core-to-core flows as in the SHiC method. Actually, in this mapping, we apply our mapping rules on the critical path so the core-to-I/O flow is still unblocked. We reduce the internal congestion by applying rules of the SHiC method. Then, we allocate the tasks presenting the maximum number of communications, i.e. Vzc and Vac, at the center of the remaining region 3×2.

This work was based on a Tilera-like architecture. Thus, it is important to consider this **second perspective**: generalize the problem addressed in this work regardless of NoC architecture. This needs first to address this problem on different number of architectures. Then, we plan

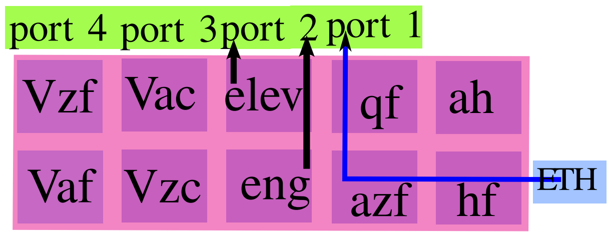


Figure 8.1: Other possibility of the task mapping for ROSACE using our approach.

to study how to adapt our mapping strategy when assuming for instance a torus network and non-blocking routers, as in the MPPA many-core from Kalray [dDvAPL14], or simply different arbitration mechanisms than Round-Robin, such as priority arbitration [BHI14].
Besides, the RC_{NoC} method could be also generalized regardless of the NoC architectures. Actually, the properties of this method are based on the pipeline transmission of the wormhole networks which is the most used in the different NoC architectures. Thus, we have to study the WCTT when assuming buffers within routers having a capacity of more than one flit and/or supporting of multiple VCs.

As reducing the WCTT is important in real-time applications, then our **third perspective** is: consider different mechanisms to reduce the WCTT of the flows. For this purpose, an extension of the RC_{NoC} is to add the assumptions at the application level: [DNNP14], is thus complementary to our work and could be used to further reduce the pessimism of computed WCTTs. Besides, the segmentation mechanisms of the packets introduced by [LBBN16] could be extended to further reduce the computed WCTTs. Actually, the segmentations of specific packets could generate non-influent indirect flows. For example, let us consider the scenario of Figure 5.14 where f_1 the analyzed flow. If we suppose that f_2 has a size of 6 flits, then in this case f_3 is an indirect flow impacting the transmission of f_1. However, if we segment the packet of f_2 into 2 packets of size respectively of 3 flits, then f_3 becomes an indirect non-influent flow.

This work addresses the use of NoCs in real-time systems interconnected to sensors and actuators via Ethernet. As **long term perspective**, we can consider the use of NoCs in avionics domain, where NoCs could be interconnected via AFDX network. Then, it is interesting in this case to study the end-to-end WCTT of flows. Besides, we can consider the problem of optimizing the latency and the bandwidth in this context.

Finally, it is interesting to study and analyze the behavior of existing NoC architectures in real-time systems. This analysis could lead to find the most adapted architecture to be used in a given real-time context with a minimal hardware complexity.

Bibliography

[AG03] Adrijean Andriahantenaina and Alain Greiner. Micro-network for soc: Implementation of a 32-port spin network. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 11128. IEEE Computer Society, 2003.

[BB04] Davide Bertozzi and Luca Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *IEEE circuits and systems magazine*, 4(2):18–31, 2004.

[BC06] Luciano Bononi and Nicola Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, pages 154–159. European Design and Automation Association, 2006.

[BCGK04] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Qnoc: Qos architecture and design process for network on chip. *Journal of systems architecture*, 50(2):105–128, 2004.

[BDM02a] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, Jan 2002.

[BDM02b] Luca Benini and Giovanni De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 418–419. IEEE, 2002.

[BHI14] Alan Burns, James Harbin, and Leandro Soares Indrusiak. A wormhole noc protocol for mixed criticality systems. In *Proc. of the IEEE 35th Real-Time Systems Symposium, RTSS*, pages 184–195, Rome, Italy, December 2014.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[BS04] Tobias Bjerregaard and Jens Sparso. Virtual channel designs for guaranteeing bandwidth in asynchronous network-on-chip. In *Norchip Conference, 2004. Proceedings*, pages 269–272. IEEE, 2004.

[BS05] Tobias Bjerregaard and Jens Sparso. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *Design, Automation and Test in Europe*, pages 1226–1231. IEEE, 2005.

[CCM07] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for dynamic task mapping in noc-based heterogeneous mpsocs. In *18th Intl Workshop on Rapid System Prototyping (RSP)*, pages 34–40, 2007.

[CM08] Chen-Ling Chou and Radu Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *IEEE Intl. Conf. on Computer Design (ICCD)*, pages 164–169, 2008.

[COM08] Chen-Ling Chou, Umit Y Ogras, and Radu Marculescu. Energy-and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1866–1879, 2008.

[CPC08] Nicola Concer, Michele Petracca, and Luca P Carloni. Distributed flit-buffer flow control for networks-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 215–220. ACM, 2008.

[CSG99] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[DA93] William J. Dally and Hiromichi Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE transactions on Parallel and Distributed Systems*, 4(4):466–475, 1993.

[Dal90] William J Dally. *Virtual-channel flow control*, volume 18. ACM, 1990.

[DBL05] Yves Durand, Christian Bernard, and Didier Lattard. Faust: On-chip distributed architecture for a 4g baseband modem soc. *Proceedings of Design and Reuse IP-SOC*, 5:51–55, 2005.

[dDvAPL14] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Proc. of the Conf. on Design, Automation & Test in Europe (DATE'14)*, pages 97:1–97:6, 2014.

[DMB06] Giovanni De Micheli and Luca Benini. *Networks on chips: technology and tools*. Academic Press, 2006.

[DNNP14] Dakshina Dasari, Borislav Nikoli'c, Vincent N'elis, and Stefan M Petters. Noc contention analysis using a branch-and-prune algorithm. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):113, 2014.

[DPPB$^+$12] Manel Djemal, François Pêcheux, Dumitru Potop-Butucaru, Robert De Simone, Franck Wajsburt, and Zhen Zhang. Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–8, Karlsruhe, Germany, October 2012.

[DRGR03] John Dielissen, Andrei Radulescu, Kees Goossens, and Edwin Rijpkema. Concepts and implementation of the philips network-on-chip. In *IP-Based SoC Design*, pages 1–6, 2003.

[dSCCM10] Ewerson Luiz de Souza Carvalho, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Dynamic task mapping for mpsocs. *Design & Test of Computers*, 27(5):26–35, 2010.

[DT01] William J Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[DT04] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[DYN03] Jose Duato, Sudhakar Yalamanchili, and Lionel M Ni. *Interconnection networks: an engineering approach*. Morgan Kaufmann, 2003.

[FDLP13] Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. Smart hill climbing for agile dynamic mapping in many-core systems. In *Proc. of the 50th Annual Design Automation Conference*, page 39, 2013.

[FFF09a] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A method of computation for worst-case delay analysis on SpaceWire networks. In *Proc. of the 4th Intl. Symp. on Industrial Embedded Systems (SIES)*, pages 19–27, Lausanne, Switzerland, July 2009.

[FFF09b] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A method of computation for worst-case delay analysis on spacewire networks. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 19–27. IEEE, 2009.

[FFF11] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. Using network calculus to compute end-to-end delays in spacewire networks. *ACM SIGBED Review*, 8(3):44–47, 2011.

[FFF12] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A sensitivity analysis of two worst-case delay computation methods for spacewire networks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 47–56. IEEE, 2012.

[FRD$^+$12] Mohamamd Fattah, Marco Ramirez, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. Cona: Dynamic application mapping for congestion reduction in many-core systems. In *30th Intl. Conf. on Computer Design (ICCD)*, pages 364–370, 2012.

[FRX+14] Mohammad Fattah, Amir-Mohammad Rahmani, Thomas Canhao Xu, Anil Kanduri, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Mixed-criticality run-time task mapping for noc-based many-core systems. In *22nd Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 458–465. IEEE, 2014.

[Gai15] Pierre-Emmanuel Gaillardon. *Reconfigurable Logic: Architecture, Tools, and Applications*, volume 48. CRC Press, 2015.

[GDR05] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[GDvM+03] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. Guaranteeing the quality of services in networks on chip. In *Networks on chip*, pages 61–82. Springer, 2003.

[HDV+11] Jason Howard, Saurabh Dighe, Sriram R Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011.

[HO97] SL Hary and F Ozguner. Feasibility test for real-time communication using wormhole routing. *IEE Proceedings-Computers and Digital Techniques*, 144(5):273–278, 1997.

[HVS+07] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, 2007.

[IG13] Vaishali V Ingle and Mahendra A Gaikwad. Review of mesh topology of noc architecture using source routing algorithms. *International Journal of Computer Applications*, pages 30–34, 2013.

[KHM87] Mark Karol, Michael Hluchyj, and Samuel Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, 35(12):1347–1356, 1987.

[KJS$^+$02] Shashi Kumar, Axel Jantsch, J-P Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112. IEEE, 2002.

[KKHL98] Byungjae Kim, Jong Kim, Sungje Hong, and Sunggu Lee. A real-time communication method for wormhole switching networks. In *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, pages 527–534. IEEE, 1998.

[KND02] Faraydon Karim, Anh Nguyen, and Sujit Dey. An interconnect architecture for networking systems on chips. *IEEE micro*, 22(5):36–45, 2002.

[KPN$^+$05] Jongman Kim, Dongkook Park, Chrysostomos Nicopoulos, Narayanan Vijaykrishnan, and Chita R Das. Design and analysis of an noc architecture from performance, reliability and energy perspective. In *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 173–182. ACM, 2005.

[LBBN16] Meng Liu, Matthias Becker, Moris Behnam, and Thomas Nolte. Using segmentation to improve schedulability of real-time packets on nocs with mixed traffic. In *The 14th International Workshop on Real-Time Networks*, July 2016.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[Lee03] Sunggu Lee. Real-time wormhole channels. *Journal of Parallel and Distributed Computing*, 63(3):299–311, 2003.

[LJS05] Zhonghai Lu, Axel Jantsch, and Ingo Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 960–964. IEEE, 2005.

[LRV06] Anthony Leroy, Frédéric Robert, and Diederik Verkest. Optimizing the on-chip communication architecture of low power systems-on-chip in deep sub-micron technology. 2006.

[MMM$^+$03] Fernando Gehm Moraes, Aline Mello, Leandro Möller, Luciano Ost, and Ney Laert Vilar Calazans. A low area overhead packet-switched network on chip: Architecture and prototyping. In *VLSI-SOC*, pages 318–323, 2003.

[MNTJ04] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 890–895. IEEE, 2004.

[Moh98] Prasant Mohapatra. Wormhole routing techniques for directly connected multicomputer systems. *ACM Computing Surveys (CSUR)*, 30(3):374–410, 1998.

[MTCM05] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. Virtual channels in networks on chip: implementation and evaluation on hermes noc. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183. ACM, 2005.

[Mut94] Matt W Mutka. Using rate monotonic scheduling technology for real-time communications in a wormhole network. In *Parallel and Distributed Real-Time Systems, 1994. Proceedings of the Second Workshop on*, pages 194–199. IEEE, 1994.

[NM93] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[NYP$^+$14a] Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, José Carlos Fonseca, Marko Bertogna, Eduardo Quiñones, Roberto Vargas, and Andrea Marongiu. The Challenge of Time-Predictability in Modern Many-Core Architectures. In *14th Intl. Workshop on Worst-Case Execution Time Analysis*, pages 63–72, Madridr, Spain, July 2014.

[NYP14b] Borislav Nikolić, Patrick Meumeu Yomsi, and Stefan M Petters. Worst-case communication delay analysis for many-cores using a limited migrative model. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

[OHM05] Umit Y Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in noc design: a holistic perspective. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74. ACM, 2005.

[PABB05] Antonio Pullini, Federico Angiolini, Davide Bertozzi, and Luca Benini. Fault tolerance overhead in network-on-chip flow control schemes. In *2005 18th Symposium on Integrated Circuits and Systems Design*, pages 224–229. IEEE, 2005.

[PJ06] Sandro Penolazzi and Axel Jantsch. A high level power model for the nostrum noc. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 673–676. IEEE, 2006.

[PSG$^+$14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study: from simulink specification to multi/many-core execution. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318. IEEE, 2014.

[QLD10] Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for on-chip packet-switching networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(5):802–815, 2010.

[RDG$^+$04] Andrei Radulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 878–883. IEEE, 2004.

[RGR$^+$03] Edwin Rijpkema, Kees Goossens, Andrei Radulescu, John Dielissen, Jef van Meerbergen, Paul Wielage, and Erwin Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEE Proceedings-Computers and Digital Techniques*, 150(5):294–302, 2003.

[RI12] Adrian Racu and Leandro Soares Indrusiak. Using genetic algorithms to map hard real-time on noc-based systems. In *7th Intl. Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8, 2012.

[RMB$^+$09] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. A method for calculating hard qos guarantees for networks-on-chip. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 579–586. ACM, 2009.

[RMB$^+$13] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*, 62(3):452–467, 2013.

[SB08] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 161–170. IEEE Computer Society, 2008.

[Shi09] Zheng Shi. *Real-time communication services for networks on chip*. publisher not identified, 2009.

[SKH08] Erno Salminen, Ari Kulmala, and Timo D Hamalainen. Survey of network-on-chip proposals. *white paper, OCP-IP*, 1:13, 2008.

[SRM13] Dahule Suyog, Golhar Reetesh, and Ramteke Mangesh. The behavior of round robin arbiter in noc architecture. *International Journal of Engineering and Innovative Technology (IJEIT)*, 3(5):312–314, 2013.

[STAN04] David Sigüenza-Tortosa, Tapani Ahonen, and Jari Nurmi. Issues in the development of a practical noc: the proteo concept. *Integration, the VLSI Journal*, 38(1):95–105, 2004.

[Til11] Tilera corporation. *Tile processor user architecture manual*, November 2011. UG101.

[TSSJ14] Konstantinos Tatas, Kostas Siozios, Dimitrios Soudris, and Axel Jantsch. *Designing 2D and 3D network-on-chip architectures*. Springer, 2014.

[WL03] Daniel Wiklund and Dake Liu. Socbus: switched network on chip for hard real time embedded systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[YGSP12] Bo Yang, Liang Guang, Tero Säntti, and Juha Plosila. Tree-model based contention-aware task mapping on many-core networks-on-chip. *Communications in Information Science and Management Engineering*, 2012.

[ZM12] Christopher Zimmer and Frank Mueller. Low contention mapping of real-time tasks onto tilepro 64 core processors. In *18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 131–140, 2012.