CISTER

Research Center in

Real-Time & Embedded

Computing Systems

PhD Thesis

Many-Core Platforms in the Real-Time Embedded Computing Domain

Borislav Nikolic

CISTER-TR-150603

Many-Core Platforms in the Real-Time Embedded Computing Domain

Borislav Nikolic

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Over the past few decades, the technological advancements made our lives increasingly permeated by and dependent on embedded systems. At the present day, these devices account for more than 98% of all produced computing systems, with applications that span over a wide range of areas, from medicine to avionics. Some embedded systems interact with the physical environment and have to guarantee not only that a certain action will be performed correctly, but also that the action will complete within a certain time. These devices are called real-time embedded systems, and some notable examples are medical pacemakers, airbags in cars and autopilots in airplanes.

The process of analysing the temporal behaviour of a real-time embedded system is calledreal-time analysis. In many cases, the purpose of the analysis is to derive guarantees that a devicewill perform its functions correctly, while at the same time meeting all timing requirements. Areal-time analysis is mostly performed at design-time, thus its efficiency highly depends on theamount of predictability of the entire system, whereas any non-deterministic aspect of the systembehaviour has to be accounted for in the analysis with a certain degree of pessimism. A pessimisticanalysis may cause a significant resource over-provisioning in the design phase, and consequentlylead to a severe underutilisation of available resources at runtime. Therefore, reducing the analysispessimism is one of the ever-present objectives in the real-time embedded computing domain.

The first real-time embedded systems were predominantly single-core devices with limitedsets of functionalities. However, constantly increasing demands for more advanced and sophisticatedfunctionalities required more powerful computational devices. When faced with the samechallenge, the other computing areas (e.g. general-purpose or high-performance computing) optedfor platforms consisting of several cores – multi-cores and more than a dozen of cores – manycores.It comes as no surprise that the same trends, although with an offset, are noticeable in theevolution of the real-time embedded systems, where many-core platforms present the new frontiertechnology.

Besides giving the options to implement more advanced functionalities, many-core platformsoffer other beneficial possibilities as well. For instance, multiple functionalities, that were previouslyimplemented on a set of single-core devices, can be integrated within fewer many-coreplatforms with significant design-cost reductions. Moreover, the abundance of available cores allowsto implement efficient thermal and power management strategies by deliberately performingtemporary shutdowns of idle cores. At the same time, the existence of idle cores, which can beused if necessary, makes these devices more resilient to hardware failures.Yet, despite the aforementioned benefits, the integration of many-cores into the real-time embeddeddomain is a big challenge. The most notable reasons are (i) increasingly complex designsof hardware components, promoting performance, often at the expense of predictability,and (ii) more significant and hard-to-analyse contention patterns for accesses to shared resources.These facts may contribute to a non-deterministic system behaviour, while, as explained above,every non-deterministic aspect of the system behaviour has to be accounted for in the real-timeanalysis with a certain degree of pessimism.

In this dissertation, the focus is on the analysis of real-time embedded systems deployed onmany-core platforms. Specifically, a comprehensive collection of techniques and design choicesis presented, with the common objective to make many-cores more amenable to the real-timeanalysis, and consequently more suitable and applicable to the real-time embedded domain. Theproposed methods achieve this end in several ways: (i) by extending the state-of-the-art approachesin order to reduce the analysis pessimism, (ii) by exploiting novel hardware features, as well asenforcing constraints which cause a more deterministic and analysable system behaviour, and(iii) by

elaborating on promising OS and workload paradigms, which have not been previouslyconsidered in the real-time embedded computing domain.

The contributions of this dissertation can be classified into two groups. In the first set ofcontributions the focus is on the interconnect medium, which is one of the most complex-toanalyseresources in many-core platforms. Initially, the target interconnect is the network-on-chipwith a 2-D mesh topology, which utilises the wormhole switching mechanism and the XY routingtechnique. For such a generic model, which is present in the most of contemporary many-cores,a novel worst-case communication delay analysis is proposed, and subsequently compared withthe state-of-the-art method. Then, assuming the additional hardware support in the form of virtualchannels, improvements over the state-of-the-art approaches are proposed, which, not only reducethe analysis pessimism, but also significantly reduce the requirements for hardware resources.Finally, a novel arbitration policy for NoC routers is proposed.

In the second set of contributions the focus is on a novel paradigm in the real-time embeddeddomain, called the Limited Migrative Model. This model is inspired by the latest trends in thehigh-performance and general-purpose computing. First, the model is introduced and the cost ofmaintaining it is analytically estimated, both in terms of computational and interconnect resources,where, for the later aspect, the findings from the first set of contributions are used (see the previousparagraph). Then, three aspects of the application workload are studied, namely: (i) communicationrequirements, (ii) memory requirements, and (iii) computation requirements. Thefirst aspect is addressed by imposing several constraints, which make the communication patternsmore predictable, and subsequently allow to derive a communication delay analysis. Moreover,the workload assignment to computational resources is investigated, but only from the communicationperspective, with the objective to spatially distribute the workload in such a way that alltiming constraints posed on communication delays are met. Then, the focus is shifted towards thememory requirements, and a set of analysis techniques are proposed, which can be used to checkwhether the memory traffic requirements are also fulfilled. In the final part, the computation requirementsof the application workload are studied. However, for this aspect only a coarse-grainedanalysis with several simplifying assumptions is presented. The proposed method represents aninitial step towards the complete analysis related to the computation requirements. Subsequently,assuming this initial analysis, the problem of the workload assignment to computational resourcesis revisited, but this time with an orthogonal objective, which is to assure that the computationalrequirements of the workload are fulfilled.

The findings suggest that the first set of contributions significantly improves over the state-ofthe-artmethods in the real-time analysis of interconnects. The improvements are manifested withthe reduced analysis pessimism, as well as reduced hardware requirements. Both these aspectsare essential for mitigating the resource over-provisioning effects when designing a new system.Additionally, the findings suggest that the Limited Migrative Model has a lot of potential, andrepresents a promising step towards the application of many-core platforms into the real-timeembedded computing domain

© CISTER Research Center
www.cister.isep.ipp.pt

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Many-Core Platforms in the Real-Time Embedded Computing Domain

Borislav Nikolić

Doctoral Programme in Electrical and Computer Engineering

Supervisor: Dr. Stefan Markus Ernst Petters

April 24, 2015

© Borislav Nikolić, 2015

Many-Core Platforms in the Real-Time Embedded Computing Domain

Borislav Nikolić

Doctoral Programme in Electrical and Computer Engineering

Approved by:

President: Dr. José Silva Matos
External Referee: Dr. Petru Eles
External Referee: Dr. Leandro Soares Indrusiak
Internal Referee: Dr. Luís Miguel Pinho
FEUP Referee: Dr. Pedro Ferreira do Souto
FEUP Referee: Dr. Mário Jorge Rodrigues de Sousa
Supervisor: Dr. Stefan Markus Ernst Petters

April 24, 2015

Abstract

Over the past few decades, the technological advancements made our lives increasingly permeated by and dependent on embedded systems. At the present day, these devices account for more than 98% of all produced computing systems, with applications that span over a wide range of areas, from medicine to avionics. Some embedded systems interact with the physical environment and have to guarantee not only that a certain action will be performed correctly, but also that the action will complete within a certain time. These devices are called *real-time embedded systems*, and some notable examples are medical pacemakers, airbags in cars and autopilots in airplanes.

The process of analysing the temporal behaviour of a real-time embedded system is called *real-time analysis*. In many cases, the purpose of the analysis is to derive guarantees that a device will perform its functions correctly, while at the same time meeting all timing requirements. A real-time analysis is mostly performed at design-time, thus its efficiency highly depends on the amount of predictability of the entire system, whereas any non-deterministic aspect of the system behaviour has to be accounted for in the analysis with a certain degree of pessimism. A pessimistic analysis may cause a significant resource over-provisioning in the design phase, and consequently lead to a severe underutilisation of available resources at runtime. Therefore, reducing the analysis pessimism is one of the ever-present objectives in the real-time embedded computing domain.

The first real-time embedded systems were predominantly single-core devices with limited sets of functionalities. However, constantly increasing demands for more advanced and sophisticated functionalities required more powerful computational devices. When faced with the same challenge, the other computing areas (e.g. general-purpose or high-performance computing) opted for platforms consisting of several cores – *multi-cores* and more than a dozen of cores – *many-cores*. It comes as no surprise that the same trends, although with an offset, are noticeable in the evolution of the real-time embedded systems, where many-core platforms present the new frontier technology.

Besides giving the options to implement more advanced functionalities, many-core platforms offer other beneficial possibilities as well. For instance, multiple functionalities, that were previously implemented on a set of single-core devices, can be integrated within fewer many-core platforms with significant design-cost reductions. Moreover, the abundance of available cores allows to implement efficient thermal and power management strategies by deliberately performing temporary shutdowns of idle cores. At the same time, the existence of idle cores, which can be used if necessary, makes these devices more resilient to hardware failures.

Yet, despite the aforementioned benefits, the integration of many-cores into the real-time embedded domain is a big challenge. The most notable reasons are (i) increasingly complex designs of hardware components, promoting performance, often at the expense of predictability, and (ii) more significant and hard-to-analyse contention patterns for accesses to shared resources. These facts may contribute to a non-deterministic system behaviour, while, as explained above, every non-deterministic aspect of the system behaviour has to be accounted for in the real-time analysis with a certain degree of pessimism.

In this dissertation, the focus is on the analysis of real-time embedded systems deployed on many-core platforms. Specifically, a comprehensive collection of techniques and design choices is presented, with the common objective to make many-cores more amenable to the real-time analysis, and consequently more suitable and applicable to the real-time embedded domain. The proposed methods achieve this end in several ways: (i) by extending the state-of-the-art approaches in order to reduce the analysis pessimism, (ii) by exploiting novel hardware features, as well as enforcing constraints which cause a more deterministic and analysable system behaviour, and (iii) by elaborating on promising OS and workload paradigms, which have not been previously considered in the real-time embedded computing domain.

The contributions of this dissertation can be classified into two groups. In the first set of contributions the focus is on the interconnect medium, which is one of the most complex-to-analyse resources in many-core platforms. Initially, the target interconnect is the network-on-chip with a 2-D mesh topology, which utilises the wormhole switching mechanism and the XY routing technique. For such a generic model, which is present in the most of contemporary many-cores, a novel worst-case communication delay analysis is proposed, and subsequently compared with the state-of-the-art method. Then, assuming the additional hardware support in the form of virtual channels, improvements over the state-of-the-art approaches are proposed, which, not only reduce the analysis pessimism, but also significantly reduce the requirements for hardware resources. Finally, a novel arbitration policy for NoC routers is proposed.

In the second set of contributions the focus is on a novel paradigm in the real-time embedded domain, called the *Limited Migrative Model*. This model is inspired by the latest trends in the high-performance and general-purpose computing. First, the model is introduced and the cost of maintaining it is analytically estimated, both in terms of computational and interconnect resources, where, for the later aspect, the findings from the first set of contributions are used (see the previous paragraph). Then, three aspects of the application workload are studied, namely: (i) communication requirements, (ii) memory requirements, and (iii) computation requirements. The first aspect is addressed by imposing several constraints, which make the communication patterns more predictable, and subsequently allow to derive a communication delay analysis. Moreover, the workload assignment to computational resources is investigated, but only from the communication perspective, with the objective to spatially distribute the workload in such a way that all timing constraints posed on communication delays are met. Then, the focus is shifted towards the memory requirements, and a set of analysis techniques are proposed, which can be used to check whether the memory traffic requirements are also fulfilled. In the final part, the computation requirements of the application workload are studied. However, for this aspect only a coarse-grained analysis with several simplifying assumptions is presented. The proposed method represents an initial step towards the complete analysis related to the computation requirements. Subsequently, assuming this initial analysis, the problem of the workload assignment to computational resources is revisited, but this time with an orthogonal objective, which is to assure that the computational requirements of the workload are fulfilled.

The findings suggest that the first set of contributions significantly improves over the state-of-the-art methods in the real-time analysis of interconnects. The improvements are manifested with the reduced analysis pessimism, as well as reduced hardware requirements. Both these aspects are essential for mitigating the resource over-provisioning effects when designing a new system. Additionally, the findings suggest that the Limited Migrative Model has a lot of potential, and represents a promising step towards the application of many-core platforms into the real-time embedded computing domain.

Resumo

Ao longo da últimas décadas, os avanços tecnológicos tornaram a vida das pessoas cada vez mais dependentes de sistemas embebidos. Actualmente, estes dispositivos representam mais de 98% de todos os sistemas computacionais produzidos e são usados numa grande variedade de áreas, desde a medicina até à aviação. Alguns destes sistemas embebidos interagem com o meio envolvente e têm que garantir não só que determinada acção é executada correctamente, mas também que essa acção é terminada dentro de um certo limite temporal. Este tipo de dispositivos são designados de sistemas embebidos de tempo real e alguns exemplos são os "pacemakers" usados na saúde, "airbags" nos automóveis e os pilotos automáticos nos aviões.

O processo de analisar o comportamento temporal de um sistema de embebido de tempo real é designado de análise de tempo real. Em muitos casos, o propósito da análise é garantir que o dispositivo vai executar correctamente e que as restrições temporais vão ser respeitadas. A análise de tempo real é essencialmente efectuada antes de execução, isto é, na fase de projecto. Deste modo, a sua eficiência depende muito do determinismo do sistema, uma vez que qualquer não determinismo resultará na introdução de pessimismo na análise. Uma análise pessimista pode causar um excessivo uso de recursos na fase de projecto, o que, geralmente, tem como consequência uma baixa utilização desses recursos em tempo de execução. Portanto, reduzir o pessimismo na análise é um dos objectivos prementes nesta área dos sistemas embebidos de tempo real.

Os primeiros sistemas embebidos de tempo real eram maioritariamente dispositivos com uma única unidade de execução (designados de "single-core") com um conjunto limitado de funcionalidades. Porém, a crescente necessidade de suportar funcionalidades cada vez mais avançadas e mais sofisticadas viria a exigir dispositivos com maior capacidade computacional. Noutras áreas da computação (como por exemplo, computação de alto desempenho e mesmo na computação usada sem nenhuma especificidade, isto é, de uso geral) este requisito de maior poder computacional foi abordado com recurso a sistemas compostos por várias unidades de execução (designados por "many-cores"). Consequentemente, e sem qualquer surpresa, a mesma estratégia tem vindo a ser seguida no domínio dos sistemas embebidos de tempo real, onde as plataformas "many-core" constituem o mais recente desafio tecnológico.

Além da possibilidade de implementar funcionalidades mais avançadas, as plataformas "many-core" oferecem outras possibilidades. Por exemplo, muitas funcionalidades que eram implementadas num conjunto de sistemas "single-core", podem ser integradas num conjunto mais reduzido de plataformas "many-core" com uma redução significativa dos custos de projecto. Para além disso, a abundância de unidades de execução permite implementar estratégias eficientes de gestão energética e térmica, desligando ou "adormecendo", temporariamente, as unidades de execução que não tenham nenhuma tarefa para executar. Ao mesmo tempo, a existência de unidades de execução "adormecidas", que podem ser "acordadas" sempre que necessário, torna estes dispositivos mais resilientes a falhas de hardware.

Contudo, apesar dos benefícios referidos anteriormente, a integração das plataformas "many-core" nos sistemas embebidos de tempo real apresenta-se como um grande desafio. As razões para

tal são (i) a cada vez maior complexidade dos componentes de hardware, que permite aumentar o desempenho, contudo, muitas vezes diminuindo a previsibilidade dos sistemas, e (ii) a crescente dificuldade na análise da contenção no acesso à utilização dos recursos partilhados. Estes factos podem contribuir para um comportamento menos determinista dos sistemas, o que, como referido anteriormente, implica adopção de níveis de pessimismo na analise.

O enfoque desta dissertação é a analise de sistemas embebidos de tempo real para plataformas "many-core". Especificamente, uma compreensiva colecção de técnicas e opções de projecto é apresentada, com o objectivo comum de tornar as plataformas "many-core" mais acessíveis (ou menos complexas) à analise de tempo real e consequentemente, mais apropriadas e aplicadas na área dos sistemas embebidos de tempo real. Os métodos propostos alcançam este objectivo de vários formas: (i) estendendo as abordagens existentes por forma a reduzir o pessimismo na analise, (ii) explorando novas características (ou elementos) de hardware e aplicando restrições que aumentem o determinismo e a analisabilidade dos sistemas e, por fim (iii) explorando novos paradigmas e funcionalidades presentes em alguns Sistemas Operativos e estratégias inovadoras ainda não consideradas no domínio dos sistemas computacionais de tempo real.

As contribuições apresentadas nesta dissertação podem ser classificadas em dois grupos. O primeiro grupo de contribuições está relacionada com o meio de interconexão, que é o mais complexo de analisar nas plataformas "many-core". Inicialmente, o meio de interconexão considerado foi o "Network-on-Chip" (NoC) com uma topologia em malha 2-D, que utiliza o mecanismo de comutação "wormhole" e a técnica de encaminhamento XY. Nesta dissertação é proposta uma nova análise para determinar o pior atraso nas comunicações para este modelo tão presente na maior parte das plataformas "many-cores" de hoje. Depois, assumindo suporte adicional de hardware na forma de canais virtuais, foram propostos melhoramentos às analises existentes, que além de reduzirem o pessimismo, também reduzem significamente os requisitos de recursos de hardware. Finalmente, foi também proposta uma nova politica de arbítrio para os encaminhadores NoC.

O segundo grupo de contribuições está relacionado com um novo paradigma no domínio dos sistemas embebidos de tempo real designado de "Limited Migrative Model". Este modelo é inspirado na tendência actual dos sistema computacionais de alto desempenho e de utilização geral. Em primeiro lugar, o modelo é descrito e o custo da sua manutenção é estimado analiticamente, quer em termos de recursos computacionais quer em termos de recursos de interconexão, sendo para este caso, utilizadas as primeiras contribuições apresentadas no último parágrafo. Depois, são estudados três requisitos relacionados com cada tarefa, nomeadamente requisitos de: (i) comunicações, (ii) memória e (iii) computacionais. Em relação ao primeiro requisito, é imposto um conjunto de restrições, que tornam os padrões de comunicação mais determinísticos e consequentemente permitem a derivação analítica do atraso das comunicações. Para além disso, e de forma a garatir que os requisitos temporais impostos às comunicações são cumpridos, a carga atribuída aos diversos recursos computacionais é tratada apenas da perspectiva das comunicações. Em seguida, o enfoque é alterado para a memória, e é proposto um conjunto de métodos de análise tendo como objectivo verificar se os requisitos são também aqui cumpridos. Por fim, são estudados os requisitos computacionais para uma determinada carga. Contudo, em relação a este último aspecto, a análise apresentada é bastante simplificada uma vez que se trata de uma primeira abordagem para uma análise mais completa. Assim, o problema a alocação de carga computacional é revisitado, desta vez com o objectivo ortogonal de garantir que também os requisitos computacionais são cumpridos.

Os resultados mostram que o primeiro grupo de contribuições melhora substancialmente o estado da arte na análise de tempo real de interconexões. Estas melhorias estão sobretudo patentes na redução do pessimismo da análise e também na diminuição dos requisitos de hardware. Estes dois aspectos revelam-se essenciais para mitigar os efeitos do sobre-provisionamento aquando do

projecto de um novo sistema. Para além disso, os resultados sugerem que o “Limited Migrative Model” possui bastante potencial e este apresenta-se como bastante promissor para a suportar a aplicação de plataformas “many-core” no domínio dos sistemas embebidos e de tempo real.

Acknowledgements

Working towards the Ph.D. Degree was the most demanding and challenging activity in my life. It took me four hard working years to finish all the cool things described in this dissertation. During that period, I was never alone, and I would like to express my most sincere gratitude to the people that made me feel this way:

- **Stefan M. Petters** – my Supervisor, for everything he has done for me, from the day when he selected me to be his Ph.D. student, until the day he finished the review of the last page of this dissertation. Stefan, thank you for always believing in me, for pushing me when I needed to be pushed, and for teaching me many valuable things, not only about research, but also about life.

- **Dobrica** and **Ivan Nikolić** – my parents, and **Özge Güngör** – my girlfriend, for giving me strength, courage and motivation to persist on this long journey, especially during the most challenging period, when I was writing this dissertation. Mum, dad, Özge, you are the best support that I could ever have, and with you by my side nothing is impossible.

- **Leandro Soares Indrusiak**, for giving me the opportunity to spend three months under his supervision, with the Real-Time Systems Group of the University of York, and for always being available and enthusiastic to discuss new ideas. Leandro, like we said several times already: "*This is just the beginning!*". I truly believe that.

- **Hazem Ismail Ali**, for being an excellent friend and colleague, and for always having words of encouragement, a priceless advice and a pack of Dutch waffles up his sleeve. Mr. President, from the bottom of my heart I wish you to reach this stage very soon, *mabrouk we 32bel 3endak*.

- **Patrick Meumeu Yomsi**, for joining my crew during the second half of my Ph.D. trip, and for becoming an invaluable part of it ever since. Patrick, you know which are my favourite publications. As I always say during CISTER Seminars: "*Mömö, you are the man!*".

- **Muhammad Ali Awan**, **Dakshina Dasari** and **Konstantinos Bletsas**, for very productive collaborations, as well as very interesting and enjoyable discussions, especially those not related to the work. "*Guys, I want to tell you a story...*".

- **Paulo Baltarejo Sousa** and **Ricardo Severino**, for translating the abstract into Portuguese. Paulo, Ricardo, muito obrigado pela vossa ajuda e força F.C. Porto!

- **Eduardo Tovar**, for creating an outstanding work environment in CISTER Research Center.

This work was partially supported by FCT (Fundação para a Ciência e a Tecnologia) under the individual doctoral grant SFRH/BD/81087/2011.

List of Publications

The following publications and technical reports have been developed in the scope of the research activities presented in this dissertation.

Journals (in chronological order)

- [27] Dakshina Dasari, Borislav Nikolić, Vincent Nélis and Stefan M. Petters, "NoC Contention Analysis using a Branch and Prune Algorithm", *ACM Transactions on Embedded Computing Systems (TECS) - Special Issue on Design Challenges for Many-Core Processors*, Volume 13, Issue 3s, Article number 113, March 2014.

- [70] Borislav Nikolić and Stefan M. Petters, "Real-Time Application Mapping for Many-Cores Using a Limited Migrative Model", *Real-Time Systems*, To appear.

Conferences (in chronological order)

- [73] Borislav Nikolić, Muhammad Ali Awan and Stefan M. Petters, "SPARTS: Simulator for Power Aware and Real-Time Systems", *In Proceedings of the 8th IEEE International Conference on Embedded Software and Systems (IEEE ICESS-11)*, pages 999-1004, Changsha, China, November 16-18, 2011.

- [68] Borislav Nikolić and Stefan M. Petters, "Towards Network-On-Chip Agreement Protocols", *In Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT 2012)*, pages 207-216, Tampere, Finland, October 7-12, 2012.

- [75] Borislav Nikolić, Patrick Meumeu Yomsi and Stefan M. Petters, "Worst-Case Memory Traffic Analysis for Many-Cores using a Limited Migrative Model", *In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2013)*, pages 42-51, Taipei, Taiwan, August 19-21, 2013.

- [74] Borislav Nikolić, Hazem Ismail Ali, Stefan M. Petters and Luís Miguel Pinho, "Are Virtual Channels the Bottleneck of Priority-Aware Wormhole-Switched NoC-Based Many-Cores?", *In Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 13-22, Sophia Antipolis, France, October 16-18, 2013.

- [76] Borislav Nikolić, Patrick Meumeu Yomsi and Stefan M. Petters, "Worst-Case Communication Delay Analysis for Many-Cores using a Limited Migrative Model", *In Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2014)*, pages 1-10, Chongqing, China, August 20-22, 2014. **Short-listed for the best paper award and invited for the journal extension.**

- [69] Borislav Nikolić and Stefan M. Petters, "EDF as an Arbitration Policy for Wormhole-Switched Priority-Preemptive NoCs – Myth or Fact?", *In Proceedings of the 12th ACM International Conference on Embedded Software (EMSOFT 2014)*, pages 1-10, New Delhi, India, October 12-17, 2014.

Technical Reports

- [71] Borislav Nikolić, Konstantinos Bletsas and Stefan M. Petters, "Priority Assignment and Application Mapping for Many-Cores Using a Limited Migrative Model", 2013.
- [72] Borislav Nikolić, Leandro Soares Indrusiak and Stefan M. Petters, "A Tighter Real-Time Communication Analysis for Wormhole-Switched Priority-Preemptive NoCs", 2014.

“The dawn cannot come before the morning.”

Nebojša Čović

“If you do not believe in miracles, then miracles will not happen to you.”

Dejan Radonjić

Contents

Abstract i

Acknowledgements vii

List of Publications ix

1 Introduction 1
- 1.1 Real-Time Embedded Systems . . . 1
- 1.2 Real-Time Analysis . . . 2
- 1.3 Single-Core ⇒ Multi-Core ⇒ Many-Core . . . 3
- 1.4 Benefits of Many-Cores . . . 4
- 1.5 Many-Cores in Real-Time Embedded Computing Domain . . . 4
  - 1.5.1 Computation Process . . . 4
  - 1.5.2 Interconnect medium . . . 9
  - 1.5.3 Data Input and Output . . . 27
  - 1.5.4 Recapitulation . . . 32
- 1.6 Thesis Statement . . . 34

2 Several Steps Closer to Real-Time NoCs 35
- 2.1 Traffic Model . . . 35
- 2.2 NoCs with the Round-Robin Arbitration Policy . . . 36
  - 2.2.1 State-of-the-Art Method . . . 36
  - 2.2.2 Analysis Pessimism . . . 40
  - 2.2.3 Branch and Prune (BP) Method . . . 43
  - 2.2.4 Branch, Prune and Collapse (BPC) - More Efficient Method . . . 48
  - 2.2.5 Experimental Evaluation . . . 52
  - 2.2.6 Discussion . . . 59
- 2.3 NoCs with Priority-Preemptive Arbitration Policies . . . 59
  - 2.3.1 State-of-the-Art Method for NoCs with Fixed-Priority Arbitration Policy 61
  - 2.3.2 Priority-Share Policy . . . 63
  - 2.3.3 Relaxing Hardware Requirements . . . 65
  - 2.3.4 EDF as Arbitration Policy . . . 78
  - 2.3.5 Reducing Analysis Pessimism . . . 93

3 Limited Migrative Model - LMM 107
- 3.1 LMM in Detail . . . 107
  - 3.1.1 Operating System (OS) . . . 107
  - 3.1.2 Application Layer . . . 108
  - 3.1.3 LMM Benefits . . . 109

3.2 Application Workload . . . 110
3.3 Agreement Protocols . . . 111
3.3.1 Master-Slave Protocol . . . 112
3.3.2 List Protocol . . . 120
3.3.3 Hybrid Protocol . . . 125
3.3.4 Experimental Evaluation . . . 132
3.4 Inter-application Communication . . . 136
3.5 Towards More Deterministic Communication Patterns . . . 139
3.5.1 Supermessages and Proxies . . . 140
3.5.2 Maximum Number of Message Occurrences . . . 142
3.5.3 Performing the Analysis . . . 148
3.5.4 Discussion . . . 150
3.5.5 Experimental Evaluation . . . 151
3.6 Application Mapping . . . 159
3.6.1 Problem Statement . . . 160
3.6.2 Mapping Quality . . . 160
3.6.3 Mapping Process . . . 163
3.6.4 Experimental Evaluation . . . 170
3.6.5 Discussion . . . 178
3.7 Memory Traffic . . . 179
3.7.1 Workload . . . 179
3.7.2 Challenges and Inapplicability of Existing Techniques . . . 179
3.7.3 Access Constraints and Bounding Messages . . . 180
3.7.4 Solution to Mutually Exclusive Bounding Messages . . . 182
3.7.5 Approach One: Per-Packet Analysis . . . 182
3.7.6 Intermediate Step: Partial Per-Pattern Analysis . . . 183
3.7.7 Approach Two: Full per-pattern analysis . . . 184
3.7.8 Experimental Evaluation . . . 185
3.8 Computation . . . 189
3.8.1 Core Shutdowns . . . 189
3.8.2 Workload . . . 190
3.8.3 Problem Statement . . . 191
3.8.4 Offline Schedulability Guarantees . . . 191
3.8.5 Online Schedulability . . . 192
3.8.6 Semi-Schedulability Guarantees . . . 193
3.8.7 Blind Synchronisation . . . 197
3.8.8 Parent-Child Relationship . . . 198
3.8.9 Schedulability and Agreement Protocols . . . 198
3.8.10 Priority Assignment and Mapping . . . 199
3.8.11 Experimental Evaluation . . . 201
3.8.12 Discussion . . . 207

4 Conclusions and Future Work **209**

Appendix **211**

References **215**

List of Figures

1.1 Fully partitioned scheduling . . . 5
1.2 Semi-partitioned scheduling . . . 6
1.3 Global scheduling . . . 7
1.4 Clustered scheduling . . . 8
1.5 Point-to-point communication . . . 9
1.6 Network-on-Chip communication . . . 10
1.7 Different NoC topologies . . . 11
1.8 Packet transfer over NoC . . . 12
1.9 Minimal dimension-ordered routing algorithms . . . 13
1.10 Simplified 2-D mesh NoC router architecture . . . 14
1.11 Store-and-forward switching . . . 14
1.12 Wormhole switching . . . 15
1.13 Credit-based flow control mechanism . . . 16
1.14 Packet traversals without contentions . . . 17
1.15 Packet traversals with contentions . . . 18
1.16 Contending packets . . . 18
1.17 Indirect contentions . . . 20
1.18 Indirect interference with virtual channels . . . 22
1.19 Packet preemptions . . . 23
1.20 Assumed NoC platform . . . 26
1.21 Memory system of single-core devices . . . 28
1.22 Memory system of multi-core devices . . . 29
1.23 Assumed memory system . . . 32

2.1 Traffic flows (example 1) . . . 36
2.2 Traffic flows (example 2) . . . 37
2.3 Computation tree for flow f_2 from Figure 2.2, in isolation (Ferrandiz et al. [32] method) . . . 39
2.4 Computation tree for flow f_2 from Figure 2.2 (Ferrandiz et al. [32] method) . . . 40
2.5 Computation subtrees for Figure 2.4 . . . 40
2.6 Traversal of packets from two adjacent periods . . . 41
2.7 Traversal of packets from several successive periods . . . 42
2.8 Computation tree for flow f_2 from Figure 2.2 (BP method) . . . 44
2.9 Computation subtrees constructed after branching in Figure 2.8 . . . 44
2.10 Traffic flows (revisited example 2) . . . 50
2.11 Distribution of WCTT improvement across flows (legends represent improvement ranges) 54
2.12 BPC method with varying SIRL vs. Ferrandiz et al. [32] method . . . 55
2.13 Inter-SIRL ratios . . . 56

2.14 All-to-one contending flows . . . 57
2.15 Impact of MIRT on flow f_1 . . . 58
2.16 Traffic flows (example 3) . . . 61
2.17 Deadline miss for flow-set from Figure 2.16 and Table 2.2 . . . 62
2.18 Assignment of virtual channels to flow-set with distinctive priorities . . . 66
2.19 Assignment of virtual channels to flow-set with priority-share policy . . . 66
2.20 Per-router assignment of virtual channels to flow-set with distinctive priorities . . . 66
2.21 Core selection methodology proposed by Ali et al. [4] . . . 70
2.22 Virtual channels do scale with respect to number of flows . . . 75
2.23 Additional virtual channels do **not** help in schedulability guarantees . . . 76
2.24 Link bandwidths are bottlenecks . . . 77
2.25 Traffic flows (example 4) . . . 80
2.26 Deadline miss for flow-set from Figure 2.25 and Table 2.5 . . . 81
2.27 Chain of dependencies for Figure 2.25 . . . 83
2.28 Traffic flows (example 5) . . . 86
2.29 EDF vs RM . . . 90
2.30 EDF vs HSA . . . 90
2.31 EDF vs RM,HSA (1-hop) . . . 91
2.32 Analysis time variations . . . 92
2.33 Influence of Δ on EDF . . . 92
2.34 Traffic flows (example 6) . . . 94
2.35 Detailed interference analysis . . . 95
2.36 Traffic flows (example 7) . . . 99
2.37 Traffic flows (example 8) . . . 100
2.38 Improvement in the worst-case traversal time of f_2, for two contending flows f_1 and f_2, where $P(f_1) > P(f_2)$, $|\mathscr{L}(f_1)| = |\mathscr{L}(f_2)|$, and f_1 preempts f_2 only once . 101
2.39 Improvements wrt flow sizes . . . 103
2.40 Improvements wrt path sizes . . . 103
2.41 Improvements wrt flow-set sizes . . . 104
2.42 Improvements wrt flow and path sizes . . . 104
2.43 Improvements wrt priorities . . . 105
2.44 Analysis tightness . . . 105

3.1 Limited Migrative Model . . . 108
3.2 Example of application's computation, memory access and communication patterns 111
3.3 Agreement protocol messages are master-dependent . . . 114
3.4 Analysis pessimism . . . 134
3.5 Impact of dispatchers on WCPD . . . 135
3.6 Inter-application communication . . . 137
3.7 Application which shape and messages comply with Definitions 3-4 . . . 140
3.8 Supermessages for different application shapes . . . 140
3.9 Inter-application communication . . . 142
3.10 Intra-application messages . . . 143
3.11 Analysis improvements (1/2) . . . 153
3.12 Analysis improvements (2/2) . . . 154
3.13 Performance comparison (1/2) . . . 155
3.14 Performance comparison (2/2) . . . 156
3.15 Performance comparison for Hybrid protocol . . . 157
3.16 Analysis tightness across number of dispatchers . . . 157

3.17 Analysis tightness . . . 158
3.18 Different dispatcher mappings for the same application-shape . . . 162
3.19 Shape comparison . . . 171
3.20 Rerouting penalty estimation . . . 171
3.21 Influence of application-set size on analysis time (1/2) . . . 173
3.22 Influence of application-set size on analysis time (2/2) . . . 174
3.23 Influence of grid size on analysis time . . . 175
3.24 Influence of parameter P on mapping process . . . 176
3.25 Influence of parameter G on mapping process . . . 177
3.26 Memory operations . . . 179
3.27 Master volatility problem for memory operations . . . 180
3.28 Memory operations via proxy dispatchers . . . 180
3.29 Bounding messages . . . 181
3.30 Analyses comparison . . . 186
3.31 Improvement with single controller . . . 187
3.32 Different memory operations . . . 187
3.33 Full per-pattern vs. per-packet . . . 188
3.34 Penalty of multiple controllers . . . 188
3.35 Semi-schedulability . . . 194
3.36 Non-synchronised semi-schedulability . . . 195
3.37 Future release schedulability . . . 196
3.38 Priority assignment upon dispatchers . . . 199
3.39 Speculative mapping . . . 200
3.40 Impact of RTA priorities and SS on schedulability guarantees . . . 203
3.41 Impact of number of dispatchers on schedulability guarantees . . . 203
3.42 Impact of mapping strategies on schedulability guarantees . . . 204
3.43 Efficiency of online tests when using remaining computation times . . . 204
3.44 Efficiency of online tests without remaining computation times . . . 205
3.45 Impact of number of dispatchers on runtime behaviour, without core shutdowns . . . 205
3.46 Impact of number of dispatchers on runtime behaviour, with core shutdowns . . . 206
3.47 Impact of mapping strategies on runtime behaviour, without core shutdowns . . . 206
3.48 Impact of mapping strategies on runtime behaviour, with core shutdowns . . . 207
3.49 Blind synchronisation mode (BSM) . . . 207

4.1 General form of Bordered Hessian for n variables and one constraint . . . 213
4.2 Bordered Hessian for $\lambda = 0$ . . . 213
4.3 Bordered Hessian for $\lambda \neq 0$ . . . 213

List of Tables

1 List of symbols used in this dissertation (1/6) xxi
2 List of symbols used in this dissertation (2/6) xxii
3 List of symbols used in this dissertation (3/6) xxiii
4 List of symbols used in this dissertation (4/6) xxiv
5 List of symbols used in this dissertation (5/6) xxv
6 List of symbols used in this dissertation (6/6) xxvi

1.1 List of symbols related to the assumed platform 33

2.1 Analysis parameters for Section 2.2.5 53
2.2 Flow-set parameters for Figure 2.16 (example 1) 62
2.3 Flow-set parameters for Figure 2.16 (example 2) 64
2.4 Analysis parameters for Section 2.3.3.5 74
2.5 Flow-set parameters for Figure 2.25 80
2.6 Flow-set parameters for two contending flows 86
2.7 Flow-set parameters for Figure 2.28 87
2.8 Comparison of approaches (schedulability) 88
2.9 Analysis parameters for Section 2.3.4.5 89
2.10 Flow-set parameters for Figure 2.34 (example 1) 98
2.11 Flow-set parameters for Figure 2.34 (example 2) 100
2.12 Analysis parameters for Section 2.3.5.5 102

3.1 OS operations related to agreement protocols 112
3.2 Analysis and simulation parameters for Section 3.3.4 133
3.3 OS operations related to inter-application communication 136
3.4 Analysis and simulation parameters for Section 3.5.5 152
3.5 Analysis and simulation parameters for Section 3.6.4 172
3.6 Analysis parameters for Section 3.7.8 185
3.7 Speculative mapping computation for Figure 3.39 201
3.8 Analysis and simulation parameters for Section 3.8.11 202

List of Symbols

Table 1: List of symbols used in this dissertation (1/6)

	Symbol	Description
HARDWARE	Ψ	The assumed platform.
	Π	The set of processing elements (cores) of Ψ.
	z	The number of cores in Π.
	π_i	The ith core of Π, where $i \in \{1, ..., z\}$.
	Δ	The maximum clock skew between any two cores of Π.
	η	A generic 2-D mesh NoC interconnect of Ψ.
	η_{RR}	η with the round-robin arbitration policy and a single channel.
	η_{PP}	η with the priority-preemptive arbitration policy and virtual channels.
	x	The horizontal dimension of η, where $x \cdot y = z$.
	y	The vertical dimension of η, where $x \cdot y = z$.
	ρ	The set of routers of η.
	ρ_i	The ith router of ρ, where $i \in \{1, ..., z\}$.
	λ	The set of links of η dedicated to the communication traffic.
	$\lambda_{i,j}$	The communication link from ρ_i (source) to ρ_j (destination).
	$\lambda_{c,i}$	The communication link from π_i (source) to ρ_i (destination).
	$\lambda_{i,c}$	The communication link from ρ_i (source) to π_i (destination).
	σ_{flit}	The size of the flit and the width of each link of λ.
	$\widehat{v}_{lim}$	The number of virtual channels of η.
	δ_L	The time it takes a flit to traverse a link.
	δ_ρ	The time it takes a header flit to traverse a router.
	δ_R	The time it takes to reroute one packet.
	ν_ρ	The frequency of routers.

Table 2: List of symbols used in this dissertation (2/6)

	Symbol	Description
HARDWARE	B_{link}	Link bandwidth, $B_{link} = \frac{\sigma_{flit}}{\delta_p + \delta_L}$.
	Ξ	The non-coherent memory system of Ψ.
	μ	The set of memory controllers of Ξ.
	μ_i	The ith controller of μ, where $i \in \{1,2,3,4\}$.
	κ	The set of cache memories of Ξ.
	κ_i	The ith cache memory of κ, where $i \in \{1,...,z\}$.
	λ^M	The set of links of η dedicated to the memory traffic.
	$\lambda^M_{i,j}$	The memory link from ρ_i (source) to ρ_j (destination).
	$\lambda^M_{c,i}$	The memory link from π_i (source) to ρ_i (destination).
	$\lambda^M_{i,c}$	The memory link from ρ_i (source) to π_i (destination).
	$\lambda^M_{m,i}$	The memory link from the memory controller (source) to ρ_i (destination).
	$\lambda^M_{i,m}$	The memory link from ρ_i (source) to the memory controller (destination).
OS	$\delta^{\rightarrow}_P$	The time it takes to send the protocol message.
	$\delta^{\leftarrow}_P$	The time it takes to receive the protocol message.
	$\delta^{\rightarrow}_C$	The time it takes to send the execution context.
	$\delta^{\leftarrow}_C$	The time it takes to receive the execution context.
	δ_Q	The time it takes to get the info. from the kernel about the next job release.
	δ_E	The time it takes to elect the next master, or generate the sorted master list.
	$\delta^{\rightarrow}_I$	The time it takes to send the inter-application message.
	$\delta^{\leftarrow}_I$	The time it takes to receive the inter-application message.
	$\widehat{M}$	The maximum number of concurrent masters on one core.
	K	The maximum number of concurrent core shutdowns.
SOFTWARE	$\mathscr{F}$	The set of flows.
	w	The number of flows in $\mathscr{F}$.
	f_i	The ith flow of $\mathscr{F}$, where $i \in \{1,...,w\}$.
	$src(f_i)$	The source of f_i.
	$dst(f_i)$	The destination of f_i.
	$\mathscr{L}(f_i)$	The set of links of λ which f_i traverses.
	$\sigma(f_i)$	The size of f_i.

Table 3: List of symbols used in this dissertation (3/6)

	Symbol	Description
SOFTWARE	$T(f_i)$	The minimum inter-arrival period of f_i.
	$D(f_i)$	The deadline of f_i.
	$P(f_i)$	The priority of f_i.
	$C(f_i)$	The traversal time of f_i, in isolation.
	$J_R(f_i)$	The release jitter of f_i.
	$J_N(f_i)$	The network jitter of f_i.
	$WCTT(f_i)$	The analytically computed worst-case traversal time of f_i.
	$WCTT^*(f_i)$	The exact worst-case traversal time of f_i.
	$U(f_i)$	The utilisation of $\mathscr{L}(f_i)$.
	$\mathscr{F}_D(f_i)$	The set of flows directly contending with f_i.
	$\mathscr{F}_I(f_i)$	The set of flows indirectly contending with f_i.
	$W(f_i)$	The busy period of f_i.
	$\mathscr{T}_{crit}(f_i)$	The set of critical instants of f_i within $W(f_i)$.
	$L(f_i,t)$	The abs. analytically computed worst-case traversal time of f_i released at t.
	$WCTT(f_i,t)$	The analytically computed worst-case traversal time of f_i released at t.
	$\mathring{f}$	The composite flow.
	$\mathscr{F}_C(\mathring{f})$	The set of flows constituting $\mathring{f}$.
	$I(f_i \rightarrow f_j)$	The interference that a single packet of f_i causes to f_j.
	$\mathscr{L}_{i,j}^{pre-CD}$	The part of $\mathscr{L}(f_i)$ before f_i and f_j start contending.
	$\mathscr{L}_{i,j}^{CD}$	The part of $\mathscr{L}(f_i)$ where f_i and f_j contend.
	$\mathscr{L}_{i,j}^{post-CD}$	The part of $\mathscr{L}(f_i)$ after f_i and f_j stop contending.
	$\gamma_{i,j}^{pre-CD}$	The time it takes a header flit of f_i to traverse $\mathscr{L}_{i,j}^{pre-CD}$.
	$\gamma_{i,j}^{post-CD}$	The time it takes a tail flit of f_i to traverse $\mathscr{L}_{i,j}^{post-CD}$.
	$WCTT^+(f_i)$	The tighter analytically computed worst-case traversal time of f_i.
	F	The set of functionalities.
	F_i	The ith functionality of F, where $i \in \{1,...,z\}$.
	$\mathscr{F}_S(F_i)$	The set of flows sent by F_i.
	$\mathscr{F}_R(F_i)$	The set of flows received by F_i.
	$\mathscr{M}$	The mapping of the workload onto the platform.

Table 4: List of symbols used in this dissertation (4/6)

	Symbol	Description
SOFTWARE	$\widehat{v}(\mathscr{M})$	The minimum number of virtual channels required by $\mathscr{M}$.
	$\widehat{S}(\mathscr{M})$	The fraction of flow-sizes for which $\mathscr{M}$ is schedulable, $0 < \widehat{S}(\mathscr{M}) \leq 1$.
	$\mathscr{A}$	The set of applications.
	v	The number of applications is $\mathscr{A}$.
	Q	The quality of $\mathscr{A}$.
	$\mathscr{A}_H$	The subset of $\mathscr{A}$, representing the high-priority applications.
	$\mathscr{A}_L$	The subset of $\mathscr{A}$, representing the high-priority applications.
	P	The parameter which decides the breakdown of $\mathscr{A}$ on $\mathscr{A}_H$ and $\mathscr{A}_L$.
	G	The parameter which decides the allowed shape sizes for $\mathscr{A}_H$.
	a_i	The ith application of $\mathscr{A}$, where $i \in \{1,...,v\}$.
	$P(a_i)$	The priority of a_i.
	$M(a_i)$	The migration coefficient of a_i.
	q_i	The quality of the current mapping of a_i.
	$S^{min}(a_i)$	The surface of the narrow mapping of a_i.
	$S^{max}(a_i)$	The surface of the wide mapping of a_i.
	$T(a_i)$	The minimum inter-arrival period of a_i.
	$D(a_i)$	The deadline of a_i.
	$\mathscr{D}(a_i)$	The set of dispatchers of a_i.
	$C^{\tau}(a_i)$	The computation requirements of a_i.
	$D^{\tau}(a_i)$	The computation deadline of a_i.
	$D^{\mu}(a_i)$	The memory traffic deadline of a_i.
	$D^{\tau+\mu}(a_i)$	The sum of $D^{\tau}(a_i)$ and $D^{\mu}(a_i)$.
	$D^{\eta}(a_i)$	The communication deadline of a_i.
	$\mathscr{F}^{\mu}(a_i)$	The set of flows related to the memory operations of a_i.
	$\mathscr{F}^{\eta}(a_i)$	The set of flows related to the intra- and inter-application comm. of a_i.
	$\mathscr{F}^{\eta}_D(a_i)$	The set of directly interfering flows of any flow from $\mathscr{F}^{\eta}(a_i)$.
	$\mathscr{F}^{\eta}_S(a_i)$	The set of inter-application messages sent by a_i.
	$\mathscr{F}^{\eta}_R(a_i)$	The set of inter-application messages received by a_i.
	$\widehat{\mathscr{F}}(a_i)$	The set of supermessages of a_i.

Table 5: List of symbols used in this dissertation (5/6)

	Symbol	Description
SOFTWARE	$\widehat{f}_i^j$	The jth supermessage of $\widehat{\mathscr{F}}(a_i)$.
	$\mathscr{F}^P(a_i)$	The set of inter-proxy messages of a_i.
	$f_{i,j}^P$	The inter-proxy message of $\mathscr{F}^P(a_i)$.
	$\overline{\mathscr{F}}(a_i)$	The set of bounding messages belonging to a_i.
	$\overline{f_i}$	The bounding message.
	$\mathscr{F}_E(\overline{f_i})$	The set of mutually exclusive bounding messages of $\overline{f_i}$, including $\overline{f_i}$.
	$\mathscr{F}_R(\overline{f_i})$	The reduced set of interfering messages of $\overline{f_i}$.
	$\overline{\mathscr{F}}_R(a_i)$	The reduced set of bounding messages belonging to a_i.
	$\omega(f_i)$	The max. num. of occurrences of f_i during one period of its application.
	$\omega_P(f_i)$	The max. num. of occurrences of f_i during one protocol execution.
	$\omega_C(f_i)$	The max. num. of occurrences of f_i during one context transfer.
	$\omega_I(f_i)$	The max. num. of occurrences of f_i during inter-application communication.
	$C_P(\widehat{f}_i^j)$	The traversal time of $\widehat{f}_i^j$ related to the protocol execution of a_i, in isolation.
	$C_C(\widehat{f}_i^j)$	The traversal time of $\widehat{f}_i^j$ related to the context transfer of a_i, in isolation.
	$C_I(\widehat{f}_i^j)$	The traversal time of $\widehat{f}_i^j$ related to inter-application comm. of a_i, in isolation.
	$C_M^\eta(a_i)$	The delay of comm.-related OS ops of the current master of a_i, in isolation.
	$C_S^\eta(a_i)$	The delay of comm.-related OS ops of the slave of a_i, in isolation.
	$C_N^\eta(a_i)$	The delay of comm.-related OS ops of the next master of a_i, in isolation.
	$C_\#^\eta(a_i)$	The delay of communication traffic of a_i, in isolation.
	$C^\eta(a_i)$	The communication delay of a_i, in isolation.
	$I_\#^\eta(a_i,t)$	The worst-case network interference that a_i can suffer within t.
	$R^\eta(a_i)$	The analytically computed worst-case communication delay of a_i.
	$R_*^\eta(a_i)$	The worst-case communication delay of a_i obtained via simulations.
	$C_R^\eta(a_i)$	The rerouting delay of the communication traffic of a_i, in isolation.
	$C_{RP}^\eta(a_i)$	The rerouting delay of one protocol execution of a_i, in isolation.
	$C_{RS}^\eta(a_i)$	The rerouting delay of sent inter-application traffic of a_i, in isolation.
	$C_{RR}^\eta(a_i)$	The rerouting delay of received inter-application traffic of a_i, in isolation.
	$C_{\#P}^\eta(a_i)$	The delay of protocol-related traffic of a_i, in isolation.
	$C_{\#S}^\eta(a_i)$	The delay of sent inter-application traffic of a_i, in isolation.

Table 6: List of symbols used in this dissertation (6/6)

	Symbol	Description
SOFTWARE	$C^{\eta}_{\#R}(a_i)$	The delay of received inter-application traffic of a_i, in isolation.
	$I^{\eta}_{R}(a_i,t)$	The worst-case rerouting interference that a_i can suffer within t.
	$R^{\mu}(a_i)$	The analytically computed worst-case memory traffic delay of a_i.
	$maxhops(a_i)$	The max. distance (in hops) between any two dispatchers of a_i.
	$maxhops(a_i,a_j)$	The max. distance (in hops) between any two dispatchers of a_i and a_j.
	σ_{prt}	The size of the protocol message.
	σ_{ctx}	The size of the execution context message.
	σ_{iam}	The size of the inter-application message.
	d_i^j	The jth dispatcher of a_i.
	$P(d_i^j)$	The priority of d_i^j.
	$\pi(d_i^j)$	The core of d_i^j.
	$\rho(d_i^j)$	The rerouter connected to $\pi(d_i^j)$.
	$\mathscr{D}_{\pi(d_i^j)}$	The set of dispatchers residing on $\pi(d_i^j)$, including d_i^j.
	$I^{\eta}(d_i^j,t)$	The worst-case on-core interference that d_i^j can suffer within t.
	$r(d_i^j)$	The max. num. of reroutings of $\rho(d_i^j)$ due to d_i^j, during one period of a_i.
	$r_P(d_i^j)$	The max. num. of reroutings on $\rho(d_i^j)$ due to d_i^j, during one protocol of a_i.
	$r_I(d_i^j)$	The max. num. of reroutings on $\rho(d_i^j)$ due to d_i^j, during inter-app. comm. of a_i.

Chapter 1

Introduction

In this chapter, the context and motivation for the research activities conducted in the scope of this dissertation are provided.

1.1 Real-Time Embedded Systems

Over the course of the last few decades, the technological advancements allowed dramatic improvements in the production process of electronic devices. As the possibilities of electronic systems continue to grow, we delegate to them more and more of our daily activities. Nowadays, without any exaggeration, we can say that our lives are dependant on electronic devices, and that these systems have become an integral part of our life.

In many cases, electronic devices are designed to perform only a specific set of functions. Usually, such devices are implemented on a specialised hardware, with limited capacities, and/or power consumption constraints. These systems are called the *embedded devices*. At the present date the embedded devices account for more than 98% of all produced electronic equipment [31], where the scope of their application spans over a wide range of areas, such as medicine, automotive industry and avionics.

In many applications, the only requirement is that the embedded system executes its functionalities correctly. However, depending on the purpose of the system, some additional requirements may be present, for instance, to correctly execute the functionalities within a certain time period. This is usually the case with the embedded devices that have a strong connection with the physical environment. The most notable examples are medical pacemakers, airbag systems in cars, autopilot functionalities in aircrafts, where a timely reaction to the outside stimuli is of crucial importance. That is, when a car accident occurs, an airbag system must commence the inflation process exactly at the critical moment. If it fails to do so, or does it too early, or too late, catastrophic consequences may occur. The embedded systems of such kind, where both the correct execution and temporal behaviour are important, are called the *real-time embedded systems*, and they are central to this dissertation.

The importance of the temporal behaviour of the real-time embedded system highly depends on its purpose. For instance, in some scenarios, not fulfilling the posed timing requirements has mild, almost negligible effects, usually expressed as a degradation in the quality of service. Examples of this category are live audio/video systems, and these devices are referred to as the *soft real-time systems*. Conversely, in some other scenarios, fulfilling all timing requirements is an absolute imperative, while any failure to do so may have catastrophic consequences. The aforementioned example with the airbag system falls in this category, and these devices are called the *hard real-time systems*.

In this dissertation, the focus is on hard real-time systems.

1.2 Real-Time Analysis

Simulations and measurement-based techniques are some of the methods that can be used to observe the temporal behaviour of the system. However, these techniques face several limitations. For instance, they are often time-consuming. Moreover, they give the possibility to observe the system behaviour during a certain time interval, which is usually not enough to capture all possible system states. These limitations become very obvious in the context of hard real-time systems, where one of the requirements is to investigate whether a given system will always meet all timing requirements, even under the worst-case conditions. In such cases the only viable approach is the *real-time analysis*.

Real-time analysis is a technique where an analytic description of the system characteristics is used to derive conclusions regarding the system runtime behaviour. The main advantage of this technique is that it can be used to analytically describe corner cases (worst-case scenarios), often with a certain degree of pessimism, whereas the analysis complexity and pessimism are usually in a trade-off relationship. If the focus of the analysis is on extracting the temporal properties of an identified, or artificially generated worst-case scenario, the analysis is called the *worst-case analysis*. This analysis is predominantly used in the hard real-time domain, where one of the fundamental requirements is to derive guarantees that a system behaviour will never violate any temporal constraint, not even in the worst-case scenarios.

The worst-case analysis is usually performed at design-time. Evidently, its efficiency highly depends on an accurate prediction of the system behaviour at runtime. In order to prevent the underestimation of the worst-case scenario, that is, to keep the worst-case analysis safe, any unpredictable and non-deterministic aspect of the system behaviour has to be accounted for in the analysis with a certain degree of pessimism. A more pessimistic analysis may cause a significant resource over-provisioning at design-time, and consequently lead to a severe underutilisation of platform resources at runtime. Thus, the foremost objective in the hard real-time domain is to propose analyses which are (i) safe, (ii) with acceptable computational complexity and (iii) with a tolerable amount of pessimism.

In this dissertation, the focus is on worst-case analysis.

1.3 Single-Core ⇒ Multi-Core ⇒ Many-Core

As the technological advancements provided the environment for thriving of real-time embedded systems, the demands for more advanced and sophisticated functionalities kept emerging each year. Until recently, a steady progress in the semi-conductor technology was able to successfully cope with the constantly rising requirements, thus resulting in a vast amount of transistors accommodated within a single processing unit. However, the miniaturisation process hit the limit [60], due to inability to manage dissipated heat and consequently increasing temperatures at high frequencies, thus hindering further continuation of an established trend of processing power enhancements related to single-core processors.

In order to continue the progress of computational devices, a paradigm shift in the processor design was needed. The chip manufacturers applied a different strategy; rather than enhancing the abilities of a single-core device, the idea is to integrate multiple cores within a single chip. From that moment onwards, the number of cores integrated within a same chip started to grow, and that trend is present even today. The first such platforms that emerged contained only a few cores, and they were referred to as *multi-cores*. When the number of cores integrated within a single chip outgrew the number of ten, the new term was coined for such devices - they were called *many-cores*. Even though that it may appear at this stage that the difference between multi-cores and many-cores is just in the mere number of cores, that is not true. In fact, multi-core and many-core platforms significantly differ in numerous aspects, such as: the interconnect medium, the memory system, the process of managing the correct and consistent system-wide state, etc.

The scientific area which transitioned the fastest and the smoothest from single-cores to multi- and many-cores is the high-performance computing. This comes as no surprise, as the development trends in the chip design are predominantly driven by the need for more powerful and efficient computational devices (better performance), which entirely coincides with the requirements of the aforementioned area. Very similar evolution trends, although with a small offset, are visible in the general-purpose computing area. The real-time embedded computing area lags even more behind the aforementioned areas. In fact, at the present date, single-cores and multi-cores present the majority of the systems that are considered for practical implementations in the real-time embedded domain, while many-core platforms are perceived only as an emerging technology that will be used in the forthcoming years.

The integration of many-core platforms in the real-time embedded domain goes slowly because the advancements in the field of chip design significantly differ, and, to an extent, contradict the requirements from the said domain. Specifically, the system predictability is one of the essential prerequisites for the efficient real-time analysis, while the topmost objective of chip designers is to improve the efficiency and the performance of the system, usually at the expense of the system predictability. In fact, there is a common opinion among researchers that the many-core domain presents a significantly more challenging environment for the real-time research, than the single-core domain.

In this dissertation, the focus is on many-core platforms.

1.4 Benefits of Many-Cores

Many-core platforms have the potential to bring numerous benefits to the real-time embedded domain, of which only few will be listed here.

- **Functionality enhancements**. The abundance of cores brings more computational power, which gives to possibility to extend the existing functionalities, or implement new ones.
- **Cost reductions**. Integrating into fewer many-core devices the functionalities that were previously executed on numerous single-core systems can significantly reduce design costs.
- **Flexibility**. The abundance of cores allows to accommodate workload changes at runtime. That is, admission tests can be performed and subsequently the existing functionalities can be extended, or new ones deployed. Similarly, obsolete functionalities can be removed from the system, so as to save more capacities for future extensions/deploys.
- **Energy/thermal management**. The abundance of cores allows to perform load balancing for energy and thermal management. That is, the workload can be efficiently migrated around the platform to prevent thermal hotspots and/or contribute to the durability of the system. Moreover, in cases where the capacity of the platform significantly exceeds the requirements, idle cores can be temporary shut down for two reasons: (i) to increase the durability of the system, and (ii) to decrease the power consumption.
- **Better fault tolerance**. The abundance of cores allows to recover from hardware failures. That is, once a malfunctioning of one core has been suspected or noticed, that core can be preventively shut down and the platform can continue to work with the remaining capacities.

1.5 Many-Cores in Real-Time Embedded Computing Domain

Despite the aforementioned benefits, numerous challenges arise when researchers try to put many-core platforms in the real-time embedded context. Perhaps the greatest challenge is the complex design. In order to circumvent this problem, the researchers usually do not analyse an entire system, but rather focus on a subset of aspects, typically one type of resources, e.g. processing elements, interconnect mediums, memory subsystems, and analyse them independently of the rest of the platform. Subsequently, based on the analysed aspect, the state-of-the-art approaches can be classified into categories.

In the rest of this section the overview of the many-cores and their perspective in the real-time embedded domain will be given. However, rather than focusing on the entire platform, the individual aspects will be emphasised and discussed independently.

1.5.1 Computation Process

Each functionality implemented within a many-core platform occasionally requires certain computation processes to be performed. These activities are executed on the processing elements –

cores. Besides in the mere number of cores, which can range from a dozen to hundreds, many-core platforms differ from each other in other aspects as well, such as types of cores. Nowadays, the most established trend in the chip design are platforms with cores which are identical in terms of physical characteristics, called the *homogeneous many-cores*. The most notable examples are the Tilera family of processors [93], the Single-Chip-Cloud Computer (SCC) manufactured by Intel [42] and the Epiphany processors designed by Adapteva [3]. One of the implications of such a design is that a computation requirements related to some functionality will always be the same, irrespective of the actual core that has been selected to perform the computation.

In recent years, chip manufacturers also explored the possibilities to integrate different types of cores within the same many-core platform. Usually, cores of a certain type are specialised in some activities, and the types are combined within the platform in such a way to complement each other. These computers are referred to as the *heterogeneous platforms*, and one example is the MPPA-256 Manycore Processor developed by Kalray [44]. The focus of the majority of works in the real-time domain is on homogeneous platforms, and the same is true with this dissertation.

How to assign the available cores to the functionalities existing within the platform has been, and still is, one of the most extensively studied problems in the real-time analysis of many-cores. The area that covers this aspect is called the *multiprocessor scheduling theory*. If the functionality does not have the ability to change cores, but always has to perform its computation on the same core, in the scheduling theory it is referred to as *non-migrative*. Conversely, if the functionality has the possibility to use multiple cores, it is considered as *migrative*. Depending on the level of migrative freedom given to functionalities, all state-of-the-art approaches can be classified into several groups.

1.5.1.1 Non-migrative (Fully Partitioned) Approaches

These approaches are in the scheduling theory also known as the *fully partitioned approaches*. Each functionality is statically, at design-time, assigned to a specific core where it has to perform its computation. Once assigned, the functionality cannot switch to another core (migrate), but instead, it has to always perform the computation on the same core. The scheduling of the workload on each core is performed by a local scheduler, in a single-core fashion, independent of the rest of the system. An illustrative example of this scheme is given in Figure 1.1.

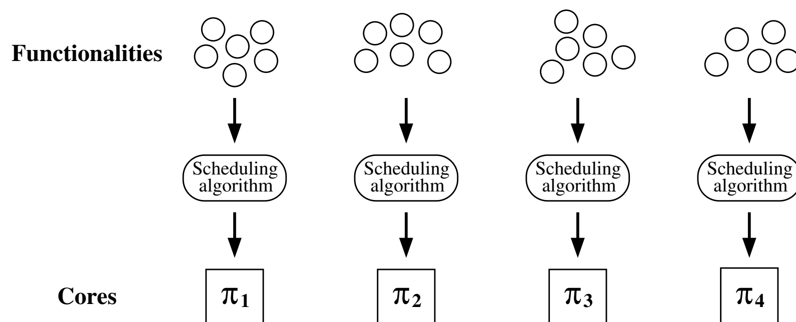


Figure 1.1: Fully partitioned scheduling

The advantage of this concept is reflected in the fact that the prevention of functionality migrations makes the system more predictable and hence analysable, which is beneficial when deriving real-time guarantees. Moreover, the problem of workload scheduling is, relatively speaking, significantly simplified, because each core can be perceived (analysed) as an independent single-core system, and the uniprocessor scheduling theory [18, 58], which is very advanced, can be applied in this context. However, the advantage of this concept is at the same time its weakness. The inability to perform migrations makes these approaches rigid and inflexible, which renders them ineffective in scenarios with dynamic load changes, or scenarios where runtime load balancing is required, for energy/thermal management and fault tolerance reasons. These limitations significantly narrow the application domain of the fully partitioned approaches.

1.5.1.2 Semi-Partitioned Approaches

These techniques (e.g. [15, 47]), to some extent, embrace the concept of workload migrations. However, each migrative functionality still has its computation pattern defined at design-time. That is, a migrative functionality may have its computation split among two (or more) cores, but always has to perform a prescribed amount of work on each of the cores where it migrates, in a given order. A semi-partitioned scheduling is illustrated in Figure 1.2.

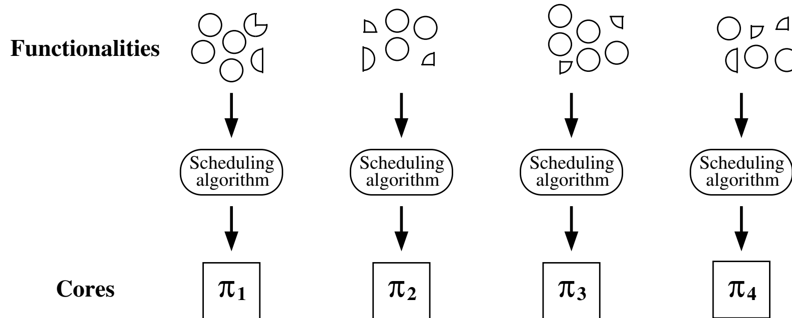


Figure 1.2: Semi-partitioned scheduling

When compared to the fully partitioned concept, the semi-partitioned one allows for more efficient workload assignment and consequently more effective resource utilisation [15]. However, these approaches suffer from the same limitations as the previous category, that is, they are inapplicable in scenarios where dynamic load changes occur, or where the runtime load balancing is required. In terms of scheduling, both the aforementioned approaches tend to perceive the many-core platform as a collection of independent single-core systems. Evidently, this strategy removes the platform flexibility, which has been already identified as one of its greatest benefits (see Section 1.4).

1.5.1.3 Fully Migrative (Global) Approaches

These approaches (e.g. [7, 9]), in the scheduling theory also known as the *global approaches*, allow unconstrained workload migrations, which means that each functionality has the possibility

to migrate at any time to any core within the platform, and perform its computation there. All migration decisions are made at runtime, by a single global scheduler. An illustrative example of global scheduling is given in Figure 1.3.

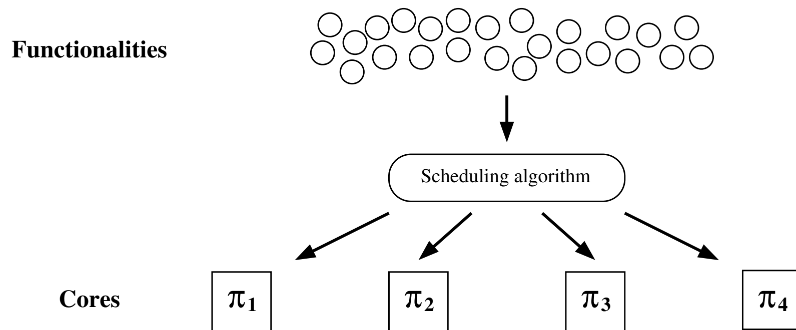


Figure 1.3: Global scheduling

Global approaches are much more flexible than the two aforementioned categories, and as such do not suffer from the same limitations. That is, due to the fact that migration-related decisions are made at runtime, the global approaches can successfully cope with dynamic load changes. Moreover, the load balancing is inherently supported by the design itself, which makes it possible to implement some energy/thermal management policies by shutting down some of the cores at runtime. However, the global approaches require a global notion of time, and also global, centralised structures, such as a scheduler and a ready queue. As the number of cores and the workload within the platform increase, the contentions for the global structures also increase, thus scalability issues become more apparent. It is important to mention that these scalability issues are indeed some of the key reasons why multi-core and many-core platforms are considered as different systems. That is, in multi-cores some fully migrative scheduling approaches can be efficiently implemented, because the scalability issues are mild due to the small number of cores. Conversely, in the many-core domain, global scheduling is not considered as an efficient and viable approach. In fact, numerous challenges arise when attempting implementations [10].

1.5.1.4 Clustered Approaches

Clustered approaches (e.g. [19]) present a concept which combines the properties of the non-migrative and fully migrative approaches. Specifically, all cores are divided into disjoint groups, where each group forms one cluster. A cluster is perceived and treated as an independent system, with the global scheduling policy applied on the cluster-level. Moreover, each functionality is assigned to exactly one cluster, and it has the possibility to freely migrate within its cluster. Figure 1.4 depicts the system with the clustered scheduling.

These approaches offer both runtime load balancing (the positive aspect of fully migrative approaches) and scalability (the positive aspect of non-migrative approaches). However, they are inefficient in scenarios where workload migrations are driven by fault tolerance or energy/thermal

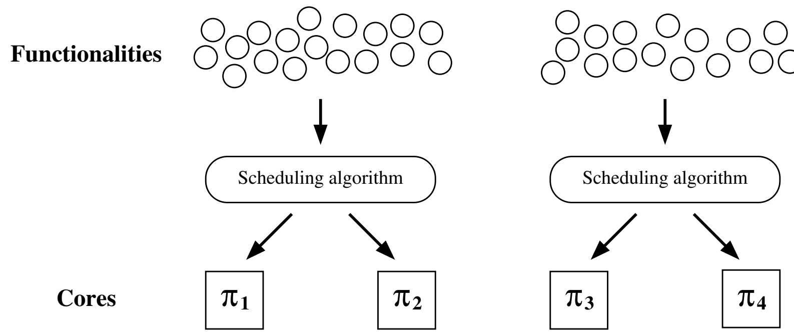


Figure 1.4: Clustered scheduling

management, where inter-cluster migrations (preferably spatially distant ones) are a necessary option. This limitation narrows the scope of application of these approaches.

1.5.1.5 Discussion

So far, different workload execution techniques have been analysed solely from the perspective of processing elements, i.e. cores. Yet, it is worth mentioning that the selection of the workload execution method impacts not only cores, but also other system resources, such as the interconnect medium. For example, in the non-migrative approaches, the core of each functionality is known at design-time. This implies that the communication between any two functionalities will always involve the same two cores, and with a deterministic routing mechanism it can be assured that the exchanged data always uses the same resources of the interconnect medium, e.g. always traverses the same path over the interconnect. This possibility to inject the predictability into the communication between functionalities can significantly ease the real-time analysis of the interconnect mediums.

Conversely, in fully migrative and clustered approaches it is impossible to predict on which core will which functionality be at runtime. This infers that the communication patterns among functionalities are unpredictable and impossible to analyse at design-time. For example, any two communicating functionalities may at some point during runtime share the same core, but later migrate to two spatially very distant cores. Performing the real-time analysis of the interconnect medium under these schemes is very hard, if not impossible.

The aforementioned discussion shows that the selection of the workload execution technique has a broad effect and impacts the entire system, not just the cores. The benefits and limitations of different techniques have been presented. One of the goals of this dissertation is to propose a novel workload execution concept, that will make many-cores more amenable to the real-time analysis. Specifically, such a concept should exploit the full flexibility of the many-core platform by allowing workload migrations, like the fully migrative and clustered approaches, but also be scalable and predictable, like the non-migrative approaches. For that purpose, in Chapter 3 a novel concept, called the *Limited Migrative Model* is proposed.

1.5.1.6 Assumptions Regarding Processing Elements

The platform under consideration Ψ contains a set of processing elements Π, which is a collection of z identical cores $\Pi = \{\pi_1, \pi_2, ..., \pi_{x-1}, \pi_z\}$. The value of z is not fixed, but it can be in the range $50 - 150$, which corresponds to the number of processors in currently available many-core platforms, e.g. [3, 42, 93]. The decision to keep the number of cores parametrised rather than constant is based on the fact that this strategy allows to test the scalability potential of the concepts proposed in this dissertation, simply by varying the parameter z.

In this dissertation, the focus is on homogeneous many-core platforms.

1.5.2 Interconnect medium

As soon as the idea to integrate multiple cores within the same platform emerged, the most important question was how to efficiently interconnect them. In the multi-core domain that problem is usually solved by providing a direct point-to-point communication between all core-pairs. An illustrative example of a point-to-point communication is given in Figure 1.5, where a data packet is sent from the core π_1 to the core π_2. Notice, that this concept is not scalable, because the number of necessary communication links grows sub-quadratically with the number of cores, i.e. for z cores in total $\frac{z \cdot (z-1)}{2}$ bidirectional links are needed. This is another major difference between the multi-core and the many-core platforms, where, for the former, the point-to-point communication is an efficient approach, while for the latter an alternative method is needed [26].

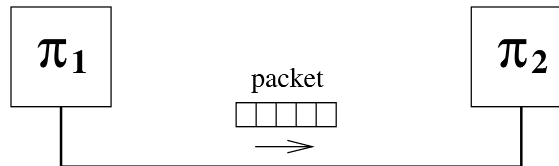


Figure 1.5: Point-to-point communication

1.5.2.1 Network-on-Chip Paradigm

The quest for scalable interconnect mediums was highly popular around 15 years ago. Eventually, the research efforts paid off, and a new interconnect paradigm emerged. It is called the *Network-on-Chip*, or simply NoC [12, 26, 37]. The NoC paradigm abstracts away the communication medium from the communicating entities. In other words, the functionalities that exchange data packets are totally agnostic about the transfer process. This is possible with the additional logic that is implemented within the NoC elements called the *routers*. Routers serve as a network interface to the cores, and cores access them in order to send/receive data. Although there are routers that are accessed by multiple cores, e.g. [42], usually in many-core platforms there is one router per core, e.g. [3, 93]. Thus, the data transfer between two functionalities that are located on different cores is performed in the following way: (i) the packet is sent by the sender's core

to the local router, (ii) the packet is transferred through the network of routers until it reaches the destination router, and (iii) the packet is sent by the receiver's router to the local core.

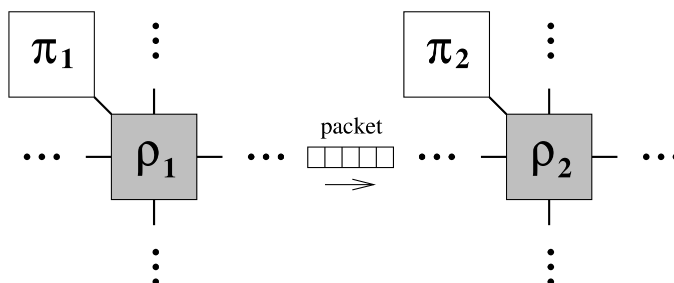


Figure 1.6: Network-on-Chip communication

It is worth mentioning that not all pairs of routers have a direct link between each other. In fact, in the majority of NoCs, each router is connected only with a small set of neighbouring routers. In such cases, a packet may need to traverse multiple routers before reaching the destination. An illustrative example of the communication over the NoC architecture is given in Figure 1.6, where the data packet is exchanged between the two cores π_1 and π_2. The shaded rectangles ρ_1 and ρ_2 depict the routers belonging to π_1 and π_2, respectively. Notice that ρ_1 and ρ_2 do not have a direct link connection, so the packet needs to traverse one or more intermediate routers before reaching ρ_2. For better clarity, the intermediate routers were omitted from the figure.

In this dissertation, the focus is on the NoC-based many-core platforms.

1.5.2.2 Topologies

How to efficiently arrange the routers inside the NoC architecture and how to organise the inter-router connections is one of the most important questions in the NoC design theory. This is understandable, as the selection of the topology has a significant impact on numerous aspects, such as the system performance, the design complexity, the design cost, etc. Some NoC topologies are depicted in Figure 1.7. For clarity purposes, a single line was used to represent the connection between all directly connected routers, whereas the connection is typically implemented with two unidirectional links of opposite directions. Notice, that the fully connected topology (Figure 1.7(d)) is very similar to the point-to-point approach.

To better understand the differences between topologies, a simple comparison is performed between the 2-D mesh topology (Figure 1.7(c)) and the fully connected one. In terms of required resources, the 2-D mesh design is much more scalable, as illustrated with the following numerical example. Assuming the platform with 100 cores, the fully connected design requires in total $\frac{100 \cdot 99}{2} = 4950$ bidirectional links, while the 10×10 NoC architecture requires only $2 \cdot 10 \cdot 9 = 180$ bidirectional links, which represents less than 4% of the number of links required by the former design. However, it is also worth mentioning that in the fully connected approach all packets traverse only two routers, irrespective of the locations of the sender and the receiver, while on a 2-D mesh platform that is not always the case. Specifically, on a 10×10 2-D mesh, a packet that is

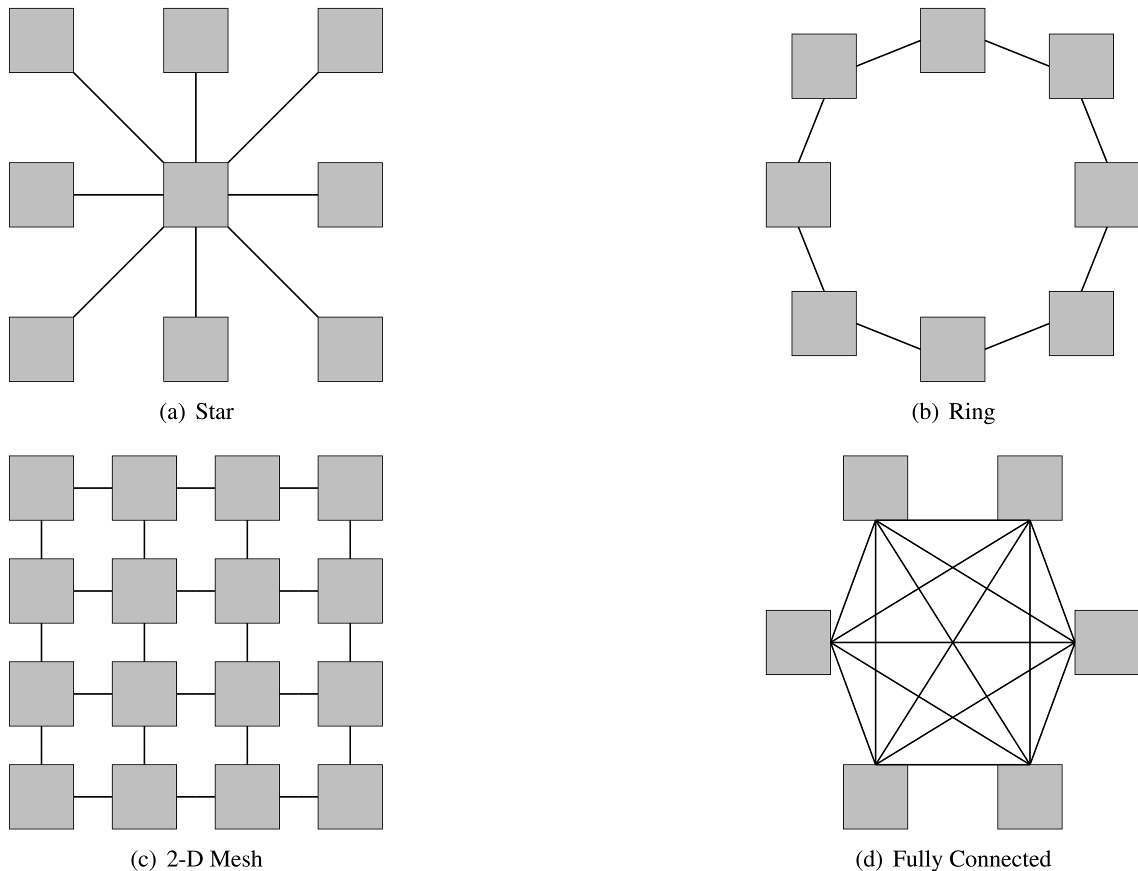


Figure 1.7: Different NoC topologies

exchanged by two diagonally placed corner routers must traverse $10+9=19$ routers! This infers that, as pointed out in the beginning of this section, different topologies have significantly different properties, and the preference of one topology over another highly depends on the purpose of the system. An interested reader may consult the work of Abba and Lee [1], where a comprehensive comparison of numerous different NoC topologies is performed.

Despite the variety of possible topologies, so far, only few of them have been used as the interconnect mediums in many-cores. For example, a 2-D mesh is one of the most popular choices, and it is present in some currently available many-cores, e.g. [3, 42, 93]. Additionally the ring topology is also present in the Xeon Phi family of processors [43], manufactured by Intel, while in the MPPA-256 Manycore Processor developed by Kalray [44] a 2-D torus topology is used.

In this dissertation, the focus is on the 2-D Mesh NoC interconnect medium.

1.5.2.3 Routing

As already described in the previous section, in many topologies, including the 2-D mesh, not all the router-pairs are directly connected. Therefore, in such cases, depending on the position of the sender and the receiver, a packet may need to travel across multiple intermediate links and

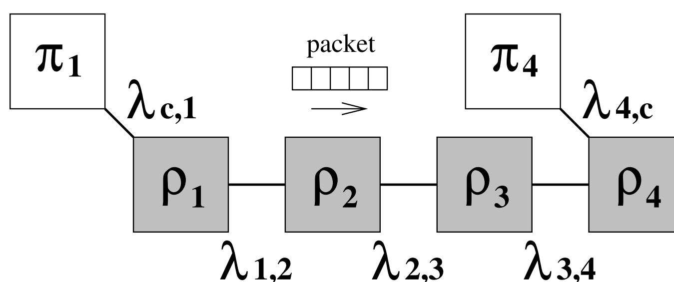


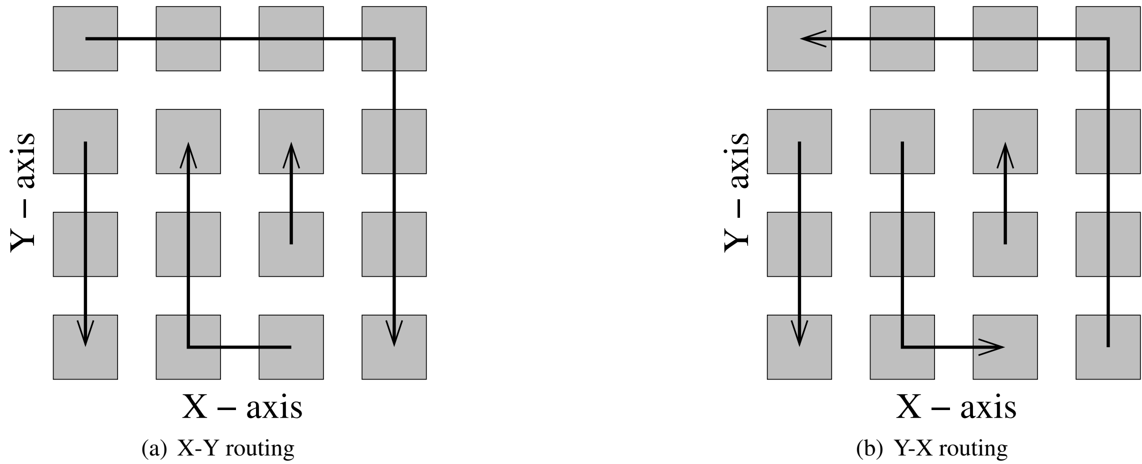
Figure 1.8: Packet transfer over NoC

routers. A set of traversed network elements (routers and links) is in the literature called the *packet route*, while the number of traversed links is usually referred to as the cardinality of the path, or the *number of hops*. An illustrative example of one packet transfer is given in Figure 1.8. A packet is sent from the core π_1 to the core π_4. During its transfer, the packet traverses the links $\lambda_{c,1}, \lambda_{1,2}, \lambda_{2,3}, \lambda_{3,4}, \lambda_{4,c}$, and the routers $\rho_1, \rho_2, \rho_3, \rho_4$, which jointly constitute the route of 5 hops. Note, that for clarity purposes, the cores belonging to ρ_2 and ρ_3 have been omitted from the figure.

The process of transferring the packet from its source to its destination is called the *routing*, and that action is the responsibility of the routers. Once a packet reaches the router, the router decides in which direction the packet will be forwarded. The logic inside the router that is responsible for making this decision is called the routing algorithm. Evidently, there are numerous criteria based on which the routing decisions can be derived. For instance, one option is to minimise the path, and hence derive routing decisions such that the packet always traverses the minimal possible number of hops. This class of routing algorithms is called the *minimal routing*. Moreover, if the packets between the same source and the same destination are always routed across the same path, then it is referred to as the *deterministic routing*. Alternatively, the routing decisions can be made at runtime, for example, based on the status and the load of individual links. Such techniques are called the *adaptive routing*. The adaptive routing can improve the performance of the system (the average case behaviour), however, at the expense of the predictability. Conversely, the deterministic routing is predictable and much easier to implement, but may cause an inefficient utilisation of the NoC resources, whereas some links may be heavily congested, while some other links may be completely idle.

The selection of the routing mechanism depends on the purpose of the system. As already mentioned, in the real-time embedded domain the predictability of the system is essential, because it allows to analyse the temporal behaviour of the system with significantly less pessimism. Thus, in the real-time domain, the deterministic routing techniques are a preferable option. Unintuitively, this coincides with the development trends of many-core platforms, because the deterministic routing techniques are indeed the most common choice in the currently available many-cores, e.g. [3, 42, 93]. However, this coincidence did not occur from the intentions of chip manufacturers to enforce predictability, but it occurred solely because the deterministic routing is much simpler to implement. Note that this is a very rare case where the trends in the development of many-core platforms suit the real-time requirements.

One class of popular minimal deterministic routing algorithms in 2-D mesh NoCs is the *dimension-ordered routing*. Assuming these schemes, the packets are firstly routed along one dimension of the NoC, and after reaching the coordinate of the destination, if needed, continue the transfer along the other dimension. One of the most popular routing algorithms of this class is the *X-Y routing*, where the horizontal axis of the platform is usually denoted with the letter X, while the vertical axis is denoted with the letter Y. The X-Y routing policy is deadlock and livelock free [40].



(a) X-Y routing (b) Y-X routing

Figure 1.9: Minimal dimension-ordered routing algorithms

In Figure 1.9 are illustrated several packets routed with the X-Y routing algorithm (Figure 1.9(a)) and with the Y-X routing algorithm (Figure 1.9(b)). For clarity purposes, the links were omitted from the figures. The X-Y routing is one of the most popular routing algorithms in currently available many-cores, e.g. [3, 42, 93].

In this dissertation, the focus is on the X-Y routing algorithm.

1.5.2.4 Switching

When the NoC resources are free, a packet intermittently traverses routers and links on its path towards the destination. For example, the packet illustrated in Figure 1.8 traverses NoC elements in the following order: $\lambda_{c,1} \rightarrow \rho_1 \rightarrow \lambda_{1,2} \rightarrow \rho_2 \rightarrow \lambda_{2,3} \rightarrow \rho_3 \rightarrow \lambda_{3,4} \rightarrow \rho_4 \rightarrow \lambda_{4,c}$. However, in the presence of other traffic, it may happen that one of the links on the path of a packet is busy transferring some other packet. In such cases, the former packet is stalled and it is stored inside the router element called the *port*. Usually, the NoC router contains multiple ports, each dedicated to the traffic to/from a specific link. A simplified illustration of the 2-D mesh NoC router architecture is given in Figure 1.10. Towards each neighbouring router with which it has a direct link connection (e.g. north, south, east and west direction), a router may have: (i) only an input port, (ii) only an output port, or (iii) both an input and an output port. Input and output ports are used to store incoming and outgoing packets in a given direction, respectively. Inside the

router, the switch crossbar is used to transfer the packets between different input and output ports. Note that the router may have two more ports, namely the *core input port* and the *core output port*. These ports are used to exchange the data with the local core. The core ports and some other router elements (e.g. the control unit, the arbitration unit) have been omitted in Figure 1.10 for clarity purposes.

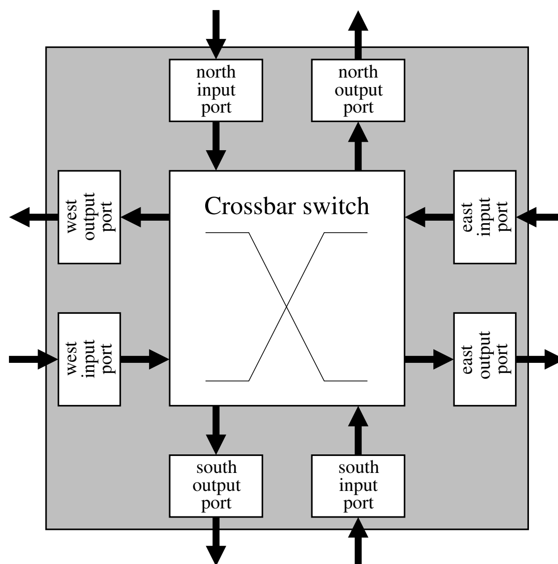


Figure 1.10: Simplified 2-D mesh NoC router architecture

In the earliest router designs, one of the most popular choices was to construct the ports with the capacity to store entire packets. This concept allows to implement the data transfer technique where each packet traverses NoC elements in a sequential manner. In other words, a packet is first entirely transferred between two adjacent routers, and only then it continues its progress further. This approach is called the *store-and-forward switching* [92]. An example of this data transfer technique is depicted in Figure 1.11, where a data packet of 5 bytes is being sent from the core π_1 to the core π_4.

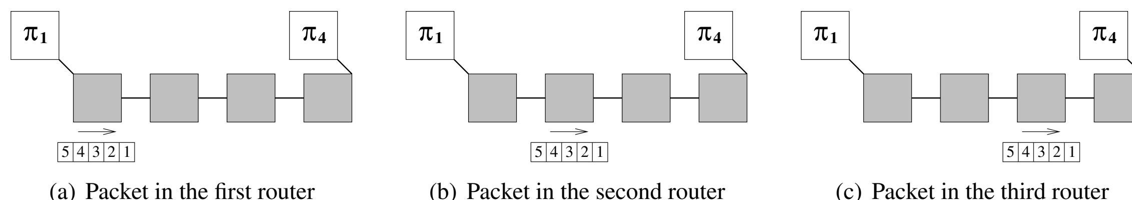
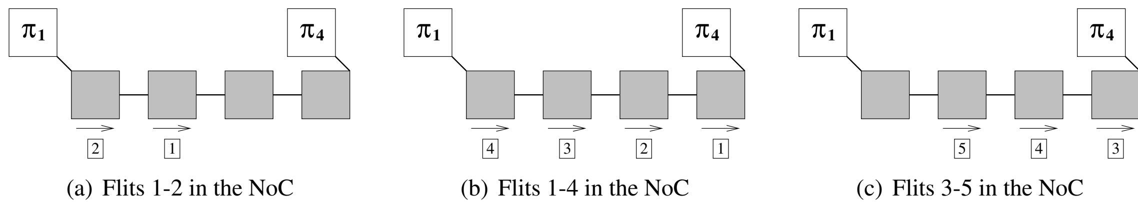


Figure 1.11: Store-and-forward switching

One of the greatest limitations of the store-and-forward switching is the port capacity requirement. Evidently, this technique can be applied only in cases where the port capacity is sufficient to store entire packets. However, over the years, the demands for more sophisticated functionalities implemented on many-core platforms also required more frequent data transfers and larger

and larger packets. As the buffering within ports became a challenge, the focus shifted towards alternative switching techniques.

The *wormhole switching* [67] is a data transfer technique that is conceptually the opposite of the store-and-forward switching. Specifically, prior to sending, a data packet is divided into small elements of fixed size, called *flits*. Usually the flit size is set to be equal to the link width, which is in the range 1 – 64 bytes in currently available many-cores. A flit is a basic, transferable unit across the NoC and it is indivisible. The first flit of the packet is called the *header flit*, while the last flit of the packet is referred to as the *tail flit*. Once the packet is divided into flits, the transfer commences. The flits are released into the NoC in an orderly fashion, the header and the tail flit are injected first and last, respectively. The header flit establishes the path, and the rest of the flits follow in a pipeline manner. In Figure 1.12 is illustrated the wormhole switching, where a 5-byte packet is transferred between the cores π_1 and π_4, while the size of the flit in this example is 1 byte.



(a) Flits 1-2 in the NoC (b) Flits 1-4 in the NoC (c) Flits 3-5 in the NoC

Figure 1.12: Wormhole switching

One benefit of this scheme is that the flits are allowed to travel in parallel, thus the throughput significantly increases [48]. Moreover, the buffering requirements are considerably reduced, and it is not any more necessary to have ports with capacities to store entire packets. In fact, wormhole-switched NoCs can successfully operate with ports which capacity is equal to the size of only 1 flit! However, the storage buffers inside the ports of currently available many-cores are usually larger than 1 flit. This is especially helpful in scenarios where the header flit is being blocked due to the busy link, and the rest of the flits do not need to stall in their current ports, but can continue their progress through the intermediate routers, and group themselves in the ports as close to the busy link as possible. At the present date, the wormhole switching technique is one of the predominant data transfer choices in many-core platforms. e.g. [42, 44, 93].

In order to avoid the overflow of port buffers, a certain mechanism has to be implemented to prevent new flits to enter routers which ports have already been full with other flits. One of the most popular techniques is called the *credit-based flow control*, and it works as follows. For each empty space to store one flit, a port has one credit. When the flit enters the port, the credit is consumed and sent to the previous port. Indeed, the credits always travel in the opposite direction of the flits, as illustrated with Figure 1.13. Note that flits may progress only if there are available credits in the receiving port.

Notice that the number of credits directly influences the number of flits that can be concurrently stored inside the port. As already stated, the wormhole switching mechanism can successfully

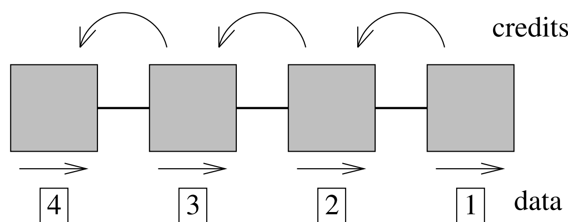


Figure 1.13: Credit-based flow control mechanism

work even if each port has only a single credit (flit-sized buffer), however, the chip designers usually dimension the port buffers and assign the credits in such a way that multiple flits can be concurrently stored.

The credit-based is not the only flow control mechanism for wormhole-switched NoCs. For example, one alternative scheme is the *back suction* [28]. However, this technique requires the additional hardware support in the form of virtual channels, which will be explained later.

In this dissertation, the focus is on the wormhole switching technique with the credit-based flow control mechanism.

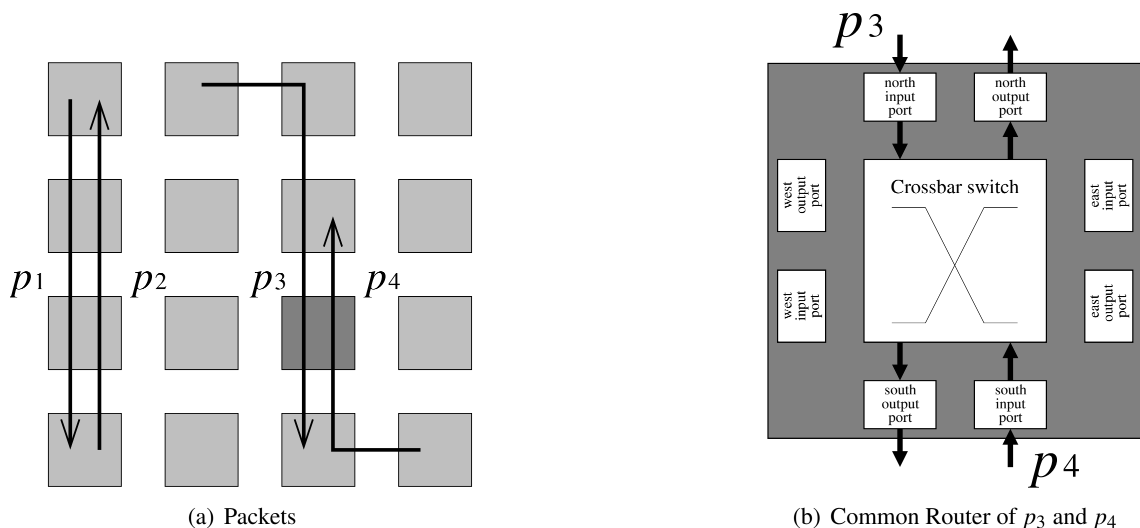
1.5.2.5 Arbitration

In the previous sections, the transfer of a single packet over the NoC has been analysed. However, it rarely happens at runtime that all the routers and the links on the path of a traversing packet are free. In fact, due to the numerous functionalities existing within many-core platforms, the amount of generated traffic can cause significant contentions inside the NoC medium, where two of more packets concurrently try to access the same resource, e.g. a link.

Several approaches have been proposed with the same fundamental idea to avoid traffic contentions. These methods are referred to as the *contentionless approaches*. One way to achieve this is by pre-allocating times slots at which packets may traverse, e.g. [36, 63, 77]. Note that this strategy is very similar to the time-division-multiple-access method – a popular access technique for shared mediums in networks. Another possibility is to pre-allocate an entire path of the packet prior to sending, and subsequently transfer it without contentions. The pre-allocated path is called the *virtual circuit*, and one implementation that exploits this concept is the MANGO NoC [14]. Conversely, the *contention-aware approaches* are the approaches in which the contentions among traffic packets are allowed. In the rest of this section the focus is on such NoCs.

Note that having a common router is only a necessary, but not a sufficient condition for two packets to contend, while having a common link is both a necessary and a sufficient condition for the contention. This is explained with the illustrative examples given in Figures 1.14 and 1.15. For clarity purposes only the routers and the packets are depicted, however, recall that between each pair of directly connected routers two unidirectional links exist. First, consider the example given in Figure 1.14(a), where 4 packets exist, namely p_1, p_2, p_3 and p_4. Notice, that p_1 and p_2 have 4 common routers and no common links. Similarly, p_3 and p_4 have 3 common routers but no common links. Thus, according to the aforementioned explanation, there are no contentions among

these packets. To understand that this is indeed the case, consider Figure 1.14(b), which depicts the router that is traversed by p_3 and p_4 (the same router is in Figure 1.14(a) emphasised with a darker color). Indeed, it is visible that these two packets require different resources and hence can concurrently traverse the router without interfering with each other. A similar conclusion can be reached for the rest of the routers in Figure 1.14(a), which are traversed by two packets. This confirms the first part of the statement, that having a common router is not a sufficient condition for the contention between packets.



(a) Packets

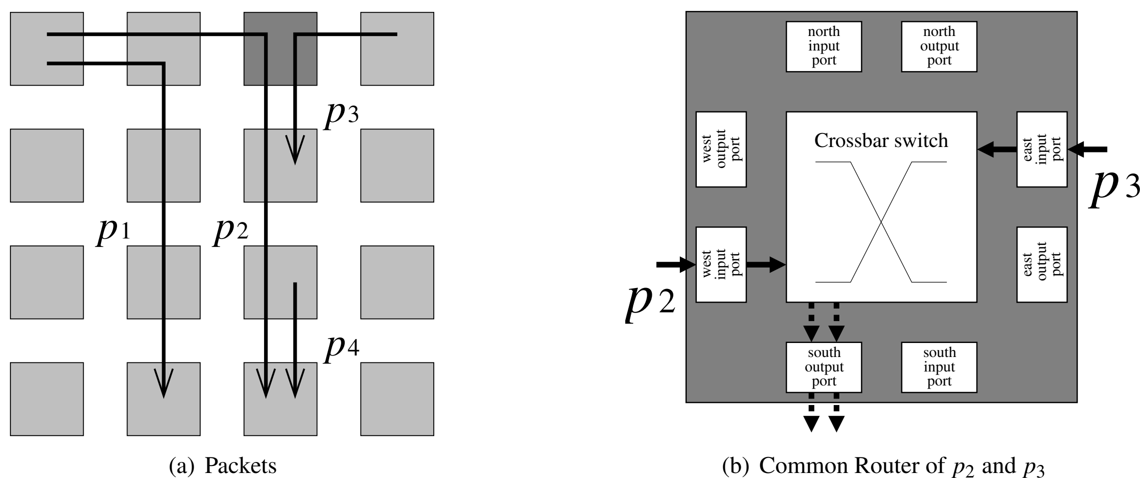
(b) Common Router of p_3 and p_4

Figure 1.14: Packet traversals without contentions

Now, consider the illustrative example given in Figure 1.15(a). Again, 4 packets exist, p_1, p_2, p_3 and p_4. However, in this example p_2, besides sharing the routers, also shares at least one common link with each of the remaining packets. On the other hand, the remaining packets p_1, p_3 and p_4 do not have the common routers, nor links with any other packet except p_2. Figure 1.15(b) depicts the router which is common for p_2 and p_3 (the same router is in Figure 1.15(a) emphasised with a darker color). It is visible that these two packets contend for the common resource (the south output port) and hence interfere with each other.

The aforementioned example demonstrates that the common link is indeed a necessary and a sufficient condition for the contention between two or more packets. Since the common link requires at least one common router, it can be concluded that the common router is only a necessary, but not a sufficient condition for the contention.

When contentions do occur, there exists a policy that decides which packet has the precedence among all packets contending for the same output port (and hence the link connected to it). Only the packet with the precedence will be able to progress, while the rest of the contending packets will be stalled. The policy that decides which packet has the precedence is called the *arbitration policy*. One of the most commonly used arbitration policies is called the *round-robin*, and it is explained with the following example.



(a) Packets

(b) Common Router of p_2 and p_3

Figure 1.15: Packet traversals with contentions

Assume that packets p_1, p_2, p_3 and p_4 enter the common router from the input ports *west*, *east*, *north* and *core* respectively, as illustrated in Figure 1.16. Moreover, consider that all the aforementioned packets need to progress in the south direction, hence try to access the south output port of the depicted router. Evidently, these packets contend, so only one of them can progress at any time instance, while the other packets will be stalled.

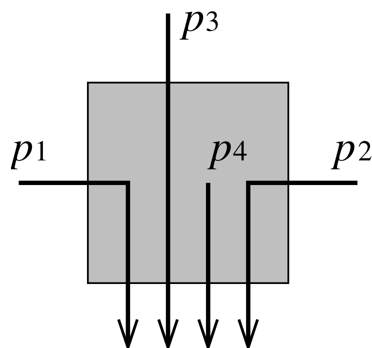


Figure 1.16: Contending packets

As already described, the arbitration policy decides which packet has the precedence among all contending packets. The round-robin arbitration policy changes the precedence of packets, based on an input port from which they entered the router, in a orderly fashion. The change occurs after each transfer, that is, if p_1 had the precedence to progress, another packet from the west input port, say p'_1, will not be able to progress until p_2, p_3 and p_4 have progressed. However, p'_1 will have the precedence over the other packets from the input ports of p_2, p_3 and p_4, say p'_2, p'_3 and p'_4, respectively. Thus, one round-robin arbiter, applied to the aforementioned example, might give the precedence to the packets in the following order: $p_1, p_2, p_3, p_4, p'_1, p'_2, p'_3, p'_4$. Note, that other round-robin implementations are also possible, and that the only fundamental property of this scheme can be summarised as follows: any packet, coming from the input port of the last transferred packet, will not be able to progress, until the packets from all other input ports have been considered. Also note, that if there are no packets coming from a certain port, the same is immediately skipped, and the packets from the next port are considered. The round-robin arbitration policy assures fairness among packets and prevents starvation, which makes it very popular choice in the currently available many-cores, e.g. [3, 93].

In this dissertation, the round-robin arbitration policy is analysed.

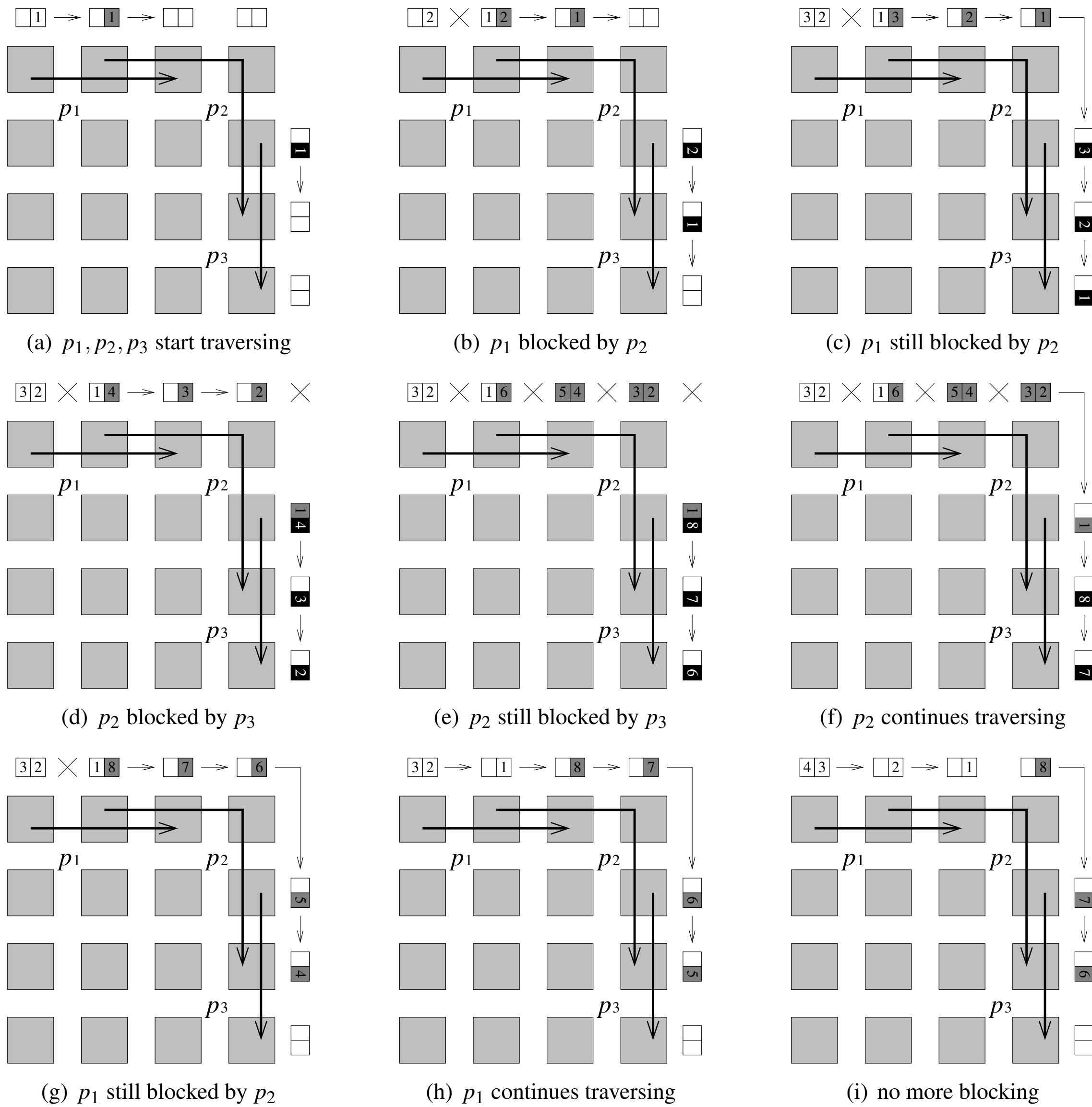
1.5.2.6 Worst-Case Real-Time Analyses Applicable to Round-Robin-Arbitrated NoCs

In the early works related to the timing analysis of wormhole-switched networks, e.g. [23, 29], the researchers attempted to apply the concepts from the queuing theory. The focus of these studies is on the performance (the average-case behaviour), hence are not suitable for the hard real-time domain. Moreover, there was an attempt to employ the concepts of the network calculus theory [54] in the worst-case analysis of wormhole-switched networks, assuming a weighted round-robin arbitration policy [79], however, one significant limitation of this approach is that it is overly pessimistic [33]. An additional element called the *wormhole section* was introduced [34], with the primary objective to decrease the pessimism of the worst-case analysis, which is based on the network calculus theory. Yet, this approach targets SpaceWire networks, which are small-scale interconnects with few links and flows, while it faces challenges, such as scalability and pessimism, when applied to larger networks with more flows, like NoCs for many-cores [35]. Moreover, the approach for the worst-case analysis called the *recursive calculus method* [32] was proposed, also for SpaceWire networks. This method is scalable, and hence can be applied to NoCs, however, its greatest limitation is that it does not take into account the inter-arrival times of packets and, as comprehensively covered in Section 2, it can be very pessimistic. So far, there is a lack of adequate techniques for the worst-case analysis of round-robin-arbitrated wormhole-switched NoCs, and one of the objectives of this dissertation is to propose a novel method to compute safe and tight upper-bound estimates on the worst-case delays of individual traffic packets.

1.5.2.7 Indirect Contentions

As already explained, the main advantages of the wormhole switching technique are good throughput and small buffering requirements [48]. These benefits come from the fact that flits of a packet may traverse in parallel. However, concurrent use of multiple resources (i.e. routers and links on the path of the packet), may, at the same time, cause very complex contention scenarios. Specifically, a traversing packet can suffer interference not only from the packets with which it shares some resources, but also from other packets with which it does not. This is explained with an illustrative example given in Figure 1.17.

As it is visible from the figure, there exist 3 packets, p_1, p_2 and p_3. Each packet has 8 flits, and each port has the capacity to accept 2 flits. The ports of interest are depicted outside of their respective routers. Packets p_1 and p_2 share 2 routes and 1 link, which is both a necessary and a sufficient condition for the contention between them. The same is true for p_2 and p_3. However, p_1 and p_3 do not have common resources, and hence they do not contend. Consider that all packets start traversing at the same time (Figure 1.17(a)). When the flits of p_1 reach the router that is common for p_1 and p_2, they cannot progress further, because the output port of that router, and its respective link, are busy transferring the flits of p_2. Therefore, the further progress of p_1 is impossible, and it is said that p_1 suffers blocking from p_2 (Figure 1.17(b)). Because the links on their paths are still free, p_2 and p_3 uninterruptedly progress (Figure 1.17(c)). However, when the flits of p_2 reach the router which is common for p_2 and p_3, the blocking occurs again



(a) p_1, p_2, p_3 start traversing
(b) p_1 blocked by p_2
(c) p_1 still blocked by p_2
(d) p_2 blocked by p_3
(e) p_2 still blocked by p_3
(f) p_2 continues traversing
(g) p_1 still blocked by p_2
(h) p_1 continues traversing
(i) no more blocking

Figure 1.17: Indirect contentions

(Figure 1.17(d)). Notice that p_1 is blocked by p_2, which is in turn blocked by p_3. This infers that, due to the traversal of p_3, the progress of p_1 will be delayed even more, despite the fact that p_1 and p_3 do not have common resources. In other words, p_3 indirectly blocks p_1. The indirect blocking will be revisited at the end of the example.

p_1 and p_2 remain blocked until the last flit of p_3 leaves the common router of p_2 and p_3 (Figure 1.17(e)). When the last flit leaves the router, p_2 continues traversing (Figure 1.17(f)). Similarly, p_1 remains blocked as long at there is at least one flit of p_2 in their common router (Figure 1.17(g)). After the last flit of p_2 leaves the common router, p_1 continues traversing (Figure 1.17(h)). Finally, all packets finish their traversals uninterrupted (Figure 1.17(i)).

The previous example demonstrated that a packet may suffer the delay from packets with which it contends directly, but may also suffer the delay from packets with which it contends

indirectly (no common resources). The indirect contentions can significantly affect the timing properties of packets. This is demonstrated with a simple numerical computation for the aforementioned example. For the sake of simplicity, consider that the traversal of each flit between each pair of adjacent routers takes 1 time unit. The time instances at which p_1, p_2 and p_3 complete their traversals can be easily deduced from the illustrated example, and are $21, 16$ and 9, respectively. However, if p_3 did not exist, the traversal of p_1 would be completed at time instance 16. Thus, the existence of p_3 caused the additional delay of 5 time units to the traversal of p_1. From this example it is obvious that, in order to perform the worst-case analysis of NoC traffic, it is necessary to take into account both direct and indirect contentions.

1.5.2.8 Virtual Channels

The effects of indirect contentions can be, to some extent, mitigated by enforcing a rule on the usage of port-buffers. Consider the example introduced in the previous section, but with one additional rule, that each packet may store at most 1 flit in each port-buffer along its path. In other words, inside each port-buffer, each packet has a private, dedicated slot of the size of 1 flit. This is illustrated in Figure 1.18, where the slots to different packets are depicted with different colors. Moreover, for clarity purposes, only the slots of interest are shown, which implies that for the routers that are traversed by only one packet, only a single slot is shown (the corner routers in this example).

Consider again the scenario where p_1, p_2 and p_3 are released at the same time (Figure 1.18(a)). Like in the previous example, p_1 is blocked when its flits reach the router that is common for p_1 and p_2 (Figure 1.18(b)). The links on the paths of p_2 and p_3 are free, so these packets uninterruptedly progress (Figure 1.18(c)). When the flits of p_2 reach the router that is common for p_2 and p_3, the blocking occurs again (Figure 1.18(d)). Recall, that in the previous example this was the moment when the indirect contention between p_1 and p_3 occurred, when only p_3 could progress, while p_1 and p_2 were blocked. However, due to the existence of per-packet slots inside port buffers, this problem can be mitigated. Specifically, once p_2 becomes blocked, p_1 can continue its traversal because all resources on its path are free. Therefore, the moment when p_2 becomes blocked by p_3 is the moment when p_1 continues traversing (Figure 1.18(d)).

The situation remains the same until the moment when the last flit of p_3 leaves the router that is common for p_2 and p_3 (Figure 1.18(e)). When that last flits leaves the router, p_2 can continue its traversal. At the same time the traversal of p_2 causes another blocking to p_1 (Figure 1.18(f)). p_1 remains blocked until the last flit of p_2 leaves the router that is common for p_1 and p_2 (Figure 1.18(g)). When that happens, p_1 can continue its progress (Figure 1.18(h)). Finally, all packets finish their traversals without further contentions (Figure 1.18(i)).

The rule of allowing each packet to consume only 1 slot in each router it traverses can significantly improve the throughput, as demonstrated here. Consider again that the traversal of each flit between two adjacent routers takes 1 time unit. For the example illustrated in Figure 1.18 it can be deduced that p_1, p_2 and p_3 complete their traversals at time instances $17, 16$ and 9, respectively. Notice that the completion times of p_2 and p_3 remained the same as in the previous example, while

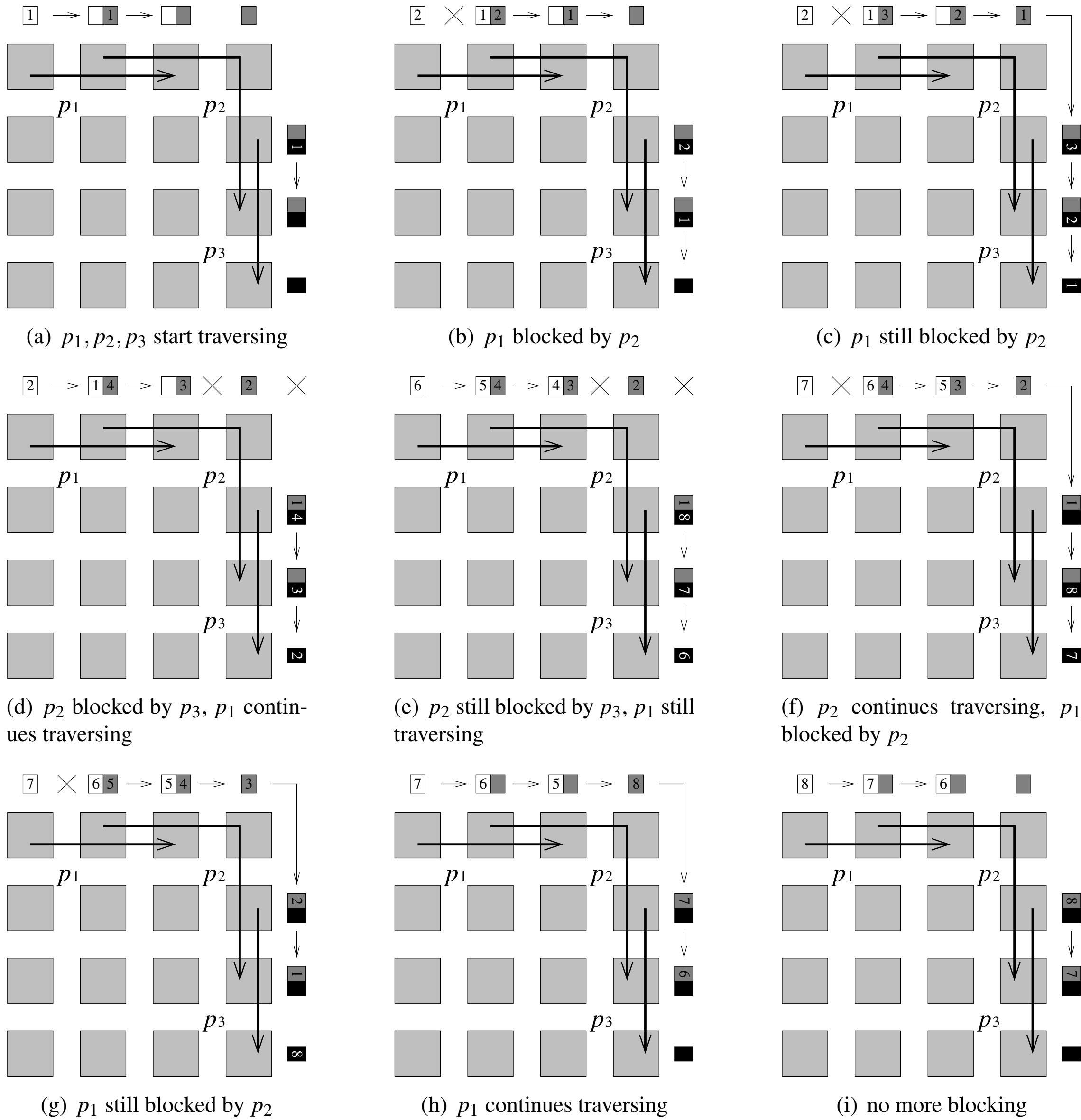


Figure 1.18: Indirect interference with virtual channels

the completion time of p_1 decreased from 21 to 17. This is expected, because the newly introduced rule of dedicated per-packet flit-sized slots within each port indeed reduces the effects of indirect contentions. However, the aforementioned rule does not entirely remove the effects of indirect contentions, it just makes its effects milder. This will receive additional attention in Chapter 2.

This possibility to improve the performance of wormhole-switched networks via dedicated slots has been recognised shortly after the wormhole switching mechanism was introduced. In the literature, this strategy is called the *virtual channels* [24, 25], and some currently available many-cores support this concept, e.g. [42]. If a packet has a dedicated slot in each router along its path, it is said that the packet has a virtual channel. A reader may notice that, counter-intuitively, the expression virtual channel does not refer to the links, but to the slots in port-buffers.

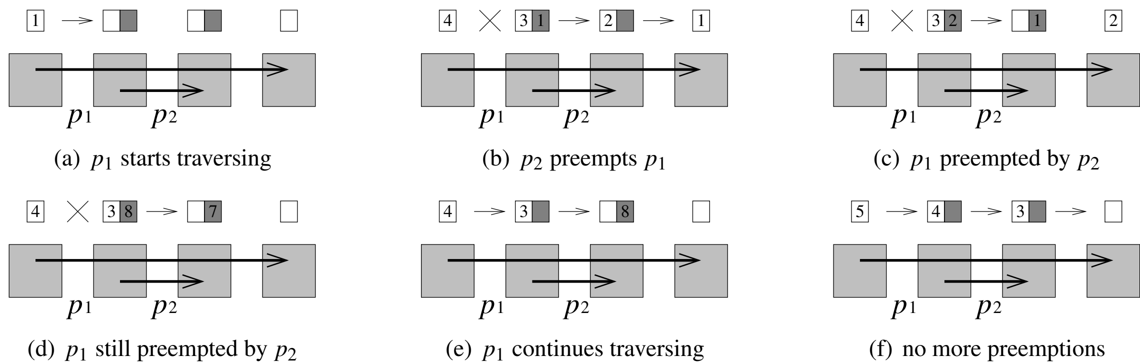
As demonstrated with the examples in Figure 1.17 and Figure 1.18, virtual channels allow for more efficient use of available resources. That is, in the former case, when p_2 was blocked by p_3, p_1 could not use the links along its path, despite the fact that all the links were idle. Conversely, virtual channels always allow to exploit available links, which has been in the latter example demonstrated by a traversal of p_1 while p_2 was blocked.

In this dissertation, the NoCs both with and without virtual channels are analysed.

1.5.2.9 Flit-Level Preemptions and Priority-Preemptive Arbitration

Besides improving the throughput, the concept of virtual channels offers one more very beneficial possibility – to enforce *preemptions* among packets. The preemption is a well-established term in the scheduling theory, and it refers to the situation where the computation process of the lower priority functionality on a core is temporary suspended, so that the computation process of the higher priority functionality can be performed. Similarly, in NoCs, the preemption is the scenario where a lower priority packet is temporary stalled, in order to allow the traversal of a higher priority packet over the resources that these two packets have in common.

Note, in the example illustrated in Figure 1.18 the preemptions are implicitly assumed, and one such scenario is visible in Figure 1.18(f). Due to the fact that p_3 is not blocking p_2 any more, p_2 continues its progress by reclaiming the resources (links) from p_1, causing p_1 to be blocked again.



(a) p_1 starts traversing (b) p_2 preempts p_1 (c) p_1 preempted by p_2 (d) p_1 still preempted by p_2 (e) p_1 continues traversing (f) no more preemptions

Figure 1.19: Packet preemptions

The concept of packet preemptions is explained in detail with the illustrative example given in Figure 1.19. Consider two packets p_1 and p_2, where both packets consist of 8 flits, and where the latter packet is considered to be of higher priority. Let p_1 start traversing (Figure 1.19(a)). At the moment when the first flit of p_1 reached the destination, the packet p_2 was released (Figure 1.19(b)). Due to the fact that p_2 has a higher priority than p_1, the former packet preempts the latter, causing the third and the fourth flit of p_1 to stall within their respective routers, while p_2 progresses (Figure 1.19(c)). This scenario is referred to as the preemption, and p_1 is called the *preempted packet*, while p_2 is called the *preempting packet*. The situation remains the same as long as there are flits of p_2 traversing the common link (Figure 1.19(d)). When the last flit of p_2

traverses the common link, p_1 can continue its transfer (Figure 1.19(e)). After that, p_1 completes its transfer and no more preemptions occur (Figure 1.19(f)).

Since the preemptions occur with the flit-level granularity, this concept is called the *flit-level preemptions* [89]. Notice, that in order to be implemented, this concept requires an adequate arbitration policy. That is, in Figure 1.19(b), when p_2 was released, the arbitration mechanism must compare the priorities of p_1 and p_2, and subsequently make a decision which packet will be given the permission to progress. The arbitration policy which makes decisions based on packet priorities is called the *priority-preemptive arbitration*.

In this dissertation, the priority-preemptive arbitration policy is analysed.

1.5.2.10 Worst-Case Real-Time Analyses Applicable to Priority-Preemptive NoCs

Many researchers have studied the timing properties of priority-aware interconnects. Some of these studies do not analyse the worst-case scenarios, e.g. [16, 38, 89], and hence are not applicable to the hard real-time domain. Of studies that analyse the worst-case scenarios, some have been classified as pessimistic [8, 49], while some other have been classified as optimistic [59, 66]. The limitations of the aforementioned approaches are covered in more detail in the dissertation of Zheng Shi [83].

Later, the method for the worst-case analysis was proposed [85] with the following assumptions: (i) flit-level preemptions, (ii) per-traffic-flow distinctive priorities, (iii) per-priority virtual channels, and (iv) a traffic model with constrained deadlines. For this model the priority assignment algorithm for traffic flows was proposed [84]. For the same model the workload mapping approaches based on genetic algorithms were proposed [62, 80]. Subsequently, the method for the worst-case analysis was extended to cover the model with arbitrary traffic deadlines [88].

One limitation of the aforementioned concept is that the number of virtual channels should be at least equal to the number of traffic-flows, which is a requirement that is not easy to fulfil, e.g. the SCC platform [42] has only 8 virtual channels, and according to this method, it can accommodate at most 8 traffic flows. As a solution to this problem, the novel concept called the *priority share policy* and its accompanying analysis were introduced. This method reduces the total number of required virtual channels by forcing some traffic flows to share the same channel [86]. Based on the priority-share policy and its worst-case analysis, the application mapping process was proposed [87]. Note, that both the aforementioned concepts require that each virtual channels has the capacity to store at least one flit.

Very recently, a novel approach called the *stage level analysis* [46] was introduced. When compared with the aforementioned approaches, this method indeed renders tighter estimates, however, it has the same requirement regarding the number of virtual channels as the initial method [85]. Yet, the fundamental limitation of this approach is the fact that it requires unrealistically big port buffers, where, in many cases, the capacity of virtual channels should be such that an entire packet can be stored [45] (and not a single flit, like in the aforementioned approaches). Notice that this

requirement is very similar to the buffering requirements of the *store-and-forward* switching technique, which is the main reason why the wormhole-switching was introduced in the first place.

1.5.2.11 Discussion

The design choices and the evolution trends of NoCs are mostly driven by performance reasons, usually at the expense of predictability. These strategies not only further increase the gap between the average-case and the worst-case behaviour of NoCs, but also make it increasingly hard to perform any meaningful worst-case analysis. For example, some currently available NoCs (e.g. [3, 93]) use a round-robin arbitration policy in routers, because it promotes fairness and avoids starvation, both of which are beneficial from the performance perspective. However, the round-robin arbitration policy displays a non-deterministic behaviour, where the precedence between two contending packets depends not only on the packets and their paths, but also on the state of routers at the very moment of contention (i.e. which arbitration decisions immediately preceded the moment of observation). Due to this fact, it is impossible to deduce the arbitration decisions at design-time, which subsequently has to be reflected in the worst-case analysis with a certain degree of pessimism. This further implies that the round-robin arbitration policy, despite its wide presence in the currently available many-cores, may not be the most suitable option for the real-time embedded domain.

The existence of virtual channels inside the NoC architecture (e.g. [42]) significantly changes the perspective. Specifically, multiple virtual channels offer the possibility to introduce priority-preemptive arbitration policies, and in that way make the NoC platform more predictable and hence suitable for the worst-case analysis. For instance, assuming that packets have fixed priorities, the one with the higher priority will always have the precedence over the other, irrespective of the moment of observation. This infers that all arbitration decisions are known at design-time, and consequently the worst-case analysis can be performed for priority-preemptive NoCs with substantially less pessimism than the analysis for round-robin-arbitrated NoCs. It comes as no surprise that the majority of the real-time works addressing NoCs indeed assume that the platform provides virtual channels and priority-preemptive arbitration techniques.

In the next chapter the focus will be on improving over the state-of-the-art methods for both types of platforms, with and without virtual channels. Moreover, assuming that multiple virtual channels are available, a novel arbitration policy for routers will be proposed. These aspects are covered in detail in Chapter 2.

1.5.2.12 Assumptions Regarding Interconnect Medium

Recall (Section 1.5.1.6), that the platform under consideration in this dissertation Ψ contains a set of z processing elements $\Pi = \{\pi_1, \pi_2, ..., \pi_{z-1}, \pi_z\}$. Moreover, the platform contains an interconnect medium η, which is a 2-D mesh NoC of dimensions x and y. The NoC contains a collection of $y \cdot x$ routers $\rho = \{\rho_1, \rho_2, ..., \rho_{y \cdot x-1}, \rho_{y \cdot x}\}$, where the number of routers is equal to the number of cores, i.e. $y \cdot x = z$. Thus, one router corresponds to one core, like in Epiphany [3], or Tilera [93]

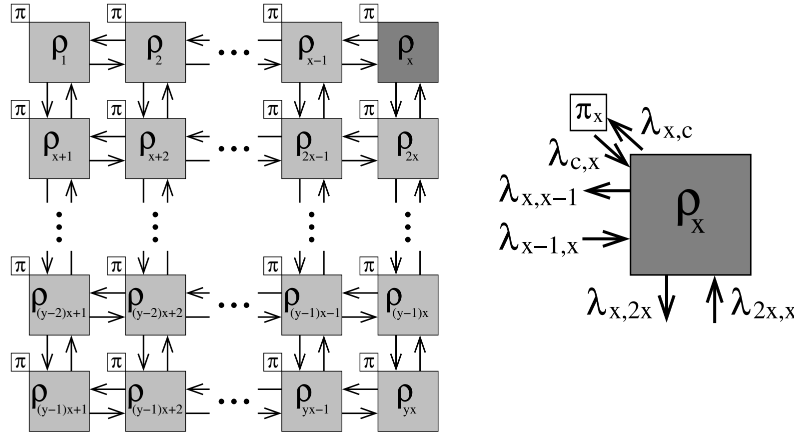


Figure 1.20: Assumed NoC platform

platforms. Figure 1.20 depicts the assumed NoC platform. Each router can have at most 10 other ports, namely *East Input*, *East Output*, *West Input*, *West Output*, *North Input*, *North Output*, *South Input*, *South Output*, *Core Input* and *Core Output*. A router uses core ports to exchange the data with the local core, while the rest of the ports are used for the communication between the router and its neighbouring routers. The number of ports inside a router depends on the position of the router within the NoC. Central routers have all 10 ports (e.g. ρ_{x+2} in Figure 1.20), routers on the edges have 8 (e.g. ρ_2 in Figure 1.20), whereas corner routers have only 6 ports (e.g. ρ_1 in Figure 1.20).

Each pair of communicating routers is connected with two unidirectional links, between their closest ports. For instance, in Figure 1.20 the routers ρ_{x-1} and ρ_x are connected with two unidirectional links $\lambda_{x-1,x}$ and $\lambda_{x,x-1}$, where the first number represents the source router of the link and the second number denotes the destination router. Therefore, over $\lambda_{x,x-1}$, the packets are transferred from the west output port of ρ_x to the east input port of ρ_{x-1}. Moreover, the core ports of each router ρ_i, are connected with the local core π_i via two unidirectional links $\lambda_{i,c}$, and $\lambda_{c,i}$. Note, that for clarity purposes, the links between the cores and the routers have been omitted from the left part of Figure 1.20.

The platform employs the X-Y routing policy and a wormhole switching technique with the credit-based flow control mechanism. All links have identical physical characteristics. The width of each link is σ_{flit}, which is equal to the size of one flit. All flits travel with the same speed, and it takes δ_L clock cycles for a flit to travel between the output port of the sender and the input port of the receiver router, i.e. to traverse one link. For example, it takes δ_L clock cycles for a flit from the core π_x to traverse the link $\lambda_{c,x}$ and enter the core input port of ρ_x, and it also takes δ_L clock cycles for a flit from the west output port of ρ_x to traverse $\lambda_{x,x-1}$ and enter the east input port of ρ_{x-1}. Additionally, the flits travel through routers. A header flit contains the information about the destination, which is used to pre-compute the path which will be followed by the remaining flits. Therefore the routing delay is suffered only by a header flit and it takes δ_ρ clock cycles for a header to traverse one router. All routers work on the same frequency ν_ρ.

Moreover, the NoC η may or may not provide the support for virtual channels. In this dissertation, both types of NoCs will be analysed. Therefore, let η_{RR} and η_{PP} be the NoCs without and with virtual channels, respectively. For η_{RR}, it is assumed that the round-robin arbitration policy is used. Moreover, the capacity of the buffers inside ports should be such that at least 1 flit can be stored, i.e. σ_{flit}. As explained in previous sections, assuming a single channel, it is not possible to implement inter-packet preemptions. Note that the Epiphany [3] and Tilera [93] platforms, in terms of their characteristics, can be classified as η_{RR}.

For η_{PP}, the number of virtual channels should be at least equal to the maximum number of traffic-flows. This requirement guarantees that each packet can be (if needed) successfully stored inside a dedicated virtual channel along its path. Subsequently, each virtual channel should be able to accommodate at least 1 flit. Moreover, the priority-preemptive arbitration mechanism is assumed, where a higher-priority packet can preempt a lower-priority packet with the flit-level granularity.

1.5.3 Data Input and Output

In many cases, in order to perform the computation, functionalities need some input data. Depending on the purpose of the system, the required data can be (i) values sensed from the environment with which the device is interacting, and/or (ii) the results from the previous computations of the functionality itself, as well as other functionalities interacting with it. When the computation is performed, in-core registers are used to manipulate the data. However, the capacity of the registers is not enough to store the necessary data for all functionalities, not even for a single functionality. Thus, there must exist an external medium where data can be stored, and there must exist an accompanying mechanism which performs the data transfer between the registers and the external medium, i.e. to send data to the medium when it is not needed any more, and retrieve it when it is needed again. The part of the hardware that is responsible for these operations is called the *memory system*.

1.5.3.1 Memory Systems in Single-Core and Multi-Core Platforms

Over the years, the memory systems underwent a multitude of changes. Chip manufacturers decided to implement changes in order to improve the performance of the system, usually at the expense of a more complex design. These evolution trends of memory systems are very beneficial for some areas, such as the high-performance or general purpose computing, where the average-case behaviour (performance) is an important aspect. However, a more complex design, at the same time, makes it increasingly hard to analyse the temporal behaviour of the memory system, which is an important aspect in other areas, such as the real-time embedded domain. It is no surprising that, as the memory design grows more complex, new challenges in the real-time analysis of memory systems keep emerging.

Figure 1.21 depicts a typical memory system of a single-core platform. As already stated, the computation process uses in-processor registers. When the data that is needed for the computation

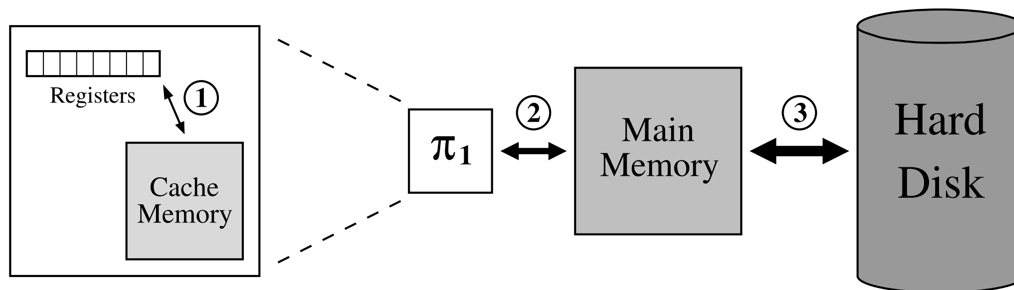


Figure 1.21: Memory system of single-core devices

is not present in the registers, an in-processor memory called the *cache memory* is checked for the needed content (see Arrow ① in Figure 1.21). The cache memory is a fast memory with a limited capacity, usually in the range 1 – 64 kilobytes. If the required data is not in the cache memory, it has to be looked for outside of the core. In such cases, the module called *the main memory* is checked (see Arrow ② in Figure 1.21). When compared with the cache memory, the main memory can store much more data, e.g. in currently available single- and multi-core computers the capacity of the main memory is in the range of gigabytes. However, the main memory is significantly slower than the cache memory, and fetching the data also requires the communication between the processor and the external memory module, usually over the part of the NoC dedicated to the memory traffic, while fetching the data from the cache memory is performed inside the processor.

Finally, if the desired data is not in the main memory, the component called the *hard disk* is used to fetch the data (see Arrow ③ in Figure 1.21). The comparison between the main memory and the hard disk is very similar to that between the cache memory and the main memory. In other words, the capacity of currently available hard disks is in the range of terabytes, thus the hard disk can store much more data than the main memory. However, the access to the hard disk is more time consuming than the access to the main memory.

From the previous description and illustration it can be concluded that the cache memory and the main memory are used as the means to bridge the gap between the latencies of accessing the data that is (i) in the registers and (ii) on the hard disk. Evidently, the presence and absence of the required data in the cache and the main memory has a significant impact on the performance of the system. Thus, one of the main objectives of chip manufacturers is to organise the data transfer such that the required data is as close to the core as possible, at the very moment when it is needed. There are many techniques which are used to achieve this, the most popular ones employ prediction heuristics, which are based on temporal and spatial locality principles, however, in this dissertation, the data transfer policies are not of interest.

In Figure 1.22 is depicted a typical memory system for the multi-core platform. It is visible that this design has one more element called the *Level* 2 *cache memory*. This memory module is usually shared with two neighbouring cores, and it is often referred to as the *shared cache*, to distinguish from the in-core cache memories, which are, for the same reason, called the *private caches*. The characteristics of the shared cache fall between the private cache and the main memory, both in terms of the capacity and the access latency.

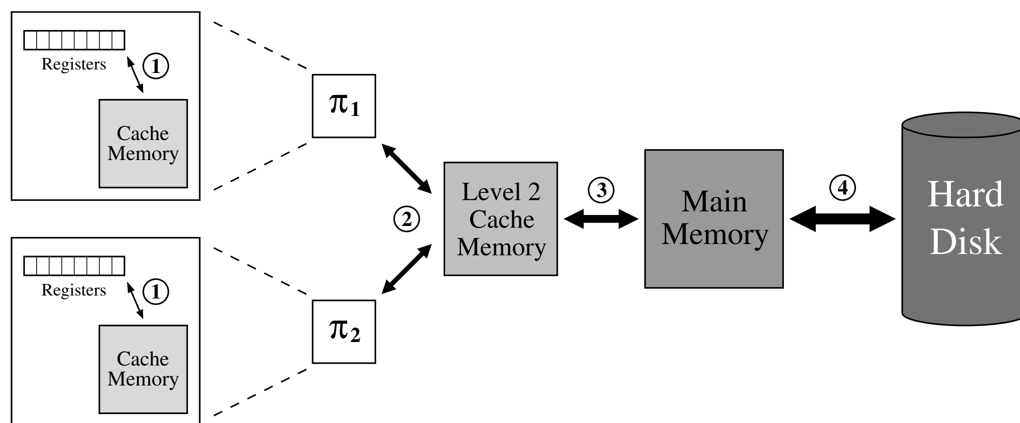


Figure 1.22: Memory system of multi-core devices

Notice, that Figures 1.21-1.22 are examples of memory systems in single-core and multi-core platforms. However, there exist designs where a core has multiple levels of internal cache, not only one. Similarly, cores may share multiple levels of shared cache, where the number of cores that share the cache usually grows with the level, e.g. the first level shared cache is shared by two cores, the second level shared cache is shared by four cores, etc.

1.5.3.2 Consistency and Coherence

Consider a case where two functionalities share some parameter. Moreover, assume that one functionality modified that parameter. Subsequently, the other functionality mast be instantly notified about that change, so as to take it into account in its future computations. When such a mechanism is in place, it is said that the system is in a *consistent state*. Having a consistent state is an imperative for a correct functioning of the system. Notice, that unless some mechanisms to enforce parallelism are applied, in a single-core system, at any time instant, at most one functionality may execute and hence manipulate the data. Therefore, maintaining a consistent state in the single-core domain is usually trivial. However, in multi-core platforms, several functionalities may concurrently perform computations on different cores. This infers that there must exist a mechanism which will maintain the consistent system state.

Consider the multi-core case where two functionalities share some parameter. Moreover, assume that the functionalities execute concurrently, and that the shared parameter is in both private caches. Once the parameter has been modified by one functionality, the value of that parameter in the private cache of the other functionality also has to be modified. Specifically, assuming the memory architecture from Figure 1.22, once the functionality changes the value of the parameter, the new value has to be propagated back into the shared cache, and then into the private cache of the other functionality. Furthermore, if these two functionalities do not have a common shared cache, the information has to be propagated back into the main memory, and then into the private cache of the other functionality. This process is called the *memory coherence mechanism*.

Notice, that the coherence mechanism can generate significant traffic. In the corner cases, for a single modified parameter, it might be necessary to propagate the information back to the main memory, and subsequently broadcast that information to all remaining cores within the platform. Evidently, the coherence mechanism is not scalable, and as the number of cores grows, the overheads become more apparent. In fact, due to the coherence-related overheads, in some cases adding more cores can have a negative impact on the system and cause performance drops [11]. Another limitation of this approach is the fact that it is very hard, if not impossible, to efficiently analyse the temporal behaviour of coherence mechanisms at design-time, because the decisions taken by the coherence mechanism highly depend on the system state at runtime.

1.5.3.3 Message-Passing

Despite the fact that the coherence mechanism can be used to assure a consistent system state, it is not applicable to the many-core domain due to its poor scalability potential. In fact, yet another key difference between multi- and many-core platforms lays in the fact that for the former, the coherence mechanism presents an efficient approach, while for the latter, an alternative mechanism is necessary.

An alternative to the coherence mechanism is called the *message-passing technique*. A long time ago it has been shown that the coherence and the message passing are dual approaches [53, 55], and that the dominance of one over another depends on a platform upon which it is implemented. However, for many years the coherence mechanism was a predominant choice. Yet, as many-core platforms emerged, the limitations of the coherence mechanism became evident, and the academia as well as the industry focused on the message-passing technique [52].

The message passing paradigm is build around the principles that are opposite to those of coherence mechanisms. For instance, the message passing approach suggests that data should not be shared between multiple processes, which is why it is also called the *share nothing approach*. Specifically, instead, of sharing the same parameter by two or more functionalities, the parameter should independently exist within the scope of each functionality, while for any modification of the parameter, an explicit messages should be exchanged between respective functionalities.

Notice, that the problem of consistency is now elevated from the hardware (coherence mechanisms) to the software (exchange of messages). In other words, it is the responsibility of implementers and system designers to assure that the execution of functionalities does not cause an inconsistent system state. However, the greatest benefit of this technique is that it is scalable and can be successfully applied to the many-core domain [11, 52]. Another benefit is that data dependencies among functionalities are known at design-time, and so are the messages that must be exchanged in order to maintain a consistent state. This implies that the message-passing approach is also much more predictable and easier to analyse, which carries a significant importance in the real-time domain.

1.5.3.4 Hardware Support for Message-Passing

Many existing many-core platforms rely on the message-passing mechanism to maintain the consistent system state, e.g. Epiphany [3], SCC [42] and MPPA-256 [44], while in some other platforms both the coherence and the message-passing mechanisms are supported, e.g. Tile 64™ [93]. For instance, inside each router of the SCC platform, a buffer called the *message-passing buffer* exists, and it presents a hardware support for the message-passing mechanism. In Tile 64™ a feature called the *dynamic distributed cache* is present, which allows access to caches of remote cores, also as a support to the message-passing mechanism. In the Epiphany platform each core is mapped to a specific region in the main memory, which eases the inter-core communication and promotes the message-passing.

Evidently, the trends of the chip manufacturers suggest that future many-core platforms will be non-coherent systems where the consistent state will be assured via a message-passing mechanism. Recall, that the main reason to study the message-passing technique was the fact that it was the only viable approach for many-cores. It is also worth mentioning that its predictable nature allows to efficiently analyse the system behaviour at design-time, and makes it suitable to the real-time analysis. Thus, this is another case where the trends in the many-core design are also beneficial for the real-time domain.

1.5.3.5 Assumptions Regarding Memory System

Recall, in Sections 1.5.1.6-1.5.2.1 it was mentioned that the platform under consideration Ψ contains the set of processing elements Π and the interconnect medium η. In addition to that, the platform contains the non-coherent memory system Ξ. The memory space of Ξ is divided among 4 different memory controllers $\mu = \{\mu_1, \mu_2, \mu_3, \mu_4\}$, where each controller arbitrates the accesses to a different part of the address space. Moreover, there are $z = x \cdot y$ private cache memories in the system $\kappa = \{\kappa_1, \kappa_2, ..., \kappa_{z-1}, \kappa_z\}$, where z is equal to the number of cores in the system and also to the number of routers, i.e. each core has a private cache, like depicted in Figure 1.23. Note, that for clarity purposes the cores have been omitted from the figure.

A functionality gets its data from the cache of its core. If the required data is not in the cache, it is fetched from the memory controller. The communication between the cache memory and the memory controller is performed over the NoC interconnect η. The memory controllers are accessible from the topmost or the bottommost row of routers (see Figure 1.23). Additionally, each controller provides a concurrent access to $\frac{x}{2}$ routers from its access row, e.g. the controller μ_1 to the routers $\rho_1, \rho_2, ..., \rho_{\frac{x}{2}-1}, \rho_{\frac{x}{2}}$, the controller μ_2 to the routers $\rho_{\frac{x}{2}+1}, \rho_{\frac{x}{2}+2}, ..., \rho_{x-1}, \rho_x$, etc.

Memory read and write requests originate from the core of the functionality, and traverse the X-Y routed path until reaching the memory controller. Similarly, responses originate from the memory controller, and traverse the X-Y routed path to the core from which the request was sent. Note, that in this dissertation, the focus is only on the delays of memory traffic over the NoC, while the delays occurring inside the memory controller are out of scope, and have been extensively

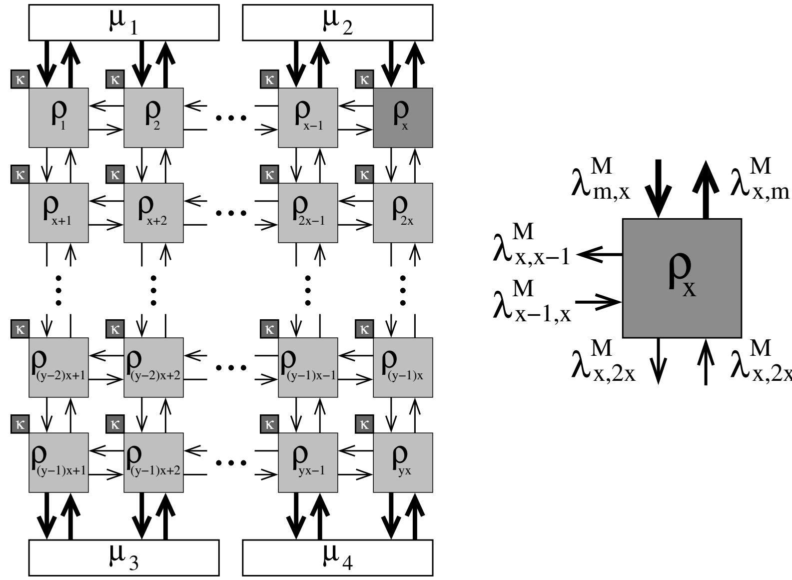


Figure 1.23: Assumed memory system

studied in the following studies [78, 95]. Merging the contributions of this dissertation, with those of the mentioned works, into a unified approach, is a potential future work.

Memory traffic uses dedicated NoC resources: dedicated physical links and dedicated port buffers, which have identical physical characteristics to those dedicated to the communication traffic, e.g. between the routers ρ_{x-1} and ρ_x in Figure 1.23 there are two unidirectional links $\lambda^M_{x-1,x}$ and $\lambda^M_{x,x-1}$, and the link and routing delays are δ_L and δ_ρ, respectively. Similarly, between the memory controller μ_2 and the router ρ_x there are two unidirectional links $\lambda^M_{x,m}$ and $\lambda^M_{m,x}$. Also, between the core π_x and the router ρ_x there are two unidirectional links $\lambda^M_{x,c}$, and $\lambda^M_{c,x}$. Note, that for clarity purposes, the cores and their respective links were not depicted in Figure 1.23. Moreover, the memory traffic is arbitrated in the same way as the communication traffic, that is, if the latter is arbitrated with the round-robin policy, so is the former, while if the latter employs the priority-preemptive policy, so does the former. Thus, the communication and memory traffic do not contend, and the respective analyses can be performed separately.

In this dissertation, the focus is on non-coherent many-core platforms with the message-passing mechanism.

1.5.4 Recapitulation

In the previous sections, the elements of interest of the assumed platform Ψ were introduced. For better readability, all symbols that have been previously introduced are summarised in Table 1.1.

Table 1.1: List of symbols related to the assumed platform

Symbol	Description
Ψ	The assumed platform.
Π	The set of processing elements (cores) of Ψ.
z	The number of cores in Π.
π_i	The ith core of Π, where $i \in \{1,...,z\}$.
η	A generic 2-D mesh NoC interconnect of Ψ.
η_{RR}	η with the round-robin arbitration policy and without virtual channels.
η_{PP}	η with the priority-preemptive arbitration policy and with virtual channels.
x	The horizontal dimension of η, where $x \cdot y = z$.
y	The vertical dimension of η, where $x \cdot y = z$.
ρ	The set of routers of η.
ρ_i	The ith router of ρ, where $i \in \{1,...,z\}$.
λ	The set of links of η dedicated to the communication traffic.
$\lambda_{i,j}$	The communication link from ρ_i (source) to ρ_j (destination).
$\lambda_{c,i}$	The communication link from π_i (source) to ρ_i (destination).
$\lambda_{i,c}$	The communication link from ρ_i (source) to π_i (destination).
σ_{flit}	The size of one flit and the width of each link.
δ_L	The number of clock cycles that it takes for one flit to traverse one link.
δ_ρ	The number of clock cycles that it takes for a header flit to traverse one router.
ν_ρ	The frequency of routers.
Ξ	The non-coherent memory system of Ψ.
μ	The set of memory controllers of Ξ.
μ_i	The ith controller of μ, where $i \in \{1,2,3,4\}$.
κ	The set of cache memories of Ξ.
κ_i	The ith cache memory of κ, where $i \in \{1,...,z\}$.
λ^M	The set of links of η dedicated to the memory traffic.
$\lambda^M_{i,j}$	The memory link from ρ_i (source) to ρ_j (destination).
$\lambda^M_{c,i}$	The memory link from π_i (source) to ρ_i (destination).
$\lambda^M_{i,c}$	The memory link from ρ_i (source) to π_i (destination).
$\lambda^M_{m,i}$	The memory link from the memory controller (source) to ρ_i (destination).
$\lambda^M_{i,m}$	The memory link from ρ_i (source) to the memory controller (destination).

1.6 Thesis Statement

Many-core platforms are the imperative and the *sine qua non* in the design of future real-time embedded systems. With:

- adequate hardware support for (i) the message passing communication paradigm and (ii) virtual channels,
- operating system design choices which promote scalability and predictability via message-passing,
- mindful and thoughtful worst-case analyses,

it is possible to make many-cores amenable to real-time analysis and applicable to the real-time embedded domain.

Chapter 2

Several Steps Closer to Real-Time NoCs

In this chapter, a set of contributions concerning the NoC interconnect η of the assumed many-core platform Ψ is presented. The common aim of all contributions is to make the existing NoC platforms more amenable to the real-time domain, by reducing the analysis pessimism and/or by reducing the hardware requirements. This chapter is organised as follows. In Section 2.1 the traffic model is introduced. This model is used to analytically describe the communication and memory traffic. Then, in Section 2.2, the existing method for the worst-case traffic analysis of round-robin-arbitrated NoCs is covered, and subsequently the novel, less pessimistic method called the *Branch, Prune and Collapse* is proposed. In Section 2.3 the focus is on priority-preemptive NoCs. First, the existing approach for the worst-case traffic analyses is introduced, and then the new method to reduce the requirements for hardware resources is proposed. After that, in Section 2.3.4 the novel arbitration policy for priority-preemptive NoCs is covered. This policy is based on the Earliest Deadline First scheduling paradigm. Finally, in Section 2.3.5 an improvement over the existing methods for priority-preemptive NoCs is presented. This improvement allows to perform a less pessimistic analysis.

2.1 Traffic Model

The communication (core-to-core) traffic, as well as the memory (core-to-memory) traffic is modelled by a sporadic flow-set $\mathscr{F}$, which is a collection of w flows $\mathscr{F} = \{f_1, f_2, ..., f_{w-1}, f_w\}$. Each flow f_i has a source $src(f_i)$, and a destination $dst(f_i)$, where the source and the destination can be cores or memory controllers, i.e. $src(f_i) \in \{\pi \cup \mu\} \wedge dst(f_i) \in \{\pi \cup \mu\}$. Additionally, f_i is characterised by a set of traversed links $\mathscr{L}(f_i)$, its size $\sigma(f_i)$, its minimum inter-arrival period $T(f_i)$, its deadline $D(f_i)$, and its priority $P(f_i)$. Each flow f_i generates a potentially infinite sequence of packets. A packet released from $src(f_i)$ at the time instant t should be received by $dst(f_i)$ no later than $t + D(f_i)$. Otherwise, it has *missed a deadline*. In this dissertation, it is assumed that all flows have constrained or implicit deadlines, i.e. $\forall f_i \in \mathscr{F} \mid D(f_i) \leq T(f_i)$.

In Figure 2.1 are depicted 3 flows, f_1, f_2 and f_3. From the figure it is visible that the flow characteristics are as follows:

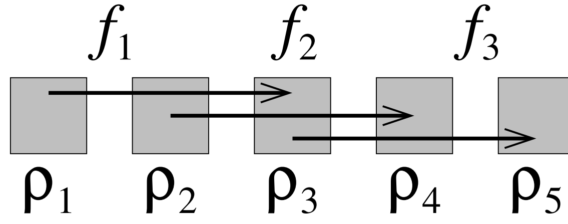


Figure 2.1: Traffic flows (example 1)

$$src(f_1) = \pi_1,\, dst(f_1) = \pi_3,\, \mathscr{L}(f_1) = \{\lambda_{c,1}, \lambda_{1,2}, \lambda_{2,3}, \lambda_{3,c}\}.$$
$$src(f_2) = \pi_2,\, dst(f_2) = \pi_4,\, \mathscr{L}(f_2) = \{\lambda_{c,2}, \lambda_{2,3}, \lambda_{3,4}, \lambda_{4,c}\}.$$
$$src(f_3) = \pi_3,\, dst(f_3) = \pi_5,\, \mathscr{L}(f_3) = \{\lambda_{c,3}, \lambda_{3,4}, \lambda_{4,5}, \lambda_{5,c}\}.$$

Notice that the flows f_1 and f_2 are directly competing because of the common link $\lambda_{2,3}$. The same is also true for the flows f_2 and f_3, because of the common link $\lambda_{3,4}$. However, f_1 and f_3 are not directly competing because they do not have any common link.

2.2 NoCs with the Round-Robin Arbitration Policy

Irrespective of the type of the traffic (communication or memory), and irrespective of the arbitration policy (round-robin or priority preemptive), if every packet of one flow meets its deadline, then that flow is considered *schedulable*. Subsequently, if every flow of the flow-set is schedulable, then the flow-set is considered *schedulable*. The schedulability can be investigated via simulations or measurement-based techniques, however, as mentioned in Section 1.2, these methods have several fundamental limitations. In the hard real-time domain the most common approach to test the schedulability is to perform the worst-case traffic analysis. Specifically, it is investigated whether a packet of the analysed flow can meet a deadline, while facing the worst-case occurrence patterns of packets belonging to other flows. This case is called the worst-case scenario. If the packet of the analysed flow can meet a deadline in the worst-case scenario, it means that every packet of that flow will be able to meet its deadline, and hence the flow is schedulable.

One of the key challenges is to recognise and analytically describe the worst-case scenario. Depending on the conditions, finding the worst-case scenario can be a relatively straightforward activity, or it can be an intractable process, and both situations will be encountered throughout this dissertation. In the latter case, as will be shown in this very section, the goal is to identify an artificial worst-case scenario. Its advantages are that it is easier to capture that the actual worst-case scenario and that it is safe, i.e. never leads to an underestimation. However, its disadvantage is that it is more pessimistic than the actual worst-case scenario.

2.2.1 State-of-the-Art Method

Ferrandiz et al. [32] proposed a method to perform the worst-case analysis for round-robin-arbitrated SpaceWire networks, and this method is also applicable to NoCs. This approach is

based on two observations:

Observation 1. *A packet may, at any router, be blocked by at most one packet from each of the other input ports.*

Observation 2. *A packet may be directly blocked by at most one packet of each flow.*

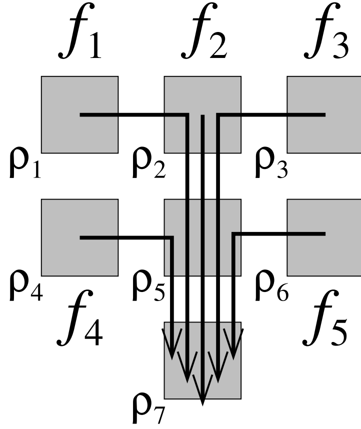


Figure 2.2: Traffic flows (example 2)

Observations 1-2 are the consequence of the round-robin arbitration policy. Figure 2.2 is used to illustrate the implications of these observations. Consider that the flow f_2 is under analysis. Upon its release, it can be blocked by f_1 and f_3. However, after the blocking, when it progresses to ρ_5, it cannot suffer any more blocking from f_1 and f_3 (Observation 2). Moreover, if there exist other flows f_1' and f_3', with the same paths as f_1 and f_3, respectively, only one flow from each direction would be able to block f_2 (Observation 1). That is, only one of f_1 and f_1' would be able to block f_2, and the same is true for f_3 and f_3'.

Although a flow can be blocked directly by any flow at most once, it may happen that the analysed flow is additionally blocked by the same flow multiple times indirectly. In other words, in Figure 2.2, when f_2 is blocked by f_1, then f_1 can be blocked by f_4 and f_5 in ρ_5, which causes indirect blocking to f_2. However, when f_2 reaches ρ_5, it can also be blocked by f_4 and f_5, but this time directly. In fact, in the illustrated example, f_2 can be blocked three times by f_4 and f_5, two times indirectly when f_1 and f_3 reach ρ_5, and once directly when itself reaches ρ_5. This scenario is illustrated with the following flow traversal sequence: $f_4, f_5, f_1, f_4, f_5, f_3, f_4, f_5, f_2$.

Ferrandiz et al. [32] proposed to compute the worst-case traversal time (WCTT) of flows by invoking the Algorithm 1, with the following parameters $delay(f_i, get(\mathscr{L}(f_i), 1))$, where f_i is the flow under analysis, and $get(\mathscr{L}(f_i), 1)$ denotes the first link on the path $\mathscr{L}(f_i)$. First, it is checked whether $\lambda_{m,n}$ is null (line 2). If it is, this means that the header of f_i reached the destination, so the only remaining activity is to transfer the rest of the flits (line 3). Otherwise, it is checked whether $\lambda_{m,n}$ is the first link on the flow path. If it is, this means that f_i can be blocked on $\lambda_{m,n}$ only by the flows with the same source as f_i. All such flows are grouped into the set $\mathscr{F}_{src}$ (lines $7-9$). Then, WCTT of f_i is equal to the sum of: (i) the delay of all flows from $\mathscr{F}_{src}$ to traverse $\lambda_{m,n}$, (ii) the delay of all flows from $\mathscr{F}_{src}$ to reach the destination, (iii) the delay of f_i to traverse $\lambda_{m,n}$, and (iv) the delay of f_i to reach the destination (lines $10-13$).

Conversely, if $\lambda_{m,n}$ is not the first link of f_i, then f_i can suffer blocking only from flows which also traverse $\lambda_{m,n}$, but enter the router ρ_m from a different link than the one from which f_i does (which is $\lambda_{k,m}$). Thus, all such links are added to the set of blocking links BL (lines $18-20$).

For each blocking link $\lambda_{g,m} \in BL$, a set of flows $\mathscr{F}_{g,m}$ is created. $\mathscr{F}_{g,m}$ contains all flows that traverse $\lambda_{g,m}$ and subsequently compete with f_i on $\lambda_{m,n}$ (lines $21-26$). For each $\lambda_{g,m} \in BL$, a

Algorithm 1 $delay(f_i, \lambda_{m,n})$

Input: flow f_i, link $\lambda_{m,n}$

Output: WCTT of f_i, starting from $\lambda_{m,n}$, to the destination

1: **if** $(\lambda_{m,n} = null)$ **then**

2: // the header flit reached the destination, so the remaining flits have to be transferred

3: **return** $\left\lceil \frac{\sigma(f_i)}{\sigma_{flit}} \right\rceil \cdot \delta_L$;

4: **end if**

5: **if** $(get(\mathscr{L}(f_i), 1) = \lambda_{m,n})$ **then**

6: // $\lambda_{m,n}$ is the first link of f_i, so only the flows with the same source as f_i can block

7: **for each** $(f_j \in \mathscr{F} \mid f_j \neq f_i \wedge get(\mathscr{L}(f_j), 1) = \lambda_{m,n})$ **do**

8: $\mathscr{F}_{src} \leftarrow f_j$; // find all flows with the same source as f_i

9: **end for**

10: $VAL_1 \leftarrow \sum_{\forall f_j \in \mathscr{F}_{src}} (\delta_L + delay(f_j, get(\mathscr{L}(f_j), 2)))$;

11: $VAL_2 \leftarrow \delta_L + delay(f_i, get(\mathscr{L}(f_j), 2))$;

12: $WCTT \leftarrow VAL_1 + VAL_2$;

13: **return** $WCTT$;

14: **end if**

15: // find $\lambda_{k,m}$, which f_i traversed immediately before $\lambda_{m,n}$

16: $\lambda_{k,m} \leftarrow getprev(\mathscr{L}(f_i), \lambda_{m,n})$;

17: // f_i can be blocked by flows which traverse $\lambda_{m,n}$, but have previous link different than $\lambda_{k,m}$

18: **for each** $(g \in \mathbb{N} \mid \lambda_{g,m} \in \lambda \wedge g \neq k)$ **do**

19: $BL \leftarrow \lambda_{g,m}$;

20: **end for**

21: **for each** $(g \in \mathbb{N} \mid \lambda_{g,m} \in BL)$ **do**

22: // find the set of flows $\mathscr{F}_{g,m}$ that traverse $\lambda_{g,m}$ and $\lambda_{m,n}$, and hence can block f_i

23: **for each** $(f_j \in \mathscr{F} \mid \lambda_{g,m} \in \mathscr{L}(f_j) \wedge \lambda_{m,n} \in \mathscr{L}(f_j))$ **do**

24: $\mathscr{F}_{g,m} \leftarrow f_j$;

25: **end for**

26: **end for**

27: $VAL_1 \leftarrow \sum_{\forall \lambda_{g,m} \in BL} \max_{\forall f_j \in \mathscr{F}_{g,m}} \{\delta_\rho + \delta_L + delay(f_j, getnext(\mathscr{L}(f_j), \lambda_{m,n}))\}$;

28: $VAL_2 \leftarrow \delta_\rho + \delta_L + delay(f_i, getnext(\mathscr{L}(f_i), \lambda_{m,n}))$;

29: $WCTT \leftarrow VAL_1 + VAL_2$;

30: **return** $WCTT$;

single flow from $\mathscr{F}_{g,m}$ is found, such that it can cause the maximum delay to f_i. Subsequently, for all such flows the following values are computed: (i) the delay to traverse ρ_m, (ii) the delay to traverse $\lambda_{m,n}$, and (iii) the delay to reach the destination. All these delays jointly contribute to the delay of f_i (line 27). Then, the following values are computed: (i) the delay of f_i to traverse ρ_m, (ii) the delay of f_i to traverse $\lambda_{m,n}$, and (iii) the delay of f_i to reach the destination (line 28). Finally, the worst-case traversal time of f_i is equal to the sum of these terms (line 29).

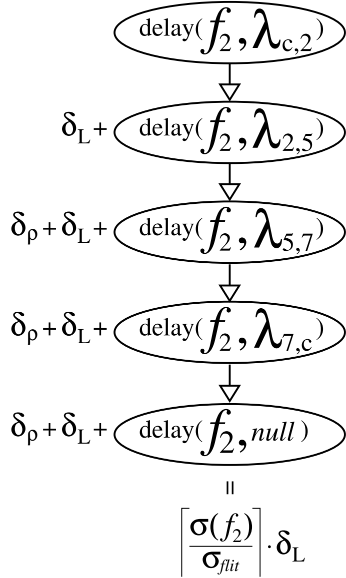


Figure 2.3: Computation tree for flow f_2 from Figure 2.2, in isolation (Ferrandiz et al. [32] method)

The execution of Algorithm 1 is demonstrated by using the example given in Figure 2.2. Consider that f_1, f_3, f_4 and f_5 do not exist, and that f_2 is the only flow, which, due to inexistence of other flows, can uninterruptedly progress. By invoking Algorithm 1 for this example, the computation process follows the steps illustrated in Figure 2.3, and returns the following result:

$WCTT(f_2) = 4 \cdot \delta_L + 3 \cdot \delta_\rho + \left\lceil \frac{\sigma(f_2)}{\sigma_{flit}} \right\rceil \cdot \delta_L$.

This result is intuitive, because f_2 traverses four links (four hops) and three routers, so its header flit suffers the link traversal delay four times, and the routing delay three times before reaching the destination. Additionally, when the header flit is received, the rest of the flits are transferred, and the last term in the aforementioned computation corresponds to that delay.

This example allows to make a more general observation:

Observation 3. *Consider a schedulable flow-set $\mathscr{F}$. A packet of a flow $f_i \in \mathscr{F}$, when traversing in isolation, will suffer a constant delay $C(f_i)$, expressed with Equation (2.1).*

$$C(f_i) = |\mathscr{L}(f_i)| \cdot \delta_L + (|\mathscr{L}(f_i)| - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma(f_i)}{\sigma_{flit}} \right\rceil \cdot \delta_L \quad (2.1)$$

Recall, that $|\mathscr{L}(f_i)|$ denotes the cardinality of the path, which is also called the number of hops, and it represents the number of links that f_i traverses, i.e. the number of elements in the $\mathscr{L}(f_i)$ set.

$C(f_i)$ is in the literature also called the *basic network latency* [30]. When the flow traverses in isolation, its worst-case traversal time is equal to its basic network latency: $WCTT(f_i) = C(f_i)$, which also holds for the previous example: $WCTT(f_2) = C(f_2) = 4 \cdot \delta_L + 3 \cdot \delta_\rho + \left\lceil \frac{\sigma(f_2)}{\sigma_{flit}} \right\rceil \cdot \delta_L$. However, in the presence of other flows, f_i may suffer additional delay, which can make $WCTT(f_i)$ significantly greater than $C(f_i)$. Thus, $C(f_i)$ it is the minimum delay that any packet of f_i may

suffer while traversing the NoC, and $C(f_i) \leq D(f_i)$ is a necessary but not a sufficient condition for the schedulability of f_i.

Now, consider the example from Figure 2.2, but this time assuming all depicted flows. By invoking Algorithm 1 to compute $WCTT(f_2)$, the computation trees illustrated in Figures 2.4-2.5 will be generated. The computation process returns the following value:

$$WCTT(f_2) = 22 \cdot \delta_L + 21 \cdot \delta_\rho + 3 \cdot \left(\left\lceil \frac{\sigma(f_4)}{\sigma_{flit}} \right\rceil + \left\lceil \frac{\sigma(f_5)}{\sigma_{flit}} \right\rceil \right) \cdot \delta_L + \left(\left\lceil \frac{\sigma(f_1)}{\sigma_{flit}} \right\rceil + \left\lceil \frac{\sigma(f_3)}{\sigma_{flit}} \right\rceil + \left\lceil \frac{\sigma(f_2)}{\sigma_{flit}} \right\rceil \right) \cdot \delta_L$$

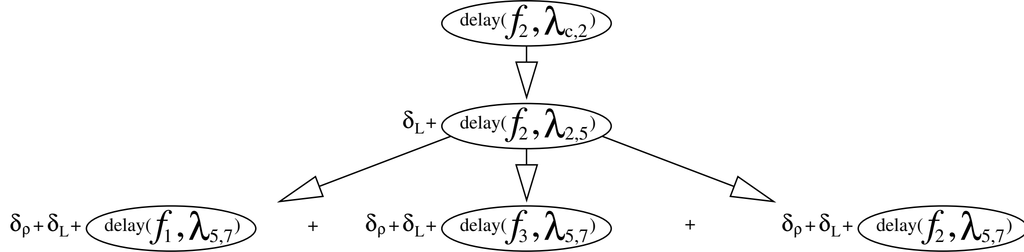


Figure 2.4: Computation tree for flow f_2 from Figure 2.2 (Ferrandiz et al. [32] method)

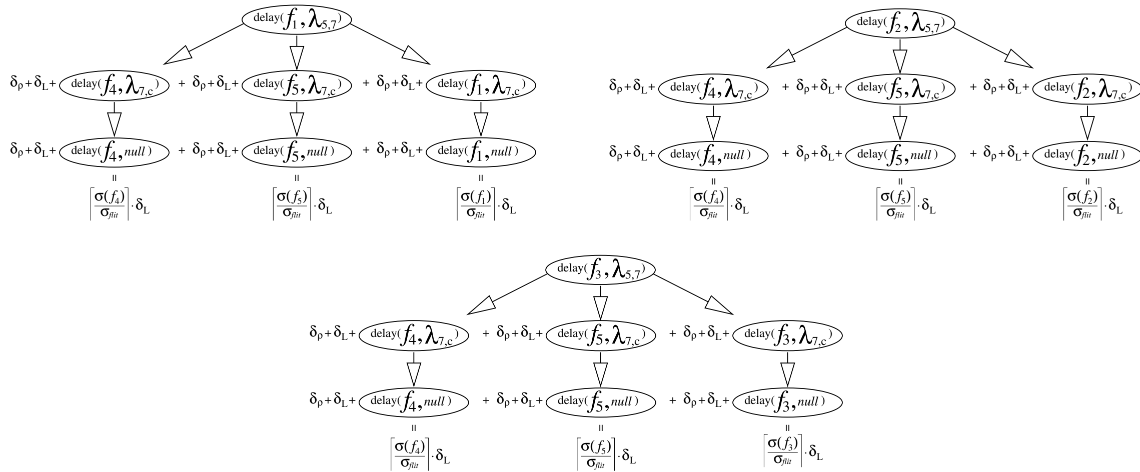


Figure 2.5: Computation subtrees for Figure 2.4

Notice, that in this case, due to the presence of other flows, the worst-case traversal time of f_2 is significantly greater than its basic network latency, i.e. $WCTT(f_2) > C(f_2)$.

2.2.2 Analysis Pessimism

The method proposed by Ferrandiz et al. [32] renders safe upper-bound estimates on $WCTT$, and the related computation terminates within a reasonable time, as demonstrated by the authors in their work. However, this method does not take into account traffic characteristics when constructing contention trees, which may lead to pessimistic results. This is illustrated with the example from Figure 2.2, and the corresponding computation tree, given in Figure 2.4. It is visible from the figure that both f_4 and f_5 occurred three times during the traversal of the analysed flow f_2. However, if the periods of f_4 and f_5 are greater than the worst-case delay of f_2, then these flows

cannot appear that often, and hence cannot contribute that significantly to $WCTT(f_2)$. Yet, the state-of-the-art method does not have a mechanism to take into account this information, but instead allows each flow to appear at the maximum possible rate, which clearly may lead to overly pessimistic worst-case traversal time estimates.

In this dissertation, two sources of pessimism of the state-of-the-art method [32] are identified, referred to as the *packet-level pessimism* and the *flow-level pessimism*.

The packet-level pessimism is related to the fact that, in the existing method, it is assumed that two packets of a flow, released during adjacent periods, can reach the same router successively. However, the time interval between two packets from adjacent periods depends on their characteristics, as demonstrated with Theorem 1.

Theorem 1. *Consider a schedulable flow-set $\mathscr{F}$. Two packets of a flow $f_i \in \mathscr{F}$, related to two adjacent periods, cannot traverse the same router in the time interval which is less than or equal to $T(f_i) - D(f_i) + C(f_i)$.*

Proof. Proven directly. Consider the case when the first packet of f_i starts traversing as late as possible, while the second packet starts traversing as early as possible, as illustrated in Figure 2.6. The traversal of the first packet can be delayed by at most $D(f_i) - C(f_i)$, as delaying it any further would cause a deadline miss, which contradicts the initial assumption of a schedulable flow-set. The second packet can be released at earliest immediately after the period, i.e. at $T(f_i)$. Let ε be an infinitesimally small but finite value, representing the delay of the second packet to leave the source and reach the NoC. Thus, the time interval between two packet occurrences within any router of the NoC can not be less than $T(f_i) - (D(f_i) - C(f_i)) + \varepsilon > T(f_i) - D(f_i) + C(f_i)$. □

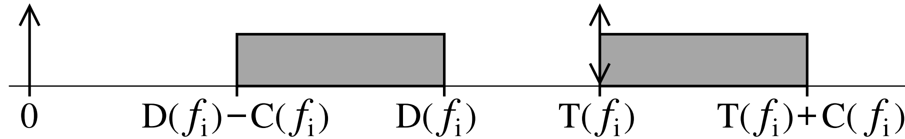


Figure 2.6: Traversal of packets from two adjacent periods

Theorem 1 can be used to reduce the packet-level pessimism in the following way. Within each router ρ_i that the flow f_j traverses, a pair $< f_j, t^* >$ is stored, where t^* denotes the time-stamp of the last traversal of f_j through ρ_i. Subsequently, during the analysis, the next traversal of f_j through ρ_i would be possible only at $t^* + T(f_j) - D(f_j) + C(f_j)$. Of course, every new feasible traversal of f_j through ρ_i would be followed by an update of the time-stamp.

Similarly, the flow-level pessimism is related to the fact that, in the existing approach, it is assumed that an arbitrary number of packets of a flow, released during adjacent periods, can reach the same router successively. However, Theorem 1 demonstrated that there indeed exists a time interval between the occurrences of two adjacent packets, and the same is true for an arbitrary number of successive packets, as proven with Theorem 2.

Theorem 2. *Consider a schedulable flow-set $\mathscr{F}$. Within the time interval t, the maximum number of packets of a flow $f_i \in \mathscr{F}$ that can reach the same router is $\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil$.*

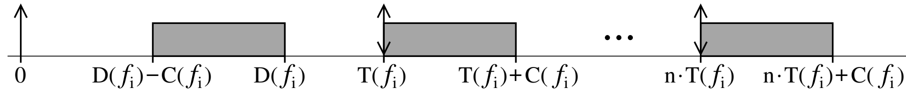


Figure 2.7: Traversal of packets from several successive periods

Proof. Two cases need to be investigated:

1. $t \leq T(f_i) - D(f_i) + C(f_i)$: Proven directly. From Theorem 1 it follows that at most 1 packet can reach the same router within t. By substituting the value of t into the computation for this theorem, it follows that $\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil < \left\lceil \frac{T(f_i)}{T(f_i)} \right\rceil = 1$, which coincides with the result of Theorem 1.

2. $t > T(f_i) - D(f_i) + C(f_i)$: Proven by contradiction. Consider that within the time interval t, at most $\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil + 1$ packets can reach the same router. By initial assumptions, $\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil + 1 \geq 3$. Consider that the first packet reached the router as late as possible, while all the other packets reached it as early as possible, as illustrated in Figure 2.7. Consider $\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil - 1$ packets, that are surrounded by the first and the last. These packets are referred to as the *inner packets*. All the inner packets contribute to t with their entire periods, therefore require time interval of at least $\left(\left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil - 1\right) \cdot T(f_i)$ where only these can reach the router. According to Theorem 1, the first packet could not reach the router later than $T(f_i) - D(f_i) + C(f_i)$ time units before the first inner packet. Finally, the last packet can reach the router only ε time units after its period. Recall, that ε is an infinitesimally small but finite value representing the delay of the packet to leave the source and reach the NoC.

$$T(f_i) - D(f_i) + C(f_i) + \left(\left\lceil \frac{t + D(f_i) - C(f_i)}{T(f_i)} \right\rceil - 1\right) \cdot T(f_i) + \varepsilon =$$

$$\left\lceil \frac{t + D(f_i) - C(f_i)}{T(f_i)} \right\rceil \cdot T(f_i) - D(f_i) + C(f_i) + \varepsilon \geq$$

$$\left(\frac{t + D(f_i) - C(f_i)}{\cancel{T(f_i)}}\right) \cdot \cancel{T(f_i)} - D(f_i) + C(f_i) + \varepsilon =$$

$$t + \varepsilon \leq t$$

The contradiction has been reached. □

Theorem 2 can be used to decrease the flow-level pessimism in the following way. Within each router ρ_i that the flow f_j traverses, not only the time-stamp of the last traversal is stored, as proposed for the reduction of the packet-level pessimism, but a time-stamp of each traversal of f_j through ρ_i is stored. Specifically, for each flow f_j that traverses through ρ_i, the traversal information is stored in the form $< f_j, t_0^*, t_1^*, ..., t_n^* >$, where t_0^* denotes the time-stamp of the traversal of the first packet of f_j through ρ_i, t_1^* of the second packet, etc. Subsequently, during the analysis,

the traversal of f_j, through ρ_i, at the time instant t, will be possible only if $n+1 \leq \left\lceil \frac{t+D(f_i)-C(f_i)}{T(f_i)} \right\rceil$, where n denotes the number of packets of f_j that already traversed ρ_i.

2.2.3 Branch and Prune (BP) Method

In this section, a novel method to compute worst-case traversal times of flows will be presented. This method is called the *Branch and Prune (BP) Method*, and it renders tigher estimates than the state-of-the-art approach [32]. This method is developed from the same fundamental idea as the existing work, which is to recursively track the progress of flows over the NoC. However, the BP method additionally takes into account the traffic characteristics, so as to reduce the packet- and flow-level pessimism.

2.2.3.1 Overview

As in the existing approach, in the BP method the tracking of the flow progress remains the same. That is, within each router that the flow traverses, a set of input links from which blocking flows may arrive is considered. Then, for each blocking link, the flow which can cause the greatest delay is identified. However, in the existing method [32], the maximum delay from each blocking link was computed before the next blocking link was considered. This causes the depth-first traversal of the computation tree. For example, when computing the worst-case traversal time for f_2 from Figure 2.2, first the blocking delay caused by f_1 is computed, then the delay by f_3, and then f_2 progresses to the next router. Note that the order in which the blocking links and hence blocking flows are considered is entirely irrelevant. In other words, the computed value is the same, irrespective of whether the blocking delay was computed first for f_1, and then for f_3, or the other way around. This is possible, because the existing method does not take into account flow characteristics.

Conversely, in the BP method, before the computation process starts, a construction of all possible orderings of blocking flows at the current router is performed. This has to be done, because, as explained later, in the BP method the flow ordering *does* matter. One ordering of flow traversals through the router is called the *interfering scenario*, and all possible traversals constitute the *list of interfering scenarios – LIS*. For the example with the flow f_2, illustrated in Figure 2.2, the *LIS* for the traversal of f_2 through ρ_2 contains the following elements: $LIS(f_2, \rho_2) = \{f_1, f_3, f_2\}$, $\{f_3, f_1, f_2\}$, $\{f_1, f_2\}$, $\{f_3, f_2\}$, $\{f_2\}$. Notice that this is equivalent to branching into several independent computation trees, where each interfering scenario represents one computation tree, like illustrated in Figures 2.8-2.9. Then, each scenario (branch) is evaluated whether the flows can arrive in the given order, without violating the constraints of Theorems 1-2. If an investigated scenario is infeasible, the entire branch related to it is pruned. The tests for compliance with Theorems 1-2 are applied every time a new branching occurs, which gives the possibility to detect and prune infeasible scenarios (branches) as early as possible. Notice that pruning a given scenario in fact prunes an entire sub-tree that would result from it, which vastly reduces the search space. This is especially beneficial for loaded networks, where, due to the complex contention patterns,

the solution space may grow exponentially with the number of flows and path lengths. Therefore, an efficient and timely pruning can significantly reduce the computational complexity.

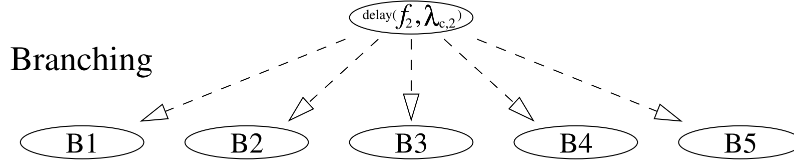
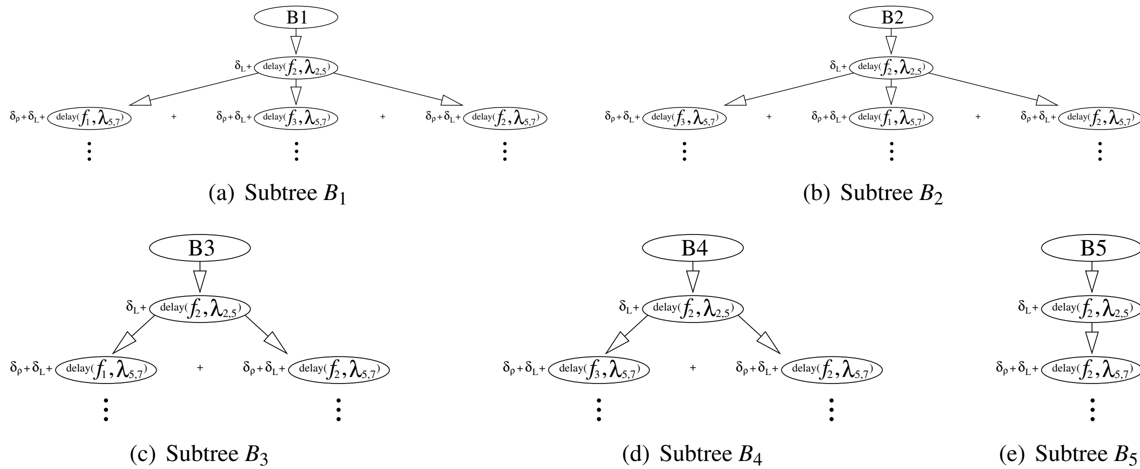


Figure 2.8: Computation tree for flow f_2 from Figure 2.2 (BP method)



(a) Subtree B_1 (b) Subtree B_2 (c) Subtree B_3 (d) Subtree B_4 (e) Subtree B_5

Figure 2.9: Computation subtrees constructed after branching in Figure 2.8

2.2.3.2 Necessity to Investigate All Scenarios

After pruning infeasible scenarios, any of the remaining ones may lead to the worst-case delay, and therefore all of them have to be investigated. Note that although it is counter-intuitive, it is not safe to assume that a scenario S_1, which elements are only a subset of some other scenario S_2, will always lead to a smaller delay than S_2. Specifically, for the given example, it is not safe to assume that $S_1 = \{f_1, f_3, f_2\}$ (Figure 2.9(a)) would always lead to a larger delay than $S_3 = \{f_1, f_2\}$ (Figure 2.9(c)) and $S_5 = \{f_2\}$ (Figure 2.9(e)), and therefore discard S_3 and S_5 for further consideration. The explanation is as follows. Although S_3 and S_5 may lead to the smaller delay at the given router (local maximum), these scenarios may in the future progression allow for other subscenarios which would be infeasible in S_1, and which could contribute to the worst-case delay of f_2 with the significant blocking delay (global maximum). Therefore, all feasible interfering scenarios have to be investigated.

Scenarios from *LIS* and investigated sequentially. When considering $S_1 = \{f_1, f_3, f_2\}$ for example, first the worst-case traversal time of f_1 is computed, recursively, then the worst-case traversal time of f_3, and finally f_2 is allowed to progress to the next router on its path. However, before computing the delay for f_1 (and the same applies to f_3 later), the pessimism of the existing method

is reduced by applying the two optimization mechanisms that determine whether f_1 is "feasible". Being infeasible implies that it is impossible for a given flow to release a packet at the given time, considering that it either released a packet too close in time relative to its previous packet (violating Theorem 1 and thereby accounting for the packet-level pessimism) or it has already exceeded the upper bound on the number of packets it could possibly generate from the beginning of the computation (violating Theorem 2 and thereby accounting for the flow-level pessimism). If the flow f_1 is deemed infeasible, then the entire interfering scenario S_1 is discarded, and the algorithm moves on to the next interfering scenario. Conversely, if f_1 is deemed feasible, then the analysis of S_1 continues, and the traversal of f_3 is investigated. Similarly, if f_3 is deemed infeasible, the entire interfering scenario S_1 is discarded. Otherwise, the flow f_2 progresses to the next router ρ_5 (see Figure 2.2), and new scenarios are generated and subsequently investigated: $S_6 = \{f_4, f_5, f_2\}$, $S_7 = \{f_5, f_4, f_2\}$, $S_8 = \{f_4, f_2\}$, $S_9 = \{f_5, f_2\}$, $S_{10} = \{f_2\}$.

2.2.3.3 Need for Ordering

Since the BP method considers the input flow characteristics, the order of flows within each scenario cannot be ignored, as it can lead to different results. This is illustrated with an example given in Figure 2.2. Consider two possible flow traversal sequences:

$$Seq_1 = \{f_4, f_5, f_3, f_5, f_4, f_2, f_4, f_5, f_1\} \quad \wedge \quad Seq_2 = \{f_4, f_5, f_3, f_4, f_5, f_2, f_4, f_5, f_1\}.$$

These two sequences differ only in the order in which f_4 and f_5 block f_2. However, notice that in Seq_1 the first and the second packet of f_5 are distanced only by f_3 , while the second and the third packet of f_4 are distanced only by f_2. Conversely, in Seq_2, any two packets of the same flow are distanced by the packets of at least two other flows. Depending on flow characteristics, in some cases the entire Seq_2 might be feasible, while Seq_1 would require the pruning of some packets of f_4 and/or f_5. Thus, considering only Seq_1 in the analysis may result in unsafe worst-case estimates, and in order to capture the exact worst-case, it is necessary to investigate all possible flow orderings (scenarios) at every traversed router.

Indeed, at a given router ρ_i along the path of the flow under analysis f_j, the list of interfering scenarios can be constructed as explained above. However, identifying which blocking flows are infeasible within each of the scenarios requires the information about the flow traversal history. That is, it is necessary to keep the track of which flows have already progressed, in which order, and through which routers, before f_j reaches ρ_i. Without this knowledge, the pruning mechanisms would not be able to determine whether a flow listed in an interfering scenario is feasible or not. This leads to the concept of the *context*, which is the key component of the BP method.

2.2.3.4 Context

The context is a data structure which stores all the information that characterises the unique sequence of flow progressions throughout the network, before the analysed flow f_i reaches a certain router ρ_j at the time instant t. The context contains the following information: (i) the order in

which the flows have progressed over the network in the interval $[0-t]$, (ii) the time-stamps of the past traversals of all directly or indirectly contending flows of f_i through the routers on their respective paths, and (iii) the delay incurred by f_i before reaching ρ_j. The context has all the information that is necessary to check whether a traversal of a certain flow would violate Theorems 1-2. In other words, the context is the infrastructure that is needed to perform an efficient pruning of infeasible scenarios.

Notice that the context contains the information about the unique system state, i.e. it corresponds to a unique sequence of flow traversals. However, when the branching occurs, the context must also be maintained for each of the newly generated sequences of flow traversals. Therefore, each flow sequence that may potentially lead to the worst-case scenario must have its private context. At the very end, when the analysed flow f_i reaches the destination, $WCTT(f_i)$ will be found by comparing all the flow sequences (i.e., the contexts) in which f_i reaches its destination.

2.2.3.5 Algorithm

Algorithm 2 $delay(f_i, \lambda_{m,n})$

Input: flow f_i, link $\lambda_{m,n}$
Output: WCTT of f_i, starting from $\lambda_{m,n}$, to the destination

1: $ctx.sequence \leftarrow \emptyset; ctx.delay \leftarrow 0;$ // the computation starts with the initial (empty) context
2: $ctxSet \leftarrow getContexts(f_i, \lambda_{m,n}, ctx);$ // find the set of all possible contexts
3: $WCTT \leftarrow \max_{\forall ctx \in ctxSet} ctx.delay;$ // find the maximum delay
4: **return** $WCTT;$

The computation process starts by invoking the Algorithm 2 with the following input parameters: the flow under analysis f_i, the first link on its path $\lambda_{m,n}$. First, an initial context is created (line 1). The initial context contains an empty flow sequence and a delay of zero. Then, in order to obtain all possible contexts for the traversal of f_i, Algorithm 3 is invoked (line 2). The context with the largest delay is found by enumerating all contexts (line 3). Finally, the largest delay is returned as the worst-case traversal time of f_i.

All contexts are generated with Algorithm 3, which takes as input parameters (i) the flow under analysis f_i, (ii) the first link on its path $\lambda_{m,n}$, and (iii) the newly created initial context *currCtx*. First, it is checked whether $\lambda_{m,n}$ is null. If it is, this means that the header of f_i reached the destination, so the only remaining activities are to transfer the rest of the flits and to append f_i to the sequence of traversed flows (line 3). Otherwise, it is checked whether $\lambda_{m,n}$ is the first link on the flow path. If it is, this means that f_i can be blocked on $\lambda_{m,n}$ only by the flows with the same source as f_i. All such flows are grouped into the set $\mathscr{F}_{src}$ (lines $8-10$). Then, the list of interfering scenarios $LIS(f_i, \rho_n)$ for the flow f_i at the router ρ_n is created (line 11). $LIS(f_i, \rho_n)$ contains all possible combinations of flows from $\mathscr{F}_{src}$ (including an empty set) with the flow f_i appended to the end of each flow sequence. Consider the flow f_2 in the example illustrated in Figure 2.2. $LIS(f_2, \rho_2) = f_2$, because f_2 is the only flow originating from the router ρ_2.

Algorithm 3 $getContexts(f_i, \lambda_{m,n}, currCtx)$

Input: flow f_i, link $\lambda_{m,n}$, current context $currCtx$
Output: A set of contexts $ctxSet$

1: **if** $(\lambda_{m,n} = null)$ **then**
2: // the header flit reached the destination, so the remaining flits have to be transferred
3: $currCtx.delay += \left\lceil \frac{\sigma(f_i)}{\sigma_{flit}} \right\rceil \cdot \delta_L$; $currCtx.sequence.Append(f_i)$;
4: **return** $currCtx$;
5: **end if**
6: **if** $(get(\mathscr{L}(f_i), 1) = \lambda_{m,n})$ **then**
7: // $\lambda_{m,n}$ is the first link of f_i, so only the flows with the same source as f_i can block
8: **for each** $(f_j \in \mathscr{F} \mid f_j \neq f_i \wedge get(\mathscr{L}(f_j), 1) = \lambda_{m,n})$ **do**
9: $\mathscr{F}_{src} \leftarrow f_j$; // find all flows with the same source as f_i
10: **end for**
11: $LIS(f_i, \rho_n) \leftarrow$ Set of local interfering scenarios based on $\mathscr{F}_{src}$;
12: **else**
13: $\lambda_{k,m} \leftarrow getprev(\mathscr{L}(f_i), \lambda_{m,n})$; // find $\lambda_{k,m}$, which f_i traversed immediately before $\lambda_{m,n}$
14: // f_i can be blocked by flows which traverse $\lambda_{m,n}$, but have previous link different than $\lambda_{k,m}$
15: **for each** $(g \in \mathbb{N} \mid \lambda_{g,m} \in \lambda \wedge g \neq k)$ **do**
16: $BL \leftarrow \lambda_{g,m}$;
17: **end for**
18: **for each** $(g \in \mathbb{N} \mid \lambda_{g,m} \in BL)$ **do**
19: // find the set of flows $\mathscr{F}_{g,m}$ that traverse $\lambda_{g,m}$ and $\lambda_{m,n}$, and hence can block f_i
20: **for each** $(f_j \in \mathscr{F} \mid \lambda_{g,m} \in \mathscr{L}(f_j) \wedge \lambda_{m,n} \in \mathscr{L}(f_j))$ **do**
21: $\mathscr{F}_{g,m} \leftarrow f_j$;
22: **end for**
23: **end for**
24: $LIS(f_i, \rho_n) \leftarrow$ Set of local interfering scenarios based on BL;
25: **end if**
26: $GCList \leftarrow \emptyset$;
27: **for each** $(S_i \in LIS(f_i, \rho_n))$ **do**
28: $SCList \leftarrow \{currCtx\}$;
29: **for each** $(f_j \in S_i)$ **do**
30: **while** $(SCList \neq \emptyset)$ **do**
31: $ctx_k \leftarrow SCList.pop()$;
32: **if** $(CompliantThm1(f_j, ctx_k) \wedge CompliantThm2(f_j, ctx_k))$ **then**
33: **if** $(get(\mathscr{L}(f_j), 1) = \lambda_{m,n})$ **then**
34: $ctx.delay += \delta_L$;
35: **else**
36: $ctx.delay += \delta_\rho + \delta_L$;
37: **end if**
38: $FCList_k \leftarrow getContexts(f_j, getnext(\mathscr{L}(f_j), \lambda_{m,n}), ctx_k)$;
39: **end if**
40: **end while**
41: $SCList \leftarrow \bigcup_{\forall k} FCList_k$;
42: **end for**
43: $GCList \leftarrow GCList \cup SCList$;
44: **end for**
45: **return** $GCList$;

Conversely, if $\lambda_{m,n}$ is not the first link of f_i, then f_i can suffer blocking only from flows which also traverse $\lambda_{m,n}$, but enter the router ρ_m from a different link than the one from which f_i does (which is $\lambda_{k,m}$). Thus, all such links are added to the set of blocking links *BL* (lines $15-17$). For each blocking link $\lambda_{g,m} \in BL$, a set of flows $\mathscr{F}_{g,m}$ is created. $\mathscr{F}_{g,m}$ contains all flows that traverse $\lambda_{g,m}$ and subsequently compete with f_i on $\lambda_{m,n}$ (lines $18-23$). Then, the list of interfering scenarios $LIS(f_i, \rho_n)$ is created (line 24). $LIS(f_i, \rho_n)$ contains all combinations of flow sequences in which there is at most one flow from each link in *BL*, and f_i is appended to the end of each flow sequence. Consider the flow f_2 in the example illustrated in Figure 2.2. As already mentioned and illustrated in Figures 2.8-2.9, $LIS(f_2, \rho_2) = \{f_1, f_3, f_2\}$, $\{f_3, f_1, f_2\}$, $\{f_1, f_2\}$, $\{f_3, f_2\}$, $\{f_2\}$. Similarly, when f_2 progresses to the next router ρ_5, it is $LIS(f_2, \rho_5) = \{f_4, f_5, f_2\}$, $\{f_5, f_4, f_2\}$, $\{f_4, f_2\}$, $\{f_5, f_2\}$, $\{f_2\}$.

Now, the algorithm investigates all scenarios from $LIS(f_i, \rho_n)$ in a sequential manner (line 27). For each scenario S_i, a list *SCList* is created (line 28). Ultimately, *SCList* will contain all possible contexts that are produced as a result of applying this scenario at the given router to all existing contexts.

When a certain scenario $S_i \in LIS(f_i, \rho_n)$ has been selected (line 27), the flows from its flow sequence are considered consecutively (line 29). The traversal of the first flow from the list, denoted f_j, is attempted assuming all possible contexts. Specifically, all existing contexts are considered sequentially (line 30), and it is checked whether the traversal of f_j violates Theorems 1-2 for each given context (line 32). If it does, the combination of the current context ctx_k and the current scenario S_i is discarded, which is equivalent to pruning one branch of the computation tree. Otherwise, f_j is allowed to progress (lines $33-37$). Then, all traversal sequences of f_j in the subsequent routers are generated via a recursive call, and stored within the list $FCList_k$ (line 38). After testing the traversal of f_j upon all contexts, the *SCList* is updated with all newly produced contexts (line 41), and the next flow from the current scenario S_i is considered. This process is repeated for all the flows in the current scenario S_i.

When the scenario S_i is applied to all current contexts, the resulting scenarios are added into the final list of all contexts *GCList* (line 43), and the next scenario from $LIS(f_i, \rho_n)$ is considered. After applying all scenarios to all contexts, *GCList* will contain the complete list of all possible contexts for the analysed flow f_i. Finally, *GCList* is returned (line 45), and the Algorithm 3 terminates.

2.2.4 Branch, Prune and Collapse (BPC) - More Efficient Method

The existing method proposed by Ferrandiz et al. [32] scales well, because the contexts are not constructed and maintained, but only the maximum delay incurred at each router is retained. However, due to that fact, this method may derive pessimistic worst-case estimates. Conversely, the proposed BP method takes into account traffic characteristics and constructs contexts, which helps to identify an exact flow sequence that leads towards the worst-case traversal time of an analysed flow. This allows to compute the exact worst-case traversal times for all traffic flows. However, the benefits of the BP method come at the price of a significant computational complexity (processing

power), and a significant spatial complexity (memory consumption), which suggests that BP does not scale, and that it may not be the most efficient approach for large flow-sets.

These two extreme approaches may be merged into a hybrid method in the following way. Periodically, a certain context information can be dropped, in order to reduce the computational and spatial complexity of the method. In such cases, the maximum delay observed until the dropping moment should be retained, and the computation process may continue. This method is called the *Branch, Prune and Collapse* (BPC).

2.2.4.1 Overview

As already observed, the identification of infeasible sequences in the BP method was possible due to the explicit book-keeping of all possible contexts. However, the computational and spatial complexity of the BP method may render it inapplicable to large flow-sets. Motivated by this fact, the BPC method is introduced. BPC presents a more generalised method, of which the BP method and the existing method of Ferrandiz et al. [32] are special cases. Specifically, BPC has a configurable parameter called the *Sequence Information Retention Limit* (SIRL), which creates a trade-off between the analysis tightness (the amount of pessimism) on one side, and the computational and spatial complexity on another. The parameter SIRL acts as a threshold on the number of flow sequences whose contexts are retained.

In the BP method, all investigated flow sequences and their contexts are combined with all possible scenarios at the current router, so as to generate new flow sequences for the next router on the path of the analysed flow. Conversely, in the BPC method, when the number of investigated sequences reaches a pre-set limit of SIRL, a new flow sequence containing a single dummy flow is created. The context of this sequence is populated as follows: (i) the delay field is set to the maximum of the delays of all sequences investigated thus far, and (ii) the history information, i.e the time-stamps of past packet occurrences of each flow at each router are set to *null* (or zero, as appropriate). This process is called the *collapse phase*, because during it a set of investigated sequences is "collapsed" (merged) into a single dummy sequence containing a single dummy flow with the conservative delay estimate, and no history information regarding the traversal of other flows. Now, instead of continuing the computation process by managing all the sequences and their contexts (which number is at least SIRL), as it would be done in the BP method, only this one dummy sequence is created, and the method continues to behave in the same way as the BP approach. This holds until the number of investigated sequences again exceeds SIRL, and thereby invokes another collapse phase.

Note, that in cases where SIRL $\rightarrow \infty$, the collapse phase never occurs, and the behaviour of the BPC method is identical to that of the BP method. Conversely, in cases where SIRL $= 1$, the collapse occurs for every combination of sequences and scenarios at every router, which is identical to the behaviour of the existing method [32]. Also, notice that the parameter SIRL controls the frequency of collapse phases, and the smaller the SIRL is, the more frequently collapses occur.

2.2.4.2 Necessity to Create Dummy Sequence

At any intermediate stage of the analysis, when the number of sequences reaches the value of the parameter SIRL, a single sequence that will provably lead to the WCTT cannot be detected. This has already been explained in Sections 2.2.3.2-2.2.3.3, and this is the main reason why in the BP method all contexts need to be combined with all scenarios, and subsequently why all resulting contexts need to be retained for future computations. Therefore, during the collapse phase, from the set of the constituent sequences, a specific single sequence cannot be solely carried forward in the computation. This is due to the fact that a single sequence, that was recognised as the local maximum at the intermediate stage, may in the later stages of the computation be subject to pruning, because of its flow history, and thereby may not contribute to the global maximum delay. In order to prevent this, the history information regarding the previous flow traversals is entirely dropped (thereby reducing the chances of the collapsed sequence to be subject to pruning during the later stages of the computation process). In fact, only the local maximum delay is retained within a new dummy sequence, consisting of a single dummy flow. To summarize, during the collapse phase, the BPC method creates a dummy sequence which inherits the delay of the local maximum, but entirely drops the traversal history of the flows constituting that sequence.

2.2.4.3 Illustrative Example

The behaviour of the BPC method is illustrated with an example of flows given in Figure 2.10, which is identical to the example depicted in Figure 2.2, but has been repeated here for the reader's convenience.

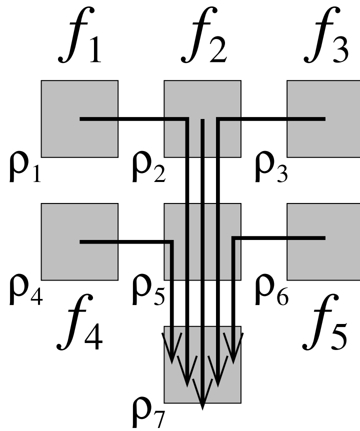


Figure 2.10: Traffic flows (revisited example 2)

Consider the flow f_2 between the cores π_2 and π_7. f_2 traverses the routers ρ_2, ρ_5, and ρ_7. As mentioned in Section 2.2.1, the flows f_4 and f_5 can potentially block f_2 three times: twice indirectly by blocking f_1 and f_3 at ρ_5 (f_1 and f_3 block f_2 directly at ρ_2), and finally once directly at ρ_5 when f_2 reaches it. Thus, f_4 and f_5 are the promising candidates for pruning (line 32 of Algorithm 3).

Assume that SIRL $= 5$. On the link $\lambda_{c,2}$ the analysed flow f_2 can suffer blocking only from flows originating from the same core π_2. However, there are no such flows, so f_2 uninterruptedly progresses to the router ρ_2. For the next link on the path of f_2, i.e. $\lambda_{2,5}$, the BPC method constructs the *LIS* for f_2 as follows: $LIS(f_2, \rho_2) = \{f_1, f_3, f_2\}$, $\{f_3, f_1, f_2\}$, $\{f_1, f_2\}$, $\{f_3, f_2\}$, $\{f_2\}$. First, the scenario $\{f_1, f_3, f_2\}$ is explored. At this time, the list *SCList* is reseted to the current (initial) context (at line 28 of Algorithm 3). Note, that *SCList* will ultimately contain all generated contexts arising from the analysis of this scenario $\{f_1, f_3, f_2\}$, before being appended to the global list of contexts *GCList*

(line 43 of Algorithm 3). Now, a recursive call is invoked for the router ρ_5 with f_1 being the analysed flow. This will result in a new *LIS* constructed at ρ_5 as: $LIS(f_1,\rho_5) = \{f_4, f_5, f_1\}$, $\{f_5, f_4, f_1\}$, $\{f_4, f_1\}$, $\{f_5, f_1\}$, $\{f_1\}$. Similarly, *LIS* is generated for the flow f_3 at the router ρ_5 as: $LIS(f_3,\rho_5) = \{f_4, f_5, f_3\}$, $\{f_5, f_4, f_3\}$, $\{f_4, f_3\}$, $\{f_5, f_3\}$, $\{f_3\}$.

These sequences are propagated back, and should be combined with the current (initial) context before f_2 progresses to ρ_5. As both $LIS(f_1,\rho_5)$ and $LIS(f_3,\rho_5)$ contain 5 elements, the combined set of sequences may contain 25 elements (i.e. 5 sequences from f_1 combined with 5 from f_3). It is obvious that for large flow-sets this back-propagation may produce a large number of sequences and even cause intractability, which is the main drawback of the BP method. Given that SIRL $= 5$ in this example, the collapsing requirement is met. Those 25 sequences are collapsed into a single one, which contains a dummy flow f_X, while its delay is set to maximum delay amongst all collapsed sequences. When f_2 finally progresses to ρ_5 and encounters the potentially blocking flows f_4 and f_5, the method checks the context related to the current sequence $\{f_X\}$. Since there is no prior information regarding f_4, nor f_5, the algorithm considers that these two flows are arriving for the first time and thus allows them to pass and additionally block f_2. The resulting sequences would be $\{f_X, f_4, f_5, f_2\}$, $\{f_X, f_5, f_4, f_2\}$, $\{f_X, f_4, f_2\}$, $\{f_X, f_5, f_2\}$ and $\{f_X, f_2\}$. Conversely, BP would have retained the contexts for all sequences, which would give the possibility to potentially prune another traversals of f_4 and f_5, but at the expense of investigating $25 \cdot 5 = 125$ sequences. Note that this includes only the investigation of the first scenario $\{f_1, f_3, f_2\}$. Similarly, the number of sequences that can arise from the scenario $\{f_1, f_3, f_2\}$ is also 5^3. The number of sequences from the scenarios $\{f_1, f_2\}$ and $\{f_3, f_2\}$ is 5^2, while the number of sequences from the scenario $\{f_1\}$ is 5^1. Thus, the total number of possible flow sequences for this example is $5^3 + 5^3 + 5^2 + 5^2 + 5^1 = 305$.

2.2.4.4 Proof of Safety of BPC

From the previous example it is obvious that the BPC method, by collapsing upon reaching the SIRL threshold and hence discarding some history information, may output more pessimistic WCTT estimates than the BP method. In this section, it is proven that the values obtained by the BPC method will under no circumstance lead to an unsafe WCTT estimate.

Let *wcs* (short for the **w**orst-**c**ase **s**equence) be the flow sequence leading to the worst-case delay of the analysed flow. By definition, *wcs* is a *feasible* flow sequence. In order to prove that BPC is safe, it will be proven that it will never eliminate *wcs* from the set of investigated sequences, i.e. BPC will never return a value which is smaller than the delay of *wcs*.

First, it should be noted that if SIRL $\rightarrow \infty$, then the BPC method behaves like the BP method: it performs an exhaustive enumeration of all possible sequences at each router, and thus considers all possible blocking patterns of other flows during the traversal of the analysed flow (a brute-force approach, which is inherently safe). The pruning mechanisms (line 32 of Algorithm 3) exploit the information regarding the previous flow occurrences, in order to identify infeasible flow sequences, and in that way reduce the list of sequences that need to be investigated. By definition, *wcs* is a feasible flow sequence and therefore, it will not be eliminated by these pruning techniques.

Given the loss of the history information during the collapse phase, the BPC method is unable to identify as many infeasible flow sequences as the BP method. These infeasible sequences contain flows that cannot traverse that frequently due to Theorems 1-2, and hence additionally contribute to the worst-case traversal time of the analysed flow. Eventually, the set of explored sequences will also include the set of feasible sequences, which includes *wcs*, but will also include some infeasible sequences, which were not identified due to loss of history information during the collapse phase. Finally, after computing the maximum traversal time of all the sequences, the method will return a value which is greater than, or equal to the WCTT corresponding to *wcs*, and therefore the method is safe.

To summarize, the BP method investigates all feasible flow sequences (including *wcs*) and all infeasible ones. The infeasible sequences are pruned due to the retained contexts, and thereby the method is safe. BPC investigates all feasible and infeasible sequences, prunes only a fraction of infeasible ones, and as a consequence is still safe but may be pessimistic.

2.2.4.5 Proof of Termination of Algorithm 3

Let $\mathscr{S}$ be the set of flow-link pairs $\langle f_i, \lambda_{k,l} \rangle$, where $f_i \in \mathscr{F}$ and $\lambda_{k,l} \in \lambda$, such that $\langle f_i, \lambda_{k,l} \rangle \in \mathscr{S} \iff \lambda_{k,l} \in \mathscr{L}(f_i)$. Since λ and $\mathscr{F}$ are the sets of the finite number of elements, it holds that $\mathscr{S}$ has a finite number of elements as well. A progress of a flow f_i from a link $\lambda_{k,l}$ to a subsequent link $\lambda_{l,m}$ on its path is equivalent to the progress from the pair $\langle f_i, \lambda_{k,l} \rangle$ to the pair $\langle f_i, \lambda_{l,m} \rangle$. If some flow f_j blocks the flow f_i on the link $\lambda_{k,l}$, it corresponds to the progress from the pair $\langle f_i, \lambda_{k,l} \rangle$ to the pair $\langle f_j, \lambda_{k,l} \rangle$. For a given flow f_i, and a current link $\lambda_{k,l}$, the algorithm progresses in a forward manner to the next link $getnext(\mathscr{L}(f_i), \lambda_{k,l})$ on the path of the flow f_i by invoking the function $getContexts()$ (line 38 of Algorithm 3). Starting from any pair $\langle f_i, \lambda_{k.l} \rangle \in \mathscr{S}$, Algorithm 3 investigates all flow-link pairs from $\mathscr{S}$ that are reachable, assuming the round-robin arbitration policy. Then, the algorithm is recursively invoked for every transitioned pair. From the deadlock-free property of the X-Y routing mechanism [40] it follows that the initial pair $\langle f_i, \lambda_{k,l} \rangle$ will never be revisited. As soon as all the explored contexts are popped (line 31 of Algorithm 3), the algorithm will terminate.

2.2.5 Experimental Evaluation

In this section, the experimental evaluation is performed with the dual objective: to compare the BP and BPC methods with the existing approach [32], and to study the impact of different parameters on the tightness of derived WCTT estimates. The analysis parameters are summarized in Table 2.1, where an asterisk sign denotes a randomly generated value, assuming a uniform distribution.

2.2.5.1 Experiment 1: BPC vs Existing Method of Ferrandiz et al. [32]

The improvements of the BPC method over the existing one cannot be quantified in a general sense, because the results largely depend on the characteristics of the flow-set upon which the

Table 2.1: Analysis parameters for Section 2.2.5

NoC topology and size	**2-D mesh with 8 $\times$ 8 routers**
Router frequency ν_ρ	**250 MHz**
Routing delay δ_ρ	**3 cycles (12 ns)**
Link traversal delay δ_L	**1 cycle (4 ns)**
Flow packet size $\sigma(f_i), \forall f_i \in \mathscr{F}$	**512 bytes**
Link width = flit size σ_{flit}	**16 bytes**
Testing platform	**Intel dual-core desktop & Java (Max heap-size: 4 GB)**

techniques are applied. Therefore, in order to observe the trends and the ranges of improvements that are achieved by employing the proposed approach, the comparison is performed on a wide range of different flow-sets.

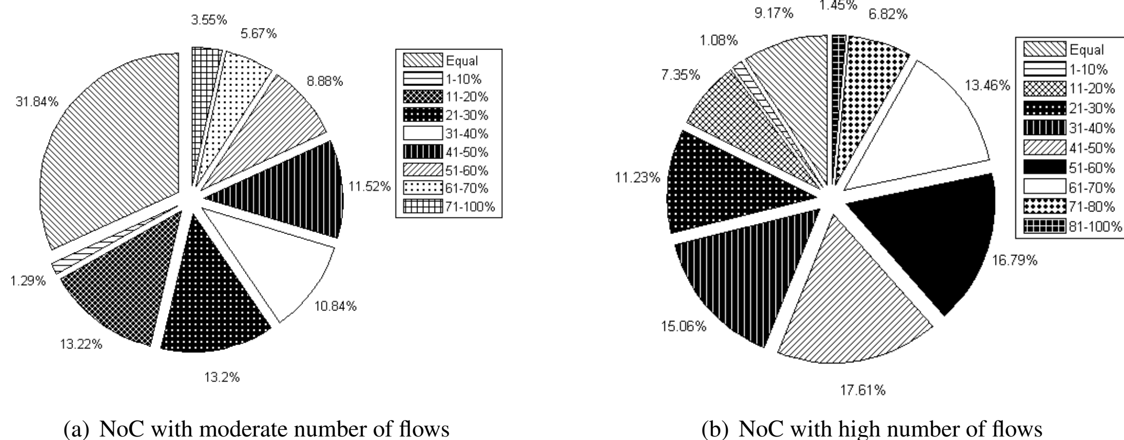
Experiment 1a (NoC with moderate number of flows): 200 random flow-sets are generated, each consisting of 64 flows. The flows originate from each core, but terminate at a randomly generated destination by following an X-Y routed path. A deadline and a period for each flow are randomly generated values in the range $[20-100]$ μs, assuming a uniform distribution, where $D(f_i) \leq T(f_i), \forall f_i \in \mathscr{F}$. The upper-bounds on the worst-case traversal times (WCTT) of each flow are computed, using both the approaches. Subsequently, the results are compared. For the BPC method, the value of the parameter SIRL $= 10000$.

In order to quantify the range of improvements, a metric called the *Percentage Improvement Ratio* (PIR) is used. PIR $= \frac{100 \cdot (WCTT_U - WCTT_O^{10000})}{WCTT_U}$, where $WCTT_U$ denotes the upper-bound on WCTT computed by the existing approach, which is also referred to as the *unoptimized WCTT*, and $WCTT_O^{10000}$ is the value computed by the BPC method for SIRL $= 10000$, which is also reffered to as the *optimized WCTT*. Therefore, PIR $= 25\%$ implies that BPC derived a 25% smaller (tighter) WCTT upper-bound.

Figure 2.11(a) illustrates the results. For 31.84% of the flows, the bounds computed by both methods are equal, that is $WCTT_O^{10000} = WCTT_U$, and hence PIR $= 0\%$. For the rest of the 68.16% the BPC method renders tighter WCTT estimates. Specifically, 1.29% of the flows have PIR in the range $[1-10\%]$, 13.22% in the range $[11-20\%]$, and so on. At the higher end of the PIR scale, for 3.55% of the analysed flows the BPC method computed $[71-100\%)$ tighter WCTT bounds.

The WCTT parameter*: If the computation finishes with the number of investigated sequences not exceeding SIRL, this means that no collapses occurred, and that the BPC method computed a value of the traversal time as would be computed by the BP method. This further implies that all possible flow sequences were investigated, and the one causing the greatest worst-case delay to the analysed flow was identified. The delay of such a sequence is denoted WCTT*.

Conversely, in cases where collapses do occur, the returned value presents only an upper-bound on the worst-case traversal time of the analysed flow, without any additional information on how tight that bound actually is. When viewed from that perspective, the existing approach [32] presents a special case of the BPC method where SIRL $= 1$. Therefore, the existing approach obtains WCTT* only when the number of possible flow sequences is equal to 1.



(a) NoC with moderate number of flows

(b) NoC with high number of flows

Figure 2.11: Distribution of WCTT improvement across flows (legends represent improvement ranges)

Of those 31.84% of the flows for which both methods returned equal WCTT values (i.e. $WCTT_O^{10000} = WCTT_U$), for $\frac{3}{4}$ of them, (23.96% of all the flows), the existing method was able to capture WCTT*, inferring that these cases were simple and involved the investigation of a single flow sequence. Therefore, in these cases there was no scope for improvements.

Based on the computed bounds, it can be concluded that the BPC method performs equally well or dominates the existing method. Also, for the selected SIRL value, BPC managed to capture WCTT* in 92.13% of the cases, implying that any additional increase in SIRL would not provide significantly tighter WCTT bounds, but would require factorially greater amount of computation time.

The analysis completes within 24 hours, averaging a little bit more than 7 minutes per flow-set (each with 64 flows). The most complex flow-set took around an hour to compute all WCTT values. This finding suggests that the execution times may vary drastically, when analysing flow-sets with identical characteristics but different flow routes. Indeed, for flow-sets with complex contention patterns, the analysis time can be an order of magnitude greater than the average analysis time for the flow-set with similar traffic characteristics.

Experiment 1b (NoC with high number of flows): The main purposes of this experiment are to test the scalability potential of the BPC method, and to observe its efficiency when applied to larger flow-sets. Again, 200 random flow-sets are generated, each consisting of 128 flows, with two flows originating from each core, but terminating at a random destination by following an X-Y routed path. Flow deadlines and periods are randomly generated values in the range $[0.1-1]$ ms, assuming a uniform distribution, where $D(f_i) \leq T(f_i), \forall f_i \in \mathscr{F}$. For all flows, the values of $WCTT_U$ and $WCTT_O^{10000}$ were computed and compared. The analysis completed in 5 days, averaging 36 minutes per flow-set. The most complex ones consumed around 3 hours, demonstrating that the BPC method is scalable and applicable to practical scenarios involving hundreds of concurrent flows.

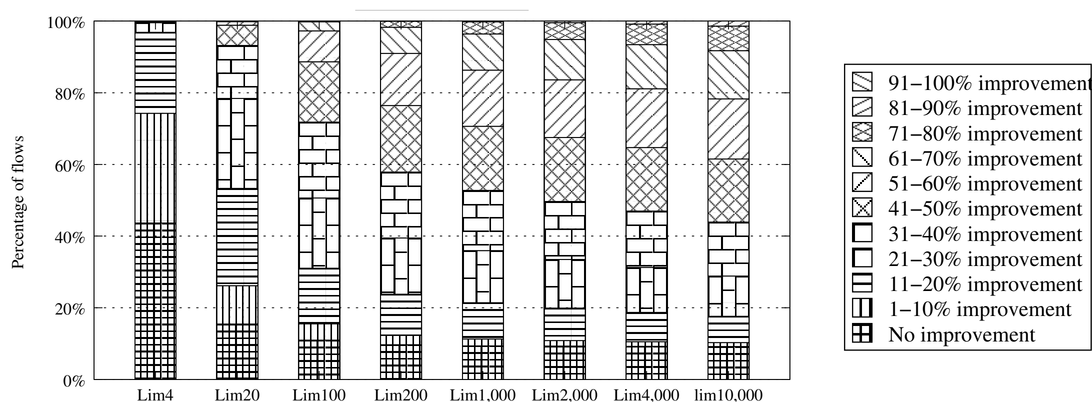


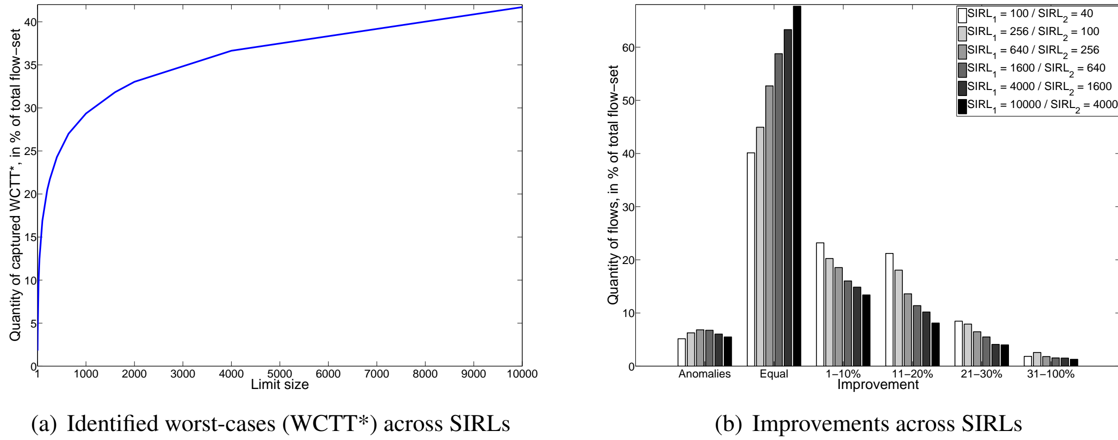
Figure 2.12: BPC method with varying SIRL vs. Ferrandiz et al. [32] method

As in the previous experiment, the PIR metric is used to quantitatively express the improvements of the BPC method over the existing one [32]. Figure 2.11(b) illustrates the results. For 9.23% of the flows, no improvements were made (PIR = 0%). For most of the flows without any improvements (8.11%), the existing method indeed managed to capture WCTT*, which implies that for these simple sequences consisting of a single flow no improvements were possible. For the rest, i.e. 90.77% of the analysed flows, it holds that $WCTT_O^{10000} < WCTT_U$, that is, the BPC method rendered tighter estimates. It is interesting to see that, for more than 13% of the flows, the improvements were in the range $[61-70\%]$, while for more than 8% of them the improvements were greater than 70%.

Due to more complex traffic patterns, resulting from the increased amount of traffic, BPC with SIRL = 10000 identified WCTT* for 41.71% of the flows, which is significantly less, when compared with the same for moderately loaded NoCs. This suggests that in these cases the improvements can be achieved by increasing SIRL beyond 10000, but at the expense of additional computational and spatial complexity. Although the computation time of the BPC method is longer than that of the existing method, BPC clearly dominates the existing approach in terms of tightness of obtained bounds. The selection of the parameter SIRL creates a trade-off between the computational and spatial complexity on one side, and the analysis tightness on another, as will be investigated in the next experiment.

2.2.5.2 Experiment 2: Impact of SIRL on Analysis

The objective of this experiment is to observe the impact of SIRL on the tightness of computed WCTT estimates. Intuitively, retaining more information about flow sequences provides more opportunities for pruning (eliminating infeasible flow sequences), and therefore leads to tighter estimates. To validate this assumption, considering the flow-sets from Experiment 1b, the BPC method is invoked and the WCTT estimates are obtained for different values of the parameter SIRL. Subsequently, the obtained bounds are compared against the bounds obtained by the existing method [32]. The results are illustrated in Figure 2.12. As in Experiment 1, the PIR metric is used.



(a) Identified worst-cases (WCTT*) across SIRLs

(b) Improvements across SIRLs

Figure 2.13: Inter-SIRL ratios

It is visible that as SIRL increases, the percentage of flows with no improvements decreases. Thus, with SIRL $= 4$, the WCTT estimates computed for 43.6% of the flows exhibit no improvements, while with SIRL $= 2000$ only 9.29% of the flows show no improvements (and the rest 90.71% of the flows have tighter WCTT bounds). Notice, that the percentage of flows within high PIR categories increases as SIRL increases. This is in accordance with the method rationale that the retention of contexts, and within them the information regarding past flow traversals, can provide opportunities for pruning and tightening the WCTT estimates. But, as seen in the shift from SIRL $= 4000$ to SIRL $= 10000$, the distribution of improvements across PIR categories does not differ much, because almost all opportunities for pruning infeasible sequences are exhausted. This further implies that choosing limits beyond a given SIRL will only burden the system to retain the contexts of flow sequences which have very small chances to lead towards a tighter WCTT estimate. So, a judicious decision must be taken by the system designer, considering the desired tightness of results, and the time in which the analysis must complete. In that respect, the BPC method provides the full flexibility via the parameter SIRL.

2.2.5.3 Experiment 3: Inter-SIRL Ratios

In the previous experiment, the BPC method with different values of the SIRL parameter was compared against the existing approach [32]. In order to get a deeper insight into the impact of SIRL on the tightness of derived bounds, in this experiment the bounds are obtained assuming BPC with different SIRL values, where the flow-sets from Experiment 1b are used. The obtained bounds are compared against each other, and subsequently the results are plotted in Figure 2.13. The results coincide with the intuition, suggesting that greater values of SIRL improve the chances of capturing WCTT*. This claim is confirmed with a logarithmic growth in the number of non-collapsed sequences, as SIRL increases (Figure 2.13(a)).

Figure 2.13(b) demonstrates that the relative improvements across SIRLs diminish as SIRL increases. That is, in 60% of the cases the BPC method with SIRL $= 100$ shows improvements

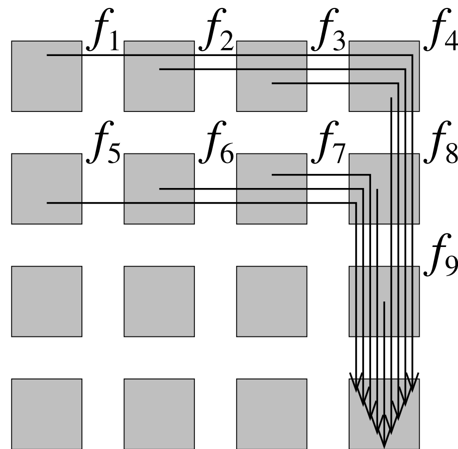


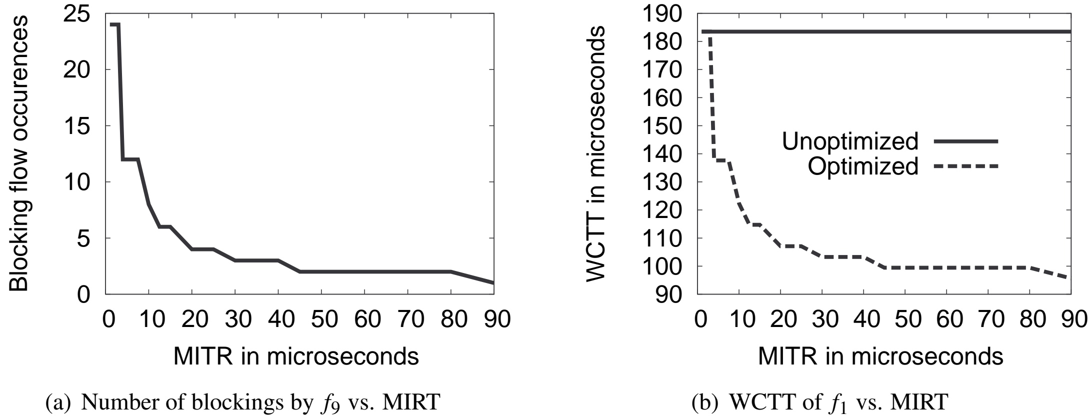
Figure 2.14: All-to-one contending flows

over BPC with SIRL = 40, while the improvements are reported only in 30% of the cases when comparing results of SIRL = 10000 and SIRL = 4000. Thus, the number of cases with no improvements increases with SIRL. Conversely, as SIRL increases, the cases with improvements decrease across all improvement ranges, suggesting that it may not be efficient to perform the analysis with very high values of SIRL. The benefits of the analysis with higher SIRL diminish as SIRL increases (especially for sequences comprising of flows that can occur at most once within the observed interval).

As already stated, the value of the SIRL influences the frequency of collapses. However, one interesting and counter-intuitive observation is the fact that higher SIRL does not necessarily always lead to a tighter WCTT upper-bound. This is explained with the following example. Consider the flow f_2 from the example depicted by Figure 2.10. Assume that f_4 and f_5 are potential candidates for pruning. Now, assume that greater SIRL performs a collapse between occurrences of f_1 and f_2. As the history information is lost, the flows f_4 and f_5 will contribute to the delays of both f_1 and f_2. On the other hand, a smaller SIRL might trigger a collapse, for example, before (and after) the appearance of f_1 and f_2. In this case, it may successfully prune one appearance of f_4 and f_5, thereby causing the situation (which is referred to as an *anomaly*), where, with a smaller value of SIRL, BPC returns a tighter WCTT estimate. As is visible from the results, the number of anomalies never exceeds 8%, for all the analysed flow-sets.

2.2.5.4 Experiment 4: All-to-One Case Study

Figure 2.14 shows one flow-set where all flows have the same destination. This example is used to demonstrate some interesting properties. Consider the flow f_1. The flows f_2, f_3, f_4, f_5, f_6, f_7, f_8 and f_9 are the other flows which can block f_1. By applying the existing approach [32], one of the sequences which might lead to the worst-case scenario for f_1 is: $Seq_1 = \{f_9, f_8, f_9, f_5, f_9, f_4, f_9, f_8, f_9, f_5, f_9, f_3, f_9, f_8, f_9, f_5, f_9, f_4, f_9, f_8, f_9, f_5, f_9, f_2, f_9, f_8, f_9, f_5, f_9, f_4, f_9, f_8, f_9, f_5, f_9, f_3, f_9, f_8, f_9, f_5, f_9, f_4, f_9, f_8, f_9, f_5, f_9, f_1\}$. This sequence is a good example of the



(a) Number of blockings by f_9 vs. MIRT

(b) WCTT of f_1 vs. MIRT

Figure 2.15: Impact of MIRT on flow f_1

potential packet-level and flow-level pessimism, and illustrates the case of a highly over-estimated WCTT, when infeasible sequences are not pruned. Flows f_9 and f_8 are positioned in such a way that they can frequently block the other flows. That is, in the aforementioned sequence, during one traversal of f_1, the flows f_9 and f_8 appear 24 and 8 times, respectively. However, due to their characteristics, it may be impossible that the packets of f_9 and f_8 appear so often, inferring that the computed sequence may be overly pessimistic. Yet, in order to assess whether the aforementioned sequence is indeed pessimistic, and to what extent, flow characteristics must be taken into account. In the reminder of the experiment the focus will be on this aspect.

As already observed, the flows originating closer to the destination have a higher tendency to block f_1 directly and indirectly (by blocking the other flows which are also on its path). To verify this, the deadline and the period of each flow are assigned the same value, which is equal to the worst-case traversal time of the respective flow, computed with the existing method [32]. For example, the deadline and period of the flow f_9 are assigned the delay of the sequence $Seq_9 = \{f_1, f_9\}$. Similarly, the deadline and period of f_8 are assigned the delay of the following sequence: $Seq_8 = \{f_9, f_1, f_9, f_8\}$. This process is repeated for all flows, and it assures that all flows can generate their packets as frequently as possible, while still being schedulable.

Assuming this flow-set, the worst-case traversal times are computed for each flow. Then, the periods of all flows are increased by the value of the newly introduced parameter MITR, while the deadlines remain the same, i.e. $D_{new}(f_i) = D_{old}(f_i) \wedge T_{new}(f_i) = T_{old}(f_i) + MITR, \forall f_i \in \mathscr{F}$. The worst-case traversal times are computed again, and the obtained values are compared. The objective of this experiment is to see how the obtained values change with the increase in flow periods (parameter MITR).

Figure 2.15 shows that as MIRT increases, the number of times the other flows can block f_1 decreases, and thus the WCTT estimate of f_1 decreases, as expected. In contrast, since the existing approach [32] does not take into account flow characteristics, the infeasible sequences are not pruned and as a result, irrespective of the change in the flow parameters, the obtained WCTT bounds remain constant (see solid line in Figure 2.15(b)).

2.2.6 Discussion

Branch, Prune and Collapse (BPC) is a method for the worst-case analysis of wormhole-switched round-robin-arbitrated NoCs. BPC uses a branch and prune technique, which improves over the work of Ferrandiz et al. [32] by taking into account the flow characteristics, and thereby provides tighter upper-bound estimates on the worst-case traversal times of traffic flows. In order to tackle the complexity issues of the branch and prune technique, a collapse phase was introduced. The concept of collapses allows the system designer to efficiently use the BPC method, and, via a configurable parameter, control the trade-off between the computational and spatial complexity on one side and the analysis tightness on another. A large set of experiments demonstrated the performance of the proposed method in comparison with the existing approach. In particular, BPC dominates the state-of-the-art method by yielding tighter WCTT estimates at the cost of additional computational and spatial resources, where these effects can be, to some extent, mitigated by the right selection of the configurable parameter.

This inability of the existing methods to efficiently derive tight bounds for large complex flow-sets is the consequence of two inherent properties of the round-robin arbitration policy itself, namely the *indirect contentions* and the *commutative property*. As already noticed, each flow can be blocked not only by other flows with which it directly competes for some links on its path, but also by other flows with which it does not. Consequently, the set of possible flow sequences that needs to be investigated constitutes a vast solution space, which may be impossible to efficiently explore within a reasonable time, even for small flow-sets. For example, for the flow-set of only 5 flows, illustrated in Figure 2.10, there are 305 possible flow sequences, while for the example of 9 flows illustrated in Figure 2.14, a single flow sequence may contain 48 elements. Moreover, the commutative property applied to flow blocking may hold in the context of the round-robin arbitration policy. In other words, if two flows f_i and f_j compete for some link on their path, in some cases f_i would be given a precedence, while in some others f_j would be allowed to progress. This infers that f_i can block f_j, but the opposite is also true. Since the arbitration decisions are based on the router state at runtime (previous routing decisions), it is impossible to predict routing decisions at design-time, which means that this arbitration non-determinism has to be covered in the worst-case analysis either by exploring all possible scenarios, or by making some pessimistic assumptions. Thus, it can be concluded that, despite its high popularity and wide presence in currently available many-core interconnects, the round-robin arbitration policy may not be the most efficient arbitration technique for the real-time many-cores.

2.3 NoCs with Priority-Preemptive Arbitration Policies

In this section, the focus is on priority-preemptive arbitration policies. As already discussed and demonstrated in Chapter 1, by allowing priority-based preemptions among flows, the effects of indirect contentions are significantly mitigated, which will be explored in detail later in this section. Moreover, for schemes with fixed flow priorities, the commutative property applied to flow

preemptions does not hold. In other words, if two flows f_i and f_j contend for some link on their paths, such that $P(f_i) > P(f_j)$, then f_i would always preemt f_j. Both the aforementioned facts allow to perform the worst-case analysis with much less pessimism and/or computational complexity, which infers that the priority-preemptive NoCs may be a preferable interconnect medium for real-time many-cores. Before the state-of-the-art methods for the worst-case analysis are presented, several basic concepts are introduced.

Definition 1 (Directly contending flow)**.** *If a flow f_j shares a part of the path with the flow under analysis f_i, and has a higher priority than f_i, it is considered as the directly contending flow, and it belongs to the set $\mathscr{F}_D(f_i)$. Formally:*

$$\forall f_j \in \mathscr{F} \mid P(f_j) > P(f_i) \wedge \mathscr{L}(f_j) \cap \mathscr{L}(f_i) \neq \emptyset \Rightarrow f_j \in \mathscr{F}_D(f_i)$$

Definition 2 (Indirectly contending flow)**.** *If a flow f_k does not share a part of the path with the flow under analysis f_i, but shares it with some other flow f_j, which is either directly or indirectly contending flow of f_i (recursive definition), and has a higher priority than f_j, it is considered as the indirectly contending flow, and it belongs to the set $\mathscr{F}_I(f_i)$. Formally:*

$$\forall f_k \in \mathscr{F} \mid f_k \notin \mathscr{F}_D(f_i) \wedge \exists f_j \in \mathscr{F} \wedge f_j \in \{\mathscr{F}_D(f_i) \cup \mathscr{F}_I(f_i)\} \wedge f_k \in \mathscr{F}_D(f_j) \Rightarrow f_k \in \mathscr{F}_I(f_i)$$

Recall, that for NoCs with the round-robin arbitration policy, the analysed flow could be blocked by both directly and indirectly contending flows. However, assuming priority-preemptive NoCs, the flow under analysis can suffer the interference only from directly contending flows (see Theorem 3).

Theorem 3. *In wormhole-switched NoCs, with per-flow distinctive priorities, per-priority virtual channels and flit-level preemptions, any flow f_i can not suffer interference from indirectly contending flows.*

Proof. Proven by contradiction. Consider three flows f_i, f_j and f_k. Let f_k be a directly contending flow of f_j, and f_j be a directly contending flow of f_i, where f_k and f_i do not share a common part of the path. Thus, by Definition 2, f_k is an indirectly contending flow of f_i. Assume that f_k can cause interference to f_i. By initial assumption, f_i is preempted. Furthermore, as f_k causes the interference, it is traversing. Due to the traversal of f_k, the flow f_j is either preempted, or does not exist at that time instant. In either case, it is not progressing. Thus, all routers on the path of f_i are idle. Due to per-priority virtual channels, f_i can uninterruptedly reach its destination, even though preempted packets of f_j may exist on its path (which would not be the case in a scheme with a single virtual channel, where f_i would have to wait until f_j passes). The contradiction has been reached. □

One implication of Theorem 3 is that in the example illustrated in Figure 2.16, the flow f_1 can not suffer the interference from the flow f_3. However, notice that f_3 can preempt f_2, influence its occurrence patterns, and in that way indirectly contribute to the delay of f_1. This is a very important fact, and it will receive additional attention later, when the state-of-the-art methods for the worst-case analysis will be introduced.

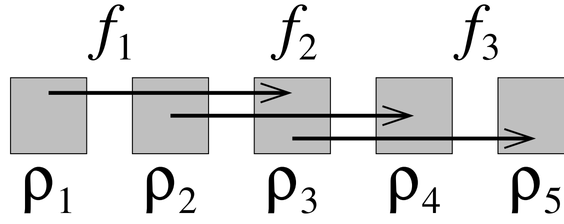


Figure 2.16: Traffic flows (example 3)

Observation 4. *The analysed flow f_i can be preempted by any directly interfering flow $f_j \in \mathscr{F}_D(f_i)$, and consequently suffer interference. The delay caused to f_i by a single preemption of f_j is equivalent to its basic network latency – $C(f_j)$, which is computed by solving Equation 2.1.*

2.3.1 State-of-the-Art Method for NoCs with Fixed-Priority Arbitration Policy

Shi and Burns [85] proposed a method for the worst-case analysis of NoCs with the fixed-priority arbitration. This approach is based on the following assumptions: (i) flit-level preemptions, (ii) per-flow distinctive priorities, (iii) per-priority virtual channels, and (iv) a traffic model with constrained or implicit deadlines, i.e. $D(f_i) \leq T(f_i), \forall f_i \in \mathscr{F}$. The authors proposed to treat the entire path of the flow under analysis as an indivisible resource, while any request for any of its parts by directly contending flows is considered as interference[1]. This allows to reuse the concepts from the single-core scheduling theory. Specifically, the problem of computing the worst-case traversal time of a flow translates into the problem of computing the worst-case response time of the task which is scheduled upon a single-core platform. By straightforwardly applying the concepts of the single-core scheduling theory, the worst-case traversal time of the flow f_i in the presence of its directly contending flows $\mathscr{F}_D(f_i)$ can be computed as follows:

$$WCTT(f_i) = C(f_i) + \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \left\lceil \frac{WCTT(f_i) + J_R(f_j)}{T(f_j)} \right\rceil \cdot C(f_j) \tag{2.2}$$

Equation 2.2 is interpreted as follows. The worst-case traversal time of a flow is equal to the sum of its isolation delay and its interference delay. The interference is computed by summing up the interferences from all higher-priority flows, where individual terms are obtained by calculating the maximum load that each higher-priority flow can generate within the observed time interval, augmented by the additional term $J_R(f_j)$. $J_R(f_j)$ is called the *release jitter*, which is defined as the maximum deviation of successive packet releases from its period [6].

Notice that the term $WCTT(f_i)$ exists on the both sides of Equation 2.2. This equation is solved using an iterative technique, where the first iteration is performed by assuming that the term $WCTT(f_i)$ from the right-hand side is equal to 0. The computed value (the term $WCTT(f_i)$ from the left-hand side) is then fed back into the calculation as the right-hand side term, and the computation is performed iteratively. The process terminates when $WCTT(f_i)$ does not change in

[1]Notice the difference between this approach and the methods for the round-robin-arbitrated NoCs, where the analysis was performed per-link, recursively.

two successive iterations. The obtained value represents the smallest value of $WCTT(f_i)$ which satisfies Equation 2.2. This is a well-known and widely used technique in the real-time domain [6].

Now, consider the example of flows illustrated in Figure 2.16 with the flow parameters given in Table 2.2.

Table 2.2: Flow-set parameters for Figure 2.16 (example 1)

Flow	**Priority**	***C(f)***	***J_R(f)***	***D(f) = T(f)***
f_1	$P(f_1)$	3	0	10
f_2	$P(f_2) < P(f_1)$	2	0	6
f_3	$P(f_3) < P(f_2)$	2	0	5

The worst-case traversal times of the flows can be obtained as follows:

$$WCTT(f_1) = C(f_1) = 3$$

$$WCTT(f_2) = C(f_2) + \left\lceil \frac{WCTT(f_2) + J_R(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) = 5$$

$$WCTT(f_3) = C(f_3) + \left\lceil \frac{WCTT(f_3) + J_R(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) = 4$$

Since the worst-case traversal times of all the flows are less than their respective deadlines, it may be concluded that this flow-set is schedulable. **But that is not true!** Figure 2.17 demonstrates that the packet of f_3 can miss its deadline. The explanation is as follows. Even though f_1 cannot directly interfere with f_3 because they do not have a common part of the path, f_1 can influence the occurrence pattern of f_2 and in that way indirectly contribute to the delay of f_3. Notice in Figure 2.17 that f_1 delayed the first packet of f_2, causing its two successive packets to be distanced by less than T_2. Consequently, f_3 experienced more interference from f_2 than what was computed with Equation 2.2. In particular, within the observed interval, f_3 suffered the interference from two packets of f_2, while the analysis considered the interference from only one packet. Thus, in the presence of indirectly contending flows, assuming periodic occurrences of higher-priority flows is not safe.

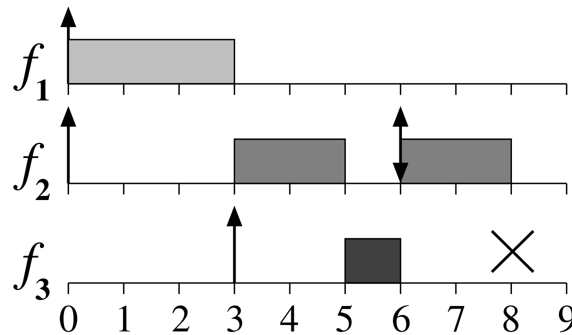


Figure 2.17: Deadline miss for flow-set from Figure 2.16 and Table 2.2

Shi and Burns [85] noticed this effect, and proposed the modification to Equation 2.2, which takes into account the impacts of indirectly contending flows. The intuition behind their approach is the following: for each higher-priority directly contending flow, which releases can be deferred due to indirectly contending flows, assume that the first packet is delayed as much as possible, while all the other packets are released as early as possible. The maximum delay that the first packet may experience while still being schedulable is $J_N(f_i) = WCTT(f_i) - C(f_i)$, which is in the literature known as the *maximum network jitter*. Thus, the authors propose to assume the maximum jitter for each higher-priority contending flow f_j that can suffer the interference from at least one flow f_k, which is indirectly contending with the analysed flow f_i. Conversely, if f_j *cannot* suffer the interference from any flow which is indirectly contending with f_i, then its jitter is equal to zero, i.e. $J_N(f_j) = 0$. The jitter computation can be expressed with Equation 2.3.

$$J_N(f_j) = \begin{cases} WCTT(f_j) - C(f_j) & \text{if } \exists f_k \in \mathscr{F} \mid f_k \in \mathscr{F}_D(f_j) \wedge f_k \in \mathscr{F}_I(f_i) \\ 0 & otherwise \end{cases} \tag{2.3}$$

Notice from Equation 2.3 that the network jitter does not depend only on the higher-priority flow f_j, but also on the analysed flow f_i and indirectly interfering flows (if any). This implies that if two different flows f_i and $f_{i'}$ have the same directly interfering higher-priority flow f_j, it may happen that the network jitters of f_j in these two cases have different values.

Now, a safe upper-bound on the worst-case traversal time of f_i can be computed by iteratively solving Equation 2.4.

$$WCTT(f_i) = C(f_i) + \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \left\lceil \frac{WCTT(f_i) + J_R(f_j) + J_N(f_j)}{T(f_j)} \right\rceil \cdot C(f_j) \tag{2.4}$$

Consider again the flow f_3 from the previous example. The indirect interference that f_1 causes to f_3 is manifested with the jitter of f_2, i.e. $J_N(f_2) = WCTT(f_2) - C(f_2) = 3$. And indeed, after recomputing the worst-case traversal time of f_3 with Equation 2.4, it is confirmed that f_3 is in fact unschedulable:

$$WCTT(f_3) = C(f_3) + \left\lceil \frac{WCTT(f_3) + J_R(f_2) + J_N(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) = 6 > D(f_3)$$

2.3.2 Priority-Share Policy

The greatest limitation of the aforementioned method is the requirement that each flow must have a distinctive priority, and that the platform must provide the number of virtual channels which is at least equal to the number of flows in the flow-set. This requirement is in most of the cases unrealistic. For example, the SCC platform [42] provides only 8 virtual channels. This infers that only flow-sets with at most 8 flows can be accommodated, which is a huge limitation. Thus, there is a need to more efficiently organise the access of the flows to the existing virtual channels.

In order to reduce the number of needed virtual channels, Shi and Burns [86] proposed the method where multiple flows can share the same priority, called a priority-share policy. However, the existence of flows with the same priority brings significant overheads and more complex blocking and interference patterns, similar to those involving a single virtual channel and the round-robin arbitration policy. That is, a flow can be blocked not only by the other same-priority flows with which it shares some parts of the path, but also by others with which it does not. For example, if the flows f_1, f_2 and f_3 from Figure 2.16 share the same priority, then f_3 can indirectly block f_1.

In order to circumvent this problem the authors propose to group all flows with the same priority into a single entity called the *composite flow* – $\mathring{f}$. Let $\mathscr{F}_C(\mathring{f})$ be a set of same-priority flows constituting $\mathring{f}$. Now, the basic network latency of the composite flow $\mathring{f}$ is computed as follows:

$$C(\mathring{f}) = \sum_{\forall f_i \in \mathscr{F}_C(\mathring{f})} C(f_i) \tag{2.5}$$

Let $\mathscr{F}_D(\mathring{f})$ be a set of flows which can cause direct interference to any flow from $\mathscr{F}_C(\mathring{f})$, and hence to $\mathring{f}$. Formally:

$$\forall f_j \in \mathscr{F} \mid \exists f_i \in \mathscr{F}_C(\mathring{f}) \wedge f_j \in \mathscr{F}_D(f_i) \Rightarrow f_j \in \mathscr{F}_D(\mathring{f}) \tag{2.6}$$

$\mathring{f}$ can suffer interference from any $f_j \in \mathscr{F}_D(\mathring{f})$. Now, the worst-case traversal time of the composite flow can be computed by solving Equation 2.7.

$$WCTT(\mathring{f}) = C(\mathring{f}) + \sum_{\forall f_j \in \mathscr{F}_D(\mathring{f})} \left\lceil \frac{WCTT(\mathring{f}) + J_R(f_j) + J_N(f_j)}{T(f_j)} \right\rceil \cdot C(f_j) \tag{2.7}$$

Subsequently, the worst-case traversal time of each flow $f_i \in \mathscr{F}_C(\mathring{f})$ can be computed as follows:

$$WCTT(f_i) = WCTT(\mathring{f}) + J_R(f_i) \tag{2.8}$$

Notice two potential sources of pessimism: (i) all same-priority flows are grouped into one entity, while some of them may not be able to (in)directly block each other, and hence could be treated independently, and (ii) not all directly interfering flows can cause interference during an entire period $WCTT(\mathring{f})$. To emphasise the limitations of the priority-share policy, consider the example illustrated in Figure 2.16 with the flow parameters given in Table 2.3.

Table 2.3: Flow-set parameters for Figure 2.16 (example 2)

Flow	**Priority**	***C(f)***	***J_R(f)***	***D(f) = T(f)***
f_1	$P(f_1)$	2	0	10
f_2	$P(f_2) < P(f_1)$	2	0	10
f_3	$P(f_3) < P(f_2)$	2	0	10

Assuming that each flow has a dedicated virtual channel, the worst-case traversal times are as follows:

$$WCTT(f_1) = C(f_1) = 2$$
$$WCTT(f_2) = C(f_2) + \left\lceil \frac{WCTT(f_2) + J_R(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) = 4$$
$$WCTT(f_3) = C(f_3) + \left\lceil \frac{WCTT(f_3) + J_R(f_2) + WCTT(f_2) - C(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) = 4$$

However, when assuming that all three flows share the same priority (and hence the virtual channel), the worst-case traversal times of the flows are:

$$WCTT(\mathring{f}) = C(\mathring{f}) = C(f_1) + C(f_2) + C(f_3) = 6$$
$$WCTT(f_1) = C(\mathring{f}) + J_R(f_1) = 6$$
$$WCTT(f_2) = C(\mathring{f}) + J_R(f_2) = 6$$
$$WCTT(f_3) = C(\mathring{f}) + J_R(f_3) = 6$$

The aforementioned example demonstrated that the priority-share policy can indeed reduce the number of needed virtual channels to an arbitrary value, however, at the expense of a substantial increase in the worst-case traversal times. Notice that this can have a significant impact on the schedulability, where many flows that are schedulable with per-flow priorities are not schedulable when some flows start sharing the same priority. This finding suggests that the priority-share policy may not be the most efficient method for the reduction of required virtual channels, and some alternative techniques are desirable.

2.3.3 Relaxing Hardware Requirements

The state-of-the-art method for flows with distinctive priorities [85] poses an unrealistic requirement regarding the number of virtual channels. The subsequently proposed priority-share policy [86] gives the possibility to relax that requirement (reduce the number of needed virtual channels), but at the expense of schedulability. In this section, the focus is on providing an alternative method, which allows to reduce the number of needed virtual channels, and at the same time does not have a significant impact on the schedulability.

2.3.3.1 Inefficiency of Existing Methods

The common underlying assumption of the two aforementioned approaches is that each flow traverses its entire path through the same, statically assigned virtual channel, which is dedicated to its priority. In other words, the assignment of virtual channels to priorities has to be done in a consistent manner across the entire platform, which means that many port-buffers belonging to a certain virtual channel will remain unused, simply because there are no flows with that priority

which traverse them. The limitation of this approach is explained with Figure 2.18 where 4 flows with distinctive priorities traverse 4 virtual channels, and where different virtual channels are depicted with different colors. Notice that the virtual channel for f_1 (the darkest color) is used only in the first two routers, while in the rest of the platform remains unused.

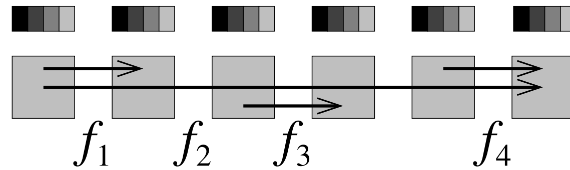


Figure 2.18: Assignment of virtual channels to flow-set with distinctive priorities

The priority-share policy suffers from the same limitation, which is illustrated with Figure 2.19. Consider the depicted flow-set where flows f_1 and f_4 are grouped within the same priority (virtual channel), and the same is true for f_2 and f_3. Notice that the virtual channel shared by f_1 and f_4 (the darker color) is used only in the first two and the last two routers, while in the middle two routers remains unused.

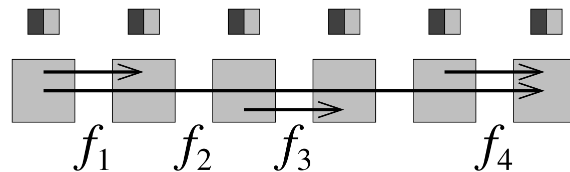


Figure 2.19: Assignment of virtual channels to flow-set with priority-share policy

2.3.3.2 Dynamically Changing Virtual Channels

The SCC platform [42] provides a feature that can be used to overcome the aforementioned limitation. Specifically, SCC offers the possibility that a flow can dynamically change virtual channels along its path. This feature was described using an analogy about how cars switch lanes on the highway. This implies that the assignment of virtual channels to priorities does not need to be done in a consistent manner across the entire platform, but can be performed individually, for each contending element – link (port). The benefits of this feature are illustrated in Figure 2.20, where for 4 flows with distinctive priorities only 2 virtual channels are needed. Notice, that f_2 changes virtual channels twice along its path (the darkest color).

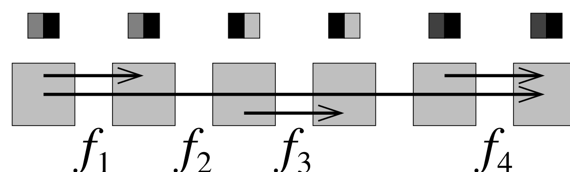


Figure 2.20: Per-router assignment of virtual channels to flow-set with distinctive priorities

This approach brings two benefits. First, within each port, the virtual channels are assigned to a certain priority only if necessary (the flow or flows with that priority traverse that port). Second, the number of required virtual channels within the entire platform is significantly reduced. Specifically, assuming that each flow has a distinctive priority, the number of necessary virtual channels decreases from the total number of flows in the flow-set to the maximum number of contentions for any link within the platform. In the illustrated example, it dropped from 4 (Figure 2.18) to only 2 (Figure 2.20). Notice, that unlike the priority-share policy, this technique to reduce the number of virtual channels does not have an impact on the schedulability and that the analysis remains the same as in the case with the per-flow distinctive priorities (Section 2.3.1), which makes it a more preferable approach than the priority-share policy.

One overhead of this approach is that, for each flow, a list of traversed virtual channels should be specified. This can be implemented in the following way. The header of each flow f_i is extended with the list of virtual channels $\mathscr{V}(f_i)$, where the number of elements in $\mathscr{V}(f_i)$ is equal to the number of ports on the path $\mathscr{L}(f_i)$ of the flow f_i, i.e. if routers have only input or output ports, $|\mathscr{V}(f_i)| = |\mathscr{L}(f_i) - 1|$, otherwise $|\mathscr{V}(f_i)| = 2 \cdot |\mathscr{L}(f_i) - 1|$. In other words, for each port that f_i traverses, there is an information in which virtual channel it should be stored. One alternative implementation is that the flows are agnostic with respect to virtual channels, but the information is stored within each router, and upon the arrival of a flow, it is router's responsibility to retrieve that information and select the corresponding virtual channel.

2.3.3.3 Importance of Mapping

As soon as the methods for the worst-case analysis of NoCs were proposed, the researchers realised that the schedulability of flow-sets is highly dependent on the mapping process. This is expected, because as observed in the previous sections, the flow can suffer the interference only from directly contending flows, while flow relationships are in fact the consequence of the mapping process. In their studies, Mesidis and Indrusiak [62] and Racu and Indrusiak [80] focus on deriving mappings which maximise the schedulability potential of the flow-sets, but without any impact on the number of employed virtual channels, because the authors assumed the model with distinctive per-flow priorities. Similarly, Shi and Burns [87] explore the flow-set mapping problem, but for the model with the priority-share policy. Their primary objective is to improve the schedulability, while the secondary objective is to decrease the number of employed virtual channels by packing as much flows in the same priority (virtual channel) as possible.

So far, no work has considered that flows may dynamically change virtual channels. Notice, that under this assumption, the mapping process becomes even more important, because the number of necessary virtual channels is equal to the maximum number of contentions for any link, which is directly dependant on the mapping process.

At this stage several questions can be raised: 1. What is reduction in the number of virtual channels that can be achieved by allowing flows to change virtual channels? 2. What is the further reduction in the number of virtual channels that can be achieved through mapping? 3. Does SCC (or any other platform with a desired feature) offer enough virtual channels to satisfy the

requirements of present and future real-time systems? 4. If not, how big is the gap between the existing platforms and the platforms suitable for the real-time domain? In order to answer these questions, the novel approach will be proposed, which combines the dynamically changing flow priorities with the workload mapping.

2.3.3.4 Proposed Mapping Method

Mapping Workload

Before the mapping process can be described, the mapping workload has to be introduced. Recall, so far it has been assumed that each flow is specified with the source core and the destination core. However, flows model the communication between functionalities, thus the source and destination cores of each flow are in fact the cores of the sending and receiving functionality, respectively. Therefore, the mapping workload is a collection of functionalities $F = \{F_1, F_2, ...F_{z-1}, F_z\}$, where z is equal to the number of cores/routers. The assumption valid in this section is that each functionality can be mapped to any core within the platform, however, each core can accept at most one functionality. Each functionality F_i is characterised by a set of sent flows $\mathscr{F}_S(F_i)$ and the set of received flows $\mathscr{F}_R(F_i)$.

Method Overview

The mapping method is based on the assumption that, while traversing, flows are allowed to change virtual channels. The method derives a mapping plan (solution) $\mathscr{M}$, for a given set of functionalities F, on a given platform Ψ, i.e. $\mathscr{M} = F \rightarrow \Psi$. The primary objective is to minimise the maximum number of contentions for any link on the platform. The motivation behind this concept is to relax the requirements for virtual channels as much as possible. The secondary objective is to maximise the schedulability, without increasing the number of necessary virtual channels.

Such a mapping method allows to identify the minimum platform characteristics (e.g. the number of virtual channels, the link bandwidths) that are necessary to accommodate a given workload. A mapping method should be used by a system designer, and should ideally provide a mapping $\mathscr{M}$, of the given workload F, to the given platform Ψ, such that the flow-set is schedulable, i.e. $WCTT(f_i) \leq D(f_i), \forall f_i \in \mathscr{F}$. In cases where this requirement can not be fulfilled, the method should provide information about the service that a given platform can provide to a given workload, e.g. (i) the number of available virtual channels is enough to accommodate only half of the flow-set, or (ii) assuming initial flow sizes the system is not schedulable, however, assuming that each flow has a half of its initial size the system is schedulable. This information is useful for two reasons. First, it gives system designer feedback on how close/far the current configuration is from the one that can accommodate the entire workload and provide real-time guarantees. Consequently the system designer can reconfigure the platform characteristics and repeat the mapping process until finding the configuration which guarantees the fulfilment of all constraints, with as less resources as possible. Second, if the platform is already specified, the designer can decide (i) which least essential functionalities (and their respective flows) should be dropped from the

system, such that the constraint on the number of virtual channels is satisfied, and/or (ii) to which sizes should the flows be reduced, such that timing constraints of all flows can be met. Note, that variable flow sizes are in many scenarios acceptable, e.g. [13], and proportional to the provided quality of service. Hence, when mapping such a workload, the system designer might find it more convenient to use a platform where a workload is schedulable with a certain quality (flow sizes), rather than buying a much more expensive platform which will guarantee the best quality (schedulable system with initial flow sizes). This approach also allows to study the effect of different platform characteristics on the provided guarantees and will help to identify the bottlenecks of NoCs in commodity many-cores.

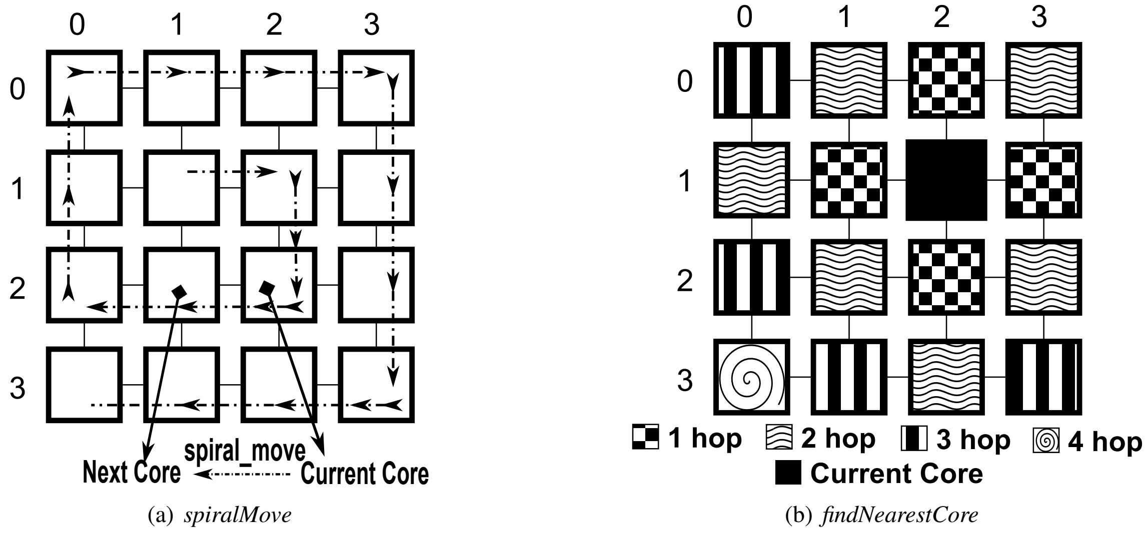
Detailed Description of Mapping Method

The proposed method consists of three mapping stages: (i) *initial phase*, (ii) *VC minimisation phase* and (iii) *workload-exploration phase*.

The initial phase is inspired by the mapping algorithm of dataflow applications on many-core platforms [4], where the objective is to map the set of functionalities F on the set of cores Π in such a way that heavily communicating functionalities are mapped as close to each other as possible. This strategy should help in reducing contentions and provide a "good" starting point, i.e. initial solution $\mathscr{M}_{in}$, which will undergo further optimisations during subsequent phases. While mapping, the initial phase does not consider flow, nor platform properties (i.e. flow sizes, timing constraints, link bandwidths, available virtual channels).

The communicating functionalities are mapped near to each other by using a core selection methodology proposed by Ali et al. [4], that proved to decrease communication overhead and response times of applications. This methodology consists of two main functions, *spiralMove* and *findNearestCore*, which are illustrated in Figure 2.21. The first function, *spiralMove*, defines a fixed spiral path on the platform that is followed while mapping the set of functionalities F. The *spiralMove* function returns the next core on the spiral path every time it is called, as shown in Figure 2.21(a). The second function, *findNearestCore*, takes a reference core as an input and starts searching for a free core one hop away from the reference core. If not possible, it searches for a free core two hops away, and so on, until finding a possible core to map the functionality. The search criteria starts by finding the nearest core in this order: North, South, East and West. The first core that the *findNearestCore* function finds is returned for mapping. Figure 2.21(b) shows the searching regions, classified according to the distance from the reference core.

The mapping algorithm of the initial phase (shown in Algorithm 4), starts by sorting the functionalities in a non-increasing order of total number of flows (both sent and received). Then, it picks the functionalities in that order and checks whether they are mapped or not. If the functionality F_i was already mapped, the algorithm checks the children of F_i (the functionalities that communicate with F_i, either via received or sent flows), and maps them near F_i by using the function *findNearestCore*. On the other hand, if the picked up functionality was not mapped yet, the algorithm calls the *spiralMove* function to determine the next free core on the spiral path to map it. Then, it maps its children functionalities using the *findNearestCore* function in the same way



(a) *spiralMove* (b) *findNearestCore*

Figure 2.21: Core selection methodology proposed by Ali et al. [4]

Algorithm 4 $mapInitialPhase(F, \Psi)$

Input: set of functionalities F, platform Ψ
Output: initial mapping $\mathscr{M}_{in}$

1: $F_{ord} \leftarrow orderFunctionalities(F)$; // sort functionalities by the number of flows, non-increasingly
2: **for each** $(F_i \in F_{ord})$ **do**
3:   **if** $(F_i.mapped \neq true)$ **then**
4:     $\pi_k \leftarrow spiralMove()$; // find the best core for mapping
5:     $map(F_i, \pi_k)$;
6:     $F_i.mapped \leftarrow true$;
7:   **end if**
8:   $F_{ch}(i) \leftarrow findChildFunctionalities(F_i)$; // find all child functionalities of F_i
9:   **if** $(F_{ch}(i) \neq \emptyset)$ **then**
10:     **for each** $(F_j \in F_{ch}(i))$ **do**
11:       $\pi_m \leftarrow findNearestCore(F_i)$;
12:       $map(F_j, \pi_m)$;
13:       $F_j.mapped \leftarrow true$;
14:     **end for**
15:   **end if**
16: **end for**
17: **return** $\mathscr{M}_{in}$;

as explained previously.

The VC minimisation phase takes the solution of the initial phase $\mathscr{M}_{in}$ as an input, and optimises it in such a way that the number of needed virtual channels is minimised. The VC minimisation phase is implemented as a Simulated Annealing (SA) meta-heuristic [50]. The justification for this approach is the fact that the problem of workload mapping is equivalent to the quadratic assignment problem, which is NP-Hard, hence searching for the optimal solution can be

Algorithm 5 $mapVCMinPhase(F, \Psi, \mathscr{M}_{in})$

Input: set of functionalities F, platform Ψ, initial solution $\mathscr{M}_{in}$
Parameters: t_{min} – min SA temperature, t_{max} – max SA temperature, t_{step} – temperature step, c – iterations in each step, P_w – probability of transitioning from the current state
Output: intermediate solution $\mathscr{M}_{min}$

```
1:  M_min ← M_in;
2:  for (t_curr ← t_max; t_curr > t_min; t_curr ← t_curr − t_step) do
3:      it ← 0; // number of iterations at the current temperature
4:      while (it ≤ c) do
5:          {F_i, F_j | (F_i ∧ F_j) ∈ F, F_i ≠ F_j} // select two random functionalities F_i and F_j
6:          M_curr ← swap(M_min, F_i, F_j); // create new solution by swapping the positions of F_i and F_j
7:          if (v̂(M_curr) ≤ v̂(M_min)) then
8:              if ((v̄(M_curr) < v̄(M_min)) ∨ (rnd() < P_w · t_curr/t_max)) then
9:                  M_min ← M_curr; // new solution accepted
10:             end if
11:         end if
12:         it ← it + 1;
13:     end while
14: end for
15: return M_min;
```

prohibitively expensive even for small NoCs [40], e.g. 4×4. Note that the majority of workload mapping methods are also heuristic-based.

As the single objective of the VC minimisation phase is to minimise the number of virtual channels, this phase also does not consider flow nor platform properties. This phase is described with Algorithm 5. Through the exploration of the solution space, the algorithm makes a transition from one solution $\mathscr{M}_{min}$ to another $\mathscr{M}_{curr}$ with respect to an *acceptance test*. This test consists of two conditions, of which at least one must be satisfied in order to accept the new solution. These two conditions are: (i) the average number of contentions (computed for all links) of the new solution is less than the average number of contentions of the solution which currently requires the minimal number of virtual channels $\bar{v}(\mathscr{M}_{curr}) < \bar{v}(\mathscr{M}_{min})$, and (ii) the acceptance probability test function $(rnd() < P_w \cdot {}^{t_{curr}}/_{t_{max}})$, where $rnd()$ is a random number from the range $[0,1]$, P_w is the probability of accepting a new solution, while t_{curr} and t_{max} are the current and the maximum temperature of the algorithm, respectively. As is visible, the acceptance probability test is a function of t_{curr}. This means that at high temperatures the algorithm has a higher tendency to accept worse solutions in an attempt to transition to a new solution space where it can find a better solution. As the algorithm temperature cools down, this tendency decreases and the algorithm starts to lock on the best solution in the current neighbourhood.

As shown in Algorithm 5, the algorithm starts its iterations by setting the current temperature t_{curr} to a high value t_{max}. Then, two functionalities are randomly selected and swapped. Consequently, the rerouting of their respective incoming and outgoing flows is performed. If the number of needed virtual channels for the new solution $\hat{v}(\mathscr{M}_{curr})$ is less than or equal to the number of virtual channels for the currently best solution $\hat{v}(\mathscr{M}_{min})$, the new solution is eligible to evaluate

its acceptance probability using the acceptance test previously described, otherwise, it is rejected. In cases where the new solution progresses to the acceptance test and subsequently passes, the algorithm accepts the new solution, i.e. $\mathscr{M}_{min} \leftarrow \mathscr{M}_{curr}$. Otherwise, the best solution remains the same.

The workload-exploration phase is the last phase and its main role is to discover and quantify the guarantees that can be provided for a given workload F by a given platform Ψ. It takes the output of the VC minimisation phase $\mathscr{M}_{min}$ as an input, which assures that the starting point is the solution with the minimal number of virtual channels $\widehat{v}(\mathscr{M}_{min})$, and performs the optimisation with the objective to derive a feasible solution $\mathscr{M}_{we}$ (if possible), assuming specific platform characteristics (i.e. link bandwidth $B_{link} = \frac{\sigma_{flit}}{\delta_\rho + \delta_L}$ and available virtual channels $\widehat{v}_{lim} \geq \widehat{v}(\mathscr{M}_{min})$) as well as workload temporal constraints. The workload-exploration phase is also implemented as a SA meta-heuristic, and it should ideally derive a feasible solution $\mathscr{M}_{we}$. In cases where it is not possible, the workload-exploration phase investigates to what percentage of initial sizes all flows have to be uniformly reduced, such that the feasibility can be guaranteed, and consequently tries to find a solution where the reduction is the least possible.

The workload-exploration phase attempts to maximize the size of all flows $\widehat{S}$ traversing the NoC, such that flows' timing constraints are satisfied, and that the number of employed virtual channels does not exceed the limit – $\widehat{v}_{lim}$, which represents the number of channels provided by the platform Π. Similar to the VC minimisation phase, this phase is also based on a SA meta-heuristic, but with a different goal, which is in this case to find a solution $\mathscr{M}_{we}$, such that $\widehat{S}(\mathscr{M}_{we})$ is maximised. This algorithm also makes a transition from one solution to another with respect to an acceptance test. However, the acceptance test of the workload-exploration phase differs in the first condition, where it compares the new and the best values of maximum feasible flow sizes, $\widehat{S}(\mathscr{M}_{curr}) > \widehat{S}(\mathscr{M}_{we})$, in order to accept new solutions.

As shown in Algorithm 6, the algorithm starts its iterations by setting the current temperature t_{curr} to a high value t_{max}. Then, two functionalities F_i and F_j are randomly picked and swapped in order to generate a new solution $\mathscr{M}_{curr}$. This also requires the rerouting of incoming and outgoing flows of the remapped functionalities. If the number of needed virtual channels of the new solution $\widehat{v}(\mathscr{M}_{curr})$ exceeds the number of virtual channels provided by the platform $\widehat{v}_{lim}$, the new solution $\mathscr{M}_{curr}$ is rejected. Otherwise, the maximum schedulable flow sizes of the new solution $\widehat{S}(\mathscr{M}_{curr})$ and the best one $\widehat{S}(\mathscr{M}_{we})$ are compared as a part of the acceptance test. These values are calculated by incrementing sizes of all flows of the flow-set uniformly, from zero up to their original sizes, and repeatedly testing whether all flows' timing constraints are still satisfied. Assuming the mapping $\mathscr{M}$, the value $0 < \widehat{S}(\mathscr{M}) \leq 1$ represents the maximum flow sizes for which the flow-set is still schedulable, where the value of 1 corresponds to the original flow sizes.

In cases where the new solution passes the acceptance test, the algorithm accepts the new solution as the best one, i.e. $\mathscr{M}_{we} \leftarrow \mathscr{M}_{curr}$. Otherwise, the new solution is discarded. After several iterations, the maximum flow size $\widehat{S}(\mathscr{M}_{we})$ can reach the value of 1. This means that the current mapping $\mathscr{M}_{we}$ of the set of functionalities F on the set of cores Π is schedulable with original sizes of all flows, so the mapping process can terminate. Otherwise, if $\widehat{S}(\mathscr{M}_{we})$ is less

Algorithm 6 *mapWorkloadExplorationPhase*$(F, \Psi, \mathscr{M}_{min})$

Input: set of functionalities F, platform Ψ, intermediate solution $\mathscr{M}_{min}$
Parameters: t_{min}, t_{max}, t_{step}, c, P_w, $\widehat{v}_{lim}$ – number of virtual channels available within the platform
Output: final solution $\mathscr{M}_{we}$

1: $\mathscr{M}_{we} \leftarrow \mathscr{M}_{min}$;
2: **for** $(t_{curr} \leftarrow t_{max}; t_{curr} > t_{min}; t_{curr} \leftarrow t_{curr} - t_{step})$ **do**
3: $it \leftarrow 0$; // number of iterations at the current temperature
4: **while** $(it \leq c)$ **do**
5: $\{F_i, F_j \mid (F_i \wedge F_j) \in F, F_i \neq F_j\}$ // select two random functionalities F_i and F_j
6: $\mathscr{M}_{curr} \leftarrow swap(\mathscr{M}_{min}, F_i, F_j)$; // create new solution by swapping the positions of F_i and F_j
7: **if** $(\widehat{v}(\mathscr{M}_{curr}) \leq \widehat{v}_{lim})$ **then**
8: **if** $((\widehat{S}(\mathscr{M}_{curr}) > \widehat{S}(\mathscr{M}_{we})) \vee (rnd() < P_w \cdot {}^{t_{curr}}/_{t_{max}}))$ **then**
9: $\mathscr{M}_{we} \leftarrow \mathscr{M}_{curr}$;
10: **if** $(\widehat{S}(\mathscr{M}_{we}) = 1)$ **then**
11: **return** $\mathscr{M}_{we}$;
12: **end if**
13: **end if**
14: **end if**
15: $it \leftarrow it + 1$;
16: **end while**
17: **end for**
18: **return** $\mathscr{M}_{we}$;

than 1, this means that the system can guarantee the schedulability, but only assuming that sizes of all flows are uniformly reduced to $\widehat{S}(\mathscr{M}_{we})$, so the mapping process will keep trying to find a better solution in subsequent iterations.

2.3.3.5 Experimental Evaluation

In this section, assuming the proposed mapping approach, several tests are performed in order to observe how platform characteristics influence derived schedulability guarantees. Specifically, it is investigated how guarantees change with varying number of virtual channels and link bandwidths. In order to provide a complete and comprehensive study, and cover a wide range of scenarios ranging from lightly to extremely loaded networks, different workloads are generated and consequently analysed.

Experimental Setup

The analysis parameters are given in Table 2.4, where flow sizes are randomly generated values, assuming a uniform distribution. For each flow a source and a functionalities F_S and F_D are randomly selected.

Experiment 1: Do Virtual Channels Scale?

The proposed approach is applicable to the mapping solution $\mathscr{M}$ only if a platform provides at least $\widehat{v}(\mathscr{M})$ virtual channels, where $\widehat{v}(\mathscr{M})$ is equal to the maximum number of contentions for any port within the NoC. The objective of this experiment is to test how (un)realistic that

Table 2.4: Analysis parameters for Section 2.3.3.5

NoC topology and size	**2-D mesh with 10 × 10 routers**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Link width = flit size σ_{flit}	**16 bytes**
Flow packet size $\sigma(f_i), \forall f_i \in \mathscr{F}$	**$[32B - 32kB]^*$ bytes**
Number of functionalities $\|F\|$	**100**

requirement is. This is performed as follows. First, for a given workload F, the mapping process is performed until reaching the final solution $\mathscr{M}_{we}$, and subsequently the number of needed virtual channels is identified. Then it is observed how the number of needed virtual channels change when the number of flows increases. For that purpose, different workload categories are generated, ranging from 50 to 1000 flows for an entire set of functionalities. Each category consists of 1000 sets of functionalities. For each functionality the mapping is performed (only initial and VC minimisation phases of the proposed approach, which output is the intermediate solution $\mathscr{M}_{min}$), and subsequently the minimum number of virtual channels needed to support it – $\widehat{v}(\mathscr{M}_{min})$ is computed. The results are given in Figure 2.22, where a horizontal axis corresponds to the number of flows and a vertical axis stands for the number of virtual channels. The whiskers were set to 25th and 75th percentile.

As already known, the NoC architecture is widely accepted due to its scalability potential related to traffic in general, hence an intuitive guess would be that the number of needed virtual channels also scales. Figure 2.22 confirms that assumption with an almost-linear dependency between the number of flows and virtual channels, and also gives a quantitative estimate regarding how many channels are needed for various workloads. In particular, currently available SCC platform, with its 8 virtual channels, can accommodate around 300 flows. On average, an addition of one channel increases the potential of the platform to accept 50 new flows. Recall, that the traditional distinctive-priority approach requires that the number of virtual channels is equal to the number of flows, and notice the reduction achievable with the proposed approach where flows are allowed to dynamically change virtual channels, i.e for 1000 flows the proposed method requires, on average, only 23 channels. The practical implications of these findings are further elaborated in Section 2.3.3.6.

Experiment 2: Do Virtual Channels Help?

The previous experiment demonstrated that the proposed approach efficiently maps the workload, such that the number of needed virtual channels is minimised to the level beyond any reasonable expectation. This implies that the mapping process evenly distributes the contentions across the entire platform and avoids hotspots. However, as the minimisation of port-contentions (i.e. virtual channels) is the central criteria, this can result in solutions where some flows are forced to traverse long routes, which can in turn have an impact on the schedulability. Conversely, placing highly communicative functionalities close to each other would minimise the flow routes

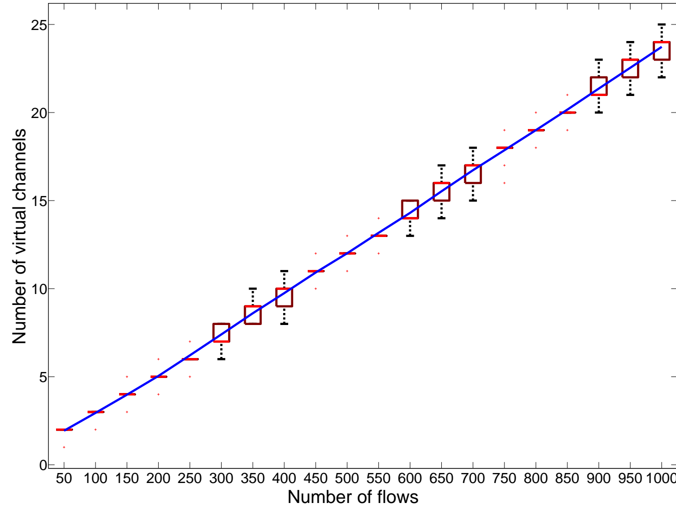
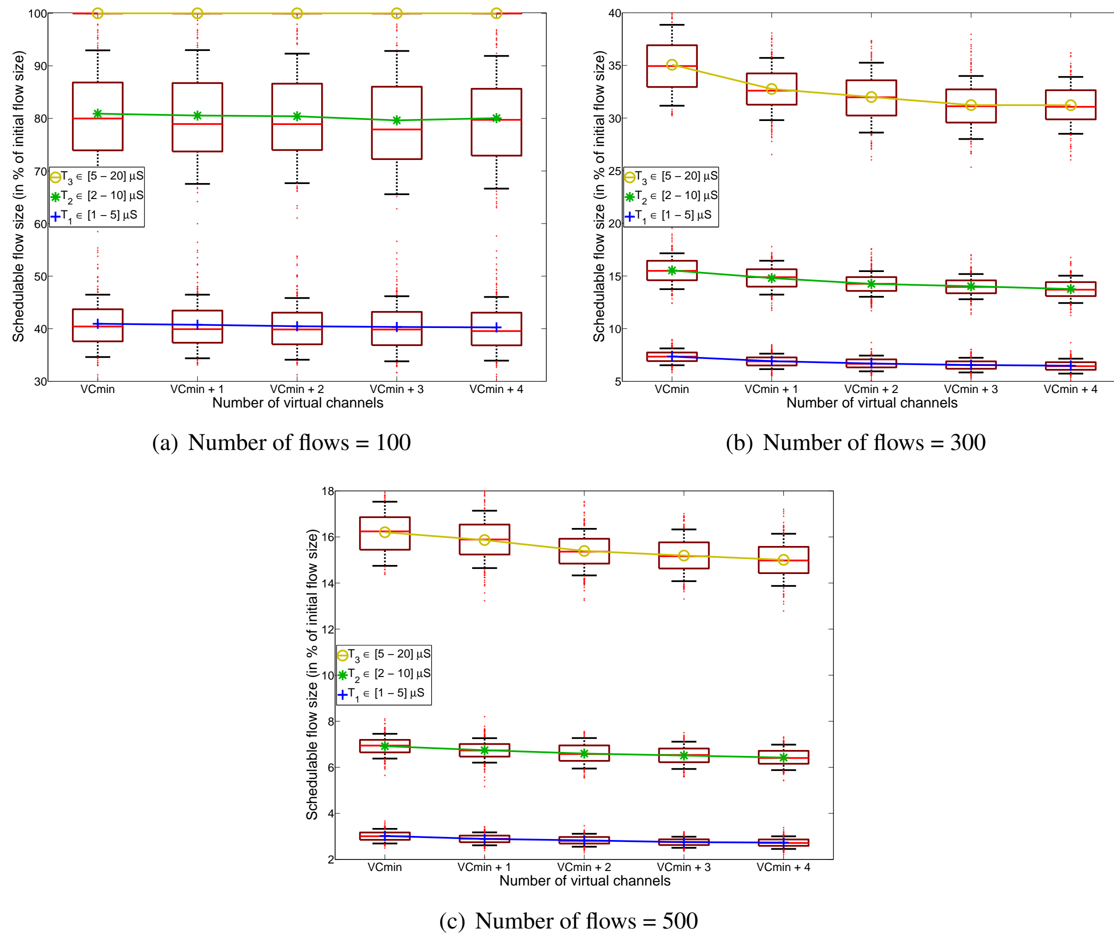


Figure 2.22: Virtual channels do scale with respect to number of flows

and might potentially improve the schedulability, but would create hotspots and cause more contentions for some ports. Thus, an intuitive guess would be that the schedulability highly depends on the number of available virtual channels, and the objective of this experiment is to quantify that trade-off, that is, to observe to what degree can the schedulability be improved by employing an additional virtual channel.

3 workload categories are generated, containing 100, 300 and 500 flows with implicit deadlines, i.e. $D(f_i) = T(f_i), \forall f_i \in \mathscr{F}$. For each category there are 3 subcategories with the following flow minimum inter-arrival periods: $T_1 \in [1-5]\mu s$, $T_2 \in [2-10]\mu s$ and $T_3 \in [5-20]\mu s$. For each subcategory 200 sets of functionalities are generated and consequently mapped. The results are given in Figure 2.23, where a horizontal axis stands for the number of virtual channels provided by the platform – $\widehat{v}_{lim}$, and the vertical axis stands for the provided schedulability guarantees, that is, to what percentage of initial sizes all flows have to be uniformly reduced, such that the system is schedulable. A value VC_{min} corresponds to the number of virtual channels obtained by the VC minimisation phase, i.e. $VC_{min} = \widehat{v}(\mathscr{M}_{min})$.

This experiment reported highly unexpected and unintuitive results, which implies that adding virtual channels might negatively impact the efficiency of the mapping process and produce a solution which is worse than the one obtained with the minimum number of virtual channels $\widehat{v}(\mathscr{M}_{min})$. These surprising results may be related to specific workloads or the consequence of inherent properties of SA meta-heuristic itself. To rule out the former possibility, the experiment was repeated several times, as described above, with very diverse traffic loads. However, the results remained very similar and consistently demonstrated a systematic decrease in schedulability guarantees as the number of virtual channels increased. To rule out the latter possibility, more experimentation is needed, with different heuristic and meta-heuristic approaches. Thus, we can conclude that, when mapping the workload with the aforementioned method, virtual channels are **not** bottlenecks. This can be interpreted in the following way: minimising virtual channels (contentions) indeed contributes to the schedulability. As this objective tends to distribute the contentions on



(a) Number of flows = 100

(b) Number of flows = 300

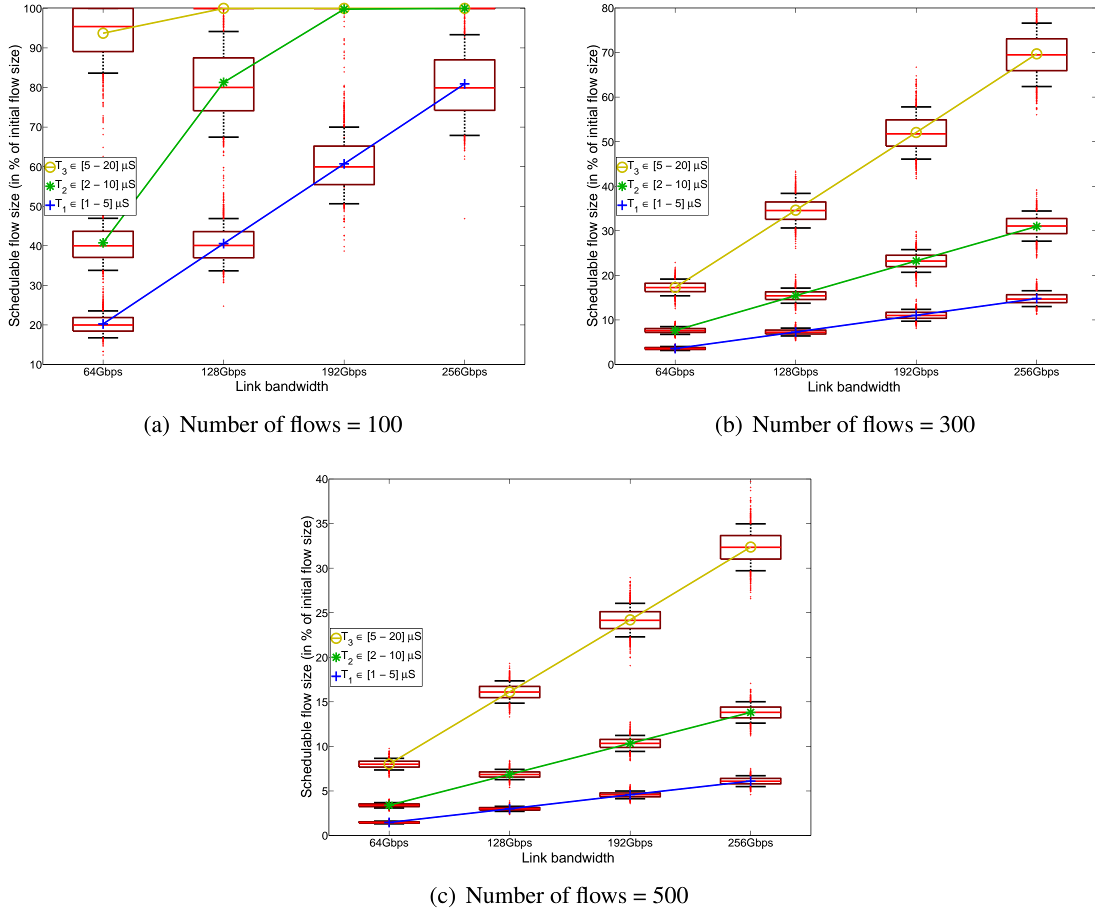
(c) Number of flows = 500

Figure 2.23: Additional virtual channels do **not** help in schedulability guarantees

the grid as evenly as possible, it might cause longer traversal paths of some flows. However, as the load is equally spread across all the links, even assuming those longer paths, better guarantees can be provided. Conversely, minimising flow distances causes hotspots and even if some flow traverses a short distance, links it consumes might be heavily loaded, hence highly impacting its schedulability. Therefore, the conclusion is that (i) the solution obtained with the minimal number of virtual channels is one of near-optimal solutions, and (ii) adding more virtual channels, in most cases, unnecessarily expands the solution space which causes SA to drift away from the "good" solution space and frequently conclude the search with some worse solution. Of course, this can not be analytically and/or experimentally proven as the workload mapping is an NP-Hard problem, which is computationally intractable even for small grids, e.g. 4×4 [40]. However, small scale experiments with exhaustive enumeration and different meta-heuristics are the possibilities to further support or deny the aforementioned claims, and these activities are potential future work. The implications of these surprising findings are further discussed in Section 2.3.3.6.

Experiment 3: Then, What is the Bottleneck?

As previous experiment provided experimental evidence which suggests that virtual channels



(a) Number of flows = 100

(b) Number of flows = 300

(c) Number of flows = 500

Figure 2.24: Link bandwidths are bottlenecks

are not the bottleneck, in order to test the limits of the platform, the focus of this experiment is on another parameter – the link bandwidth B_{link}. The individual link bandwidth of the SCC platform is around 256 Gbps, while Tilera platforms provide around 166 Gbps. Assuming a variety of network loads, it is observed how different link bandwidths influence provided schedulability guarantees.

3 workload categories are generated, consisting of 100, 300 and 500 flows with implicit deadlines. Each category has 3 subcategories, which differ in flow minimum inter-arrival periods: $T_1 \in [1-5]\mu s$, $T_2 \in [2-10]\mu s$ and $T_3 \in [5-20]\mu s$. For each subcategory, 1000 sets of functionalities are generated and consequently mapped, assuming that the platform provides only the minimal number of virtual channels, i.e. $\widehat{v}_{lim} = \widehat{v}(\mathscr{M}_{min})$. The results are given in Figure 2.24, where the horizontal axis stands for the link bandwidth, and the vertical axis represents the provided schedulability guarantees, expressed as the maximum percentage of initial flow sizes, to which all packets have to be reduced, such that the system is schedulable.

This experiment reported results which are expected, suggesting that the derived guarantees linearly depend on link bandwidths. The trend is similar for scenarios which cover moderately and extremely loaded networks. The only exceptions are cases for 100 flows and T_1 as well as T_2 periods (Figure 2.24(a)), where, due to lighter load, solutions can be found for which the

schedulability of initial flow sizes can be guaranteed even with smaller bandwidths. Thus, in general, link bandwidths can be perceived as the bottleneck of the system, which coincides with the intuition.

2.3.3.6 Discussion

The proposed technique to employ the existing feature of the SCC platform to minimise the number of needed virtual channels significantly relaxes the requirements for platform characteristics which are needed for the real-time analysis. Through experiments, it has been demonstrated that (i) the number of needed virtual channels scales linearly with the increasing traffic and (ii) the number is not unreasonably high and should be achievable with forthcoming generations of many-core platforms. It has been shown that limiting the number of channels to the minimum while mapping the workload is, in most cases, a beneficial approach, and leads to a near-optimal solution. The aforementioned facts altogether answer the questions posed in Section 2.3.3.3. However, these answers raise another question: After all, is the priority-share policy needed? As it has been demonstrated with the experiments, the distinctive-priority analysis can be performed with little overheads in terms of needed virtual channels, and thus it is very probable that the future real-time-oriented interconnect mediums will provide sufficient virtual channels to afford per-flow distinctive priorities and avoid the priority-share policy and the pessimism related to it.

Furthermore, link bandwidths were recognised as the bottleneck of the system. As this characteristic is equally important in high-performance and general-purpose computing, which are the main drivers for the advancements in interconnect mediums, it is reasonable to expect that link bandwidths are going to continue their increase in future years, which will be also appreciated by the real-time community.

2.3.4 EDF as Arbitration Policy

All the methods for the worst-case analysis of NoCs that have been introduced so far have been developed from the same underlying assumption that each flow has a fixed priority. In this section, it will be investigated whether allowing flows to dynamically change their priorities can have a positive impact on the schedulability. Specifically, a novel arbitration policy for NoC routers is proposed, which is based on the EDF paradigm [58] – a well-established concept in the scheduling theory. In this section, the following questions will be answered: 1. What are the prerequisites to enforce the EDF arbitration policy within NoC routers? (Section 2.3.4.1) 2. How to perform the worst-case traffic delay analysis? (Section 2.3.4.2) 3. Does this approach outperform the state-of-the-art methods, under which conditions and by how much? (Section 2.3.4.5) 4. What are the practical limitations of the approach? (Sections 2.3.4.5-2.3.4.6) 5. Ultimately, do its merits justify the overhead of enforcing a novel arbitration policy? (Section 2.3.4.6).

2.3.4.1 Prerequisites

EDF (Earliest Deadline First) is a well-known concept in the single-core scheduling theory [58]. EDF has been proven optimal in the following sense: if any scheduling policy (including the previously mentioned fixed-priority ones) can render the task-set schedulable, EDF will also be able to do so. EDF has also been studied from the perspective of multi-core platforms [9], however, it faces several challenges in that context: (i) it is shown that it is not very efficient, and (ii) due to the necessity to maintain global structures e.g. a ready-queue, scalability issues may arise. So far, no work has considered EDF as an arbitration policy for NoC routers and the work presented in this section is motivated by that fact.

Irrespective of whether it is applied to the scheduling or the NoC contention theory, the EDF policy arbitrates the access to the shared resource (e.g. a core, a link) based on the latest time instant until which contending entities (e.g. jobs, packets) have to complete their execution. Thus, one of the prerequisites to enforce such a policy in NoC routers is that, prior to sending, at time instant t, a packet of a flow f_i is tagged with its deadline, expressed in absolute values $d(f_i) = t + D(f_i)$. In the ideal case, if two packets, belonging to two distinctive flows f_i and f_j, with their respective deadlines $d(f_i)$ and $d(f_j)$, encounter each other and contend for some link on their paths, the one with the earlier deadline should have the precedence and win the arbitration. That is, if $d(f_i) < d(f_j)$, then the packet of f_i will be able to preempt the packet of f_j, and vice versa.

What are the requirements to implement such a policy? With respect to the router's logic, very few changes are needed; instead of packet priorities, packet deadlines are compared. However, tagging packets with their deadlines might be a challenging activity, because the timestamps may consume a lot of space. Also, notice that the value of the deadline highly depends on the perception of time of the core releasing it. Due to the finite signal propagation speed, different temperatures and different physical compositions, it is quite common that two components of the same system receive the same signal at different time instants. This infers that two cores may perceive the same time instant at two different moments, which is in the circuit design theory called the *clock skew*. Ultimately, the clock skew is inevitable and chip manufacturers employ various techniques to mitigate its effects, however, that topic is beyond the scope of this dissertation. Here, it is assumed that the parameter Δ denotes the maximum clock skew. The implications are explained with the following example. Consider that, in a hypothetical case, two cores release two packets with the identical deadline D, one at the absolute time $t + D$ and the other slightly later at $t + D + \varepsilon$, where $\varepsilon < \Delta$. Due to the clock skew, the latter core may still tag its packet with the deadline value which is less than the one with which the former cores tagged its packet. Consequently, if these packets contend, the latter will win the arbitration, despite the fact that it was indeed released ε time units after the former. This effect has to be considered in the worst-case analysis.

2.3.4.2 Worst-Case Analysis

As already described, the worst-case traversal time of a flow f_i consists of two components, namely the isolation delay C_i and the interference I_i. Notice, that the first term is independent of the

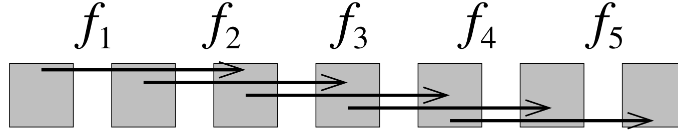


Figure 2.25: Traffic flows (example 4)

arbitration policy. So, in order to be able to test the schedulability of the flow-set upon a NoC with the EDF arbitration policy, it is necessary to obtain the interference component.

Consider the example illustrated in Figure 2.25 with the flow characteristics given in Table 2.5. Assume that the analysis for an EDF-scheduled single-core system with constrained or implicit deadlines has been straightforwardly applied to this example. For such a model, both the necessary and sufficient condition for schedulability is that the total system utilisation does not exceed the value of one [58], i.e. $U \leq 1$. In previous sections it was mentioned that the worst-case analysis for priority-preemptive NoCs is performed in such a way that a path of a flow under analysis is treated as an indivisible resource, and any request for any of its parts by other flows is considered as the potential interference. This implies that, in this example, it has to be checked whether the utilisation of the path of each flow, treated as a single-core system, fulfils the schedulability condition, i.e. $U(f_x) \leq 1, \forall f_x \in \{f_1, f_2, f_3, f_4, f_5\}$.

Table 2.5: Flow-set parameters for Figure 2.25

Flow	***C(f)***	***J_R(f)***	***D(f) = T(f)***
f_1	3	0	9.98
f_2	3	0	9.99
f_3	1	0	7
f_4	6.02	0	11.01
f_5	3	0	10

The computation renders the following values:

$$U(f_1) = \frac{C(f_1) + J_R(f_1)}{T(f_1)} + \frac{C(f_2) + J_R(f_2)}{T(f_2)} \approx 0.6$$

$$U(f_2) = \frac{C(f_1) + J_R(f_1)}{T(f_1)} + \frac{C(f_2) + J_R(f_2)}{T(f_2)} + \frac{C(f_3) + J_R(f_3)}{T(f_3)} \approx 0.74$$

$$U(f_3) = \frac{C(f_2) + J_R(f_2)}{T(f_2)} + \frac{C(f_3) + J_R(f_3)}{T(f_3)} + \frac{C(f_4) + J_R(f_4)}{T(f_4)} \approx 0.99$$

$$U(f_4) = \frac{C(f_3) + J_R(f_3)}{T(f_3)} + \frac{C(f_4) + J_R(f_4)}{T(f_4)} + \frac{C(f_5) + J_R(f_5)}{T(f_5)} \approx 0.99$$

$$U(f_5) = \frac{C(f_4) + J_R(f_4)}{T(f_4)} + \frac{C(f_5) + J_R(f_5)}{T(f_5)} \approx 0.85$$

The results suggest that the flow-set is schedulable. However, Figure 2.26 demonstrates that

missed deadlines may occur! The explanation is identical to that of the fixed-priority example (Section 2.3.1 and Figures 2.16-2.17), indirect interferences cause network jitters, which were not considered in the analysis and yet have an impact on the schedulability. In this particular example, f_2 indirectly interferes with f_4 in a sense that it causes the jitter to f_3, which eventually leads to a missed deadline of f_4. Thus, in the presence of network jitters, having the utilisation of all paths less than or equal to 1 is not a sufficient condition for a flow-set to be schedulable. Therefore, in order to test the schedulability of a flow-set, network jitters must be included in the analysis.

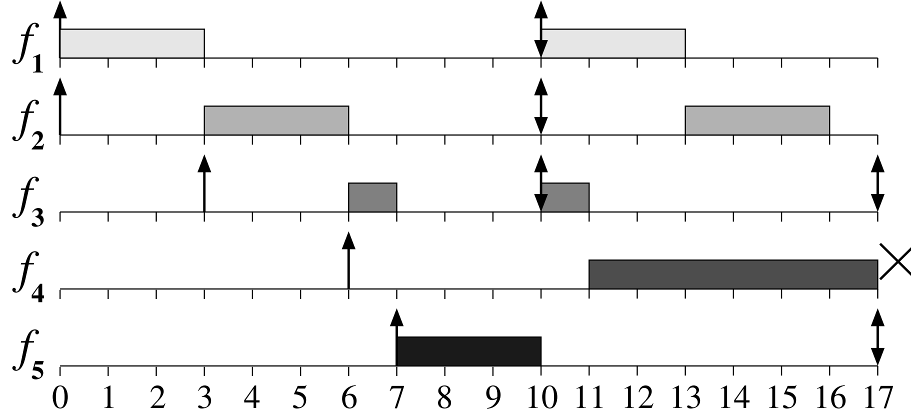


Figure 2.26: Deadline miss for flow-set from Figure 2.25 and Table 2.5

In the scheduling theory, the model that bears the closest resemblance to this model is the one that involves EDF-scheduled single-core systems with release jitters. For that model Spuri [90] proposed the worst-case analysis. He defined a *busy period*, which represents the time interval of maximal length with a continuous execution demand, assuming that the first jobs of all tasks were released with the maximum jitters. Subsequently, he proved that if no missed deadlines occur within the busy period, the system is schedulable.

Recall, that in the worst-case analysis of priority-preemptive NoCs, a path of each flow is treated as a single-core system. Thus, unlike in the scheduling theory where only one busy period is computed, here one busy period for each flow (path) has to be computed. For now, assume that the jitters are known, and later it will be explained how to compute them. The length of the busy period $W(f_i)$ (Equation 2.9) is obtained by summing up the maximum load that can be generated by the flow under analysis f_i, and the maximum load that can be generated by all the other flows that are directly contending with f_i (share a part of the path with it).

$$W(f_i) = \left\lceil \frac{W(f_i) + J_R(f_i) + J_N(f_i)}{T(f_i)} \right\rceil \cdot C(f_i) + \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \left\lceil \frac{W(f_i) + J_R(f_j) + J_N(f_j)}{T(f_j)} \right\rceil \cdot C(f_j) \quad (2.9)$$

Once the busy period has been computed, it has to be checked whether the analysed flow f_i can miss a deadline during that period. A set of time instants, for which it has to be checked, are those where the deadlines of f_i and at least one of the potentially interfering flows coincide, i.e.

$\mathscr{T}_{crit}(f_i) = \bigcup_{\forall f_j \in \mathscr{F}_D((f_i)} \{k * T(f_j) - T(f_i), k \in \mathbb{N}_0\} \cap [0, W(f_i)]$, hereafter referred to as the *critical instants*. Assuming that a packet is released at the critical instant $t \in \mathscr{T}_{crit}(f_i)$, its worst-case traversal time $L(f_i,t)$ (expressed in absolute values) is computed by solving Equation 2.10.

$$L(f_i,t) = \left(1 + \left\lfloor \frac{t + J_R(f_i) + J_N(f_i)}{T(f_i)} \right\rfloor\right) \cdot C(f_i) +$$

$$\sum_{\substack{\forall f_j \in \mathscr{F}_D(f_i) \\ T(f_j) \le t + T(f_i) + J_R(f_j) + J_N(f_j) + \Delta}} \min \left\{ \left\lceil \frac{L(f_i,t) + J_R(f_j) + J_N(f_j)}{T(f_j)} \right\rceil, \left\lfloor \frac{t + T(f_i) + J_R(f_j) + J_N(f_j) + \Delta}{T(f_j)} \right\rfloor \right\} \cdot C(f_j) \tag{2.10}$$

Equation 2.10 consists of the sum of the latencies of all packets of f_i, which were released in the interval $[0 - L(f_i,t)]$, augmented by the interference that the packets of f_i may suffer from the packets of other flows. Individual terms are obtained by finding the smaller between (i) the maximum number of releases of an interfering flow within the interval $[0 - L(f_i,t)]$, and (ii) the maximum number of those releases which deadline falls before, or coincides with $L(f_i,t)$. Note, Equation 2.10 is similar to the one Spuri [90] proposed for an EDF-scheduled single-core systems with release jitters. The differences are that this approach considers the maximum clock skew (parameter Δ), two types of jitters and constrained or implicit deadlines.

The worst-case traversal time of a packet of the flow f_i, released at the critical instant t, denoted by $WCTT(f_i,t)$, is computed by subtracting its release t from the obtained value $L(f_i,t)$. The additional remark is that the result cannot be less than $C(f_i)$ (Equation 2.11).

$$WCTT(f_i,t) = \max\{C(f_i), L(f_i,t) - t\} \tag{2.11}$$

Finally, upon obtaining the traversal times for all the critical instants, the worst-case traversal time of a flow f_i, denoted by $WCTT(f_i)$, can be computed by finding the maximum for all critical instants (Equation 2.12). Of course, both the necessary and sufficient schedulability condition is $WCTT(f_i) \le D(f_i)$.

$$WCTT(f_i) = \max\{WCTT(f_i,t), \forall t \in \mathscr{T}_{crit}(f_i)\} \tag{2.12}$$

2.3.4.3 Network Jitter Computation

The analysis proposed in the previous section is applicable under the assumption that network jitters of all flows are known in advance. Recall (Equation 2.3), network jitters are the way to model the effects of indirect interferences, and have non-zero values only for those directly competing flows which are involved in the indirect interference relationship (from the perspective of the analysed flow). Thus, it is trivial to see that the network jitter of the analysed flow is always zero, i.e. in Equations 2.9-2.10 $J_N(f_i) = 0$.

Remember that in the analysis for the fixed-priority model (Section 2.3.1), it is safely assumed that each contending flow that is involved in the indirect interference relationship (from

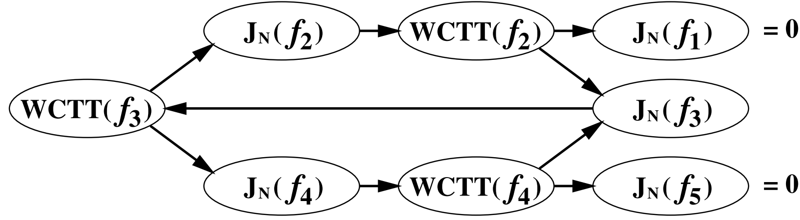


Figure 2.27: Chain of dependencies for Figure 2.25

the perspective of the analysed flow) experiences the maximum possible jitter, while still being schedulable, i.e. $J_N(f_i) = WCTT(f_i) - C(f_i)$. That approach was straightforwardly applicable, because in the fixed-priority model the commutative property applied to preemptions does not hold. In other words, for any two contending flows f_i and f_j it holds that, if $P(f_i) > P(f_j)$, then f_i can preempt f_j, but f_j cannot preempt f_i. Thus, the existence of f_j has no effect on the analysis of f_i and the worst-case traversal time and the jitter can be computed first for f_i and then for f_j. In fact, by sorting the flows decreasingly by their priorities, and by performing the analysis in that order, the worst-case traversal times and jitters of all flows can be obtained in a single pass.

Conversely, in the model with the EDF arbitration policy, the commutative property applied to preemptions may hold, that is, two contending flows f_i and f_j may preempt each other. This infers that if the aforementioned approach to compute jitters ($J_N(f_i) = WCTT(f_i) - C(f_i)$) is applied to this model, circular dependencies may occur. This is demonstrated with an illustrative example. Consider the flow-set from Figure 2.25. Moreover, assume that the objective is to compute the worst-case traversal time of the flow f_3. It is visible that $WCTT(f_3)$ depends on both $J_N(f_2)$ and $J_N(f_4)$, which both have non-zero values due to the existence of f_1 and f_5, respectively. Both jitters $J_N(f_2)$ and $J_N(f_4)$ depend on the respective worst-case traversal times $WCTT(f_2)$ and $WCTT(f_4)$. Furthermore, $WCTT(f_2)$ depends on $J_N(f_1)$ and $J_N(f_3)$. $J_N(f_1)$ has a value of zero, because f_1 does not have directly competing flows which may cause indirect interference to f_2. However, $J_N(f_3)$ has a non-zero value, due to the existence of f_4. $J_N(f_3)$ depends on $WCTT(f_3)$. Notice, that the computation of $WCTT(f_3)$ is reached for the second time, which infers that a circular dependency has been encountered. And indeed, by constructing the chain of dependencies for the given example (Figure 2.27), it can be noticed that

$$WCTT(f_3) \rightarrow J_N(f_2) \rightarrow WCTT(f_2) \rightarrow J_N(f_3) \rightarrow WCTT(f_3)$$
and
$$WCTT(f_3) \rightarrow J_N(f_4) \rightarrow WCTT(f_4) \rightarrow J_N(f_3) \rightarrow WCTT(f_3)$$

are circles of dependencies.

As discussed above, the commutative property with respect to preemptions may hold for EDF-arbitrated NoCs, and may cause circular dependencies. Therefore, jitters and the worst-case traversal times cannot be straightforwardly computed like in the fixed-priority model. One way to solve this problem in a nested iterative approach, where inner iterations correspond to individual flows,

Algorithm 7 *isSchedulable*($\mathscr{F}$)

Input: flow-set $\mathscr{F}$
Output: the information whether $\mathscr{F}$ is schedulable

```
1:  for each (f_i ∈ F) do
2:     WCTT(f_i) ← C(f_i); // New WCTT
3:     WCTT_old(f_i) ← 0; // Old WCTT
4:  end for
5:  while (∃f_i ∈ F | WCTT_old(f_i) ≠ WCTT(f_i)) do
6:     for each (f_i ∈ F) do
7:        WCTT_old(f_i) ← WCTT(f_i);
8:        WCTT(f_i) ← compWCTT(f_i, F);
9:        if (WCTT(f_i) > D(f_i)) then
10:          return false;
11:       end if
12:    end for
13: end while
14: return true
```

similar to the fixed-priority model, while outer iterations correspond to the entire flow-set. Algorithms 7-8 describe how to compute jitters and the worst-case traversal times, in an interleaved fashion. The computation process starts by invoking the function described with Algorithm 7. For each flow f_i of the flow-set the previously computed value of the worst-case traversal time $WCTT_{old}(f_i)$ and the newly computed value $WCTT(f_i)$ are kept. Initially, the computation starts by setting the worst-case traversal time of each flow to its isolation latency (line 2). Then, for each flow, the function described with the Algorithm 8 is invoked, which computes the worst-case traversal time of a flow (line 8). While there exists at least one flow for which the new and the old value of the worst-case traversal times are different, the process is repeated (line 5). Finally, when the worst-case traversal times from two successive iterations are equal for every flow, the computation process terminates. That is, the stopping condition is: $WCTT_{old}(f_i) = WCTT(f_i), \forall f_i \in \mathscr{F}$. If, at any stage, the computed worst-case traversal time of any flow exceeds its deadline, the entire flow-set is rendered unschedulable (lines $9-11$).

The computation of the worst-case traversal time of an individual flow is described with Algorithm 8. The algorithm has three stages. The first one generates a list $\mathscr{F}_D(f_i)$, which contains all flows that are directly contending with the analysed flow f_i (lines $1-5$). Then, for each flow $f_j \in \mathscr{F}_D(f_i)$, it is tested whether it has a contending flow f_k which may indirectly interfere with f_i. If at least one such flow exists, the network jitter $J_N(f_j)$ has a non-zero value and is computed by subtracting the isolation latency of f_j from its worst-case traversal time (lines $8-10$). Conversely, if there exists no f_k which can cause indirect interference to f_i through f_j, it follows that $J_N(f_j) = 0$ (lines $10-12$). Once the jitters of all contending flows are obtained, $WCTT(f_i)$ is computed from Equations 2.9-2.12 (lines $15-22$).

Algorithm 8 $compWCTT(f_i, \mathscr{F})$

Input: analysed flow f_i, flow-set $\mathscr{F}$
Output: the worst-case traversal time $WCTT(f_i)$

1: // 1. Find all flows contending with f_i
2: $\mathscr{F}_D(f_i) \leftarrow \emptyset$;
3: **for each** $(f_j \in \mathscr{F} \mid f_j \neq f_i \wedge \mathscr{L}(f_j) \cap \mathscr{L}(f_i) \neq \emptyset)$ **do**
4: add $(\mathscr{F}_D(f_i), f_j)$;
5: **end for**
6: // 2. Compute jitters of all contending flows
7: **for each** $(f_j \in \mathscr{F}_D(f_i))$ **do**
8: **if** $(\exists f_k \in \mathscr{F} \mid f_k \notin \mathscr{F}_D(f_i) \wedge \mathscr{L}(f_k) \cap \mathscr{L}(f_j) \neq \emptyset)$ **then**
9: $J_N(f_j) \leftarrow WCTT(f_j) - C(f_j)$;
10: **else**
11: $J_N(f_j) \leftarrow 0$;
12: **end if**
13: **end for**
14: // 3. Compute the WC traversal time of f_i
15: $W(f_i) \leftarrow compBusyInterval(f_i \cup \mathscr{F}_D(f_i))$; // Equation 2.9
16: $\mathscr{T}_{crit}(f_i) \leftarrow findCritPts(f_i \cup \mathscr{F}_D(f_i), W(f_i))$; // Section 2.3.4.2
17: $WCTT(f_i) \leftarrow C(f_i)$;
18: **for each** $(t \in \mathscr{T}_{crit}(f_i))$ **do**
19: $L(f_i, t) \leftarrow compAbsTime(f_i \cup \mathscr{F}_D(f_i), t)$; // Equation 2.10
20: $WCTT(f_i) \leftarrow max\{WCTT(f_i), L(f_i, t) - t\}$; // Equation 2.12
21: **end for**
22: **return** $WCTT(f_i)$;

2.3.4.4 Discussion

In the previous two sections, the method to compute the worst-case traversal times of flows was proposed. This method is applicable to the NoCs with the EDF arbitration policy. Consequently, if it holds that $WCTT(f_i) \leq D(f_i), \forall f_i \in \mathscr{F}$, then the entire flow-set $\mathscr{F}$ is schedulable.

Despite the fact that the worst-case analysis of NoCs is similar to the single-core scheduling theory, there are some differences as well, which have several interesting implications. For instance, Shi and Burns [84] showed that the rate-monotonic priority assignment technique is not optimal in the NoC context where flows have fixed priorities, while the opposite is the well-known fact in the single-core scheduling theory [58]. Similarly, it is well-known that EDF is the optimal scheduling policy for single-core systems [58], so it will be interesting to investigate if there exist cases in the NoC context where the fixed-priority arbitration policy outperforms EDF. The following two case studies further explore these ideas.

Case-study 1 (EDF outperforms FP)

Consider the example of only two contending flows, with the characteristics given in Table 2.6. Since only two flows are involved, there are no indirect interferences, i.e. both jitters $J_N(f_1)$ and $J_N(f_2)$ are zero. Consider the schedulability test for this flow-set, assuming that the priorities have been assigned in the rate-monotonic fashion: $T(f_1) < T(f_2) \Rightarrow P(f_1) > P(f_2)$.

Table 2.6: Flow-set parameters for two contending flows

Flow	C(f)	J_R(f)	D(f) = T(f)
f_1	5	0	10
f_2	6	0	15

$$WCTT(f_1) = C(f_1) = 5 < D(f_1)$$

$$WCTT(f_2) = C(f_2) + \left\lceil \frac{WCTT(f_2) + J_R(f_1) + J_N(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) = 16 > D(f_2)$$

The flow f_2 is unschedulable. Now consider again the schedulability test if the priorities are assigned differently: $P(f_2) > P(f_1)$.

$$WCTT(f_2) = C(f_2) = 6 < D(f_2)$$

$$WCTT(f_1) = C(f_1) + \left\lceil \frac{WCTT(f_1) + J_R(f_2) + J_N(f_2)}{T(f_2} \right\rceil \cdot C(f_2) = 11 > D(f_1)$$

In this case, the flow f_1 is unschedulable. Now, consider the schedulability test of this flow-set assuming the EDF arbitration policy. Since both network jitters $J_N(f_1)$ and $J_N(f_2)$ are zero, it is sufficient to only test if the utilisations of their paths $\mathscr{L}(f_1)$ and $\mathscr{L}(f_2)$ are less than one.

$$U(f_1) = U(f_2) = \frac{C(f_1)}{T(f_1)} + \frac{C(f_2)}{T(f_2)} = 0.9$$

As $U(f_1) = U(f_2) < 1$, the flow-set is schedulable.

Case-study 2 (FP outperforms EDF)

Consider the example of flows given in Figure 2.28, with the flow characteristics given in Table 2.7. By applying the rate-monotonic priority assignment policy it follows that $P(f_1) > P(f_2)$ and $P(f_3) > P(f_2)$, thus, there are again no indirect interferences and jitters. Consider the schedulability test for this example.

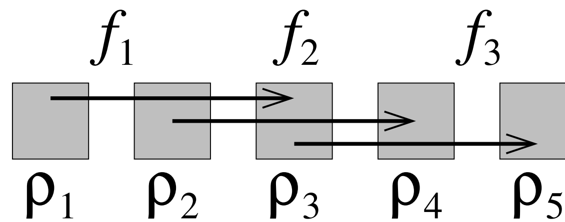


Figure 2.28: Traffic flows (example 5)

$$WCTT(f_1) = C(f_1) = 2 < D(f_1)$$

$$WCTT(f_3) = C(f_3) = 2 < D(f_3)$$

Table 2.7: Flow-set parameters for Figure 2.28

Flow	C(f)	J_R(f)	D(f) = T(f)
f_1	2	0	6
f_2	3	0	7
f_3	2	0	6

$$WCTT(f_2) = C(f_2) + \left\lceil \frac{WCTT(f_2) + J_R(f_1) + J_N(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) + $$
$$+ \left\lceil \frac{WCTT(f_2) + J_R(f_3) + J_N(f_3)}{T(f_3)} \right\rceil \cdot C(f_3) = 11 > D(f_2)$$

The flow f_2 is unschedulable. Now consider the schedulability test for this example when the priorities are assigned differently: $P_2 > P_1 \wedge P_2 > P_3$. It is easy to see that again indirect interferences do not exist, and hence jitters are equal to zero.

$$WCTT(f_2) = C(f_2) = 3 < D(f_2)$$
$$WCTT(f_1) = C(f_1) + \left\lceil \frac{WCTT(f_1) + J_R(f_2) + J_N(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) = 5 < D(f_1)$$
$$WCTT(f_3) = C(f_3) + \left\lceil \frac{WCTT(f_3) + J_R(f_2) + J_N(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) = 5 < D(f_3)$$

The flow-set is now schedulable. Finally, consider the schedulability test for the EDF arbitration policy. Notice, that in this case, indirect interferences do exist, i.e. when f_1 is under analysis, then f_3 can indirectly interfere, and vice versa. First, consider the busy period for the flow f_2.

$$W(f_2) = \left\lceil \frac{W(f_2) + J_R(f_2) + J_N(f_2)}{T(f_2)} \right\rceil \cdot C(f_2) +$$
$$\left\lceil \frac{W(f_2) + J_R(f_1) + J_N(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) + \left\lceil \frac{W(f_2) + J_R(f_3) + J_N(f_3)}{T(f_3)} \right\rceil \cdot C(f_3) \to \infty$$

As the busy period cannot be computed, the flow-set is unschedulable. The same conclusion can be reached by computing the utilisation of $\mathscr{L}(f_j)$, because $U(f_j) \approx 1.095$.

These two case studies have shown that there are scenarios where the EDF arbitration policy can schedule a flow-set, while no priority assignment exists which can cause the fixed-priority arbitration policy to do the same (the first row in Table 2.8). Similarly, it has been demonstrated that the opposite is true as well, and also the findings of Shi and Burns [84] regarding the existence of a priority-assignment that can schedule the flow-set which is unschedulable with priorities assigned in the rate-monotonic fashion have been confirmed (the second row in the Table 2.8). The case-studies for which the third and the fourth row of Table 2.8 are true are omitted, however, in Section 2.3.4.5 it will be demonstrated that such cases indeed exist. Finally, it is trivial to see that there exist flow-sets for which the fifth and the sixth row are true. This allows to identify the second major difference between the single-core scheduling theory and the worst-case analysis of

priority-preemptive NoCs. Although in the former EDF is proven to be optimal [58], in the latter a flow-set can be schedulable with the fixed-priority arbitration policy, but not with EDF (see Case-study 2).

Table 2.8: Comparison of approaches (schedulability)

		EDF	**Rate-monotonic**	**Exists priority assignment**
POSSIBLE SCENARIOS	1	✓	X	X
	2	X	X	✓
	3	✓	X	✓
	4	X	✓	✓
	5	✓	✓	✓
	6	X	X	X

2.3.4.5 Experimental Evaluation

Recall, in this dissertation, it is assumed that cores access local routers via a core link and a core port. However, in many cases, the manufacturers identify these elements as bottlenecks, and in order to improve the performance, allow cores to directly access the ports of the local router. This is equivalent to eliminating the first and the last link on the path of each flow. The benefit of this approach is that two flows that are originating from the same core, and going into opposite directions, do not have a common link any more, and thus do not contend with each other. This further implies that the flow between two adjacent cores does not traverse three links, but only one.

In this section, it will be assumed that cores can directly access the ports in their respective routers. The reason is that this approach allows to study flow-sets where flow paths consist of a single hop, and in such scenarios the EDF arbitration policy displays some interesting properties, as will be covered later in this section. The evaluation is performed by comparing the proposed analysis for EDF-arbitrated NoCs, in the further text referred to as the *EDF method*, against the two existing state-of-the-art approaches for FP-arbitrated NoCs [84, 85]. In the former, the priorities are assigned in the rate-monotonic fashion. In the latter, the heuristics-based search algorithm (HSA) is used to find a priority ordering, if one exists, such that the flow-set is schedulable. These methods are referred to as the *RM method* and the *HSA method*, respectively. Although HSA is based on the heuristics, it has one limitation: if it is unable to find a priority ordering with which the flow-set is schedulable, it will exhaustively enumerate all possible priority orderings. This infers that HSA has a factorial computational complexity (i.e. for flow-sets with $|\mathscr{F}|$ flows there exist $|\mathscr{F}|!$ different priority orderings), which further implies that HSA can be inapplicable in scenarios where flow-sets consist of 50 or more flows. Thus, in this section, for HSA is imposed the limit on the maximum number of orderings that can be evaluated. Specifically, for the flow-set of $|\mathscr{F}|$ flows, HSA is allowed to attempt at most $5 \cdot |\mathscr{F}|$ different priority orderings. If HSA fails to find an ordering with which the flow-set is schedulable, the process terminates.

Evaluation Metrics and Parameters

The comparison of the approaches is performed through the *sensitivity analysis* with respect to flow sizes. Specifically, if a flow-set is unschedulable with initial flow sizes, then the sizes of all flows are uniformly decreased until the flow-set becomes schedulable. Similarly, if a flow-set is schedulable with initial sizes, then the sizes of all flows are uniformly increased until the flow-set becomes unschedulable. The maximum flow sizes for which one method can guarantee the schedulability of a flow-set is called the *schedulability threshold* (ST). Of course, a higher ST infers that the method is more efficient. Upon obtaining the STs for the same flow-set with all the approaches, the comparison is performed. Let ST_{EDF}, ST_{RM} and ST_{HSA} be the STs obtained for the EDF method and with the two state-of-the-art methods. The improvements of the proposed approach over the existing ones are measured in the following way: $Imp_{EDF/RM} = \frac{ST_{EDF} - ST_{RM}}{ST_{RM}}$, and $Imp_{EDF/HSA} = \frac{ST_{EDF} - ST_{HSA}}{ST_{HSA}}$.

The flow-set and analysis parameters are summarised in Table 2.9, where an asterisk sign denotes a randomly generated value assuming a uniform distribution.

Table 2.9: Analysis parameters for Section 2.3.4.5

NoC topology and size	**2-D mesh with 8 $\times$ 8 routers**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Link width = flit size σ_{flit}	**16 bytes**
Flow packet size $\sigma(f_i), \forall f_i \in \mathscr{F}$	**$[1-128]^*$ kB**
Flow periods $D(f_i) = T(f_i), \forall f_i \in \mathscr{F}$	**$[20-100]^*$ μs**
Testing platform	**Intel dual-core desktop & Java (Max heap-size: 4 GB)**

Experiment 1: Overall improvements

In this experiment, the improvement trends related to flow path lengths are observed. This is done by imposing a constraint that the maximum path length (expressed in hops), of any flow of the flow-set, cannot exceed the value of the newly introduced parameter LIM, i.e. $|\mathscr{L}(f_i)| \leq LIM, \forall f_i \in \mathscr{F}$. The parameter LIM is varied in the range $[1-14]$. $LIM = 1$ means that only single-hop paths are allowed. Conversely, $LIM = 14$ allows all possible paths, because, assuming the XY routing, the maximum path length on a 8×8 platform is 14 hops long. For each value of the parameter LIM, 1000 flow-sets are randomly generated, each consisting of 200 flows. For each flow, the initial size and the period are randomly generated, while the source and the destination router were generated in a way that the path length does not exceed the imposed limit LIM. Subsequently, ST_{EDF}, ST_{RM} and ST_{HSA} are computed for each flow-set, and the obtained values are compared.

Figure 2.29 shows the improvements of EDF over RM. It is visible that, for cases where $LIM = 1$, EDF dominates RM. The explanation is as follows. When $LIM = 1$ the transitivity property always holds, and consequently indirect interferences do not exist. This infers that the rules from the uniprocessor scheduling theory also hold in this context: (i) EDF is the optimal

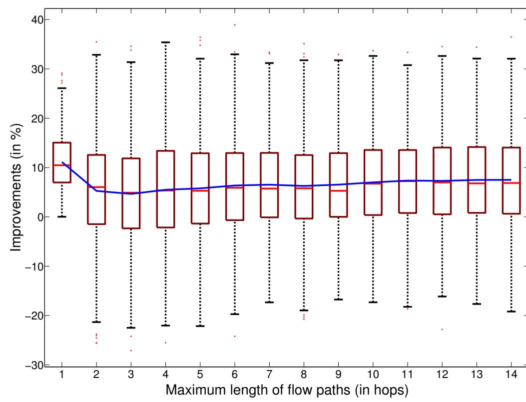


Figure 2.29: EDF vs RM

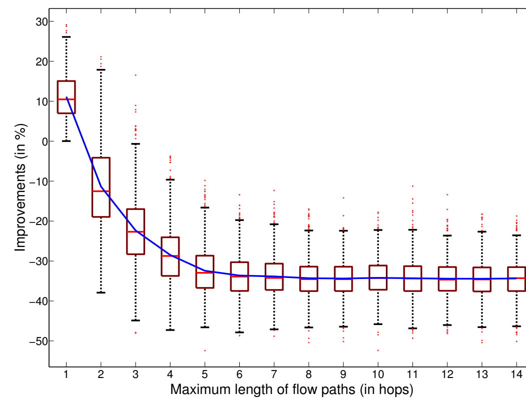


Figure 2.30: EDF vs HSA

policy and hence always outperforms RM, and (ii) RM is optimal among fixed-priority policies. On the rest of the domain ($LIM > 1$), the transitivity property may not hold, which implies that indirect interferences are possible. Thus, as exhibited in Case Studies 1-2, there exist scenarios where EDF outperforms RM, but the opposite is also true. Yet, the cases in which EDF performs better are more frequent, and on average EDF outperforms RM by 7%. As *LIM* grows, the average improvements also grow, however, the increase is barely noticeable.

Figure 2.30 illustrates the improvements of EDF over HSA. It is visible, that for $LIM = 1$, EDF also dominates HSA, in fact, the improvements are the same as in the example with RM. This is expected, because, as stated in the previous paragraph, RM is optimal among fixed-priority policies, and hence it should be HSA = RM. Thus, it can be concluded that EDF presents a very promising approach for systems where flows mostly traverse single-hop distances. On the rest of the domain ($LIM > 1$) HSA significantly outperforms EDF, and as the lengths of flow paths increase, the dominance of HSA becomes more apparent. This is a very counter-intuitive finding and the explanation is as follows. In EDF the commutative property holds, and in order a flow-set to be schedulable, the necessary condition is that the utilisation of the path of each flow is less than or equal to 1. Conversely, in the fixed-priority scheme, the commutative property does not hold, and the flow can be schedulable even though the utilisation of its path is greater than 1 (see Case Study 2). In fact, it can be concluded that, when $LIM > 1$, almost always there exists a priority ordering that can schedule a flow-set which is unschedulable with EDF. This is a crucial finding which suggests that EDF (or any other arbitration policy with dynamically changing flow priorities) may not be the most efficient arbitration technique for flow-sets with arbitrary path lengths. Yet, before this can be discussed, another important question has to be answered: how hard it is to find a priority ordering which will succeed in cases where EDF fails? This is covered in Experiment 2.

Assuming that all flows traverse single-hop distances, Figure 2.31 illustrates the improvements of EDF over the optimal fixed-priority scheme (i.e. RM). The same data as in Figures 2.29-2.30 was used, but the focus is on $LIM = 1$ only . It is visible that the average improvements are around 10%, while in some cases can reach up to 30%.

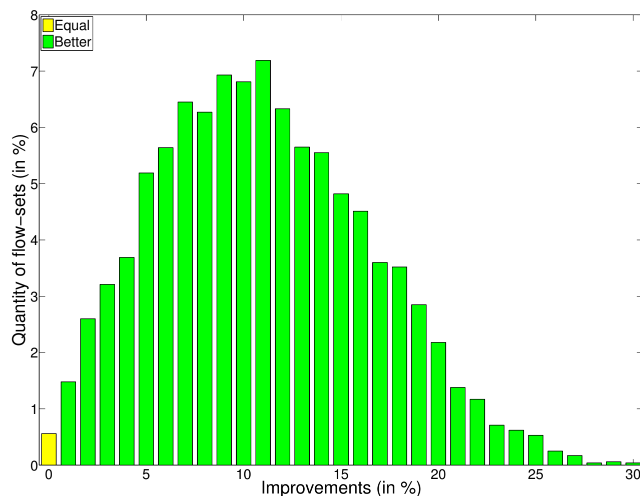


Figure 2.31: EDF vs RM,HSA (1-hop)

Experiment 2: Scalability

In this experiment, the objective is to observe how the analysis duration time changes as a function of the flow-set size. In other words, the aim is to investigate how scalable EDF, RM and HSA are. The number of flows constituting the flow-set is varied in the range $[100-400]$, with an incremental step of 50 flows. For each flow-set size 1000 flow-sets are randomly generated. For each flow the initial size, the period, the source and the destination routers are randomly generated, without the maximum path length constraint, i.e. $LIM = 14$. For each flow-set, the time it takes to compute ST_{EDF}, ST_{RM} and ST_{HSA} is measured. Subsequently, the obtained values are compared.

Figure 2.32 demonstrates that the computational complexity of EDF is higher than that of RM. This is expected, because in RM the commutative property does not hold, and hence the worst-case traversal times and the jitters can be obtained in a single pass, if flows are ordered by their priorities, decreasingly. Conversely, in EDF the commutative property holds, and hence several passes are needed until the worst-case traversal times and jitters stabilise for two successive passes (see Section 2.3.4.3 and Algorithms 7-8).

In spite of imposing the limit on the maximum number of orderings in HSA to only $5 \cdot |\mathscr{F}|$, its computational complexity significantly surpasses that of EDF and RM, and the duration time of the analysis grows exponentially. This occurs because HSA exhaustively enumerates the maximum allowed number of orderings before eventually rendering the flow-set unschedulable, while RM and EDF reach that conclusion much faster. These findings imply that HSA is the least scalable of the compared approaches, and that searching for ST_{HSA} may be prohibitively expensive for flow-sets with more than 500 flows. Therefore, it can be concluded that, for massive flow-sets, RM and EDF are preferable options.

Experiment 3: Applicability

The common underlying assumption of the previous experiments is that the clock skew does not exist, i.e. $\Delta = 0$ in Equation 2.10. In this experiment, the objective is to quantify the sensitivity

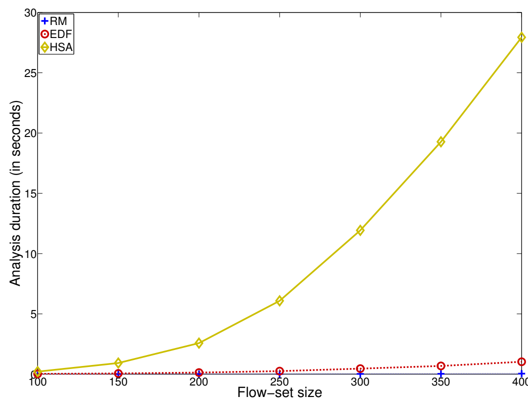


Figure 2.32: Analysis time variations

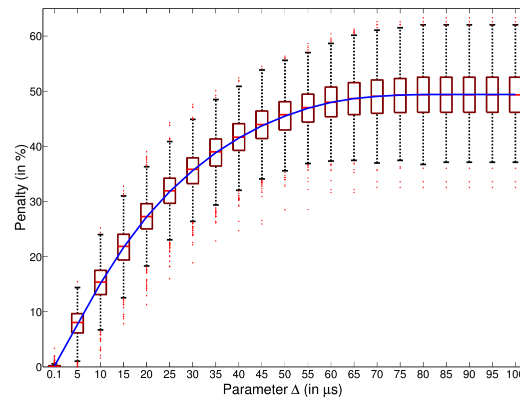


Figure 2.33: Influence of Δ on EDF

of the EDF method with respect to the clock skew. Based on the findings, it will be possible to derive some conclusions regarding the practical limitations of the EDF arbitration policy. The experiment is conducted in the following way. First, through the sensitivity analysis, ST_{EDF} of one flow-set is obtained, when the clock skew is equal to zero. That value serves as a baseline for comparisons. Then, for a non-zero value of the clock skew the sensitivity analysis is performed again, and the value $ST_{EDF'}$ if obtained for the same flow-set. These values are then compared and the penalty suffered due to the clock skew is expressed with the following metric: $penalty = \frac{ST_{EDF}-ST_{EDF'}}{ST_{EDF}}$. The process is repeated for 1000 flow-sets with the fixed flow-set size $|\mathscr{F}| = 200$ and $LIM = 14$. Subsequently, the clock skew value is changed ($\Delta \in [0.1 - 100\mu s]$), and the penalty is computed again.

The results are depicted in Figure 2.33. It is noticeable that for $\Delta \leq 0.1\mu s$ the effects are negligible. In the range $0.1\mu s < \Delta \leq 100\mu s$ the penalty grows logarithmically until reaching a saturation point at $\Delta = 80\mu s$. The conclusion is that the schedulability is very sensitive to the values of the clock skew that are within the range of flow periods, which is intuitive. It is also interesting that the saturation point exists. The explanation is that, even in the hypothetical case, with the infinite clock skew, the interference that one flow can cause to another ultimately has an upper-bound which is equal to the maximum number of releases of the interfering flow within the interval of interest (the first term in the min function of Equation 2.10). Regarding the most important question about the applicability of EDF, it can be concluded that the clock skew values that have an impact on the schedulability are several orders of magnitude greater than the ones in the real systems, inferring that EDF, counter-intuitively, does not face practical limitations.

2.3.4.6 Discussion

The experiments demonstrated that EDF dominates all fixed-priority schemes in cases where traffic flows traverse single-hop distances. In such cases the system inherits the properties of the uniprocessor scheduling theory, where EDF is optimal, and where system resources can be utilised in the most efficient way (even up to 100%). Can this finding be exploited on our quest towards efficient and real-time oriented multiprocessors? In order to provide an answer to this question,

further research in the area of application mapping is needed, because flow paths are inevitably dependant on the position of functionalities within the platform.

For longer flow paths, there are cases where EDF performs better than RM, but the opposite is also true. However, the number of cases where EDF performs better than RM are more frequent, and, on average, EDF outperforms RM by 7%. Moreover, HSA systematically outperforms EDF for all cases where path lengths exceed 3 hops. This finding suggests that, for flow-sets with arbitrary path lengths, EDF (or any other arbitration scheme with dynamically changing flow priorities) may not be the most efficient arbitration policy, which is a negative but important finding. However, the experiments also suggest that searching for the priority ordering that outperforms EDF indeed can be prohibitively expensive. Therefore, the applicability of EDF to a specific flow-set highly depends on the parameters of the flow-set.

Finally, the experiments demonstrated that EDF does not suffer practical limitations with respect to the clock skew. Specifically, the values of the clock skew, for which the analysis becomes sensitive, exceed the clock skews in the real systems by several orders of magnitude.

2.3.5 Reducing Analysis Pessimism

In Chapter 1, it was mentioned that the efficiency of the worst-case analysis highly depends on the amount of predictability of the analysed system, whereas any non-deterministic system behaviour has to be accounted for in the analysis with a certain degree of pessimism. A more pessimistic analysis may lead to a significant resource overprovisioning and/or underutilisation of platform resources. Conversely, a less pessimistic approach allows to save on design costs, e.g. by choosing a cheaper platform with fewer resources, which still guarantees the fulfilment of all timing constraints. Moreover, with the less pessimistic analysis, the resources of the platform can be exploited more efficiently, for example, by accommodating the additional workload, or by decreasing the power consumption via core shutdowns and smaller router frequencies. Thus, deriving the analysis with the least pessimism possible is of paramount importance in the real-time domain. In this section, one source of pessimism of the existing methods to perform the worst-case analysis of priority-preemptive NoCs is identified. Subsequently, the extension to the existing methods is proposed, which overcomes the aforementioned limitation, and allows for a less pessimistic worst-case analysis.

Irrespective of the arbitration policy, in previous methods the following assumption was used: the interference that a flow f_1 causes to a flow f_2 is equal to the isolation delay of f_1, i.e. $C(f_1)$ (as mentioned in Observation 4). However, this assumption may be pessimistic, as described below.

2.3.5.1 Motivational Example

Consider the example of two flows, f_1 and f_2, illustrated in Figure 2.34. Without loss of generality with respect to arbitration policies, consider that the packet of the flow f_1 can preempt the packet of f_2. Recall, that in the existing analyses, the entire traversal of f_1 would be considered as the interference to f_2 (Observation 4). Notice, that f_1 and f_2 share the common part of the path, which

consists of one link, and which is hereafter referred to as the *contention domain* – CD[2]. Let the path of f_1 be divided into 3 parts: (i) before f_1 and f_2 start sharing a common part of the path – **pre-CD**, (ii) while f_1 and f_2 share the common part of the path – **CD**, and (iii) after f_1 and f_2 stop sharing the common part of the path – **post-CD**.

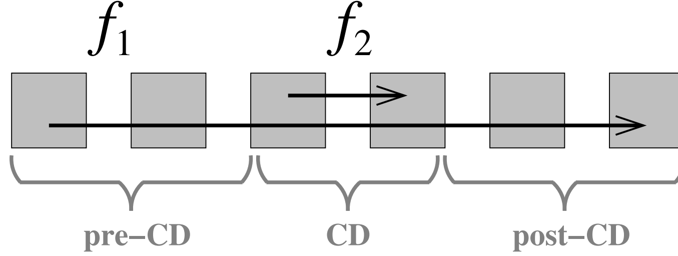


Figure 2.34: Traffic flows (example 6)

Notice that while the header flit of f_1 traverses pre-CD, f_1 does not cause any interference to f_2 (see Figure 2.35(a)). Thus, in the existing methods, the traversal of the header flit of f_1 through pre-CD is unnecessarily considered as the interference that f_1 causes to f_2. If that delay is excluded from the interference that f_1 causes to f_2, the analysis pessimism can be reduced, and consequently a tighter upper-bound estimate on the worst case traversal time of f_2 can be obtained.

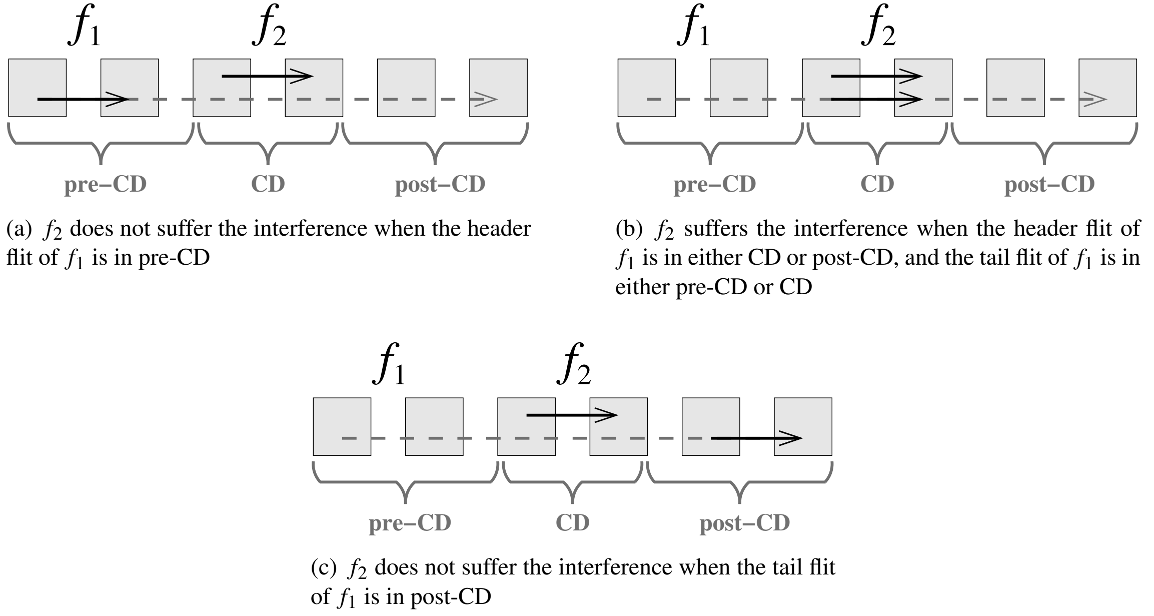
Once the header flit of f_1 reaches CD, f_2 starts suffering the interference (see Figure 2.35(b)). When the tail flit of f_1 leaves CD, f_2 stops suffering the interference from f_1, and may continue its progress (see Figure 2.35(c)). Thus, the traversal of the tail flit of f_1 through post-CD is also unnecessarily considered as the interference in the existing methods. By excluding this delay from the interference that f_1 causes to f_2, the analysis pessimism can be further reduced, and even a tighter upper-bound estimate on the worst-case traversal time of f_2 can be obtained.

2.3.5.2 Proposed Method

In this section, an extension to the existing methods for the worst-case analysis of priority-preemptive NoCs is proposed. This extension allows to obtain a less pessimistic estimate of the interference that directly interfering flows cause to the flow under analysis. Subsequently, tighter upper-bound estimates on the worst-case traversal times can be obtained.

Consider again the example illustrated in Figure 2.34. Let the parts of $\mathscr{L}(f_1)$ constituting sections pre-CD, CD and post-CD (with respect to flow f_2) be $\mathscr{L}_{1,2}^{pre-CD}$, $\mathscr{L}_{1,2}^{CD}$ and $\mathscr{L}_{1,2}^{post-CD}$, respectively. It is trivial to see that the union of these parts forms the entire path of f_1, i.e. $\mathscr{L}(f_1) = \mathscr{L}_{1,2}^{pre-CD} \cup \mathscr{L}_{1,2}^{CD} \cup \mathscr{L}_{1,2}^{post-CD}$. Now, as described in the previous section, the less pessimistic estimate of the interference that a single traversal of f_1 causes to f_2, denoted as $I(f_1 \to f_2)$ (Equation 2.13), can be computed by subtracting (i) $\gamma_{1,2}^{pre-CD}$, which is the time it takes a header

[2]The XY-routing mechanism assures that any two flows with a direct interference relationship have exactly one CD



(a) f_2 does not suffer the interference when the header flit of f_1 is in pre-CD

(b) f_2 suffers the interference when the header flit of f_1 is in either CD or post-CD, and the tail flit of f_1 is in either pre-CD or CD

(c) f_2 does not suffer the interference when the tail flit of f_1 is in post-CD

Figure 2.35: Detailed interference analysis

flit of f_1 to traverse pre-CD, and (ii) $\gamma_{1,2}^{post-CD}$, which is the time it takes a tail flit of f_1 to traverse post-CD, from the entire traversal time of f_1.

$$I(f_1 \rightarrow f_2) = C(f_1) - \gamma_{1,2}^{pre-CD} - \gamma_{1,2}^{post-CD} \tag{2.13}$$

where $\gamma_{1,2}^{pre-CD}$ and $\gamma_{1,2}^{post-CD}$ are computed as follows:

$$\gamma_{1,2}^{pre-CD} = |\mathscr{L}_{1,2}^{pre-CD}| \cdot \delta_L + max\left\{0, \left(|\mathscr{L}_{1,2}^{pre-CD}| - 1\right)\right\} \cdot \delta_\rho \tag{2.14}$$

$$\gamma_{1,2}^{post-CD} = |\mathscr{L}_{1,2}^{post-CD}| \cdot \delta_L \tag{2.15}$$

If this reasoning is applied for all directly interfering flows of a flow under analysis f_i, a tighter worst-case traversal time $WCTT^+(f_i)$ can be obtained. Specifically, for the fixed-priority arbitration scheme, Equation 2.4 is substituted with Equation 2.16.

$$WCTT^+(f_i) = C(f_i) + \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \left\lceil \frac{WCTT^+(f_i) + J_R(f_j) + J_N(f_j)}{T(f_j)} \right\rceil \cdot I(f_j \rightarrow f_i) \tag{2.16}$$

Similarly, for EDF-arbitrated NoCs, Equations 2.9-2.12 are substituted with Equations 2.17-2.20.

$$W^+(f_i) = \left\lceil \frac{W(f_i)+J_R(f_i)+J_N(f_i)}{T(f_i)} \right\rceil \cdot C(f_i) + \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \left\lceil \frac{W(f_i)+J_R(f_j)+J_N(f_j)}{T(f_j)} \right\rceil \cdot I(f_j \to f_i) \tag{2.17}$$

$$L^+(f_i,t) = \left(1 + \left\lfloor \frac{t+J_R(f_i)+J_N(f_i)}{T(f_i)} \right\rfloor \right) \cdot C(f_i) + I(f_j \to f_i) \cdot$$

$$\sum_{\substack{\forall f_j \in \mathscr{F}_D(f_i) \\ T(f_j) \leq t+T(f_i)+J_R(f_j)+J_N(f_j)+\Delta}} \min \left\{ \left\lceil \frac{L(f_i,t)+J_R(f_j)+J_N(f_j)}{T(f_j)} \right\rceil, \left\lfloor \frac{t+T(f_i)+J_R(f_j)+J_N(f_j)+\Delta}{T(f_j)} \right\rfloor \right\} \tag{2.18}$$

$$WCTT^+(f_i,t) = \max\left\{C(f_i), L^+(f_i,t) - t\right\} \tag{2.19}$$

$$WCTT^+(f_i) = \max\{WCTT^+(f_i,t), \forall t \in \mathscr{T}_{crit}(f_i)\} \tag{2.20}$$

2.3.5.3 Observations

When comparing the existing and the new methods to compute the worst-case traversal times of individual flows, several interesting facts can be noticed:

Observation 5. *When compared with the existing ones, the newly proposed methods never perform worse. Indeed, the new methods always derive upper-bounds which are either the same, or less pessimistic. In other words, it always holds that $WCTT^+(f_i) \leq WCTT(f_i), \forall f_i \in \mathscr{F}$. Theorem 4 provides the proof.*

Theorem 4. *Irrespective of the arbitration policy, the worst-case traversal time, of any flow of the flow-set, computed with the new method, is either equal to, or less pessimistic than the one obtained with the existing method.*

Proof. Proven directly. Consider two flows f_i and f_j, where the packets of f_j can preempt the packets of f_i. Let $\gamma_{j,i}$ be the difference in the interference caused by a single preemption of f_j to f_i, computed with the existing and the proposed method (Equation 2.21).

$$\gamma_{j,i} = C_j - I(f_j \to f_i) = \gamma_{j,i}^{pre-CD} + \gamma_{j,i}^{post-CD} = |\mathscr{L}_{j,i}^{pre-CD}| \cdot \delta_L +$$

$$max\left\{0, \left(|\mathscr{L}_{j,i}^{pre-CD}| - 1\right)\right\} \cdot \delta_\rho + |\mathscr{L}_{j,i}^{post-CD}| \cdot \delta_L \tag{2.21}$$

Since all the terms of Equation 2.21 are non-negative values, it follows that $\gamma_{j,i} \geq 0$. Let $K_{j,i}$ be the number of preemptions that f_j can cause to f_i during the worst-case traversal time of f_i. Also,

let γ_i be the difference in the total interference caused to f_i by all its directly interfering flows, computed with the existing and the proposed method (Equation 2.22).

$$\gamma_i = \sum_{\forall f_j \in \mathscr{F}_D(f_i)} \gamma_{j,i} \cdot K_{j,i} \tag{2.22}$$

Since all the terms of Equation 2.22 have non-negative values, it follows that $\gamma_i \geq 0$. Moreover, as the existing and the new method differ only in the way how the interference is computed, it follows that $WCTT(f_i) - WCTT^+(f_i) = \gamma_i \geq 0$. □

Theorem 5 provides a proof that the obtained upper-bounds are safe.

Theorem 5. *The traversal time of any packet belonging to the flow f_i can not be greater than $WCTT^+(f_i)$ (Equation 2.16 or Equation 2.20), even in the worst-case conditions.*

Proof. Proven directly. Depending on the arbitration policy, the worst-case traversal time of a flow can be computed either by solving Equation 2.4 for fixed-priority-arbitrated NoCs (Section 2.3.1), or Equation 2.12 for EDF-arbitrated NoCs (Section 2.3.4). If it can be proven that, irrespective of the arbitration policy, there exists some discontinuous time interval γ_i which is a part of $WCTT(f_i)$, and during which f_i does not progress, nor any of its interfering flows causes the interference to it, that will prove that $WCTT(f_i) - \gamma_i$ is also a safe upper-bound on the worst-case traversal time of f_i.

Consider two traffic flows f_i and f_j, where the packets of f_j can preempt the packets of f_i. According to Observation 4, the entire traversal of f_j is considered as the interference that f_j causes to f_i, and hence entirely contributes to $WCTT(f_i)$. Let $\gamma_{j,i}$ (Equation 2.21) be the sum of: (i) $\gamma_{j,i}^{pre-CD}$, which is the interval when the header flit of f_j traverses pre-CD and (ii) $\gamma_{j,i}^{post-CD}$, which is the interval when the tail flit of f_j traverses post-CD. By definition, both a necessary and sufficient condition for the contention between f_i and f_j is that both of them attempt to traverse the CD section at the same time. However, during $\gamma_{j,i}^{pre-CD}$ and $\gamma_{j,i}^{post-CD}$, the flow f_j does not traverse CD, hence f_i can safely progress. Thus, the maximum interference that one packet of f_j can cause to f_i has a safe upper-bound, which is $C(f_j) - \gamma_{j,i}$. Subsequently, if during the traversal of f_i, packets of f_j can appear at most $K_{j,i}$ times, then the safe upper-bound on the interference that f_j can cause to f_i is $K_{j,i} \cdot (C(f_j) - \gamma_{j,i})$. By elevating this reasoning, it can be concluded that the worst-case traversal time of f_i has a safe upper-bound $WCTT^+(f_i)$ (Equation 2.23).

$$WCTT^+(f_i) = WCTT(f_i) - \sum_{\forall f_j \in \mathscr{F}_D(f_i)} K_{j,i} \cdot \gamma_{j,i} = WCTT(f_i) - \gamma_i \tag{2.23}$$

□

Observe the improvements of the proposed method over the existing one on a small-scale example given in Figure 2.34, where the flow characteristics are given in Table 2.10, and the NoC characteristics are given in Table 2.12. Without the loss of generality in terms of arbitration policies, in this and subsequent examples the fixed-priority arbitration policy will be assumed,

however, note that any conclusion reached for this scheme will also hold for the EDF arbitration policy.

Table 2.10: Flow-set parameters for Figure 2.34 (example 1)

Flow	Priority	$\sigma(f)$	$J_R(f)$	$D(f) = T(f)$
f_1	$P(f_1)$	48 B	0	1 μs
f_2	$P(f_2) < P(f_1)$	48 B	0	1 μs

Since only two flows exist, there are no indirect interferences, thus network jitters are equal to zero, i.e. $J_N(f_1) = J_N(f_2) = 0$. f_1 is the higher-priority flow, so its worst-case traversal time will be the same with both methods:

$$WCTT(f_1) = WCTT^+(f_1) = C(f_1) = |\mathscr{L}(f_1)| \cdot \delta_L + (|\mathscr{L}(f_1)| - 1) \cdot \delta_p + \left\lceil \frac{\sigma(f_1)}{\sigma_{flit}} \right\rceil \cdot \delta_L = 14ns$$

However, the worst-case traversal time of f_2 is different. First, the value will be obtained with the existing analysis. For that, $C(f_2)$ is needed.

$$C(f_2) = |\mathscr{L}(f_2)| \cdot \delta_L + (|\mathscr{L}(f_2)| - 1) \cdot \delta_p + \left\lceil \frac{\sigma(f_2)}{\sigma_{flit}} \right\rceil \cdot \delta_L = 6ns$$

Now, $WCTT(f_2)$ can be computed as follows:

$$WCTT(f_2) = C(f_2) + \left\lceil \frac{WCTT(f_2) + J_R(f_1) + J_I(f_1)}{T(f_1)} \right\rceil \cdot C(f_1) = 20ns$$

Now, $WCTT^+(f_2)$ will be obtained. To do so, first $I(f_1 \rightarrow f_2)$ has to be computed. Recall, that $I(f_1 \rightarrow f_2)$ denotes the interference that f_1 causes to f_2 with a single preemption. The lengths of the relevant sections are: $|\mathscr{L}_{1,2}^{pre-CD}| = 3$, $|\mathscr{L}_{1,2}^{CD}| = 1$ and $|\mathscr{L}_{1,2}^{post-CD}| = 3$ (see Figure 2.34). Remember that the first and the last link of each flow are the core links, which have been omitted from Figure 2.34 for clarity purposes.

$$I(f_1 \rightarrow f_2) = C(f_1) - |\mathscr{L}_{1,2}^{pre-CD}| \cdot \delta_L - max\left\{0, \left(|\mathscr{L}_{1,2}^{pre-CD}| - 1\right)\right\} \cdot \delta_p - |\mathscr{L}_{1,2}^{post-CD}| \cdot \delta_L = 8ns$$

Now, $WCTT^+(f_2)$ can be obtained as follows:

$$WCTT^+(f_2) = C(f_2) + \left\lceil \frac{WCTT^+(f_2) + J_R(f_1) + J_N(f_1)}{T(f_1)} \right\rceil \cdot I(f_1 \rightarrow f_2) = 14ns$$

It is visible that the worst-case traversal time of f_2 obtained with the new approach is $14ns$, while it was $20ns$ with the existing method, which is an improvement of $6ns$, or in relative terms, an improvement of 30%.

Observation 6. *The improvements of the proposed approach over the existing one depend on the lengths of the pre-CD, CD and post-CD sections of interfering flows.*

Consider again the example from Figure 2.34, where $P(f_1) > P(f_2)$. Assuming that the path of f_1 is constant, i.e. $|\mathscr{L}(f_1)| = const$, from Equations 2.21-2.22 it straightforwardly follows that the improvements in the worst-case traversal time of f_2 are greater when both $|\mathscr{L}_{1,2}^{pre-CD}|$ and $|\mathscr{L}_{1,2}^{post-CD}|$ are bigger. Since, the interference relationship between f_1 and f_2 is possible if and only if a contention between these two flows exist, i.e. $|\mathscr{L}_{1,2}^{CD}| \geq 1$, it follows that the necessary condition for the maximum improvements is: $|\mathscr{L}_{1,2}^{pre-CD}| + |\mathscr{L}_{1,2}^{post-CD}| = |\mathscr{L}(f_1)| - 1$.

This is demonstrated with an illustrative example given in Figure 2.36. Consider the same flow characteristics as in the previous example (Table 2.10). If the worst-case traversal times of both flows are computed with both methods, the following results are produced: $WCTT(f_1) = WCTT^+(f_1) = 14ns$, $WCTT(f_2) = 24ns$ and $WCTT^+(f_2) = 20.5ns$. When compared with the previous example, it is visible that the improvements in the worst-case traversal time of f_2 dropped from $6ns$ to $3.5ns$, or expressed relatively, from 30% to less than 15%.

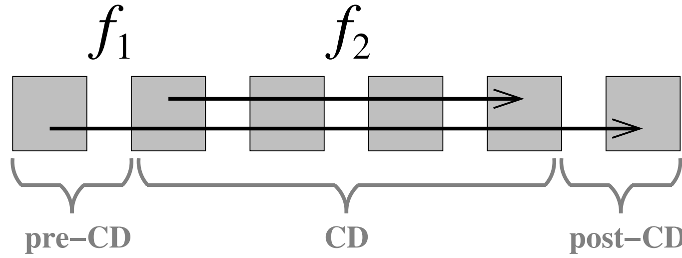


Figure 2.36: Traffic flows (example 7)

Note, a special case occurs when paths of interfering flows entirely overlap, i.e. $|\mathscr{L}_{1,2}^{pre-CD}| = \mathscr{L}_{1,2}^{post-CD}| = 0$, $|\mathscr{L}_{1,2}^{CD}| = |\mathscr{L}(f_1)|$. In such scenarios there are no improvements because both approaches return the same values.

Observation 7. *The improvements of the proposed approach over the existing one depend on the position of the CD section of interfering flows.*

Consider again the example given in Figure 2.34, where $P_1 > P_2$. Assuming that the path of f_1 and the length of the CD section are constant, i.e. $|\mathscr{L}(f_1)| = const, |\mathscr{L}_{1,2}^{CD}| = const$, from Equations 2.21-2.22 it straightforwardly follows that the improvements in the worst-case traversal time of f_2 are more influenced by the length of the pre-CD than the post-CD section. Thus, a necessary and sufficient condition for maximum improvements is: $|\mathscr{L}_{1,2}^{pre-CD}| = |\mathscr{L}(f_1)| - 1, |\mathscr{L}_{1,2}^{CD}| = 1$ and $|\mathscr{L}_{1,2}^{post-CD}| = 0$.

To demonstrate that, consider the example from Figure 2.37, which differs from the example from Observation 5 (Figure 2.34) only in the position of the CD section. If the worst-case traversal times for both flows are computed again, assuming the same flow characteristics (Table 2.10), the following values are produced: $WCTT(f_1) = WCTT^+(f_1) = 14ns$, $WCTT(f_2) = 20ns$ and $WCTT^+(f_2) = 12.5ns$. When compared with the example from Observation 5, it is visible that the improvements in the worst-case traversal time of f_2 increased from $6ns$ to $7.5ns$, or expressed relatively, from 30% to 37.5%.

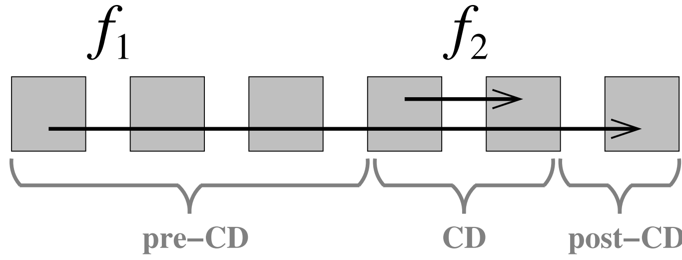


Figure 2.37: Traffic flows (example 8)

Observation 8. *The improvements of the proposed approach over the existing one do not depend on flow-sizes.*

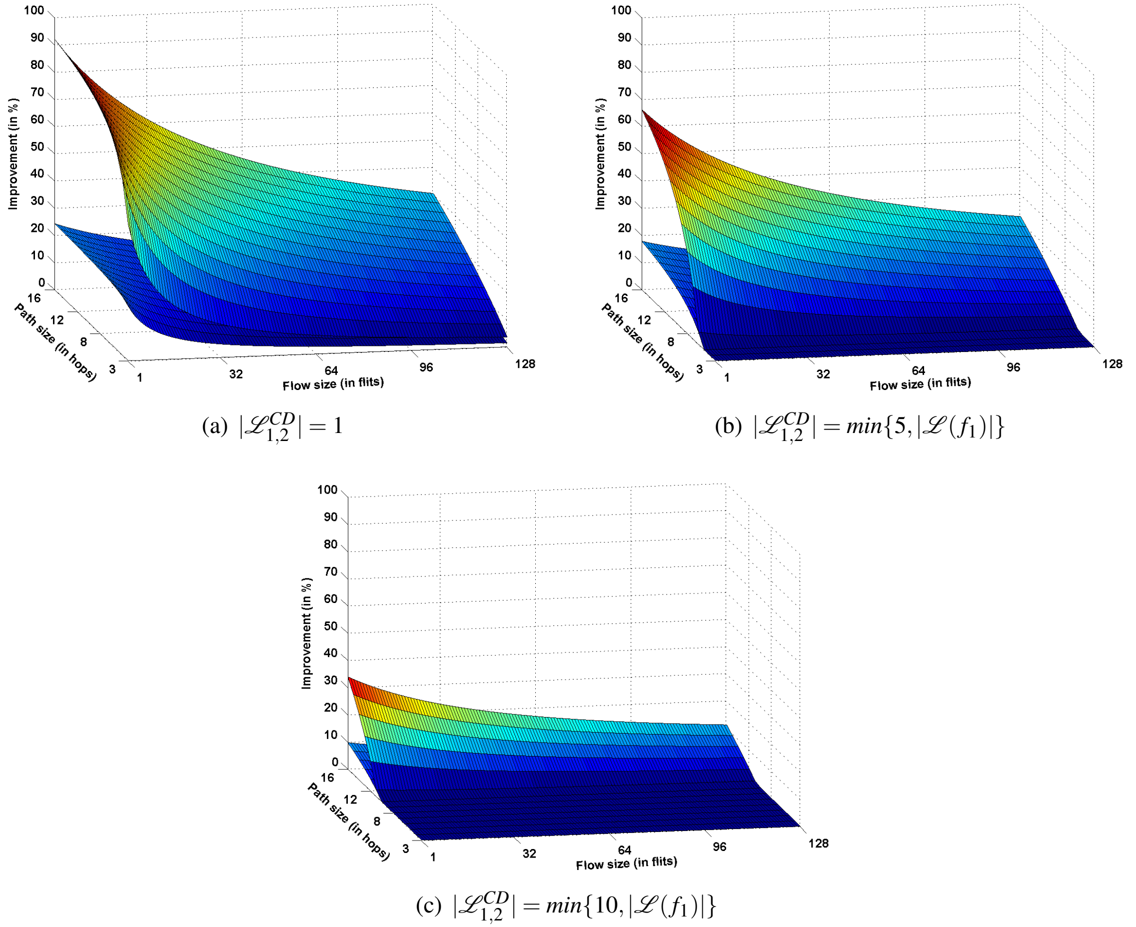
Indeed, from Equations 2.21-2.22 it is visible that only the paths, but not the sizes of the flows influence the improvements. Therefore, with the increase in flow sizes, the worst-case traversal times also grow, but the improvements remain the same in absolute values, and hence report decrease in relative values. If the worst-case traversal times are computed for the example given in Figure 2.34, but this time with bigger flow sizes (Table 2.11), the following results are produced: $WCTT(f_1) = WCTT^+(f_1) = 17.5ns$, $WCTT(f_2) = 27ns$ and $WCTT^+(f_2) = 21ns$. Like in the equivalent case for $\sigma(f_1) = \sigma(f_2) = 48B$, again $WCTT(f_2) - WCTT^+(f_2) = 6ns$, however, the relative improvements drop from 30% to 22.2%. This implies that, as flow sizes increase, the absolute improvements remain unaffected, however, the relative improvement decrease. In fact, in a hypothetical case with flows of infinitely large sizes, the relative improvements asymptotically converge towards 0%.

Table 2.11: Flow-set parameters for Figure 2.34 (example 2)

Flow	Priority	σ(f)	J_R(f)	D(f) = T(f)
f_1	$P(f_1)$	160 B	0	1 μs
f_2	$P(f_2) < P(f_1)$	160 B	0	1 μs

2.3.5.4 Numerical Example

In order to get a better insight into how different parameters influence analysis improvements, a small-scale numerical example is used. Consider two contending flows f_1 and f_2, where $P_1 > P_2$, $\mathscr{F}_D(f_2) = \{f_1\}$ and f_1 preempts f_2 only once. The length of the CD section, the sizes, and the paths of the flows are varied parameters. For each particular scenario the worst-case traversal times of f_2 are computed with both approaches. The objective is to observe the relative improvements, achieved by the proposed method. Figure 2.38 demonstrates the results. In each subfigure, a lower surface covers a corner case where the length of the pre-CD section is equal to zero, i.e. $|\mathscr{L}_{1,2}^{pre-CD}| = 0$ and $|\mathscr{L}_{1,2}^{post-CD}| = |\mathscr{L}(f_1)| - |\mathscr{L}_{1,2}^{CD}|$, while the upper surface represents the opposite corner case, i.e. the length of the post-CD section is equal to zero, i.e.



(a) $|\mathscr{L}_{1,2}^{CD}| = 1$

(b) $|\mathscr{L}_{1,2}^{CD}| = min\{5, |\mathscr{L}(f_1)|\}$

(c) $|\mathscr{L}_{1,2}^{CD}| = min\{10, |\mathscr{L}(f_1)|\}$

Figure 2.38: Improvement in the worst-case traversal time of f_2, for two contending flows f_1 and f_2, where $P(f_1) > P(f_2)$, $|\mathscr{L}(f_1)| = |\mathscr{L}(f_2)|$, and f_1 preempts f_2 only once

$|\mathscr{L}_{1,2}^{pre-CD}| = |\mathscr{L}(f_1)| - |\mathscr{L}_{1,2}^{CD}|$ and $|\mathscr{L}_{1,2}^{post-CD}| = 0$. The trends in Figure 2.38 entirely coincide with all the conclusions from Observations 5-8. Moreover, it is visible that the improvements are equal to zero in cases where the path of f_1 entirely belongs to the CD section, i.e. $|\mathscr{L}_{1,2}^{CD}| = |\mathscr{L}(f_1)|$, which has already been mentioned in Observation 6.

2.3.5.5 Experimental Evaluation

In the previous section, the improvements of the proposed analysis over the existing one were analysed on small-scale illustrative examples consisting of only two flows. In this section, a large-scale comprehensive experimental evaluation is performed, with the main objective to quantify the improvements of the new approach over the existing ones, but this time assuming flow-sets with hundreds of flows. This will help us to investigate whether the improvement trends from a small-scale example also hold for large flow-sets, and to what extent. Subsequently, scenarios (flow characteristics) can be identified, for which the proposed improvement reports the best results.

Analysis parameters are given in Table 2.12[3]. Note, that again the fixed-priority arbitration policy is assumed, however, any conclusions reached for this scheme also hold in the context of the EDF arbitration policy.

Table 2.12: Analysis parameters for Section 2.3.5.5

NoC topology and size	**2-D mesh with 8 $\times$ 8 routers**
Link width = flit size σ_{flit}	**16 bytes**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Flow periods $D(f_i) = T(f_i), \forall f_i \in \mathscr{F}$	$[\mathbf{10-100}]^*$ μ**s**

Experiment 1: Improvements wrt flow sizes

In this experiment, the objective is to investigate how different flow sizes influence the improvements of the proposed method. Several categories of flow-sets are generated, with different flow size ranges: $1B-16B, 16B-64B, ..., 64kB-256kB$. For each category 100 flow-sets are generated, where a flow-set consists of 200 flows. The size of each flow is randomly generated, within the limits of the respective flow-set category. Also, for each flow, the priority, the source core and the destination core are randomly generated, where flow paths comply with the XY routing policy. Subsequently, for each flow of the flow-set, the worst-case traversal time is computed with both methods, and the improvements are measured in relative terms.

Figure 2.39 shows the results. Since the improvements do not depend on flow sizes (see Observation 8), the absolute improvements of the new method are constant, irrespective of the flow sizes. However, as the increase in the flow size causes a uniform increase in the worst-case traversal times obtained with both methods, the relative improvements of the new approach (y-axis) decrease as the flow sizes increase (x-axis). Another interesting finding is that, irrespective of the flow sizes, there are always flows for which the proposed method does not yield better results. These are the highest-priority flows which do not suffer any interference, hence for them both methods return the same results.

Experiment 2: Improvements wrt paths sizes

In this experiment, the lengths of flow paths are varied, and subsequently their influence on the improvements are observed. Again, several categories of flow-sets are generated, but this time with different lengths of flow paths: $3-4, 3-6, ..., 3-16$. For each category 100 flow-sets are generated, where a flow-set consists of 200 flows. The priority, the source and destination cores are generated randomly for each flow, but in accordance with the constraint on the maximum path size, posed by the respective category to which the flow-set belongs. Each flow has a size which

[3] A period of each flow is randomly generated, within the given limits. If a generated flow-set is not schedulable, then periods of all flows are uniformly increased (even beyond the limits) until the flow-set becomes schedulable.

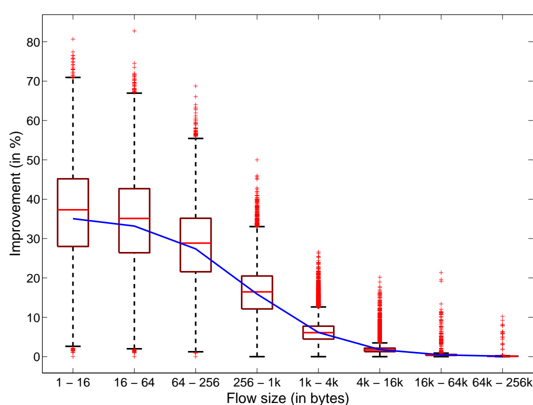


Figure 2.39: Improvements wrt flow sizes

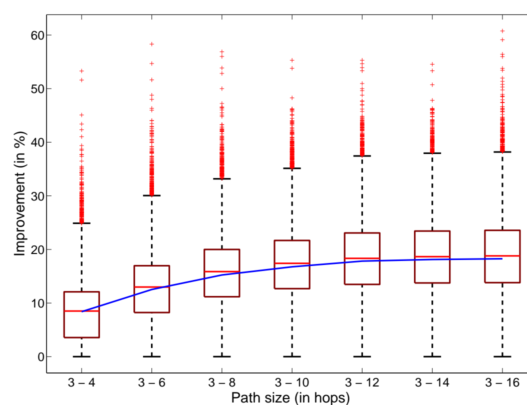


Figure 2.40: Improvements wrt path sizes

is randomly generated in the range $[1B - 1kB]$. The worst-case traversal times of all flows are computed with both methods, and the improvements are measured in relative terms.

Figure 2.40 demonstrates that as the paths of the flows increase (x-axis), the relative improvements also increase (y-axis). The explanation is that short paths substantially decrease the number of contentions and interferences, thus decreasing the scenarios in which the new approach can cause improvements. In fact, even in scenarios where contentions do occur, due to short flow paths, CD sections cover large fractions of them, which has a significant impact on the improvements (Observation 6). Conversely, longer paths cause more interferences, but also longer pre-CD and post-CD sections. All these facts have a positive effect on the improvements (see Observation 6).

Experiment 3: Improvements wrt flow and path sizes

The objective of this experiment is to get a better insight into how both the aforementioned flow characteristics (the flow size and the path size) together influence the improvements of the new method. Different flow-set categoties are generated, where both parameters are varied. Again, each category consists of 100 flow-sets, each with 200 flows with randomly generated priorities, source and destination cores. For each flow the worst-case traversal times are computed with both methods. Subsequently, for each category the average improvements achieved with the new method are computed and expressed in relative terms.

Figure 2.41 shows the improvement trends (z-axis) associated with flow sizes (x-axis) and path sizes (y-axis). The improvement trends are identical to those from Experiments 1-2, inferring that the increase in the flow sizes and the decrease in the path sizes both have a negative effect on the relative improvements. This infers that small flows with long paths benefit the most from the proposed approach. Note, that a similar conclusion was reached for a small-scale numerical example with two flows (see Section 2.3.5.4 and Figure 2.38).

Experiment 4: Improvements wrt flow-set sizes

The emphasis of this experiment is on the flow-set size. In other words, the objective is to investigate how the improvement trends change with the number of flows constituting a flow-set.

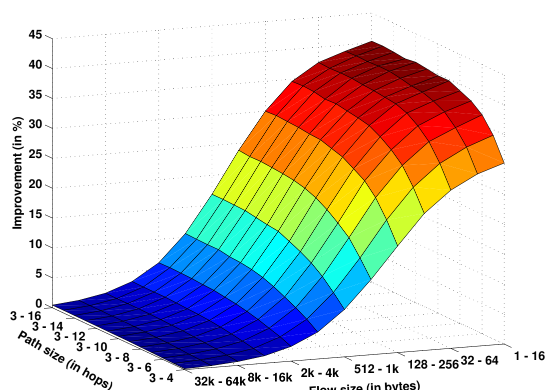


Figure 2.41: Improvements wrt flow-set sizes

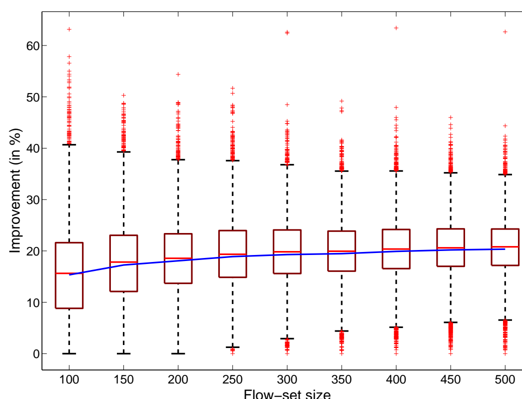


Figure 2.42: Improvements wrt flow and path sizes

Several flow-set categories are generated, where each category has the number of flows equal to one of these values: $100, 150, ..., 500$. For each category 100 flow-sets are generated, with random priorities, source and destination cores. Moreover, each flow has a size which is randomly generated in the range $[1B - 1kB]$. Subsequently, for each flow the worst-case traversal times are obtained with both methods, and then compared. The improvements are measured in relative terms.

Figure 2.42 demonstrates that as the flow-set size increases, so do the improvements. The explanation is that larger flow-sets have more substantial contentions, which favours the proposed approach. Yet, irrespective of the flow-set size, there always exist the highest-priority flows which do not suffer interference and hence for them no improvements can be achieved. However, as the flow-set size increases, the highest-priority flows constitute smaller and smaller fraction of the entire flow-set, hence for sets with more than 200 flows these cases are below the 25th percentile, and are therefore classified as outliers (depicted with red crosses in Figure 2.42).

Experiment 5: Improvements wrt priorities

The objective of this experiment is to investigate how the improvement trends change with different flow priorities. 100 flow-sets are generated, each with 200 flows, where priorities, flow sizes, source and destination cores are randomly generated, and flow sizes are in the following range: $[1B - 1kB]$. For each flow, the worst-case traversal times are computed with both methods, and the improvements achieved by the new approach are measured in relative terms.

Figure 2.43 confirms that, as flow priorities decrease (bigger numbers on the x-axis), the relative improvements increase (y-axis). This confirms the initial assumption that the new approach does not produce significant improvements for the highest-priority flows, because these flows suffer very little interference (if at all). As flow priorities decrease, the interference that flows suffer becomes more substantial, which favours the proposed approach.

Experiment 6: Analysis tightness

The objective of this experiment is to investigate the tightness of the obtained upper-bounds on the worst-case traversal times. To achieve this, a single flow-set consisting of 42 flows is

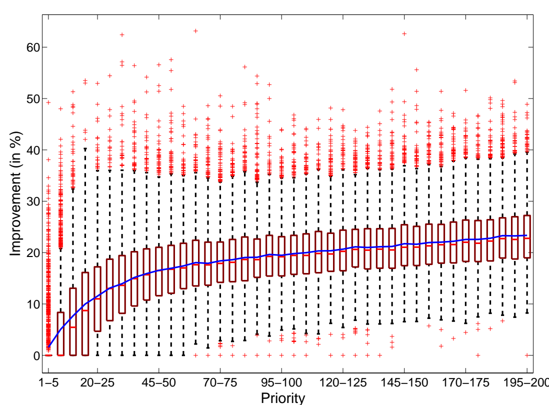


Figure 2.43: Improvements wrt priorities

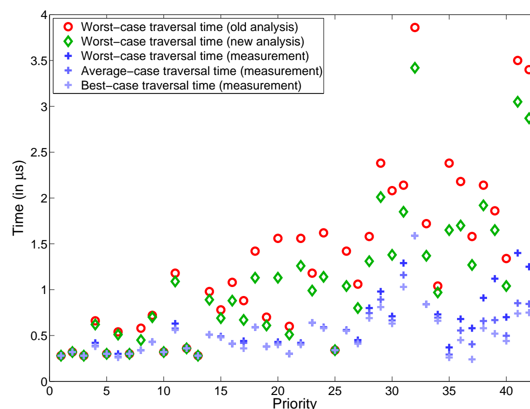


Figure 2.44: Analysis tightness

generated, and subsequently mapped on a 6×6 platform with randomly generated source and destination cores. Each flow has a payload in the range $[2-48]$ flits, and one additional header flit. Moreover, each flow has a period in the range $[0.5-9]ms$, and a unique priority. The router frequency is 100 MHz. The worst-case traversal times of all flows are computed with both analyses, and also the execution is simulated on a cycle-accurate simulator. The simulated time is 2 hyper-periods[4]. Subsequently, the analysis estimates are compared with the values obtained via simulations, namely (i) the worst-case, (ii) the average-case and (iii) the best-case.

Results are depicted in Figure 2.44. It is visible that for high-priority flows both the analyses provide tight bounds. This is expected, given that these flows suffer very little interference, if at all. With the decrease in the priority (bigger numbers on the x-axis), the differences between the observed values and the analysis results become more noticeable (y-axis), because the analysis pessimism accumulates. Also, notice that the difference between the estimates increases with the decrease in priorities. In fact, in some cases the proposed approach provides significantly tighter estimates, which entirely coincides with the conclusions from the previous experiments, and further motivates this work. However, the results also suggest that there is still room for improvement, and this area remains a potential topic for future work.

2.3.5.6 Discussion

In the previous section, the pessimism of the state-of-the-art methods for the worst-case analysis of wormhole-switched priority-preemptive NoCs was identified. Consequently, an improvement was proposed, which overcomes the identified limitations. Through the experimental evaluation, it has been observed that the proposed approach efficiently reduces the pessimism of the existing methods, with respect to direct interference between contending traffic flows. The experiments demonstrate that the proposed approach yields significant improvements in almost all cases, while the greatest pessimism reductions are achieved in scenarios with large flow-sets, where flows have small sizes and traverse long paths. These traffic characteristics correspond to control core-to-core traffic as well as to read requests and write responses in core-to-memory traffic. Given that these

[4] A hyper-period is the least common multiplier of all flow-periods.

traffic types constitute a significant fraction of the entire NoC traffic, the proposed method not only can help to exploit the platform more efficiently and decrease the resource over-provisioning, but also can render many flow-sets schedulable, even though the existing methods classified them as unschedulable. These findings will most likely motivate and elicit further research efforts in the area of the worst-case analysis of wormhole-switched priority-preemptive NoCs, with the main objective to additionally decrease the analysis pessimism, especially in the domain of indirect interferences, which still remains an unexplored topic.

Chapter 3

Limited Migrative Model - LMM

In this chapter, a new workload execution paradigm is presented. This approach is called the *Limited Migrative Model*, *LMM* hereafter. There are two major differences between *LMM* and the existing approaches:

1. In *LMM*, each functionality may be executed on an arbitrary number of cores. The candidate cores for each functionality are selected at design-time, and during runtime the functionality may freely migrate across its respective candidate cores.

2. The release/migration decisions of each functionality are made by the functionality itself, which removes the requirement of a mandatory centralised scheduling entity. That is, the functionality decides on which core it will perform its computation, while a local kernel on that core is responsible to schedule the execution in a single-core fashion.

This unique design allows to embrace the positive characteristics of the existing techniques. For example, on each core a local kernel is responsible to schedule the workload, which is a scalable approach, very similar to the systems with the fully-partitioned scheduling. At the same time, each functionality has the possibility to migrate across its candidate cores, which makes the approach flexible, similar to the systems with the global scheduling. So far, *LMM* appears to be a promising approach for integration of many-cores into the real-time embedded domain, and the research activities presented in the rest of this chapter are motivated with that reasoning.

The graphical representation of *LMM* is given in Figure 3.1, where dotted arrows symbolise candidate cores for each functionality.

3.1 LMM in Detail

3.1.1 Operating System (OS)

As already mentioned, the *LMM* approach has been developed from the assumption that each core has an independent local kernel. A kernel is responsible to schedule the execution on its core, i.e. to organise and arbitrate the computation requests by the functionalities which reside on that

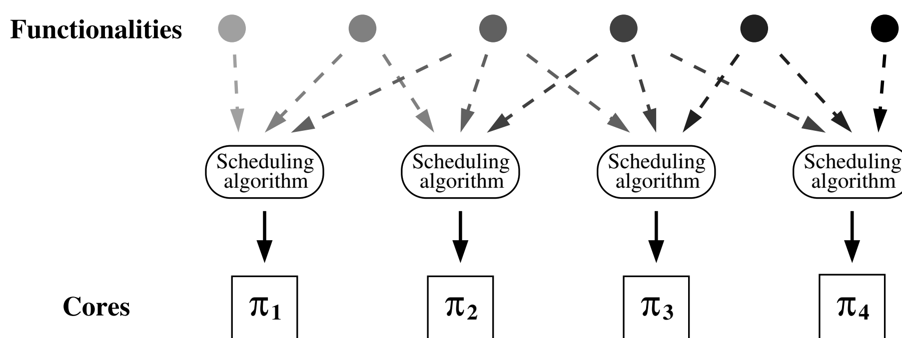


Figure 3.1: Limited Migrative Model

core. Moreover, a kernel exposes some of its features to local functionalities via system calls, and functionalities invoke those system calls in order to e.g. request processing resources for their computations, or communicate with other functionalities located on the same or other cores.

Each kernel is able to communicate with other kernels over the NoC interconnect. The communication among kernels is necessary, among other things, to maintain the coherent and consistent system-wide state. This means that *LMM* uses the message-passing technique as a communication primitive. By combining the concepts of independent kernels with the message passing, a novel operating system paradigm is created, in the literature known as the *multi-kernel*. The multi-kernel is a relatively new, yet very promising approach in the design of operating systems for many-core platforms. Some notable examples of multi-kernels are Barrelfish [11], fOS [94] and Quest-V [57].

3.1.2 Application Layer

As mentioned earlier, each functionality (application) may perform the computation on an arbitrary set of candidate cores, which are for each application selected at design-time. On each of the candidate cores an application's execution code exists, encapsulated within an entity called the *dispatcher*. Thus, the number of dispatchers of each application is equal to the number of its candidate cores.

All dispatchers of one application (each located on a different core) communicate with each other. The communication of dispatchers is termed the *agreement protocol*. The purpose of the agreement protocol is to elect one dispatcher, called the *master dispatcher*. After the election process, the master is responsible to perform the computation on its core, on behalf of the entire application. In order to do that, the master invokes a system call, by which it requests from the local kernel to consider its requirements when deriving future scheduling decisions. The computation process must be performed entirely on the core of the master. When the computation completes, the master is responsible to initiate the next instance of the agreement protocol, so as to elect the master dispatcher for the next computation process. If the elected dispatcher is not the current master, a migration occurs.

All applications are single-threaded, therefore, at any time instance there can be only one master per application. The rest of the dispatchers are called the *slave dispatchers*, and their purpose is to participate in the agreement protocol. When the master initiates the protocol, slaves invoke system calls of their kernels, requesting the information regarding the possibility to perform the next computation on their cores. After receiving the answers from their respective kernels, dispatchers communicate with each other. Based on the information provided by the kernels, the next master is elected[1]. If the newly elected dispatcher is the existing master, nothing changes. Otherwise, the newly elected dispatcher becomes a master, while the old master becomes a slave. Additionally, the execution context has to be transferred from the old to the new master. Finally, after the computation is successfully completed on its core, the new master will initiate the new instance of the agreement protocol.

Notice that being the master is only a temporary role of a dispatcher. Perceived from the application's perspective, its dispatchers exchange one master token. This property is termed the *master volatility*, and it has several interesting implications which will be covered later when performing the timing analysis.

The agreement protocols are classified as the intra-application communication, because the communication is performed only between the dispatchers of the same application. Additionally, applications may communicate with each other for e.g. synchronisation or data sharing purposes. This inter-application communication is implemented by a message exchange between current master dispatchers of interacting applications.

3.1.3 LMM Benefits

• **Configurability:** The greatest power of *LMM* lies within its configurability. When allowing each application to have only a single dispatcher, the behaviour of the system is identical to the one with the fully-partitioned scheduling policy. Conversely, if every application has a dispatcher on every core of the platform, the system acquires the properties of the ones with the global scheduling policy, although at the expense of extensive communication related to agreement protocols. Thus, by consciously choosing the number of dispatchers for each application, a desired trade-off between the flexibility and the amount of protocol-related communication can be achieved to fit the actual purpose.

• **Scalability:** Irrespective of the number of dispatchers, the scheduling decisions are always local, made by the kernels, which makes the approach scalable. This is possible because in *LMM* release/migration decisions are derived on the application level, and are explicitly detached from scheduling decisions, which is a novel concept in the real-time domain. The greatest benefit of this distributed decision making process is that the centralised entity (e.g. ready queue) is not needed.

[1] In this dissertation, the timing analysis of agreement protocols is of interest. The election policy problem (how to choose the master dispatcher) depends on the purpose of the system and has no effect on the timing analysis, which makes it immaterial for the discussion in this dissertation.

• **Flexibility:** Each application has the migrative freedom, based on the number of its dispatchers. Thus, *LMM* is able to efficiently exploit the potential of the underlying many-core platform by performing the energy/thermal management via runtime load balancing.
• **Resilience:** Failures of individual cores or clusters of cores can be overcome by excluding the dispatchers located on those cores from agreement protocols. In a similar way, voluntary core shutdowns can be implemented for various beneficial reasons (e.g. to save power, to prolong hardware life).

3.2 Application Workload

The workload consists of an application-set $\mathscr{A}$, which is a collection of v applications (functionalities): $\mathscr{A} = \{a_1, a_2, ..., a_{v-1}, a_v\}$. An application a_i has a unique priority $P(a_i)$, a set of u dispatchers $\mathscr{D}(a_i) = \{d_i^1, d_i^2, ..., d_i^{u-1}, d_i^u\}$, a minimum inter-arrival period $T(a_i)$ and an implicit deadline $D(a_i) = T(a_i)$.

Computation: The computation requirements of any application a_i are modelled by a single sporadic task (recall that applications are single-threaded), which is a source of an infinite number of recurring jobs released with the minimum inter-arrival period equal to $T(a_i)$. A job is released by the local kernel, upon a request from the current master dispatcher of a_i. The released job inherits the priority of its application, has a constant execution time $C^\tau(a_i)$, and has a deadline denoted by $D^\tau(a_i) < D(a_i)$. In other words, when analysing only the computation process, a job released at the time instant t has to execute for $C^\tau(a_i)$ time units until $t + D^\tau(a_i)$. If it fails to do so, it has missed a deadline. Conversely, if guarantees can be provided that every job of a_i can meet its deadline, then a_i is considered *schedulable with respect to its computation requirements*.

Memory: During the computation process, jobs of the application a_i may need to access and manipulate data. This means that during a single job execution, multiple accesses to the memory controllers might be necessary. Therefore, the memory operations of the jobs belonging to a_i are modelled by a flow-set $\mathscr{F}^\mu(a_i)$. A detailed description of $\mathscr{F}^\mu(a_i)$ will be given in Section 3.7, when the timing analysis of the memory traffic will be performed. At this stage it is sufficient to only mention that all flows belonging to $\mathscr{F}^\mu(a_i)$ have a joint deadline $D^\mu(a_i) < D(a_i)$. In other words, when considering only memory operations, all memory traffic of the application a_i has to complete its transfer over the NoC within $D^\mu(a_i)$. If it fails to do so, it has missed a deadline. Conversely, if guarantees can be provided that the memory traffic of a_i will not miss a deadline, the application is considered *schedulable with respect to its memory requirements*.

Notice that neither the computation process, nor the memory access process are continuous intervals. In fact, these two processes are mutually interleaved, and that has to be taken into account when performing the timing analysis. To that aim, another deadline is defined, which is equal to the sum of the aforementioned deadlines, i.e. $D^{\tau+\mu}(a_i) = D^\tau(a_i) + D^\mu(a_i)$. In order the application to be schedulable with respect to both its computation and memory requirements, it has to be proven that its computation takes no longer than $D^\tau(a_i)$, and its memory operations take no longer than $D^\mu(a_i)$, when considering $D^{\tau+\mu}(a_i)$ as the time interval of interest.

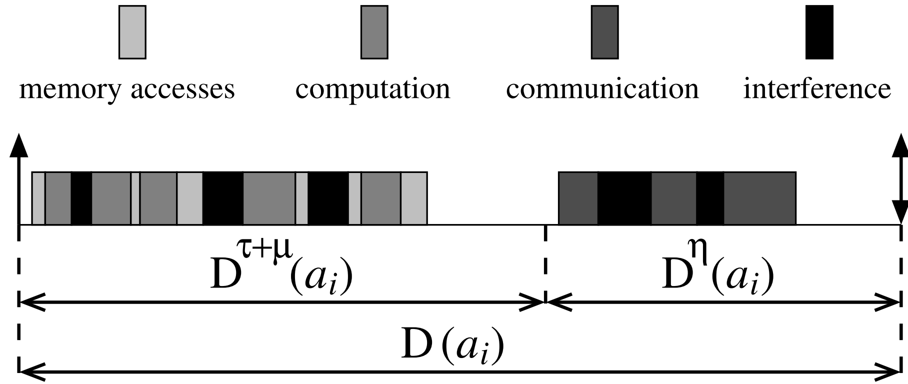


Figure 3.2: Example of application's computation, memory access and communication patterns

Communication: Each application may perform two types of communication: intra- and inter-application. The former covers the message exchange by the dispatchers of the same application and one example of such communication is the agreement protocol. Conversely, the inter-application traffic models the message exchange between the current masters of two interacting applications.

Let $\mathscr{F}^{\eta}(a_i)$ be a flow-set which models all intra- and inter-application communication of the application a_i. A detailed description of $\mathscr{F}^{\eta}(a_i)$ will be given in Sections 3.3-3.5, when the timing analysis of the communication will be performed. At this stage it is sufficient to only mention that all flows belonging to $\mathscr{F}^{\eta}(a_i)$ have a joint deadline $D^{\eta}(a_i) < D(a_i)$. In other words, all communication traffic of the application a_i has to complete its transfer over the NoC within $D^{\eta}(a_i)$. If it fails to do so, it has missed a deadline. Conversely, if guarantees can be provided that the communication traffic of a_i will not miss a deadline, the application is considered *schedulable with respect to its communication requirements*. If the application is schedulable with respect to its (i) computation requirements, (ii) memory requirements, and (iii) communication requirements, then it is considered *totally schedulable*. If all applications of the application-set are totally schedulable, the application-set is totally schedulable.

Notice that while the computation process and the memory access process are mutually interleaved and form discontinuous time intervals, the communication process of each application forms a continuous time interval, and therefore, when performing the timing analysis of the communication of a_i, only the interval $D^{\eta}(a_i)$ needs to be considered. Also note that the sum of the three aforementioned deadlines is equal to the inter-arrival period of the application and thus its deadline, i.e. $D^{\tau}(a_i) + D^{\mu}(a_i) + D^{\eta}(a_i) = D^{\tau+\mu}(a_i) + D^{\eta}(a_i) = D(a_i)$. These facts are illustrated with Figure 3.2.

3.3 Agreement Protocols

When performing the agreement protocol, the dispatchers may invoke six different OS operations, as demonstrated in Table 3.1, where symbols denote the latencies of respective operations. Note

that these operations are executed on a first-come-first-serve basis and always have the same priority (the highest). Therefore, the protocol-related OS operations, invoked by any application, will always preempt the job execution of any other application, irrespective of their priorities.

Table 3.1: OS operations related to agreement protocols

$\delta_P^{\rightarrow}$	Send the protocol message (performed by all dispatchers)
$\delta_P^{\leftarrow}$	Receive the protocol message (performed by all dispatchers)
$\delta_C^{\rightarrow}$	Send the execution context (performed by the old master during migration)
$\delta_C^{\leftarrow}$	Receive the execution context (performed by the newly elected master during migration)
δ_Q	Get the info. from the local kernel about the next job release (performed by all dispatchers)
δ_E	Elect the next master (performed by the old master)

In the following sections three agreement protocols will be introduced.

3.3.1 Master-Slave Protocol

3.3.1.1 Protocol Description

The dispatcher behaviour under this protocol is illustrated with Algorithm 9. Within every inter-arrival period of application, after its computation and memory access deadlines expire, the agreement protocol begins. At that time instant, the master initiates the protocol by sending messages to all slaves (lines $4-7$). When a slave receives the message from the master, it requests from the local kernel the information whether the next job can be released on that core or not (lines $25-26$). Upon receiving that information, the slave sends it back to the master (line 27). The master waits until it receives the replies from all slaves (lines $9-12$). After that, the master gets the information from its kernel (line 14), and compares it with the information received from all slaves, in order to elect the next master (line 15). If the elected master is the same as the old master, nothing changes; the master requests the next job release from the kernel, which is deferred until the communication deadline $D^{\eta}(a_i)$ expires (line 18). Then, the master waits for the time instant to start the next protocol (line 4).

Conversely, if the elected dispatcher is not the current master, the migration occurs. The old master sends the execution context to the new master, and demotes itself to the slave role (lines $20-21$). At the same time, each slave waits for one of two events: (i) the context transfer from the old master, or (ii) the beginning of the new protocol (line 28). In the former case, the slave promotes itself to the master role, and requests the next job release from the kernel (lines $29-32$).

3.3.1.2 Timing Analysis

During this protocol, in the worst-case the master can perform the following OS operations: (i) send the protocol message $|\mathscr{D}(a_i)|-1$ times, (ii) receive the protocol message $|\mathscr{D}(a_i)|-1$

Algorithm 9 $run()$

```
1:  while (true) do
2:    if (isMaster = true) then
3:      // dispatcher is master
4:      wait(startTime); // wait for the time instant to start protocol
5:      for each (d_i^j ∈ 𝒟(a_i) | d_i^j ≠ this) do
6:        sendMsg(d_i^j); // send messages to all slaves
7:      end for
8:      rcvdMsgs ← 0;
9:      while (rcvdMsgs < |𝒟(a_i)|) do
10:       wait(msgRcvd);
11:       rcvdMsgs++;
12:     end while
13:     // replies from all slaves received
14:     getNextReleaseInfo();
15:     nextMaster ← chooseNextMaster();
16:     if (nextMaster = this) then
17:       // master remains the same
18:       releaseDeferredJob();
19:     else
20:       sendCtx(nextMaster); // master changes (migration occurs)
21:       isMaster ← false;
22:     end if
23:   else
24:     // dispatcher is slave
25:     wait(msgRcvd);
26:     getNextReleaseInfo();
27:     sendMsg(master);
28:     wait(ctxRcvd, startTime);
29:     if (ctxRcvd = true) then
30:       isMaster ← true; // master changes (migration occurs)
31:       releaseDeferredJob();
32:     end if
33:   end if
34: end while
```

times, (iii) query the kernel for the information regarding the next release, (iv) elect the next master, and (v) transfer the context to the new master. Thus, the delay of the protocol execution on the core of the master, denoted by $C_M^\eta(a_i)$, can be expressed with Equation 3.1.

$$C_M^\eta(a_i) = (|\mathscr{D}(a_i)| - 1) \cdot \delta_P^{\rightarrow} + (|\mathscr{D}(a_i)| - 1) \cdot \delta_P^{\leftarrow} + \delta_Q + \delta_E + \delta_C^{\rightarrow} \tag{3.1}$$

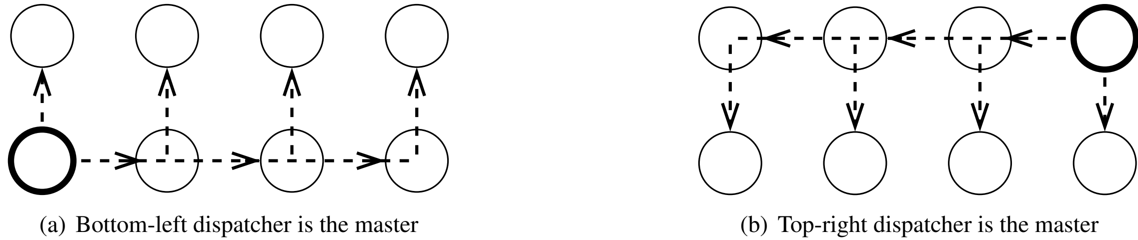
Similarly, the delay of the protocol execution on the core of the slave that will not become the next master, denoted by $C_S^\eta(a_i)$, can be expressed with Equation 3.2.

$$C_S^\eta(a_i) = \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} \tag{3.2}$$

Finally, the delay of the protocol execution on the core of the slave that will become the next master, denoted by $C_N^\eta(a_i)$, can be expressed with Equation 3.3.

$$C_N^\eta(a_i) = \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_C^{\leftarrow} \tag{3.3}$$

Besides the aforementioned terms, in order to compute the total delay of the protocol execution, it is necessary to calculate the delay of the message transfer over the NoC interconnect. However, due to the master volatility, that task is not trivial. This problem is depicted in Figure 3.3, where the master broadcast (the beginning of the protocol) of the same application is captured at two different time instants. Notice that depending on which dispatcher is the current master (emphasised circle), produced messages may traverse entirely different routes.



(a) Bottom-left dispatcher is the master

(b) Top-right dispatcher is the master

Figure 3.3: Agreement protocol messages are master-dependent

This problem can be circumvented in the following way. Recall (Equation 2.1) that the traversal delay of each flow (message) f_j in isolation is the function of two properties: (i) it's size $\sigma(f_j)$ and (ii) its path length $|\mathscr{L}(f_j)|$. The size of the message is deterministic, but the path is not. Let $maxhops(a_i)$ be the maximum distance between any two dispatchers of the application a_i to which f_j belongs. As $maxhops(a_i)$ is an upper-bound on the path length of each protocol message f_j, the analysis covers the worst-case by assuming that each f_j traverses that distance, $|\mathscr{L}(f_j)| = maxhops(a_i), \forall f_j \in \mathscr{F}^\eta(a_i)$.

With this assumption, the message transfer delay $C_\#^\eta(a_i)$ can be computed by solving Equation 3.4. In total $2 \cdot (|\mathscr{D}(a_i)| - 1)$ protocol messages are exchanged between the master and all slaves, followed by a context transfer to the new master.

$$C^{\eta}_{\#}(a_i) = 2 \cdot (|\mathscr{D}(a_i)| - 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$

$$maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.4}$$

In Equation 3.4, σ_{prt} and σ_{ctx} denote the size of one protocol message and one context message, respectively.

After these terms have been obtained, the total delay of the protocol execution $C^{\eta}(a_i)$ can be computed by solving Equation 3.5. Notice that because all slaves perform their protocols in parallel, it is sufficient to take into account only one of them (the next master).

$$C^{\eta}(a_i) = \overbrace{C^{\eta}_{M}(a_i)}^{\text{old master delay}} + \overbrace{C^{\eta}_{N}(a_i)}^{\text{next master delay}} + \overbrace{C^{\eta}_{\#}(a_i)}^{\text{network delay}} \tag{3.5}$$

Equation 3.5 represents the delay of the protocol execution of an application in isolation, assuming that it does not suffer any interference from other applications. In order to obtain the worst-case protocol delay, the interference from other applications has to be taken into account.

First, consider the interference that any dispatcher, master or slave, may suffer on its core while performing the protocol-related OS operations. The dispatcher may suffer interference from other masters or slaves residing on the same core and performing their agreement protocols. In order to compute the maximum interference that a dispatcher may suffer, Theorem 6 is used.

Theorem 6. *The number of protocol executions of any application a_i within the time interval t can be at most* $1 + \left\lceil \frac{t - D^{\tau+\mu}(a_i)}{T(a_i)} \right\rceil$.

Proof. Proven by contradiction. Assume that $2 + \left\lceil \frac{t - D^{\tau+\mu}(a_i)}{T(a_i)} \right\rceil$ protocol executions occurred within the time interval t. There are $\left\lceil \frac{t - D^{\tau+\mu}(a_i)}{T(a_i)} \right\rceil$ protocol executions surrounded by the first and the last and these are refer to as the *inner executions*. All the inner executions contribute to t with their entire application inter-arrival period $T(a_i)$, and therefore require time interval of at least $\left\lceil \frac{t - D^{\tau+\mu}(a_i)}{T(a_i)} \right\rceil \cdot T(a_i)$ where only these can execute. Additionally, assume that ε is infinitesimally small but finite value representing the shortest possible duration of the protocol and that the first protocol execution with the duration of ε was delayed as much as possible and hence completed just before the interval of the inner executions started. Finally, the last protocol execution could not start before the joint deadline of the computation and memory accesses – $D^{\tau+\mu}(a_i)$ expired.

$$\varepsilon + \left\lceil \frac{t - D^{\tau+\mu}(a_i)}{T(a_i)} \right\rceil \cdot T(a_i) + D^{\tau+\mu}(a_i) \geq \varepsilon + \left(\frac{t - D^{\tau+\mu}(a_i)}{\cancel{T(a_i)}} \right) \cdot \cancel{T(a_i)} + D^{\tau+\mu}(a_i) = \varepsilon + t \leq t$$

The contradiction has been reached.

□

The worst-case interference that a dispatcher d_i^j can suffer on its core within the time interval t, termed $I^\eta(d_i^j,t)$ can be computed by solving Equation 3.6.

$$I^\eta(d_i^j,t) = \sum_{\forall d_k^m \in \mathscr{D}_{\pi(d_i^j)} | d_k^m \neq d_i^j} \left(1 + \left\lceil \frac{t - D^{\tau+\mu}(a_k)}{T(a_k)} \right\rceil \right) \cdot \begin{cases} C_M^\eta(a_k) & \text{if } d_k^m \text{ is master} \\ C_N^\eta(a_k) & \text{if } d_k^m \text{ is slave (next master)} \\ C_S^\eta(a_k) & \text{if } d_k^m \text{ is slave (not next master)} \end{cases} \tag{3.6}$$

In Equation 3.6 the term $\pi(d_i^j)$ represents the core of the dispatcher d_i^j, while $\mathscr{D}_{\pi(d_i^j)}$ denotes all dispatchers residing on that core.

In addition to the on-core interference, when performing its agreement protocol, an application may also suffer the interference within the NoC interconnect, called the network interference. The network interference that the application of interest a_i suffers from other applications within the time interval t, denoted by $I_\#^\eta(a_i,t)$, can be computed by solving Equation 3.7.

$$I_\#^\eta(a_i,t) = \sum_{\forall a_k \in \mathscr{A} | P(a_k) > P(a_i)} \left(1 + \left\lceil \frac{t - D^{\tau+\mu}(a_k)}{T(a_k)} \right\rceil \right) \cdot C_\#^\eta(a_k) \tag{3.7}$$

In other words, it is assumed that the protocol messages of each higher-priority application a_k will cause the interference to the protocol messages of a_i, irrespective of their potential paths. This indeed is a conservative assumption, however, it is one way to circumvent the problem of non-deterministic message paths caused by the master volatility property.

The worst-case protocol delay can be computed by summing up the aforementioned terms. That is expressed with Equation 3.8, where the *latest slave interference* corresponds to the delay of the slave which was the last to deliver its response to the master. Additionally, the terms t^* and t^+ are the sub-intervals of the worst-case protocol delay, and correspond to the time intervals during which the latest slave and the next master can suffer the interference, respectively.

$$R^\eta(a_i) = \overbrace{C^\eta(a_i)}^{\text{isolation delay}} + \overbrace{I^\eta(d_i^j,R^\eta(a_i))}^{\text{master interference}} + \overbrace{I^\eta(d_i^k,t^*)}^{\text{latest slave interference}} + \overbrace{I^\eta(d_i^m,t^+)}^{\text{next master interference}} + \overbrace{I_\#^\eta(a_i,R^\eta(a_i))}^{\text{network interference}} \tag{3.8}$$

Notice that in Equation 3.8 the first and the last term and master-independent. However, the remaining terms depend not only on the decision which dispatchers perform the roles of the master, the latest slave and the next master of the analysed application a_i, but also on the roles that dispatchers of other applications perform during the analysed period. Thus, in order to solve Equation 3.8 it is necessary to identify roles of all dispatchers that will lead to the worst-case (the biggest protocol delay).

Algorithm 10 demonstrates how to identify individual dispatcher roles on the core of the analysed dispatcher, which will lead to its worst-case interference within the time interval t. The value

Algorithm 10 $maxDispInf(d_i^j, isMaster, t)$

Input: analysed dispatcher d_i^j, boolean variable *isMaster*, time interval of interest t
Output: worst-case interference of d_i^j within t

1: $\mathscr{D}_M \leftarrow \emptyset$; // initialise set of on-core masters
2: $\mathscr{D}_S \leftarrow \emptyset$; // initialise set of on-core slaves
3: **if** $(isMaster)$ **then**
4: $\quad \mathscr{D}_M \leftarrow \mathscr{D}_M \bigcup \{d_i^j\}$;
5: **else**
6: $\quad \mathscr{D}_S \leftarrow \mathscr{D}_S \bigcup \{d_i^j\}$;
7: **end if**
8: **for each** $(d_k^m \in \mathscr{D}_{\pi(d_i^j)} \mid d_k^m \neq d_i^j)$ **do**
9: $\quad$ **if** $(|\mathscr{D}_M| < \widehat{M})$ **then**
10: $\quad\quad \mathscr{D}_M \leftarrow \mathscr{D}_M \bigcup \{d_k^m\}$; // add dispatcher to the list of masters
11: $\quad\quad master(d_k^m) \leftarrow true$;
12: $\quad$ **else**
13: $\quad\quad$ // find master which causes the minimum interference
14: $\quad\quad minMasterDelay \leftarrow 0$;
15: $\quad\quad$ **for each** $(d_n^p \in \mathscr{D}_M \mid d_n^p \neq d_i^j)$ **do**
16: $\quad\quad\quad currMasterDelay \leftarrow I^\eta(d_n^p, t)$; // Equation 3.6
17: $\quad\quad\quad$ **if** $(currMasterDelay < minMasterDelay)$ **then**
18: $\quad\quad\quad\quad minMasterDelay \leftarrow currMasterDelay$;
19: $\quad\quad\quad\quad minMaster \leftarrow d_n^p$;
20: $\quad\quad\quad$ **end if**
21: $\quad\quad$ **end for**
22: $\quad\quad$ **if** $(I^\eta(d_k^m, t) > minMasterDelay)$ **then**
23: $\quad\quad\quad \mathscr{D}_M \leftarrow \mathscr{D}_M \setminus \{minMaster\}$; // remove the minimum from current masters
24: $\quad\quad\quad master(minMaster) \leftarrow false$;
25: $\quad\quad\quad \mathscr{D}_S \leftarrow \mathscr{D}_S \bigcup \{minMaster\}$; // add the minimum to current slaves
26: $\quad\quad\quad \mathscr{D}_M \leftarrow \mathscr{D}_M \bigcup \{d_k^m\}$; // add dispatcher to current masters
27: $\quad\quad\quad master(d_k^m) \leftarrow true$;
28: $\quad\quad$ **else**
29: $\quad\quad\quad \mathscr{D}_S \leftarrow \mathscr{D}_S \bigcup \{d_k^m\}$; // add dispatcher to current slaves
30: $\quad\quad$ **end if**
31: $\quad$ **end if**
32: **end for**
33: // all roles assigned so compute maximum interference
34: $maxDispInf \leftarrow 0$;
35: **for each** $(d_k^m \in \mathscr{D}_{\pi(d_i^j)} \mid d_k^m \neq d_i^j)$ **do**
36: $\quad maxDispInf \leftarrow maxDispInf + I^\eta(d_k^m, t)$; // Equation 3.6
37: **end for**
38: **return** $maxDispInf$;

$\widehat{M}$ denotes the maximum number of concurrent masters on a single core, and it is assumed that this value has already been specified.

Algorithm 10 works as follows. First the lists of on-core masters and slaves are initialised (lines $1-2$). Then, if the analysed dispatcher d_i^j is the master, it is added to the list of masters (line 4). Otherwise, it is added to the list of slaves (line 6). After that, each dispatcher d_k^m that shares the core with d_i^j is considered for assignment to one of the aforementioned lists. If the current number of assigned masters is less than the maximum, d_k^m is added to the list of masters (lines $10-11$). Otherwise, the master dispatcher which causes the minimum interference is identified (lines $14-21$). If the interference caused by the current dispatcher d_k^m is greater than the interference caused by the identified $minMaster$, then $minMaster$ is transferred to the list of slaves, while d_k^m is added to the list of masters (lines $23-27$). Otherwise, d_k^m is added to the list of slaves (line 29). This process is repeated for every dispatcher. After the lists of masters and slaves have been populated with all on-core dispatchers, the worst-case interference is computed (lines $34-37$) and returned (line 38).

Algorithm 10 demonstrates how to identify on-core master and slave dispatchers of other applications which lead to the worst-case delay to the analysed dispatcher within the observed time interval. However, in order to compute the worst-case protocol delay, it is necessary to also identify dispatcher roles for the application under analysis. For that Algorithm 11 is used.

Algorithm 11 is divided into 4 parts. First, the master of the analysed application is identified (lines $4-11$) in the following way. For each dispatcher d_i^j, the maximum interference that it can suffer within the observed interval is computed. That value is obtained by invoking previously described Algorithm 10 (line 7). The dispatcher with the maximum interference is identified, and assigned the master role (line 9).

Then, the latest slave of the analysed application is identified in a similar way (lines $12-23$). The process is slightly more complex than identifying the master, because the latest slave can suffer the interference only during t^*, which is a sub-interval of the entire worst-case protocol delay. For each dispatcher d_i^k that is not the master, the analysed interval t^* is initially computed for the minimal value – $\delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow}$ (line 18). It is obtained by invoking Algorithm 10. After the new value of t^* is obtained, it is fed back into the computation as the input. The process is repeated until the fixed converging point is reached. The stopping condition is the same value of t^* for two consecutive iterations (line 19). The process is repeated for every dispatcher, until the one with the biggest interval t^* is identified and assigned the latest slave role (line 21).

After that, the next master is identified ($24-35$). Notice that it can also be the latest slave, thus the same dispatcher may be identified for both roles. The computation process is very similar to the previous one. The only difference is that the analysed interval t^+ is initially computed for the minimal value, which represents a single OS operation – receiving the execution context (line 30). Again, the stopping condition is that the value of t^+ has the same value for two successive iterations (line 31). Of all slave dispatchers, the one with the biggest interval t^+ is identified and assigned the next master role (line 33).

Algorithm 11 $compWorstCaseDelay(a_i, \mathscr{A})$

Input: application a_i, application-set $\mathscr{A}$
Output: worst-case protocol delay of a_i

1: $R^{\eta}(a_i) \leftarrow 0$; // initialise the worst-case protocol delay
2: **repeat**
3: $R^{\eta}_{old}(a_i) \leftarrow R^{\eta}(a_i)$;
4: // 1. identify the master
5: $maxMasterInf \leftarrow 0$;
6: **for each** $(d_i^k \in \mathscr{D}(a_i))$ **do**
7: $currMasterInf \leftarrow maxDispInf(d_i^k, true, R^{\eta}_{old}(a_i))$; // Algorithm 10 and Equation 3.6
8: **if** $(currMasterInf > maxMasterInf)$ **then**
9: $maxMasterInf \leftarrow currMasterInf$; $master \leftarrow d_i^j$;
10: **end if**
11: **end for**
12: // 2. identify the latest slave
13: $maxSlaveInf \leftarrow 0$;
14: **for each** $(d_i^k \in \mathscr{D}(a_i) | d_i^k \neq master)$ **do**
15: $t^* \leftarrow 0$;
16: **repeat**
17: $t^*_{old} \leftarrow t^*$;
18: $t^* \leftarrow maxDispInf(d_i^k, false, t^*_{old} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow})$; // Algorithm 10 and Equation 3.6
19: **until** $(t^*_{old} = t^*)$;
20: **if** $(t^* > maxSlaveInf)$ **then**
21: $latestSlave \leftarrow d_i^j$; $maxSlaveInf \leftarrow t^*$;
22: **end if**
23: **end for**
24: // 3. identify the next master
25: $maxNextMasterInf \leftarrow 0$;
26: **for each** $(d_i^k \in \mathscr{D}(a_i) | d_i^k \neq master)$ **do**
27: $t^+ \leftarrow 0$;
28: **repeat**
29: $t^+_{old} \leftarrow t^+$;
30: $t^+ \leftarrow maxDispInf(d_i^k, false, t^+_{old} + \delta_C^{\leftarrow})$; // Algorithm 10 and Equation 3.6
31: **until** $(t^+_{old} = t^+)$;
32: **if** $(t^+ > maxNextMasterInf)$ **then**
33: $nextMaster \leftarrow d_i^j$; $maxNextMasterInf \leftarrow t^+$;
34: **end if**
35: **end for**
36: // 4. compute the worst-case protocol delay;
37: $iso \leftarrow C^{\eta}(a_i)$; // compute isolation delay (Equation 3.5)
38: $netInf \leftarrow I^{\eta}_{\#}(a_i, R^{\eta}_{old}(a_i))$; // compute network interference (Equation 3.7)
39: $R^{\eta}(a_i) \leftarrow iso + netInf + maxMasterInf + maxSlaveInf + maxNextMasterInf$; // Equation 3.8
40: **until** $(R^{\eta}_{old}(a_i) = R^{\eta}(a_i))$;
41: **return** $R^{\eta}(a_i)$;

Finally, the worst-case protocol delay can be computed. As shown in Equation 3.8, it consists of five components: (i) the isolation delay, (ii) the master interference, (iii) the latest slave interference, (iv) the next master interference, and (v) the network interference. In previous steps the second, the third and the fourth component were obtained. Thus, the remaining components are obtained by solving Equations 3.5 and Equation 3.7 (lines $37-38$). The worst-case protocol delay is equal to the sum of these terms (line 39 and Equation 3.8). Similarly to intervals t^* and t^+, the worst-case protocol delay $R^\eta(a_i)$ is also computed iteratively, until the computed value is the same in two successive iterations. The schedulability condition is that the obtained value is less than or equal to the communication deadline, i.e. $R^\eta(a_i) \leq D^\eta(a_i)$.

3.3.1.3 Discussion

The decision made on the core of the master dispatcher is based on the information received from every individual slave. However, in the moment when the master makes a decision, there are no guarantees that the state of the system on the core of each slave is still identical to the one made during the individual observations. One extreme, yet possible scenario occurs when one slave gives a positive reply to its master regarding the next job execution on its core. Additionally, some other dispatchers from the same core might have also sent positive replies to their respective masters during their protocol executions. As a result, multiple applications might elect dispatchers from that particular core for the next masters, and hence overload the core, which inevitably leads towards missed deadlines. Therefore, *the race condition* is identified as the greatest flaw of this protocol. The performance of the protocol will receive additional attention in Section 3.3.4.

3.3.2 List Protocol

3.3.2.1 Protocol Description

In this protocol, dispatchers form a singly linked list, and each dispatcher knows its successor, termed *nextDisp*. The dispatcher behaviour under this protocol is illustrated with Algorithm 12. Within every inter-arrival period of application, after its computation and memory access deadlines expire, the agreement protocol begins. At that time instant, the master initiates the protocol by getting the information regarding the next job release from its kernel (line 5). If the kernel provides a positive reply, nothing changes; the master requests the next job release from the kernel (line 8), and waits for the time instant to start the next protocol (line 4).

In cases when the kernel provides a negative reply, the master sends the message to its successor dispatcher (line 10). When that dispatcher receives the message (line 17), it requests the information regarding the next job release from its kernel (line 18). If the kernel does not allow a new job release, the dispatcher sends the message to its successor dispatcher (line 25). This process continues until the kernel of one dispatcher provides a positive reply. That dispatcher will become the next master. Therefore, it informs the old master about the outcome by requesting the execution context (line 20), and waits for the context to be transferred (line 21). The old master

sends the context (line 12), and demotes itself to the slave role (line 13). After receiving the context, the new master claims the master role (line 22), and requests the next job release from its kernel (line 23).

Algorithm 12 *run*()

```
 1: while (true) do
 2:   if (isMaster = true) then
 3:     // dispatcher is master
 4:     wait(startTime); // wait for the time instant to start protocol
 5:     getNextReleaseInfo();
 6:     if (canReleaseDeferredJob() = true) then
 7:       // master remains the same
 8:       releaseDeferredJob();
 9:     else
10:       sendMsg(nextDisp); // send a message to the successor dispatcher
11:       wait(ctxReqRcvd);
12:       sendCtx(nextMaster); // master changes (migration occurs)
13:       isMaster ← false;
14:     end if
15:   else
16:     // dispatcher is slave
17:     wait(msgRcvd);
18:     getNextReleaseInfo();
19:     if (canReleaseDeferredJob() = true) then
20:       sendCtxReq(master); // send a context request to the master
21:       wait(ctxRcvd);
22:       isMaster ← true; // master changes (migration occurs)
23:       releaseDeferredJob();
24:     else
25:       sendMsg(nextDisp); // send a message to the successor dispatcher
26:     end if
27:   end if
28: end while
```

3.3.2.2 Timing Analysis

The worst-case scenario occurs when only the last dispatcher of the list is granted the permission to release the next job, and that case has to be considered when performing the analysis. Providing guarantees that, at any time instant, at least one of the dispatchers will be able to accommodate the next job on its core falls into the domain of the schedulability analysis, and that topic will receive additional attention in Section 3.8.

During this protocol, in the worst-case the master can perform the following OS operations: (i) query the kernel for the information regarding the next release, (ii) send the protocol message, (iii) receive the context request, and (iv) transfer the context to the next master. Thus, the delay

of the protocol execution on the core of the master, denoted by $C_M^\eta(a_i)$, can be expressed with Equation 3.9.

$$C_M^\eta(a_i) = \delta_Q + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_C^{\rightarrow} \tag{3.9}$$

Similarly, the delay of the protocol execution on the core of the slave that will not become the next master, denoted by $C_S^\eta(a_i)$, can be expressed with Equation 3.10.

$$C_S^\eta(a_i) = \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} \tag{3.10}$$

The delay of the protocol execution on the core of the slave that will become the next master, denoted by $C_N^\eta(a_i)$, can be expressed with Equation 3.11.

$$C_N^\eta(a_i) = \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_C^{\leftarrow} \tag{3.11}$$

The message transfer delay $C_\#^\eta(a_i)$ can be computed by solving Equation 3.12. In total $|\mathscr{D}(a_i)|$ protocol messages are exchanged when traversing the singly linked list of dispatchers, followed by a context transfer to the new master.

$$C_\#^\eta(a_i) = |\mathscr{D}(a_i)| \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) + \\ maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.12}$$

After these terms have been obtained, the total delay of the protocol execution $C^\eta(a_i)$ can be computed by solving Equation 3.13. Notice that in this protocol the slaves perform their protocol-related OS operations sequentially, and therefore all of them have to be taken into account when computing the total protocol delay.

$$C^\eta(a_i) = \overbrace{C_M^\eta(a_i)}^{\text{old master delay}} + \overbrace{(|\mathscr{D}(a_i)| - 2) \cdot C_S^\eta(a_i)}^{\text{delay of all slaves except next master}} + \overbrace{C_N^\eta(a_i)}^{\text{next master delay}} + \overbrace{C_\#^\eta(a_i)}^{\text{network delay}} \tag{3.13}$$

Equation 3.13 represents the delay of the protocol execution of an application in isolation, assuming that it does not suffer any interference from other applications. In order to obtain the worst-case protocol delay, the interference from other applications has to be taken into account.

The interference that any dispatcher, master or slave, may suffer while performing its protocol-related OS operations within the time interval t, termed $I^\eta(d_i^j, t)$, has already been computed for the Master-Slave protocol (Equation 3.6). Since this term is protocol-independent, it can be reused in this context. The same holds for the network interference that an application might suffer on the NoC within the time interval t, denoted by $I_\#^\eta(a_i, t)$ (Equation 3.7).

After all the relevant terms have been identified, the worst-case protocol delay can be computed by solving Equation 3.14.

$$R^{\eta}(a_i) = \overbrace{C^{\eta}(a_i)}^{\text{isolation delay}} + \overbrace{I^{\eta}(d_i^j, R^{\eta}(a_i))}^{\text{master interference}} + \overbrace{\sum_{\forall d_i^k \in \mathscr{D}(a_i) | d_i^k \neq d_i^j} I^{\eta}(d_i^k, t^*)}^{\text{all slaves interference}} +$$
$$\overbrace{I^{\eta}(d_i^m, t^+)}^{\text{next master interference}} + \overbrace{I_{\#}^{\eta}(a_i, R^{\eta}(a_i))}^{\text{network interference}} \tag{3.14}$$

The term t^* corresponds to the individual time interval during which each slave suffers the interference, while t^+ represents an additional interval during which the next master suffers the interference when receiving the context. Of course, both t^* and t^+ are the sub-intervals of $R^{\eta}(a_i)$.

Notice that in Equation 3.14 the first and the last term and master-independent. However, the remaining terms depend not only on the decision which dispatchers of the analysed application a_i perform the roles of the old and the new master, but also on the roles that dispatchers of other applications perform during the analysed period. Thus, like in the previous case, in order to solve Equation 3.14 it is necessary to identify roles of all dispatchers that will lead to the worst-case (the biggest protocol delay).

Algorithm 10 was introduced before, and it was used to compute the maximum interference that an individual dispatcher might suffer during the time interval t, due to the other on-core dispatchers. Since that algorithm is protocol-independent, it will also be reused in this context. Therefore, the only remaining activity is to identify the dispatcher roles for the application under analysis. For that, Algorithm 13 is used.

Algorithm 13 is divided into 4 parts. First, the master of the analysed application is identified (lines $4-11$). This process is identical to the process of identifying the master for the Master-Slave protocol.

Then, the joint interference suffered by all slaves of the analysed application is computed (lines $12-21$). Each slave can suffer the interference only during its individual interval t^*. This term represents an interval during which the slave performs its protocol-related OS operations, and it is the sub-interval of the entire worst-case protocol delay. For each dispatcher d_i^k that is not the master, the analysed interval t^* is initially computed for the minimal value $-\delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow}$ (line 18). It is obtained by invoking Algorithm 10. After the new value of t^* is computed, it is fed back into the computation as the input. The process is repeated until the fixed converging point is reached. The stopping condition is the same value of t^* for two consecutive iterations (line 19). The process is repeated for every slave dispatcher, and the obtained values of t^* are summed up (line 20). The summation corresponds to the maximum interference that all slaves might suffer during the protocol execution.

After that, the next master is identified $(22-33)$. This process is identical to the process of identifying the next master of the Master-Slave protocol.

Algorithm 13 $compWorstCaseDelay(a_i, \mathscr{A})$

Input: application a_i, application-set $\mathscr{A}$

Output: worst-case protocol delay of a_i

1: $R^{\eta}(a_i) \leftarrow 0$; // initialise the worst-case protocol delay
2: **repeat**
3: $R^{\eta}_{old}(a_i) \leftarrow R^{\eta}(a_i)$;
4: // 1. identify the master
5: $maxMasterInf \leftarrow 0$;
6: **for each** $(d_i^k \in \mathscr{D}(a_i))$ **do**
7: $currMasterInf \leftarrow maxDispInf(d_i^k, true, R^{\eta}_{old}(a_i))$; // Algorithm 10 and Equation 3.6
8: **if** $(currMasterInf > maxMasterInf)$ **then**
9: $maxMasterInf \leftarrow currMasterInf$; $master \leftarrow d_i^j$;
10: **end if**
11: **end for**
12: // 2. compute interference suffered by all slaves
13: $maxSlavesInf \leftarrow 0$;
14: **for each** $(d_i^k \in \mathscr{D}(a_i) | d_i^k \neq master)$ **do**
15: $t^* \leftarrow 0$;
16: **repeat**
17: $t^*_{old} \leftarrow t^*$;
18: $t^* \leftarrow maxDispInf(d_i^k, false, t^*_{old} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow})$; // Algorithm 10 and Equation 3.6
19: **until** $(t^*_{old} = t^*)$;
20: $maxSlavesInf \leftarrow maxSlavesInf + t^*$;
21: **end for**
22: // 3. identify the next master
23: $maxNextMasterInf \leftarrow 0$;
24: **for each** $(d_i^k \in \mathscr{D}(a_i) | d_i^k \neq master)$ **do**
25: $t^+ \leftarrow 0$;
26: **repeat**
27: $t^+_{old} \leftarrow t^+$;
28: $t^+ \leftarrow maxDispInf(d_i^k, false, t^+_{old} + \delta_C^{\leftarrow})$; // Algorithm 10 and Equation 3.6
29: **until** $(t^+_{old} = t^+)$;
30: **if** $(t^+ > maxNextMasterInf)$ **then**
31: $nextMaster \leftarrow d_i^j$; $maxNextMasterInf \leftarrow t^+$;
32: **end if**
33: **end for**
34: // 4. compute the worst-case protocol delay;
35: $iso \leftarrow C^{\eta}(a_i)$; // compute isolation delay (Equation 3.13)
36: $netInf \leftarrow I^{\eta}_{\#}(a_i, R^{\eta}_{old}(a_i))$; // compute network interference (Equation 3.7)
37: $R^{\eta}(a_i) \leftarrow iso + netInf + maxMasterInf + maxSlavesInf + maxNextMasterInf$; // Equation 3.14
38: **until** $(R^{\eta}_{old}(a_i) = R^{\eta}(a_i))$;
39: **return** $R^{\eta}(a_i)$;

Finally, the worst-case protocol delay can be computed. As shown in Equation 3.14, it consists of five components: (i) the isolation delay, (ii) the master interference, (iii) the interference of all slaves, (iv) the next master interference, and (v) the network interference. In previous steps the second, the third and the fourth component were obtained. Thus, the remaining components are obtained by solving Equations 3.13 and Equation 3.7 (lines $35-36$). The worst-case protocol delay is equal to the sum of these terms (line 37 and Equation 3.14). Similarly to intervals t^* and t^+, the worst-case protocol delay $R^\eta(a_i)$ is also computed iteratively, until the computed value is the same in two successive iterations. The schedulability condition is that the obtained value is less than or equal to the communication deadline, i.e. $R^\eta(a_i) \leq D^\eta(a_i)$.

3.3.2.3 Discussion

The execution of the protocol stops at the moment when one of the dispatchers of the traversed singly linked list announces the possibility to accommodate the next job release. In many cases that dispatcher might not be the optimal choice, e.g. the core of some other dispatcher, yet not traversed, might offer better execution environment (less interference). As implicitly stated, the greatest limitation of this protocol is that the dispatchers are traversed in a static, predefined order, whereas the decision regarding the new release may be derived without considering all of them. This means that it is not possible to implement any selective scheduling policy, which are necessary for energy/thermal management and other beneficial purposes. The performance of the protocol will receive additional attention in Section 3.3.4.

3.3.3 Hybrid Protocol

3.3.3.1 Protocol Description

This protocol is the combination of the aforementioned two protocols and it consists of two phases. The dispatcher behaviour is illustrated with Algorithm 14. Since the logic of the protocol is more complex, the behaviour of the master dispatcher has been covered with the auxiliary Algorithm 15, while the auxiliary Algorithm 16 describes the behaviour of slave dispatchers.

Algorithm 14 *run*()

```
1: while (true) do
2:   if (isMaster = true) then
3:     // dispatcher is master
4:     runMaster(); // Algorithm 15
5:   else
6:     // dispatcher is slave
7:     runSlave(); // Algorithm 16
8:   end if
9: end while
```

Within every inter-arrival period of application, after its computation and memory access deadlines expire, the master initiates the agreement protocol (line 1 of Algorithm 15). At that time

Algorithm 15 $runMaster()$

```
1:  wait(startTime); // wait for the time instant to start protocol
2:  for each (d_i^j ∈ 𝒟(a) | d_i^j ≠ this) do
3:     sendMsg(d_i^j); // send messages to all slaves
4:  end for
5:  rcvdMsgs ← 0;
6:  while (rcvdMsgs < |𝒟(a_i)|) do
7:     wait(msgRcvd);
8:     rcvdMsgs++;
9:  end while
10: // replies from all slaves received
11: getNextReleaseInfo();
12: orderedList ← sortDispatchers(); // create ordered list of all dispatchers
13: if (first(orderedList) = this) then
14:    // master remains the same
15:    releaseDeferredJob();
16: else
17:    sendMsg(first(orderedList));
18:    wait(msgRcvd, ctxReqRcvd);
19:    if (ctxReqRcvd) then
20:       sendCtx(nextMaster); // master changes (migration occurs)
21:       isMaster ← false;
22:    else
23:       getNextReleaseInfo();
24:       if (canReleaseDeferredJob() = true) then
25:          // master remains the same
26:          releaseDeferredJob();
27:       else
28:          sendMsg(next(orderedList)); // send a message to the successor dispatcher
29:          wait(ctxReqRcvd);
30:          sendCtx(nextMaster); // master changes (migration occurs)
31:          isMaster ← false;
32:       end if
33:    end if
34: end if
```

instant, the first phase begins. This phase is very similar to the Master-Slave protocol. The master sends messages to all slaves (lines $2-4$ of Algorithm 15). Upon receiving the message from the master (line 1 of Algorithm 16), each slave requests from the local kernel the information whether the next job can be released on that core (line 2 of Algorithm 16). After receiving that information, each slave sends it back to the master (line 3 of Algorithm 16). The master waits until it receives the replies from all slaves (lines $6-9$ of Algorithm 15). Once all the replies are received, the master requests the information from its kernel (line 11 of Algorithm 15), and together with the information received from all slaves generates a list where all dispatchers are sorted according to their likelihood of accommodating the next job release (line 12 of Algorithm 15). At this moment the first phase finishes.

Algorithm 16 *runSlave*()

```
1:  wait(msgRcvd);
2:  getNextReleaseInfo();
3:  sendMsg(master);
4:  wait(msgRcvd, startTime);
5:  if (msgRcvd = true) then
6:     getNextReleaseInfo();
7:     if (canReleaseDeferredJob() = true) then
8:        sendCtxReq(master); // send a context request to the master
9:        wait(ctxRcvd);
10:       isMaster ← true; // master changes (migration occurs)
11:       releaseDeferredJob();
12:    else
13:       sendMsg(next(orderedList)); // send a message to the successor dispatcher
14:    end if
15: end if
```

The second phase is very similar to the List protocol. If the first dispatcher in the sorted list is the same as the old master, nothing changes; the master requests the next job release from its kernel (line 15 of Algorithm 15), and waits for the time instant to start the next protocol (line 1 of Algorithm 15).

Conversely, the master sends the message to the first dispatcher of the list (line 17 of Algorithm 15). At this stage, each slave dispatcher waits for one of two possible events: (i) the arrival of the protocol message, or (ii) the beginning of the new protocol (line 4 of Algorithm 16). In the former case, the slave again requests the information regarding the next job release from its kernel (line 6 of Algorithm 16). If the kernel does not allow a new job release, the dispatcher sends the message to the next dispatcher from the list (line 13 of Algorithm 16). This process repeats until the kernel of one dispatcher provides a positive reply, and that dispatcher will become the next master.

If that dispatcher is the current master, nothing changes; the master requests the next job release from its kernel (line 26 of Algorithm 15), and waits for the time instant to start the next protocol (line 1 of Algorithm 15). Otherwise, if that dispatcher is the slave, it informs the old

master about the outcome by requesting the execution context (line 8 of Algorithm 16), and waits for the context to be transferred (line 9 of Algorithm 16). Then, the old master sends the context to the new master (line 20 or line 30 of Algorithm 16), and demotes itself to the slave role (line 21 or line 31). After it receives the context, the next master promotes itself to the master role and requests the next job release from its kernel (lines $10-11$ of Algorithm 16).

3.3.3.2 Timing Analysis

The worst-case scenario occurs when only the last dispatcher of the ordered list is granted the permission to release the next job, and that case has to be considered when performing the analysis.

During this protocol, in the worst-case the master can perform the following OS operations: (i) send the protocol message $|\mathscr{D}(a_i)|-1$ times, (ii) receive the protocol message $|\mathscr{D}(a_i)|-1$ times, (iii) query the kernel for the information regarding the next release, (iv) generate the sorted list of all dispatchers, (v) send the protocol message to the first dispatcher from the list, (vi) receive the protocol message, (vii) query the kernel for the information regarding the next release, (viii) send the protocol message, (ix) receive the context request, and (x) transfer the context to the next master. Thus, the delay of the protocol execution on the core of the master, denoted by $C_M^{\eta}(a_i)$, can be expressed with Equation 3.15.

$$C_M^{\eta}(a_i) = (|\mathscr{D}(a_i)|-1)\cdot\delta_P^{\rightarrow} + (|\mathscr{D}(a_i)|-1)\cdot\delta_P^{\leftarrow} + \delta_Q + \delta_E + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_C^{\rightarrow} \tag{3.15}$$

Similarly, the delay of the protocol execution on the core of the slave that will not become the next master, denoted by $C_S^{\eta}(a_i)$, can be expressed with Equation 3.16. Of interest for the analysis is to identify the parts of $C_S^{\eta}(a_i)$ that were executed during the first and the second phase, termed $C_{S1}^{\eta}(a_i)$ and $C_{S2}^{\eta}(a_i)$, respectively.

$$C_S^{\eta}(a_i) = \overbrace{\delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow}}^{C_{S1}^{\eta}(a_i)} + \overbrace{\delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow}}^{C_{S2}^{\eta}(a_i)} \tag{3.16}$$

The delay of the protocol execution on the core of the slave that will become the next master, denoted by $C_N^{\eta}(a_i)$, can be expressed with Equation 3.17.

$$C_N^{\eta}(a_i) = \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_C^{\leftarrow} \tag{3.17}$$

The message transfer delay $C_{\#}^{\eta}(a_i)$ can be computed by solving Equation 3.18. During the first phase, $2\cdot(|\mathscr{D}(a_i)|-1)$ protocol messages are exchanged between the master and all slaves. During the second phase, $|\mathscr{D}(a_i)|$ protocol messages are exchanged when traversing the singly linked list of dispatchers, followed by a context request from the new master and the subsequent context transfer.

$$C_{\#}^{\eta}(a_i) = 2 \cdot (|\mathscr{D}(a_i)| - 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$

$$(|\mathscr{D}(a_i)| + 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$

$$maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.18}$$

After these terms have been obtained, the total delay of the protocol execution $C^{\eta}(a_i)$ can be computed by solving Equation 3.19. Notice that during the first phase all slaves perform their protocols in parallel, thus it is sufficient to take into account only one of them, e.g. the next master. During the second phase, slaves perform their protocol-related OS operations sequentially, so all of them have to be taken into account when computing the total protocol delay. This is why $C_S^{\eta}(a_i)$ had to be divided into $C_{S1}^{\eta}(a_i)$ and $C_{S2}^{\eta}(a_i)$ in Equation 3.16.

$$C^{\eta}(a_i) = \overbrace{C_M^{\eta}(a_i)}^{\text{old master delay}} + \overbrace{C_N^{\eta}(a_i)}^{\text{next master delay}} + \overbrace{(|\mathscr{D}(a_i)| - 2) \cdot C_{S2}^{\eta}(a_i)}^{\text{2}^{\text{nd}}\text{ phase delay of all remaining slaves}} + \overbrace{C_{\#}^{\eta}(a_i)}^{\text{network delay}} \tag{3.19}$$

Equation 3.19 represents the delay of the protocol execution of an application in isolation, assuming that it does not suffer any interference from other applications. In order to obtain the worst-case protocol delay, the interference from other applications has to be taken into account.

The interference that any dispatcher, master or slave, may suffer while performing its protocol-related OS operations within the time interval t, termed $I^{\eta}(d_i^j, t)$, has already been computed for the Master-Slave protocol (Equation 3.6). Since this term is protocol-independent, it can be reused in this context. The same holds for the network interference that an application might suffer on the NoC within the time interval t, denoted by $I_{\#}^{\eta}(a_i, t)$ (Equation 3.7).

After all the relevant terms have been identified, the worst-case protocol delay can be computed by solving Equation 3.20.

$$R^{\eta}(a_i) = \overbrace{C^{\eta}(a_i)}^{\text{isolation delay}} + \overbrace{I^{\eta}(d_i^j, R^{\eta}(a_i))}^{\text{master interference}} + \overbrace{I^{\eta}(d_i^k, t^*)}^{\text{latest slave interference in 1}^{\text{st}}\text{ phase}} +$$

$$\overbrace{\sum_{\forall d_i^m \in \mathscr{D}(a_i) | d_i^m \neq d_i^j} I^{\eta}(d_i^m, t^+)}^{\text{all slaves interference in 2}^{\text{nd}}\text{ phase}} + \overbrace{I^{\eta}(d_i^n, t^{\triangle})}^{\text{next master interference}} + \overbrace{I_{\#}^{\eta}(a_i, R^{\eta}(a_i))}^{\text{network interference}} \tag{3.20}$$

The term *latest slave interference in 1st phase* corresponds to the delay of the slave which was the last to deliver its response to the master during the first phase. Additionally, the terms t^*, t^+ and $t^\triangle$ are the sub-intervals of the worst-case protocol delay. t^* corresponds to the time interval of the first phase during which the latest slave suffers the interference. t^+ denotes to the individual time interval of the second phase during which each slave suffers the interference. $t^\triangle$ represents an additional interval during which the next master suffers the interference while receiving the context.

Notice that in Equation 3.20 the first and the last term are master-independent. However, the remaining terms depend not only on the decision which dispatchers of the analysed application a_i perform the roles of the master, the latest slave and the next master, but also on the roles that dispatchers of other applications perform during the analysed period. Thus, like in the previous case, in order to solve Equation 3.14 it is necessary to identify roles of all dispatchers that will lead to the worst-case (the biggest protocol delay).

Algorithm 10 was introduced before, and it was used to compute the maximum interference that an individual dispatcher might suffer during the time interval t, due to the other on-core dispatchers. Since that algorithm is protocol-independent, it will also be reused in this context. Therefore, the only remaining activity is to identify the dispatcher roles for the application under analysis. For that, Algorithm 17 is used, which is divided into 5 parts. First, the master of the analysed application is identified (lines $4-11$). This process is identical to the process of identifying the master for the Master-Slave and List protocols.

Then, the latest slave of the analysed application during the first phase is identified (lines $12-18$). This process is identical to the process of identifying the latest slave for the Master-Slave protocol.

After that, the joint interference suffered by all slaves of the analysed application during the second phase is computed (lines $19-25$). This process is identical to the process of computing the joint delay of all slaves for the List protocol.

Then, the next master is identified ($26-32$). This process is identical to the process of identifying the next master of the Master-Slave and List protocols.

Finally, the worst-case protocol delay can be computed. As shown in Equation 3.20, it consists of six components: (i) the isolation delay, (ii) the master interference, (iii) the interference of the latest slave during the first phase, (iv) the interference of all slaves during the second phase, (v) the next master interference, and (vi) the network interference. In previous steps the second, the third, the fourth and the fifth component were obtained. Thus, the remaining components are obtained by solving Equations 3.19 and Equation 3.7 (lines $34-35$). The worst-case protocol delay is equal to the sum of these terms (line 36 and Equation 3.20). Similarly to intervals t^*, t^+ and $t^\triangle$, the worst-case protocol delay $R^\eta(a_i)$ is also computed iteratively, until the computed value is the same in two successive iterations. The schedulability condition is that the obtained value is less than or equal to the communication deadline, i.e. $R^\eta(a_i) \leq D^\eta(a_i)$.

Algorithm 17 $compWorstCaseDelay(a_i, \mathscr{A})$

Input: application a_i, application-set $\mathscr{A}$
Output: worst-case protocol delay of a_i

1: $R^{\eta}(a_i) \leftarrow 0$; // initialise the worst-case protocol delay
2: **repeat**
3: $R^{\eta}_{old}(a_i) \leftarrow R^{\eta}(a_i)$;
4: // 1. identify the master
5: $maxMasterInf \leftarrow 0$;
6: **for each** $(d_i^k \in \mathscr{D}(a_i))$ **do**
7: $currMasterInf \leftarrow maxDispInf(d_i^k, true, R^{\eta}_{old}(a_i))$;
8: **if** $(currMasterInf > maxMasterInf)$ **then**
9: $maxMasterInf \leftarrow currMasterInf$; $master \leftarrow d_i^j$;
10: **end if**
11: **end for**
12: // 2. identify the latest slave
13: $maxSlaveInf \leftarrow 0$;
14: **for each** $(d_i^k \in \mathscr{D}(a_i) | d_i^k \neq master)$ **do**
15: $t^* \leftarrow 0$;
16: **repeat** $t^*_{old} \leftarrow t^*$; $t^* \leftarrow maxDispInf(d_i^k, false, t^*_{old} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow})$;
until $(t^*_{old} = t^*)$;
17: **if** $(t^* > maxSlaveInf)$ **then** $latestSlave \leftarrow d_i^j$; $maxSlaveInf \leftarrow t^*$;
end if
18: **end for**
19: // 3. compute interference suffered by all slaves
20: $maxSlavesInf \leftarrow 0$;
21: **for each** $(d_i^m \in \mathscr{D}(a_i) | d_i^m \neq master)$ **do**
22: $t^+ \leftarrow 0$;
23: **repeat** $t^+_{old} \leftarrow t^+$; $t^+ \leftarrow maxDispInf(d_i^m, false, t^+_{old} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow})$;
until $(t^+_{old} = t^+)$;
24: $maxSlavesInf \leftarrow maxSlavesInf + t^+$;
25: **end for**
26: // 4. identify the next master
27: $maxNextMasterInf \leftarrow 0$;
28: **for each** $(d_i^n \in \mathscr{D}(a_i) | d_i^n \neq master)$ **do**
29: $t^{\triangle} \leftarrow 0$;
30: **repeat** $t^{\triangle}_{old} \leftarrow t^{\triangle}$; $t^{\triangle} \leftarrow maxDispInf(d_i^n, false, t^{\triangle}_{old} + \delta_C^{\leftarrow})$;
until $(t^{\triangle}_{old} = t^{\triangle})$;
31: **if** $(t^{\triangle} > maxNextMasterInf)$ **then** $nextMaster \leftarrow d_i^n$; $maxNextMasterInf \leftarrow t^{\triangle}$;
end if
32: **end for**
33: // 5. compute the worst-case protocol delay;
34: $iso \leftarrow C^{\eta}(a_i)$; // compute isolation delay (Equation 3.19)
35: $netInf \leftarrow I^{\eta}_{\#}(a_i, R^{\eta}_{old}(a_i))$; // compute network interference (Equation 3.7)
36: $R^{\eta}(a_i) \leftarrow iso + netInf + maxMasterInf + maxSlaveInf + maxSlavesInf + maxNextMasterInf$; // Equation 3.20
37: **until** $(R^{\eta}_{old}(a_i) = R^{\eta}(a_i))$;
38: **return** $R^{\eta}(a_i)$;

3.3.3.3 Discussion

During the first phase of the protocol, all dispatchers are requesting the information regarding the next job release from their respective kernels, and sending it to the current master. During the second phase, the dispatchers are sequentially traversed with the objective to identify the one which will be able to accommodate the release of the next job. The strategy of the protocol is to firstly attempt to release the job on cores of those dispatchers which kernels, when queried during the first phase, reported the most promising environment for the accommodation of that workload. Due to its optimistic nature, when compared with the other two protocols, this one has a higher probability to finish the protocol well before the computed worst-case protocol delay. In fact, since the most promising dispatchers are traversed early during the second phase, it is reasonable to expect that their kernels will be able to accommodate the next job and hence complete the protocol with only a few dispatchers traversed during the second phase. However, this comes at the expense of a more significant amount of traffic. The performance of the protocol will receive additional attention in Section 3.3.4.

3.3.4 Experimental Evaluation

In this section, the experimental evaluation of the agreement protocols is performed. The experiments are conducted on the extended version of the simulator *SPARTS* [73]. For protocol-related OS operations (i.e. to send/receive protocol messages, to send/receive execution contexts, to request of the next job release information from the local kernel, to elect the master dispatcher, to generate the sorted list of dispatchers) are assumed latencies that are valid for present micro-kernels.

The aim of this evaluation is to observe the relations between the analytically obtained worst-case protocol delay (WCPD) values – $R^{\eta}(a_i)$ (computed by the methods described in the previous sections), against the WCPD values obtained during simulations – $R^{\eta}_{*}(a_i)$. Moreover, it will be observed how these trends change for different protocols and different workloads. Finally, by varying the number of dispatchers it will be investigated how this protocol parameter and the amount of traffic influence aforementioned relations and affect the overall protocol behaviour.

Workload, analysis and simulation parameters

When generating the workload for the evaluation, dispatchers of each application are randomly assigned to the cores. The only restriction is that no two dispatchers of the same application can be assigned to the same core. The mapping problem will be thoroughly investigated in Section 3.6. The execution of each application-set will be simulated in two different scenarios: with synchronous and asynchronous protocol releases. In the former case, the idea is to trigger and observe the scenario where all applications start their protocols at the same time (i.e. to generate significant contention), while the latter models a more realistic scenario.

The analysis and simulation parameters are given in Table 3.2, where an asterisk sign denotes a randomly generated value assuming a uniform distribution.

Table 3.2: Analysis and simulation parameters for Section 3.3.4

NoC topology and size	**2-D mesh with 10 × 10 routers**
Link width = flit size σ_{flit}	**16 bytes**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Protocol-related OS operations $\delta_P^{\leftarrow}$, $\delta_P^{\rightarrow}$, $\delta_C^{\leftarrow}$, $\delta_C^{\rightarrow}$, δ_Q, δ_E	**10000 cycles (5 μs)**
Application periods $D(a_i) = T(a_i), \forall a_i \in \mathscr{A}$	**$[100-1000]^*$ ms**
Communication deadlines $D^\eta(a_i), \forall a_i \in \mathscr{A}$	**$0.25 \cdot D(a_i)$**
Protocol message size σ_{prt}	**16 bytes**
Execution context size σ_{ctx}	**1 Kbyte**
Application-set size $\lvert\mathscr{A}\rvert$	**200 applications**
Maximum concurrent on-core masters $\widehat{M}$	**10**
Simulated time	**100 s**

Experiment 1: Analysis pessimism

In this experiment the focus is on the analysis pessimism. In particular, the ratio between $R_*^\eta(a_i)$ and $R^\eta(a_i)$ is observed. Each application is represented with 5 dispatchers, i.e. $|\mathscr{D}(a_i)| = 5, \forall a_i \in \mathscr{A}$.

In Figure 3.4, the horizontal axis represents the $R_*^\eta(a_i)$ values, expressed relatively, as the percentage of the corresponding $R^\eta(a_i)$ values. The vertical axis stands for the amount of applications which fall into a given category (certain ratio between observed and calculated WCPD), expressed as the percentage of the total application-set size.

Since the number of generated messages for the **Master-Slave protocol** is always constant, this protocol exhibits the least amount of pessimism of all three protocols. As a consequence, the $R_*^\eta(a_i)$ values represent greater fractions of the corresponding $R^\eta(a_i)$ ones.

As expected, the timing analysis of the **List protocol** in most cases overestimates the number of messages. That is, the analysis always takes into account the traversal of the entire list, while during simulations such scenarios rarely occurred. Consequently, this protocol exhibits greater pessimism than the Master-Slave protocol.

The **Hybrid protocol** is the combination of the two aforementioned approaches. Since the messages exchanged during the first phase of the protocol follow the logic explained for the Master-Slave protocol (their number is constant), and since they constitute at least $2/3$ of the total number of messages, it is reasonable to expect that the pessimism of the Hybrid protocol will be greater than that of the Master-Slave protocol, but less than that of the List protocol. However, that is not true. The explanation for this surprising finding is as follows. During the second phase, the Hybrid protocol traverses the dispatchers in such a way that the most promising candidates are visited early. Consequently, the number of traversed dispatchers during the second phase in simulations is very small. This infers that the protocol is efficient, which is manifested with small WCPD ratios, i.e. the high amount pessimism. On the other hand, the List protocol pays the price of a non-intelligent pre-determined order by which the dispatchers are being traversed. In other

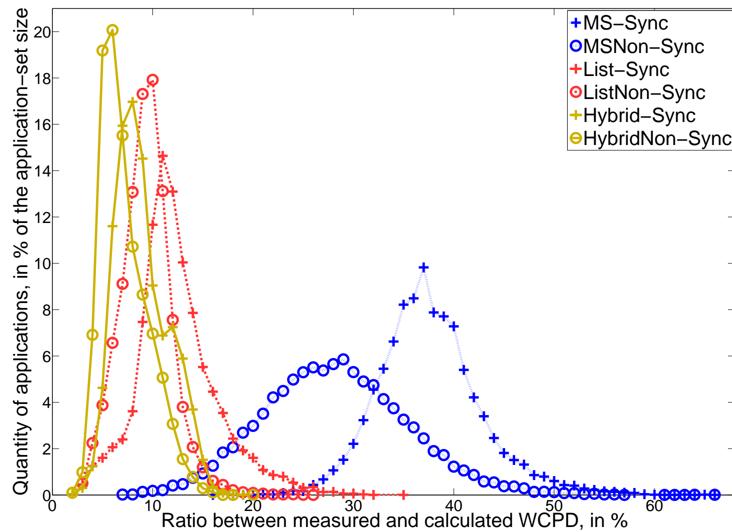


Figure 3.4: Analysis pessimism

words, more dispatchers have to be visited before eventually finding the next master, which is manifested with the smaller amount of pessimism. Due to these facts, the amount of pessimism is greater in the Hybrid than in the List protocol.

As expected, it holds for all three protocols that the **synchronous releases** cause higher WCPD ratios, due to the extensive amount of traffic generated in short periods of time.

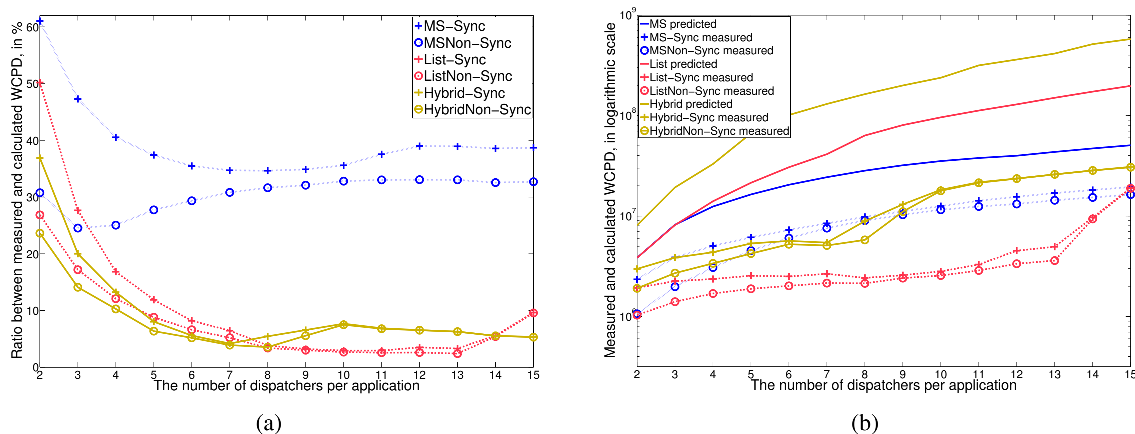
Experiment 2: Scalability

The objective of this experiment is to explore the scalability potential of the proposed agreement protocols. To that end, the number of dispatchers of each application is varied in the on the range $2-15$, i.e. $|\mathscr{D}(a_i)| \in \{2,...,15\}, \forall a_i \in \mathscr{A}$. Subsequently, it is observed how the ratio between the WCPD values, obtained via simulations and via analysis, changes as a function of the number of dispatchers.

In Figure 3.5, the horizontal axis represents the number of dispatchers per application. The vertical axis in Figure 3.5(a) stands for the WCPD values obtained via simulations – $R^{\eta}_{*}(a_i)$, expressed relatively, as the percentage of the analytically obtained WCPD ones – $R^{\eta}(a_i)$, while in Figure 3.5(b) the values of the aforementioned terms are presented in a logarithmic scale.

The **Master-Slave protocol** with non-synchronised releases exhibits almost constant amount of pessimism on the entire observed domain. The visible increase of the pessimism in the left side of Figure 3.5(a) is caused by the pessimistically obtained network interference component, which leads to the noticeable discrepancy between the simulations and analysis; a small number of dispatchers causes fewer NoC contentions during simulations, while the analysis considers that all higher priority traffic existing within the NoC will cause interference to the analysed application. As the number of dispatchers and hence messages increase, the pessimism slowly decreases.

Conversely, for the Master-Slave protocol with synchronised releases, the pessimism is the least for the small number of dispatchers. This occurs for two reasons. First, due to the synchronous releases, contentions always occur, irrespective of the number of per-application dis-



(a) (b)

Figure 3.5: Impact of dispatchers on WCPD

patchers. Second, for fewer messages the difference between the predicted (analysis) and observed (simulations) worst-case scenarios is the least, thus the obtained network interference component is the least pessimistic. Until a certain point, the network successfully copes with the increased amount of dispatchers and messages, hence causing the raise of pessimism. Near the end of the graph, the traffic congestion becomes more significant and a similar trend of a slight pessimism decrease is noticeable. As expected, both the **List protocol** runs (with and without synchronous releases) display an increase of the pessimism as the number of dispatchers increases. The explanation is that in many cases the dispatchers placed near the end of the list are not traversed, while the analysis always considers the worst-case (the traversal of the entire list). Figure 3.5(b) demonstrates that on most of the observed domain additional dispatchers cause a barely noticeable increase of the WCPD, which confirms the previous statement that the dispatchers positioned near the end of the list are in most cases not visited. However, after a certain point, the protocol starts to pay the price of a static non-intelligent traversing, thus causing numerous futile visits of dispatchers which can't accommodate the next job release. Therefore, a significant decrease of the pessimism on the right side of Figure 3.5(a) is visible, leading to a counter-intuitive conclusion that this protocol does not scale when the number of dispatchers is more than a dozen.

Finally, when compared with the List protocol, the **Hybrid protocol** demonstrates a similar behaviour. Due to the safe assumption of the entire list traversal, a steady increase in the pessimism is noticeable as the number of the dispatchers increases. However, after a certain point, the Hybrid protocol pays the price of the extensive communication, leads to network congestions and shows a steady but only temporary increase of the measured WCPD values for $7-10$ dispatchers. One surprising conclusion drawn from Figure 3.5(b) is that there exists an interval (between 3 and 8 dispatchers) where the Hybrid protocol has a shorter measured WCPD than the Master-Slave protocol, despite the fact that it always induces more messages. The explanation is that the Hybrid protocol efficiently selects the next master dispatcher, while the Master-Slave protocol, due to race conditions, causes fragmentation and highly loaded cores, where the on-core protocol-related interference becomes a predominant factor. Additional surprising fact is that the Hybrid protocol

successfully copes with the network congestion by efficiently finding the next master dispatcher, and hence drastically minimising the duration of its second phase. As is visible in Figure 3.5, the Hybrid protocol scales well, displays a good average and worst-case performance in both relative and absolute terms, however, for the very same reasons, exhibits a significant amount of pessimism.

3.4 Inter-application Communication

The analyses proposed in the previous section consider only intra-application communication (agreement protocols). This means that these analyses are applicable only to application-sets with independent applications. To overcome this limitation, in this section the aforementioned analyses will be extended, such that the inter-application communication is considered as well.

Recall, that the inter-application messages are exchanged by the master dispatchers of the interacting applications. Thus, the only components of the aforementioned analyses that are affected are: the master's communication-related OS operations, denoted by $C_M^\eta(a_i)$ (Equation 3.1 for the Master-Slave protocol, Equation 3.9 for the List protocol and Equation 3.15 for the Hybrid protocol), and the delay of the messages over the NoC, termed $C_\#^\eta(a_i)$ (Equation 3.4 for the List protocol, Equation 3.12 for the List protocol and Equation 3.18 for the hybrid protocol). These terms should be extended to also cover the inter-application communication.

In order to do that, first, the new OS operations are introduced, as shown in Table 3.3.

Table 3.3: OS operations related to inter-application communication

$\delta_I^{\rightarrow}$	Send the inter-application message (performed the master dispatcher)
$\delta_I^{\leftarrow}$	Receive the inter-application message (performed by the master dispatcher)

Let $\mathscr{F}_S^\eta(a_i)$ be a set of inter-application messages sent by an application a_i to other applications, such that a message $f_{i,j} \in \mathscr{F}_S^\eta(a_i)$ is sent from the master of the application a_i to the master of the application a_j. Similarly, let $\mathscr{F}_R^\eta(a_i)$ be a set of inter-application messages received by a_i, such that a message $f_{j,i} \in \mathscr{F}_R^\eta(a_i)$ is sent from the master of a_j to the master of a_i.

With these assumptions, Equation 3.1, which covers the communication-related OS operations performed by the master of the application with the Master-Slave protocol, is substituted with Equation 3.21.

$$C_M^\eta(a_i) = \overbrace{(|\mathscr{D}(a_i)|-1)\cdot\delta_P^{\rightarrow} + (|\mathscr{D}(a_i)|-1)\cdot\delta_P^{\leftarrow} + \delta_Q + \delta_E + \delta_C^{\rightarrow}}^{\text{protocol}} + \overbrace{|\mathscr{F}_S^\eta(a_i)|\cdot\delta_I^{\rightarrow} + |\mathscr{F}_R^\eta(a_i)|\cdot\delta_I^{\leftarrow}}^{\text{inter-application traffic}} \quad (3.21)$$

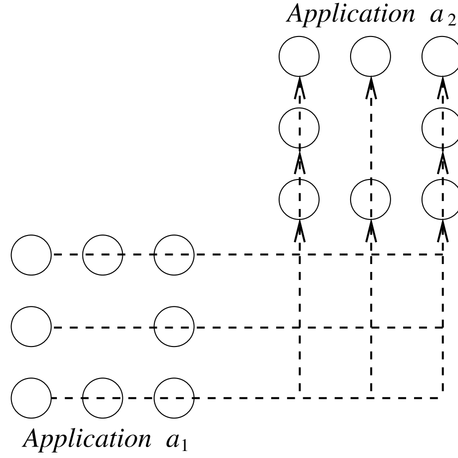


Figure 3.6: Inter-application communication

In a similar way, Equation 3.9, which corresponds to the List protocol is substituted with Equation 3.22.

$$C_M^\eta(a_i) = \overbrace{\delta_Q + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_C^{\rightarrow}}^{\text{protocol}} + \overbrace{|\mathscr{F}_S^\eta(a_i)| \cdot \delta_I^{\rightarrow} + |\mathscr{F}_R^\eta(a_i)| \cdot \delta_I^{\leftarrow}}^{\text{inter-application traffic}} \tag{3.22}$$

Finally, Equation 3.15, which covers the Hybrid protocol is substituted with Equation 3.23.

$$C_M^\eta(a_i) = \overbrace{(|\mathscr{D}(a_i)| - 1) \cdot \delta_P^{\rightarrow} + (|\mathscr{D}(a_i)| - 1) \cdot \delta_P^{\leftarrow} + \delta_Q + \delta_E + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_Q + \delta_P^{\rightarrow} + \delta_P^{\leftarrow} + \delta_C^{\rightarrow}}^{\text{protocol}} +$$

$$\overbrace{|\mathscr{F}_S^\eta(a_i)| \cdot \delta_I^{\rightarrow} + |\mathscr{F}_R^\eta(a_i)| \cdot \delta_I^{\leftarrow}}^{\text{inter-application traffic}} \tag{3.23}$$

Due to the master volatility property, the analysis of the inter-application traffic is not trivial. This problem is illustrated with Figure 3.6, where two applications a_1 and a_2 communicate. Notice that depending on which dispatchers of both applications perform the master role, a single inter-application message $f_{1,2}$ may traverse any of the paths illustrated in Figure 3.6.

One way to circumvent this problem is to apply an approach which is similar to the one used for the intra-application traffic. Specifically, let $maxhops(a_i, a_j)$ be the maximum distance between any two dispatchers of two interacting applications a_i and a_j. Subsequently, the analysis of a_i covers the worst-case by assuming that its every inter-application message traverses its longest possible distance, that is: $|\mathscr{L}(f_{i,j})| = maxhops(a_i, a_j), \forall f_{i,j} \in \mathscr{F}_S^\eta(a_i)$ and $|\mathscr{L}(f_{j,i})| = maxhops(a_j, a_i), \forall f_{j,i} \in \mathscr{F}_R^\eta(a_i)$. Moreover, it is assumed that each intra- and inter-application message with the priority higher than that of a_i: (i) also traverses its longest possible distance, and (ii) causes interference to a_i, irrespective of its potential path.

Note that inter-application messages inherit the priority of the sender application, i.e. $P(f_{i,j}) = P(a_i)$, and it is sender's responsibility to deliver the message to the receiver within its own deadline.

With these assumptions, Equation 3.4, which covers the traffic delay of the application with the Master-Slave protocol, is substituted with Equation 3.24.

$$C_{\#}^{\eta}(a_i) = 2 \cdot (|\mathscr{D}(a_i)| - 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$
$$maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$
$$\sum_{\forall f_{i,j} \in \mathscr{F}_S^{\eta}(a_i)} maxhops(a_i, a_j) \cdot \delta_L + (maxhops(a_i, a_j) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{i,j}}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$
$$\sum_{\forall f_{j,i} \in \mathscr{F}_R^{\eta}(a_i)} maxhops(a_j, a_i) \cdot \delta_L + (maxhops(a_j, a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{j,i}}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.24}$$

In a similar way, Equation 3.12, which corresponds to the List protocol is substituted with Equation 3.25.

$$C_{\#}^{\eta}(a_i) = |\mathscr{D}(a_i)| \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$
$$maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$
$$\sum_{\forall f_{i,j} \in \mathscr{F}_S^{\eta}(a_i)} maxhops(a_i, a_j) \cdot \delta_L + (maxhops(a_i, a_j) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{i,j}}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$
$$\sum_{\forall f_{j,i} \in \mathscr{F}_R^{\eta}(a_i)} maxhops(a_j, a_i) \cdot \delta_L + (maxhops(a_j, a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{j,i}}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.25}$$

Finally, Equation 3.18, which covers the Hybrid protocol is substituted with Equation 3.26.

$$C_{\#}^{\eta}(a_i) = 2 \cdot (|\mathscr{D}(a_i)| - 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$
$$(|\mathscr{D}(a_i)| + 1) \cdot \left(maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \right) +$$
$$maxhops(a_i) \cdot \delta_L + (maxhops(a_i) - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$

$$\sum_{\forall f_{i,j} \in \mathscr{F}_S^{\eta}(a_i)} maxhops(a_i,a_j) \cdot \delta_L + (maxhops(a_i,a_j)-1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{i,j}}}{\sigma_{flit}} \right\rceil \cdot \delta_L +$$

$$\sum_{\forall f_{j,i} \in \mathscr{F}_R^{\eta}(a_i)} maxhops(a_j,a_i) \cdot \delta_L + (maxhops(a_j,a_i)-1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{f_{j,i}}}{\sigma_{flit}} \right\rceil \cdot \delta_L \quad (3.26)$$

Now, depending on the employed agreement protocol, Algorithm 11, Algorithm 13 or Algorithm 17 can be used to compute the worst-case delay of the application a_i, termed $R^{\eta}(a_i)$. However, the obtained delay does not represent the worst-case protocol delay (WCPD) any more, but the entire worst-case communication delay (WCCD). Notice that the condition for the schedulability with respect to the communication remains $R^{\eta}(a_i) \leq D^{\eta}(a_i)$.

3.5 Towards More Deterministic Communication Patterns

Recent insights into priority-preemptive, wormhole-switched NoCs suggest that the dominant factor in the worst-case delay of a flow (message) is not the length of its path, but rather the interference it suffers [74]. Thus, the pessimism related to the approach from the previous section can be attributed mostly to the conservative method which is used to compute the network interference component $I_{\#}^{\eta}$ (by assuming that all higher priority traffic can cause interference, irrespective of (im)possible contentions). Motivated by this reasoning, the novel approach is proposed in this section. This approach relies on enforcing constraints, in order to make *LMM* traffic more deterministic and predictable.

Definition 3 (Application shapes)**.** *Dispatchers of an application can be positioned only on the edges of a rectangular $x \times y$ structure, such that no corner is left unoccupied and $x, y \in \mathbb{N}$. The special case is a line-like shape, where one or both dimensions of the shape are equal to* "1".

Definition 4 (Rerouting operations)**.** *Intra-application messages travel only on the edges of the shape its application forms, and re-routing occurs where needed to comply with the global XY routing policy. An individual message rotation (i.e. clockwise or counterclockwise) is chosen such that the traversal distance is minimised.*

Figure 3.7 demonstrates how dispatchers should be mapped and messages consequently routed. Shaded dispatchers denote locations where reroutings occur. Reroutings are router routines, performed in an interrupt-like manner, which can be, for example, implemented by instrumenting the Hardwall™ technology of Tilera platforms [93]. It is assumed that routers contain sufficient logic and information to manually perform reroutings, without the need to consult local cores. The latency of one rerouting operation is denoted by δ_R.

(a) Bottom-left dispatcher is the master

(b) Top-right dispatcher is the master

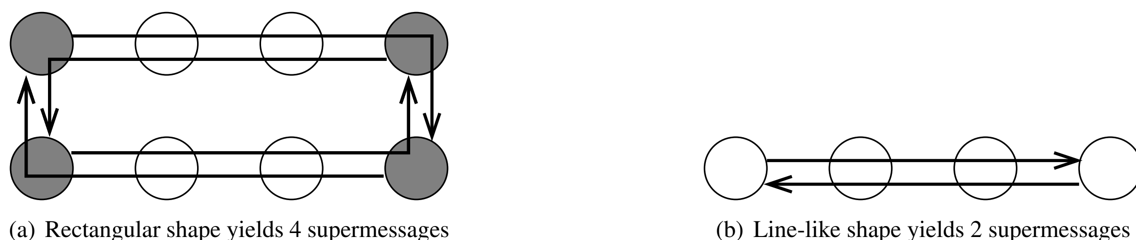
Figure 3.7: Application which shape and messages comply with Definitions 3-4

3.5.1 Supermessages and Proxies

Because of Definitions 3-4, the part of the network that intra-application traffic uses is now deterministic. In order to make the traffic master independent as well, two additional concepts are introduced, namely the *supermessages* and the *proxies*.

Definition 5 (Supermessage)**.** *A supermessage is a message which connects (i) diagonally-placed dispatchers if an application has a rectangular shape, or (ii) terminal dispatchers if an application has a line-like shape.*

According to Definition 5, an application a_i with a line-like shape has 2 supermessages – $\widehat{f_i^{\ell 1}}$, $\widehat{f_i^{\ell 2}}$, and does not involve reroutings, while an application with a rectangular shape has 4 supermessages, of which 2 are with the clockwise orientation $\widehat{f_i^{cw1}}$, $\widehat{f_i^{cw2}}$, and 2 with the counter-clockwise $\widehat{f_i^{cc1}}$, $\widehat{f_i^{cc2}}$ (see Figure 3.8). Without any loss of generality, in the rest of this section only rectangular shapes are analysed, because line-like shapes are the special case of the rectangular shapes. Indeed, any conclusion reached for a rectangular shape can be applied to a line-like shape by considering only one supermessage of each orientation and treating the remaining supermessages as non-existent (i.e. $\widehat{f_i^{\ell 1}} = \widehat{f_i^{cw1}}$; $\widehat{f_i^{\ell 2}} = \widehat{f_i^{cc1}}$; $\widehat{f_i^{cw2}} = null$; $\widehat{f_i^{cc2}} = null$;). Moreover, applications with line-like shapes do not need reroutings for the intra-application traffic (no shaded dispatchers in Figure 3.8(b)).



(a) Rectangular shape yields 4 supermessages

(b) Line-like shape yields 2 supermessages

Figure 3.8: Supermessages for different application shapes

Theorem 7. *Any intra-application message of an application can be expressed by at most 2 distinct, same-orientation supermessages, and at most* 1 *rerouting.*

Proof. Proven by contradiction. Any intra-application message $f_{i,i}$ of an application a_i assumes the orientation such that the distance between the dispatchers is minimised (see Definition 4). Let

c denote the circumference of the application shape. Then, it holds that $|\mathscr{L}(f_{i,i})| \leq \frac{c}{2}$. As each supermessage of a_i connects diagonal corners of its shape, the following holds:
$|\mathscr{L}(\widehat{f_i^{cw1}})| = |\mathscr{L}(\widehat{f_i^{cw2}})| = |\mathscr{L}(\widehat{f_i^{cc1}})| = |\mathscr{L}(\widehat{f_i^{cc2}})| = \frac{c}{2}$.

Assume that $f_{i,i}$ can be expressed with at least 3 same-orientation supermessages. Note, as there are only two same-orientation supermessages (e.g. $\widehat{f_i^{k1}}$ and $\widehat{f_i^{k2}}$, where $k \in \{cc, cw\}$), one of them has to appear twice, i.e. the sequence would be $\{\widehat{f_i^{k1}}, \widehat{f_i^{k2}}, \widehat{f_i^{k1}}\}$ or $\{\widehat{f_i^{k2}}, \widehat{f_i^{k1}}, \widehat{f_i^{k2}}\}$. In either case, the middle supermessage entirely belongs to $f_{i,i}$, while the first and the last belong with fractions $\varepsilon_1 > 0$ and $\varepsilon_2 > 0$, respectively. Hence:

$$|\mathscr{L}(f_{i,i})| = \varepsilon_1 + |\mathscr{L}(\widehat{f_i^{k1}})| + \varepsilon_2 = \varepsilon_1 + |\mathscr{L}(\widehat{f_i^{k2}})| + \varepsilon_2 = \varepsilon_1 + \frac{c}{2} + \varepsilon_2 > \frac{c}{2}$$

The contradiction has been reached. Additionally, as reroutings occur only on places where supermessages meet, and since any message can be expressed by at most 2 distinct supermessages, it can involve at most 1 rerouting. □

Notice that supermessages are master-independent and their number is significantly smaller than the number of possible message paths. Thus, the intuitive idea behind the novel approach is to transform every intra-application message into the corresponding supermessage(s) with eventual rerouting, and perform the analysis on that model.

A similar method is applied to inter-application traffic:

Definition 6 (Proxy)**.** *A proxy dispatcher is a dispatcher which is selected at design-time, and which participates in the inter-application communication. It mediates in the message exchange between its master and the proxy dispatcher of the other (interacting) application.*

An illustrative example of Definition 6 is given in Figure 3.9. In this scenario, an inter-application message is divided into 5 different components: 1) a message from the master sender to its proxy, 2) a rerouting on the router of the proxy sender, 3) a message between the proxies, 4) a rerouting on the router of the proxy receiver, 5) a message from the proxy receiver to its master. Proxies are decided at design-time, thus a message $f_{i,j}^{P}$ between proxy dispatchers of applications a_i and a_j is also deterministic and master-independent. An application can have multiple proxies, each responsible for the communication with a different application. If a proxy receives an inter-application message during the agreement protocol of its application, the message is stalled inside the proxy until the protocol completes and the master (destination) is decided.

Theorem 8. *Any inter-application message can be expressed by (i) at most 2 distinct, same-orientation supermessages on the sender's side, (ii) a message between a proxy sender and a proxy receiver, (iii) at most 2 distinct, same orientation supermessages on the receiver's side and (iv) at most 4 reroutings.*

Proof. Proven directly. A message exchanged between a master sender and its proxy complies with the rules of the intra-application traffic, hence, according to Theorem 7, it can be expressed by at most 2 distinct same-orientation supermessages. The same conclusion holds for the message

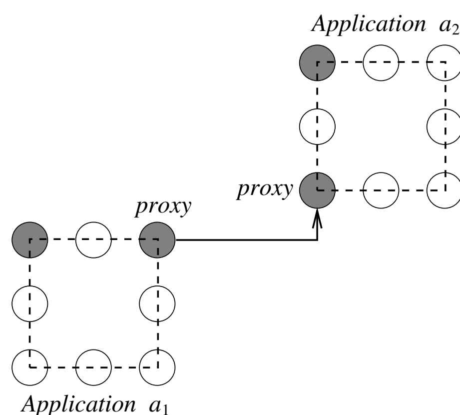


Figure 3.9: Inter-application communication

between a receiver proxy and its master. A message between proxies is a non-constrained point-to-point message.

Messages between masters and their proxies on both sides (sender and receiver) each yield at most 1 rerouting (Theorem 7). Finally, an inter-proxy message causes 1 rerouting on the router of the proxy sender and 1 on the router of the proxy receiver. □

The introduction of placement and rerouting constraints, supermessages and proxies, was a consciously made decision to potentially "sacrifice" performance (i.e. messages may traverse longer distances and may involve reroutings). Nonetheless, this approach yields predictable and deterministic message paths, which allows to perform a more detailed and less pessimistic analysis (covered later in Section 3.5.3), and derive tighter worst-case communication delay estimates.

According to Theorems 7-8, all intra- and inter-application traffic of all applications can be expressed with (i) a set of supermessages – $\widehat{\mathscr{F}}$, (ii) a set of proxy-to-proxy messages – $\mathscr{F}^P$, and (iii) a set of reroutings – $\mathscr{R}$, which are all master-independent and known at design-time. In order to be able to perform the timing analysis of the application a_i, the maximum number of occurrences $\omega(\widehat{f_i^k})$ of its each supermessage $\widehat{f_i^k} \in \widehat{\mathscr{F}}(a_i) \mid k \in \{cw1, cw2, cc1, cc2\}$ and the maximum number of occurrences $\omega(f_{i,j}^P)$ of its each inter-proxy message $f_{i,j}^P \in \mathscr{F}^P(a_i)$ within its minimum inter-arrival period has to be computed. $\widehat{\mathscr{F}}(a_i)$ and $\mathscr{F}^P(a_i)$ denote the sets of supermessages and inter-proxy messages of a_i, respectivelly. Additionally, the maximum number of rerouting occurrences $r(d_i^j) \in \mathscr{R}$, caused by each dispatcher $d_i^j \in \mathscr{D}(a_i)$ on its router $\rho(d_i^j)$ within the same interval has to be computed.

3.5.2 Maximum Number of Message Occurrences

As is evident from the previous section, the maximum number of occurrences of both supermessages and reroutings of each application depends on (i) the employed agreement protocol and (ii) the amount of its inter-application traffic. In Section 3.3, three agreement protocols have been proposed (*Master-Slave*, *List* and *Hybrid*). Since the first one suffers from race conditions, in this section only the second and third will be considered.

3.5.2.1 List Protocol

Before going into details on how to analytically express the List protocol as a function of supermessages and reroutings, two constructs which will aid in that cause are introduced.

Definition 7 (Simple/Complex message)**.** *An intra-application message is called the **simple message** - f^s if it can be expressed with a single supermessage. Otherwise, it is called the **complex message** - f^c.*

(a) Simple messages (b) Complex messages

Figure 3.10: Intra-application messages

An illustrative example of Defnition 7 is given in Figure 3.10.

Theorem 9. *The List protocol (including the context transfer) of an application a_i with $|\mathscr{D}(a_i)|$ dispatchers can be expressed as a linear combination of supermessages, where clockwise supermessages can exist at most $|\mathscr{D}(a_i)|+1$ times, and counter-clockwise can exist at most twice.*

Proof. Proven directly. As neighbouring dispatchers always share the same edge, all messages exchanged by neighbouring dispatchers are *simple messages*. Thus, they can be expressed by only one clockwise supermessage and do not involve reroutings. Unless the exact position of neighbouring dispatchers on the application shape is known, it is not possible to deduce which one of clockwise supermessages will be used. Therefore, each of clockwise supermessages might appear in all $|\mathscr{D}(a_i)|-1$ messages, starting from the master until reaching the last slave. The message from the last slave to the master, and the following context transfer are not necessarily exchanged by neighbouring dispatchers, therefore, in order to cover the worst-case, both have to be considered as *complex messages*. Moreover, these two messages may be of an arbitrary orientation, thus any supermessage may appear once in each of them (Theorem 7). □

Equations 3.27-3.29 express the maximum number of occurrences of each supermessage during one protocol execution. Since contexts and protocol messages may have different sizes and hence different traversal delays, supermessage occurrences related to protocols and contexts have to be counted separately.

$$\omega_P(\widehat{f_i^{cw1}}) = \omega_P(\widehat{f_i^{cw2}}) = |\mathscr{D}(a_i)| \tag{3.27}$$

$$\omega_P(\widehat{f_i^{cc1}}) = \omega_P(\widehat{f_i^{cc2}}) = 1 \tag{3.28}$$

$$\omega_C(\widehat{f_i^{cw1}}) = \omega_C(\widehat{f_i^{cw2}}) = \omega_C(\widehat{f_i^{cc1}}) = \omega_C(\widehat{f_i^{cc2}}) = 1 \tag{3.29}$$

Now, the maximum number of reroutings as a consequence of one protocol execution can be computed by employing Theorem 10.

Theorem 10. *The List protocol of an application a_i with $|\mathscr{D}(a_i)|$ dispatchers can involve at most* 2 *reroutings.*

Proof. Proven directly. Until reaching the last slave all messages are simple messages, can be expressed with one supermessage and hence do not involve reroutings (Theorem 7). A message from the last slave to the master and a subsequent context transfer can be complex messages and, each can contribute with one rerouting (Theorem 7). The maximum number of reroutings is 2. □

From Theorem 10 it straightforwardly follows that the maximum rerouting delay of the List protocol can be computed by solving Equation 3.30. Recall that δ_R denotes the latency of one rerouting operation.

$$C^{\eta}_{RP}(a_i) = 2 \cdot \delta_R \tag{3.30}$$

Note that protocol-related reroutings can be performed only by $\rho(d_i^j)$, such that d_i^j is located in the corner of the application-shape. The maximum number of protocol-related rerouting operations that $\rho(d_i^j)$ may perform, due to d_i^j, is noted down as $r_P(d_i^j)$ (Equation 3.31). Also note that a 4-dispatcher application does not involve reroutings, as all corner dispatchers are mutually reachable via supermessages. In such cases, $C^{\eta}_{RP}(a_i) = 0 \wedge r_P(d_i^j) = 0, \forall d_i^j \in \mathscr{D}(a_i)$. The same is true for applications with line-like shapes.

$$r_P(d_i^j) = 2 \tag{3.31}$$

Note that the supermessage $\widehat{f_i^k}$ may have two different sizes, one for the protocol message – σ_{prt}, and one for the context transfer – σ_{ctx}. Therefore, the traversal delay of the supermessage can be computed either with Equation 3.32 (applicable to protocol messages) or Equation 3.33 (applicable to contexts).

$$C_P(\widehat{f_i^k}) = |\mathscr{L}(\widehat{f_i^k})| \cdot \delta_L + (|\mathscr{L}(\widehat{f_i^k})| - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{prt}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.32}$$

$$C_C(\widehat{f_i^k}) = |\mathscr{L}(\widehat{f_i^k})| \cdot \delta_L + (|\mathscr{L}(\widehat{f_i^k})| - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{ctx}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.33}$$

Now the network delay of the List protocol, termed $C^{\eta}_{\#P}(a_i)$ (Equation 3.34), can be derived. Until reaching the last slave, all requests are simple messages, thus are expressed by only one clockwise supermessage (the first term in Equation 3.34). The response from the last slave and the

context transfer are complex messages of arbitrary orientation, and by Theorem 7 are expressed as two same-orientation supermessages.

$$C^{\eta}_{\#P}(a_i) = \overbrace{(|\mathscr{D}(a_i)|-1)\cdot C_P(\widehat{f_i^k})+}^{\text{until the last slave}} \overbrace{2\cdot C_P(\widehat{f_i^k})}^{\text{last slave to master}} + \overbrace{2\cdot C_C(\widehat{f_i^k})}^{\text{context transfer}} \tag{3.34}$$

3.5.2.2 Hybrid Protocol

Theorem 11. *The Hybrid protocol (including the context transfer) of an application a_i with $|\mathscr{D}(a_i)|$ dispatchers can be expressed as a linear combination of supermessages, where each supermessage can exist at most $3\cdot|\mathscr{D}(a_i)|$ times.*

Proof. In the Hybrid protocol, messages are not necessarily exchanged by neighbouring dispatchers. Therefore, in order to analyse the worst-case, all messages have to be considered as complex messages of arbitrary orientation. By Theorem 7, each message can involve 2 supermessages of the same orientation. Thus, each supermessage can exist once within each protocol message. The first phase involves requests sent to all $|\mathscr{D}(a_i)|-1$ slaves and their $|\mathscr{D}(a_i)|-1$ replies, rendering at most $2\cdot(|\mathscr{D}(a_i)|-1)$ occurrences of each supermessage. During the second phase $|\mathscr{D}(a_i)|+1$ messages are exchanged until reaching the master, resulting in additional $|\mathscr{D}(a_i)|+1$ appearances of each supermessage. Finally, the context transfer yields one additional occurrence of each supermessage. □

The maximum number of occurrences of supermessages during one protocol execution and during one context transfer are given in Equations 3.35-3.36.

$$\omega_P(\widehat{f_i^{cw1}}) = \omega_P(\widehat{f_i^{cw2}}) = \omega_P(\widehat{f_i^{cc1}}) = \omega_P(\widehat{f_i^{cc2}}) = \overbrace{2\cdot(|\mathscr{D}(a_i)|-1)+}^{\text{first phase}} \overbrace{|\mathscr{D}(a_i)|+1}^{\text{second phase}} = 3\cdot|\mathscr{D}(a_i)|-1 \tag{3.35}$$

$$\omega_C(\widehat{f_i^{cw1}}) = \omega_C(\widehat{f_i^{cw2}}) = \omega_C(\widehat{f_i^{cc1}}) = \omega_C(\widehat{f_i^{cc2}}) = 1 \tag{3.36}$$

Now, the maximum number of reroutings as a consequence of one protocol execution can be computed by employing Theorem 12.

Theorem 12. *The Hybrid protocol of an application a_i with $|\mathscr{D}(a_i)|$ dispatchers can involve at most $3\cdot|\mathscr{D}(a_i)|$ reroutings.*

Proof. Proven directly. Each message is treated as a complex message and, by Theorem 7, can cause at most one rerouting. Thus, the maximum number of reroutings is equal to the maximum number of messages, hence in total $3\cdot|\mathscr{D}(a_i)|$ reroutings might occur. □

From Theorem 12 it straightforwardly follows that the maximum rerouting delay of the Hybrid protocol can be computed by solving Equation 3.37.

$$C_{RP}^{\eta}(a_i) = 3 \cdot |\mathscr{D}(a_i)| \cdot \delta_R \tag{3.37}$$

A protocol-related rerouting can be performed by $\rho(d_i^j)$, only if d_i^j is positioned in the corner of the application shape. The maximum number of protocol-related rerouting operations that $\rho(d_i^j)$ may perform, due to d_i^j, is noted down as $r_P(d_i^j)$ (Equation 3.38). Like in the List protocol, 4-dispatcher applications and applications with line-like shapes involve no reroutings: $C_{RP}^{\eta}(a_i) = 0 \wedge r_P(d_i^j) = 0, \forall d_i^j \in \mathscr{D}(a_i)$.

$$r_P(d_i^j) = 3 \cdot |\mathscr{D}(a_i)| \tag{3.38}$$

Now the network delay of the Hybrid protocol – $C_{\#}^{\eta}(a_i)$ can be computed by solving Equation 3.39. Every message has to be treated as a complex message, thus is represented as a sum of two same-orientation supermessages. As in the List protocol, protocol messages and context transfers are treated separately, because $C_P(\widehat{f_i^k}) \neq C_C(\widehat{f_i^k})$.

$$C_{\#P}^{\eta}(a_i) = \overbrace{(3 \cdot |\mathscr{D}(a_i)| - 1) \cdot 2 \cdot C_P(\widehat{f_i^k})}^{\text{first and second phase}} + \overbrace{2 \cdot C_C(\widehat{f_i^k})}^{\text{context transfer}} \tag{3.39}$$

3.5.2.3 Inter-application Traffic

Theorem 13. *Any inter-application message, which is, by applying Theorem 8, expressed as a function of supermessages and the inter-proxy message, can yield at most one occurrence of (i) each supermessage and (ii) the inter-proxy message.*

Proof. Follows directly from Theorems 7-8. □

The implications of Theorem 13 are that each sent and received inter-application message triggers one occurrence of the inter-proxy message and each of the supermessages (Equation 3.40). As different inter-application messages can have different sizes, their occurrences should not be summed up, but instead should be counted separately.

$$\omega_I(\widehat{f_i^{cw1}}) = \omega_I(\widehat{f_i^{cw2}}) = \omega_I(\widehat{f_i^{cc1}}) = \omega_I(\widehat{f_i^{cc2}}) = \omega_I(f_{i,j}^P) = 1 \tag{3.40}$$

The maximum number of reroutings induced by one inter-application message can be computed by applying Theorem 14.

Theorem 14. *Any inter-application message can involve at most* 2 *reroutings on the sender's side, of which at most one can appear on any router.*

Proof. Proven directly. Consider an inter-application message from a master sender to its proxy dispatcher d_i^j. By Theorem 7, it involves one rerouting at the router of the intermediate corner

dispatcher d_i^k which is not d_i^j since it is the destination of that message. When the router of d_i^j is reached, one additional rerouting occurs before the inter-proxy message is forwarded to the receiver proxy d_j^m. Perceived from the sender's perspective, at most 2 reroutings may occur, one performed on $\rho(d_i^j)$, and the other on the router $\rho(d_i^k)$ of any other corner-placed dispatcher d_i^k. □

Equation 3.41 presents the maximum number of reroutings occurring at the router of the same dispatcher d_i^j which is either a corner dispatcher or a proxy. Recall, $\mathscr{F}_S^\eta(a_i)$ and $\mathscr{F}_R^\eta(a_i)$ denote sets of sent and received inter-application messages of the application a_i, respectively.

$$r_I(d_i^j) = |\mathscr{F}_S^\eta(a_i)| + |\mathscr{F}_R^\eta(a_i)| \tag{3.41}$$

The proof for the receiver's side is very similar and is therefore omitted.

Now, the maximum rerouting delay, induced by all sent inter-application traffic of an application a_i, is given in Equation 3.42, while Equation 3.43 holds for all received traffic.

$$C_{RS}^\eta(a_i) = 2 \cdot |\mathscr{F}_S^\eta(a_i)| \cdot \delta_R \tag{3.42}$$

$$C_{RR}^\eta(a_i) = 2 \cdot |\mathscr{F}_R^\eta(a_i)| \cdot \delta_R \tag{3.43}$$

Now consider the traversal delay of inter-application traffic. During the analysis, an inter-application message from the application a_i to the application a_j will be decomposed into 2 disjoint segments: the responsibility of a_i is to deliver the message to the proxy of a_j, while delivering it to its master is the responsibility of a_j. As already stated, the traversal latencies of supermessages related to inter-application messages have to be computed separately, as their sizes σ_{iam} may be different than those of protocol messages and contexts transfers, i.e. $\sigma_{iam} \neq \sigma_{prt} \neq \sigma_{ctx}$. The traversal delay of the supermessage involved in the inter-application communication is computed with Equation 3.44.

$$C_I(\widehat{f_i^k}) = |\mathscr{L}(\widehat{f_i^k})| \cdot \delta_L + (|\mathscr{L}(\widehat{f_i^k})| - 1) \cdot \delta_\rho + \left\lceil \frac{\sigma_{iam}}{\sigma_{flit}} \right\rceil \cdot \delta_L \tag{3.44}$$

Now, the traversal delay of all outgoing inter-application traffic of the application a_i is described with Equation 3.45. According to Theorem 7, each message between the master sender and its proxy may involve two supermessages.

$$C_{\#S}^\eta(a_i) = \sum_{\forall f_{i,j} \in \mathscr{F}_S^\eta(a_i)} \Big(\overbrace{2 \cdot C_I(\widehat{f_i^k})}^{\text{master to proxy}} + \overbrace{C_I(f_{i,j}^P)}^{\text{inter proxy}} \Big) \tag{3.45}$$

Equation 3.46 describes the traversal delay of all incoming inter-application traffic of the application a_i.

$$C^{\eta}_{\#R}(a_i) = \sum_{\forall f_{j,i} \in \mathscr{F}^{\eta}_{R}(a_i)} \overbrace{2 \cdot C_I(\widehat{f_i^k})}^{\text{proxy to master}} \tag{3.46}$$

3.5.2.4 Summary

In summary, between any two consecutive job releases of a_i, its supermessage $\widehat{f_i^k}$ can appear $\omega_P(\widehat{f_i^k})$ times during the protocol execution, $\omega_C(\widehat{f_i^k})$ times during the context transfer and $\omega_I(\widehat{f_i^k})$ times for every sent and received inter-application message. Furthermore, each proxy-to-proxy message $f^P_{i,j}$ may appear $\omega_I(f^P_{i,j})$ times due to inter-application traffic. Additionally, the maximum number of reroutings that occur on the router $\rho(d_i^j)$ of each dispatcher d_i^j is equal to the sum of reroutings during the protocol execution and during the inter-application communication – $r(d_i^j) = r_P(d_i^j) + r_I(d_i^j)$.

Moreover, the network delay of a_i when performing all communication (the agreement protocol and the inter-application traffic) can be computed by solving Equation 3.47, where the individual terms present the network delay of protocol-related messages, outgoing inter-application traffic and incoming inter-application traffic, respectively.

$$C^{\eta}_{\#}(a_i) = C^{\eta}_{\#P}(a_i) + C^{\eta}_{\#S}(a_i) + C^{\eta}_{\#R}(a_i) \tag{3.47}$$

Finally, the rerouting delay of a_i when performing all communication (the agreement protocol and the inter-application traffic), can be computed by solving Equation 3.48, where the individual terms present the rerouting delay of protocol-related messages, outgoing inter-application traffic and incoming inter-application traffic, respectively.

$$C^{\eta}_{R}(a_i) = C^{\eta}_{RP}(a_i) + C^{\eta}_{RS}(a_i) + C^{\eta}_{RR}(a_i) \tag{3.48}$$

3.5.3 Performing the Analysis

Recall, the worst-case communication delay of an application a_i, termed $R^{\eta}(a_i)$, computed by the existing path-abstracting method (Equation 3.14 for the List protocol and Equation 3.20 for the Hybrid protocol), consists of several components, which can be divided into three groups:

1. **Communication delay in isolation** – $C^{\eta}(a_i)$ (the first term in both Equation 3.14 and Equation 3.20).

2. **On-core interference** (the sum of the second, third and fourth term in Equation 3.14, and the sum of the second, third, fourth and fifth term in Equation 3.20). Let $I^{\eta}(a_i)$ denote the aforementioned sums.

3. **Network interference** – $I_{\#}^{\eta}(a_i, R^{\eta}(a_i))$ (the last term in both Equation 3.14 and Equation 3.20).

Thus, Equation 3.14 and Equation 3.20 can be rewritten as Equation 3.49.

$$R^{\eta}(a_i) = C^{\eta}(a_i) + I^{\eta}(a_i) + I_{\#}^{\eta}(a_i, R^{\eta}(a_i)) \tag{3.49}$$

As already mentioned, the approach presented in this section relies on enforcing placement constraints and reroutings, in order to make the traffic more predictable and deterministic. Therefore, assuming the new approach, the worst-case communication delay consists of two more components, specifically the rerouting delay in isolation, termed $C_R^{\eta}(a_i)$, and the rerouting interference, termed $I_R^{\eta}(a_i, R^{\eta}(a_i))$.

Now, the worst-case communication delay can be obtained by solving Equation 3.50.

$$R^{\eta}(a_i) = C^{\eta}(a_i) + C_R^{\eta}(a_i) + I^{\eta}(a_i) + I_{\#}^{\eta}(a_i, R^{\eta}(a_i)) + I_R^{\eta}(a_i, R^{\eta}(a_i)) \tag{3.50}$$

Note that $C^{\eta}(a_i)$ remains unaffected (Equation 3.13 for the List protocol, and Equation 3.19 for the Hybrid protocol), although its subcomponent corresponding to the message transfer delay in isolation $C_{\#}^{\eta}(a_i)$ is not any more computed with Equations 3.25-3.26, but with Equation 3.47. This is expected, because in the previous approach the messages were free point-to-point communication between dispatchers, while in the novel approach the message transfer has to be in compliance with Definition 4.

The rerouting delay in isolation $C_R^{\eta}(a_i)$ can be computed by solving Equation 3.48 (explained in the previous section).

The on-core interference component $I^{\eta}(a_i)$ remains unaffected, as the novel approach only affects the messages, not the communication-related OS operations that dispatchers perform.

The network interference component $I_{\#}^{\eta}(a_i, R^{\eta}(a_i))$ can now be computed in a different significantly less pessimistic way than Equation 3.7, due to the fact that the analysis will be performed on supermessages and proxy-to-proxy messages which are deterministic and master-independent. Note that obtaining a significantly less pessimistic value of this component was in fact the main incentive for these restrictions.

Finally, the rerouting interference component $I_R^{\eta}(a_i, R^{\eta}(a_i))$ is yet to be computed.

First, the focus will be on obtaining the network interference component. The application under analysis a_i can suffer the network interference from higher-priority supermessages and interproxy messages. Let $\mathscr{F}_D^{\eta}(a_i)$ be a set of all higher-priority messages that can cause direct interference to a_i. A message belongs to the set $\mathscr{F}_D^{\eta}(a_i)$ if it is directly contending with any message of a_i, where a direct contention between two messages has been introduced with Definition 1. Formally, $\mathscr{F}_D^{\eta}(a_i)$ is defined as follows:

$$\forall \widehat{f_q^w} \in \widehat{\mathscr{F}}, \exists \left(\widehat{f_i^j} \in \widehat{\mathscr{F}}(a_i) \vee f_{i,k}^P \in \mathscr{F}^P(a_i) \right) \mid \widehat{f_q^w} \in \left\{ \mathscr{F}_D(\widehat{f_i^j}) \cup \mathscr{F}_D(f_{i,k}^P) \right\} \Rightarrow \widehat{f_q^w} \in \mathscr{F}_D^{\eta}(a_i)$$

$$\wedge$$

$$\forall f^P_{m,n} \in \mathscr{F}^P, \exists \left(\widehat{f^j_i} \in \widehat{\mathscr{F}}(a_i) \vee f^P_{i,k} \in \mathscr{F}^P(a_i) \right) \mid f^P_{m,n} \in \left\{ \mathscr{F}_D(\widehat{f^j_i}) \cup \mathscr{F}_D(f^P_{i,k}) \right\} \Rightarrow f^P_{m,n} \in \mathscr{F}^\eta_D(a_i)$$

Now, the network interference that a_i might suffer from all messages from $\mathscr{F}^\eta_D(a_i)$ within the time interval t can be computed with Equation 3.51.

$$I^\eta_\#(a_i,t) = \overbrace{\sum_{\forall f^P_{m,n} \in \mathscr{F}^\eta_D(a_i)} \omega_I(f^P_{m,n}) \cdot C_I(f^P_{m,n})}^{\text{inter-application traffic}} \cdot \overbrace{\left(1 + \left\lceil \frac{t - D^{\tau+\mu}(a_m)}{T(a_m)} \right\rceil \right)}^{\text{number of inter-arrivals}} +$$

$$\sum_{\forall \widehat{f^w_q} \in \mathscr{F}^\eta_D(a_i)} \left(\overbrace{\omega_P(\widehat{f^w_q}) \cdot C_P(\widehat{f^w_q})}^{\text{protocol}} + \overbrace{\omega_C(\widehat{f^w_q}) \cdot C_C(\widehat{f^w_q})}^{\text{context transfer}} + \overbrace{\sum_{\forall f_{i,j} \in \mathscr{F}^\eta_S(a_i), \forall f_{j,i} \in \mathscr{F}^\eta_R(a_i)} \omega_I(\widehat{f^w_q}) \cdot C_I(\widehat{f^w_q})}^{\text{inter-application traffic}} \right) \cdot$$

$$\overbrace{\left(1 + \left\lceil \frac{t - D^{\tau+\mu}(a_q)}{T(a_q)} \right\rceil \right)}^{\text{number of inter-arrivals}} \tag{3.51}$$

Recall that inter-proxy messages participate only in the inter-application traffic, hence there is only one interference-related term in the first row of Equation 3.51. Conversely, supermessages participate in the protocol messages, in the context transfers and in the inter-application traffic, and hence there are three interference-related terms in the second row of Equation 3.51. Also recall that $\mathscr{F}^\eta_S(a_i)$ and $\mathscr{F}^\eta_R(a_i)$ denote sets of sent and received messages of the application a_i. Note that inter-application messages may have different sizes, so their occurrences cannot be summed up, but have to be computed independently. Hence the inner summation in the second row of Equation 3.51.

The last outstanding term is the rerouting interference, and it can be computed by solving Equation 3.52.

$$I^\eta_R(a_i,t) = \sum_{\forall d^j_i \in \mathscr{D}(a_i)} \sum_{\forall d^m_k \in \mathscr{D}_{\pi(d^j_i)} \mid d^m_k \neq d^j_i} r(d^m_k) \cdot \overbrace{\left(1 + \left\lceil \frac{t - D^{\tau+\mu}(a_k)}{T(a_k)} \right\rceil \right)}^{\text{number of inter-arrivals}} \cdot \delta_R \tag{3.52}$$

3.5.4 Discussion

The previous, path-abstracting method for the worst-case communication delay analysis, presented in Sections 3.3-3.4, does not involve reroutings, and hence does not have rerouting-related components $C^\eta_R(a_i)$ and $I^\eta_R(a_i,t)$ (compare Equations 3.49-3.50). However, the limitation of that approach is that it treats the path non-determinism in a pessimistic way, and computes the interference on the application level. Conversely, the method presented in this section employs dispatcher placement and rerouting constraints, in order to express the traffic as a function of supermessages, inter-proxy messages and reroutings. This strategy imposes longer message distances and employs a rerouting

mechanism, which both additionally contribute to the worst-case communication delay. However, this approach makes message paths deterministic and known at design-time. Consequently, the fine-grained analysis can be performed on the message-level, and the interference component can be computed with much less pessimism, which is a fundamental requirement for a tight worst-case communication delay analysis. This claim is also backed up with the recent insights in the priority-preemptive wormhole-switched NoCs [74], which suggest that the most dominant factor in the worst-case analysis is indeed the interference component.

3.5.5 Experimental Evaluation

This section describes the evaluation of the newly proposed approach. Specifically, the objective is to find answers to the following questions:

- How does the novel method for the worst-case analysis compare against the previous method, presented in Sections 3.3-3.4? See **Experiment-Set 1**
- What is the penalty for enforcing the traffic determinism via reroutings and supermessages, how does it affect the performance, and is it justifiable? See **Experiment-Set 2**.
- How pessimistic is the novel method, i.e. how the analytically obtained upper-bound estimates compare against the corresponding values observed during simulations? See **Experiment-Set 3**.

3.5.5.1 Evaluation Metrics, Analysis and Simulation Parameters

In Experiment-Set 1, the novel method for the worst-case analysis is compared against the existing method, presented in Sections 3.3-3.4. Specifically, for each application a_i, the estimates on the worst-case communication delay are obtained with the novel method $R^{\eta}_{new}(a_i)$, and the existing method $R^{\eta}_{old}(a_i)$. Then, the improvements of the novel approach are computed with the following metric: $imp = \frac{R^{\eta}_{old}(a_i)-R^{\eta}_{new}(a_i)}{R^{\eta}_{old}(a_i)}$. In cases where the new method underperforms, the penalty has a negative value, and it is expressed with the following metric: $pen = \frac{R^{\eta}_{old}(a_i)-R^{\eta}_{new}(a_i)}{R^{\eta}_{new}(a_i)}$.

In Experiment-Set 2, the workload execution is simulated. The simulations are performed on the extended version of the SPARTS [73] simulator. Two different scenarios are simulated: (i) the communication is free point-to-point, which corresponds to the approach presented in Sections 3.3-3.4, and (ii) the reroutings are employed, in order to comply with Definition 4. For each application a_i, two worst-case delays are captured, one for the first simulated scenario (the existing approach) $R^{\eta}_{old*}(a_i)$, and one for the second simulated scenario (the novel approach) $R^{\eta}_{new*}(a_i)$. Similarly to Experiment-Set 1, the improvements and the penalty were expressed with the following metrics: $imp^* = \frac{R^{\eta}_{old*}(a_i)-R^{\eta}_{new*}(a_i)}{R^{\eta}_{old*}(a_i)}$, and $pen^* = \frac{R^{\eta}_{old*}(a_i)-R^{\eta}_{new*}(a_i)}{R^{\eta}_{new*}(a_i)}$.

In Experiment-Set 3, assuming the novel approach, the analytic estimates that were obtained with Experiment-Set 1 are compared against the corresponding worst-case delays observed during simulations in Experiment-Set 2. Then, the following metric is used to express the tightness of the new method: $tgh = \frac{R^{\eta}_{new*}(a_i)}{R^{\eta}_{new}(a_i)}$.

The analysis and simulation parameters are summarised in Table 3.4. An asterisk sign denotes a randomly generated value, assuming a uniform distribution.

Table 3.4: Analysis and simulation parameters for Section 3.5.5

NoC topology and size	**2-D mesh with 10 × 10 routers**
Link width = flit size σ_{flit}	**16 bytes**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Rerouting delay δ_R	**10000 cycles (5 μs)**
OS operations $\delta_P^{\leftarrow}$, $\delta_P^{\rightarrow}$, $\delta_C^{\leftarrow}$, $\delta_C^{\rightarrow}$, $\delta_I^{\leftarrow}$, $\delta_I^{\rightarrow}$, δ_Q, δ_E	**10000 cycles (5 μs)**
Application periods $D(a_i) = T(a_i), \forall a_i \in \mathscr{A}$	$[\mathbf{100} - \mathbf{1000}]^*$ **ms**
Communication deadlines $D^\eta(a_i), \forall a_i \in \mathscr{A}$	$\mathbf{0.25 \cdot D(a_i)}$
Protocol message size σ_{prt}	**16 bytes**
Execution context size σ_{ctx}	$[\mathbf{1} - \mathbf{128}]^*$ **Kbytes**
Inter-application message size σ_{iam}	$[\mathbf{1} - \mathbf{128}]^*$ **Kbytes**
Application-set size $\lvert\mathscr{A}\rvert$	**200 applications**
Maximum concurrent on-core masters $\widehat{M}$	**10**
Probability of inter-app. comm.	**5%**
Simulated time	**100 s**

3.5.5.2 Experiment-Set 1: Analyses Comparison

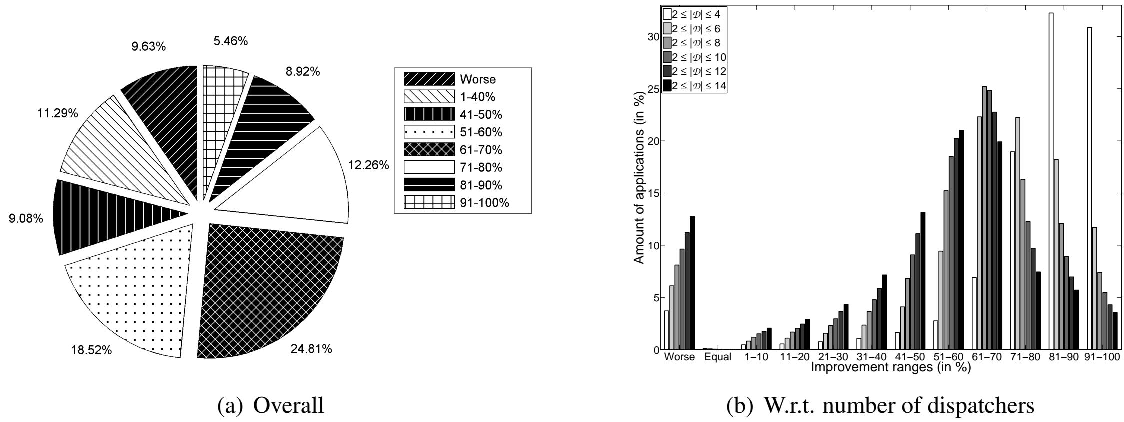
Experiment 1a: Overall Improvements

In this experiment an overall analytic comparison of the novel method and the existing one is conducted. Specifically, each application has $[2-10]^*$ dispatchers, and is randomly mapped on the grid with an arbitrary (rectangular or line-like) shape, assuming dispatcher placement constraints (Definition 3). Half of the applications execute the List protocol, and the other half Hybrid. Then, the analytic upper-bound on $R^\eta(a_i)$ is obtained for each application a_i, with both approaches. Finally, the obtained values are compared, where the improvements of the novel approach are computed with the metric described in Section 3.5.5.1. The process is repeated for 1000 application-sets.

Figure 3.11(a) shows the improvements of the new approach over the existing one. In only 9.63% of cases the novel method rendered worse results. This is further investigated in Experiment 1c. In the remaining 90.37% of scenarios the novel approach reports improvements. Specifically, in more than half of cases the improvements are greater than 50%, which means that a derived upper-bound is at most half the one obtained with the existing approach. Finally, in 5.46% of scenarios the improvements are above 90%, corresponding to an estimate that is at most **one-tenth** of the value against which it is compared! That is, $\boldsymbol{R^\eta_{new}(a_i) \leq \frac{1}{10} \cdot R^\eta_{old}(a_i)}$.

Experiment 1b: Improvements w.r.t. Number of Dispatchers

In order to test the scalability of the novel approach, and in order to see how the improvements change with the amount of dispatchers, the number of dispatchers constituting each application



(a) Overall

(b) W.r.t. number of dispatchers

Figure 3.11: Analysis improvements (1/2)

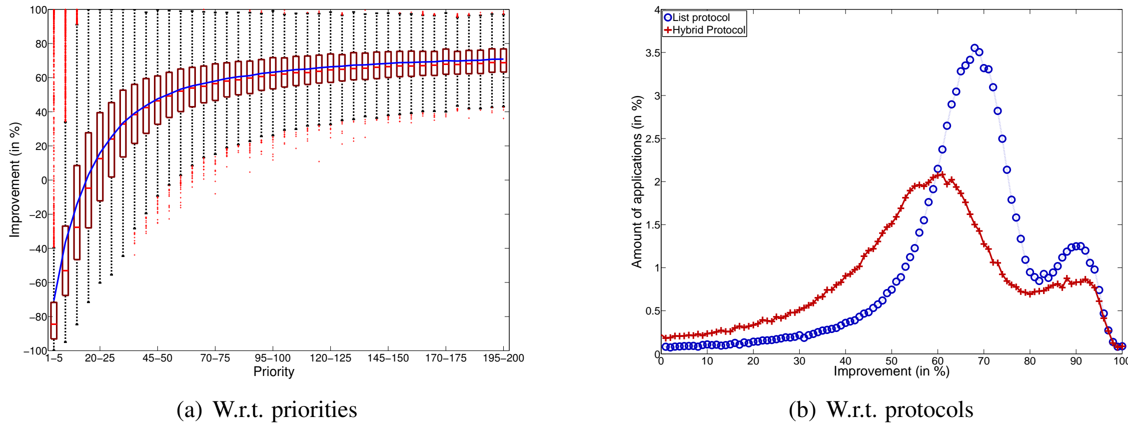
is varied within the following range $|\mathscr{D}(a_i)| = [2-x]^*, \forall a_i \in \mathscr{A}$. The parameter x is varied in the range $x \in \{4, 6, 8, 10, 12, 14\}$ (see the legend of Figure 3.11(b)). Assuming a certain range (e.g. $|\mathscr{D}(a_i)| = [2-4]^*$), all applications are generated and randomly mapped. Again, half of the applications execute each of the protocols. Then, for each application a_i, the analytic $R^{\eta}(a_i)$ upper-bound estimates are obtained with (i) the existing method and (ii) the novel method. Subsequently, the obtained values are compared with the same metric as in the previous experiment. This is repeated for 1000 application-sets, and for every range (Figure 3.11(b)).

As the number of dispatchers increases, so does the category with worse results and the categories with smaller improvements (until 60%), while other categories with more significant improvements (above 60%) decrease. The explanation is twofold. First, assuming the novel approach, more dispatchers cause more reroutings. This in turn causes more significant rerouting interferences, which have an impact on the derived estimates. Conversely, in the existing approach reroutings do not occur. Additionally, more dispatchers cause more messages, leading to more significant and complex message interference scenarios. In such cases, making an assumption that every higher priority message existing within the network will indeed cause interference might not be too pessimistic. Thus, as the number of dispatchers increases, the existing method becomes less pessimistic and hence improvements achieved by the new method are slowly decreasing.

Experiment 1c: Improvements w.r.t. Priorities

In this experiment, the improvements of the novel method over the existing one are observed, with an emphasis on application priorities. This experiment also helps to recognise and investigate the cases where the new method underperforms. The values obtained in Experiment 1a are used, but the comparison between the approaches is additionally performed per-priority, as depicted in Figure 3.12(a). The metric is the same as in the previous experiments.

It is evident that the novel approach performs worse for applications with higher priorities (smaller numbers on the x-axis of Figure 3.12(a)). As these applications do not suffer significant interference, the additional delay in the new approach is caused by reroutings and longer message



(a) W.r.t. priorities (b) W.r.t. protocols

Figure 3.12: Analysis improvements (2/2)

distances (i.e. traversal constraints expressed by Definition 4). As priorities decrease, the interference becomes a more dominant term in the derived upper-bounds, hence the penalty of the novel approach slowly decays. At the priority level 15, both the approaches provide similar results. Any additional decrease in the priority favours more the new method, where improvements report logarithmic growth. On the far right end of the domain, the average improvements of the novel approach asymptotically converge towards 70%.

Experiment 1d: Improvements w.r.t. Protocols

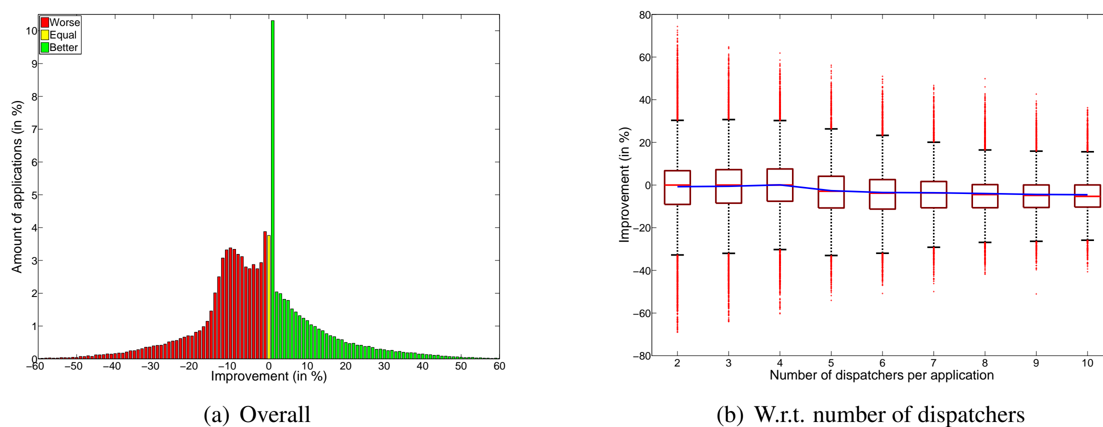
In this experiment, the novel and the existing approach are compared with an emphasis on the employed agreement protocols. The comparison is performed for 2 different scenarios: (i) all applications are utilising the List protocol, (ii) all applications are utilising the Hybrid protocol. For each application a_i, the $R^{\eta}(a_i)$ value is obtained with both approaches. The process is repeated for 1000 application-sets. The results are given in Figure 3.12(b), where the improvements achieved by the novel approach are expressed with the same metric used in the previous experiments.

The conclusions are very similar to those for Experiment 1b. The Hybrid protocol involves more messages, which in the novel approach induce more reroutings. Hence, larger improvements are reported for the List protocol than for the Hybrid protocol. Additionally, as the improvement metric is based on the ratio, similarly to the logarithmic scale, each additional improvement percentage covers larger part of the domain; e.g. for improvements of $49-50\%$ the following holds: $R^{\eta}_{old}(a_i) \in \{1.96 \cdot R^{\eta}_{new}(a_i), 2 \cdot R^{\eta}_{new}(a_i)\}$, while for improvements of $89-90\%$ the following holds: $R^{\eta}_{old}(a_i) \in \{9.09 \cdot R^{\eta}_{new}(a_i), 10 \cdot R^{\eta}_{new}(a_i)\}$. Due to that fact, improvement ranges around 90% cover large parts of the domain, and cause local maximums (Figure 3.12(b)).

3.5.5.3 Experiment-Set 2: Performance Comparison

Experiment 2a: Overall Comparison

The purpose of this experiment is to perform an overall runtime performance comparison of the novel and the existing approach. In order to do that, the application-sets generated for



(a) Overall (b) W.r.t. number of dispatchers

Figure 3.13: Performance comparison (1/2)

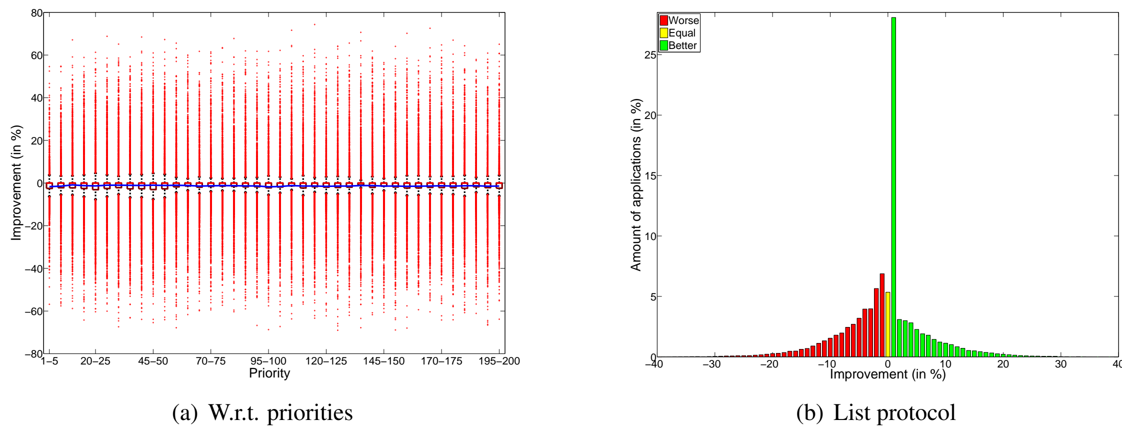
Experiment 1a are used and the simulations are performed for two different scenarios: with and without traversal constraints (Definition 4). Within each approach, $R^{\eta}_{*}(a_i)$ is measured for each application a_i of the application-set. Consequently, the obtained values are compared with the metric described in Section 3.5.5.1, and the results are presented in Figure 3.13(a).

Since the novel approach causes longer message distances and employs the rerouting mechanism, an intuitive guess would be that it will systematically suffer a significant runtime performance penalty, when compared with the existing method. However, a surprising conclusion is reached: **not only was the penalty negligible in almost all cases, but also for almost 40% of the scenarios the novel approach outperformed the existing one!** These unexpected findings are interpreted in the following way. Unpredictable message paths may lead to corner cases where a traffic becomes heavily concentrated in certain links of the grid, resulting in significant contentions and (almost) unbounded interference delays. This reflects the importance of having predictable message-paths, and further implies that the overall efficiency of the system heavily depends on the application-mapping process, which will be covered in the next section.

Experiment 2b: Comparison w.r.t. Number of Dispatchers

In this experiment, the runtime performance of the two approaches is investigated, assuming variable dispatcher numbers. The values obtained in the previous experiment are used, but before performing the comparison, the applications are divided into different categories, based on the number of dispatchers. Then, the comparison is performed for each category, independently. The results are depicted in Figure 3.13(b).

It is visible that for applications with $2-4$ dispatchers, where reroutings do not occur, in almost half of the cases the novel approach dominates the existing one. However, for applications with more dispatchers and rectangular shapes, the reroutings are necessary, which is reflected with a slight decrease of the average line in Figure 3.13(b), as the number of dispatchers increases. Nonetheless, even for applications with 10 dispatchers, there are still numerous cases where the new approach performs better. In cases where the novel approach underperforms, the perfor-



(a) W.r.t. priorities

(b) List protocol

Figure 3.14: Performance comparison (2/2)

mance loss is incomparably smaller than the gains achieved with the novel analysis method (see Experiment-Set 1).

Experiment 2c: Comparison w.r.t. Priorities

The objective of this experiment is to investigate how the runtime performance of the two approaches changes with the application priorities. The results from Experiment 2a are used, but before comparing the obtained R_*^η values, applications are classified into different categories, based on their priorities. Then, the comparison is performed for each category, independently. Figure 3.14(a) illustrates the results.

Figure 3.14(a) shows that, in the majority of cases, irrespective of the priority, both the approaches display very similar performance. It is barely noticeable that only for the highest priorities, the novel approach underperforms slightly more than in other cases. This occurs, because in these scenarios the existing method is not pessimistic (no network interference), while the existing method still pays the penalty of rerouting operations and longer message paths.

Experiment 2d: Comparison w.r.t. Protocols

In this experiment the runtime performance of the two approaches is observed, assuming different agreement protocols. The application-sets generated for Experiment 1d are used, where, in the first case, all applications executed the List protocol, and in the second case all application executed the Hybrid protocol. First, assuming the application-sets where all applications execute the List protocol, the simulations are performed, and subsequently for each application a_i the $R_{old*}^\eta(a_i)$ and $R_{new*}^\eta(a_i)$ values are obtained. Then, the obtained values are compared. The results are illustrated in Figure 3.14(b). Finally, the same process is repeated for the application-sets where all applications execute the Hybrid protocol. The results for that case are illustrated in Figure 3.15.

The explanation is similar to that for Experiment 1d. The List protocol involves less messages and less rerouting operations, which favours more the novel approach. Indeed, from Figure 3.14(b) it is visible that in more than half of the cases the new method outperforms the existing one. Conversely, the Hybrid protocol involves more messages and more rerouting operations, both of which

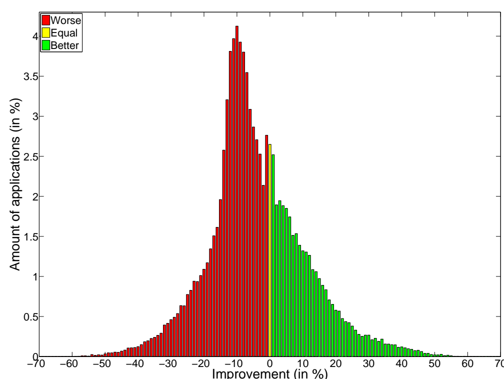


Figure 3.15: Performance comparison for Hybrid protocol

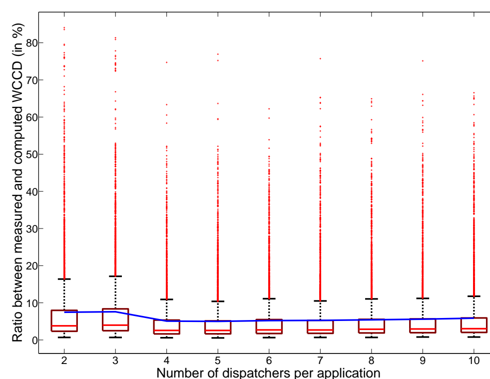


Figure 3.16: Analysis tightness across number of dispatchers

have a negative impact on the worst-case delays obtained for the novel approach. Consequently, the new approach performs better than the existing one in only 30% of the cases (see Figure 3.15).

3.5.5.4 Experiment-Set 3: Analysis Tightness

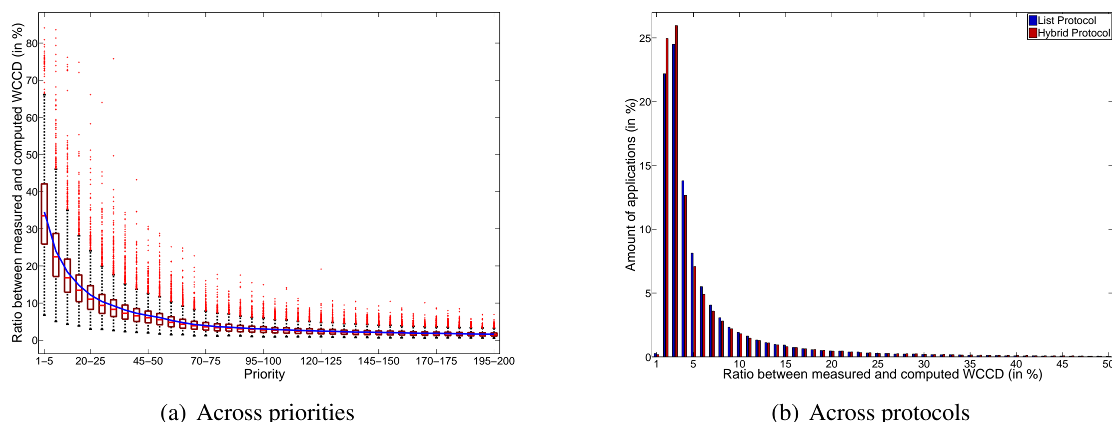
Experiment 3a: Tightness w.r.t. Number of Dispatchers

In this experiment the objective is to observe the pessimism of the novel method for the worst-case communication delay analysis. In order to do that, the results from Experiment 1a and Experiment 2a are used, where for each application a_i the values of $R^{\eta}_{new}(a_i)$ and $R^{\eta}_{new*}(a_i)$ were computed. Before performing the comparison, the applications are classified into categories, based on the number of dispatchers. Then, the comparison is performed for each category, independently. The results are depicted in Figure 3.16, where the metric described in Section 3.5.5.1 is used.

It is visible that for the applications with fewer dispatchers, there exist cases where the ratio between the observed and the analytically computed worst-case communication delay is very high. Indeed, in the category of two-dispatcher applications, there exists an application with the ratio around 85%, which suggests that the analysis is correct and renders very tight estimates. As the number of dispatchers increases, the interference patterns become more complex, and the scenarios considered by the worst-case analysis are less likely to be captured at runtime. This is manifested with a slight decrease in the ratio, for the higher number of dispatchers. However, the decrease is not significant, and there exist applications with 10 dispatchers, for which the ratio between R^{η}_{new*} and R^{η}_{new} is around 65%.

Experiment 3b: Tightness w.r.t. Priorities

The goal of this experiment is to observe how the ratio between R^{η}_{new*} and R^{η}_{new} changes with application priorities. For that, the same values as in the previous experiment are used (the results from Experiment 1a and Experiment 2a). However, this time the applications are divided into categories based on their priorities. The comparison is performed for each category, independently. Figure 3.17(a) illustrates the findings.



(a) Across priorities

(b) Across protocols

Figure 3.17: Analysis tightness

It comes as no surprise that the highest observed ratio from the previous experiment (around 85%) was in this experiment measured for the application with the highest priority (the smallest numbers on the x-axis of Figure 3.17(a)). In other words, the ratio of 85% belongs to the two-dispatcher application with the highest priority. This implies that the highest ratio occurs for high-priority applications with few dispatchers. Furthermore, the results demonstrate an average ratio of around 35% for applications with high priorities. As the priorities decrease, so does the ratio, which asymptotically converges towards 2% for applications with very low priorities. The explanation is as follows. As the priority decreases, the analysis considers more and more complex interference (worst-case) scenarios, which are less and less likely to occur during simulations.

The experiments demonstrate that there is a small probability for lower-priority applications to encounter analytically identified worst-case scenarios. Having that in mind, together with the fact that in many practical scenarios occasional missed deadlines of lower-priority applications are in fact tolerable, several questions could be raised. For instance, **is the worst-case analysis the right approach to treat the lower-priority applications?** Or should some other (e.g. probabilistic) techniques be applied? These questions represent good starting points for the future work.

Experiment 3c: Tightness w.r.t. Protocols

In this experiment the objective is to investigate how the ratio between R^{η}_{new*} and R^{η}_{new} changes with the agreement protocols. In order to do that, the results from Experiment 1d and Experiment 2d are used, where R^{η}_{new*} and R^{η}_{new} are obtained for two different scenarios: (i) all applications execute the List protocol, and (ii) all applications execute the Hybrid protocol. The comparison is performed for each protocol, independently. The results are depicted in Figure 3.17(b).

It is visible that the ratio is higher for the List protocol, which entirely coincides with the findings of Experiment 1d and Experiment 2d. Specifically, the Hybrid protocol involves more messages and reroutings, both of which lead towards more complex interference patterns. Consequently, the analytically identified worst-case scenarios are less likely to occur at runtime, leading to lower ratios. Conversely, the List protocol involves less messages and reroutings. This infers

that identified worst-case scenarios involve less complex interference patterns, which are more likely to be captured at runtime, leading to higher rations between R^{η}_{new*} and R^{η}_{new}.

3.5.5.5 Discussion

In this section, a novel method for the worst-case communication delay analysis of *LMM* was proposed. The proposed approach potentially "sacrifices" the performance (via supermessages and proxies), in order to gain in predictability (i.e. deterministic message paths). The novel method was compared with the existing one, both in terms of the analysis and the performance. The experiments demonstrate that the novel approach not only renders tighter upper-bound estimates in more than 90% of the cases, but also demonstrates a comparable runtime performance, which reflects the importance of having deterministic traffic routes.

3.6 Application Mapping

Notice that the efficiency of the aforementioned analyses highly depends on dispatcher positions. In other words, the positions of individual dispatchers affect all terms contributing to the worst-case communication delay (Equations 3.49-3.50), irrespective of the employed analysis. Specifically, the communication delay in isolation, the on-core interference, the network interference all depend on the positions of dispatchers of the analysed application. Therefore, the application mapping is an important aspect of the worst-case analysis of *LMM* and it should be addressed.

It comes as no surprise that the application mapping for many-cores has been one of the most investigated topics over the last decade, thus resulting in a vast amount of works. In this dissertation only a brief overview of the state-of-the-art methods will be given, whereas only the works that are closely related to the real-time embedded domain will be covered. An interested reader may consult a comprehensive survey of Sahu and Chattopadhyay [82], which covers scientific publications related to the entire application mapping topic.

The problem of application mapping is equivalent to the quadratic assignment problem, which is NP-Hard, hence searching for the optimal solution can be prohibitively expensive even for small NoCs [40], e.g. 4×4. Therefore, the existing approaches are predominantly heuristics-based. Lei and Kumar assumed different processor types and developed a two-stage genetic algorithm [56], where the workload is firstly mapped to a specific processor type and in the second pass to a particular processor. Moein et al. [64] use a modification of the aforementioned heuristics, called the chaos-genetic algorithm. By employing the branch-and-bound technique Hu and Marchulescu [40] investigate mappings which minimise the power consumption within the network. The authors present a power model to calculate the energy spent by the NoC infrastructure, which is used as an objective function to evaluate derived mappings. The concept of minimising the energy consumption was further extended to include performance enhancements [22], network contentions [21] and runtime mappings [20]. Murali and De Micheli [65] elaborate on bandwidth constraints and minimise the average communication delay, while Hung et al. [41] study thermal-aware placements. Marcon et al. [61] are the first to consider the ordering and dependencies among

traffic messages, while Srinivasan and Chatha [91] introduce per-message latency constraints. Ascia et al. [5] present an approach where not one, but a group of mappings is considered (*Pareto mappings*), so as to derive a solution to the multi-objective approach. Kreutz et al. [51] explore the same topic for interconnect mediums other than NoC, while Hu and Marculescu [39] exploit the routing flexibility in order to reduce the routing energy consumption and improve the performance.

As already mentioned in Chapters 1-2, Shi and Burns proposed two worst-case communication delay analyses for wormhole-switched priority-preemptive NoCs [85, 86]. The former analysis was combined with genetic algorithms with the objective of mapping hard real-time applications [62, 80], while the latter approach was further combined with a priority assignment algorithm with the same aim [87]. Note that the studies mentioned in this paragraph are the most related to the topics covered in this dissertation. That is, the authors of these works analyse the application-mapping problem from the real-time perspective, where the main emphasis is on deriving a mapping such that all temporal constraints posed on communication delays are always fulfilled, even under the worst-case conditions. Also note that these approaches rely on the assumption that each application is statically assigned to a specific core at design-time, and does not have the possibility to migrate (see fully partitioned approaches in Section 1). Conversely, in this dissertation of interest is the application mapping problem for *LMM*.

3.6.1 Problem Statement

The objective is to propose an application mapping method which finds a mapping of a given application workload onto a given platform ($\mathscr{M} = \mathscr{A} \rightarrow \Psi$) which is schedulable with respect to communication requirements. A mapping $\mathscr{M}$ is schedulable if all applications are schedulable, i.e. $R^{\eta}(a_i) \leq D^{\eta}(a_i), \forall a_i \in \mathscr{A}$.

The secondary objective is to provide a schedulable mapping, such that the applications can perform migrations across spatially distributed dispatchers. This objective is motivated with the fact that far migrations are the efficient means to implement energy/thermal management, and increase the resilience to core/cluster malfunctions. To address this aspect, the definition of applications is extended with one additional property called the *migration coefficient* $M(a_i)$, which reflects the importance of a distributed mapping of the application a_i, and its purpose will be explained later.

3.6.2 Mapping Quality

The multi-objective nature of the proposed approach is reflected by the fact that the goal is not just to provide a mapping which is schedulable, but to provide a schedulable mapping which maximises the abilities of the application-set to perform runtime load balancing via application migrations. Two aspects are identified as the most relevant for the migrative potential of an application, namely the *dimensions* of the rectangular $x \times y$ shape its dispatchers are forming and the *distribution* of the dispatchers on that shape. The strategy that is used to evaluate the mapping of the application can be summarised with the following two observations:

Observation 9. *The greater the shape of the application is, the greater its migrative abilities are.*

Observation 10. *Assuming a fixed shape, the more even the distances between the application dispatchers are, the greater its migrative abilities are.*

The reasoning is that, given that the migration has to occur, it is better to accommodate the next job execution on some far core, rather than on some near core, because far migrations allow efficient global load balancing, energy/thermal management and make the system more resilient to clustered failures (i.e. a part of the chip starts malfunctioning). Conversely, near migrations only partially solve these issues, or don't solve them at all. Thus, the intention behind this approach is to prevent near migrations by maximising the shape of the application (Observation 9), and distributing its dispatchers on the shape, such that the inter-dispatcher distances are as even as possible (Observation 10).

Although it may seem that these objectives superficially contradict the schedulability requirement, this is not entirely true. If perceived as an optimisation problem, migrative abilities of applications are the objective function which has to be maximised, and the schedulability is a constraint which must be fulfilled. Subsequently, the solution should be found such that (i) all the applications are schedulable, (ii) the dimensions of the application shapes are as big as possible, (iii) the distances between the dispatchers of the same application are as equal as possible.

In order to qualitatively evaluate different mappings, a proper metric has to be established such that it implements the aforementioned reasoning. Let (d_i^j, d_i^k) denote a pair of neighbouring dispatchers of one application, and let $hops(d_i^j, d_i^k)$ be the distance between them, expressed in hops. Moreover, let $\mathscr{D}_P(a_i)$ denote all such pairs of neighbouring dispatchers of the application a_i. A quality of a mapping of the application a_i, denoted by q_i (Equation 3.53), is equal to the product of the distances between its neighbouring dispatchers, multiplied by its migration coefficient $M(a_i)$.

$$q_i = M(a_i) \cdot \prod_{\forall (d_i^j, d_i^k) \in \mathscr{D}_P(a_i)} hops(d_i^j, d_i^k) \tag{3.53}$$

The migration coefficient $M(a_i)$ symbolises the importance of application's spatially distributed mapping. In other words, the more the system would benefit from the application's spatially distributed mapping, the greater this parameter is. For example, a computationally demanding application may have a significant impact on the thermal properties of the core where it is executing (and its surrounding), therefore, having the possibility to perform far migrations of that application is desirable from the thermal perspective. Subsequently, for every such application a_i its coefficient $M(a_i)$ should be set high. Similarly, allowing far migrations for a critical application may improve its resilience towards core/cluster failures. Thus, each such application a_i should have its $M(a_i)$ coefficient set high. In this dissertation it is assumed that the values of migration coefficients have already been specified, and that the same are used as a means to classify the applications according to the importance of their spatially distributed mappings.

There are three reasons why a product of inter-dispatcher distances is used as the evaluation metric:

• Primarily, because it is a computationally cheap operation. That is, due to the rectangular or line-like structure of the application shape (Definition 3), the inter-dispatcher distances can be obtained in a single traversal across the circumference of the application's shape in either clockwise or couter-clockwise direction. Since during the mapping process the evaluation of many different shapes of all applications will be performed, it is of paramount importance to limit its complexity.
• Second, the product of inter-dispatcher distances is monotonically increasing with the shape size (see Observation 9).
• Finally, when assuming that the shape has been decided, the product of inter-dispatcher distances reaches the maximum when all inter-dispatcher distances are as even as possible (see Observation 10 and for the formal proof see Theorem 17 in Appendix).

Note that a zero-distance between dispatchers presents a special case where the same are located on a common core. In the assumed model, such a mapping is meaningless and the metric expressed with Equation 3.53 also penalises such mappings, hence returning $q_i = 0$.

Assuming that the circumference of an application shape c and the number of dispatchers $|\mathscr{D}(a_i)|$ are given, choosing and applying an optimal dispatcher placement would be trivial: if possible - make all distances equal $\frac{c}{|\mathscr{D}(a_i)|}$, otherwise make distances either $\left\lceil \frac{c}{|\mathscr{D}(a_i)|} \right\rceil$ or $\left\lfloor \frac{c}{|\mathscr{D}(a_i)|} \right\rfloor$. However, the corners of the application shape pose implicit constraints regarding the inter-dispatcher distances and, in some cases, prevent optimal solutions. Also, not all the cores located on the edges of the application shape might be available due to on-core schedulability reasons (Section 3.8), thus further preventing optimal solutions. Therefore, the purpose of the aforementioned evaluation metric is to provide a qualitative comparison between different possible suboptimal dispatcher placements, in cases when optimal ones are not possible.

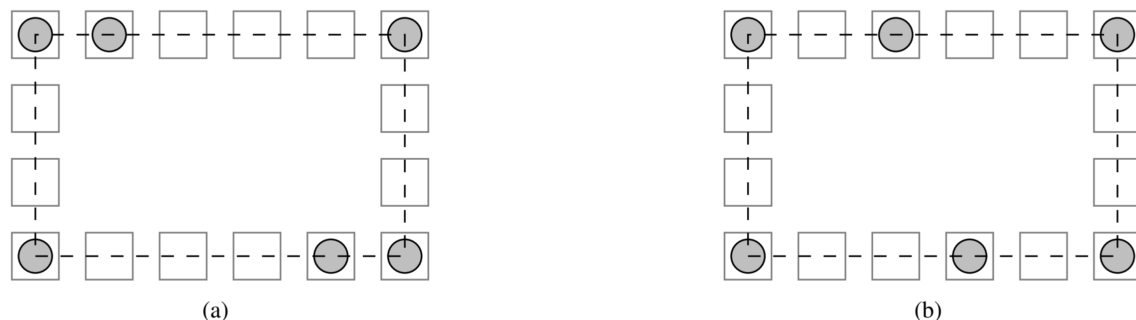


Figure 3.18: Different dispatcher mappings for the same application-shape

The example illustrated in Figure 3.18 is used to demonstrate how different mappings are evaluated. Two possibilities are presented, in both of them the application claims the same shape, but the placement of the dispatchers differs. For clarity purposes, assume that the migration coefficient is equal to 1. After solving Equation 3.53, the value of q_i for Figure 3.18(a) is 144, while for the mapping illustrated in Figure 3.18(b), $q_i = 324$. Thus, the mapping from Figure 3.18(b) is a more favourable option. This coincides with the reasoning and intentions. Notice that for a given example $c = 16, |\mathscr{D}(a_i)| = 6$, and $\frac{c}{|\mathscr{D}(a_i)|} = \frac{16}{6} = 2.67$. Therefore, the mapping from Figure 3.18(b) is one of optimal solutions, due to the fact that its all dispatcher distances are either 2 or 3.

Upon defining the individual, per-application quality metric, the quality of the mapping of an entire application-set can be defined. It is equal to the sum of individual qualities of all applications comprising an application-set $\mathscr{A}$ (Equation 3.54).

$$Q = \sum_{\forall a_i \in \mathscr{A}} q_i \tag{3.54}$$

3.6.3 Mapping Process

The proposed method consists of three mapping stages: the *Initial Phase* ($\mathscr{IP}$), the *Feasibility Phase* ($\mathscr{FP}$) and the *Optimisation Phase* ($\mathscr{OP}$).

Before the mapping process begins, based on their priorities, all applications are divided into two groups: a set of high-priority applications $\mathscr{A}_H$, and a set of low-priority applications $\mathscr{A}_L$ (Equation 3.55).

$$\begin{cases} P(a_i) > P^{min} + P \cdot (P^{max} - P^{min}) \Rightarrow a_i \in \mathscr{A}_H \\ P(a_i) \leq P^{min} + P \cdot (P^{max} - P^{min}) \Rightarrow a_i \in \mathscr{A}_L \end{cases} \tag{3.55}$$

P^{max} and P^{min} denote the maximum and the minimum system priorities, respectively, while the parameter P $(0 \leq P \leq 1)$ is arbitrarily chosen by the system designer. All applications belonging to the group $\mathscr{A}_H$ will be mapped during $\mathscr{IP}$ and will not be subject to any changes during $\mathscr{FP}$ and $\mathscr{OP}$. The rationale for this decision is that any change in the mapping of an application $a_i \in \mathscr{A}_H$ (e.g. its shape, the position of its dispatchers) requires a new schedulability check of both a_i and all lower priority applications which messages are directly interfered by the messages of a_i. Thus, the recalculation triggered by a high-priority application might be computationally expensive, as there might be a substantial number of directly interfered applications. Therefore, the parameter P is introduced as a means to control the complexity of the entire mapping process.

3.6.3.1 Initial Phase ($\mathscr{IP}$)

As already described, during this phase only the high-priority applications are mapped. The applications are sorted non-increasingly with respect to their priorities, and mapped sequentially in that order. By performing the mapping in this manner, in most cases, the mapping of one application will not have an impact on the schedulability of previously mapped (higher-priority) applications. Exceptions are cases when there exists an inter-application message from some already mapped higher priority application a_j, to the currently mapping application a_i. In such cases, the mapping of a_i also maps the inter-proxy message $f^P_{j,i}$ which has the priority of a_j. This invokes a schedulability recheck of a_j and all interfered applications with intermediate priorities. Depending on the frequency of these situations, the complexity of $\mathscr{IP}$ ranges between $O(|\mathscr{A}_H|)$ (no schedulability rechecks necessary) and $O(|\mathscr{A}_H|^2)$ (the mapping of every application invokes schedulability rechecks), where $|\mathscr{A}_H|$ denotes the amount of applications in $\mathscr{A}_H$. As will be seen in Section 3.6.4, the actual complexity is closer to the former estimate (a linear complexity).

When mapping an application, several mapping options are possible. A *narrow mapping* presents a mapping where an application shape covers the minimum possible surface, i.e. a mapping where all dispatchers of an application occupy consecutive cores. For instance, the possible narrow mappings for a 4-dispatcher application are the following shapes: $1\times4, 4\times1, 2\times2$. Conversely, a *wide mapping* is a mapping where an application shape covers the maximum possible surface, in most cases the boundaries of the NoC. The surfaces of the narrow and the wide mapping are referred to as $S^{min}(a_i)$ and $S^{max}(a_i)$, respectively.

Since the applications from $\mathscr{A}_H$ are mapped during $\mathscr{I}\mathscr{P}$, and are not subject to any changes in the subsequent mapping phases, it is essential to dedicate a proper shape to each of these applications. Assuming wide mappings for most applications during $\mathscr{I}\mathscr{P}$ will create significant high priority traffic within the network, which might cause the applications from $\mathscr{A}_L$ to be unable to reach the schedulability. Conversely, restricting the applications from $\mathscr{A}_H$ to claim the narrow mappings might unnecessarily preserve the network resources underutilised, as there might be very few applications in $\mathscr{A}_L$. In order to manage this design trade-off, the parameter G id introduced. G presents an upper-bound on the allowed application shapes, and it controls the mapping "greediness" of high-priority applications. For instance, if $G=0$ only the shapes which surface S is less than or equal to $S^{min}(a_i)$ are allowed. Similarly, if $G=1$, shapes which fulfil $S\leq S^{max}(a_i)$ are allowed, which includes all rectangular/linear shapes. If $0<G<1$, allowed shapes are calculated by Equation 3.56. Specifically, if the mapping of a 4-dispatcher application has to be performed on a 8×8 platform with $G=0.5$, then $S^{min}(a_i)=4$, $S^{max}(a_i)=64$ and $S\leq34$. Consequently, the following shapes would be excluded from the consideration: $\{8\times8, 8\times7, 7\times8, 7\times7, 8\times6, 6\times8, 7\times6, 6\times7, 6\times6\}$.

$$S \leq S^{min}(a_i) + G\cdot(S^{max}(a_i) - S^{min}(a_i)) \tag{3.56}$$

Algorithm 18 illustrates how the $\mathscr{I}\mathscr{P}$ stage is performed. First, the high-priority applications are selected and sorted by their priorities, non-increasingly (lines $1-4$). The applications are treated sequentially; the values of $S^{min}(a_i)$ and $S^{max}(a_i)$ are obtained (line 6), and a shape surface threshold S is calculated (line 7). Then, only the shapes which surface is less than or equal to the calculated threshold S are selected and sorted by the shape surface, non-increasingly (lines $8-12$). The mapping is attempted on the entire grid with the selected application and the biggest allowed shape (lines $15-16$). Note that this process involves (i) the placement of the shape at the particular location on the grid, (ii) the generation of the supermessages, (iii) the assignment of the individual proxy roles for both the application under analysis and the other applications communicating with it, (iv) the generation of the inter-application messages assuming the elected proxies, (v) the schedulability check for the analysed application. Each inter-application message sent to the currently mapping application has a higher priority, hence each such message triggers the schedulability recheck of its application (lines $18-20$). This may also trigger the rechecks of other applications influenced by these updates, therefore a schedulability recheck is performed for each such application (lines $22-25$).

Algorithm 18 $\mathscr{I}\mathscr{P}(\mathscr{A},\Psi,P,G)$

Input: application-set $\mathscr{A}$, platform Ψ, parameter P, parameter G
Output: initial mapping $\mathscr{M}_{ip}$

```
1:  for each (a_i ∈ A | P(a_i) > P^min + P · (P^max − P^min)) do
2:      A_H ← A_H ⋃ {a_i}; // select all high-priority applications
3:  end for
4:  sort(A_H, P ↓); // sort by priority, non-increasingly
5:  for each (a_i ∈ A_H) do
6:      S^min(a_i) ← minArea(a_i);  S^max(a_i) ← maxArea(a_i); // compute S^min(a_i) and S^max(a_i)
7:      S ← S^min(a_i) + G · (S^max(a_i) − S^min(a_i)); // find a shape surface threshold
8:      Allowed ← ∅;
9:      for each (shape | shape.S() ≤ S) do
10:         Allowed ← Allowed ⋃ {shape}; // select allowed shapes
11:     end for
12:     sort(Allowed, S ↓); // sort by shape surface, non-increasingly
13:     schedulable ← false;
14:     while (schedulable ≠ true ∧ Allowed ≠ ∅) do
15:         shape ← remove(Allowed); // get the biggest existing shape
16:         schedulable ← test(a_i, shape);
17:         if (schedulable = true) then
18:             for each (a_j ∈ Senders(a_i)) do
19:                 update(a_j); // update inter-application messages
20:             end for
21:             if (Senders(a_i) ≠ ∅) then
22:                 for each (a_k ∈ A_H | recheck(a_k) = true) do
23:                     schedulable ← schedulable ∧ test(a_k, shape(a_k)); // schedulability recheck
24:                     if (schedulable ≠ true) then break;
                        end if
25:                 end for
26:             end if
27:         end if
28:         if (schedulable = true) then
29:             location ← findBestLocation(a_i, shape); // find the best location
30:             map(a_i, Ψ, shape, location);
31:             distributeDispatchers(a_i); // maximise the mapping quality
32:         end if
33:     end while
34:     if (schedulable ≠ true) then
35:         mappingFailed(a_i); // declare failure of IP
36:     end if
37: end for
38: mappingSuccess(); // declare success of IP
39: return M_ip;
```

If the application can be mapped with the selected shape on multiple places of the grid, the location is found such that the worst-case delay of the application is minimised. In this way the algorithm searches for the position on the grid where the application suffers the least interference from other traffic. Notice that this strategy forces interacting applications to be mapped close to each other. A special case occurs when proxies of two interacting applications share the same core and the inter-proxy message does not exist. Furthermore, if there are several locations on the grid where the application can be mapped and for which the delay is minimised, the approach selects the one for which the sum of the dispatcher distances from the center of the grid is minimised. The intention behind mapping the application as close to the center of the grid as possible is as follows: any lower-priority application which is yet to be mapped, and which performs the communication with the said application, will have the possibility to evaluate more mapping options so as to minimise the communication penalty, while that would not be the case if the said application was mapped on the border of the grid. Note that these two location selection criteria are not explicitly mentioned in Algorithm 18, but it is assumed that this logic is encapsulated within the method in the line 29.

Once the best location is found, the application is considered mapped (line 30). Subsequently, the rest of dispatchers (if any) are positioned on the edges of the shape, such that the mapping quality q_i is maximised (line 31).

If the application cannot be mapped on the grid with the current shape, the mapping is attempted with the next shape from the collection of allowed shapes. The process is repeated until a proper shape and location are found, such that the schedulability constraints are satisfied for the currently mapping and the previously mapped applications. If the schedulability cannot be reached with any of the shapes, the mapping process declares a failure (line 35). Conversely, when all the applications from $\mathscr{A}_H$ are mapped, $\mathscr{I}\mathscr{P}$ declares a success (line 38) and returns the initial mapping $\mathscr{M}_{ip}$ (line 39).

3.6.3.2 Feasibility Phase ($\mathscr{F}\mathscr{P}$)

During this phase the mapping of low-priority applications is performed, with the primary objective to derive a schedulable mapping $\mathscr{M}_{fp}$ where the entire application-set is mapped. Therefore, during $\mathscr{F}\mathscr{P}$, every application is mapped with the narrow mapping. $\mathscr{F}\mathscr{P}$ is described by Algorithm 19. First, all low-priority applications are grouped in $\mathscr{A}_L$ and sorted by priority, non-increasingly (lines $1-4$). The applications are treated sequentially; a surface of a narrow mapping $S^{min}(a_i)$ is found (line 6), and subsequently it is used to find all possible shapes which correspond to the narrow mapping of that application (lines $7-9$). The mapping is attempted with the first shape from the list, without any preference, as all selected shapes represent narrow mappings (lines 12-13). The same logic used in $\mathscr{I}\mathscr{P}$ applies here: if needed, the inter-application messages of higher-priority applications are updated (lines $15-17$) and subsequently the schedulability of affected applications is rechecked (lines $19-22$). Similarly, if multiple grid locations are available for the same shape, the one with the minimum delay is selected, while if more than one location

Algorithm 19 $\mathscr{F}\mathscr{P}(\mathscr{A}, \mathscr{M}_{ip}, \Psi, P)$

Input: application-set $\mathscr{A}$, initial mapping $\mathscr{M}_{ip}$, platform Ψ, parameter P
Output: feasible mapping $\mathscr{M}_{fp}$

1: **for each** $(a_i \in \mathscr{A} \mid P(a_i) \leq P^{min} + P \cdot (P^{max} - P^{min}))$ **do**
2: $\quad \mathscr{A}_L \leftarrow \mathscr{A}_L \bigcup \{a_i\}$; // select all low-priority applications
3: **end for**
4: $sort(\mathscr{A}_L, P\downarrow)$; // sort by priority, non-increasingly
5: **for each** $(a_i \in \mathscr{A}_L)$ **do**
6: $\quad S^{min}(a_i) \leftarrow minArea(a_i)$; // compute $S^{min}(a_i)$
7: $\quad$ **for each** $(shape \mid shape.S() = S^{min}(a_i))$ **do**
8: $\quad\quad Allowed \leftarrow Allowed \bigcup \{shape\}$; // select narrow-mapping shapes
9: $\quad$ **end for**
10: $\quad schedulable \leftarrow false$;
11: $\quad$ **while** $(schedulable \neq true \wedge Allowed \neq \emptyset)$ **do**
12: $\quad\quad shape \leftarrow remove(Allowed)$; // get the first allowed shape
13: $\quad\quad schedulable \leftarrow test(a_i, shape)$;
14: $\quad\quad$ **if** $(schedulable = true)$ **then**
15: $\quad\quad\quad$ **for each** $(a_j \in Senders(a_i))$ **do**
16: $\quad\quad\quad\quad update(a_j)$; // update inter-application messages
17: $\quad\quad\quad$ **end for**
18: $\quad\quad\quad$ **if** $(Senders(a_i) \neq \emptyset)$ **then**
19: $\quad\quad\quad\quad$ **for each** $(a_k \in \mathscr{A} \mid recheck(a_k) = true)$ **do**
20: $\quad\quad\quad\quad\quad schedulable \leftarrow schedulable \wedge test(a_k, shape(a_k))$; // schedulability recheck
21: $\quad\quad\quad\quad\quad$ **if** $(schedulable \neq true)$ **then** break;
$\quad\quad\quad\quad\quad$ **end if**
22: $\quad\quad\quad\quad$ **end for**
23: $\quad\quad\quad$ **end if**
24: $\quad\quad$ **end if**
25: $\quad\quad$ **if** $(schedulable = true)$ **then**
26: $\quad\quad\quad location \leftarrow findBestLocation(a_i, shape)$; // find the best location
27: $\quad\quad\quad map(a_i, \Psi, shape, location)$;
28: $\quad\quad$ **end if**
29: $\quad$ **end while**
30: $\quad$ **if** $(schedulable \neq true)$ **then**
31: $\quad\quad mappingFailed(a_i)$; // declare failure of $\mathscr{F}\mathscr{P}$
32: $\quad$ **end if**
33: **end for**
34: $mappingSuccess()$; // declare success of $\mathscr{F}\mathscr{P}$
35: **return** $\mathscr{M}_{fp}$;

report the same delay, the one closer to the center of the grid has the precedence (line 26). Once the best location is found, the application is mapped (line 27).

If the attempted shape violates the schedulability of the currently mapping application, or any other already mapped application, the mapping is attempted with the next one from the list of allowed shapes. The process is repeated until a shape is found such that all applications are schedulable. If none of the shapes satisfies this requirement, the mapping process declares a failure (line 31). Conversely, once all the applications are mapped, $\mathscr{FP}$ declares a success (line 34) and returns the feasible mapping $\mathscr{M}_{fp}$ (line 35). The computational complexity of $\mathscr{FP}$ also varies. If there are no schedulibility rechecks necessary, the complexity is equal to $O(|\mathscr{A}_L|)$, where $|\mathscr{A}_L|$ denotes the number of low-priority applications. Conversely, if the mapping of every application requires schedulability rechecks, the complexity is $O(|\mathscr{A}_L| \cdot |\mathscr{A}|)$, where $|\mathscr{A}|$ denotes the total number of applications in the application-set, i.e. $|\mathscr{A}| = |\mathscr{A}_H| + |\mathscr{A}_L|$. As will be seen in Section 3.6.4, the actual complexity is closer to the former estimate (linear). Note that the output of $\mathscr{FP}$ is the first schedulable solution of an entire application-set.

3.6.3.3 Optimisation Phase ($\mathscr{OP}$)

The objective of this phase is to improve on the feasible mapping $\mathscr{M}_{fp}$ and produce the final mapping $\mathscr{M}_{op}$. This is performed by attempting to extend the shapes of narrow mappings that low-priority applications claimed during $\mathscr{FP}$. The process ends when no further extensions are possible. That also marks the end of the entire mapping process and the reached mapping $\mathscr{M}_{op}$ presents the final output. $\mathscr{OP}$ is depicted by Algorithm 20.

Similarly to $\mathscr{FP}$, applications from $\mathscr{A}_L$ are selected and sorted non-increasingly (lines $1-4$), but during $\mathscr{OP}$ by the migration coefficient M. As mentioned in Section 3.6.2, it represents the significance of the spatially distributed mapping of an application, i.e. the more the system would benefit from the spatially distributed mapping of an application a_i, the greater its parameter $M(a_i)$ is. The positive side is that applications for which distribution matters more are given the possibility to claim resources before others, thus increasing the chances of the mapping process to derive a good quality solution. The downside is that the number of necessary schedulability rechecks significantly increases; an expanding application can invoke not only schedulability rechecks mentioned in the previous mapping stages, but also of all its directly interfered lower-priority applications, which was not possible in the previous stages as applications were sorted by their priorities, non-increasingly. Thus, unlike the previous mapping stages, where schedulability rechecks were an exception, during $\mathscr{OP}$ the same will be performed regularly. The computational complexity of $\mathscr{OP}$ is identical to that of $\mathscr{FP}$, however, as will be seen in Section 3.6.4, the actual complexity is closer to the higher (sub-quadratic) estimate.

The applications are treated sequentially; the expanded shape is found and the mapping attempted (lines $8-9$). The process consists of an attempt to stretch the application shape: (i) the supermessages are re-generated, (ii) the proxy roles are re-assigned for both the application under analysis and the other applications interacting with it, (iii) the inter-application messages are re-generated assuming the newly elected proxies. As described above, this may require a significant

Algorithm 20 $\mathscr{OP}(\mathscr{A}, \mathscr{M}_{fp}, \Psi, P)$

Input: application-set $\mathscr{A}$, feasible mapping $\mathscr{M}_{fp}$, platform Ψ, parameter P
Output: final mapping $\mathscr{M}_{op}$

```
1:  for each (a_i ∈ 𝒜 | P(a_i) ≤ P^min + P·(P^max − P^min)) do
2:      𝒜_L ← 𝒜_L ⋃ {a_i}; // select all low-priority applications
3:  end for
4:  sort(𝒜_L, M ↓); // sort by migration coefficient, non-increasingly
5:  for each (a_i ∈ 𝒜_L) do
6:      schedulable ← true;
7:      while (schedulable = true) do
8:          newShape ← expand(shape(a_i)); // find expanded shape to attempt
9:          schedulable ← test(a_i, newShape);
10:         if (schedulable = true) then
11:             for each (a_j ∈ 𝒜 | recheck(a_j) = true) do
12:                 schedulable ← schedulable ∧ test(a_j, shape(a_j)); // schedulability recheck
13:                 if (schedulable ≠ true) then break;
                    end if
14:             end for
15:             if (schedulable = true) then
16:                 a.shape ← newShape; // claim expanded shape
17:                 rearrangeDispatchers(a_i); // maximise mapping quality
18:             end if
19:         end if
20:     end while
21: end for
22: mappingSuccess(); // declare success of 𝒪𝒫 and entire mapping process
23: return ℳ_op;
```

amount of schedulability rechecks (lines $11-14$). The application is expanded until any further stretches will cause unschedulability of either itself or any other application. Every expansion is followed by the rearrangement of dispatchers of an expanding application, so as to equalise the inter-dispatcher distances as much as possible and improve the mapping quality (line 17). Once this process is performed for all applications from $\mathscr{A}_L$, the entire mapping process concludes (line 22), and the derived mapping $\mathscr{M}_{op}$ is returned as the final output (line 23).

3.6.4 Experimental Evaluation

In this section, the evaluation of the proposed mapping approach is performed. Specifically, through different case studies, the overall efficiency and applicability of the proposed application mapping method are investigated, as well as how different choices and trade-offs, achievable through parameter manipulations, influence the quality of derived mapping solutions.

3.6.4.1 Case Study 1: Application Shapes and Overheads of Constrained Routes

In many situations, several shapes have similar or identical characteristics, especially during $\mathscr{F}\mathscr{P}$ when only narrow mappings are considered. In order to asses the differences between shape types and reason about their applicability, their relevant characteristics are analysed, namely a traversal distance of the average and the longest intra-application message. For easier calculation, in all cases narrow mappings are assumed, while for rectangular shapes only an even number of dispatchers is considered.

Equation 3.57 calculates the average traversal distance of the intra-application message, assuming a n-dispatcher application with a line-like shape. If the $i-th$ dispatcher of a horizontally stretched application is the master, it communicates with $i-1$ dispatchers on the left and the rest $n-i$ dispatchers on its right (the terms in the brackets of Equation 3.57). The outer summation is performed so as to account for all possible masters. In order to obtain the distance of the average message, the result is divided by the total number of messages (n masters sent $n-1$ messages each). Similarly, the value is calculated for the rectangular shape, where the message routes obey to Definition 4 (Equation 3.58).

$$\text{Line-AVG} = \frac{\sum_{i=1}^{n}\left(\sum_{k=1}^{i-1} k + \sum_{j=1}^{n-i} j\right)}{n(n-1)} = \frac{n+1}{3} \quad (3.57)$$

$$\text{Rect-AVG} = \frac{\sum_{i=1}^{\lceil \frac{n-1}{2} \rceil} i + \sum_{i=1}^{\lfloor \frac{n-1}{2} \rfloor} i}{n-1} = \frac{n^2}{4(n-1)} \quad (3.58)$$

Finding the maximum message distance for both shapes is trivial (Equations 3.59-3.60).

$$\text{Line-MAX} = n-1 \quad (3.59)$$

$$\text{Rect-MAX} = \left\lceil \frac{n-1}{2} \right\rceil \quad (3.60)$$

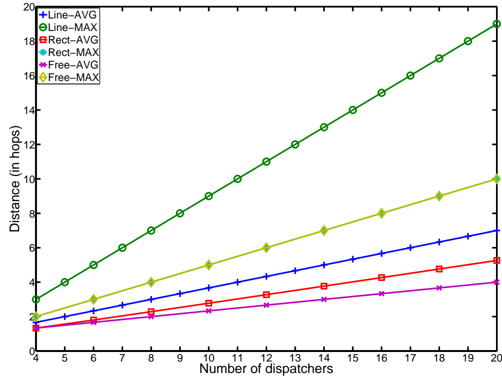


Figure 3.19: Shape comparison

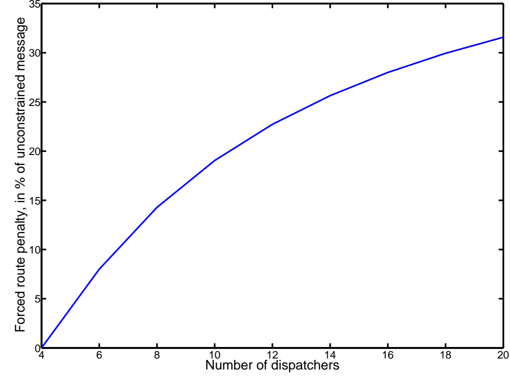


Figure 3.20: Rerouting penalty estimation

Figure 3.19 illustrates how the number of dispatchers influences the distance traversed by the average and the longest intra-application message of these two shape types. The rectangular shape type performs better than the line one, for both the average and the longest message, but at the expense of potential reroutings, which do not occur in the latter case. The decision regarding the shape type precedence can be made by the system designer, or left to the mapping process to decide.

In order to estimate the penalty of constrained intra-application message routes, traversal distances of the average and the longest message are computed for a rectangular shape with the free point-to-point communication between dispatchers (Equations 3.61-3.62), and subsequently plotted in Figure 3.19. Equation 3.61 is derived by using the intermediate results of Equation 3.57.

$$\text{Free-AVG} = \frac{2\sum_{i=1}^{\frac{n}{2}}\left(\sum_{k=1}^{i-1} k + \sum_{j=1}^{\frac{n}{2}-i} j\right) + \frac{n^2}{4}}{\frac{n}{2}(n-1)} = \frac{n^2+3n-4}{6(n-1)} \tag{3.61}$$

$$\text{Free-MAX} = \left\lceil \frac{n-1}{2} \right\rceil \tag{3.62}$$

Figure 3.19 demonstrates that the removal of Definition 4 improves the performance for the average message, but keeps the same longest message and, as already mentioned, makes the analysis pessimistic. Unlike Figure 3.19, which visualises the overhead of forced paths expressed in absolute values, Figure 3.20 depicts the same overhead expressed relatively. In practical terms, Definition 4 causes a 19% longer traversal path for the average intra-application message of a 10-dispatcher application with a rectangular shape.

3.6.4.2 Analysis Parameters

In the subsequent experiments the mapping process is performed, assuming the workload synthetically generated by using the parameters from Table 3.5. An asterisk sign denotes a randomly generated value, assuming a uniform distribution. To assure that all considered application-sets

are indeed schedulable, the individual per-application constraints on the worst-case communication delays were derived as follows. First, each application-set is mapped with the parameters $P = G = 0$. This approach assures that all applications will be mapped during the $\mathscr{IP}$ phase with shapes that correspond to narrow mappings, and will consequently suffer small worst-case communication delays, called *narrow shape delays* (NSDs), hereafter. Thus, by verifying that the generated communication deadline of each application is equal to, or greater than its respective NSD (i.e. $D^{\eta}(a_i) \geq$NSD(a_i)), it can be confirmed that the given application-set is schedulable with at least one selection of the mapping parameters ($P = G = 0$). In the subsequent experiments, the communication deadlines are set to be equal to multiples of NSDs of respective applications, i.e. $D^{\eta}(a_i) = k{\cdot}\text{NSD}(a_i), \forall a_i \in \mathscr{A}$. Notice, that a bigger value of k gives applications more "freedom" to claim wider shapes, but also decreases the computation and memory deadlines of applications, because $D^{\eta}(a_i) = D(a_i) - D^{\tau+\mu}(a_i)$.

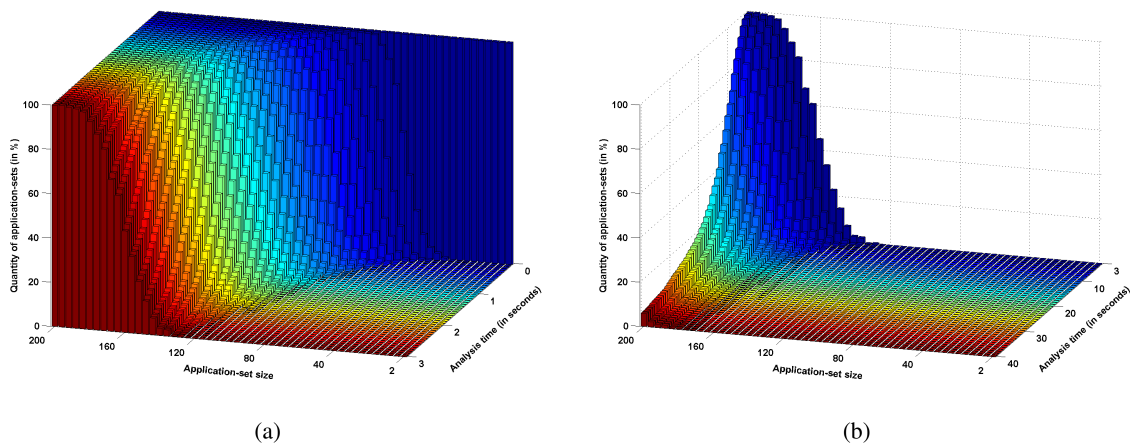
Table 3.5: Analysis and simulation parameters for Section 3.6.4

NoC topology	**2-D mesh with**
Link width = flit size σ_{flit}	**16 bytes**
Router frequency ν_{ρ}	**2 GHz**
Routing delay δ_{ρ}	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Rerouting delay δ_R	**10000 cycles (5 μs)**
OS operations $\delta_P^{\leftarrow}$, $\delta_P^{\rightarrow}$, $\delta_C^{\leftarrow}$, $\delta_C^{\rightarrow}$, $\delta_I^{\leftarrow}$, $\delta_I^{\rightarrow}$, δ_Q, δ_E	**10000 cycles (5 μs)**
Application periods $D(a_i) = T(a_i), \forall a_i \in \mathscr{A}$	$[\mathbf{100} - \mathbf{1000}]^*$ **ms**
Communication deadlines $D^{\eta}(a_i), \forall a_i \in \mathscr{A}$	$\mathbf{0.25{\cdot}D(a_i)}$
Protocol message size σ_{prt}	**16 bytes**
Execution context size σ_{ctx}	$[\mathbf{1} - \mathbf{128}]^*$ **Kbytes**
Inter-application message size σ_{iam}	$[\mathbf{1} - \mathbf{128}]^*$ **Kbytes**
Application-set size $\lvert\mathscr{A}\rvert$	**200 applications**
Application migration coefficient M_i	$[\mathbf{0} - \mathbf{50}]^*$
Number of dispatchers per application $\lvert\mathscr{D}(a_i)\rvert$	$[\mathbf{2} - \mathbf{10}]^*$
Probability of inter-app. comm.	**5%**
Testing platform	**Intel dual-core desktop & Java (Max heap-size: 4 GB)**

3.6.4.3 Case Study 2: Scalability

The objective of this case study is to test the scalability potential of the proposed approach. The number of applications is varied in the range $[2 - 200]$. For each given value of the application-set size, 1000 application-sets are generated and mapped on a 8×8 grid, in accordance with the values from Table 3.5. The other parameters are: $G = 0.3$, $P = 0.5$ and $D^{\eta}(a_i) = 10{\cdot}\text{NSD}(a_i), \forall a_i \in \mathscr{A}$. The timing analysis is performed by capturing the duration of the mapping process of each run.

Figure 3.21 demonstrates the results. The horizontal axis stands for the application-set size. For clarity purposes a reciprocal cumulative graph is used for the presentation, where the vertical axis depicts the quantity of the runs that still did not complete the mapping process within a given



(a) (b)

Figure 3.21: Influence of application-set size on analysis time (1/2)

time interval (depth axis). For small application-sets, almost all the runs complete very fast, which is visible from Figure 3.21(a). As the number of applications increases, the mapping process takes more time, thus the slope of the curve flattens. At the end of the first observed period in Figure 3.21(a) ($0-3$ seconds), almost all the smaller sets finished, while very few of the largest ones report the completion of the mapping process.

The second observed period ($3-40$ seconds) is presented in Figure 3.21(b). Most of the runs for application-sets up to 150 applications already finished, while the runs for the larger sets keep the completion rate steady. Note that the observation period is larger, thus the slope looks steeper. At the end of the observed period, only a small fraction of the largest application-sets did not complete.

The results demonstrate an obvious trend regarding the duration of the mapping process across application-set sizes, however, the mappings of two sets of the same size can report significantly different durations, sometimes by an order of magnitude. To emphasize this fact, a different representation of the same data is given in Figure 3.22. The horizontal axis stands for the application-set size. The left and the right vertical axis represent the duration of the mapping process, in the linear and logarithmic scale, respectively. Even though the whiskers were set to the 25th and 75th percentiles, which corresponds to the 99.3% coverage for the normal distribution, it is visible that a non-negligible amount of runs falls outside the aforementioned area. This infers that the duration of the mapping process may vary hugely, and highly depends on properties of the application-set upon which it is being applied.

Overall, the duration of the mapping process exponentially increases with the number of applications. However, it is averaging at 12 seconds for the sets consisting of 200 applications, which confirms that the proposed approach is applicable to most of realistic scenarios in the real-time embedded domain.

Another scalability test is performed by varying the grid size (from 8×8 to 16×16), while keeping all the other parameters at the same values. The size of the application-set is 150. 1000 sets were generated and subsequently mapped. Figure 3.23(a) shows how the variation of the

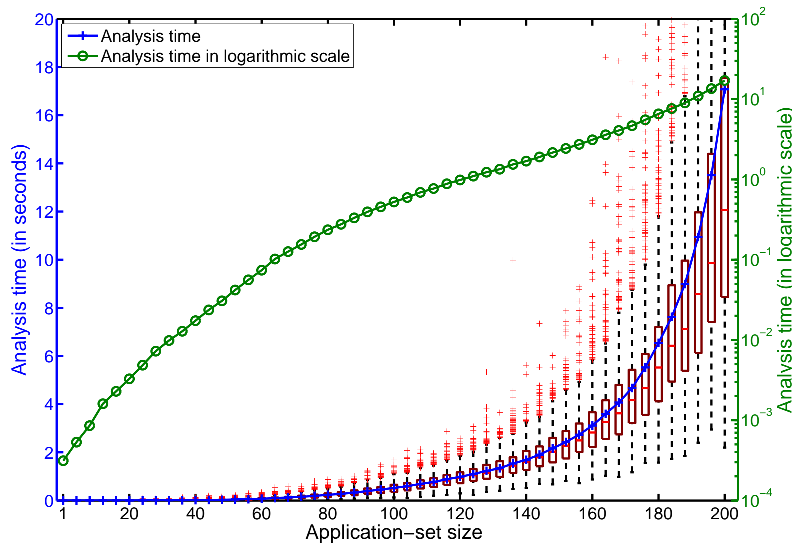


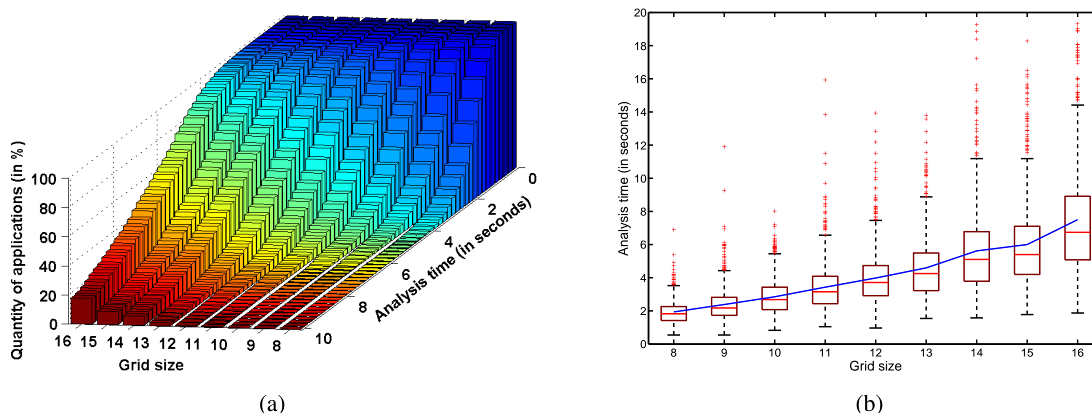
Figure 3.22: Influence of application-set size on analysis time (2/2)

grid size (horizontal axis) influences the analysis time (depth axis). Again a cumulative reciprocal representation is used, where vertical axis stands for the number of application-sets for which the mapping did not finish within a given time.

It is visible that, within 6 seconds, in almost all cases the mapping completed for smaller grids, while larger grids are more time consuming and report fewer completions in the same time interval. Conversely to application-set size variations, the grid size variations cause a linear increase in the duration of the mapping process, visible in Figure 3.23(b). Note that these results infer that on average, the mapping process on a platform with 144 cores takes only 2 times more than on a 64-core one, assuming the same workload is mapped in both cases. The explanation is as follows. Even though larger grids require more locations to be checked while mapping, at the same time this gives the opportunity to map the applications in such a way that complex interference scenarios are avoided, which decreases the duration of schedulability rechecks, since fewer contentions occur. The number of outliers again confirms huge variations in duration times of the mapping process, demonstrating that for some application-sets the duration of the mapping process can significantly exceed the average time needed for that particular workload and platform size.

3.6.4.4 Case Study 3: Parameter P

The parameter P controls the amount of applications which will be grouped in $\mathscr{A}_H$ and hence mapped during $\mathscr{I}\mathscr{P}$, while the rest of the applications will undergo a two-stage mapping process in $\mathscr{F}\mathscr{P}$ and $\mathscr{O}\mathscr{P}$. Note that $\mathscr{F}\mathscr{P}$ and $\mathscr{O}\mathscr{P}$ are computationally more intensive than $\mathscr{I}\mathscr{P}$, so mapping more applications during $\mathscr{F}\mathscr{P}+\mathscr{O}\mathscr{P}$ can cause a significant increase in the computation time. However, this process is more thorough in search and potentially has higher chances of finding better application mappings. Conversely, mapping most of the applications during $\mathscr{I}\mathscr{P}$



(a) (b)

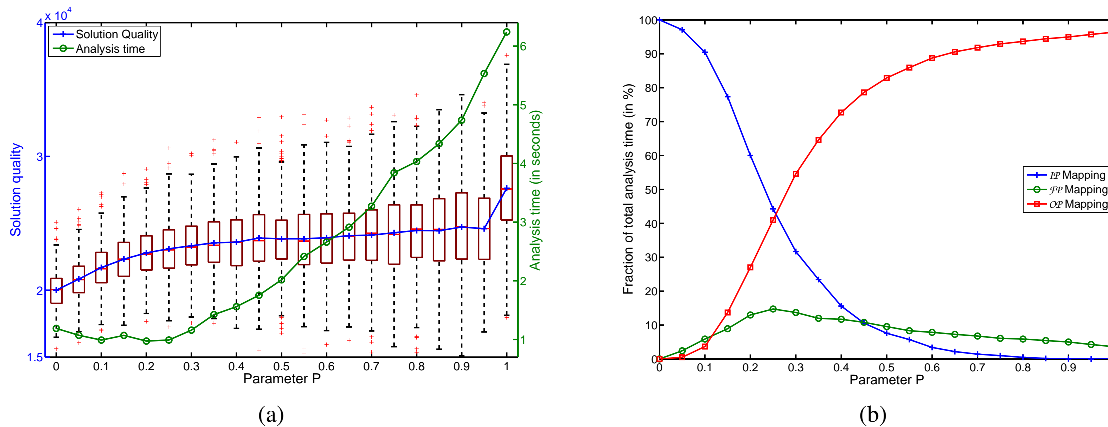
Figure 3.23: Influence of grid size on analysis time

may save the computation time, but makes the efficiency of the mapping process highly dependant on the right selection of the parameter G, as is shown in Case Study 4. Thus, an intuitive assumption is that the selection of the parameter P creates a trade-off between the analysis time and the solution quality.

Assuming $G = 0.3$ and $D^{\eta}(a_i) = 10 \cdot \mathrm{NSD}(a_i), \forall a_i \in \mathscr{A}$, all the application-sets are schedulable across the entire observed domain, where the parameter P is varied in the range $[0-1]$. The application-set size is 150, and 1000 sets are created and mapped on a 8×8 platform, for each incremental step of P, as shown in Figure 3.24(a). The timing analysis is performed (the right vertical axis), but also the qualities of generated solutions are collected (the left vertical axis), since the objective is to observe the effect of P on both the analysis time and the solution quality.

As expected, the increase of the parameter P causes the duration time of the mapping process to grow exponentially. As P increases, more applications undergo a more computationally extensive mapping process, which results in longer analysis times. However, the results report a logarithmic growth of the solution quality, almost on the entire domain. The only exception is the case when $P = 1$, that is, when all applications are mapped during $\mathscr{FP}$ and $\mathscr{OP}$.

The results suggest that putting more applications in $\mathscr{A}_L$ and placing them during $\mathscr{FP}$ and $\mathscr{OP}$ might not produce a solution with a significantly better quality. The explanation is that the final mapping phase - $\mathscr{OP}$, sorts the applications non-increasingly by the migration coefficient and tries to optimise their placements in that order. Since $\mathscr{OP}$ does not have a greediness control mechanism, every application will greedily assume the widest possible mapping such that its schedulability is preserved. Thus, most of the available network resources are consumed by the applications that are considered early during $\mathscr{OP}$, which leaves the ones considered later to be able to barely optimise their placements, if at all. Another interesting conclusion is that mapping some applications during $\mathscr{IP}$ (i.e. $P < 1$) may prevent the $\mathscr{OP}$ phase to optimise the mapping in the most efficient way. This is confirmed with the substantial increase in the solution quality when all applications undergo the optimisation phase ($P = 1$).



(a) (b)

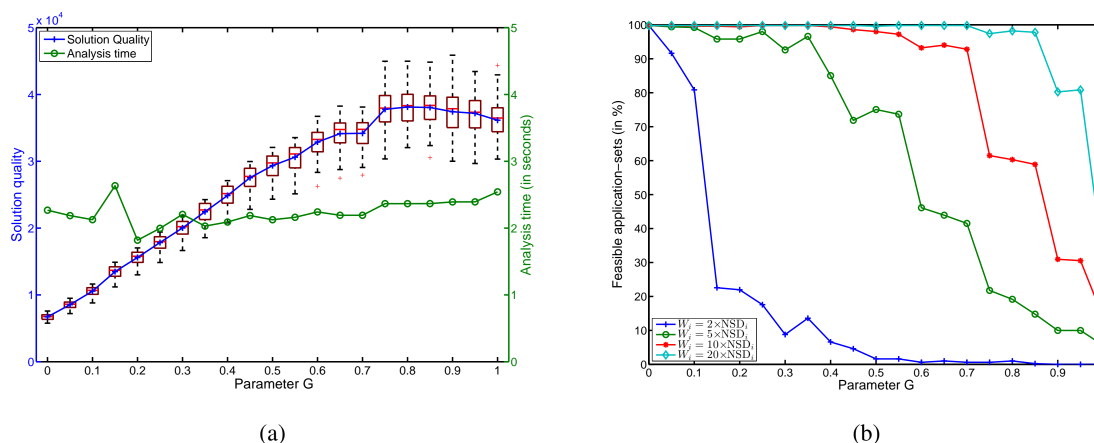
Figure 3.24: Influence of parameter P on mapping process

In order to gain a more detailed insight into the influence of P on the duration of the mapping process, the durations of the individual mapping phases is observed. Figure 3.24(b) shows the fraction of the total analysis time that each phase consumes and compares them in relative terms. For small values of P, almost all applications are mapped during $\mathscr{IP}$, hence leaving less workload for the subsequent phases. Until $P = 0.25$, $\mathscr{IP}$ witnesses an exponential decrease of the analysis time, while the other two phases report a linear increase. Notice, that although all three stages have the sub-quadratic complexity, as P increases the complexity of $\mathscr{OP}$ becomes a dominant factor, while $P > 0.75$ produces scenarios where the analysis time is almost entirely spent in $\mathscr{OP}$. This is explained with the amount of necessary schedulability rechecks, which in the first two stages occur rarely and cause their almost linear complexity, while in the last mapping stage are performed regularly, thus contributing to the true sub-quadratic complexity.

3.6.4.5 Case Study 4: Parameter G

The parameter G controls the amount of greediness allowed to high-priority applications. An intuitive reasoning suggests that the greater value of G causes the greater amount of traffic as a consequence of spatially distributed mappings of applications from $\mathscr{A}_H$. Consequently, low-priority applications, which get placed during $\mathscr{FP}$ and $\mathscr{OP}$, will have less possibilities to assume well distributed placements, since the network resources were greedily consumed by the applications from $\mathscr{A}_H$. In some cases this may cause applications from $\mathscr{A}_L$ to be unable to claim schedulability even with narrow mappings, resulting in a failure of the mapping process. Oppositely, smaller G may limit the distribution of high-priority workload and unnecessary preserve the network for applications from $\mathscr{A}_L$, while there might be only few of them. This can significantly impact the solution quality.

The effects of the parameter G highly depend on the parameter P, since G applies only to high-priority applications, which amount is directly controlled by the parameter P. For small values of P, the influence of G is amplified the most. As P increases, the effects of G mitigate and at some point become negligible. To better observe the impact of G on the mapping process, the



(a) (b)

Figure 3.25: Influence of parameter G on mapping process

following parameters were used: $P = 0$ and initially $D^{\eta}(a_i) = 10{\cdot}\mathrm{NSD}(a_i), \forall a_i \in \mathscr{A}$. Notice that this parameter selection caused some application-sets to be unschedulable on the entire domain, where the parameter G is varied in the range $[0-1]$. Thus, in this part of the experiment only the application-sets which are schedulable on the entire domain are considered. The application-set size is 150, and 1000 sets are created and mapped on a 8×8 platform, for each incremental step of G, as shown in Figure 3.25(a). The timing analysis is performed (the right vertical axis), but also the qualities of derived solutions are observed (the left vertical axis), so as to study the effect of G on both the analysis time and the solution quality.

It is visible that the duration of the mapping process is constant and does not depend on the parameter G. However, G has a significant effect on the solution quality. Specifically, as G increases, high priority applications gain more freedom in assuming mappings wider than narrow ones, resulting in a constant increase of the solution quality. This effect is noticeable until $G = 0.75$. For higher values of G, the solution quality starts to decrease, since high G allows greedy mappings of high-priority applications, and hence preserves less resources for the later mapping stages. In fact, for $G > 0.75$, the network is so greedily consumed by high-priority applications, that low-priority ones barely claim the schedulability with narrow mappings. Note that high values of G not only significantly limit the optimisation of lower priority workload during $\mathscr{OP}$, but also severely affect the schedulability, as explained below.

As already mentioned, Figure 3.25(a) considered only those application-sets which are schedulable across the entire domain of G. Conversely, Figure 3.25(b) illustrates the impact of the parameter G on the schedulability. The application-set size is 150, and 1000 sets are created. For each generated application-set the individual per-application communication deadlines are varied to be the following multiples of the respective NSDs: $2, 5, 10$ and 20. Subsequently, assuming again $P = 0$, the application-sets are mapped on a 8×8 platform and of interest is how the number of schedulable application-sets (the vertical axis) changes with the parameter G (the horizontal axis).

It is visible that the parameter G significantly impacts the schedulability, and as G increases, the number of schedulable application-sets decreases. This is expected, as giving more freedom to

high-priority applications allows them to more greedily consume the available network resources. This inevitably leaves low-priority applications with fewer resources, which, in many cases are not enough to claim the schedulability even with narrow mappings. Notice, that even if the communication deadlines are 20 times greater than the respective NSDs, when $G = 1$ only 30% of the application-sets are schedulable.

3.6.5 Discussion

The efficiency of the mapping process significantly depends on the right selection of the parameters P and G. But more than that, it highly depends on the characteristics of the application-set upon which it is being applied. As observed through the experiments, there are no individual values of P and G which derive the best solution for every application-set. These experiments are useful to recognise and explain the general trends associated to these parameters, but also to show that different mapping strategies can in some cases provide competitive results, and conversely, the same mapping strategy may vary substantially in terms of efficiency when applied to different application-sets.

For instance, for sets with **tight communication deadlines**, setting low values to P and G will very fast produce a schedulable mapping but without a significant quality. Increasing P in such cases may result in a solution with the better quality, but at the expense of the additional time. If the computational complexity is not the problem, in such cases setting $P = 1$ is the preferable option. For sets where applications have **similar communication deadlines** the best strategy is to invoke the balanced network consumption by keeping P low and G low or moderate, depending on the tightness of the deadlines. Conversely, the sets with **significant differences in communication deadlines** will benefit the most from a spatial partialisation approach invoked by moderate or high values of P and high values of G. This parameter selection will allow only a limited number of applications to claim wide mappings and utilise the routes on the boundaries of the grid, and at the same time preserve the central cores free for the applications with tight communication deadlines. Finally, for sets with **relaxed communication deadlines** setting P low or moderate and G high may result in a mapping with the good solution quality, but may also deem the application-set unschedulable with that particular parameter selection. In such cases, decreasing the parameter G until the application-set becomes schedulable is a preferable option.

The advantage of the proposed method is that by setting P and G low, a schedulable solution can be derived very fast. This option may be useful in scenarios where limited computational capacities are available, and the system designer is not very knowledgeable about the nature of the workload. Alternatively, if additional computational capacities are available, the designer might choose to increase P, until reaching a desirable trade-off between the solution quality and the computational complexity. Moreover, if the designer knows workload characteristics (e.g. the number of applications, the number of dispatchers, the tightness of the deadlines), he can take that knowledge into account and make an informed decision regarding the parameter G, which may additionally improve the solution quality.

3.7 Memory Traffic

In this section, two worst-case analysis of the memory traffic are presented. The proposed methods only consider the delays of memory operations within the NoC, while the delays occurring within the memory controllers are out of scope of this dissertation, and have been extensively studied in other works (e.g. [78, 95]).

3.7.1 Workload

Each application a_i performs a set of memory operations which are modelled as the flow-set $\mathscr{F}^{\mu}(a_i)$. Each flow inherits the priority of its application, i.e. $P(f_j) = P(a_i), \forall f_j \in \mathscr{F}^{\mu}(a_i)$, and each flow f_j can be one of the following memory operations: (i) a read request, (ii) a read response, (iii) a write request, and (iv) a write response. Additionally, $\omega(f_j)$ denotes the maximum number of occurrences of a given memory operation within one inter-arrival of its application. An illustrative example of memory operations is given in Figure 3.26, where the flow f_1 represents the read request sent to the memory controller μ_1, while the flow f_2 symbolises the subsequent read response. Similarly, f_3 represents the write request sent to the memory controller μ_2, while f_4 depicts the subsequent write response.

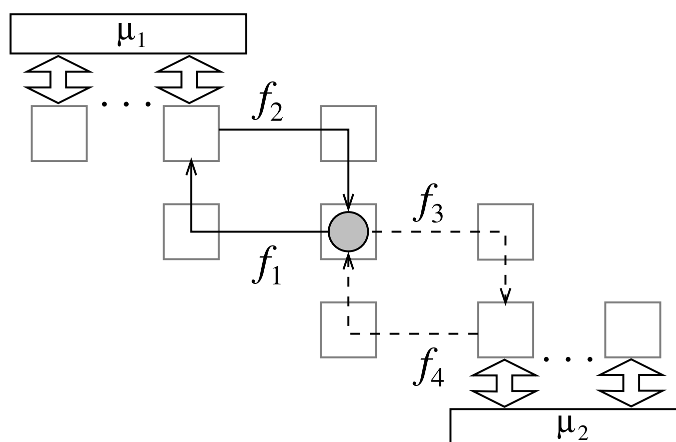


Figure 3.26: Memory operations

3.7.2 Challenges and Inapplicability of Existing Techniques

The problem of non-deterministic flow-paths, due to the master volatility property, is also present in the memory traffic. This is illustrated with Figure 3.27, where a 4-dispatcher application sends a read request to the memory controller μ_1. As is visible, depending on which dispatcher is the current master, the memory operation may involve any of the flows f_1, f_2, f_3 and f_4, which are mutually exclusive.

Given that the accessed memory controller is the same, irrespective of the current master dispatcher, one way to solve the aforementioned problem is to use the proxy dispatcher for memory operations. An example of such an approach is illustrated in Figure 3.28. In this scenario, a

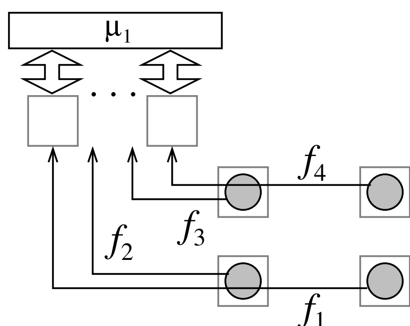


Figure 3.27: Master volatility problem for memory operations

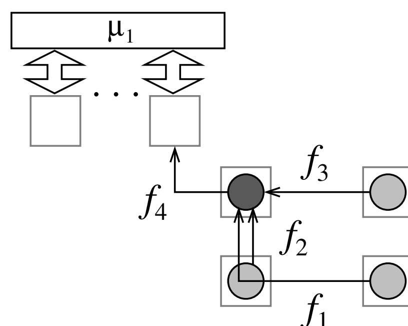


Figure 3.28: Memory operations via proxy dispatchers

master sends the read request via f_1, f_2 or f_3 to its proxy (emphasised with a darker color in Figure 3.28). Then, the proxy forwards the request via f_4 to the memory controller on behalf of its master. Upon receiving the response, the proxy forwards it back to the master. However, the number of memory requests that an application issues within its minimum inter-arrival period can reach thousands, which would create a substantial overhead on proxy cores and lead towards poor performance. Thus, although applicable to inter-application communication, this approach does not have sufficient scalability potential to be applied to memory traffic as well.

3.7.3 Access Constraints and Bounding Messages

A more efficient way to solve the message non-determinism problem is by enforcing access constraints described with Definition 8.

Definition 8 (Memory accesses). *If an application accesses a memory controller, all dispatchers access it from the same tile, which is (i) for the leftmost controllers (i.e. top and bottom) chosen such that it is either in the column of the application's leftmost dispatcher, or left from it, and (ii) for the rightmost controllers, chosen such that it is either in the column of the application's rightmost dispatcher, or right from it.*

In other words, on a platform with $x \times y$ routers, if an application's leftmost and rightmost dispatchers occupy columns i and j respectively, then the leftmost controllers can be accessed via columns $1 : i$ and the rightmost via columns $j : x$. The intention behind this approach is to cause overlapping paths of mutually exclusive messages (notice the visual difference between Figure 3.29(a) and Figure 3.27). Now, a new construct called the *bounding message* is defined. It represents a generalisation of overlapped, mutually exclusive messages.

Definition 9 (Bounding message). *The bounding message $\overline{f}$ is a message that exists for each memory operation that an application performs, and for each row and column of its shape, where it has dispatchers. It connects a memory controller, and a dispatcher from a given row/column, with the furthest distance from it.*

The per-row bounding message is directed from the tile to the memory controller, and represents a generalisation of overlapped, mutually exclusive requests from its row. An application

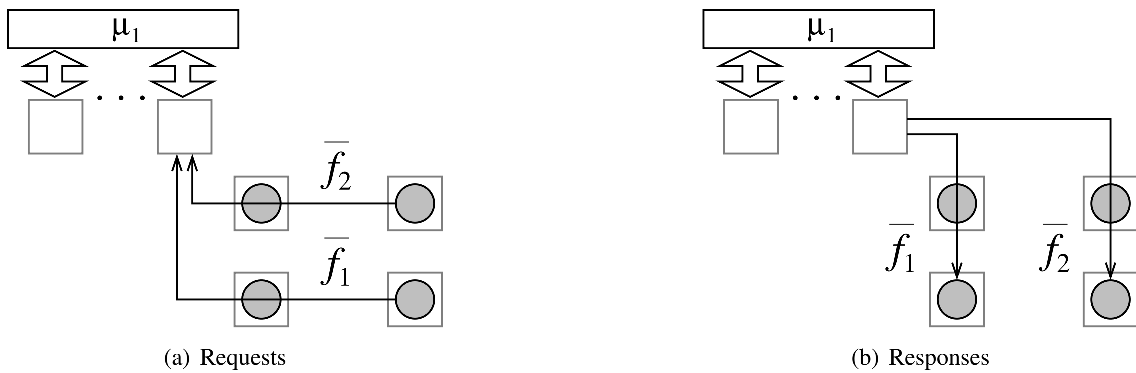


Figure 3.29: Bounding messages

with a rectangular shape and 4 dispatchers has 2 bounding messages for requests towards the memory controller (e.g. $\overline{f_1}$ and $\overline{f_2}$ towards μ_1 in Figure 3.29(a)). Similarly, the per-column bounding message is directed from the memory controller to the router, and depicts a generalisation of overlapped, mutually exclusive responses from its column (see Figure 3.29(b)).

The benefits of bounding messages can be seen by considering all possible read request messages that might be generated from the bottom row in Figure 3.29(a). These read requests are mutually exclusive, as they belong to different dispatchers of the same application. Additionally, both share a part or entirely the path with the bounding message $\overline{f_1}$. Thus, the maximum delay of $\overline{f_1}$ presents an upper-bound on the worst-case delays of respective memory operations originating from the dispatchers of the bottom row. Therefore, a bounding message can substitute all mutually exclusive read request messages from the same row, targeting the same controller. The same applies for write requests. Similarly, a bounding message can substitute all mutually exclusive read and write responses from the same controller, targeting the same column (e.g. in Figure 3.29(b) the bounding message $\overline{f_2}$ can substitute responses towards the dispatchers located in the right column). Notice that bounding messages have similar properties as supermessages, which were used for the intra-application communication traffic. Specifically, both constructs are master-independent and known at design-time. Therefore, the objective is to express all memory traffic with the set of bounding messages, and perform the worst-case analysis on such a model. It is trivial to see that if a bounding message fulfils posed time constraint, all substituted messages also fulfil it.

Note, it may appear that the analysis of bounding messages inherently brings pessimism, since many messages may share only a small fraction of the path with the bounding message, which delay they will assume as the corresponding upper-bound. But it is not so. Timing constraints are posed on groups of mutually exclusive messages and not individual messages. Thus, the only important thing is whether all possible mutually exclusive messages (arising from different dispatcher positions) meet a certain constraint, while the tightness of the estimate of individual messages is irrelevant.

Also note, a bounding message exists for every memory operation. Therefore, if an application

performs both read and write operations with a memory controller, similarly to distinct read and write messages, there will also be distinct bounding messages, for both operations.

3.7.4 Solution to Mutually Exclusive Bounding Messages

Bounding messages substitute mutually exclusive requests from the same row, and mutually exclusive responses from the same column. However, mutually exclusive bounding messages still exist. For example, in Figure 3.29(a), $\overline{f_1}$ and $\overline{f_2}$ are mutually exclusive, the same is true for $\overline{f_1}$ and $\overline{f_2}$ in Figure 3.29(b).

In this section, the method to solve this problem is proposed. Let $\mathscr{F}_E(\overline{f_j})$ be a set of all mutually exclusive bounding messages of $\overline{f_j}$, including $\overline{f_j}$. As during one minimum inter-arrival of a_i only one dispatcher can be a master, consequently only one of mutually exclusive bounding messages may exist[2]. Observe that all of them have the same flow properties (i.e. size, priority, number of occurrences), only differ in the traversal latency, and subsequently in the interference delay they can cause per single occurrence (Equation 2.1). Thus, when a bounding message has multiple mutually exclusive bounding messages in its list of directly interfering messages, it is sufficient to assume one of them, with the maximum traversal latency. The conclusion reached for $\mathscr{F}_E(\overline{f_j})$ is also valid for every set of mutually exclusive bounding messages of every application.

3.7.5 Approach One: Per-Packet Analysis

In this approach, of interest is the maximum delay of a single occurrence (a single packet traversal) of a bounding message $\overline{f_i}$. Let $\mathscr{F}_D(\overline{f_i})$ be a set of directly interfering bounding messages of $\overline{f_i}$. As already discussed, when computing the interference, not all bounding messages from $\mathscr{F}_D(\overline{f_i})$ should be considered, as some of them are mutually exclusive. Thus, a reduced set of directly interfering bounding messages is defined as follows. If two or more bounding messages from $\mathscr{F}_D(\overline{f_i})$ are mutually exclusive, only one with the highest traversal latency is added to the set $\mathscr{F}_R(\overline{f_i})$. Subsequently, only bounding messages from $\mathscr{F}_R(\overline{f_i})$ will be treated as interfering messages.

Formally, $\mathscr{F}_R(\overline{f_i})$ can be defined as follows:

$$\forall \overline{f_j} \in \mathscr{F}_D(\overline{f_i}) \mid \nexists\, \overline{f_k} \in \mathscr{F}_D(\overline{f_i}) \ \wedge\ \overline{f_k} \in \mathscr{F}_E(\overline{f_j}) \ \wedge\ C(\overline{f_k}) > C(\overline{f_j}) \Rightarrow \overline{f_j} \in \mathscr{F}_R(\overline{f_i}) \tag{3.63}$$

[2] It is assumed that memory operations are not needed during agreement protocols, and that the data which is necessary for the protocol execution (if any) is available within local caches.

The worst-case traversal time of a single occurrence (a single packet traversal) of $\overline{f_i}$ can be computed by solving Equation 3.64, where $a_{\overline{f_j}}$ denotes the application to which $\overline{f_j}$ belongs.

$$WCTT(\overline{f_i}) = C(\overline{f_i}) + \sum_{\forall \overline{f_j} \in \mathscr{F}_R(\overline{f_i})} C(\overline{f_j}) \cdot \omega(\overline{f_j}) \cdot \overbrace{\left(1 + \left\lceil \frac{WCTT(\overline{f_i}) - D^{\eta}(a_{\overline{f_j}})}{T(a_{\overline{f_j}})} \right\rceil \right)}^{\text{number of job releases}} \tag{3.64}$$

Note that the term in the brackets presents the maximum number of job releases of the application $a_{\overline{f_j}}$ within the observed time interval, and the proof is omitted because it is very similar to the proof of Theorem 6, which computes the number of protocol executions within the observed time interval.

Upon obtaining the worst-case traversal time of single occurrences of bounding messages, it is possible to derive the worst-case memory traffic delay of an entire application. First, for an application a_i is define the reduced set of bounding messages $\overline{\mathscr{F}}_R(a_i)$ as follows. From each set of mutually exclusive bounding messages of a_i only one with the maximum delay is added to $\overline{\mathscr{F}}_R(a_i)$. Formally, $\overline{\mathscr{F}}_R(a_i)$ is defined as follows:

$$\forall \overline{f_j} \in \overline{\mathscr{F}}(a_i) \mid \nexists \overline{f_k} \in \mathscr{F}_E(\overline{f_j}) \wedge WCTT(\overline{f_k}) > WCTT(\overline{f_j}) \Rightarrow \overline{f_j} \in \overline{\mathscr{F}}_R(a_i) \tag{3.65}$$

In Equation 3.65, the term $\overline{\mathscr{F}}(a_i)$ represents a set of all bounding messages belonging to a_i.

Now the worst-case memory traffic delay of an application a_i, termed $R^{\mu}(a_i)$, can be computed by solving Equation 3.66.

$$R^{\mu}(a_i) = \sum_{\forall \overline{f_j} \in \overline{\mathscr{F}}_R(a_i)} WCTT(\overline{f_j}) \cdot \omega(\overline{f_j}) \tag{3.66}$$

In this, and the subsequently proposed methods, the condition for the schedulability with respect to the memory traffic is $R^{\mu}(a_i) \leq D^{\mu}(a_i)$.

3.7.6 Intermediate Step: Partial Per-Pattern Analysis

Obtaining the worst-case delay of a single occurrence of a bounding message, and consequently multiplying it with the number of occurrences might be a beneficial approach for computation-intensive applications, where few memory requests occur. However, memory-intensive applications have hundreds, if not thousands of memory requests per job execution. Thus, assuming the worst-case scenario for every occurrence of a memory operation might be a very pessimistic approach. In this section, the new method to compute the worst-case delay of bounding messages is presented. The focus of this method is not on the single message occurrence, but rather on the group of occurrences, i.e. on the pattern. Specifically, the goal is to compute the worst-case delay of a bounding message $\overline{f_i}$, but assuming all $\omega(\overline{f_i})$ occurrences that happen within one minimum inter-arrival period of an application.

Similarly to the previous approach, for the bounding message under analysis $\overline{f_i}$, a reduced set of directly interfering messages $\mathscr{F}_R(\overline{f_i})$ is defined, i.e. a set where only one of mutually exclusive bounding messages exists. Consequently, the worst-case delay of $\overline{f_i}$ is computed by Equation 3.67. Note, as occurrences of $\overline{f_i}$ might be spread within the interval corresponding to the joint deadline of the communication and the memory traffic, the interference has to be computed within that period: $D^{\tau+\mu}(a_i)$. Due to that fact, Equation 3.67 does not have a recursive notion.

$$WCTT(\omega(\overline{f_i})) = C(\overline{f_i}) \cdot \omega(\overline{f_i}) + \sum_{\forall \overline{f_j} \in \mathscr{F}_R(\overline{f_i})} C(\overline{f_j}) \cdot \omega(\overline{f_j}) \cdot \left(1 + \left\lceil \frac{D^{\tau+\mu}(a_{\overline{f_i}}) - D^{\eta}(a_{\overline{f_j}})}{T(a_{\overline{f_j}})} \right\rceil \right) \quad (3.67)$$

Now, Equation 3.68 is used to compute the worst-case memory traffic delay of an application.

$$R^{\mu}(a_i) = \sum_{\forall \overline{f_j} \in \overline{\mathscr{F}}_R(a_i)} WCTT(\omega(\overline{f_i})) \quad (3.68)$$

3.7.7 Approach Two: Full per-pattern analysis

The previous method is referred to as the *partial per-pattern analysis*. It is strictly worse than or equal or this approach, so it is considered only to be an intermediate step. In this section, a method called the *full per-pattern analysis* is presented.

As already mentioned, distinct bounding messages exist for each memory operation, irrespective of the fact that some of the may have overlapping paths. Therefore, if an application performs read and write operations with a memory controller, distinct bounding messages will be generated for a read request, a read response, a write request and a write response. Although bounding messages of read and write requests have different sizes and number of occurrences, they have the same priority and some of them share same paths. In such cases they have the same list of directly interfering messages. Therefore, the idea behind this approach is to merge bounding messages forming read and write requests which share the same path, and compute their grouped delay. The same idea is applied to read and write responses.

Consider $\overline{f_i}$ and $\overline{f_j}$, which are bounding messages of a read and a write request of one application, and which share the same path. Thus, their joined restricted list of directly interfering messages $\mathscr{F}_R(\overline{f_{i,j}})$ is equivalent to their individual lists, i.e. $\mathscr{F}_R(\overline{f_{i,j}}) = \mathscr{F}_R(\overline{f_i}) = \mathscr{F}_R(\overline{f_j})$. Their worst-case grouped delay $WCTT(\omega(\overline{f_{i,j}}))$ can be expressed by Equation 3.69. Due to the same reasons as for the partial per-pattern analysis, Equation 3.69 does not have a recursive notion. Note that this approach behaves identically as the partial per-pattern analysis in cases where no bounding messages that can be merged exist, i.e. Equation 3.69 becomes Equation 3.67. On the other hand, intuitively, this approach should provide tighter estimates than the partial per-pattern analysis in cases where bounding messages can be merged, i.e. both read and write operations are performed with the same controller. This is further investigated in Section 3.7.8.

$$WCTT(\omega(\overline{f_{i,j}})) = C(\overline{f_i}) \cdot \omega(\overline{f_i}) + C(\overline{f_j}) \cdot \omega(\overline{f_j}) +$$

$$\sum_{\forall \overline{f_k} \in \mathscr{F}_R(\overline{f_{i,j}})} C(\overline{f_k}) \cdot \omega(\overline{f_k}) \cdot \left(1 + \left\lceil \frac{D^{\tau+\mu}(a_{\overline{f_{i,j}}}) - D^{\eta}(a_{\overline{f_k}})}{T(a_{\overline{f_k}})} \right\rceil \right) \tag{3.69}$$

Now, Equation 3.68 can be used to compute the worst-case memory traffic delay of an application.

3.7.8 Experimental Evaluation

In this section, the efficiency of the proposed approaches is evaluated. Specifically, the tightness of the results derived with all three methods are compared, and it is investigated how these trends change with different application parameters, e.g. the priority, the minimum inter-arrival period, the number of memory operations. Furthermore, some practical conclusions concerning routing policies and a distribution of memory accesses across memory controllers are derived.

3.7.8.1 Analysis parameters

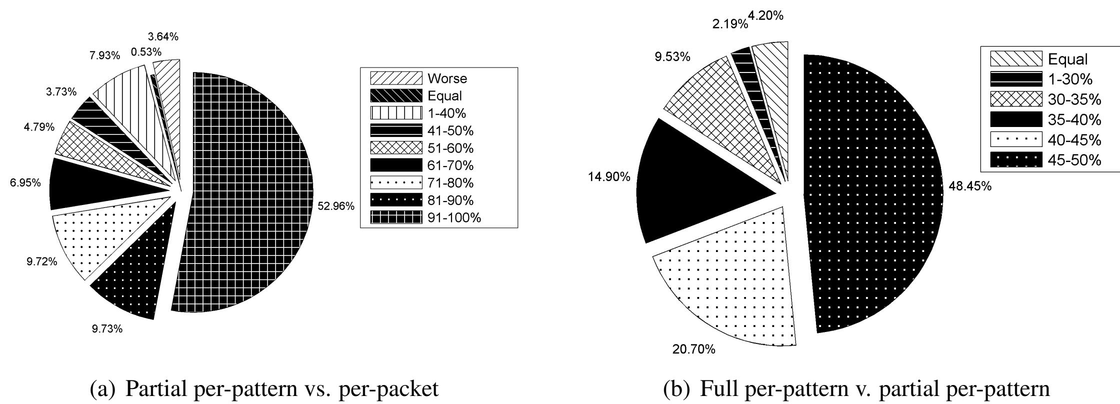
Analysis parameters are given in Table 3.6. An asterisk sign denotes a randomly generated value, assuming a uniform distribution.

Table 3.6: Analysis parameters for Section 3.7.8

Parameter	Value
NoC topology and size	**2-D mesh with 8 × 8 routers**
Link width = flit size σ_{flit}	**16 bytes**
Router frequency ν_ρ	**2 GHz**
Routing delay δ_ρ	**3 cycles (1.5 ns)**
Link traversal delay δ_L	**1 cycle (0.5 ns)**
Application-set size $\lvert\mathscr{A}\rvert$	**200 applications**
Number of dispatchers per application $\lvert\mathscr{D}(a_i)\rvert$	$[\mathbf{2-10}]^*$
Application periods $D(a_i) = T(a_i), \forall a_i \in \mathscr{A}$	$[\mathbf{30-1000}]^*$ **ms**
Joint computation and memory operations deadlines $D^{\tau+\mu}(a_i), \forall a_i \in \mathscr{A}$	$\mathbf{0.75 \cdot D(a_i)}$
Memory operations deadlines $D^{\mu}(a_i), \forall a_i \in \mathscr{A}$	$\mathbf{0.15 \cdot D(a_i)}$
Different memory controllers accessed by one application	$[\mathbf{1-4}]^*$
Memory requests of computation-intensive applications $\omega(f_i)$	$[\mathbf{1-50}]^*$
Memory requests of memory-intensive applications $\omega(f_i)$	$[\mathbf{100-1000}]^*$
Control message size (read request and write response) $\sigma(f_i)$	**32 bytes**
Content message size (read response and write request) $\sigma(f_i)$	**1 Kbyte**

3.7.8.2 Experiment 1: Overall Comparison

In this experiment, the overall comparison of the proposed approaches is performed. Each application of an application-set is randomly mapped on the platform, assuming dispatcher placement constraints (Definition 3). Consequently, bounding messages of memory operations are generated. Then, the worst-case memory traffic delay is computed for each application, with all three methods. Finally, the obtained values are compared. The process is repeated for 1000 application-sets.



(a) Partial per-pattern vs. per-packet

(b) Full per-pattern v. partial per-pattern

Figure 3.30: Analyses comparison

Figure 3.30(a) shows the improvements of the partial per-pattern analysis over the per-packet analysis. It is visible that the per-packet analysis renders tighter estimates only for 3.64% of applications. This fact will receive additional attention in Experiment 4. In 0.53% of the cases, both methods derive the same results, while in all other cases the partial per-pattern analysis performs better. Specifically, for more than half of applications, the improvements are greater than 90%, which means that the estimates are tighter by a factor of 10 or more.

Figure 3.30(b) compares the full per-pattern analysis and the partial per-pattern analysis. For 4.20% of applications both methods derive the same values. As discussed when the analyses were presented, these are the cases when an application performs only one (read or write) operation with every memory controller that it communicates with. In the rest of scenarios, the full per-pattern analysis provides improvements, which in this case grow up to 50%. Additional conclusion is that the full per-pattern analysis dominates the partial per-pattern analysis (strictly better or equal). This coincides with the intuitive assumption from the previous section.

3.7.8.3 Experiment 2: Distribution of memory accesses across controllers

In this experiment, it is investigated how the distribution of memory accesses across memory controllers impacts the analysis. Thus, it is assumed that all data that an application needs is fetched from only one memory controller. Similarly, the worst-case delays of each application of the application-set are derived, and the process is repeated for 1000 application-sets, assuming the full per-pattern analysis. The obtained values are compared with the results from the previous experiment for the full per-pattern analysis, where every application may access multiple memory controllers.

The improvements achieved by a scheme where each application accesses only a single controller are demonstrated in Figure 3.31. Surprisingly, this approach does not always yield better results. In fact, for 9.27% of applications this scheme renders worse estimates. The explanation is that, when an application accesses only a single controller, the path is "greedily" consumed by its entire traffic. Consequently, some links of the NoC infrastructure are extensively used, causing a

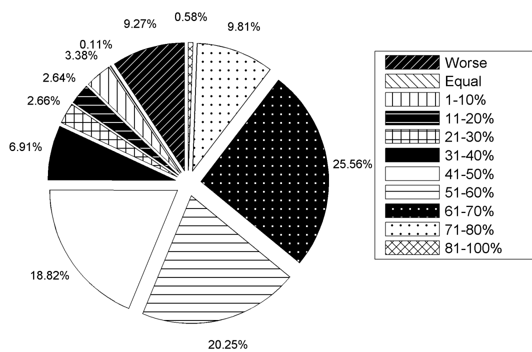


Figure 3.31: Improvement with single controller

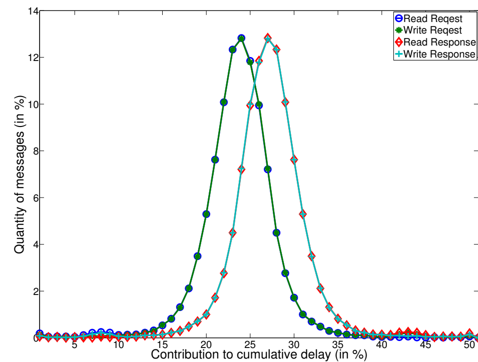


Figure 3.32: Different memory operations

substantial interference to any lower priority application which utilises that path. Oppositely, when each application accesses multiple memory controllers, the traffic is more equally distributed, thus making it possible for lower priority applications to suffer less interference. However, the rest of applications benefit from the new scheme and have tighter estimates. Thus, it can be concluded that the distribution of memory accesses plays an important role in the worst-case analysis of memory traffic and it is a potential topic for the future research.

3.7.8.4 Experiment 3: Different memory operations and messages

In this experiment the objective is to quantify the fraction of the total delay spent in each individual component, namely read requests, read responses, write requests and write responses. In order to investigate that, the per-packet analysis is performed. Of interest are only applications that contain all 4 components, i.e. perform both read and write operations with the memory controller. The analysis is performed for each such application in the application-set, and repeated for 1000 application-sets. Upon obtaining the delays of single packet occurrences, the contributions of individual delays in the cumulative delay of all 4 packets are estimated.

The results are presented in Figure 3.32. It is visible that read and write request messages are almost overlapping. Furthermore, both messages share the same path and only differ in the message size. The same is also true for both response messages. Thus, the first conclusion is that the message size has almost no influence on the delay. Additionally, responses have significantly higher delays than the requests, each averaging at 27% of the total cumulative delay. Requests consume less, around 23% each. The explanation for this surprising finding is that the X-Y routing mechanism is not very efficient for memory responses [2]. Specifically, each response message is injected into the NoC either in the topmost or the bottommost row. In either case, a message first traverses on the x-axis. However, all other responses also do the same. This can cause large amount of contention within the topmost and the bottommost row. The effects in Figure 3.32 are substantially mitigated, due to the existence of per-priority virtual channels which reduce the effects of indirect interferences. However, in scenarios with a single virtual channel, this can cause more significant impacts on delays of response messages, thus further motivating the research in the area of routing mechanisms. This topic is also a potential future work.

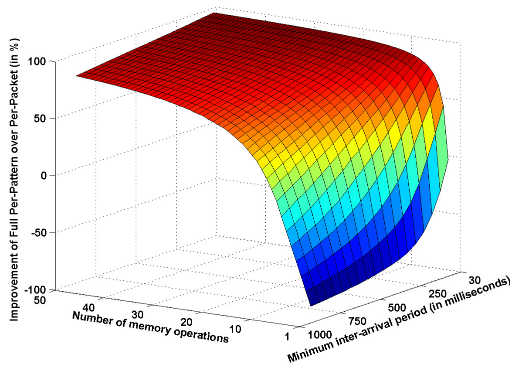


Figure 3.33: Full per-pattern vs. per-packet

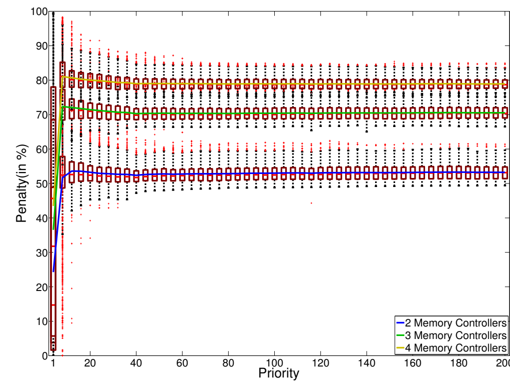


Figure 3.34: Penalty of multiple controllers

3.7.8.5 Experiment 4: Computation-intensive applications

In this experiment the focus is on the applicability of the proposed methods to computation-intensive applications. For each such application the number of memory operations and the minimum inter-arrival periods are varied. Consequently, the per-packet and the full per-pattern analysis are performed, and the obtained values are compared. The computation is repeated for 1000 application-sets.

The results are depicted in Figure 3.33. It is visible that the per-packet analysis renders tighter estimates for applications with few memory operations and long periods. As the number of operations increases, the improvements over the full per-pattern analysis decline. Additional increase favours the full per-pattern analysis, which outperforms the per-packet analysis on the rest of the domain. Similar trends apply to minimum inter-arrival periods of applications, where any decrease also decreases the benefits of the per-packet analysis.

3.7.8.6 Experiment 5: Memory-intensive applications

It is obvious that the full per-pattern analysis is the most suitable method for memory-intensive applications, thus that aspect is not investigated. In this experiment again the focus is on the distribution of memory accesses across memory controllers. Experiment 2 demonstrated the positive and negative sides of having the data of each application fetched from a single memory controller. However, in some cases an application has to be mapped on a platform which already has an existing application-set. In such cases, dedicating only one memory controller to it might be a good decision from the perspective of that application, but might have a negative impact on the already existing workload (as shown in Experiment 2). Thus, in order to minimise the effects of the new application on the existing system it might be more beneficial to distribute its memory accesses across multiple memory controllers. Motivated by this reasoning, the objective of this experiment is to quantify the individual, per-application overheads of having its memory content fetched from multiple memory controllers. For each memory-intensive application the priority is varied and the full per-pattern analysis is performed, assuming its memory operations are equally spread across:

(i) only one controller, (ii) two controllers, (iii) three controllers and (iv) four controllers. Consequently, the penalty of accessing multiple controllers is estimated, relative to the schemes where only one controller is accessed. The experiment is repeated for all 1000 application-sets.

Figure 3.34 shows the penalty of having the data fetched from multiple memory controllers. Surprisingly, the priority has a very small impact on the results. For higher priorities (smaller numbers), applications suffer fewer contentions, thus the penalty of accessing multiple controllers is predominantly composed of increased traversal distances. As priority decreases, schemes with multiple controllers suffer the interference, due to the spread traffic across the grid. Oppositely, scenarios with single controller accesses consume less NoC infrastructure, and in many cases still manage to avoid the interference. This causes the increase in the penalty. Around the priority level 10, the interference becomes predominant, resulting in significant delays even for single controller accessing schemes, thus a slight drop in the penalty is visible. Finally, the penalty stays constant on the rest of the domain. As is visible, the same trends apply for all scenarios involving accesses to multiple controllers. Moreover, the results suggest that the distribution of application's data accesses among multiple controllers is very "expensive". Thus, one of the strategies when adding new application might be to distribute its accesses among as much different memory controllers as possible, such that its temporal constraints are still fulfilled. In this way, the impact of that application on the existing system is minimised.

3.8 Computation

The focus of this section is on the computation requirements. As already defined, each application a_i releases a job with the execution time $C^{\tau}(a_i)$ and the deadline $D^{\tau}(a_i)$. Moreover, due to the fact that the computation process and the memory requests are mutually interleaved, the observed interval is $D^{\tau+\mu}(a_i)$.

In this section, only an initial step towards the worst-case computation delay analysis is proposed. In particular, this approach is based on the simplifying assumption that the delays of communication-related OS operations on each core are incomparably smaller than the computation delays and hence can be neglected, which can be expressed with Equation 3.70.

$$max\{\delta_P^{\rightarrow}, \delta_P^{\leftarrow}, \delta_C^{\rightarrow}, \delta_C^{\leftarrow}, \delta_Q, \delta_E, \delta_I^{\rightarrow}, \delta_I^{\leftarrow}\} \ll C^{\tau}(a_i), \forall a_i \in \mathscr{A} \tag{3.70}$$

The extended approach, which would render the aforementioned assumption obsolete, is a potential future work. Moreover, for the presented coarse-grained analysis a mapping process is proposed, which takes into account computation requirements.

3.8.1 Core Shutdowns

Conversely to the state of the art methods, which consider that all cores are always operational, in this approach occasional core shutdowns are allowed for various beneficial reasons, e.g. power and thermal management. Note that the previously presented communication- and memory-related

worst-case analyses of *LMM* are not violated with this assumption, and the explanation is twofold: (i) shutting down cores has no effect on the message transfer and rerouting operations, because the same are performed by routers, and (ii) dispatchers located on cores which are shut down can be treated as non-existent.

Once a core is selected for shutting down, it will continue to execute the already assigned workload, however, it will reject new job releases. When the last previously assigned job completes, the core will become temporarily unavailable (e.g. sleep interval, cooling period). Depending on the purpose, a system designer might choose to apply more/less aggressive load balancing strategies, involving more/less frequent core shutdowns. As a means to control that, a parameter K is introduced, which symbolises the maximum number of concurrent core shutdowns. The value of K is assumed to be already specified and is not elaborated upon in this dissertation.

3.8.2 Workload

Each dispatcher d_i^j of an application a_i has its own priority which can be less than or equal to the application's priority, i.e. $P(d_i^j) \leq P(a_i), \forall d_i^j \in \mathscr{D}(a_i) \wedge \forall a_i \in \mathscr{A}$. The priority assignment upon dispatchers is also one of the contributions of this section. A job released by the dispatcher d_i^j inherits its priority $P(d_i^j)$. Note that allowing dispatchers of the same application to have different priorities implies that job priorities of one application are not necessarily constant, but depend on master dispatchers which are elected to release them. This is also a novel concept in the real-time domain.

Based on their computation requirements, applications are classified into three categories: *Safety-Critical Applications* (SCA), *Real-Time Applications* (RTA) and *Best-Effort Applications* (BEA).

SCA are considered to be the highest-priority workload in the whole system. Therefore, strong guarantees regarding their computation requirements should be provided, ensuring that no missed deadlines of SCA will occur, even under circumstances that involve core shutdowns.

RTA represent the medium-priority workload, and also require schedulability guarantees. However, the guarantees only need to hold when the system is working with the full capacity (no core shutdowns).

Finally, BEA present the lowest-priority workload in the system. BEA can tolerate missed deadlines, however, that has an impact on the quality of service. Hence, the focus is not on deriving schedulability guarantees for BEA, but instead on establishing a notion of fairness, e.g. by spreading missed deadlines as evenly as possible across all BEA. This decision is motivated by the fact that, in many practical scenarios, maintaining all non-critical functionalities with reduced quality is more desirable than cancelling some of them (e.g. multimedia applications).

The assumed workload classification and respectively posed requirements are inspired by the existing workload classification which is well established in the real-time embedded area [17].

3.8.3 Problem Statement

The objective can be summarised with the following statement. Given the application-set $\mathscr{A}$, the platform Ψ and the maximum allowed number of concurrent core shutdowns K, assign priorities to dispatchers and map them onto the platform ($\mathscr{M} = \mathscr{A} \rightarrow \Psi$), such that:

- No missed deadline of SCA occurs, assuming that the system exhibits at most K concurrent core shutdowns.
- No missed deadline or RTA occurs when the system is working with the full capacity (no core shutdowns).
- The ratio of missed BEA deadlines is spread across all BEA as evenly as possible.

As already known, the application mapping for many-cores is an NP-Hard problem [40], hence searching for the optimal solution is, in most cases, prohibitively expensive. Nonetheless, even with infinite computational capacities, in this particular case the optimal solution could not be found at design time, since the "optimality" directly depends on core shutdown decisions, which are made at runtime. Therefore, in this section, an alternative heuristic-based approach is explored, with the objective of finding a sub-optimal solution of acceptable quality.

3.8.4 Offline Schedulability Guarantees

If a dispatcher of an application passes an *offline schedulability test*, performed at design time, it means that the application can perform the computation on its core as long as it is operational and will never miss a deadline due to the other workload residing on the same core. The test is performed by computing the worst-case response time for a fully preemptive fixed-priority system [58], treating that core as an independent single-core device and assuming the workload that can be generated by all dispatchers existing on that core.

$$R^{\tau}(a_i) = C^{\tau}(a_i) + \sum_{\forall d_k^m \in \mathscr{D}_{\pi(d_i^j)} \mid P(d_k^m) > P(d_i^j)} \left\lceil \frac{R^{\tau}(a_i) + D^{\mu}(a_i)}{T(a_k)} \right\rceil \cdot C^{\tau}(a_k) \tag{3.71}$$

The worst-case response time of a job released by a dispatcher d_i^j on its core $\pi(d_i^j)$ consists of two terms (Equation 3.71). The first is the computation time of that job – $C^{\tau}(a_i)$. Additionally, any higher priority dispatcher d_k^m residing on the same core may release a job which can preempt the execution of the job under analysis. Therefore, from every such dispatcher the maximum possible interference has to be computed (the second term in Equation 3.71). Note that Equation 3.71 has a recursive notion, thus is solved iteratively. If the computed value is less than or equal to the deadline, i.e. $R^{\tau}(a_i) \leq D^{\tau}(a_i)$, an application is considered *offline schedulable* with respect to its computation requirements, with its dispatcher d_i^j on the core $\pi(d_i^j)$.

3.8.5 Online Schedulability

If a dispatcher is not offline schedulable, it does not mean that a job of its application can never perform the computation on that core without missing a deadline. It only implies that the ability to do so depends on the higher priority workload residing on that core at the moment of observation, i.e. which of the higher priority dispatchers are elected by their respective applications to release the jobs on their cores. Thus, in order to determine whether it can release a job and provide the guarantees that the deadline will not be missed, a dispatcher has to perform an online response time test (Equation 3.72).

$$R^{\tau}(a_i) = C^{\tau}(a_i) + \sum_{\forall J_k^m \in RQ(\pi(d_i^j)) \mid P(J_k^m) > P(d_i^j)} \widetilde{C^{\tau}}(a_k) + \sum_{\forall d_n^p \in \mathscr{D}_{\pi(d_i^j)} \mid P(d_n^p) > P(d_i^j)} \left\lceil \frac{t + R^{\tau}(a_i) + D^{\mu}(a_i) - r_n^{p+1}}{T(a_n)} \right\rceil \cdot C^{\tau}(a_n) \tag{3.72}$$

The response time consists of the computation time $C^{\tau}(a_i)$, augmented by the higher priority content of the ready-queue $RQ(\pi(d_i^j))$, where $\widetilde{C^{\tau}}(a_k) \leq C^{\tau}(a_k)$ stands for the remaining computation time of the higher-priority ready-queue job J_k^m belonging to the application a_k, observed at the time instant t. Moreover, a conservative assumption is that all higher priority dispatchers will be elected for their future releases, thus, the potential interference has to be considered from every such dispatcher d_n^p. It is calculated as the maximum number of occurrences in the interval between the next job release of d_n^p occurring at $r_n^{p+1} \geq t$, and the absolute response time $t + R^{\tau}(a_i) + D^{\mu}(a_i)$. Notice, that r_n^{p+1} and t are absolute values, thus are represented with lower-case characters. Equation 3.72 is also recursive, and if the computed value is less than or equal to the deadline (i.e. $R^{\tau}(a_i) \leq D^{\tau}(a_i)$), the application is *online schedulable* with respect to its computation requirements at the instant t on the core of its dispatcher d_i^j.

By passing the online schedulability test, the application gets guarantees from the system that its next job, released at t, can be executed on the core of its dispatcher d_i^j without missing a deadline. The guarantees are valid only for the next job, once its execution is completed the test should be performed again. At this stage, a reader may raise two very valid concerns:

- Is it practical to perform a recursive computation online?
- Is it practical to manage remaining execution times?

These concerns are addressed in the following way. For cases where solving Equation 3.72 might be prohibitively expensive, a modification of the online schedulability test is proposed. A test is agnostic with respect to remaining execution times and completes within a single iteration, i.e. it is not recursive (Equation 3.73).

$$R^{\tau}(a_i) = C^{\tau}(a_i) + \overbrace{\sum_{\forall J_s^q \in RQ(\pi(d_i^j)) \mid P(J_s^q) > P(d_i^j)} min\{C^{\tau}(a_s), r_s^q + R^{\tau}(a_s) - t\}}^{\text{interference from jobs with guarantees}} +$$

$$\overbrace{\sum_{\forall J_u^v \in RQ(\pi(d_i^j)) \mid P(J_u^v) > P(d_i^j)} C^{\tau}(a_u)}^{\text{interference from jobs without guarantees}} + \overbrace{\sum_{\forall d_k^m \in \mathscr{D}_{\pi(d_i^j)} \mid P(d_k^m) > P(d_i^j)} \left\lceil \frac{t + D^{\tau+\mu}(a_i) - r_k^{m+1}}{T_k} \right\rceil \cdot C^{\tau}(a_k)}^{\text{interference from future releases}} \quad (3.73)$$

When compared to Equation 3.72, the first term is the same, while the computation of the interference due to future releases (the last term) is not recursive any more and is computed within a single iteration for the interval $D^{\tau+\mu}(a_i)$. Conversely, the ready-queue content (the second and the third term) is computed differently. First, for each higher-priority job J_s^q, released at r_s^q, which has either offline or online guarantees that its computation will complete until $r_s^q + R^{\tau}(a_s)$, the remaining execution time is computed by finding the smaller between (i) its total execution time and (ii) the difference between its worst-case response time, expressed in absolute values, and the current time instant t. For each higher-priority job J_u^v, which has been released without guarantees, it has to be assumed that its remaining computation time is equal to its worst-case computation time, because it *may* miss a deadline.

Note, intermediate approaches are also possible, e.g. the remaining computation times are available, but the computation should be performed within a single iteration, and vice versa. For such approaches online schedulability tests can be derived by modifying Equations 3.72-3.73, which is not covered in this dissertation. In Section 3.8.11 the focus is on how the knowledge about the remaining computation times and the allowed number of iterations influence the efficiency of online schedulability tests.

3.8.6 Semi-Schedulability Guarantees

Even though multiple dispatchers of an application might be offline schedulable, each job will be executed by only one of them (an elected master dispatcher), thus leaving the remaining dispatchers idle. This fact can be exploited, so as to derive another form of guarantees, termed the *semi-schedulability*. An application is semi-schedulable if none of its dispatchers is offline schedulable, however, at any time instant at least one is online-schedulable.

Consider two applications a_p and a_c, where the first one has a higher priority $P(a_p) > P(a_c)$, and each dispatcher inherits a priority of a respective application, i.e. $P(d_p^i) = P(a_i), \forall d_p^i \in \mathscr{D}(a_p) \wedge P(d_c^j) = P(a_c), \forall d_c^j \in \mathscr{D}(a_c)$. Let a_p and a_c share two common cores - π_x and π_y, and let a_p be offline schedulable on both of them. Additionally, assume that a_c is not offline schedulable on either, but the absence of a_p would make a_c offline schedulable on both π_x and π_y. Since a_p can execute only on one core at any time instant, say π_x, the core π_y might be able to accommodate a_c. And vice versa. Figure 3.35 illustrates one such example. The fractions represent

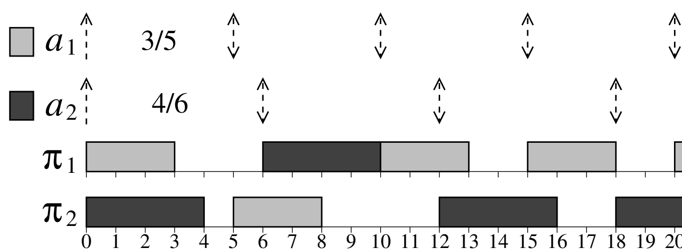


Figure 3.35: Semi-schedulability

computation times and minimum inter-arrival periods $C^{\tau}(a_i)/T(a_i)$, while for clarity purposes the communication and memory operations have been omitted from this example, i.e.:

$$C^{\eta}(a_1) = C^{\eta}(a_2) = D^{\eta}(a_1) = D^{\eta}(a_2) = C^{\mu}(a_1) = C^{\mu}(a_2) = D^{\mu}(a_1) = D^{\mu}(a_2) = 0$$

$$D^{\tau}(a_1) = T(a_1), D^{\tau}(a_2) = T(a_2)$$

Formally, this relation can be described as follows:

Definition 10 (Semi-schedulability)**.** *An application a_c is considered semi-schedulable with respect to a higher priority application a_p, if a_c and a_p share at least two common cores, on which a_p is offline schedulable and a_c is not, but the absence of a_p would make a_c offline schedulable on both of them. Then, a_c is called a semi-schedulable child, while a_p is called a semi-schedulable parent.*

The semi-schedulability provides guarantees that, even though an application does not have offline schedulable dispatchers, at any time instant there will be at least one which can claim online schedulability, thus will be able to accommodate the next job without missing a deadline. The semi-schedulability creates a co-scheduling parent-child relationship, which implies that, at certain points during runtime, some dispatchers may be prevented from being elected. Hence, semi-schedulable applications will have less flexibility when electing dispatchers for their next job releases. However, this does not violate the statement that the scheduling decision of each application is made by the application itself. This is possible because all the information that is necessary for deriving a release/migration decision (e.g. the workload state on candidate cores) is available to the dispatchers from their local kernels and is communicated during the agreement protocol. Semi-schedulability guarantees hold as long as both common cores remain operational (neither one is selected for shutting down).

The relationship between a parent and a child application is not trivial to analyse, and if the executions are not synchronised well, it can cause the child to miss deadlines. One such example is given in Figure 3.36. An application a_3 is semi-schedulable with respect to a_1. Additionally, an application a_2 exists in the system, with the priority lower than that of a_1, but higher than that of a_3, i.e. $P(a_1) > P(a_2) > P(a_3)$. On both the cores, π_1 and π_2, the applications a_1 and a_2 are offline schedulable. a_3 is not, but the absence of a_1 would also make it offline schedulable. These conclusions can be reached by solving Equation 3.71 for this example, with the job parameters

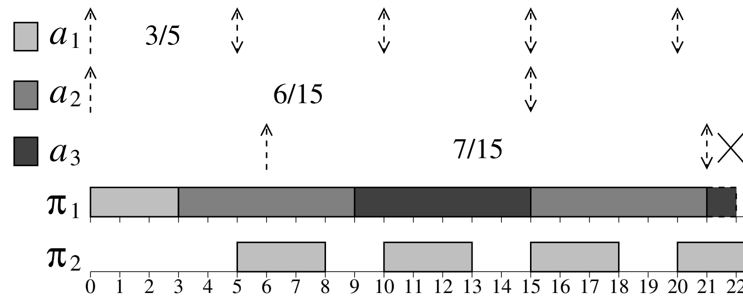


Figure 3.36: Non-synchronised semi-schedulability

given as fractions $C^{\tau}(a_i)/T(a_i)$ in Figure 3.36. Like in the previous example, the communication and memory operations have been omitted for better clarity.

a_1 completed its first job on π_1, but for the next release at $t = 5$ decided to migrate to π_2 and stays there until the end of the example. In order to release its job, a_3 runs its agreement protocol at $t = 6$ and realises that the semi-schedulable parent is on π_2, hence migrates to π_1. However, until its deadline, a job of a_3 can not complete the execution, due to the higher priority workload of a_2 and thus misses its deadline.

As seen, migrations of a parent application can make a child application unschedulable, even when organising their executions on different cores. In this example the first execution of a_1 on π_1 delayed an execution of a_2 and created a workload backlog which consequently delayed the execution of a_3. Due to this delay, a_2 requested more computation time than what is exhibited in the offline schedulability test performed for a_3, thus making it only a necessary but not a sufficient condition for semi-schedulability (i.e. the Equation 3.71 shows that a_2 induces 6 time units of interference to a_3 in the interval between its release and completion, while in the given example it sums up to 9 time units). Note that the way in which a_1 affects a_3 is very similar to the effect of indirect interferences in the domain of traffic flows (see Chapter 2).

Therefore, before performing the migration, a parent application should check whether its actions cause the unschedulability of its child. In the aforementioned example, through its agreement protocol at $t = 5$, a_1 should have checked whether its migration to π_2 would cause a_3 to be unschedulable on π_1, and if so, act accordingly (i.e. either continue executing on π_1, or go to some other core).

Thus, during its agreement protocol at time instant t, a parent application should perform an online schedulability test for the next job release of a child application a_i occurring at r_i. It is equivalent to performing an online schedulability test for an artificial application a_i^*, with all the properties of a_i, except the release is at $r_{i^*} = t$ and the period is extended to $T(a_i^*) = T(a_i) + r_i - t$. An illustrative example is given in Figure 3.37, and Theorem 15 provides the proof.

Theorem 15. *Consider an application a_c, which is on the core π_x a semi-schedulable child with respect to a parent application a_p. Also consider that after a time instant t, any new release of a_p will occur on cores other than π_x. Observed at t, the next future release of a_c, occurring at r_c, will be schedulable if and only if an artificial application a_c^* with the same priority and the computation*

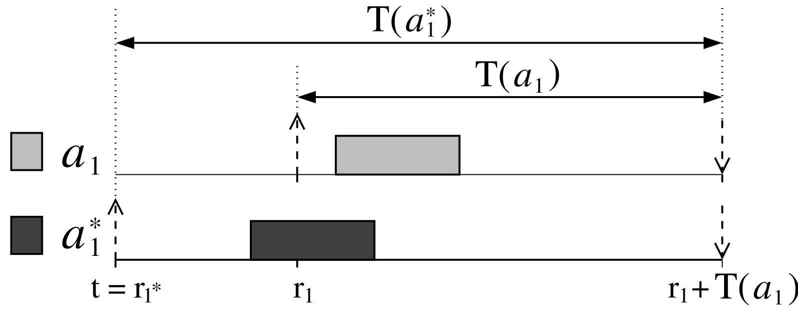


Figure 3.37: Future release schedulability

time: $P(a_c^*) = P(a_c) \wedge C^\tau(a_c^*) = C^\tau(a_c)$, *but with the extended period* $T(a_c^*) = T(a_c) + r_c - t$, *released on* π_x *at the time instant* t *is online schedulable.*

Proof. Proven by contradiction.

• Assume that a_c^* is online schedulable, but a_c is not. As a_c is not schedulable, it did not receive the required $C^\tau(a_c)$ computation time units during its inter-arrival period (the interval between r_c and $r_c + T(a_c)$). Since a_c^* is schedulable and has the same schedulability requirements ($C^\tau(a_c^*) = C^\tau(a_c)$), this means that it performed some computation before r_c. The fact that it performed some computation during the interval between r_{c^*} and r_c, suggests that an eventual busy interval of higher priority backlog caused by a_p completed before r_c. Thus, the offline schedulability condition (Equation 3.71) is sufficient, since it covers the worst-case. As a_c is offline schedulable on π_x excluding a_p, it has to be online schedulable as well. The contradiction has been reached.

• Assume that a_c^* is not online schedulable, but a_c is. As a_c is schedulable, it received the required $C^\tau(a_c)$ computation time units during its period, which is the interval between r_c and $r_c + T(a_c)$. Since a_c^* has a larger period, which is the interval between r_{c^*} and $r_c + T(a_c)$, and of which a period of a_c is just a subset, it should have received at least the same amount of the computation time. As $C^\tau(a_c^*) = C^\tau(a_c)$, it straightforwardly follows that a_c^* has to be schedulable as well. The contradiction has been reached. □

The implications of Theorem 15 are that, during its release, a semi-schedulable parent application can perform the online schedulability test for the next future release of its semi-schedulable child and make a choice regarding its own future executions, such that the schedulability of a child application is preserved on at least one of semi-schedulable cores. By co-scheduling their executions, semi-schedulable applications can safely co-exist within the system (see Theorem 16).

Theorem 16. *Consider two semi-schedulable applications, a parent* a_p *and a child* a_c*, which share only two cores -* π_x *and* π_y*. Assuming that* π_x *and* π_y *are fully operational,* a_p *and* a_c *can safely co-exist within the same system without missed deadlines.*

Proof. Proven by induction. Observe the time interval between two consecutive releases of a_p, termed the *step*.

• Step one. Since this is the first release of a_p, no backlog workload exists which can jeopardise the schedulability of a_c, thus an offline schedulability excluding a_p (which holds) is sufficient. Two scenarios are possible. If a_c has already released a job, a_p may safely select either the other core, or one of the cores which a_p and a_c do not have in common. If a_c didn't release its first job yet, a_p can, due to the inexistence of backlog, safely choose any of the cores and leave the other core for a_c. Note, that a_c also has the possibility to choose one of the cores which it does not share with a_p. In either case, the next release of a_c will be online schedulable.

• Step $n+1$. At the beginning of the step $n+1$, a_p releases its job. As the assumption is that until the end of the step n no missed deadlines occurred, assume that the previous completed execution of the job of a_c occurred on π_x. If the new execution of a_c already started before the step $n+1$, it had either safely continued its execution on the same core, or had selected some of the non-shared cores. Conversely, if the new release of a_c is yet to occur, then by applying Theorem 15 and Equation 3.72, a_p will check whether it can migrate to π_x and force a_c to go to π_y (or other cores), or not. In either case, the next release of a_c will be online schedulable. □

3.8.7 Blind Synchronisation

So far, it has been proven that as long as both semi-schedulable cores remain operational, semi-schedulable applications may safely co-reside in the same system and organise their executions in such a way that none of them misses a deadline. Yet, if the online schedulability test in not performed by solving Equation 3.72, but rather by solving a more pessimistic Equation 3.73, occasionally it may happen that the test reports the unschedulability of the child on both cores. In such cases, both semi-schedulable applications temporarily enter a *blind synchronisation mode*. By Definition 10, if a parent always executes on one semi-schedulable core, and the child on another, none of them can miss a deadline. This fact is exploited during the blind synchronisation mode, as follows:

Rule 1. *If, the core last used, from the semi-schedulable core pair, by each of the two applications (parent/child), is **different**, then each of the applications should choose the same core as before, over the other one.*

Rule 2. *If, the core last used, from the semi-schedulable core pair, by each of the two applications (parent/child), is **the same**, then whichever of them was released last on that core, should choose that core over the other one; the other application should accordingly choose the latter one over the former.*

Rule 3. *Both semi-schedulable applications can freely execute on other cores which are not semi-schedulable.*

Since there were no missed deadlines prior to the blind synchronisation mode, and given that during it no migrations of semi-schedulable applications across semi-schedulable cores will occur, from Definition 10 and Rules 1-3 it follows that no missed deadlines of these applications can occur.

3.8.8 Parent-Child Relationship

The semi-schedulability was earlier defined in the context of application pairs, i.e. a child application can have only one parent application, and vice versa. Yet, it is trivial to see that Theorems 15-16 also hold even under an extended definition of semi-schedulability wherein a parent application may have multiple semi-schedulable children applications but each child still has only one parent. For example, a 4-dispatcher parent may share two cores with one child and two different cores with another child. One could also consider allowing the children of the same parent to share cores with each other. However, such an extended model (i) would require additional proofs and (ii) more importantly, would additionally reduce the flexibility of the approach, due to the necessity to perform co-scheduling not just between a parent and its children, but also between the children of a common parent. Due to the aforementioned reasons, at this point the multiple-children approach is not employed, and during the experimental evaluation only a 1:1 parent-child relationship is considered.

Note that the semi-schedulability property can be also studied from the perspective of *mode changes* [81], where the (in)existence of semi-schedulable applications on respective cores can be perceived as different system modes. Consequently, the analysis can be performed at design time, in order to deduce under which conditions semi-schedulable applications can migrate between cores. With this approach, when releasing its job, a parent would not need to perform an online schedulability test for the next child's release. This is a potential topic for future work.

3.8.9 Schedulability and Agreement Protocols

The aforementioned schedulability constructs are employed at runtime in the following way. When an application runs its agreement protocol, all its dispatchers on currently operational cores participate. For clarity purposes, the protocol can be simplified to the extent that every dispatcher reports with a single variable $s \in \{\top, \bot\}$ about the service it can offer, regarding the next job release on its core. $\top$ means that a dispatcher can guarantee the execution on its core without missing a deadline, and $\bot$ that no guarantees can be provided.

- If a dispatcher is offline schedulable and its core is not selected for shutting down $s = \top$.
- If a dispatcher is a semi-schedulable parent and its execution will not cause the online unschedulability of a semi-schedulable child on all common cores and its core is not selected for shutting down $s = \top$.
- If a dispatcher is a semi-schedulable child and it passes the online schedulablity test on its core and its core is not selected for shutting down $s = \top$.
- If a dispatcher is not offline schedulable, nor semi-schedulable parent or child, and it passes the online schedulability test and its core is not selected for shutting down $s = \top$.
- In all other cases $s = \bot$.

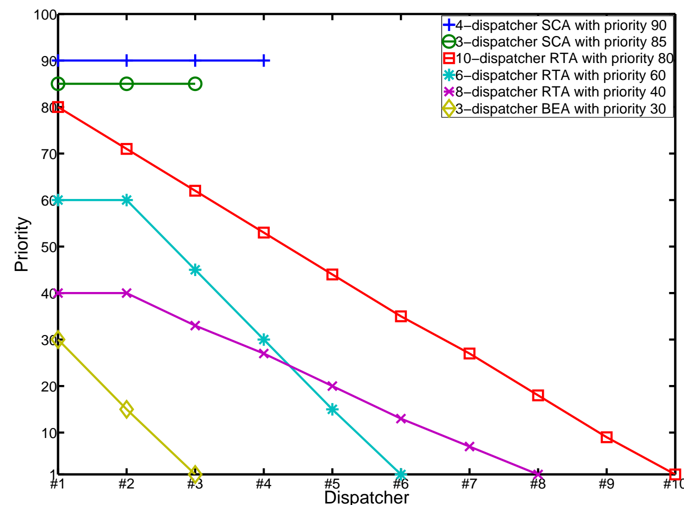


Figure 3.38: Priority assignment upon dispatchers

3.8.10 Priority Assignment and Mapping

The priority assignment and the mapping are mutually dependent activities, hence are presented in an interleaved fashion. The process starts with the most critical workload – SCA.

3.8.10.1 Mapping SCA

Since these applications tolerate no missed deadlines even under the worst-case conditions (with at most K concurrent core shutdowns), the fundamental mapping requirement for each SCA is to have at least $K+1$ dispatchers, and all of them mapped as offline schedulable. If any dispatcher can not be mapped as offline schedulable, a mapping process declares a failure. Therefore, all SCA dispatchers are assigned default priorities of their respective applications (see applications with the priorities 90 and 85 in Figure 3.38).

SCA commence the mapping on an empty system, so most (if not all) of their dispatchers will have the possibility to choose from multiple cores. An analogy can be made with the bin-packing theory; dispatchers represent elements, cores symbolise bins, and the possibility of an element to fit in the bin is equivalent to testing a dispatcher's response-time on a core (Equation 3.71), where a higher response-time is interpreted as a better mapping option (i.e. more efficiently packed elements). As an additional constraint, dispatchers of the same application can not go to the same core. Three possible mapping options of SCA are investigated: (i) Best-Fit, (ii) Worst-Fit, (iii) Alternate-Fit (alternately mapping dispatchers of an application with the Best-Fit and Worst-Fit techniques).

3.8.10.2 Mapping RTA

Once all SCA are mapped, the focus is on mapping RTA. As already stated, RTA require schedulability guarantees when the system is fully operational. Therefore, the first dispatcher of each RTA

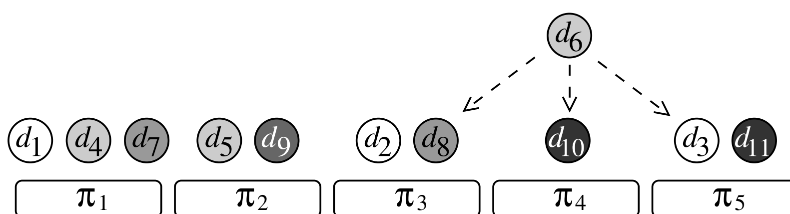


Figure 3.39: Speculative mapping

is assigned the application's priority and subsequently mapped as offline schedulable. If this is not possible, the second dispatcher is also assigned the application's priority, and both are attempted to be simultaneously mapped as semi-schedulable children of some already mapped higher priority application which is a potential semi-schedulable parent. If this is not possible either, the mapping process declares failure. When choosing the core to map the first dispatcher (in the case of offline schedulability) or when choosing the semi-schedulable parent to map the first and the second dispatcher (in the case of semi-schedulability), possible mapping options are similar to those of SCA: (i) Best-Fit, (ii) Worst-Fit and (iii) Alternate-Fit.

Once this is done, there is no need to keep the priority of the rest of the application's dispatchers at the same level. Indeed, decreasing their priorities allows to preserve more schedulability resources for lower-priority RTA whose mapping did not start yet. Hence, after fulfilling the mapping condition (offline schedulability or semi-schedulability), the rest of the dispatchers of an application are assigned linearly decreasing priorities, with the last dispatcher having the least possible system priority, and they all undergo *speculative mapping* (see the next section). An example of RTA priority assignment is given in Figure 3.38, where the application with the priority 80 managed to claim offline schedulability, while applications with priorities 60 and 40 could only claim semi-schedulability.

3.8.10.3 Speculative mapping

Unlike previous mapping stages, where the focus was on providing schedulability guarantees, when mapping speculatively the emphasis is on improving the system's overall runtime behaviour. In other words, dispatchers should not be necessarily mapped to cores where they can claim offline or semi-schedulability, but rather to cores where they have a high chance of claiming online schedulability at runtime. Therefore, the criterion for mapping is no longer the response-time, but the per-core utilisation, which is calculated as the sum of the utilisations of all contained dispatchers. The individual per-dispatcher utilisations are computed as follows: each dispatcher carries a fraction of the application's computation utilisation $U^{\tau}(a_i) = \frac{C^{\tau}(a_i)}{T(a_i)}$, but offline schedulable and semi-schedulable ones carry double the weight, as likelier to be elected.

The speculative mapping is explained with an illustrative example given in Figure 3.39, where dispatchers of the same color belong to the same application. For simplicity reasons, assume that (i) all applications have the same utilisation $U^{\tau}(a_i) = \frac{C^{\tau}(a_i)}{T(a_i)} = u$, (ii) each dispatcher is either offline

or semi-schedulable, and (iii) each dispatcher inherits the priority of its application. Consider that a dispatcher d_6, which belongs to the application with the lowest priority, will undergo a speculative mapping. The cores π_1 and π_2 are not possible mapping options, because the application of d_6 already has dispatchers there (d_4 and d_5, respectively). From the remaining cores ($\pi_3 - \pi_5$), a dispatcher is mapped to the one with the globally minimal utilisation. For example, the utilisation of the core π_3 is equal to the sum of utilisations of d_2 and d_8. Their individual utilisations are $\frac{1}{3}u$ and $\frac{1}{2}u$, as their applications have 3 and 2 dispatchers, respectively. After computing the utilisation of every core, the conclusion is that the best mapping option is the core π_4 (see Table 3.7).

Table 3.7: Speculative mapping computation for Figure 3.39

Core	π_1	π_2	π_3	π_4	π_5
Utilisation	N/A	N/A	$\frac{5}{6}u$	$\mathbf{\frac{1}{2}u}$	$\frac{5}{6}u$

3.8.10.4 Mapping BEA

When mapping BEA the objective is to spread the ratios of BEA missed deadlines across all BEA as evenly as possible, i.e. maintain this particular notion of fairness, as described in Section 3.8.3. Thus, all BEA dispatchers are mapped speculatively. In order to further equalise the consumption of schedulability resources, dispatchers of the same application are assigned linearly decreasing priorities (see the application with the priority 30 in Figure 3.38). In many cases this allows the first dispatcher of a lower priority application to have a higher priority than the second and subsequent dispatchers of some higher priority applications (even RTA), which gives it a scheduling precedence and contributes to the intended fairness.

During the entire mapping process an ordered list of all dispatchers that are still not mapped is maintained. The ordering criterion is the non-increasing priority. The dispatchers are removed from the list and subsequently mapped in that order (e.g. in the example given in Figure 3.38 first the application with the priority 90 is entirely mapped, then also entirely the application with the priority 85, then the dispatchers $1-3$ of that with the priority 80, then the dispatchers $1-2$ of that with the priority 60, etc.). The rationale for this decision is that a currently mapped dispatcher does not have an influence on already mapped (higher-priority) workload, which reduces the complexity of the entire process from sub-quadratic – $O(|\mathscr{D}(\mathscr{A})|^2)$ to linear – $O(|\mathscr{D}(\mathscr{A})|)$, where $|\mathscr{D}(\mathscr{A})| = \sum_{\forall a_i \in \mathscr{A}} |\mathscr{D}(a_i)|$ denotes the total number of dispatchers in the application-set. Note that re-orderings of the list may occasionally be required in cases where RTA claim semi-schedulability, and hence the priorities of their respective unmapped dispatchers have to be elevated.

3.8.11 Experimental Evaluation

In this section, the objective is to explore the impacts of different priority assignment and mapping strategies on several important aspects, namely: (i) schedulability guarantees, (ii) the runtime behaviour assuming no core shutdowns, (iii) the runtime behaviour assuming core shutdowns. The

simulations were performed on the extended version of the simulator *SPARTS* [73]. The simulation parameters are summarised in Table 3.8. An asterisk sign denotes a randomly generated value, assuming a uniform distribution.

Table 3.8: Analysis and simulation parameters for Section 3.8.11

NoC topology and size	**2-D mesh with 10 × 10 routers**
Application-set size $\lvert\mathscr{A}\rvert$	**200 applications**
SCA application period $T(a_i)$	$[\mathbf{30}-\mathbf{50}]^*$ **ms**
RTA application period $T(a_i)$	$[\mathbf{30}-\mathbf{100}]^*$ **ms**
BEA application period $T(a_i)$	$[\mathbf{0.1}-\mathbf{1}]^*$ **s**
Computation deadline $D^{\tau}(a_i), \forall a_i \in \mathscr{A}$	$\boldsymbol{T(a_i)}$
Comm. and memory deadlines $D^{\eta}(a_i) \wedge D^{\mu}(a_i), \forall a_i \in \mathscr{A}$	**0 ms**
Application utilisation $U^{\tau}(a_i) = \frac{C^{\tau}(a_i)}{T(a_i)}$	$(\mathbf{0}-\mathbf{0.7})^*$
Application breakdown (average) { SCA, RTA, BEA }	$\{\mathbf{10\%}, \mathbf{20\%}, \mathbf{70\%}\}$
Simulated time	**100 s**

Note that the application breakdown presents average values, and that in some generated application-sets the amount of SCA can be as low as 6% and as high as 15%, however, the average value is 10%, as written in Table 3.8. These variations occur because the individual task parameters are assigned after the generation of task types.

3.8.11.1 Experiment 1: RTA Priorities and Semi-Schedulability

The objective of this experiment is to observe the impacts of different priority assignment techniques as well as the semi-schedulability on the provided schedulability guarantees. Recall, that the fundamental requirement is that: (i) every dispatcher of every SCA is offline schedulable, and (ii) every RTA has at least one dispatcher as offline schedulable, or a pair of dispatchers as semi-schedulable children. All application-sets, for which a mapping $\mathscr{M}$ can be found, such that the aforementioned objectives are fulfilled, are referred to as *schedulable*.

In this experiment, each application consists of 8 dispachers, and the mapping is performed by employing the Best-Fit mapping technique. Three different approaches are analysed: (i) all dispatchers of one application have the same priority, (ii) the priority assignment upon dispatchers is performed as described in Section 3.8.10, and (iii) the priority assignment is as in the previous case, but the semi-schedulability property (SS) is employed. The system utilisation (x-axis) is varied and the amount of schedulable application-sets (y-axis) is observed. Figure 3.40 shows that assigning priorities as described in Section 3.8.10 is an efficient strategy. Specifically, allowing dispatchers of RTA to have decreasing priorities helps to preserve more schedulability resources for lower-priority RTA that are yet to be mapped. Additionally, the tests reported a noticeable improvement when the SS technique is employed. As these strategies have proven to be beneficial, in the rest of the experiments both the semi-schedulability and the priority assignment, as proposed in Section 3.8.10, are employed.

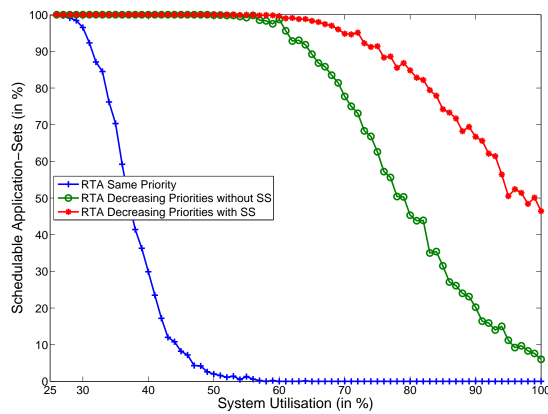


Figure 3.40: Impact of RTA priorities and SS on schedulability guarantees

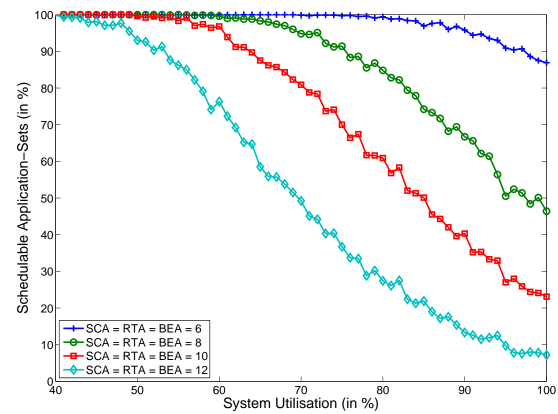


Figure 3.41: Impact of number of dispatchers on schedulability guarantees

3.8.11.2 Experiment 2: Number of Dispatchers

In this experiment, the objective is to investigate how the number of SCA dispatchers impacts provided schedulability guarantees. Again, the Best-Fit mapping technique is employed. Four different cases are analysed, where each application has $6, 8, 10$ and 12 dispatchers. The varied parameter is the system utilisation (x-axis), while the observed value is the number of schedulable application-sets (y-axis). Figure 3.41 demonstrates that the price of having more SCA dispatchers is expensive in terms of schedulability resources, however, at the same time it improves the resilience of the system and allows a higher number of concurrent core shutdowns without SCA missed deadlines.

3.8.11.3 Experiment 3: Mapping Strategies

The focus of this experiment is on different mapping techniques. Each application has 8 dispatchers. Three different approaches are analysed, where the applications were mapped with (i) the Worst-Fit technique, (ii) the Alternate-Fit technique and (iii) the Best-Fit technique, as described in Section 3.8.10. Again, the varied parameter is the system utilisation (x-axis), while the observed value is the number of schedulable application-sets (y-axis). From Figure 3.42 it is visible that the Worst-Fit manages to map the least number of application-sets as schedulable. The other two techniques perform similar to each other, although the Alternate-Fit shows marginal improvements, because it tends to distribute the dispatchers more diversely, which in some cases results in better opportunities for semi-schedulability.

3.8.11.4 Experiment 4: Online Schedulability Tests w/o Remaining Computation Times

Since the online schedulability tests are performed frequently, the performance of *LMM* depends on the trade-off between the complexity and the efficiency of these tests. In some scenarios performing the test by solving Equation 3.72 may be undesirably expensive, as it requires a fixed-point search algorithm and also the knowledge about remaining computation times. For that purpose,

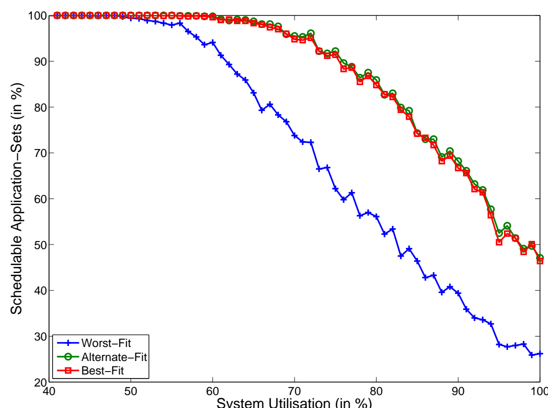


Figure 3.42: Impact of mapping strategies on schedulability guarantees

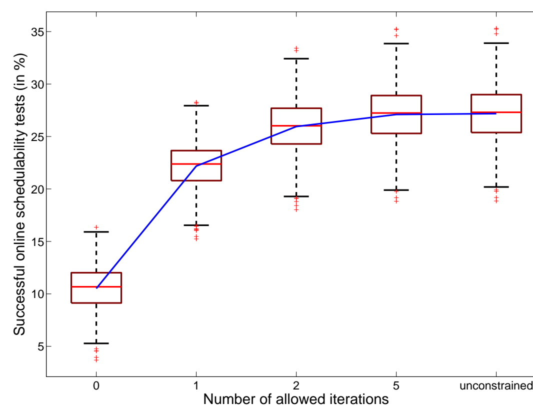


Figure 3.43: Efficiency of online tests when using remaining computation times

a lighter test was proposed (Equation 3.73), where a single computation is performed without the knowledge about the remaining computation times. In this experiment, it is investigated whether the lighter tests are practical.

Each application has 8 dispatchers, and the Best-Fit mapping technique is employed. The system utilisation is fixed to 80%. The execution is simulated and the number of successful online schedulability tests is measured. First, the tests are performed by taking into account the exact remaining computation times. The number of allowed iterative computations is varied, and if the value is not obtained within a given limit, a single computation is performed with the input $R^{\tau}(a_i) = D^{\tau}(a_i)$ in Equation 3.72. Figure 3.43 shows how the success ratio of the online schedulability test (y-axis) changes with the number of allowed iterations (x-axis). As seen, even if only two iterations are allowed, the efficiency of the test compared to the exact test (i.e. unconstrained iterations) is barely affected. This is because the test converges fast anyway, almost always within few iterations. This is not surprising, as the applications contributing interference, in the recurrence relation, are few (i.e. just those that have dispatchers on the considered core), unlike in global scheduling. In this sense, the analysis of LMM is quite scalable.

Similarly, Figure 3.44 shows the success ratio of the online schedulability tests, but this time by being agnostic with respect to remaining computation times. The average value from the previous figure is also plotted, so as to ease the visual comparison. The trends are similar to the previous case, only a few iterations are needed. Moreover, being agnostic with respect to remaining computation times, as expected, has a negative effect, however the same is very mild. Thus, a light online test which is agnostic and has the limit of 5 iterations manages to succeed in more than 90% of the cases which were successful by the exact test (Equation 3.72). This crucial finding further motivates the research related to *LMM*.

3.8.11.5 Experiment 5: Number of Dispatchers and Runtime

In this experiment, the objective is to investigate how the number of RTA and BEA dispatchers influences the runtime behaviour of the system. In other words, is it beneficial to have more RTA

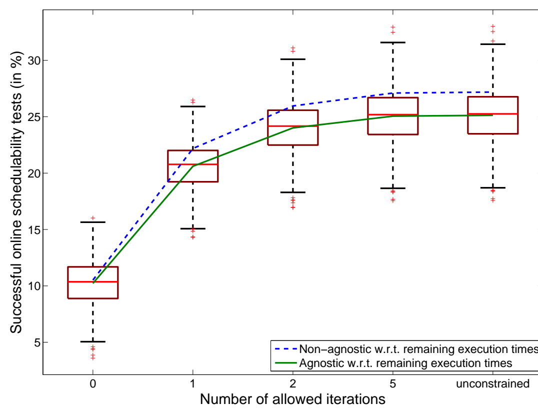


Figure 3.44: Efficiency of online tests without remaining computation times

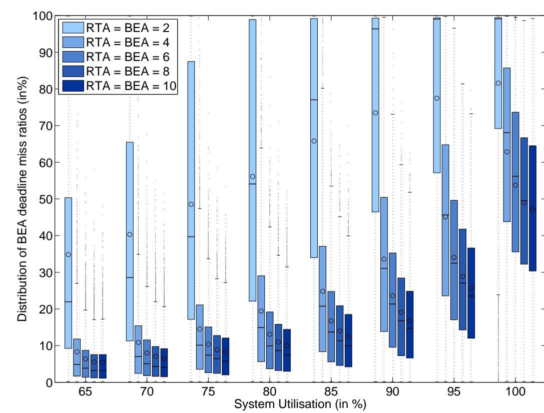


Figure 3.45: Impact of number of dispatchers on runtime behaviour, without core shutdowns

and BEA dispatchers? The following parameter setup is used: $K = 7$, i.e. a system should allow at most 7 concurrent core shutdowns. Thus, all SCA have 8 dispatchers. The number of RTA and BEA dispatchers is varied in the range $[1 - 10]$. The Best-Fit mapping technique was used, and only application-sets which are schedulable with this parameter setup are considered. First, observe the behaviour of the system when no core shutdowns occur. Given that in these conditions no SCA, nor RTA missed deadlines can occur, of interest is the distribution of BEA missed deadline ratios. Note that despite of missing their deadlines, BEA jobs continue their computation. The execution is simulated for different system utilisations, and BEA missed deadlines are captured. Figure 3.45 shows that schemes with fewer dispatchers are more rigid and concentrate all BEA missed deadlines among very few applications. Conversely, schemes with more dispatchers clearly benefit from their flexibility, in a sense that BEA missed deadlines are evenly distributed among all applications. These trends do not reach a saturation point, but show systematic improvements as the number of dispatchers increases. This also validates the efficiency of the priority assignment techniques, and proves that by assigning priorities in a strategic manner one can benefit from the high number of dispatchers per application, and yet efficiently avoid the "suffocation effect" among applications. Note, for RTA = BEA = 1, all BEA missed deadline ratios are 100%. For better clarity, this case is omitted from Figure 3.45.

Again, the runtime behaviour is investigated, but this time assuming core shutdowns. The duration of each shutdown is 1 second. In this and the next experiment the parameter P stands for the individual per-core probability of being selected for at least one shutdown, P^2 for at least two shutdowns, etc. All shutdowns of all cores must occur within the simulated interval. Time instants at which each individual core will experience a shutdown are randomly generated, but without violating a constraint that at most $K = 7$ of them can be selected concurrently. The system utilisation is fixed to 80%, the parameter P (x-axis) is varied, and the average number of RTA missed deadlines (y-axis) is observed. Figure 3.46, shows a clear benefit of having more dispatchers, for every value of P. A slight increase in the number of dispatchers may improve the resilience towards core shutdowns even by one order of magnitude, while any additional increase clearly contributes to

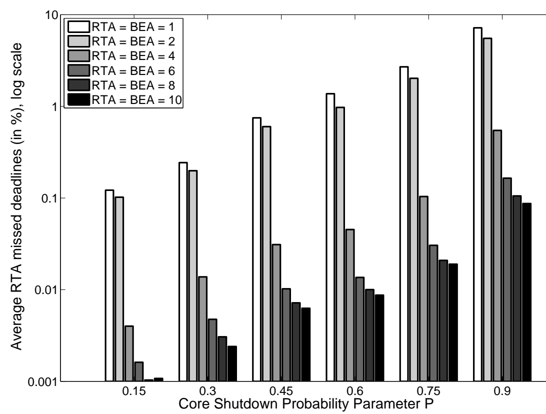


Figure 3.46: Impact of number of dispatchers on runtime behaviour, with core shutdowns

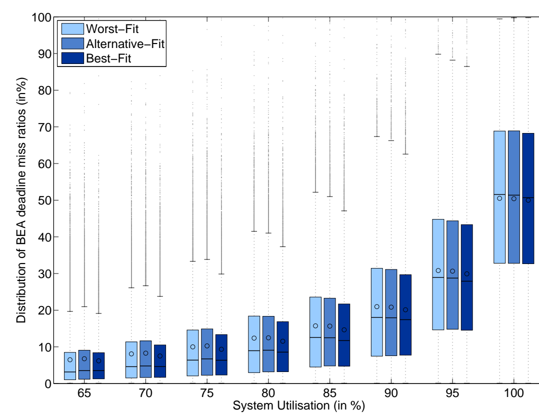


Figure 3.47: Impact of mapping strategies on runtime behaviour, without core shutdowns

the system flexibility to tolerate more frequent core shutdowns.

3.8.11.6 Experiment 6: Mapping strategies and Runtime

In this experiment, the objective is to investigate how different mapping techniques influence the runtime behaviour of the system. Again, $K = 7$. Each application has 8 dispatchers, and again only schedulable application-sets are considered. First, the system behaviour is observed when no core shutdowns occur and the focus is on the distribution of BEA missed deadline ratios. All three proposed mapping techniques for different system utilisations (x-axis) are simulated and the ratios of BEA missed deadlines (y-axis) are captured. Figure 3.47 shows the results. It is noticeable that the Best-Fit technique achieves the best results, although the differences are very subtle and almost negligible.

Now, the runtime behaviour is observed again, but this time with core shutdowns. The system utilisation is 80% and the parameter P is varied (x-axis). The focus is on RTA missed deadlines (y-axis). Figure 3.48 suggests that all techniques demonstrate a comparable performance.

3.8.11.7 Experiment 7: Blind synchronisation

In this experiment, the focus is on the blind synchronisation mode (BSM) and the frequency of its occurrences. The assumed setup is identical to that of Experiment 4, with the only difference that the releases of semi-schedulable applications which cause the BSM are observed (y-axis of Figure 3.49). The varied parameter is the allowed number of iterations in the schedulability test recurrence (x-axis). For each value of the allowed number of iterations the simulations are performed, assuming two types of online schedulability tests, ones which are agnostic with respect to remaining execution times, and ones which are not. It comes as no surprise that the agnostic tests, due to being more pessimistic, cause more frequent occurrences of the BSM, than the respective non-agnostic ones. However, the differences are negligible. The explanation for this finding is

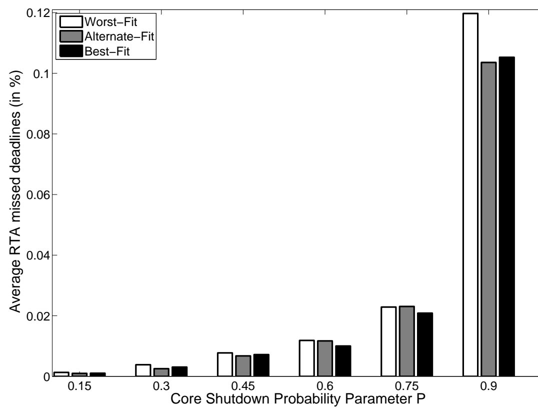


Figure 3.48: Impact of mapping strategies on runtime behaviour, with core shutdowns

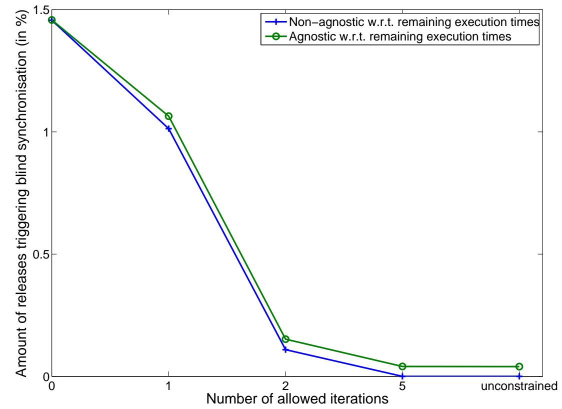


Figure 3.49: Blind synchronisation mode (BSM)

twofold. First, Experiment 4 demonstrated that the pessimism of the agnostic tests, when compared to the respective non-agnostic ones, is not significant. Second, in many cases, the BSM requires specific (worst-case) conditions, which do not occur frequently during runtime.

As expected, allowing more iterations decreases the occurrences of the BSM, because the respective online schedulability tests become less pessimistic. This coincides with the findings of Experiment 4. In any case, even when using the most pessimistic tests (e.g. Equation 3.73), the BSM is triggered, on average, in just 1.5% of the releases of semi-schedulable applications. This shows that the conditions leading to the BSM arise very rarely during runtime, and also shows that for many semi-schedulable application pairs the BSM cannot occur, not even theoretically, irrespective of the employed online schedulability test.

3.8.12 Discussion

Assigning priorities as proposed in Section 3.8.10 proved to be an efficient approach. Also, the semi-schedulability exhibited a huge positive impact on schedulability guarantees. Assigning the application's default priority to all $K+1$ dispatchers of SCA is costly, in terms of schedulability resources, but achieves the required schedulability guarrantee (i.e. at up to K concurrent core shutdowns). Understandably, providing strong guarantees to SCA, for the event of core shutdowns (which is the main objective), commensurately "withholds" resources from RTA and BEA, but this is mitigated to a large extent by the flexibility of *LMM*.

Having more RTA and BEA dispatchers proved to be beneficial in both schemes, with and without core shutdowns. Due to the efficient priority assignment technique, schedulability guarantees for RTA are not influenced by the number of dispatchers per application. The additional system flexibility, brought by multiple dispatchers, indirectly through RTA and directly through BEA, contributes to the equal distribution of missed deadlines among BEA (assuming no core shutdowns) and minimises the number of RTA missed deadlines (assuming core shutdowns). However, as the number of dispatchers increases, the benefits from additional dispatchers start to level

off, which may be an important factor when the communication delays (Sections 3.3-3.5) are taken into account as well.

Mapping with different mapping strategies has almost negligible effects. The Alternate-Fit approach is the most efficient in providing schedulability guarantees, while the Best-Fit technique is the best in terms of runtime behaviour, both with and without core shutdowns. The Worst-Fit approach performs worse than both the aforementioned techniques, in all investigated categories and its use cannot be justified.

Performing a light online schedulability test (agnostic with respect to remaining execution times, with at most 5 iterations) is in more than 90% of the cases as good as performing an exact test (Equation 3.72), while in only 0.04% of the releases of semi-schedulable applications it causes the blind synchronisation mode.

It is apparent that there exists no single strategy which yields the best results under all circumstances. Facts such as the purpose of the system, the amount and the nature of the workload, the maximum number of concurrent core shutdowns K, core shutdown policies, the tolerable amount of RTA/BEA missed deadlines, are only few factors, out of many, which a system designer should take into account when choosing the strategy. One can perceive the mapping process as an adaptive activity, where different strategies are attempted until reaching the solution with (i) the necessary amount of schedulability guarantees, (ii) the acceptable level of flexibility and resilience towards core shutdowns, and (iii) the satisfactory runtime performance.

Chapter 4

Conclusions and Future Work

During the last decade, many-core platforms became mainstream in many computing areas, e.g. high-performance and general-purpose computing. This trend is not surprising, because, when compared with their ancestors (single- and multi-core systems), many-cores offer numerous beneficial possibilities. For instance, the abundance of processing elements allows to enhance the existing functionalities, as well as to integrate new ones. Furthermore, the transition to the many-core domain gives the possibility to achieve significant design cost reductions, as functionalities previously executed on numerous single- and multi-core devices, can be accommodated within fewer many-core platforms. Moreover, many-cores are highly flexible, and efficient thermal/power management strategies can be implemented by configuring the system behaviour to fit current needs and application workload, while unused cores can be temporary shut down. Finally, the abundance of processing elements allows to develop efficient strategies for improving the platform's resilience to core failures.

However, despite all these benefits, many-core devices are still the next frontier technology in the real-time embedded domain, and their application in this area can be expected in the forthcoming years. The major drawback is the complex system design, which makes the real-time analysis of many-cores a very challenging topic.

The **ultimate objective** of this dissertation is to make many-core platforms more amenable for the real-time analysis. As demonstrated in this thesis, this goal **can be achieved by**:

- an **adequate hardware support** for (i) the message passing communication paradigm and (ii) virtual channels,
- an **efficient OS design** that promotes scalability and message-passing,
- a **mindful and thoughtful worst-case analyses**.

The contributions presented in this dissertation have been divided into two categories. The focus of the first category is on the NoC interconnect, which is one of the most complex-to-analyse shared resources in many-core platforms. First, the novel method for the worst-case analysis was proposed for the type of interconnects that are the most common choice in present many-core

platforms: 2-D mesh NoC with the round-robin arbitration policy and without the support for virtual channels. Then, the focus was on the type of interconnects which are not yet commercially available, however, they are currently considered to be the most suitable for the real-time analysis: 2-D mesh NoCs with the priority-preemptive arbitration policy and the support for virtual channels. Assuming these interconnects and the new hardware feature of the existing platforms, which allows traffic flows to dynamically change virtual channels, a novel packet-routing technique was proposed. With this technique, the worst-case analysis remains unaffected, and yet the requirements for hardware resources (i.e. the number of virtual channels) are significantly reduced.

Then, based on the EDF methodology, which is a well-established concept in the scheduling domain, a novel arbitration policy for NoC routers and the accompanying method for the worst-case analysis were proposed. It has been observed that there are cases where the new method outperforms the state-of-the-art techniques, but there are also cases where the proposed approach is less efficient. However, on average, the novel method yields better results, which further motivates research activities in this domain. Finally, the improvement over the existing methods for the worst-case analyses was proposed, which helps to derive less pessimistic worst-case traffic delay estimates. The proposed improvement exploits the fact that traffic flows can impose interference upon each other only while they are competing for the common NoC resources, i.e. links.

The second set of contributions has been developed around the novel workload execution paradigm called the Limited Migrative Model (*LMM*). The *LMM* approach is inspired by the latest trends in the general-purpose and high-performance computing areas. Specifically, it is based on the fundamental concepts of the multi-kernel OS architecture, and uses the message-passing technique as the communication primitive, which are promising steps towards scalable and predictable many-core systems. First, the model itself was introduced, and the method for the worst-case communication delay analysis was proposed. Then, assuming the aforementioned approach, the three-staged application mapping method was proposed. After that, the focus was on the memory requirements, and the method for the worst-case memory traffic analysis was presented. Finally, the computation requirements of applications are addressed with a coarse-grained method for the worst-case analysis, which has been developed with several simplifying assumptions. This approach presents only an initial step towards the complete method for the worst-case analysis of computation requirements, which is a potential future work. Consequently, the application mapping algorithm was proposed for this coarse-grained method.

It has been demonstrated that *LMM* is a beneficial and promising approach, and as such, represents one possible framework for the integration of many-cores into the real-time embedded domain. The worst-case analysis of *LMM* is not nearly complete, especially in the domain of workload computation requirements. Moreover, a unified application mapping approach is needed, such that it takes into account all three identified aspects (communication, memory and computation requirements) and derives mappings where all posed timing constraints are satisfied. Finally, implementing *LMM* concepts into existing operating systems and observing the runtime behaviour would shed a different light on this model, and would likely elicit new research activities.

Appendix

Theorem 17. *Let $X = \{x_1, x_2, ..., x_n\} \in \mathbb{N}$ be the distances between the neighbouring dispatchers of an application, and let c be the circumference of the application shape. The product of the distances between the neighbouring dispatchers $f(X) = \prod_{\forall x_i \in X} x_i$ reaches the maximum when all the distances are as even as possible.*

Proof. Proven directly. Note that the distances between the dispatchers are natural numbers, which is a subset of real numbers. There are two cases:

1) $\frac{c}{n} \in \mathbb{N}$: In this scenario the results of Theorem 18 hold, i.e. the maximum on the continuous domain of real numbers (superset) is also the maximum on the discontinuous domain of natural numbers (subset). Therefore, in these scenarios the function $f(X)$ reaches the maximum when all the distances are equal, i.e. $x_i = \frac{c}{n}, \forall x_i \in X$.

2) $\frac{c}{n} \notin \mathbb{N}$: In this case the maxima are different. As proven in Theorem 18, $f(X)$ has a unique maximum on a continuous real-number domain, inferring that the function is monotonically increasing from the boundary to the extremum, with respect to each variable, when treating all other variables as constants. Therefore, the maximum on a discontinuous natural-number domain is the point geometrically the closest to the continuous maximum, which corresponds to a set of solutions (multiple maxima) on a discontinuous domain where the distances between dispatchers are either $\lceil \frac{c}{n} \rceil$ or $\lfloor \frac{c}{n} \rfloor$, i.e. $x_i \in \left\{ \lceil \frac{c}{n} \rceil, \lfloor \frac{c}{n} \rfloor \right\}, \forall x_i \in X$.

□

Theorem 18. *Let $X = \{x_1, x_2, ..., x_n\} \in \mathbb{R}$ be a set of real number variables, such that the following holds:*

$$\sum_{\forall x_i \in X} x_i = c \tag{4.1}$$

$$x_i \geq 0, \forall x_i \in X \tag{4.2}$$

The function $f(X) = \prod_{\forall x_i \in X} x_i$ has only one maximum on the domain, and that is the point: $x_1 = x_2 = ... = x_n = \frac{c}{n}$.

Proof. Proven directly. This is a constrained optimisation problem, with one constraint expressed by the equality (Equation 4.1), and n constraints expressed by the inequalities (Inequality 4.2). Let

inequalities be temporary excluded from consideration. The extreme values of the function $f(X)$, subject to the equality constraint, can be found by the *Lagrange Multipliers Method*.

$$f(X) = \prod_{\forall x_i \in X} x_i, \quad g(X) = \sum_{\forall x_i \in X} x_i = c \qquad \Rightarrow \qquad \mathscr{L} = \prod_{\forall x_i \in X} x_i + \lambda \cdot \left(c - \sum_{\forall x_i \in X} x_i \right) \tag{4.3}$$

A new variable λ is called the Lagrange multiplier. According to the first derivative test, the necessary condition for the extreme point is that the partial derivative of the Lagrange function with respect to $\forall x_i \in X$ and λ is equal to 0 (see Equations 4.4-4.6).

$$\frac{\partial \mathscr{L}}{\partial x_1} = \frac{\partial f(X)}{\partial x_1} - \lambda \cdot \frac{\partial g(X)}{\partial x_1} = 0 \qquad \Rightarrow \qquad \prod_{\forall x_i \in X \setminus \{x_1\}} x_i = \lambda \tag{4.4}$$

$$\vdots$$

$$\frac{\partial \mathscr{L}}{\partial x_n} = \frac{\partial f(X)}{\partial x_n} - \lambda \cdot \frac{\partial g(X)}{\partial x_n} = 0 \qquad \Rightarrow \qquad \prod_{\forall x_i \in X \setminus \{x_n\}} x_i = \lambda \tag{4.5}$$

$$\frac{\partial \mathscr{L}}{\partial \lambda} = 0 \qquad \Rightarrow \qquad \sum_{\forall x_i \in X} x_i = c \tag{4.6}$$

There are two cases: 1) $\lambda = 0$ and 2) $\lambda \neq 0$.

1) $\underline{\lambda = 0}$: This is possible only if at least two of the variables are also equal to 0, that is $\exists x_i \in X, \exists x_j \in X \mid x_i = x_j = 0 \land i \neq j$. These points are both critical and stationary, and therefore should be further examined by the second derivative test.

2) $\underline{\lambda \neq 0}$: There exists only one point and that is $x_1 = x_2 = ... = x_n = \frac{C}{n}$. This point is also critical and stationary. It also holds for this point that it can be examined by the second derivative test.

Additionally, due to the inequality constraints ($x_i \geq 0, \forall x_i \in X$), it is necessary to check the boundaries of the solution space as well. The boundaries are represented with the solutions where only one of the variables is 0, (i.e. $\exists x_i \in X \land \nexists x_j \in X \mid x_i = x_j = 0 \land i \neq j$). Those are called boundary points and there exists no test to prove their properties, they have to be individually checked. In this case it is easy; it is obvious that those points represent the minima on the domain, since $f(X) = 0$ for all of them.

The next step in the analysis is the second derivative tests for the cases 1) and 2). It is conducted in the form of the *Bordered Hessian*. The process consists of finding the first partial derivatives of $g(X)$ with respect to $\forall x_i \in X$, then finding the second partial derivatives of $f(X)$ also with respect to $\forall x_i \in X$ and finally putting them into the matrix called the Bordered Hessian. The general form of the Bordered Hessian is represented by Figure 4.1.

1) $\underline{\lambda = 0}$: Figure 4.2 presents the Bordered Hessian for the case where $\lambda = 0$. The variable z_{ij} stands for the second partial derivative with respect to the variables x_i and x_j and it is described

$$\begin{bmatrix} 0 & \frac{\partial g}{\partial x_1} & \frac{\partial g}{\partial x_2} & \cdots & \frac{\partial g}{\partial x_n} \\ \frac{\partial g}{\partial x_1} & \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial g}{\partial x_2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g}{\partial x_n} & \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Figure 4.1: General form of Bordered Hessian for n variables and one constraint

by Equation 4.7. z_{ij} has a non-zero value only in cases where at most two variables are equal to zero, otherwise it is also equal to zero. A sufficient condition for the local maximum is that the Bordered Hessian is negative definite, i.e. the determinants of its principal minors alternatively change their signs (Equation 4.8). However, from Figure 4.2 it is visible that there always exists some $|H_i| = 0$, thus making this test inconclusive. In such cases, each of the points should be examined individually. Yet, in this case it is obvious that these points represent the minima on the domain, since $f(X) = 0$ for all of them.

$$\begin{bmatrix} 0 & 1 & \cdots & 1 & \cdots & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & \cdots & 0 & \cdots & z_{ij} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 1 & 0 & \cdots & z_{ji} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \end{bmatrix}$$

Figure 4.2: Bordered Hessian for $\lambda = 0$

$$\begin{bmatrix} 0 & 1 & 1 & \dots & 1 & \dots & 1 \\ 1 & 0 & z_{12} & \dots & z_{1i} & \dots & z_{1n} \\ 1 & z_{21} & 0 & \dots & z_{2i} & \dots & z_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 1 & z_{i1} & z_{i2} & \dots & 0 & \dots & z_{in} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{n1} & z_{n2} & \dots & z_{ni} & \dots & 0 \end{bmatrix}$$

Figure 4.3: Bordered Hessian for $\lambda \neq 0$

$$z_{ij} = z_{ji} = \prod_{\forall x_k \in X \setminus \{x_i, x_j\}} x_k \tag{4.7}$$

$$|H_1| < 0, |H_2| > 0, \ldots \quad \Rightarrow \quad sign(|H_i|) = (-1)^i, \forall i \in \{1, \ldots, n\} \tag{4.8}$$

2) $\underline{\lambda \neq 0}$: The Bordered Hessian for this case is represented by Figure 4.3. For z_{ij} also holds Equation 4.7, however, in this case these are all non-zero values. Note that since $x_i = \frac{c}{n}, \forall x_i \in X$, all the second partial derivatives are also equal (Equation 4.9).

$$z_{ij} = z_{ji} = z = \left(\frac{c}{n}\right)^{n-2}, \forall i \in \{1, .., n\}, \forall j \in \{1, \ldots, n\} \mid i \neq j \tag{4.9}$$

When computed, the determinants of the principal minors are $|H_1| = 2z, |H_2| = -3z^2, |H_3| = 4z^3, ..., |H_n| = (-1)^n nz^n$. Since both n and z are strictly positive, the sign of the determinant depends only on the first term of the product: $(-1)^i$, and therefore alternatively changes when successive principal minors are considered. This fulfils the sufficient condition for the maximum, so it can be concluded that the function $f(X)$ has one maximum on the domain, which is located in the point $x_1 = x_2 = ... = x_n = \frac{c}{n}$ and the value is $max(f(X)) = \left(\frac{c}{n}\right)^n$. □

References

[1] Sani Abba and Jeong-A Lee. A parametric-based performance evaluation and design trade-offs for interconnect architectures using fpgas for networks-on-chip. *Microprocessors and Microsystems*, 2014.

[2] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[3] Adapteva. *Epiphany Architecture*. www.adapteva.com/docs/epiphany_arch_ref.pdf.

[4] Hazem Ismail Ali, Luís Miguel Pinho, and Benny Akesson. Critical-Path-First Based Allocation of Real-Time Streaming Applications on 2D Mesh-Type Multi-Cores. In *Proceedings of the 19th IEEE Conference on Embedded and Real-Time Computing and Applications*, 2013.

[5] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. Multi-objective mapping for mesh-based noc architectures. In *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[6] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993.

[7] Theodore Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems Journal*, 2006.

[8] Sundar Balakrishnan and Fusun Ozguner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 1998.

[9] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems Journal*, 2008.

[10] Andrea Bastoni, Björn Brandenburg, and James Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles*, 2009.

[12] Luca Benini and Giovanni De Micheli. Networks on chips: a new soc paradigm. *The Computer Journal*, 2002.

[13] Bruno Bessette, Redwan Salami, Roch Lefebvre, Milan Jelinek, Jani Rotola-Pukkila, Janne Vainio, Hannu Mikkola, and Kari Jarvinen. The adaptive multirate wideband speech codec (amr-wb). *IEEE Transactions on Speech and Audio Processing*, 2002.

[14] Tobias Bjerregaard and Jens Sparso. Implementation of guaranteed services in the mango clockless network-on-chip. *IEE Proceedings - Computers and Digital Techniques*, 2006.

[15] Konstantinos Bletsas and Björn Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, 2009.

[16] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Qnoc: Qos architecture and design process for network on chip. *Journal of System Architecture*, 2004.

[17] Scott Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, 2003.

[18] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley Educational Publishers Inc, 2009.

[19] John Calandrino, James Anderson, and Dan Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[20] Chen-Ling Chou and Radu Marculescu. Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis*, 2007.

[21] Chen-Ling Chou and Radu Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Proceedings of the International Conference on Computer Design*, 2008.

[22] Chen-Ling Chou, Umit 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*, 2008.

[23] William Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 1990.

[24] William Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 1992.

[25] William Dally and Charles Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 1987.

[26] William Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference*, 2001.

[27] Dakshna Dasari, Borislav Nikolić, Vincent Nelis, and Stefan M. Petters. Noc contention analysis using a branch and prune algorithm. *ACM Transactions on Embedded Computing Systems*, 2013.

[28] Jonas Diemer and Rolf Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *International Symposium on Networks-on-Chip*, 2010.

[29] Jeff Draper and Joydeep Ghosh. A comprehensive analytical model for wormhole routing in multicomputer systems. *Journal of Parallel and Distributed Computing*, 1994.

[30] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. M.K. Publishers, 2002.

[31] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *IEEE Computer*, 2009.

[32] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A method of computation for worst-case delay analysis on spacewire networks. In *Proceedings of the IEEEInternational Symposium on Industrial Embedded Systems*, 2009.

[33] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. Using network calculus to compute end-to-end delays in spacewire networks. *SIGBED Rev.*, 2011.

[34] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. A network calculus model for spacewire networks. In *Proceedings of the 17th IEEE Conference on Embedded and Real-Time Computing and Applications*, 2011.

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

[36] Kees Goossens, John Dielissen, and Andrei Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 2005.

[37] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the 3rd Conference on Design Automation and Test in Europe*, 2000.

[38] Mehmet Harmanci, Nuria Escudero, Yusuf Leblebici, and Paolo Ienne. Providing qos to connection-less packet-switched noc by implementing diffserv functionalities. In *International Symposium on System-on-Chip*, 2004.

[39] Jingcao Hu and Radu Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures. In *Proceedings of the 6th Conference on Design Automation and Test in Europe*, 2003.

[40] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Proceedings of the 8th Asia and South Pacific Design Automation Conference*, 2003.

[41] Wei-Lun Hung, Charles Addo-Quaye, Theocharis Theocharides, Yuan Xie, Narayanan Vijaykrishnan, and Mary Irwin. Thermal-aware ip virtualization and placement for networks-on-chip architecture. In *Proceedings of the International Conference on Computer Design*, 2004.

[42] Intel. *Single-Chip-Cloud Computer*, . www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf.

[43] Intel. *Intel® Xeon Phi™*, .
http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[44] Kalray. *MPPA-256 Manycore Processor*.
www.kalray.eu/products/mppa-manycore/mppa-256.

[45] Hany Kashif and Hiren Patel. Bounding buffer space requirements for real-time priority-aware networks. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference*, 2014.

[46] Hany Kashif, Sina Gholamian, and Hiren Patel. Sla: A stage-level latency analysis for real-time communication in a pipelined resource model. *IEEE Transactions on Computers*, 2014.

[47] Shinpei Kato, Nobuyuki Yamasaki, and Yutaka Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[48] Nikolay Kavaldjiev and Gerard Smit. A survey of efficient on-chip communications for soc. In *Proceedings of the 4th Symposium on Embedded Systems*, 2003.

[49] Byungjae Kim, Jong Kim, Sungje Hong, and Sunggu Lee. A real-time communication method for wormhole switching networks. In *Proceedings of the 1998 International Conference on Parallel Processing*, 1998.

[50] Scott Kirkpatrick, Daniel Gelatt, and Mario Vecchi. Optimization by simulated annealing. *Science*, 1983.

[51] Marcio Kreutz, Cesar Marcon, Luigi Carro, Ney Calazans, and Altamiro Susin. Energy and latency evaluation of noc topologies. In *Proceedings of the International Symposium on Circuits and Systems*, 2005.

[52] Rakesh Kumar, Timothy Mattson, Gilles Pokam, and Rob van der Wijngaart. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*. Springer, 2011.

[53] Hugh Lauer and Roger Needham. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems*, 1978.

[54] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[55] Thomas LeBlanc and Evangelos Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *IEEE Parallel and Distributed Processing Symposium*, 1992.

[56] Tang Lei and Shashi Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, 2003.

[57] Ye Li, Matthew Danish, and Richard West. Quest-v: A virtualized multikernel for high-confidence systems. Technical report. http://www.cs.bu.edu/~richwest/quest.html.

[58] Chang Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[59] Zhonghai Lu, Axel Jantsch, and Ingo Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference*, 2005.

[60] Mark Lundstrom. Moore's law forever? *Science*, 2003.

[61] Cesar Marcon, Andre Borin, Altamiro Susin, Luigi Carro, and Flavio Wagner. Time and energy efficient mapping of embedded applications onto nocs. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference*, 2005.

[62] Paris Mesidis and Leandro Soares Indrusiak. Genetic mapping of hard real-time applications onto noc-based mpsocs – a first approach. In *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2011.

[63] 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 *Proceedings of the 7th Conference on Design Automation and Test in Europe*, 2004.

[64] Fahime Moein-darbari, Ahmad Khademzade, and Golnar Gharooni-fard. Cgmap: a new approach to network-on-chip mapping problem. *IEICE Electronics Express*, 2009.

[65] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *Proceedings of the 7th Conference on Design Automation and Test in Europe*, 2004.

[66] Matt Mutka. Using rate monotonic scheduling technology for real-time communications in a wormhole network. In *Proceedings of the 2nd International Workshop on Parallel and Distributed Processing*, 1994.

[67] Lionel Ni and Philip McKinley. A survey of wormhole routing techniques in direct networks. *The Computer Journal*, 1993.

[68] Borislav Nikolić and Stefan M. Petters. Towards network-on-chip agreement protocols. In *Proceedings of the 12th International Conference on Embedded Software*, 2012.

[69] Borislav Nikolić and Stefan M. Petters. Edf as an arbitration policy for wormhole-switched priority-preemptive nocs – myth or fact? In *Proceedings of the 14th International Conference on Embedded Software*, 2014.

[70] Borislav Nikolić and Stefan M. Petters. Real-time application mapping for many-cores using a limited migrative model. *Real-Time Systems Journal*, 2014.

[71] Borislav Nikolić, Konstantinos Bletsas, and Stefan M. Petters. Priority assignment and application mapping for many-cores using a limited migrative model. Technical report, . Available at: http://www.cister.isep.ipp.pt/people/Borislav+Nikolic/publications/.

[72] Borislav Nikolić, Leandro Soares Indrusiak, and Stefan M. Petters. A tighter real-time communication analysis for wormhole-switched priority-preemptive nocs. Technical report, . Available at: http://www.cister.isep.ipp.pt/people/Borislav+Nikolic/publications/.

[73] Borislav Nikolić, Muhammad Ali Awan, and Stefan M. Petters. SPARTS: Simulator for power aware and real-time systems. In *Proceedings of the 8th IEEEInternational Conference on Embedded Software and Systems*, 2011.

[74] Borislav Nikolić, Hazem Ismail Ali, Stefan M. Petters, and Luís Miguel Pinho. Are virtual channels the bottleneck of priority-aware wormhole-switched noc-based many-cores? In *Proceedings of the 21th International Conference on Real-Time Networks and Systems*, 2013.

[75] Borislav Nikolić, Patrick Meumeu Yomsi, and Stefan M. Petters. Worst-case memory traffic analysis for many-cores using a limited migrative model. In *Proceedings of the 19th IEEE Conference on Embedded and Real-Time Computing and Applications*, 2013.

[76] Borislav Nikolić, Patrick Meumeu Yomsi, and Stefan M. Petters. Worst-case communication delay analysis for many-cores using a limited migrative model. In *Proceedings of the 20th IEEE Conference on Embedded and Real-Time Computing and Applications*, 2014.

[77] Christian Paukovits and Hermann Kopetz. Concepts of switching in the time-triggered network-on-chip. In *Proceedings of the 14th IEEE Conference on Embedded and Real-Time Computing and Applications*, 2008.

[78] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the 47th ACM/IEEE Conference on Design Automation Conference*, 2010.

[79] Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *International Symposium on Networks-on-Chip*, 2009.

[80] Adrian Racu and Leandro Soares Indrusiak. Using genetic algorithms to map hard real-time on noc-based systems. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2012.

[81] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems Journal*, 2004.

[82] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of System Architecture*, 2013.

[83] Zheng Shi. *Real-Time Communication Services for Networks on Chip*. PhD thesis, Department of Computer Science, University of York, United Kingdom, 2009.

[84] Zheng Shi and Alan Burns. Priority assignment for real-time wormhole communication in on-chip networks. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008.

[85] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *International Symposium on Networks-on-Chip*, 2008.

[86] Zheng Shi and Alan Burns. Real-time communication analysis with a priority share policy in on-chip networks. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[87] Zheng Shi and Alan Burns. Schedulability analysis and task mapping for real-time on-chip communication. *Real-Time Systems Journal*, 2010.

[88] Zheng Shi, Alan Burns, and Leandro Soares Indrusiak. Schedulability analysis for real time on-chip communication with wormhole switching. *International Journal on Embedded and Real-Time Communication Systems*, 2010.

[89] Hyojeong Song, Boseob Kwon, and Hyunsoo Yoon. Throttle and preempt: a new flow control for real-time communications in wormhole networks. In *Proceedings of the 1997 International Conference on Parallel Processing*, 1997.

[90] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical report inria-00073920, INRIA, France, 1996.

[91] Krishnan Srinivasan and Karam Chatha. A technique for low energy mapping and routing in network-on-chip architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005.

[92] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[93] Tilera. *TILE64™ Processor*. www.tilera.com/products/processors/TILEPro_Family.

[94] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 2009.

[95] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, 2012.