Customizable Memory Controller Architecture and Service Continuity for Off-Chip SDRAM Access in NoC-Based MPSoCs

Khaldon Hassan

▶ **To cite this version:**

Khaldon Hassan. Customizable Memory Controller Architecture and Service Continuity for Off-Chip SDRAM Access in NoC-Based MPSoCs. Micro and nanotechnologies/Microelectronics. Université de Grenoble, 2011. English. NNT : . tel-00656470

HAL Id: tel-00656470
https://theses.hal.science/tel-00656470

Submitted on 4 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Micro et Nano Electronique**

Arrêté ministériel : 7 août 2006

Présentée par

Khaldon HASSAN

Thèse dirigée par **Pr. Frédéric PETROT** et
codirigée par **M. Marcello COPPOLA**

préparée au sein du **Laboratoire TIMA et STMicroelectronics**
dans **l'École Doctorale EEATS**

Architecture De Contrôleur Mémoire Configurable et Continuité de Service Pour l'Accès à la Mémoire Externe Dans Les Systèmes Multiprocesseurs Intégrés à Base de Réseaux Sur Puce

Thèse soutenue publiquement le **2 septembre 2011**
devant le jury composé de :

Mme. Lorena ANGHEL
Professeur, Grenoble INP (TIMA), Présidente

M. Gilles SASSATELLI
Directeur de Recherche, CNRS (LIRMM), Rapporteur

M. Sébastien PILLEMENT
Maître de conférence, Université de Rennes 1 (IRISA), Rapporteur

M. Yves MATHIEU
Professeur, Télécom ParisTech (LTCI), Examinateur

M. Amer BAGHDADI
Maître de conférence,Télécom Bretagne (LabSticc), Examinateur

M. Frédéric PETROT
Professeur, Grenoble INP (TIMA), Directeur de thèse

M. Marcello COPPOLA
Manager, STMicroelectronics, Co-encadrant

Customizable Memory Controller Architecture and Service Continuity for Off-Chip SDRAM Access in NoC-Based MPSoCs

Supervised by Prof. Frédéric PETROT

Khaldon HASSAN

September 2nd 2011

Grenoble

France

ISBN 978-2-84813-172-6

To my loving parents
Hayat & Mohsen
who unconditionally supported me
all along my life

Acknowledgements

This work would not have been possible without the contributions of many people. I would like to thank Mr. Marcello COPPOLA for the opportunity to carry out my master project in STMicroelectronics in Grenoble, and then to start the PhD journey with his great team: Spidergon ST Network-on-Chip Team. I would like to express my sincere gratitude to Pr. Frédéc PETROT for being my academic advisor, and a role model as mentor and scientist.

I would like to thank Riccardo LOCATELLI, who was here from the very beginning and who technically and scientifically supported me during the last 3 years. Special thanks to Giuseppe MARRUCCIA, Valerio CATALANO, Michael SOULIÉ, Florentine DUBOIS and Déborah LALOMIA for being exceptional colleagues at ST. They made my PhD life more exciting and left me lots of unforgettable memories of this period. I would also like to thank all my colleagues in the SLS group at TIMA Lab.

I would like to take this opportunity to extend my gratitude to all my friends in Syria and France, especially to Rami AL BATAL and Yanal WAZAEFI.

Most importantly I would like to acknowledge the constant support and encouragement of my family. My deepest gratitude and thanks to my parents Hayat and Mohsen for their love, encouragement, and belief in me all along my life. My special thanks to my sisters Chourouk and Soulaf, and my brother Iyas, who supported me a lot.

Finally my heartfelt thanks to Valeria MONGELLI for being the greatest girlfriend throughout the long working hours and mood swings to finish the PhD.

Abstract

THE ONGOING advancements in VLSI technology allow System-on-Chip (SoC) to integrate many heterogeneous functions into a single chip, but still demand, because of economical constraints, a single and shared main off-chip SDRAM. Consequently, main memory system design, and more specifically the architecture of the memory controller, has become an increasingly important factor in determining the overall system performance.

Choosing a memory controller design that meets the needs of the whole system is a complex issue. This requires the exploration of the memory controller architecture, and then the validation of each configuration by simulation. Although the architecture exploration of the memory controller is a key to successful system design, state of the art memory controllers are not as flexible as necessary for this task. Even if some of them present a configurable architecture, the exploration is restricted to limited sets of parameters such as queue depth, data bus size, quality-of-service level, and bandwidth distribution.

Several classes of traffic co-exist in real applications, e.g. *best effort* traffic and *guaranteed service* traffic, and access the main memory. Therefore, considering the interaction between the memory subsystem and the interconnection system has become vital in today's SoCs. Many on-chip networks provide guaranteed services to traffic classes to satisfy the applications requirements. However, very few studies consider the SDRAM access within a *system approach*, and take into account the specificity of the SDRAM access as a target in NoC-based SoCs.

This thesis addresses the topic of dynamic access to SDRAM in NoC-based SoCs. We introduce a totally customizable memory controller architecture based on fully configurable building components and design a high level cycle approximate model for it. This enables the exploration of the memory subsystem thanks to the ease of configuration of the memory controller architecture. Because of the discontinuity of services between the network and the memory controller, we also propose within the framework of this thesis an *Extreme End to*

End flow control protocol to access the memory device through a multi-port memory controller. The simple yet novel idea is to exploit information about the memory controller status in the NoC. Experimental results show that by controlling the *best effort* traffic injection in the NoC, our protocol increases the performance of the *guaranteed service* traffic in terms of bandwidth and latency, while maintaining the average bandwidth of the *best effort* traffic.

Key words: memory controller, SDRAM, NoC, MPSoC, performance analysis, traffic classes, end-to-end protocol, modelling.

Résumé

L'ÉVOLUTION de la technologie VLSI permet aux systèmes sur puce (SoCs) d'intégrer de nombreuses fonctions hétérogènes dans une seule puce et demande, en raison de contraintes économiques, une unique mémoire externe partagée (SDRAM). Par conséquent, la conception du système de mémoire principale, et plus particulièrement l'architecture du contrôleur de mémoire, est devenu un facteur très important dans la détermination de la performance globale du système.

Le choix d'un contrôleur de mémoire qui répond aux besoins de l'ensemble du système est une question complexe. Cela nécessite l'exploration de l'architecture du contrôleur de mémoire, puis la validation de chaque configuration par simulation. Bien que l'exploration de l'architecture du contrôleur de mémoire soit un facteur clé pour une conception réussite d'un système, l'état de l'art sur les contrôleurs de mémoire ne présente pas des architectures aussi flexibles que nécessaire pour cette tâche. Même si certaines d'entre elles sont configurables, l'exploration est restreinte à des ensembles limités de paramètres tels que la profondeur des tampons, la taille du bus de données, le niveau de la qualité de service et la distribution de la bande passante.

Plusieurs classes de trafic coexistent dans les applications réelles, comme le trafic de service au mieux et le trafic de service garanti qui accèdent à la mémoire partagée d'une manière concurrente. En conséquence, la considération de l'interaction entre le système de mémoire et la structur d'interconnexion est devenue vitale dans les SoCs actuels. Beaucoup de réseaux sur puce (NoCs) fournissent des services aux classes de trafic pour répondre aux exigences des applications. Cependant, très peu d'études considèrent l'accès à la SDRAM avec une approche système, et prennent en compte la spécificité de l'accès à la SDRAM dans les systèmes sur puce à base de réseaux intégrés.

Cette thèse aborde le sujet de l'accès à la mémoire dynamique SDRAM dans les systèmes sur puce à base de réseaux intégrés. Nous introduisons une architecture de contrôleur de mémoire totalement configurable basée sur des blocs fonctionnels configurables, et pro-

posons un modèle de simulation associé relativement précis temporellement et à haut niveau d'abstraction. Ceci permet l'exploration du sous-système de mémoire grâce à la facilité de configuration de l'architecture du contrôleur de mémoire. En raison de la discontinuité de services entre le réseau sur puce et le contrôleur de mémoire, nous proposons également dans le cadre de cette thèse un protocole de contrôle de flux de bout en bout pour accéder à la mémoire à travers un contrôleur de mémoire multiports. L'idée, simple sur le principe mais novatrice car jamais proposée à notre connaissance, se base sur l'exploitation des informations sur l'état du contrôleur de mémoire dans le réseau intégré. Les résultats expérimentaux montrent qu'en contrôlant l'injection du trafic de service au mieux dans le réseau intégré, notre protocole augmente les performances du trafic de service garanti en termes de bande passante et de latence, tout en préservant la bande passante moyenne du trafic de service au mieux.

Mots clés: contrôleur de mémoire, SDRAM, réseaux intégrés sur puce, multiprocesseurs, analyse de performance, classes de trafic, protocole de bout-en-bout, modélisation.

Contents

General Introduction **xxv**
1 Thesis scope . . . xxviii
2 Thesis organization . . . xxviii

1 Problem Definition **1**
1.1 DDRn SDRAM concepts . . . 3
1.2 Memory controller concepts . . . 5
1.3 Quality of service in networks-on-chip . . . 7
1.4 Continuity of services in NoC-based systems . . . 8
1.5 Experiments . . . 8
1.5.1 Simulation environment . . . 9
1.5.2 Platform configuration . . . 9
1.5.3 Simulation & results . . . 11
1.5.4 Experiments summary . . . 15
1.6 Conclusion . . . 15

2 State of the Art **17**
2.1 Memory controllers . . . 19
2.2 On-chip interconnects . . . 26
2.3 Combined interconnect-memory controller solutions . . . 30
2.4 Conclusion . . . 33

3 Memory Controller Customizable Architecture **35**
3.1 Introduction . . . 37
3.2 DDR3 SDRAM operations . . . 38

3.3 Design abstraction in system modelling . . . 41
3.4 Design approach . . . 41
3.5 Assumptions . . . 42
3.6 Front-end building components . . . 42
3.6.1 Memory mapping . . . 42
3.6.2 Generic queue . . . 43
3.6.3 Capture unit . . . 44
3.6.4 Insertion unit . . . 47
3.6.5 Generic arbiter . . . 51
3.6.6 Flow control . . . 53
3.6.7 Re-ordering unit . . . 55
3.6.8 Summary . . . 55
3.7 Examples of memory controller front-end . . . 56
3.7.1 Memory controller Alpha . . . 56
3.7.2 Memory controller Beta . . . 56
3.7.3 Memory controller Gamma . . . 57
3.7.4 Summary . . . 58
3.8 Back-end building components . . . 59
3.8.1 DDR3 SDRAM commands generator . . . 60
3.8.2 Memory manager . . . 60
3.8.3 Data handler . . . 61
3.9 DDR3 SDRAM model . . . 62
3.10 Conclusion . . . 62

4 Extreme End to End Flow Control Protocol for SDRAM Access 63
4.1 Introduction . . . 65
4.2 Credit-based flow control . . . 66
4.2.1 Analytical model for the *end-to-end* credit-based flow control . . . 67
4.3 End to end flow controls . . . 68
4.4 Pressure on the memory system in modern MPSoCs . . . 69
4.5 Guaranteed service traffic in the memory controller . . . 71
4.6 Saturation risk of the requests queue . . . 71
4.6.1 Problem description . . . 71
4.6.2 Possible solutions . . . 72
4.7 EEEP: Extreme End-to-End Protocol . . . 73
4.7.1 Novel system approach . . . 73
4.7.2 EEEP principle . . . 73

4.7.3 EEEP mechanism . . . 75
4.7.4 *Requests queue* sizing method . . . 75
4.7.5 EEEP guarantees and limitation . . . 77
4.7.6 System modifications to support EEEP . . . 78
4.8 Conclusion . . . 79

5 Implementation of the Customizable Memory Controller Architecture 81
5.1 Development environment . . . 83
5.2 NED language overview . . . 84
5.3 Model structure in OMNeT++ . . . 85
5.4 General description of a building component . . . 86
5.5 Memory controller building components parameters . . . 87
5.5.1 Memory mapping parameters . . . 88
5.5.2 Generic queue parameters . . . 88
5.5.3 Capture unit parameters . . . 88
5.5.4 Insertion unit parameters . . . 88
5.5.5 Generic arbiter parameters . . . 89
5.5.6 Re-ordering unit parameters . . . 89
5.6 EEEP components parameters . . . 89
5.7 Traffic generator . . . 90
5.8 Conclusion . . . 92

6 Experiments and Results 93
6.1 Memory system . . . 95
6.1.1 Memory controller architecture . . . 95
6.2 Standalone tests . . . 96
6.2.1 Memory controller configuration for standalone tests . . . 96
6.2.2 Memory timing tests . . . 97
6.2.3 Priority and ageing mechanism test . . . 99
6.2.4 Summary . . . 102
6.3 EEEP tests . . . 102
6.3.1 Traffic modelling . . . 103
6.3.2 EEEP in a Spidergon NoC-based SoC . . . 103
6.3.3 EEEP in a 2DMesh NoC-based SoC . . . 107
6.3.4 EEEP in an irregular NoC-based SoC . . . 107
6.3.5 Analysis . . . 111
6.4 Conclusion . . . 112

7 Conclusion and Perspectives **113**
7.1 Conclusion 115
7.2 Future work directions 116
7.2.1 3D stacking - wide I/O memories 117
7.2.2 More memory system information exploitation 117
7.2.3 *Extreme End-to-End Protocol* evolution 117

Appendixs **127**

A Problem Definition: *Simulation Platform* **127**
A.1 Spidergon STNoC building blocks 128
A.2 Platform composition 128

B Memory Controller Scheduling Algorithms **131**

C List of Publications **137**

D About the Author **139**

List of Figures

1.1 Simplified architecture of a modern DDR SDRAM 4
1.2 Time to complete a series of memory references without (a) and with (b) access reordering. 6
1.3 Simplified architecture of a memory controller 7
1.4 Off-chip memory access speedup for cache controller read requests when low-priority IPs send read requests 13
1.5 Off-chip memory access speedup of cache controller read requests when low-priority IPs send write requests 13
1.6 Off-chip memory access speedup/slowdown of cache controller write requests when low-priority IPs send write requests 14
1.7 Off-chip memory access speedup/slowdown of cache controller write requests when low-priority IPs send read requests 15

2.1 The logical view of SMC architecture, source [10] 20
2.2 Bandwidth optimized SDRAM controller architecture, source [80] 21
2.3 (a) Quality aware memory controller (b)MIS architecture, source [48] 22
2.4 Organization of the on-chip STFM memory controller, source [56] 24
2.5 Using Impulse to remap the diagonal of a dense matrix into a dense cache line. The black boxes represent data on the diagonal, whereas the gray boxes represent nondiagonal data. Source [12] 25
2.6 The Impulse memory architecture. Source [12] 25
2.7 Predator memory controller architecture, source [1] 25
2.8 Memory map interpretation when using Sonics IMT, source [72] 31
2.9 The proposed memory controller integrated in the slave-side network interface, source [21] 32

3.1 Generic architecture of a memory controller connected to SDRAM devices . . . 38

3.2 Simplified diagram of the DDR3 SDRAM FSM 39
3.3 Examples of memory mapping 43
3.4 Simplified architecture of the generic queue 44
3.5 capture unit in conjunction with a generic queue 46
3.6 Insertion unit in conjunction with a generic queue 48
3.7 Generic arbiter connection with a generic queue 51
3.8 Generic arbiter connection with a generic queue including a capture unit 51
3.9 Ordering aspects in memory controllers 55
3.10 Front-end model for the memory controller Alpha 57
3.11 Front-end model for the memory controller Beta 58
3.12 Front-end model for the memory controller Gamma 59
3.13 Back-end architecture 60

4.1 The front-end and the back-end of a memory controller 65
4.2 *Link level* credit-based flow control 66
4.3 Simplified NI architecture that supports the *end-to-end* credit-based flow control 67
4.4 Overview of TILEpro64 that includes 4 memory controllers, source [78] 69
4.5 SoC consumer portable design complexity trends, source ITRS [37] 70
4.6 Saturation risk of the requests queue in a memory controller 72
4.7 EEEP extend versus other flow controls extend 74
4.8 An overview of the EEEP implemented in a system 75
4.9 EEEP diagram for *best effort* traffic request packets 76
4.10 Memory controller modification to support EEEP 79

5.1 Model structure in OMNeT++: compound and simple modules, gates, connections 85
5.2 The NED description of the generic arbiter 86
5.3 The initialization method of the generic arbiter 86
5.4 The activity method of the generic arbiter 87
5.5 The activity method of the generic arbiter 90
5.6 The header of a stimuli file for the *back-annotated* generation mode 91

6.1 Architecture of the multi-port memory controller 96
6.2 Back-end log file: direction switching and bank preparation delay 98
6.3 Back-end log file: bank interleaving mechanism 98
6.4 Back-end log file: requests format 99
6.5 Memory controller latency histogram for core1 and core2 100
6.6 Moving average bandwidth for core1 and core2 101
6.7 Spidergon NoC-based simulation platform (across last routing) 105
6.8 Performance variation when EEEP is activated in a Spidergon NoC-based SoC . 106
6.9 2DMesh NoC-based simulation platform (XY routing) 108

6.10 Performace variation when EEEP is activated in a 2DMesh NoC-based SoC . . . 109
6.11 irregular NoC-based simulation platform (source routing) 110
6.12 Performace variation when EEEP is activated in an irregular NoC-based SoC . . 111

A.1 Simplified architecture of the simulation platform 130

List of Tables

1.1 DDRn SDRAM timing parameters description . . . 5
1.2 Traffic generator characteristics . . . 10

2.1 Main features of the state of the art memory controllers . . . 26
2.2 Main features of the state of the art networks-on-chip providing QoS . . . 29

3.1 DDR3 SDRAM state digram command definitions . . . 39
3.2 DDR3 SDRAM timing parameters description . . . 40
3.3 Brief description of the generic queue model . . . 45

5.1 Memory mapping parameters . . . 88
5.2 Generic queue parameters . . . 88
5.3 Capturing unit parameters . . . 88
5.4 Insertion unit parameters . . . 89
5.5 Generic arbiter parameters . . . 89
5.6 Re-ordering unit parameters . . . 89
5.7 EEEP parameters . . . 90
5.8 Traffic generator, example of the *constrained random* configuration . . . 91

6.1 Memory controller configuration for standalone tests . . . 97
6.2 Samsung DDR3-800 SDRAM timing parameters . . . 97
6.3 Traffic generators configuration in *constrained random* mode . . . 102
6.4 Memory controller configuration for EEEP tests . . . 103
6.5 Traffic generators configuration in *constrained random* mode for EEEP tests . . 104

A.1 Routing table of both request and response networks . . . 129

List of Algorithms

1 Row hit same direction capture 46
2 Row miss different bank capture 47
3 Master data consistency insertion 49
4 Bank splitting insertion 50
5 Round-Robin, bandwidth scheduling 52
6 Priority then Round-Robin scheduling with bandwidth limiter 54
7 Data consistency flow control 54
8 Re-ordering unit 55
9 Anticipation of the bank preparation commands 61
10 Highest priority capture 132
11 Row hit opposite direction capture 132
12 Global system data consistency insertion 133
13 Priority-based insertion 133
14 Direction grouping insertion 134
15 Round-robin scheduling 134
16 Initialize least-recently-used 134
17 Least-recently-used scheduling 135
18 Least-recently-used update 135
19 Priority scheduling 135

General Introduction

FROM the invention of the integrated circuit until now, the microelectronics industry owes its success to the miniaturization of the transistor on silicon. For nearly 40 years, this miniaturization has been the main factor that increasingly enabled the design of more complex integrated systems. Nowadays, the enhancement of technology processes allows the integration of complete systems into a single chip made of many processing engines such as processors, graphic processing units, video decoders, audio decoders, and display controllers. These systems are called: Multi Processor Systems on Chips (MPSoCs). They are almost found in all electronic devices, especially in consumer electronics such as digital cameras, games consoles, mobile phones and tablet computers.

The increasing integration density in these systems leads to an increasing number of processing engines into a single chip, which in turn requires more efficient on-chip communication systems to inter-connect these processing engines. The history of interconnection systems began with various bus-based systems. From single shared bus to complex hierarchical buses, these complex buses have rapidly shown a strong drawback not only of lack of bandwidth but also of poor scalability with the MPSoCs size. The huge growth in the number of processing engines in MPSoCs coupled with increasing requirements of high bandwidth and low latency have led to a new scalable interconnection structure: Network-on-Chip. By providing scalable performance and higher degree of communication parallelism, NoCs have emerged as suitable interconnect structures for MPSoCs communication requirements.

The development of technology processes has also led to more efficient memory devices with higher bandwidth and storage capacity. Double-Data Rate Synchronous Dynamic Random Access Memories (DDR SDRAMs) were introduced as a cost-effective path for upgrading data bandwidth to memory, and have quickly become the memory of choice in consumer electronics market. DDR SDRAMs have seen a drastic drop in price since 2001, bringing them to price parity with conventional SDRAMs. For technology reasons related to the processes of production, DDR SDRAMs are off-chip. They are often clustered in a memory subsystem made up of *off-chip* memory devices connected to an *on-chip* memory controller. In order to make cost-efficient systems, the designers always try to minimize the number of external pins in a given system. This is why the memory subsystem is often unique and always shared between processing engines, which access it through the network-on-chip.

The International Technology Roadmap for Semiconductors (ITRS) predicts that the number of processing engines in the system-on-chip consumer portable designs is going to reach 1000 processing engines in 2019. This emphasizes the increasing pressure on the shared

memory system. Even if several memory systems do exist in the same chip, the ratio between the number of processing engines and the number of memory systems is at least 10. Besides, the traffic patterns that access the shared memory can no longer be deterministic because they tightly depend on the applications run by the user. Therefore, the variety of processing engines in modern MPSoCs leads to a mixture of traffic classes in the memory controller. This mixture of *dynamic* traffic and high pressure make the task of the memory controller more complex.

1 Thesis scope

This thesis deals with the shared SDRAM access in NoC-based MPSoCs. Many researchers have focused either on network-on-chip services or on memory controller architectures. However, very few studies consider the access to the shared memory with a *system approach*: from the initiators to the memory system through the routers of the network.

High performance access to the SDRAM is firstly related to the memory subsystem itself, which is made up of SDRAMs modules and a memory controller, and secondly to an optimized sharing of the resource for the different traffic that targets it. As the shared external SDRAM is often unique in a given system, the overall system performance is tightly correlated with the memory subsystem performance.

Exploring the memory controller architecture in a given system helps the designer to find the most appropriate architecture that meets the system requirements in terms of bandwidth and latency. Therefore, we introduce in this work our *customizable architecture* of memory controller, and provide the necessary tools to create and explore memory controller architectures.

Furthermore, we highlight the negative impact on the overall system performance of the service discontinuity between the network-on-chip and the memory subsystem, and we emphasize the importance of the service extension from the NoC to the memory controller. We introduce then a way to couple the services of both network-on-chip and shared memory subsystem through our *extreme end-to-end protocol.*

2 Thesis organization

The rest of the manuscript is organized as follows:

Chapter 1 "Problem Definition" provides an overview of the DDR SDRAM access through networks-on-chip. It shows the memory controller task complexity, and how this complexity puts a lot of constraints on the design and makes the architecture exploration of the memory controller very difficult. It also shows the importance of the services coupling between the network and the memory system, and emphasizes the continuity of the guaranteed service,

which can only be ensured by the joint use of architectural and protocol mechanisms.

Chapter 2 "State of The Art" presents the state-of-the-art design of networks-on-chip and memory controller that have a relationship with the off-chip main SDRAM. A particular attention is given to the reconfigurability of the memory subsystem design, to the services provided by the network, and to traffic classes that target the main memory subsystem. At the end, a summary and synthesis of existing reviewed memory controllers and networks-on-chip is provided.

Chapter 3 "Dynamic Memory Controller Customizable Architecture" presents our customizable memory controller design. It begins by providing a detailed description of the DDR3 SDRAM operations. It describes then our high-level building components library that can model any known architecture of memory controller. These building components are cycle approximate, and can precisely simulate all access delays to the shared memory system.

Chapter 4 "Extreme End to End Flow Control Protocol for SDRAM Access" presents our novel end-to-end protocol for shared memory access through a multi-port memory controller. It describes the novelty of this protocol, and the importance of the memory controller state sharing with the network-on-chip.

Chapter 5 "Implementation of the Customizable Architecture of Memory Controller" gives an overview of the development environment (OMNeT++), and describes our method of implementation.

Chapter 6 "Experiments and Results" presents performance analysis of a memory controller architecture modelled with our building components library. It also shows the evaluation of our extreme end-to-end protocol with three different NoC topologies: Spidergon, 2D-mesh, and irregular. A complete analysis of the shared memory access with our protocol is given at the end of this chapter.

Chapter 7 "Conclusion and Perspectives" will wrap up the manuscript by summarizing the major contributions of the thesis and proposing interesting research directions as future work.

CHAPTER 1

Problem Definition

Contents

1.1 DDRn SDRAM concepts 3
1.2 Memory controller concepts 5
1.3 Quality of service in networks-on-chip 7
1.4 Continuity of services in NoC-based systems 8
1.5 Experiments 8
1.5.1 Simulation environment 9
1.5.2 Platform configuration 9
1.5.3 Simulation & results 11
1.5.4 Experiments summary 15
1.6 Conclusion 15

DDR SDRAMs stands for Double Date Rate Synchrounous Dynamic Random Access Memory. These memory were introduced as a cost-effective path for upgrading data bandwidth to memory and have quickly become the memory of choice in consumer electronics markets. DDR SDRAMs have seen a drastic drop in price since 2001, bringing them to price parity with conventional SDRAMs. For technology reasons related to the processes of production, DDR SDRAMs are off-chip. They are always shared between IP components, and accessed through an interconnect structure such as a bus or a network-on-chip (NoC).

Revolutionary changes in memory speed, efficiency, size and costs were required in the early 2000's to support the CPUs enhancements. However, these enhancements were not sufficient to fill the frequency gap between the CPU and the memory. The classical CPU-DDR SDRAM case shows that the frequency gap between CPU and main memory eventually offsets most performance gains from further improvements on the CPU speed. For instance, a cache miss is equivalent to hundreds cycles for today's CPUs, a time long enough for the processor to execute hundreds of instructions. While the DDR SDRAM IO frequency has been improving by 37% per year since 2001, the CAS[1] Latency of SDRAM that fundamentally determines its overall performance has been only improving by 5% per year [75; 76; 77]. Hennessy and Patterson showed that microprocessor performance has been improving by 55% per year since 1987, which emphasizes the growing gap between CPUs speed and SDRAM access time [35].

The requirements of MPSoCs[2] for high bandwidth and low latency makes the DDR SDRAM access become a bottleneck. The multi-threading technique used nowadays in multimedia SoCs[3] with heterogeneous cores increases the contention on the main memory system and demands memory systems with more complex architecture and higher performance.

1.1 DDRn SDRAM concepts

DDR1, DDR2 and DDR3 are the three generations of DDR SDRAM which exist on the market. DDRn SDRAM uses a double-data-rate architecture to achieve high-speed operation. The double data rate architecture is essentially a $2^n.w$ prefetch architecture with an interface designed to transfer two data words per clock cycle at the I/O pins where w represents the memory data bus width. A single read or write access for the DDRn SDRAM effectively consists of a single $2^n.w$-bit wide, one-clock-cycle data transfer at the internal DRAM core and 2^n corresponding w-bit wide, one-half-clock-cycle data transfers at the I/O pins.

DDRn SDRAMs are three-dimensional memories with the dimensions of bank, row, and column. Figure 1.1 shows a simplified architecture of modern DDRn SDRAM. Each bank is organized as a two-dimensional array of SDRAM cells, consisting of multiple rows and columns.

[1]Column Access Strobe
[2]Multi Processor System-on-Chip
[3]System-on-Chip

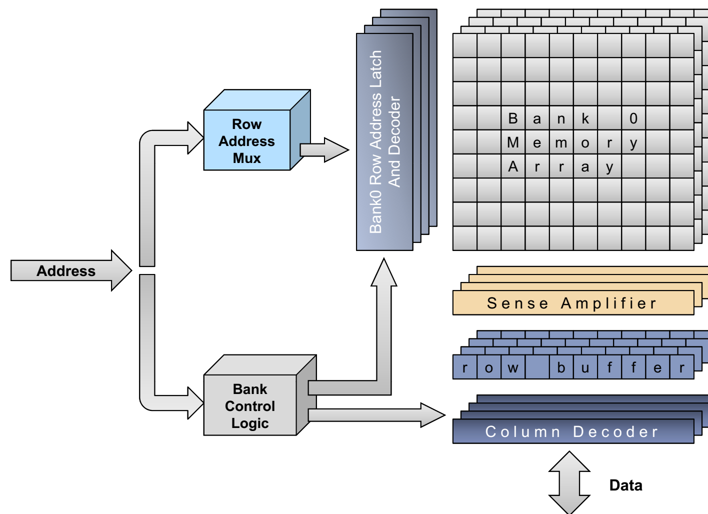


Figure 1.1: Simplified architecture of a modern DDR SDRAM

It operates independently of the other banks and contains an array of memory cells that are accessed an entire row at a time. When a row of this memory array is accessed (*row activation*) the entire row of the memory array is transferred into the bank's row buffer. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads or writes (*column accesses*) may be performed.

When the column access is completed, the cache row must be written back to the memory array by an explicit operation *bank precharge*. This operation prepares the bank for the next *row activation* command. Read and write commands can be issued with an *auto precharge* flag resulting in an automatic *precharge* at the earliest possible moment after the transfer is completed. In order to retain data, all row in the memory array must be refreshed periodically, which is done by precharching all banks and issuing a *refresh* command. The *refresh* operation takes t_{RFC} cycles and must be repeated every t_{REF} cycles. Table 1.1 shows some DDRn SDRAM nomenclatures.

A memory request falls into two different categories:

1. **Row hit**: The request is accessing the row currently in the row buffer. Only a read or a write command is needed. This case results in the lowest bank access latency t_{CL}.
2. **Row miss**: This category can be divided into two subcategories:
   - Row closed: There is no row in the row buffer. An activate command needs to be issued to open the row followed by a read or write command. The bank latency of this case is $t_{RCD} + t_{CL}$ as both a row access and a column access are required.
   - Row conflict: The access is to a row different from the one currently in the row buffer. The contents of the row buffer first need to be written back into the memory

Table 1.1: DDRn SDRAM timing parameters description

Parameter Name	Description
tCL	Column access strobe Latency
tRCD	Row to Column delay
tRP	Row Precharge delay
tWR	Write Recovery delay
tWTR	Write To Read delay
tREF	REFresh interval
tRFC	ReFresh Cycle delay

array using the *precharge* command. The required row then needs to be opened and accessed using the *activate* and *read/write* commands. This results in the highest bank access latency $t_{RP} + t_{RCD} + t_{CL}$

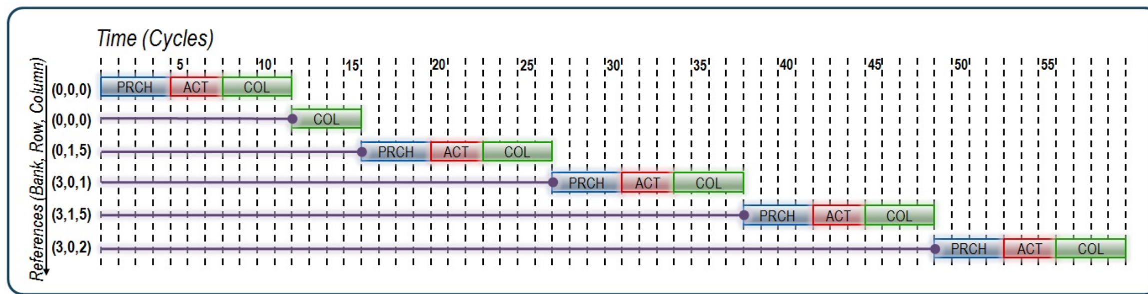
Additional delays have to be considered when the last column access is a write operation. t_{WR} defines the Write Recovery time, which is the minimum time interval between the end of a write operation and the start of a *precharge* command. t_{WTR} defines the Write To Read turnaround time that represents the minimum time interval between the end of a write operation and the start of a read operation.

To see the advantage of memory access scheduling, consider the sequence of six memory operations shown in Figure 1.2a. Each reference is represented by the triple (bank, row, column). Suppose we have a memory system utilizing a DDR SDRAM that requires 4 cycles to precharge, 3 cycles to access a row of a bank, and 4 cycles to achieve a read/write operation in a column. Once a row has been accessed, a new column access can issue each cycle until the bank is precharged. If these six references are performed in order, each requires a precharge and a row access (if the row is not ready), and then a column access. These six references require 59 clock cycles to be performed in order. If we assume that the system data consistency will remain guaranteed when we perform the references in a different order, a total of only 34 clock cycles will be needed. Figure 1.2b shows the *out-of-order* scheduling of the six references.

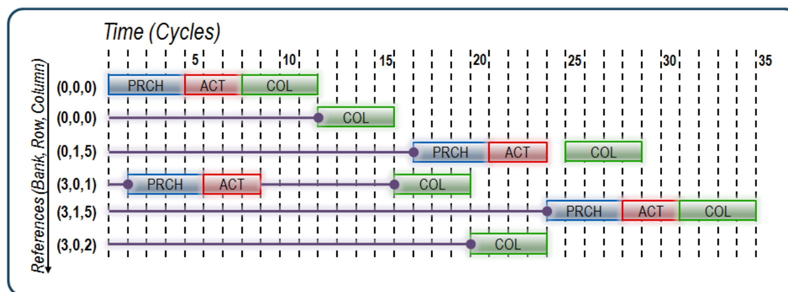
Consequently, the order in which DDR SDRAM accesses are scheduled has a dramatic impact on memory bandwidth and latency. Therefore, typical scheduling algorithms try to increase the row hit ratio to optimize the memory system efficiency. This work is done by the *memory controller*, which is the topic of the next subsection.

1.2 Memory controller concepts

The memory controller plays a principal role in the optimization process of the memory access. It is the interface between the system and the memory modules. The common tasks of a memory controller are memory mapping, request scheduling, command generation and memory management. These tasks are mapped to the memory controller architecture. A typical architecture of a memory controller is showed in Figure 1.3. It is divided into two logical



(a) Without access scheduling (59 clock cycles)



(b) With access scheduling (34 clock cycles)

Figure 1.2: Time to complete a series of memory references without (a) and with (b) access reordering.

blocks called *front-end* and *back-end* [2].

- The *front-end* includes the memory mapping and the arbiter. The memory mapping does the translation from the *logical address* space used by the requestors to the *physical address* space (bank, row, column) used by the memory. The arbiter role is to decide what request will next access the memory. The choice can depend on one or more criteria, e.g. the age of the request, the average bandwidth consumed by the requestor, the priority of the requestor, the request direction (read or write), etc...

- The *back-end* includes the commands generator and the memory manager. After the *front-end* arbiter has chosen the request to serve, the actual memory commands have to be generated and sent to the memory. The commands generator is memory-technology-dependent, and designed to target a specific SDRAM. It is programmed with the timings for a particular memory device, and needs to keep track of the state of each memory bank and ensure that no timings are violated. The memory manager guarantees the proper behaviour of the memory and carries out several tasks such as initialization, refreshing and powering down.

The quality of service (QoS) in a memory controller refers to satisfying the initiators requirements in terms of bandwidth and latency while optimizing the memory bus efficiency and guaranteeing the data consistency. Moreover, the memory controller needs to obey all

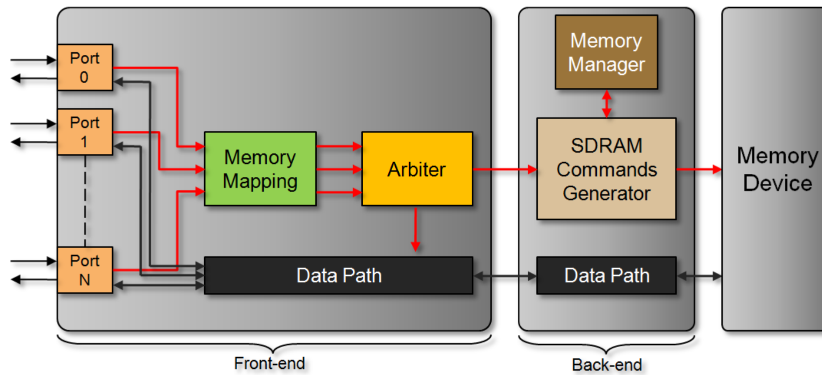


Figure 1.3: Simplified architecture of a memory controller

SDRAM timing constraints to provide correct functionality. This makes the memory controller task complicated and induces a lot of constraints on its design.

1.3 Quality of service in networks-on-chip

The huge growth in the number of embedded components and their need for higher bandwidth and lower latency have led to a new scalable interconnect structure known as Network-on-Chip (NoC). By providing scalable interconnect system and higher degree of parallelism in comparison with previous on-chip communication systems, NoCs have emerged as suitable interconnection solution for modern and future on-chip systems [27].

The quality of service (QoS) in a NoC refers to a resource reservation mechanism guaranteeing that special packets do not share the resources with other packets. These special packets are called *guaranteed service* (GS) packets while the other packets are called *best effort* packets. Best effort is the basic service of a NoC and does not provide any kind of QoS. Therefore, the latency of the packets cannot not be bounded and the throughput cannot be guaranteed. The guaranteed service in a NoC fixes a minimum throughput threshold, and a maximum latency and jitter threshold. In term of QoS, we can define two major types of guarantees [64]:

- The *hard* QoS guarantees the maximum predictability of the network. It bounds and constrains the latency, the throughput and the jitter. This kind of guarantees may be achieved by reserving exclusive accesses to the resources for the guaranteed traffic.
- The *soft* QoS, which is less strict, guarantees the same metrics as the *hard* but has some degree of unpredictability. This kind of guarantees may be achieved by mixing some exclusive and non-exclusive accesses to the shared resources.

Although the network-on-chip provides QoS to several classes of traffic, it cannot ensure the *continuity* of the QoS when the traffic is addressed to an off-chip memory device because

it crosses the boundary of the network. Neither the network nor the initiators are aware of the memory controller status. We mean by the memory controller status the current activated rows in the SDRAM; the free slots in the buffers; the pressure on each memory bank and the row miss rate after the accesses re-ordering. This lack of information in the network emphasizes the discontinuity of the QoS between the initiators and the memory device.

1.4 Continuity of services in NoC-based systems

Whatever the type of the QoS that the network-on-chip provides, either *hard* QoS or *soft* QoS, its continuity from the initiator to the target is essential to satisfy the requestors needs. A large number of paths has been taken by researchers to reduce the system overhead. These paths have been divided into two main approaches. The first one focuses on the memory devices and their scheduler, whereas the second one takes into consideration the interconnect architecture. Recent studies show that memory-oriented approaches can reduce application time execution [17; 58]. However, focusing on memory access alone is not enough. Even with zero latency SDRAM access, the overhead of primary memory system would not be eliminated, because transactions through a shared on-chip communication system such as a NoC still require time.

The interconnect latency between a master and the memory subsystem becomes trickier for latency-sensitive masters, e.g. a *cache controller*. Moreover, most of memory controllers store requests before sending them to the SDRAM, which increases the transactions latency. It makes sense to optimize the combination of external-memory controller and interconnect, and shows the importance of a *system approach* to minimize the overall latency when using a complex interconnection system such as a network-on-chip.

1.5 Experiments

Realizing the importance of a *system approach* to optimize external memory access in MPSoCs, we study in this section the interaction between the memory system and the network-on-chip in an MPSoC platform. The platform in use is a part of an internal STMicroelectronics design that includes a Spidergon STNoC and two commercial and well-spread memory controllers. The platform in use is at RTL level and implemented in VHDL. The NoC in this design represents the interconnect backbone that connects several clusters to the memory subsystems. We focus on this platform because it represents a real case study and offers QoS in both network-on-chip and memory system.

We aim, by running these simulations, at optimizing the *latency-sensitive* traffic sent by the *cache controller* when it accesses the memory subsystems. Note that the *latency-sensitive* traffic coexists with other classes of traffic sent by a *DMA* and a *streaming IP*.

Although we tried to optimize the off-chip memory access by programming the QoS in

the system and by adding dedicated hardware in the network, the QoS *continuity* is still not guaranteed when the traffic crosses the boundary of the network to enter into the memory controller. This is what we are going to show in the following subsections.

1.5.1 Simulation environment

We use in this experiment the Spidergon STNoC [16] as an on-chip interconnect structure. This interconnect is the backbone part of an STMicroelectronics design. Its role is to connect several clusters with the memory subsystems. The necessary details about the simulation platform are given in Annex A.

The QoS in Spidergon STNoC indicates the ways to manage bandwidth and latency to ensure a minimal requirement for each traffic flow. Arbitration is a critical part of the router, since it determines the level of QoS support of the network and impacts router performance in terms of critical path delay. As far as bandwidth is concerned, Spidergon STNoC supports the Fair Bandwidth Allocator (FBA) QoS mechanism. It is an end-to-end service that guarantees fair and programmable weighted bandwidth allocation on the top of a distributed network [16].

The Network Plug Switch and the Router can implement two virtual channels through one physical link with the necessary logical blocks for arbitration within a given channel, and between two channels (see Annex A.1 for more information about Spidergon STNoC building blocks). The main advantage of the virtual channels (VCs) technique is a low wire area overhead per additional virtual channel compared to the duplication of the physical link. This stems from the fact that the traffic classes are multiplexed over the same long wires.

Economically viable SDRAMs are driven by single port memory controllers. Thus, we make use of an industrial memory controller existing in the current the state of the art. The memory controller offers QoS in term of latency for read transactions. Entries are arbitrated with an algorithm that optimizes the efficiency of the memory data bus. To achieve optimum memory bus efficiency entries might be arbitrated out of order from their arrival time. The way to ensure QoS is by using priorities for read accesses that require low latency read data. The QoS for read access is determined when the arbiter receives it, and it is based on the requestor ID. No QoS exists for write accesses.

We aim at providing the same level of QoS to high-priority transactions such as *cache controller* transactions in the network and the memory controller. We separate high-priority transactions from other transactions by mapping them to a dedicated *VC*. In addition, we give this *VC* the highest priority in NIs and routers through the network. Moreover, we program the memory controller so as to minimize the stall time of transactions having high priority.

1.5.2 Platform configuration

As several configurations of each component are possible, we experimented what we feel the most relevant ones in order to be able to evaluate the performance of the NoC and measure

the DDR1 SDRAM access latency when implementing virtual channels.

Traffic generators

Each traffic generator has an AMBA AXI interface with two separate channels for read and write requests. The number of outstanding requests[4] for each channel is configurable. The traffic generator has the capability of generating constrained-random traffic in accordance with a statistical distribution which determines the inter-transaction time. We have a full control over AMBA AXI bus parameters such as address range; transaction ID and burst length. A preview of traffic generators characteristics is given in Table 1.2.

Table 1.2: Traffic generator characteristics

IP Name	Data Rate	Latency	Jitter	Burst Length[a]	Issuing Capabilities
Cache Ctrl port0	low	low	low	32[b]	2 reads, 2 writes
Cache Ctrl port1	low	low	low	32	2 reads, 2 writes
DMA	high	tolerant	tolerant	16 $\rightarrow$ 128[a,c]	2 reads, 2 writes
Streaming IP	high	tolerant	high	16 $\rightarrow$ 128	2 reads, 2 writes

[a] In Bytes
[b] Corresponds to the cache line width
[c] Allowed burst sizes are : 16, 32, 64, 96, 128 bytes

The traffic balancing of all these generators is defined as 50% towards on-chip SRAMs, and 50% towards off-chip DDR SDRAMs.

Memory subsystems

Two identical memory subsystems are connected to the NoC. Each one is made up of the combination of a single port dynamic memory controller and two x16 DDR SDRAMs. The memory controller is programmed after the reset signal. During this period we configure it in specifying the maximal admissible latency value for each initiator (identified by its unique source ID). Thus the memory controller will be able to schedule the requests according to these maximum latency values.

The QoS in the memory controller indicates the ways to manage bandwidth and latency. The latency guarantee of a flow is based on the flow ID, while the minimal bandwidth of a flow is based on the flow ID and the memory bank status (row hit/miss). Within these experiments, the *Cache Controller* read transactions latency must be less than 40 clock cycles.

[4]The number of outstanding requests is the maximum number of requests the traffic generator can send before receiving any response

Interconnect

We use in this simulation platform the Spidergon STNoC interconnect technology. We implement two separated NoCs, one for requests and one for responses. In order to minimize the number of buffers and thus the interconnect area, we only implement two channels on the path between *cache controller* ports and *memory subsystems* (see Figure A.1). The *channel splitters* differentiate among *cache controller* transactions targets and map them to two channels. The *channel splitter* can be activated or not:

- When enabled, it maps *cache controller* transactions towards off-chip memory subsystems to channel 2 (*ch2*) that provides the highest QoS in the NoC, and all other transactions to channel 1 (*ch1*).
- When disabled, it maps all *cache controller* transactions to one channel (*ch1*) with the highest priority. *DMA* and *streaming IP* transactions are also mapped to *ch1* but with low priority.

Therefore, we are able to make a fair comparison of the external memory access latencies when we use a separated channel for *cache controller* transactions only.

As we use a single port memory controller, we need a *channel merger* block to merge both channels in one AMBA AXI bus when the *channel splitter* is enabled. Note that the routing algorithm in the request network and the response network are symmetrical.

Service coupling of both Spidergon STNoC and memory controller

For the first configuration (only 1 channel), we prioritize all *cache controller* transactions towards the memory subsystems by giving them the highest priority in the router arbiters, and by choosing an arbitration algorithm based on packets priority.

For the second configuration (2 channels), we prioritize the *cache controller* transactions towards the memory subsystems by mapping them to a dedicated channel *ch2*, and by configuring the router arbiters so as to prioritize *ch2* over *ch1* without packets locking on *ch1*. In this way, the high priority requests/responses on *ch2* do not stall behind the other requests/responses on *ch1* on the same physical channel between routers (see Figure A.1).

For both platform configurations (one channel or two channels), we give the highest priority to all *cache controller* transactions.

1.5.3 Simulation & results

To evaluate the capabilities offered by the current memory controller, we measure the latency of the *cache controller* accesses to the external DDR SDRAMs. We change the mapping of its transactions through the interconnect and we measure the latency variation by means of transactions spies. We do here an exploration job in order to evaluate the influence of the burst length of *DMA* and *streaming IP* over *cache controller* transactions. The *cache controller* is sending read&write requests to all slaves; *DMA* and *streaming IP* are also sending

read&write requests to all slaves. We select one range among four burst length ranges for each experiment. The burst length ranges are: 16 to 32; 32 to 64; 64 to 96; and 96 to 128 bytes (see section (1.5.2) for more information about the traffic generation).

For each burst range, we compute the memory access speedup when we use one channel with QoS[5] activated in the memory controller compared to the basic case when the QoS in the memory controller is deactivated(see equation (1.1)). We perform a similar calculation for the memory access speedup when we use two channels with activated QoS in the memory controllers compared to the basic case when no QoS is provided in the memory controllers (see equation (1.2)). When the QoS is deactivated in the memory controller, it becomes unable to cope with the requestors requirements, because it omits the request source ID. However, it continues to provide QoS in term of memory bus efficiency, and tries to increase the row hit rate.

$$Speedup_{(1ch)} = 100 \cdot \frac{Lat_{1ch}^{DMC_QoS_on} - Lat_{1ch}^{DMC_QoS_{off}}}{Lat_{1ch}^{DMC_QoS_{off}}} \tag{1.1}$$

$$Speedup_{(2chs)} = 100 \cdot \frac{Lat_{2chs}^{DMC_QoS_{on}} - Lat_{1ch}^{DMC_QoS_{off}}}{Lat_{1ch}^{DMC_QoS_{off}}} \tag{1.2}$$

Two speedup values are computed, one based on average latency and the other one based on maximum latency. We monitor the round trip latency of *cache controller* reads and writes in four different cases:

Cache controller *read* requests versus DMA and streaming IP *read* requests

Both low priority IPs *DMA* and *streaming IP* are sending read requests to external DDR SDRAMs. Figure 1.4 shows the off-chip memory access speedup for this scenario. We can note that the implementation of two channels is useful when the burst length of read requests increases. The use of both channels guarantees a speedup of 37% for off-chip memory access based on maximum latency, when the read request burst length crosses 64 bytes.

Cache controller *read* requests versus DMA and streaming IP *write* requests

In this case, the low-priority IPs *DMA* and *streaming IP* are issuing write requests. Figure 1.5 shows the off-chip memory access speedup for *cache controller* read transactions. The speedup obtained in this case when we use two channels is more important than the previous case. Indeed, low-priority write transactions create contention on the physical links connecting the routers *req0* and *req2* to memory subsystems. By using a second channel

[5]The definition of the memory controller QoS is given in section (1.5.2).

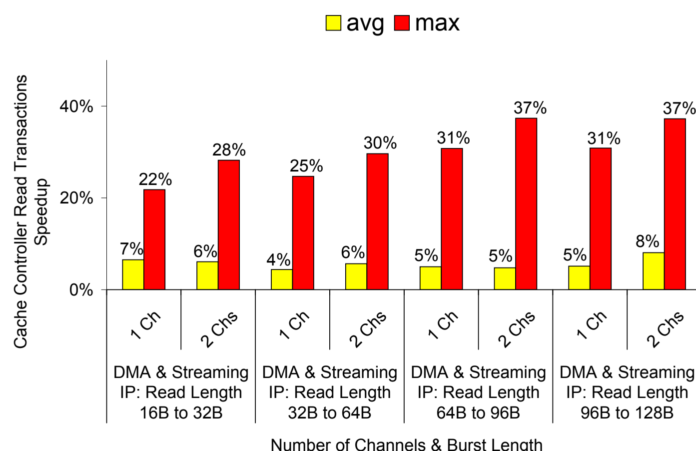


Figure 1.4: Off-chip memory access speedup for cache controller read requests when low-priority IPs send read requests

for high-priority requests, we can accelerate the memory access by bypassing the long write requests.

The results of both previous cases were expected because the services provided by Spidergon STNoC and memory controllers are coupled. The following two cases show the impact of low-priority transactions over the latency of *cache controller* transactions when the QoS is not extended to the memory controller.

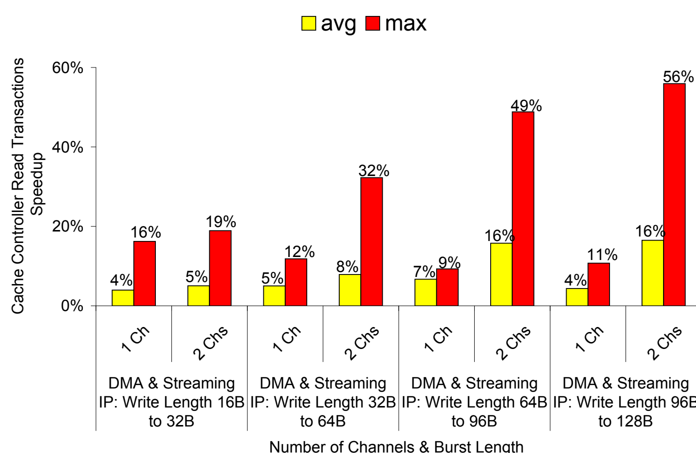


Figure 1.5: Off-chip memory access speedup of cache controller read requests when low-priority IPs send write requests

Cache controller *write* requests versus DMA and streaming IP *write* requests

As we mentioned previously, the dynamic memory controller we are using does not provide any QoS in term of latency for write requests (the memory controller QoS in term of memory bus efficiency is still enabled). Even if we guarantee a maximum latency threshold for

write transactions through the request and the response network, they may be stalled in the memory subsystem. Figure 1.6 shows the memory access speedup/slowdown. The memory access speedup which is based on the latency average value is still positive. However, we can not guarantee a maximum latency threshold for the high-priority write transactions.

As the arbiter tries to increase the memory bus efficiency, it prioritizes the requests that increase the row hit rate in DDR SDRAMs banks. The address locality of the *streaming IP* is very high because it accesses adjacent rows. For this reason the memory controller arbiter schedules consecutive write requests coming from the *streaming IP*, and accessing the same row. In consequence, some *cache controller* write requests must wait until the sequence of requests crosses the row boundary, and can be scheduled at the next *Activate Row* command.

If we focus on the last burst length range (96→128 bytes) in Figure 1.6, we see that there is no slowdown when we use 2 channels. This is explained by the fact that the memory controller cannot schedule a write request while it does not receive the entire packet to write (between 96 and 128 bytes). As the *cache controller* ports send 4-byte-write requests (see table 1.2), their packets can bypass the long packets, and the memory controller entirely receives them before the low priority packets. So the number of high priority requests which are ready to be scheduled will be grater in comparison with the other cases (low-priority burst length 16→96 bytes).

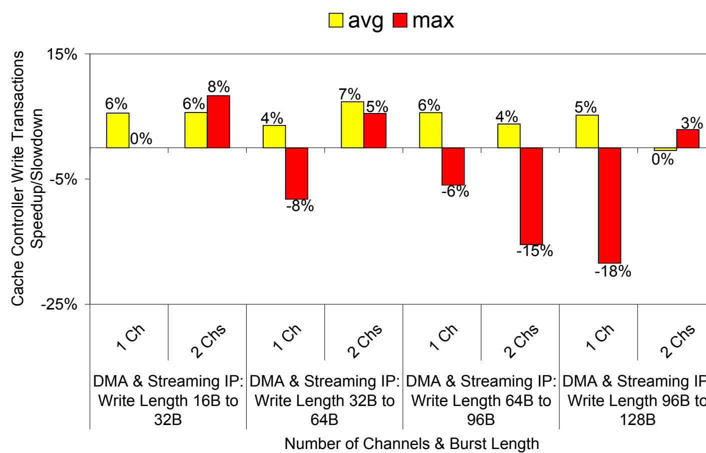


Figure 1.6: Off-chip memory access speedup/slowdown of cache controller write requests when low-priority IPs send write requests

Cache controller *write* requests versus DMA and streaming IP *read* requests

This simulation scenario is a good example of the need of an extended QoS of service throughout the request and response path between the masters and the DDR SDRAM devices. Although the Spidergon STNoC ensures a maximum latency threshold for the *cache controller* write requests, the low-priority read requests from *DMA* and *streaming IP* gain the upper hand in the memory controller. Figure 1.7 shows how the *cache controller* write requests

are slowed down when low-priority IPs issue read requests towards the external DDRs. The longer the burst length of low-priority read requests, the higher the stall time of high-priority write requests.

In this case, the memory controller arbiter gives the priority to the *DMA* and *streaming IP* read requests whenever it receives these read commands. So the 4-byte write requests of the *cache controller* ports must wait for several arbitration cycles to be scheduled.

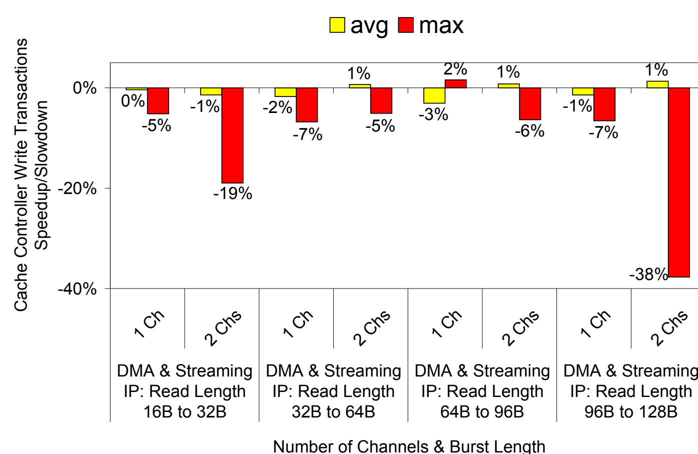


Figure 1.7: Off-chip memory access speedup/slowdown of cache controller write requests when low-priority IPs send read requests

1.5.4 Experiments summary

From the previous experiments it is clear that the services provided by the network for the *cache controller* traffic must be extended to the memory controller in order to ensure the same QoS throughout the path between the *cache controller* ports and the memory devices. Even if we map the *cache controller* traffic to a dedicated channel and we give this channel the highest priority inside the network, we cannot be sure that the same level of QoS will be guaranteed inside the memory controller. The impact of the QoS *discontinuity* is obvious when the *cache controller* sends write requests to the memory subsystems while *DMA* and *streaming IP* send read requests to the memory subsystems. Low-priority reads requests are often scheduled before high-priority write requests.

The question arises here: what are the main modifications to apply to the memory controller so that it can provide the appropriate level of quality-of-service?

1.6 Conclusion

In this chapter we provided an overview of the DDR SDRAM access through networks-on-chip. We have also shown the importance of the service coupling between the network and the memory system.

The task of a memory controller is complex because it has not only to obey all SDRAM timing constraints to provide correct functionality, but also to satisfy the initiators requirements in terms of bandwidth and latency. This puts a lot of constraints on the design and makes the exploration of the memory controller architecture very difficult. From a system perspective, the impact of the memory controller architecture on the memory subsystem performance, and consequently on the system performance, is very important. The continuity of the guaranteed service can only be ensured by the joint use of architectural and protocol mechanisms. However, these mechanisms remain to be defined in the VLSI context within its constraints in terms of area and power consumption.

Being able to explore the architecture of the memory controller and its arbitration algorithms is essential to find an optimized architecture with the appropriate arbitration algorithms. This emphasizes the importance of having a memory controller with a *totally customizable architecture*. Such a customizable architecture allows us to study the interaction between the memory system and the network-on-chip, and to measure the impact of the memory controller on the overall system performance.

Another path that is worth considering is the use of some pieces of information related to memory subsystem status in the network-on-chip. However, we should know what information the memory controller has to share with the NoC in order to enhance the network performance within the process of SDRAM requests scheduling.

Designing a memory controller with a totally customizable architecture for NoC-based MPSoCs, and implementing features for DDR SDRAM access in the network-on-chip will be addressed in this thesis with the aim of easing the architecture exploration of a dynamic memory controller and optimizing the access to shared memories in NoC-based MPSoCs.

CHAPTER 2

State of the Art

Contents

2.1 **Memory controllers** 19

2.2 **On-chip interconnects** 26

2.3 **Combined interconnect-memory controller solutions** 30

2.4 **Conclusion** 33

RECOGNIZING the importance of high performance off-chip SDRAM communication as a key to a successful system design, several memory controllers and on-chip interconnection systems have been proposed. In this chapter, the state of the art on the networks-on-chip and the memory controllers is analysed. The range of the analysis is limited to the topics that provide *guaranteed services*. We separate the related work in three categories: memory controllers, on-chip interconnects, and combined interconnect-memory controller solutions. Principles of SDRAMs and memory controllers are given in Chapter 1,sections (1.1) and (1.2) respectively.

2.1 Memory controllers

Existing SDRAM controller designs are either statically or dynamically scheduled, depending on wich kind of systems they target. Statically scheduled memory controllers combine static front-end arbitration with static scheduling of SDRAM commands in the back-end. The pre-computed schedule in the back-end makes the design unable to adapt to changes in the behaviour of the requestors. As the static arbitration couples latency and allocated bandwidth, it is not able to satisfy the requirements of latency-latency requestors with low bandwidth requirements without wasting bandwidth. Conversely, dynamically scheduled memory controllers combines dynamic front-end arbitration with dynamic back-end scheduling. These controllers target high-efficiency and flexibility to fit in high-performance systems with dynamic applications whose behaviours may not be known up front [2]. We are addressing in this section the dynamically scheduled memory controllers.

In order to improve the memory efficiency, a number of dynamic memory controllers use information about memory state when scheduling. This consideration is typically done in the back-end. However, some designs communicate memory state to the front-end arbiter, which blurs the distinction between the two. Among the information about the memory state that he front-end use we may mention the open row in each bank and the memory bus direction (read or write).

Rixner *et al.* [67] present a controller that privileges the requests which target an open row in a bank. They show that none of the fixed policies studied provide the best performance for all workloads and under all circumstances. However, the FR-FCFS (first-ready first-come-first-served) policy exploits the locality within the 3-D memory structure (bank/row/column) at best, and provides a 17% performance improvement on the whole set of applications. The optimization mechanisms presented in this study became inevitable for any efficient design of memory controller.

In addition to the open row policy some references use front-end priority-based arbitration to keep up with the requirements of *latency-sensitive* requestors. This kind of requestors often correspond to processors that stall while waiting for a cache lines. **Shao and Davis [71]** propose the *burst scheduling access reordering* mechanism which clusters memory accesses to the same rows of the same banks into bursts to maximize bus utilization of the SDRAM

device. Subject to a static threshold, memory reads are allowed to preempt ongoing writes for reduced read latency, while qualified writes are piggybacked at the end of bursts to exploit row locality in writes and prevent write queue saturation. Nevertheless, clustering memory accesses to the same rows of the same banks could not be efficient for all traffic scenarios.

Many dynamic designs use rate regulator in the front-end to protect requestors from each other. This is especially important in controllers with priority-based arbiters, since these are often prone to starvation. **Burchard *et al.* [10]** present a real time streaming memory controller (SMC) that uses a rate regulator in the front-end and considers the memory bus direction within the threads arbitration. The SMC has been designed to allow external SDRAM to be accessed from a PCI Express network. A simplified architecture of the SMC is depicted in Figure 2.1. They propose a fully parametrized credit-based arbitration algorithm. They also propose the extension of the virtual channels (VC) provided by the PCIEx inside the SMC. As they map one stream by VC, the maximum number of parallel streams accessing the SMC is limited by the number of PCIEx VCs (eight VCs), which makes this architecture not scalable.

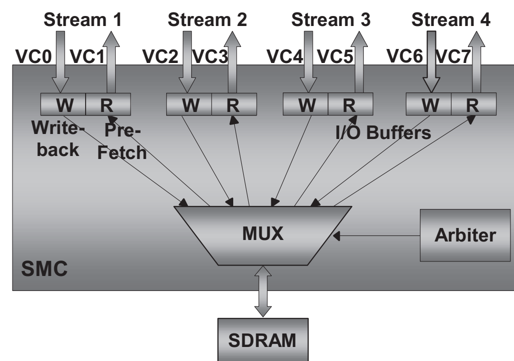


Figure 2.1: The logical view of SMC architecture, source [10]

Macian *et al.* [52] propose an *Embedded Hardware Manager* which is composed of a set of Service Managers, one per application. This design uses a rate regulator in addition to information coming from the back-end such as the open row in each bank and the memory bus direction. Every Service Manager controls access to all shared resources for one application. In case that certain requests had more stringent delay requirements, the service manager could assign priorities to them. This design has been tested for a small number of applications (between 2 and 8). The capacity of the design to safely deal with overload has not been estimated.

Heithecker and Ernst [34] present an SDRAM scheduler that supports several concurrent access sequence types with different requirements including predictable periodic real-time sequences and cache accesses with a minimum latency objective. They manage to significantly increase the system performance by combining the flow control and prioritized scheduling that takes into account the memory state. This schedule has the capability to deal with several classes of traffic such as hard real-time periodic traffic and random traffic.

Mutlu and Moscibroda [57] introduced the parallelism-aware batch scheduler (PAR-BS)

as a high performance and QoS-aware DRAM scheduler. According to their evaluation, PAR-BS significantly improves both fairness and system throughput in systems where DRAM is a shared resource among multiple threads. PAR-BS provides thread-fairness and better prevents short-term and long-term starvation through the use of request batching. Within a batch, it explicitly reduces average thread stall times via a parallelism-aware DRAM scheduling policy that improves intra-thread bank-level parallelism. This design uses a priority scheduling in addition to a rate regulator and respects the row buffer locality in order to maximize the bus efficiency. It is advantage over other complex designs is the reconfigurability and the simplicity of implementation.

Whitty and Ernst [80] present a bandwidth optimized SDRAM controller for a heterogeneous reconfigurable platform (Figure 2.2). This controller has two-stage buffered scheduler: the request scheduler and the bank scheduler. The request scheduler uses a round-robin arbitration policy for standard requests. High priority requests are served before standard requests when priority levels are enabled. The bank scheduler performs a bank interleaving to hide the bank access latency and bundles the requests to minimize stalls by read-write switches. This memory controller provides QoS for several traffic types. Two priority levels for memory access requests have been implemented in the interface via distinct access paths for high and standard priority requests. This makes this design usable in *general purpose* platforms.

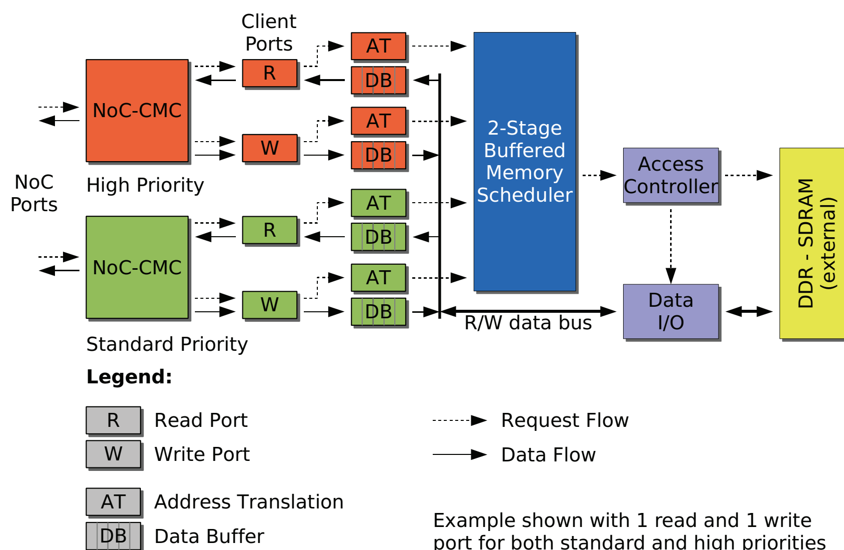


Figure 2.2: Bandwidth optimized SDRAM controller architecture, source [80]

The following designs has a priority-based arbiter with a rate regulator in the the front-end. They also exploit information about the memory state coming from the back-end (open rows and bus direction). **ARM [3]** introduced the single port memory controller PL340 which provides a QoS in term of latency for read requests only. A *max-latency* value is allocated to each thread. When a read request spends more than its *max-latency* time in the arbitration

queue, the scheduler sends it to the SDRAM to be executed in such a way that the memory bus efficiency is still respected. The arbiter considers two types of hazards: *Read After Read* and *Write After Write*. It prevents a read request from being executed while another read request with the same ID is still waiting in the arbiter queue. A similar technique is used for write requests. The negative point in this design is the mono-port interface with the interconnect. In opposition to other design, this memory controller mixes all traffic classes at the input point.

Lee *et al.* [**48**] presented a multi-layer quality aware memory controller that contains partitioned functionality layers to achieve high SDRAM utilization and meets requirements for bandwidth and latency (see Figure 2.3). In the proposed design, channels are put into three categories: *latency-sensitive*, *bandwidth-sensitive*, and *don't care*. Latency-sensitive channels are given the highest priority in the memory controller. They also benefit from two other services, *preemptive* service and *column-access-inhibition* service. The first service is used to issue latency-sensitive accesses as soon as possible by suspending the processed access from a bandwidth-sensitive or don't-care channel. This indicates that preemptive service may reduce the average bandwidth utilization. The second service is used to preserve the data bus for latency-sensitive accesses by inhibiting issuing column-access commands from bandwidth-sensitive and don't-care channels, and therefore eliminates latencies resulted from data bus congestion.

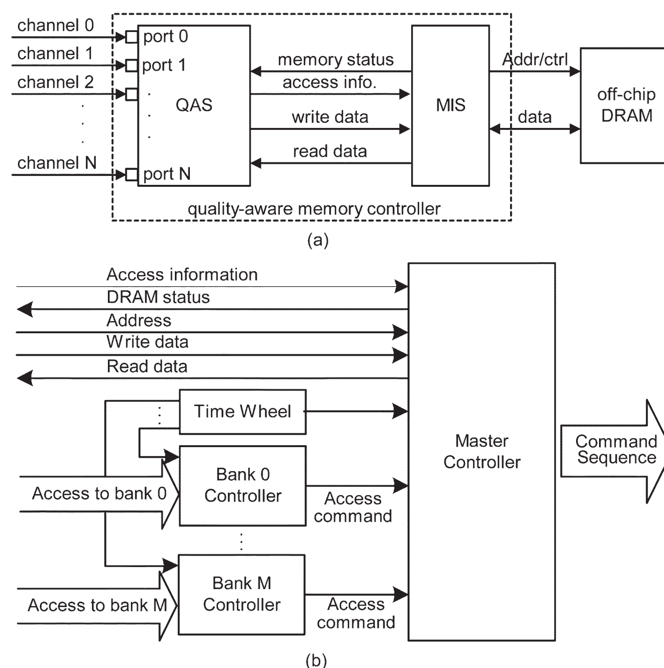


Figure 2.3: (a) Quality aware memory controller (b)MIS architecture, source [**48**]

The interference between threads that access the shared memory has been highlighted by **Zhu and Zhang** [**85**]. They evaluated contemporary multi-channel DDR DRAM and Rambus DRAM systems in simultaneous multi threading systems. Their study proves that increasing

the number of threads tends to increase the memory concurrency and thus the pressure on DRAM system; and that DRAM latency reduction through improving row buffer hit rates becomes less effective due to the increased bank contentions. They also show that thread aware memory access scheduling schemes may improve performance by up to 30% on workloads of memory-intensive applications. **Mutlu and Moscibroda [56]** introduced the concept of stalltime fair memory scheduling (STFM) that provides fair DRAM access to different threads sharing the DRAM system. The key idea that makes STFM work is that equal priority threads, when run together, should experience equal amounts of slowdown as compared to when they are run alone. The goal of the scheduler is to equalize the DRAM-related slowdown experienced by each thread due to the interference from other threads, without hurting overall system performance. They show how STFM can be controlled by system software to control the unfairness in the system and to enforce thread priorities. Figure 2.4 depicts the organization of the STFM memory controller. **Zheng *et al.* [84]** studied memory scheduling schemes for multi-core systems. They prove that scheduling schemes need to consider both the long-term and short-term effects in order to well utilize both the processor cores and memory system. Within their scheduling scheme, requests from threads that have higher memory efficiency and fewer pending requests have higher priority than requests from other threads. In addition, reads and row buffer hits have higher priority than writes and row buffer misses, respectively. Their simulation results show that for memory-intensive workloads the new policy improves the overall performance by 10% on average and up to 17% on a four-core processor, when compared with scheme that serves row buffer hit memory requests first and allowing memory reads bypassing writes. **Nesbit *et al.* [59]** propose another memory scheduler providing QoS to improve system performance. It is based on concepts developed for network fair queuing (FQ) scheduling algorithms and targets high performance Chip Multi Processors (CMPs). The FQ memory scheduler allows arbitrary fractions of memory system bandwidth to be allocated to an individual processor or a cluster of processors. It provides QoS to all of the threads in all of the workloads running on a four CMP and improves system performance by 14% on average. All previous studies deal with multi-core systems with multi-thread memory system. However, non of them considers the memory access process as a system manner. They all focus on the memory controller architecture without tackling the manner that the requests are brought to the memory system through the interconnect structure.

Depending on the traffic that accesses the shared memory, several important features could be added to traditional memory controllers. **Carter *et al.* [12]** show a new memory system architecture (Impulse) that supports application specific optimizations through configurable physical address remapping and does intelligent prefetching at the memory controller which reduces the effective latency to memory. Instead of fetching an entire cache line from the DRAM, the memory controller can be configured by an application to export dense shadow space alias that contains just the elements needed by the application(see Figure 2.5 and 2.6). This mechanism can only be useful in conventional systems, therefore, not in case

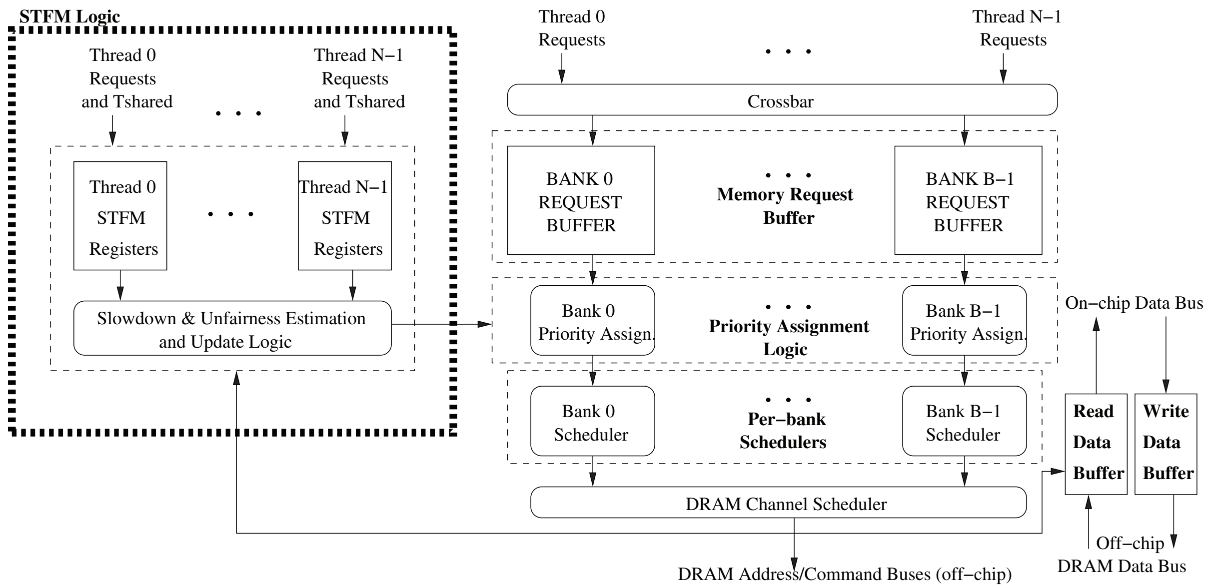


Figure 2.4: Organization of the on-chip STFM memory controller, source [56]

of mixed traffic classes. **Zhu *et al.* [86]** present a workload independent approach by focusing on optimizing fine grain priority scheduling. This mechanism splits a memory reference into sub-blocks with minimal granularity, and maps sub-blocks from a reference into different channels. All channels can be used to process a single cache line fill request. Sub-blocks that contain the desired data are marked as critical ones with higher priorities and are returned earlier than non-critical sub-blocks. **Zhu *et al.* [87]** propose a high performance SDRAM controller optimized for high definition video application. They combine multiple access units into one transaction to enable consecutive data transmission, which suits the characteristics of video decoder accessing. Afterwards, they apply the scheduling strategy that finds and issues the *Activate / Precharge* command as early as possible to allow the latency incurred to be overlapped. The high definition video applications generate traffic with high-locality addresses, this is why combining multiple access unit into one transaction is possible without creating row misses.

Akesson *et al.* [1] proposed a memory controller (Predator) that guarantees minimum bandwidth and maximum latency bounds to the IPs using a novel approach to predictable SDRAM sharing. This is accomplished by defining memory access groups corresponding to precomputed sequences of memory commands with known efficiency and latency. Then, a predictable arbiter is used to dynamically schedule these groups at run-time such that the requirements of bandwidth and latency remain guaranteed for each IPs. Figure 2.7 depicts a simplified architecture of the Predator. Although this architecture is extended to cover a part of the network interface, it does not share any information between the network and the memory controller in order to improve the overall system performance. **Akesson [2]** presented later a memory controller that offers bounds on both net bandwidth and the latency of requestors at design time, which enables configuration settings to be automatically syn-

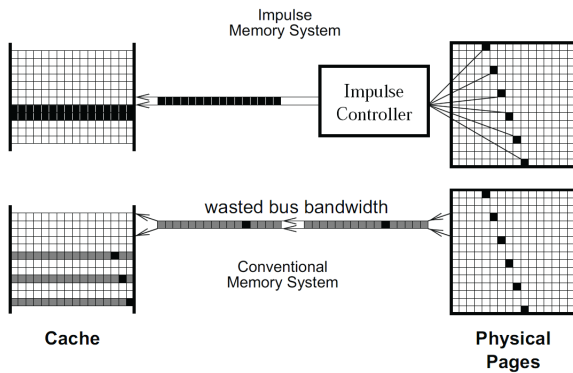


Figure 2.5: Using Impulse to remap the diagonal of a dense matrix into a dense cache line. The black boxes represent data on the diagonal, whereas the gray boxes represent nondiagonal data. Source [12]

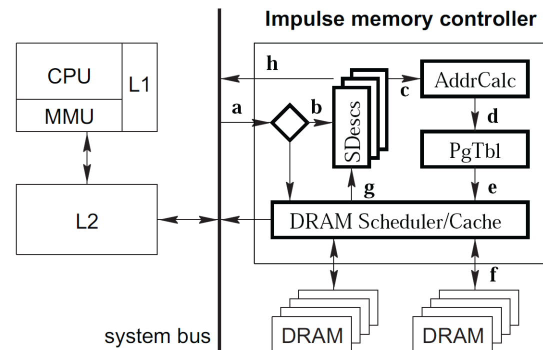


Figure 2.6: The Impulse memory architecture. Source [12]

thesized for a given set of requirements. The front-end uses predictable dynamic arbiters in the class of Latency-Rate (LR) servers, which satisfies diverse latency requirements. The command generator uses a hybrid approach based on memory patterns that is a mix between static and dynamic command scheduling. Memory patterns are precomputed sub-schedules that are dynamically combined at run-time, enabling the controller to accommodate traffic that is not fully known at design time in a predictable fashion. The strength of this study is the use of the predictable SDRAM access patterns notion and the principle of composable[6] systems to bound the bandwidth and latency of the shared memory requestors.

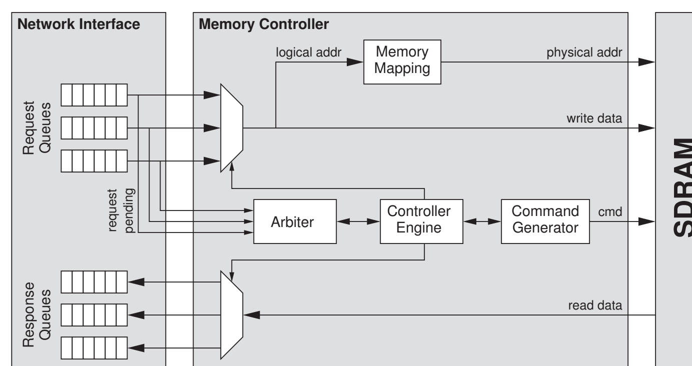


Figure 2.7: Predator memory controller architecture, source [1]

Lee and Chung [47] presented a scalable qos-aware memory controller. Even though this memory controller is designed for IP packets, and could be difficult to implement within the VLSI constraints, its architecture worth to be analyzed. The requests are stored in FIFOs according to their QoS class, direction (read/write) and the memory bank they want to access.

[6] A system is considered composable if applications cannot affect each other's behaviour in the value and time domains [2]

There is one FIFO for each class/direction/bank triplet. Bank arbiters selects requests from each set of direction/bank FIFOs and forward the requests to the class schedulers. There are only two class schedulers, one for each direction (read/write), they are based on the weighted round robin algorithm. Class schedulers forward the requests to the direction arbiter, which is a simple round robin arbiter. This arbiter is the last one in the memory controller front-end.

Table 2.1: Main features of the state of the art memory controllers

Ref.	**Main sched. scheme**		**Mem state exploitation**		**Rate regulator**	**Real time guarantees**	**Prefetching**
	Priority	RR	row	bus direction			
[67]	✓	-	✓	✓	-	-	-
[71]	✓	-	✓	✓	-	-	-
[10]	✓	-	✓	?	✓	✓	-
[52]	-	✓	✓	✓	✓	✓	-
[34]	✓	-	✓	✓	✓	✓	-
[57]	✓	-	✓	?	✓	-	-
[80]	✓	-	✓	✓	✓	✓	-
[3]	✓	-	✓	✓	-	-	-
[48]	✓	-	✓	✓	✓	✓	-
[56]	✓	-	✓	✓	✓	-	-
[59]	✓	-	✓	✓	-	✓	-
[12]	✓	-	✓	?	✓	-	✓
[1]	✓	-	✓	?	✓	✓	-
[86]	✓	-	✓	✓	✓	?	-
[2]	✓	-	✓	?	✓	✓	-
[47]	-	✓	✓	✓	?	✓	-

✓ ↔ provided - ↔ not provided ? ↔ not communicated

Some of the most important characteristics and properties of reviewed work are summarized in Table 2.1. We can notice that all these previous studies focus only on the architecture and QoS provided by memory schedulers and do not tackle neither the interconnect services nor the manner in which the masters requests are brought to the memory system. Although a few designs use back-end information in the front-end arbiter to efficiently schedule the requests, none of these designs shares any information with the interconnect structure that brings the requests to the memory controller. In addition, all previous designs are based on fixed architectures that do not provide the designer with any degrees of freedom to explore them.

2.2 On-chip interconnects

In the mid 90's **Hosseini-Khayat and Bovopoulos [36]** presented an efficient bus management scheme which allows the bus to support both continuous media transfer and regular random transactions. The algorithm ensures that continuous streams can meet their real-time constraints independently of random traffic, and random traffic is not delayed significantly by continuous traffic except when the traffic load is very high . In the early 2000 the applications needs in term of throughput have led the interconnection systems to the NoC idea that was presented by **Guerrier and Greiner [33]**, in which the support for these traffic

classes is still required. In 2001 **Dally and Towles [19]** showed that using a network to replace global wiring has advantages of structure, performance, and modularity. The on-chip network structures the global wires so that their electrical properties are optimized and well controlled.

From 2001 on, a large number of Networks-on-Chip have been proposed. Some examples are SOCBus [51], Octagon [43], QNoC [24], DSPIN [64], MANGO [7], Spidergon STNoC [16], Æthereal [31; 30], Nostrum [54], ANoC [6], Arteris NoC [4; 65], Hermes [55; 53], Chain [5], xPipes [18], QoS [26], SoCIN [82], Proteo [68] and Nexus [50].

Within the scope of this section, we are interested in NoC architectures which provide QoS to several classes of traffic.

QNoC [24] has four type of service levels. *Signaling* for urgent short packets that have the highest priority; *Real-Time* that guarantees bandwidth and latency for streamed audio and video; *Read/Write* for short memory and register accesses; and *Block-Transfer* for long messages such as DMA transfers. It combines multiple service levels (SL) with multiple equal-priority virtual channels (VC) within each level. The VCs are assigned dynamically per each link. A different number of VCs may be assigned to each SL and per each link .

The **DSPIN [64]** network-on-chip provides guaranteed service traffic by using VCs technique with a buffer per virtual channel. The advantage of this technique is a full separation of the traffic classes. Two traffic classes are defined, Best Effort (BE) and Guaranteed Service (GS) packets. Thus, when one traffic class is blocked the other is neither suspended nor blocked. Consequently, the deadlock situations can be avoided.

MANGO [7] stands for Message-passing Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces within a virtual channel approach. MANGO routers are the nodes of 2D mesh. They has five ports where one is a local port. The router consists of a BE router, a GS router and a link arbiter. The GS router is implemented as a non-blocking switching module. Each output port has seven GS communications and one BE communication. The GS communications are multiplexed using the virtual channel within a buffer per channel approach.

Spidergon STNoC [16] is a customizable on-chip communication platform that addresses heterogeneous, application specific requirements of MPSoCs. It allows customizable pseudo-regular or hierarchical topologies. As a programmable distributed hardware / software component, Spidergon STNoC offers a set of services to design advanced application features such as quality of service, security, and exception handling. Two virtual channels can be used to map traffic classes. In addition, two arbitration stages are implemented. The first one is an intra-channel arbitration, which arbitrates the packets going though the same channel. The second one is an inter-channel arbitration, which arbitrates between channels going through the same physical link.

The **Æthereal** NoC **[31; 30]** offers two types of service classes: guaranteed throughput (GT), and best effort (BE). Data that is sent on BE connections is guaranteed to arrive at the destination, but without minimum bandwidth and maximum latency bounds. End-to-end

flow control is used to ensure loss-less data transfer. GT connections use virtual channels in addition to time-division multiple access (TDMA) technique to give hard (worst-case) guarantees on minimum bandwidth and maximum latency. Both GT and BE connections use source routing, i.e. the path to the destination is decided at the initiator NI.

Nostrum [54] NoC offers guaranteed bandwidth and latency service, in addition to the basic service of best-effort (BE) service to traffic classes. The guaranteed bandwidth is accessed via *virtual circuits*. The virtual circuits are implemented using a combination of two concepts that it is called *Looped Containers* and *Temporally Disjoint Networks*. The Looped Containers are used to guarantee access to the network - independently of the current network load without dropping packets; and the TDNs are used in order to achieve several virtual circuits, plus ordinary BE traffic, in the network. The switching of packets in Nostrum is based on the concept of deflective routing, which implies no explicit use of queues where packets can get reordered, i.e. packets will leave a switch in the same order that they entered it. To avoid creation of hot-spots, routers send back-pressure signal to notify their neighbours of congestion ahead of sending packets.

ANoC [6] is an asynchronous NoC that uses two virtual channels (VCs) to provide best-effort on the low-priority VC and real time guarantees on a high priority VC. Complete paths are reserved for high-priority VC thus ensuring collision avoidance. If more simultaneous real-time connections are required to share a part of a path, the topology of the NoC has to be adapted to relax this condition. The data flow through the network is a wormhole routing. This has been chosen due to the small number of buffers required per node and the simplicity of the communication mechanism.

Arteris [4; 65] provides a commercial packet-switched NoC marketed as a bus replacement for SoCs. The NoC addresses the needs of complex designs that require high performance and a broad range of advanced interconnect features, such as QoS, multiple clock and power domain support, error handling, firewalls and extensive debug features. Packets are routed between network interfaces through a user defined topology. Not much is knowing regarding the implementation of this NoC.

The **Hermes [55; 53]** NoC is an infrastructure used to implement low area overhead packet switching NoCs, using mesh topology. It is quite classical in design (wormhole routing, credit based flow control), and it is considered here as the first open source NoC design. This infrastructure was extended to implement virtual channels. Hermes NoC implements either handshake or credit based flow control strategies. The VC implementation employs credit based flow control, due to the advantages over handshake. More services have been implemented in Hermes [11]. It consists in adding two kinds of QoS mechanisms: (1) priority with the support of two priority levels, (2) connection supporting hard QoS through circuit switching.

One of the main concerns in networks-on-chip is to be able to reduce the latency of operation and to increase the bandwidth. **Weber *et al.* [79]** outlined a simple QoS scheme that

offers service guarantees to each initiator regardless of the other initiators' offered traffic load. Three levels of QoS are available for each initiator: *priority* (optimized for low-latency up to maximal throughput); *bandwidth* (offering a guaranteed throughput); and *best-effort* (no service guarantees). **Grot *et al.* [32]** propose a QoS scheme called *Preemptive Virtual Clock* (PVC) specifically designed for cost and performance sensitive on-chip interconnects. There objectives are to minimize area and energy overhead, enable efficient bandwidth utilization, and keep router complexity manageable to minimize delay. They also aim at simplifying network management through a flexible bandwidth reservation mechanism to enable per-core, per-application, or per-user bandwidth allocation that is independent of the actual core/thread count. PVC requires neither per flow buffering in the router nor large queue in the source nodes. Instead, it provides fairness guarantees by tracking each flow bandwidth consumption over a time interval and prioritizing packets based on the consumed bandwidth. **Lee *et al.* [49]** present a new scheme called GSF (*Globally Synchronized Frames*) to implement QoS for multi-hop on-chip networks. GSF provides guaranteed and differentiated bandwidth as well as bounded network delay without increasing the complexity of the on-chip routers. They quantize the time into frames and the system only tracks a few frames into the future to reduce time management costs. Each QoS packet from a source is tagged with a frame number indicating the desired time of future delivery to the destination. At any point in time, packets in the earliest extant frame are routed with highest priority but sources are prevented from inserting new packet into this frame.

We summarize the most important features of the previous networks-on-chip in Table 2.2.

Table 2.2: Main features of the state of the art networks-on-chip providing QoS

NoC reference	Topology	Flow control	Virtual channels	Routing algorithm	Services	Other info.
Æthereal [31; 30]	Mesh	E2E and link level	Yes	static SR	GT; BE	TDMA
ANoC [6]	Custom	Flit handshake	Yes	static SR	RT/LL ; BE	Asynch.
Arteris [4; 65]	Custom	ready-valid	No	static SR	PR based; BE	-
DSPIN [64]	Mesh	CB	Yes	XY routing	GS; BE	Asynch.
Hermes [55; 53; 11]	Mesh	handshake; CB	Yes	XY routing	BE;GS	-
MANGO [7]	Mesh	handshake	Yes	static SR	GS; BE	Asynch.
Nostrum [54]	Mesh	hot potato	Yes	deflective routing	GT/LL; BE	TDMA
QNoC [24]	Custom	CB	Yes	XY routing	RT/LL; BE	Asynch.
SSTNoC [16]	Custom	CB	Yes	static SR	GT/LL; BE	FBA

SR : Source Routing
RT : Real Time
LL : Low Latency
BE : Best Effort
E2E : End to End
PR : Priority
GS : Guaranteed Service
GT : Guaranteed Throughput
CB : Credit Based
FBA : Fair Bandwidth Allocation

We can extract the following conclusions about the reviewed networks-on-chip:

- As it is seen all designs provide several levels of service to traffic classes. The continuity of these services is only guaranteed inside the NoC, this emphasizes the need of service coupling between the network and the targets.
- None of the previous NoCs have any mechanisms that use information coming from the memory subsystem in order to efficiently arbitrate and forward the packet going to the memory devices.

2.3 Combined interconnect-memory controller solutions

Few studies treat the off-chip memory system as a system matter, i.e. from the masters to the memory devices through the interconnect structures. In this section, we analyze the state of the art solutions that consider both network-on-chip and memory controller within the dynamic memory access process.

Ipek *et al.* [38] presented a new approach to design memory controller that operates using the principle of Reinforcement Learning (RL). The self-optimizing memory controller can continuously adapt its SDRAM command scheduling policy based on its interaction with the system to optimize performance. The proposed controller improves the performance of a set of parallel applications, running on a 4-core CMP with a single channel DDR2 memory subsystem, by 19% on average over a state-of-the-art FR-FCFS[7] scheduler. This design has only been evaluated in small CMPs, no information is given about its performance in bigger chips with *dynamic* traffic.

Sonics [72] has developed algorithms for memory load balancing in a multi-channel memory system (Interleaved Memory Technology or IMT) along with an advanced memory scheduler to optimize SDRAM access. The global address space covered by a SonicsSX SMART Interconnect address region may be partitioned into a set of channels. The channels are non-overlapping and collectively cover the whole region (see Figure 2.8). The number of channels for a region is a static value derived from the number of individual targets associated with the region. The memory load balancing unit distributes application workloads over memory channels through the interconnect. This solution requires fundamental modifications in the on-chip communication structure.

Daneshtalab *et al.* [21] propose a novel network interface architecture within a dynamic buffer allocation mechanism for the reorder buffer in order to increase the utilization and overall performance. The master network interface contains a shared reordering unit between the request and response path. In the slave network interface, they implement a dynamic memory controller made up of a scheduler and a physical interface which allows the slave network interface to be connected directly to the SDRAM DDR modules (see Figure 2.9). They use a constrained-random traffic in order to evaluate the performance of their solution in comparison with the baseline architecture. They prove that the utilization of memories is

[7] FR-FCFS : First Ready - First Come First Served

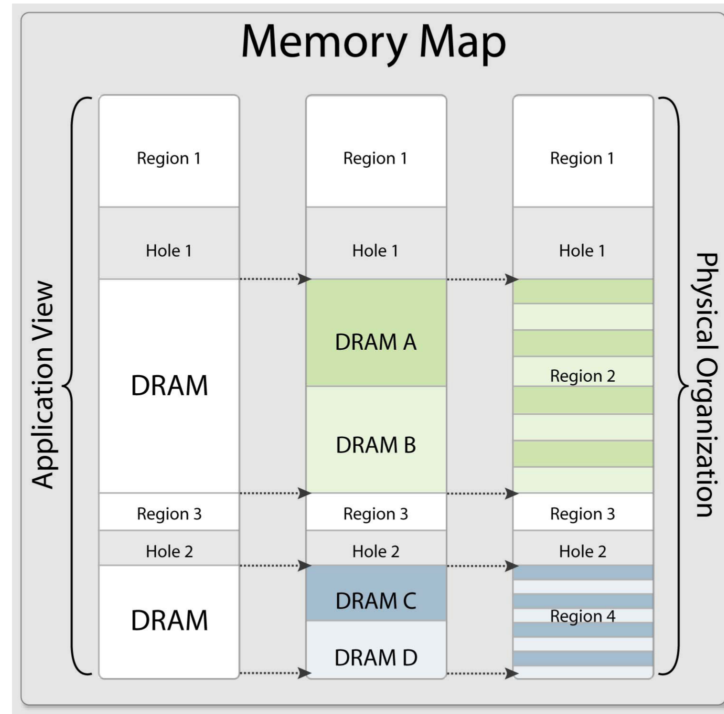


Figure 2.8: Memory map interpretation when using Sonics IMT, source [72]

improved by 22% and the average memory latency and average network latency are reduced by 19% and 12% respectively.

A network congestion-aware memory controller has been presented by **Kim *et al.* [44]**. It is based on the global information of network congestion and performs congestion aware memory access scheduling and congestion aware network entry control of read data. It prioritizes the *productive* requests from the uncongested area over *unproductive* ones. The *unproductive* requests come from the congested area and will only consume resource in memory controller without contributing to the system performance improvement. The experimental results obtained from a 5×5 tile architecture show an improvement of 18% in memory utilization. Unfortunately, the authors do not tackle the starvation problem when one shared memory system is in use.

Chen *et al.* [13] propose a micro-coded controller as a hardware module in each node to connect the core, the local memory and the network on chip. The proposed controller is programmable where the distributed shared memory functions (virtual to physical address translation, memory access synchronization) are realized using microcode. To enable concurrent processing of memory requests from the local and remote cores, the controller features two mini processors, one dealing with requests from local core and the other from remote cores. In order to evaluate the performance of this controller, two applications, matrix multiplication and 2D radix-2 DIT FFT, are mapped manually over the LEON3 processors. When the system size increases from 1 node to 64 nodes, the speedup of matrix multiplication goes up from 1 to 52; and the speedup of 2D radix-2 DIT FFT from 1 to 48. However, these

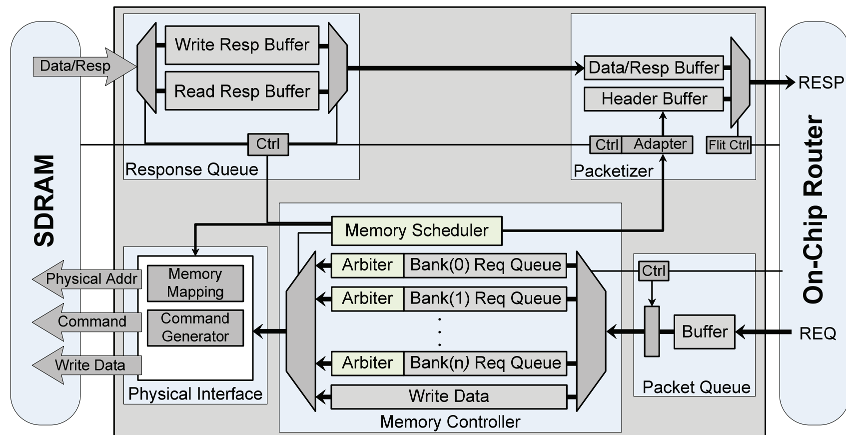


Figure 2.9: The proposed memory controller integrated in the slave-side network interface, source [21]

benchmarks have especially predictable access patterns.

Jang and Pan [40] presented an NoC router with an SDRAM-aware flow control. It improves the SDRAM utilization and latency, and decouples the Noc design cost from the number of SDRAM. The router arbiter schedules the packets to access SDRAM efficiently, the packets arrive at the memory subsystem into the order that is more friendly to SDRAM operations. In consequence, the complexity of the memory decreases while the memory performance is more improved. They analyzed the relation between the number of SDRAM-aware routers in the NoC and the system performance and hardware cost. The experiments show that the best choice is to replace three conventional routers to the SDRAM-aware routers. They propose later an application-aware NoC design for efficient SDRAM access which includes a flow controller [41]. They show that if the length of data requested by applications is neither the same as the length of data served by SDRAM nor a multiple of the length of data served by SDRAM, unnecessary data may be accessed and then thrown away. Therefore, the access granularity mismatch problem should be considered. They propose an SDRAM access granularity matching (SAGM) NoC design, which is based on SDRAM access granularity. SAGM splits a packet into short fixed-length packets and then schedules them by a specific flow controller which provides various priority services with few penalties. Their experimental results show that the application-aware NoC design improves on average 32.7% memory latency for latency-sensitive cores and on average 3.4% memory utilization compared to the SDRAM aware flow control performance [40]. Nevertheless, the presented solutions require additional and heavy hardware to be implemented in the network router, which negatively impacts the network overhead.

Diemer and Ernst [22] proposed a flow control scheme to implement service guarantees. It uses available buffer space to allow access of *guaranteed throughput* traffic (GT) during idle times. In return, the *best-effort* traffic (BE) is prioritized over GT traffic as long as the GT buffers are sufficiently filled. Unlike many flow control mechanisms, *Back-Suction* prioritizes

best-effort traffic whenever possible for optimal latency and throughput of general purpose applications. The experimental evaluation has demonstrated an improvement of the BE latency up to 32% over a standard prioritization scheme.

2.4 Conclusion

In this chapter, we presented the state of the art work on the SDRAM access through on-chip interconnects. Many networks-on-chip and memory controllers have been proposed to enhance the efficiency of the shared memory and meet the requirements of the processing engines connected through the network.

Advanced memory controllers and arbitration policies are presented in [2; 10; 47; 56; 57; 84]. Even if some of these designs present configurable memory controllers, the architecture exploration is restricted to limited sets of parameters such as FIFOs depth, data bus size, QoS level and bandwidth distribution. Moreover, none of the previous work present a totally configurable architecture to give the designer the liberty of exploring and adapting the memory controller architecture to measure the impact of its architecture on the system performance.

Many networks-on-chip provide guaranteed service to traffic classes [24; 64; 7; 16; 30; 54; 6]. A few flow controllers and arbitration schemes take into consideration the specificity of the SDRAM as a target [13; 41; 40; 72]. However, these solutions predict the state of the SDRAM, and require heavy arbitration schemes in the routers. None of them use information on the real memory state neither within its arbitration algorithms nor within the flow control.

In order to fill the gap of the state of the art solutions, the following chapters will detail our contribution. Firstly, we will focus on the specification and the design of a totally customizable architecture of memory controller which will be presented as a library of building components. This library provides the designer of the memory system with the necessary configurable components to build a specific memory controller with all required measuring tools to evaluate its performance. Secondly, in order to make the network-on-chip knowledgeable of the SDRAM state, we introduce a new flow control protocol between the network and the memory controller. This protocol exploits the memory controller state within its control policy and guarantees the extension of services from the network to the memory controller.

CHAPTER 3

Memory Controller Customizable Architecture

Contents

3.1 Introduction 37
3.2 DDR3 SDRAM operations 38
3.3 Design abstraction in system modelling 41
3.4 Design approach 41
3.5 Assumptions 42
3.6 Front-end building components 42
3.6.1 Memory mapping 42
3.6.2 Generic queue 43
3.6.3 Capture unit 44
3.6.4 Insertion unit 47
3.6.5 Generic arbiter 51
3.6.6 Flow control 53
3.6.7 Re-ordering unit 55
3.6.8 Summary 55
3.7 Examples of memory controller front-end 56
3.7.1 Memory controller Alpha 56
3.7.2 Memory controller Beta 56
3.7.3 Memory controller Gamma 57
3.7.4 Summary 58
3.8 Back-end building components 59
3.8.1 DDR3 SDRAM commands generator 60
3.8.2 Memory manager 60
3.8.3 Data handler 61
3.9 DDR3 SDRAM model 62
3.10 Conclusion 62

MAIN memory system designs and optimizations have become an increasingly important factor in determining the overall system performance. Although reducing directly the physical memory access latency is limited by the SDRAM technology advancement and cost considerations, the advance of modern memory systems has provided many opportunities to reduce the average latency for concurrent memory accesses [85]. The performance of a memory system is tightly correlated with the performance of the memory device and the memory controller. This is why the architecture of the memory controller has a tremendous impact on the overall system performance. In this chapter we will introduce a *Memory Controller Customizable Architecture* which is a totally configurable architecture used to study the memory access process through on-chip interconnects. To the best of our knowledge, such an architecture is the first of its kind.

3.1 Introduction

In the past few years, performance analysis of NoC-based SoCs has been done with focus only on the network. The use of dump slaves that return responses either immediately or with fixed delay has been recurring. However, such performance analysis is naive, and has clear limitations because it does not include the shared SDRAM access latency, which makes it incompatible with the evaluation of today's SoCs.

Srinivasan and Salminen [74] present a model for an SDRAM with its controller. They concentrate on the SDRAM device to provide a statistical model for the memory access latency. They only consider the SDRAM timing constraints without taking into account the arbitration phase in the memory controller front-end that precedes the memory device access. Their model reflects more precise values of memory access latency compared to previous models, but still does not include the total memory access latency which is due to the whole memory subsystem, i.e. the front-end, the back-end and the memory device. **Kumar *et al.* [45]** present a memory architecture exploration framework for SPRAM-Cache based memory architectures. Their framework allows the designer to determine multiple optimal design points to choose the best memory configuration. Nevertheless, the configuration of the memory system is limited to the cache-size, cache-block size, cache associativity and main memory size. Neither the aspect of real memory access latency nor the scheduling phases are included in their framework, which makes the exploration of the memory architecture very restricted.

We are going to introduce an innovative customizable memory controller architecture based on fully configurable building components, and design a high level cycle approximate model for it. Contrary to the state of the art architectures, *our configurable architecture includes all aspects that play a role in the memory access latency, ranging from the FIFOs depth in the interconnection interface, to the memory device timing constraints.* Figure 3.1 shows the principal building components of a memory controller and their instantiation in the front-end and back-end. The front-end precisely simulates all delays related to the size of its

queues, arbitration policies, and response re-ordering aspect. While the back-end simulates all delays due to the memory device timing constraints and the memory access granularity. Consequently, our architecture covers all latency aspects inside the main shared memory subsystem in a SoC, and enables real and precise performance analysis.

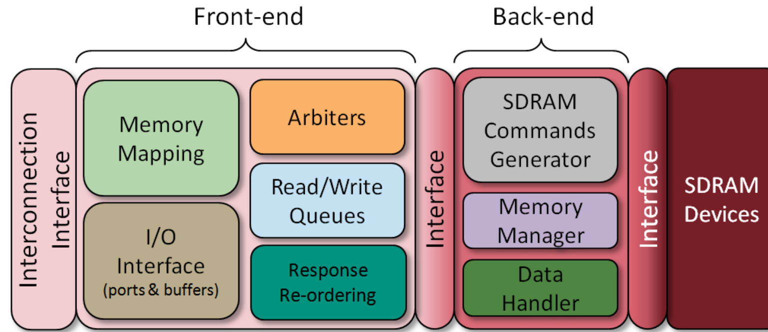


Figure 3.1: Generic architecture of a memory controller connected to SDRAM devices

This chapter is organized as follows: Section (3.2) introduces the complexity of the DDR3 SDRAM access and the memory timing constraints. Sections (3.3) and (3.4) deal with the system level modelling and our design approach. In section (3.5) we express some assumptions about the architecture design. In section (3.6), we design the cycle approximate building components of the memory controller front-end. Here we provide a simple way to adapt existing mechanisms used in industrial memory controllers, and we also introduce some novel algorithms. Three models of industrial memory controller front-ends are presented in section (3.7). In section (3.8), we design a cycle approximate DDR3 SDRAM back-end. We finally show, in section (3.9), a simple model for DDR3 SDRAM.

3.2 DDR3 SDRAM operations

DDRn SDRAMs are three-dimensional memories with the dimensions of bank, row, and column. Each bank is organized as a two-dimensional array of SDRAM cells, consisting of multiple rows and columns. It independently operates of the other banks and contains an array of memory cells that are accessed an entire row at a time. When a row of this memory array is accessed, the entire row of the memory array is transferred into the bank's row buffer. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. While a row is active in the row buffer, any number of reads or writes may be performed. The DDR3 SDRAM operations are driven by a memory manager which uses the finite state machine (FSM) of each bank to schedule appropriately the commands to it.

Each bank has its own FSM, which drives all commands going to SDRAM module in a precise order. Figure 3.2 shows a simplified version of the state diagram of the DDR3 SDRAM. We omit in this version the powering down, the calibration and the initializing states because they do not have any impact during the normal operating of the SDRAM. Table 3.1

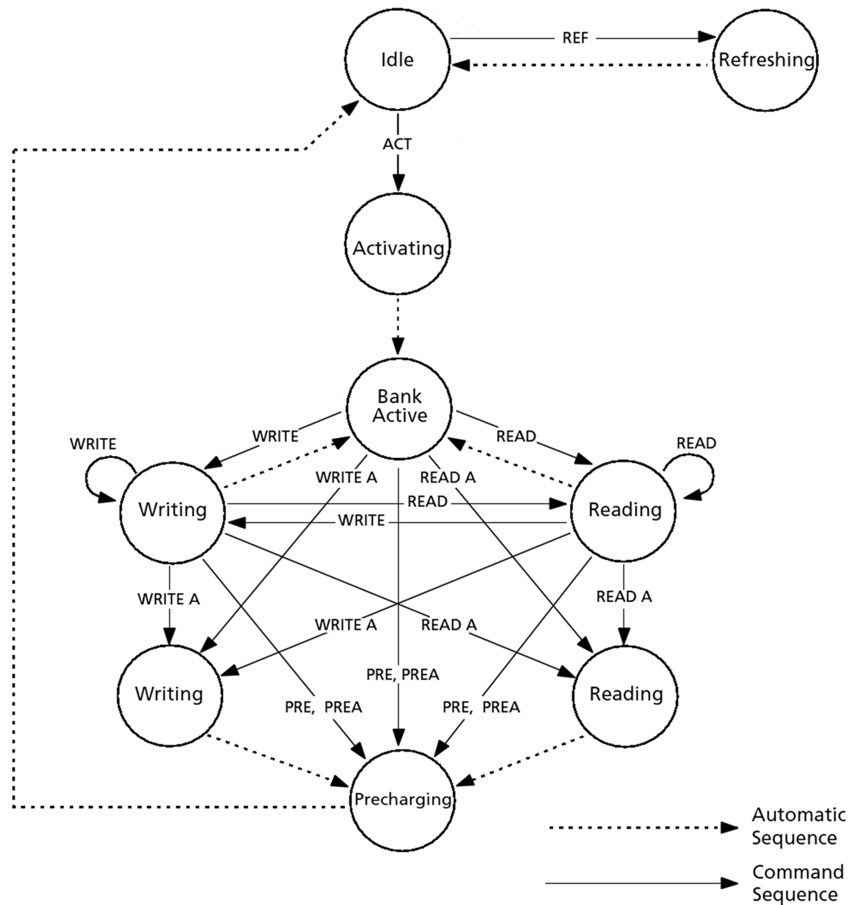


Figure 3.2: Simplified diagram of the DDR3 SDRAM FSM

Table 3.1: DDR3 SDRAM state digram command definitions

Abbreviation	Function	Short description
ACT	Activate	Open a row by copying it from the SDRAM matrix to the bank buffer
PRE	Precharge	Close an open row by copying it from the bank buffer to the SDRAM matrix
PREA	Precharge All	Close all open rows in all banks by copying them from the bank buffer to the SDRAM matrix
Read[a]	Read operation	Read from an open row in a bank
Read A[a]	Read with Auto precharge[c]	Read from an open row in a bank, and precharge the row upon the completion of the read operation
Write[b]	Write operation	Write in an open row in a bank
Write A[b]	Write with Auto precharge[c]	Write in an open row in a bank, an precharge the row upon the completion of the write operation
REF	Refresh	Refresh a bank for retaining the data in the memory matrix

[a] The READ operation is burst oriented, it can be either 4 or 8 transfers
[b] The WRITE operation is burst oriented, it can be either 4 or 8 transfers
[c] WRITE A and READ A commands are used when the row access policy is *closed row.* That means the open row will be closed upon the termination of the read/write operation

shows the state diagram command definitions. Note that some transitions between states are done upon the receiving of a command, e.g. the transition between the *Bank Active* state and the *Reading* state. Other transitions are automatically done, e.g. the transition between the *Precharging* state and the *Idle* state.

It is possible to interleave preparation commands (*Precharge*, *Activate*, and *Refresh*) to multiple banks of the memory, thereby increase the effective bandwidth. At the beginning of a read or write operation, a bank must first be activated based on the incoming address. This does not affect other banks because banks are independent. At the same time, the accessed row will become active and after that accesses to certain columns are possible. Hence, it takes some time to read the first data but the consecutive ones come faster.

The transition between the states in the bank FSM shown in Figure 3.2 takes time. These time values are the timing constraints of a DDR3 SDRAM, and they determine the latency of the memory operations. We provide in Table 3.2 a detailed description of the timing constraints.

Table 3.2: DDR3 SDRAM timing parameters description

Parameter	Description	Example[a]
tRL	Read Latency: minimum READ command to first READ-DATA delay	5
tWL	Write Latency: minimum WRITE command to first WRITE-DATA delay	5
tCCD	Column to Column Delay: minimum READ to READ or WRITE to WRITE delay	4
tRAS	Row Access Strobe delay: minimum ACTIVATE to PRECHARGE delay	14
tRCD	Row to Column Delay: minimum ACTIVATE to internal READ or WRITE delay	6
tRC	Row Cycle delay: ACTIVATE to ACTIVATE or REFRECH delay in a bank	21
tRP	Row Precharge delay	6
tRTP	Read To Precharge: minimum READ to PRECHARGE delay	4
tREFI	REFresh Interval: average periodic interval for the REFRESH	1560
tRFC	ReFresh Cycle delay	64
tWR	Write Recovery: minimum delay between last WRITE DATA and PRECHARGE	15
tRTW	Internal Read to Write delay = RL + tCCD + 2tCK - WL	6
tWTR	Internal Write To Read delay	4
tFAW	Time window in which at most 4 bank activation commands could be sent	20

[a] In clock cycles

As the row activation process correspond to copying an entire row from the memory matrix in the row buffer in a bank, this operation consumes a lot of power, leading to a power consumption peak in the memory module. For this reason, the DDR3 SDRAM standard [77] limits the number of activation commands to 4 with a tFAW time window (time Four-Activation Window). This time window depends on the row size (1KB or 2KB).

In order to retain data, all row in the memory array must be periodically refreshed. This *Refresh* command is non persistent, so it must be issued each time a refresh is required. The DDR3 SDRAM requires refresh cycles at an average periodic interval of *tREFI*. To allow for improved efficiency in scheduling and switching between tasks, some flexibility in the absolute refresh interval is provided. A maximum of 8 refresh commands can be postponed during operation of the DDR3 SDRAM.

The SDRAM burst length (BL) denotes the minimum granularity at which SDRAM ac-

cesses are done. In other words, each access to the SDRAM must be BL number of transfers of memory data. SDRAM BL, multiplied by the double width of the memory data bus, thus returns the minimum number of bytes that are transferred per access. DDR3 SDRAM supports a BL of four and eight [77]. As the DDR3 SDRAM is able to transfer data on both rising and falling edges of the memory clock, a burst of 8 transfers takes 4 clock cycles.

3.3 Design abstraction in system modelling

One of the important issues in performance evaluation is the trade-off between the rapidity in obtaining results and the accuracy of them due to different levels of abstraction. During the design of a SoC, the system is modelled in several abstraction levels. Once the model satisfies the constraints of an abstracter level, then it is refined toward a more detailed one. From a performance-analysis point of view, a lower level of abstraction gives more accurate results but causes more complexity in modelling and takes more time to simulate. Raising the level of abstraction is on the basis of hiding unnecessary details of an implementation by summarizing the important parameters into a more abstract model [27; 42].

Even though a higher level of abstraction can lead to ignore more details and consequently to lose the accuracy of results, it provides faster performance results by enhancing critical parameters like simulation speed, flexibility and time to develop. Ranging from functional to cycle-accurate bit-accurate model, each level of abstraction introduces new model details. In system modelling, the following levels of abstractions are usually considered:

- **Transaction Level**, structural models with atomic transactions;
- **Cycle Approximate**, includes the time notion and considers the transactions latency;
- **Cycle Accurate Bit Accurate**, includes the time notion and can accurately model the timing properties by executing the finite state machines of the device.

3.4 Design approach

Creating a memory system model for performance analysis can hardly be done due to the sheer complexity of the SDRAM controllers, and the need to adapt to newer SDRAM technologies as they emerge. In this chapter, we propose a modular approach relying on a set of ad-hoc components with parameters that can be used to generate a highly abstracted SDRAM controller and memory. The objective is to keep the abstraction level high enough to make development easy, and at the same time, capture the critical parameters that significantly influence the performance of the memory system.

We aim at designing a complete memory system made up of a customizable memory controller and SDRAM devices. Thanks to the numerous parameters, this memory controller is cycle approximate, and developed using a high level abstraction language. These components are easy to interface with each other, which makes their instantiation simple with the

goal of building a given architecture of a memory controller. This architecture will have the flexibility, scalability and the accuracy of a cycle approximate model.

Figure 3.1 on page 38 shows the principal building components of a memory controller and their instantiation in the front-end and back-end.

This memory controller model is designed in the context of performance analysis through simulations. It is not adapted to be used in the context of architecture validation or system verification.

3.5 Assumptions

Throughout this chapter, we are going to show several algorithms which facilitate the comprehension of the behaviour of some building components. Some conventional notations are used:

- The point operator . means a member of an object. The member can be a variable or a method.
- All variable written in *italic* are locales variables, or arguments for a method.
- All variable written in **sans serif** are global variables or structures.
- We use sometimes *for* loops inside algorithms to highlight the fact that we scan all elements inside a queue. These loops do not mean that we need several clock cycles to scan the whole queue. This is done within one clock cycle in our model, using hardware level parallelism.

3.6 Front-end building components

We provide here the description of the front-end building components of the configurable architecture of memory controller.

3.6.1 Memory mapping

The purpose of memory mapping is to decode logical addresses into physical addresses. We mean here by logical addresses the addresses that initiators send with their requests towards the memory system. The physical addresses is the translation of the logical addresses in *bank number*, *row number*, and *column number* format, which corresponds the 3D organization of an SDRAM.

There are different kinds of memory mapping schemes with different properties. Most used are:

- Row-Bank-Column (RBC), when the MSBs are the *row bits*, the LSBs are the *column bits* and the middle bits are the *bank bits*.

- Row-Column-Bank (RCB), when the MSBs are the *row bits*, the LSBs are the *bank bits* and the middle bits are the *column bits*.
- Bank-Row-Column (BRC), when the MSBs are the *bank bits*, the LSBs are the *column bits* and the middle bits are the *row bits*.

The *column bits* determine the row size (also called page size), which is 1KB or 2KB in general purpose DDR3 SDRAMs. The *row bits* define the number of rows in each bank. The *bank bits* determine the number of banks in the SDRAM device (8 bank in DDR3 SDRAM). Figure 3.3 shows how a logical address can have several physical interpretations according to memory mapping in use.

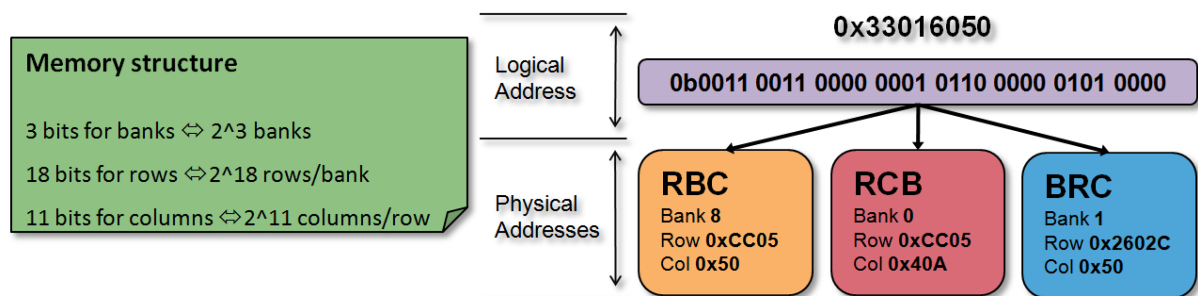


Figure 3.3: Examples of memory mapping

The RBC mapping has an advantage when the logical addresses are consecutive. A minimum amount of rows need to be accessed, effectively reducing the number of row activations and precharges. The RCB mapping helps to interleave the bank accesses in case of consecutive logical addresses.

When a request passes through the memory mapping unit, its address is decoded and the Bank/Row/Column triplet is added in a specific field. This information is necessary to help the arbitration unit to schedule the requests and optimize the efficiency of the memory data bus.

3.6.2 Generic queue

The generic queue model is used in all buffering stages in the memory controller and can be used between two building blocks or inside a building block. It contains several independent queues which can work in parallel. Their default behaviour is FIFO: we insert elements at the back using *store()*, and remove them at the front using *pop()*. We developed this model as a set of randomly readable/writeable queues. We can store and extract elements at any address in the queues. These advanced mechanisms are helpful when the generic queue is used in the arbitration unit.

The stored elements are objects that represent messages. The number of instantiated queues is one of the constructor's parameters. Figure 3.4 shows a simplified architecture of the generic queue.

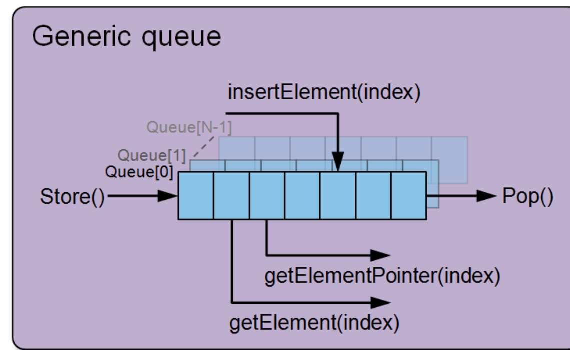


Figure 3.4: Simplified architecture of the generic queue

Before instantiating a generic queue we have to specify its length in addition to the number of the queues we want to implement. This model is able to watch the time that the elements spend inside each queue, and react when the element's age crosses the threshold of $maxAge$ clock cycles. The reaction may be the incrementation of the priority level of that element for example. The ageing mechanism is a mean to prevent starvation problems for some initiators. Table 3.3 summarizes the main features of the generic queue model.

3.6.3 Capture unit

Capture units are used in conjunction with a generic queue. Such a unit is used when the queue contains requests that address the memory device. The capturing unit is aware of all requests that the generic queue contains. Its role is to identify a request inside a queue and to extract it according to several rules. This mechanism is often required in schedulers. The use of capture unit in conjunction with a generic queue is shown in Figure 3.5.

Most schedulers in memory controllers need to know the priority of the requests that should be scheduled. Other rules can be added to the capture process like *row-hit same direction*, *row-hit opposite direction*, and *row miss different bank*. These rules can simplify the task of the memory controller back-end by forwarding the requests in a friendly order to the SDRAM access patterns.

Although these capture rules are already used in industrial memory controllers, very few publications have described them. We detail in this subsection the behaviour of the capture unit according to each rule. We also introduce by the end of this subsection a new rule called *row miss different bank*.

Table 3.3: Brief description of the generic queue model

Kind	Name	Description
Variables	int $maxLength$	Maximum number of queue slots
	int $numOfQueues$	Number of implemented queues
	bool $ageing$	Ageing mechanism enable/disable
	int $maxAge$	The maximum age that an element may have before increasing its priority
Constructors	cGenericQueue(int L, int N)	Basic constructor, implement N queues with L queue slots
	cGenericQueue(int L, int N, bool A, int maxA)	Implement N queues with L queue slots, and enable the ageing mechanism
Methods	storeElement(int N, cObject* El)	Insert the element El at the back of the queue number N
	insertElementAfter(int N, int M, cObject *El)	Insert the element El in the queue number N after the element number M
	insertElementBefore(int N, int M, cObject *El)	Insert the element El in the queue number N before the element number M
	getElement(int N)	Remove the first element from the queue number N
	getNthElement(int N, int M)	Extract the element number M from the queue number N and shift right all elements between the back of the queue and slot number M+1
	getNthElementPointer(int N, int M)	Return the pointer of the element number M in the queue number N
	getLength(int N)	Return the number of elements in the queue number N
	applyAgeing(int N)	Increment the age of all elements in the queue number N. If the element age is greater than $maxAge$, increment then the priority of this element (only if $ageing == true$)
	getFreeRoomNumber(int N)	Return the number of the free slots in the queue number N

Highest priority capture rule

The idea here is to find the request in the queue which has the highest priority. If two or more requests have the same priority, the nearest one to the queue head will be selected. This process in detailed in Algorithm 10 in Annex B.

Row hit same direction capture rule

This rule avoids to switch the direction of the memory data bus between read and write. The *write to read* delay and the *read to write* delay are not negligible. Therefore, minimizing the bus turnarounds number leads to higher memory bus efficiency. In addition to the return of the appropriate request index in the queue, this process updates a global variable (**rowHitSameDirection**) in the capture unit. Algorithm 1 describes this process.

Row hit opposite direction capture rule

The penalty of the bus turnaround is less than the row miss penalty. If we do not manage to find a request that access an open row in the same direction (**rowHitSameDirection**=false),

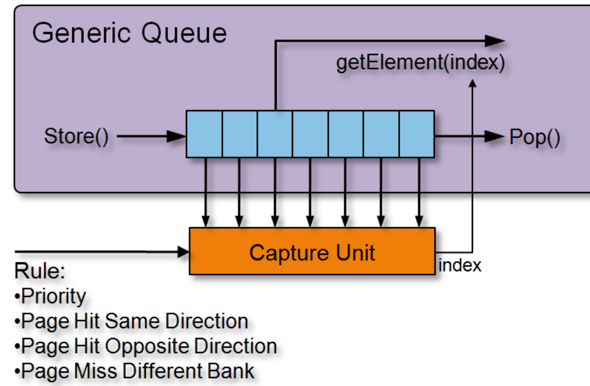


Figure 3.5: capture unit in conjunction with a generic queue

Algorithm 1 Row hit same direction capture

local variables $direction, lastDirection, bankIndex, row, lastRow, L, selectedRequestIndex$;
$L \leftarrow$ **genericQueue**$[queueIndex].getLength()$;
$selectedRequestIndex \leftarrow 0$;
rowHitSameDirection $\leftarrow false$
for $(i = 0; i < L; i++)$ **do**
  $direction \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getDirection()$;
  $bankIndex \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getBank()$;
  $row \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getRow()$;
  $lastRow \leftarrow$ **bankState**$[bankIndex].getLastRow()$;
  $lastDirection \leftarrow$ **bankState**$[bankIndex].getLastDirection()$;
  if $(row = lastRow$ **and** $direction = lastDirection)$ **then**
    rowHitSameDirection $\leftarrow true$;
    $selectedRequestIndex \leftarrow i$;
    $break$;
  end if
end for
return $selectedRequestIndex$;

we can favour the bus turnaround penalty rather than the row miss penalty. This process also updates a global variable (**rowHitOppositeDirection**) in the capture unit. Algorithm 11 in Annex B shows this process.

Row miss different bank capture rule

When we already know there is no request in the queue that accesses an open row in a bank, we should close a row and open another one. As we have seen earlier, this is the highest delay we can have due to a row miss. We propose here a simple and helpful method that minimizes the row miss impact by avoiding to do two consecutive row misses in the same bank. Our seeking process is shown in Algorithm 2.

Algorithm 2 Row miss different bank capture

```
local variables bankIndex, L, selectedRequestIndex;
L ← genericQueue[queueIndex].getLength();
//we assume that rowHitSameDirection = false and rowHitOppositeDirection = false
selectedRequestIndex ← 0;
for (i = 0; i < L; i++) do
    bankIndex ← genericQueue[queueIndex].getRequestPointer(i).getBank();
    if (bankState[bankIndex].getIsLastReqRowMiss() = false) then
        bankState[bankIndex].setIsLastReqRowMiss(true);
        selectedRequestIndex ← i;
        break;
    end if
end for
return selectedRequestIndex;
```

3.6.4 Insertion unit

Similar to the capture unit, the insertion unit is also used in conjunction with a generic queue. The insertion unit is aware of all requests that the generic queue contains (in addition to the request we want to insert). The necessary pieces of information which are required for the insertion process are transferred to the insertion unit. Some of these pieces of information are: the transaction source identification (*TrSourceID*), the priority(*Pr*), and the bank/row that the request wants to access.

This unit can use one or more rules to insert a request. According to the chosen rule(s), the insertion unit will determine the insertion position in the generic queue, and then will forward it to the generic queue where it will be used by the *insertElement()* method. Figure 3.6 shows the insertion unit in conjunction with the generic queue.

Randomly readable and insertable queues have been subject to many publications. However, very few insertion techniques used in memory controllers have been published yet to the best of our knowledge. Below, we present the most important insertion rules that a memory controller can use to increase the memory efficiency while guaranteeing the data consistency. We also introduce a novel insertion rule that tries to maximize the bank interleaving

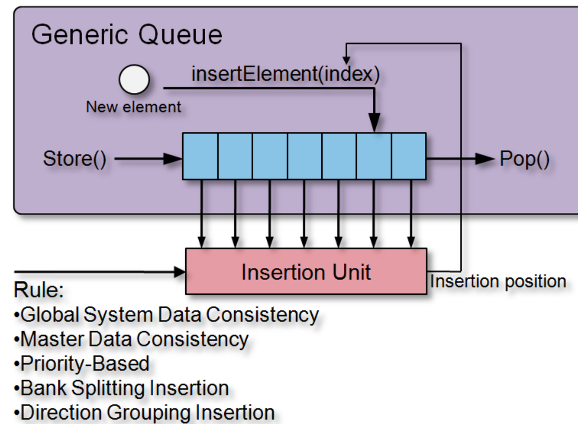


Figure 3.6: Insertion unit in conjunction with a generic queue

mechanism.

Global system data consistency insertion rule

The order in which read and write requests are processed in the memory controller is critical to proper system behaviour. While reads and writes to different addresses are independent and may be re-ordered without affecting the system performance, reads and writes that access the same address are significantly related. If we have a read after a write to the same address, then we reposition the read before the write, the read would return the original data, not the changed data. Similarly, if the read was requested ahead of the write, but accidentally positioned after the write, then the read would return the new data, not the original data prior to being overwritten. These are significant data consistency mistakes.

This rule guarantees the data consistency for all masters communicating with the memory system. The principle is to preserve the order of the requests that access the same bank, row, and column in the memory, and to determine the right *insertion point* according to this rule. Algorithm 12 in Annex B describes this rule.

Master data consistency insertion rule

This rule guarantees the data consistency for one master that communicates with the memory system. The principle is to preserve the order of the requests coming from this master. Algorithm 3 shows how to determine the right insertion point according to this rule.

Priority-based insertion rule

Priorities are used to distinguish important requests from less important requests. The insertion algorithm will attempt to place higher priority requests ahead of lower priority requests, as long as the data consistency is guaranteed. Higher priority requests will be placed lower in

Algorithm 3 Master data consistency insertion

```
local variables existingTrID, newTrID, L, insertionPosition, lowerBoundary;
masterDataConsistencyLimit ← 0;
lowerBoundary ← 0;
L ← genericQueue[queueIndex].getLength();
newTrID ← newRequest.getTrID();
insertionPosition ← L;
if (L = 0) then
   masterDataConsistencyLimit ← 0;
else
   for (i = (L − 1); i ≥ lowerBoundary; i − −) do
      existingTrID ← GenericQueue[queueIndex].getRequestPointer(i).getTrID();
      if (newTrID = existingTrID) then
         masterDataConsistencyLimit ← i;
         insertionPosition ← (i + 1);
         break;
      end if
   end for
end if
return insertionPosition;
```

the queue. If one or more requests in the queue have the same priority as the new request, the new request is inserted after them. Algorithm 13 in Annex B describes this process.

Direction grouping insertion rule

The memory suffers a small timing overhead when switching between read and write modes. For efficiency, the insertion unit will attempt to place a new read request sequentially with other read requests in a queue, or a new write request sequentially with other write requests. Algorithm 14 in Annex B describes this process.

Bank splitting insertion rule

Before accesses can be made to two different rows within the same bank, the first active row must be closed (*precharged*) and the new row must be opened (*activated*). Both activities require some timing overhead. We present here an innovative rule that attempts to insert the new command into a queue such that preparation commands to other banks may execute during this timing overhead. Algorithm 4 describes this novel insertion rule.

We presented until here the rules that the insertion unit can use. We would like to highlight the fact that two or more rules can successively be used in order to define other more sophisticated rules. To do this, the *insertion position* which is obtained by the application of a rule must be used as *lower boundary* for the following rule. For example, if we want to insert a request according to its priority, and then find the best insertion point to maximize the bank interleaving, we should use the *priority limit* as *lower boundary* in the bank splliting rule.

Algorithm 4 Bank splitting insertion

```
local variables existingBank, newBank, existingRow, newRow, L, lowerBoundary;
lowerBoundary ← 0;
L ← genericQueue[queueIndex].getLength()
newBank ← newRequest.getBank();
newRow ← newRequest.getRow();
insertionPosition ← L;
if (L ≠ 0) then
    for (i = lowerBoundary; i < L; i + +) do
        existingBank ← genericQueue[queueIndex].getRequestPointer(i).getBank();
        existingRow ← genericQueue[queueIndex].getRequestPointer(i).getRow();
        if (newBank = existingBank and newRow ≠ existingRow) then
            insertionPosition ← L;
            //insert next to a command to different bank
            for (j = L − 1; j > i; j − −) do
                existingBank ← genericQueue[queueIndex].getRequestPointer(i).getBank();
                if (newBank ≠ existingBank) then
                    insertionPosition ← (j + 1);
                    break;
                end if
            end for
            break;
        else if (newBank = existingBank and newRow = existingRow) then
            //the address collision detector is disabled
            insertionPosition ← (i + 1);
            break;
        else
            insertionPosition ← L; //insert at the end of the queue
        end if
    end for
end if
return insertionPosition;
```

3.6.5 Generic arbiter

The scheduling block is the core of the memory controller. It is responsible for scheduling the requests from the inputs according to specific arbitration policies. The architecture of this arbiter is configurable in that we can choose the number of inputs and the arbitration algorithm(s) we want to apply.

In a typical memory controller architecture, the arbiter is often connected to buffering elements in order to schedule efficiently the requests. In our generic architecture, we use the generic queue as a buffering element. Figure 3.7 shows the generic arbiter in connection with a generic queue.

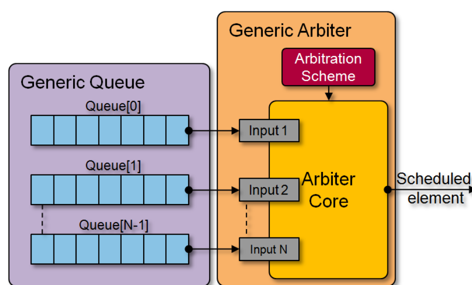


Figure 3.7: Generic arbiter connection with a generic queue

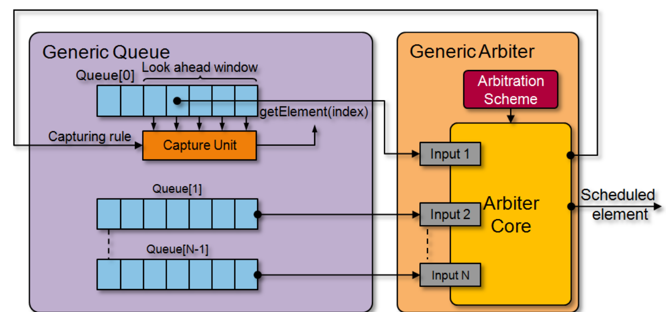


Figure 3.8: Generic arbiter connection with a generic queue including a capture unit

In order to minimize the overhead of bank conflict and the bus turnaround, some arbiters do not just simply select one of their inputs, they go further through a *look-ahead window* in the selected queue to schedule an entry that reduces the overhead. This is done in our model by coupling the generic arbiter with the generic queue and the capture unit. Figure 3.8 shows an example of this coupling when the *input 1* is selected by the arbiter core. Note that only one capture unit will be required to extract the right element from the selected queue inside the look-ahead window.

Solving conflicts in architectures that include shared resources is a traditional task. Many memory controller architectures have inherited techniques from several domains to arbitrate requesters that want to access a shared resource. Among these arbitration policies we cite *round-robin* and *least recently used*. We know that more sophisticated arbitration policies are used in modern memory controllers. Nevertheless, very few policies have been subject to publication.

The following subsections describe the scheduling policies that the generic arbiter supports. Some of them are traditional and easy to implement such as *round robin, least recently used*, and *priority*. The remaining scheduling policies are more sophisticated and combine several features in one algorithm.

Round-Robin scheduling policy

Round-Robin (RR) is one of the simplest scheduling algorithms, which assigns time slices to each request in equal portions and in circular order, handling all requests without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free. Algorithm 15 in Annex B shows an example of implementation of this arbitration.

Least-Recently-Used scheduling policy

LRU algorithm favours the least recently used inputs first. This algorithm requires keeping track of what was used at each arbitration cycle. Algorithm 16, 17 and 18 in Annex B show the initialization, the arbitration and the updating process of this arbitration policy.

Round-Robin then bandwidth scheduling policy

Fair bandwidth distribution is one of the basic function of an arbiter. The arbiter can measure the bandwidth that an input consumes during a time slice. If one of the inputs reaches the maximum amount of data, the arbiter will not grant this input until the end of the current time slice, and will favour another input that did not yet reach its maximum bandwidth. The bandwidth-based arbitration is often coupled with other policies such as round robin. Algorithm 5 shows the Round-robin then bandwidth arbitration.

Algorithm 5 Round-Robin, bandwidth scheduling

```
local variables inIndex, selectedInput;
inIndex ← 0;
for (in = 0; in < inputsNumber; in++) do
    inIndex ← (lastSelectedInput + 1 + in);
    if (inIndex > inputsNumber) then
        inIndex ← (inIndex − inputsNumber);
    end if
    if (inputValid[inIndex] = true and inputBWOverflow[inIndex] = false) then
        selectedInput ← inIndex;
        break;
    end if
end for
lastSelectedInput ← selectedInput;
updateInputBWOverflow(selectedInput);
return selectedInput;
```

Priority scheduling policy

Priority scheduling is one of the basic scheduling policies. In case of conflict, the input that has the higher priority will win the arbitration. This algorithm is really simple and easy to implement, but could create a starvation case for the inputs which have low priority.

Priority then Round-Robin scheduling policy with bandwidth limiter

We present here a novel and simple arbitration policy that combines the priority policy with the Round Robin policy and a bandwidth limiter. The priority scheduling process is firstly activated. In case we have two (or more) inputs that have the same priority level, a Round Robin scheduling will be used. The inputs that have exceeded their allowed bandwidth have to wait until their average bandwidth becomes below the allowed threshold. We present our arbitration policy in Algorithm 6. This algorithm is not starvation free because the Round Robin policy is only used between the inputs that have the same priority level. In order to avoid starvation situations, this algorithm must be used in addition to an *ageing mechanism* that increments the priority of the requests every *maxAge* clock cycles.

Finally, when the generic arbiter selects one of its inputs according to the scheduling policy in use, it can forward a *capture rule* to the capture unit in order to extract a specified request from this queue (see Figure 3.8). This is helpful if the arbiter needs to use information coming from the bank-end such as open row in a bank and memory bus direction (read or write).

3.6.6 Flow control

Flow controller determines how the memory controller resources are allocated. It regulates the commands and data flow by monitoring the queue occupancy.

Stop and go flow control

This regulation is achieved by monitoring the load and store queue occupancy and asserting specific signals in case the occupancy level becomes dangerous.

Data consistency flow control

For efficiency reasons, the elements might be arbitrated out of order from their arrival time. In case we use a capture unit in conjunction with a generic queue, there is a chance that the master data consistency is violated. The capture unit does not remember the last extracted element from the queue. Therefore, one master elements could be extracted in a such order that violates its data consistency, e.g a *Read After Write* or a *Write After Read*. In order to prevent this scenario, we should check if the element we want to insert at the end of queue is dependent on another element already queued. If there is a dependency, the new element is not inserted until the dependency is removed. Algorithm 7 describes this flow control.

Algorithm 6 Priority then Round-Robin scheduling with bandwidth limiter

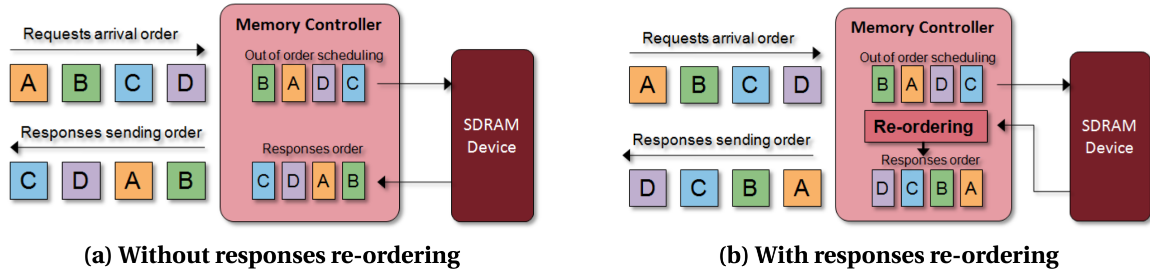
```
local variables pr, inIndex, selectedInput;
int inIndex ← 0;
for (pr = maxPriorityLevel; pr ≥ minPriorityLevel; pr − −) do
    samePriorityInputNumber ← 0
    for (in = 0; in < inputsNumber; in + +) do
        if (inputPriorityMatrix[pr][in] = 1) then
            samePriorityInputNumber + +;
            selectedInput ← in;
        end if
    end for
    if (samePriorityInputNumber = 1 ) then
        break;
    else if (samePriorityInputNumber > 1) then
        //round robin arbitration
        for (in = 0; in < inputsNumber; in + +) do
            inIndex ← (lastSelectedInput + 1 + in);
            if ( inIndex > inputsNumber) then
                inIndex ← (inIndex − inputsNumber);
            end if
            if (inputPriorityMatrix[pr][inIndex] = 1) then
                if (inputValid[inIndex] = true  and  inputBWOverflow[inIndex] = false) then
                    selectedInput ← inIndex;
                    break;
                end if
            end if
        end for
        break;
        //we have to stop the priority loop because we are sure that samePriorityInputNumber ≠ 0
    end if
end for
lastSelectedInput ← selectedInput;
updateInputBWOverflow(selectedInput);
return  selectedInput;
```

Algorithm 7 Data consistency flow control

```
local variables existingDirection, newDirection, existingTrID, newTrID, L;
bool canSend;
canSend ← true;
L ← genericQueue[queueIndex].getLength()
newOpcode ← newRequest.getDirection();
newTrID ← newRequest.getTrID();
if (L = 0) then
    canSend ← true;
else if (genericQueue[queueIndex].getCanStore()) then
    for (i = 0; i < L; i + +) do
        existingTrID ← GenericQueue[queueIndex].getRequest(i).getTrID();
        existingOpcode ← GenericQueue[queueIndex].getRequest(i).getDirection();
        if (newDirection = existingDirection and newTrID = existingTrID) then
            canSend ← false;
        end if
    end for
end if
return  canSend;
```

3.6.7 Re-ordering unit

Changing the order of the requests for efficient arbitration can be an issue for the masters that cannot re-order the responses of their requests. For this reason, some memory controller architectures include a *re-ordering* unit that puts the responses in the arrival order of their requests to the memory system. An example of response reordering is shown in Figure 3.9.



(a) Without responses re-ordering **(b) With responses re-ordering**

Figure 3.9: Ordering aspects in memory controllers

The re-ordering unit can be instantiated on each port of the memory controller to reorder the responses that must be sent back through this port. We propose a simple way to re-order the responses by using a FIFO to store a copy of the incoming requests. We tag each copy with a unique ID based on the source ID (*srcID*), the transaction (*trID*) and the port index (*portIDX*). Algorithm 8 explains how our mechanism works.

Algorithm 8 Re-ordering unit

local variables $L, inOrderID, portIndex, respToSendIndex$;
$L \leftarrow$ **responseQueue**$[portIndex].getLength()$;
$inOrderID \leftarrow$ **incomingTrOrder**$[portIndex].getRequestPointer(0).getUniqID()$;
for $(i = 0; i < L; i++)$ **do**
    if (**responseQueue**$[portIndex].getResponsePointer(i).getUniqID() = inOrderID$) **then**
        $respToSendIndex \leftarrow i$;
        incomingTrOrder$[portIndex].deleteRequest(i)$;
        $break$;
    end if
end for
return $respToSendIndex$;

3.6.8 Summary

Based on the analysis of many various architectures of memory controller, we presented in this section the building components we have designed to model any real life memory controller front-end. These components make it possible to build all architectures of industrial memory controllers we came across, and give the designer the possibility to explore them by tuning their parameters. The modelled front-end reflects all delay cycles that are due to the buffering elements number, buffers depth, number of arbiters and scheduling policies.

3.7 Examples of memory controller front-end

Based on the building components we have designed in this chapter, we now model three different architectures of industrial memory controllers. These memory controllers are well spread and used in multimedia platforms, where a mixture of dynamic traffic accesses the memory in an unpredictable way. As the back-end of these controller is SDRAM technology dependent, we will only show the front-end models.

3.7.1 Memory controller Alpha

It is a mono-port memory controller with an arbitration unit based on a randomly-readable queue. Hereafter the main features of this memory controller:

- Mono port interface with the interconnect system
- Separate read and write channels at the interface
- Mono port arbiter with randomly readable queue
- Multiple outstanding transactions
- Ageing mechanism to increase the priority of the oldest requests in the queue arbiter
- Out of order response sending
- Read After Read and Write After Write hazard detection
- QoS Mechanism based on the transaction identification, no QoS for Writes
- Scheduling policy:
    1) Minimum latency timeouts
    2) Priority
    3) Open-row hits in the same direction
    4) Open-row hits in the opposite direction

Figure 3.10 depicts the front-end modelled architecture of this memory controller.

3.7.2 Memory controller Beta

This memory controller has a multi-port interface, each port has its own priority level. Below the main features of this memory controller:

- Multi-port interface with the interconnect system
- Each port has a priority level. All requests coming through this port have the same priority
- Separated read and write channels in each port
- Read/Write arbiter in each port (read requests have higher priority than write requests)
- Multiple outstanding transactions
- Early response sending. It sends the write response back to the master when the write request is scheduled. It does not wait until the end of the write operation to send the response back

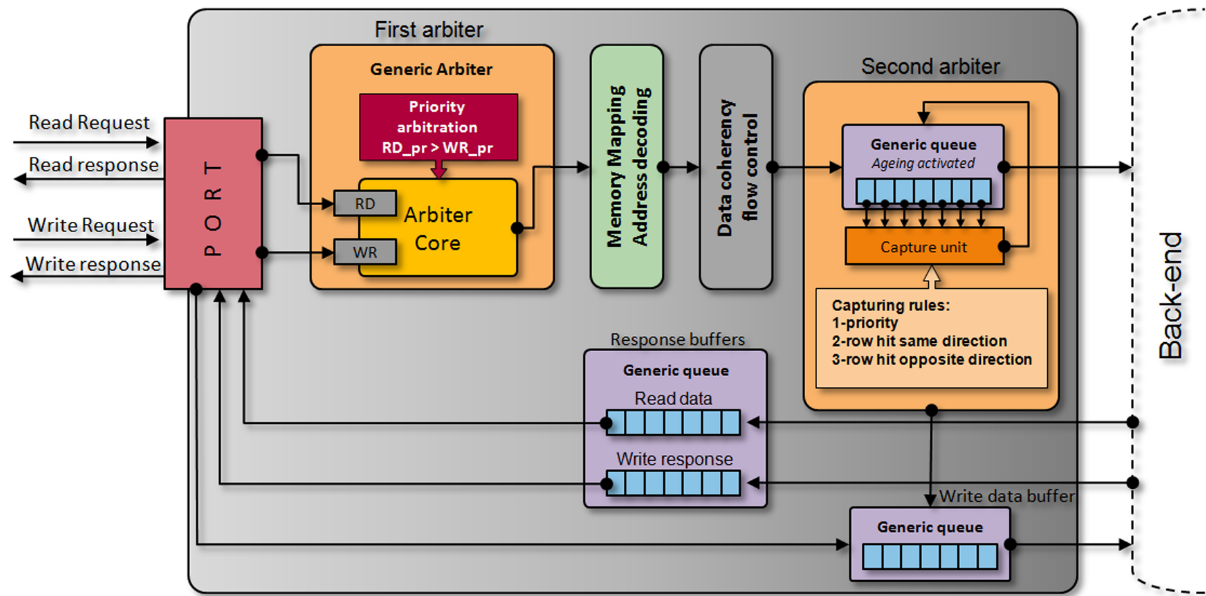


Figure 3.10: Front-end model for the memory controller Alpha

- Ageing mechanism to increase the priority of the oldest requests in the placement unit queue
- Out of order response sending
- QoS Mechanism based on the port priority and the memory bus efficiency
- Two arbitration stages:
  1) Port arbiter based on the priority of each port
  2) Placement logic unit, with commands ageing mechanism
- Placement policy:
  1) Address collision / data consistency
  2) Source ID collision
  3) Priority
  4) Read/write grouping

Figure 3.11 depicts the front-end modelled architecture of this memory controller.

3.7.3 Memory controller Gamma

Hereafter the main features of this memory controller:

- Two-port interface with the interconnect system: high priority port, and low priority port
- The low-priority requests are stored inside queues according to the memory bank they are addressed to. There are 4 queues per channel if the memory device contains 4 banks (a total of 8 queues for the read and write channel together)
- Multiple outstanding transactions

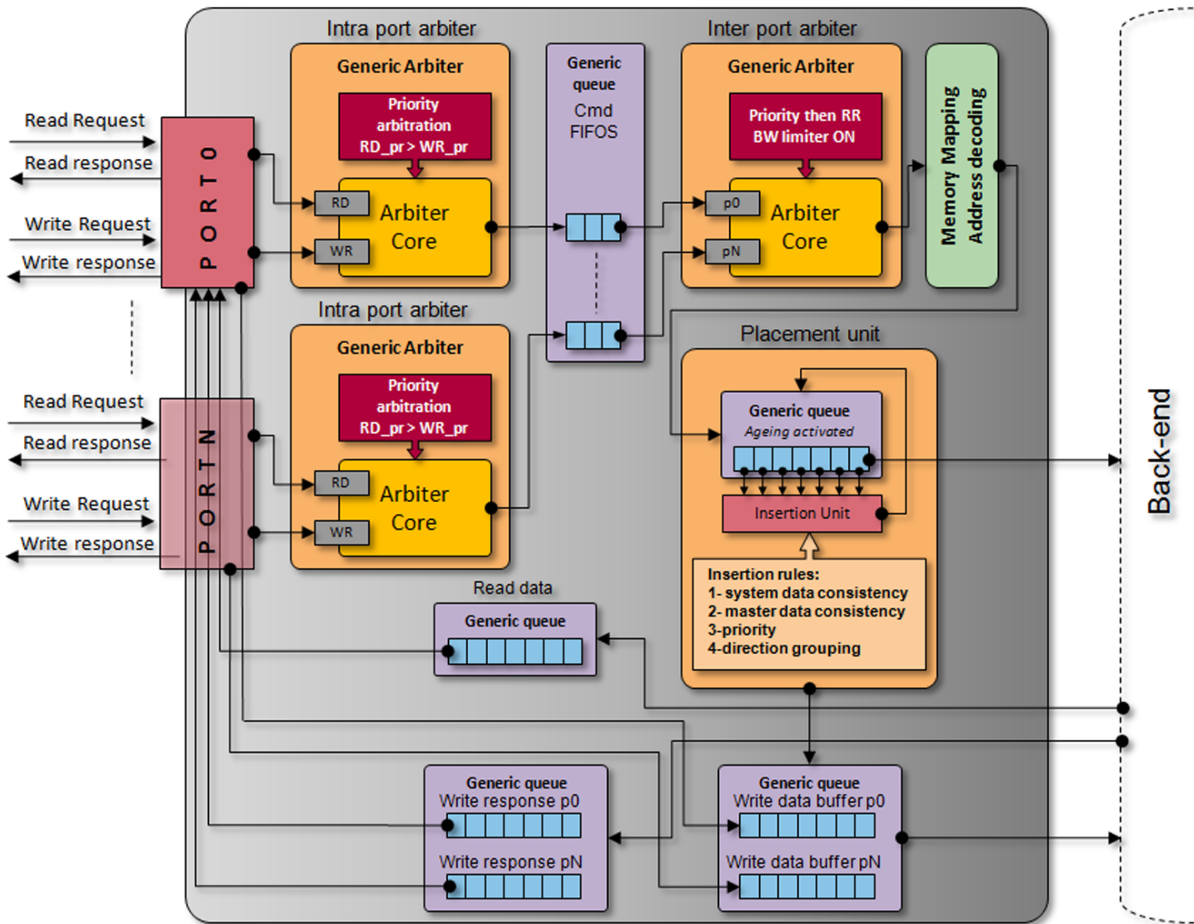


Figure 3.11: Front-end model for the memory controller Beta

- In order response sending through a re-ordering unit only for low-priority masters
- Two arbitration stages:
  1) The first arbiter is for low-priority read and write channels. It selects requests according to a Least Recently Used policy between banks, then row hit inside the selected bank, then direction grouping
  2) The second arbiter uses a priority based policy for high and low priority arbitration

Figure 3.12 shows the front-end modelled architecture of this memory controller.

3.7.4 Summary

Thanks to the building components we previously provided, the designer of the memory system can rapidly build a high-level model for a specified architecture. This architecture can easily be reconfigured through the parametrization of the building components and their interfacing with each other. This enables the memory controller design exploration.

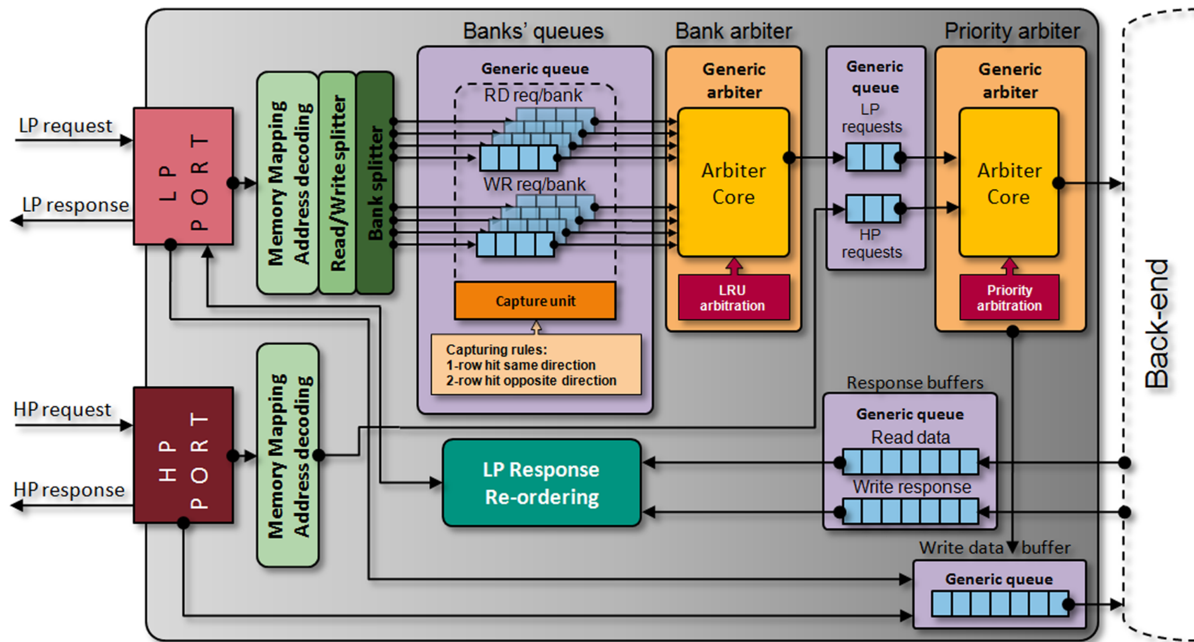


Figure 3.12: Front-end model for the memory controller Gamma

All building components, especially the generic arbiter, have been designed so as to be easily updated and developed. For instance, if the designer needs more specific arbitration algorithms, he can easily add it to the scheduling algorithms library.

3.8 Back-end building components

A cycle approximate model for the memory controller front-end is not sufficient to simulate accurately the memory access latency. Indeed, the masters requests pass through the back-end before accessing the SDRAM devices. Therefore, a cycle approximate model for a memory controller back-end is required to be able to simulate all delays that the requests spend inside the memory system.

Our back-end model is compatible with the DDR3 SDRAM, which is the latest generation of DDR$_n$ SDRAMs, and it is based on the JEDEC Standard [77]. It meets all DDR3 SDRAM parameters that come into the picture in a normal operating mode, and integrates all the states of the bank's FSM we have earlier shown in Figure 3.2 on page 39. According to our knowledge, no high-level and cycle accurate model for memory controller back-end has been published. We choose a simple and efficient architecture that includes a *commands generator*, a *memory manager*, and a *data handler* (see Figure 3.13). This back-end can easily be adapted to support earlier generations of DDR$_n$ SDRAM. The adaptation will be done by matching the bank's finite state machine and the memory timing values.

After the front-end arbiters have chosen the request to serve, the *command generator* con-

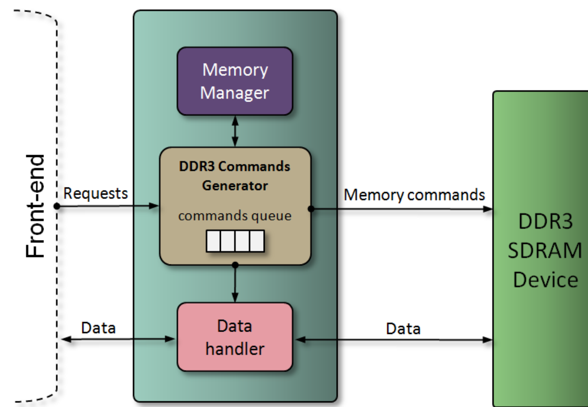


Figure 3.13: Back-end architecture

verts the request into one or more memory commands. The *memory manager* guarantees the proper behaviour of the memory and carries out several tasks such as initialization and refreshing. The role of the *data handler* is to ensure the data sending and receiving between the back-end and the memory on the rising and falling edges of the clock.

3.8.1 DDR3 SDRAM commands generator

As we previously mentioned, the memory commands format is different from the front-end format. Consequently, the back-end must format the requests coming from the front-end and store them in a *commands queue*. For data consistency reasons, the order of the read and write commands is definitive, no re-ordering process is allowed in this back-end. However, the commands generator can send bank preparation commands (*Prechage* and *Activate*) in a different order to hide a part of the row misses delay. All memory commands are sent to the memory while respecting all DDR3 SDRAM timings that we have earlier shown in Table 3.2 on page 40.

Several industrial back-end use the bank interleaving technique to hide the bank preparation delays. However, no sophisticated bank preparation algorithm has been published. We show our advanced algorithm for the anticipation of the bank preparation commands in Algorithm 9. This algorithm intelligently determines the bank to prepare at each clock cycle if needed. It increases the memory device efficiency by hiding the wasted clock cycles to prepare a bank. So the commands generator will issue either a command from the *commands queue* or a bank preparation command to the memory device while respecting the timing constraints.

3.8.2 Memory manager

The memory manager keeps track of the operation of the memory device and guarantees its proper behaviour. It carries out several tasks such as initialization, refreshing, and powering

Algorithm 9 Anticipation of the bank preparation commands

global variables **isThisBankRequested**[ ], **isThisBankBusy**[ ], **openRowInBank**[ ]; //we store the open row index for each bank
local variable $bankToPrepare \leftarrow (-1)$;
local variables $bank_0, row_0$; //bank and row for the first command in the queue
local variables $bank_n, row_n$; //bank and row for the nth command in the queue
$bank_0 \leftarrow queueOfCommands.getElementPointer().getBank()$;
$row_0 \leftarrow queueOfCommands.getElementPointer().getRow()$;
if (**isThisBankBusy**$[bank_0] = false$) **then**
  $send(queueOfCommands.getElement())$;
  isThisBankBusy$[bank_0] \leftarrow true$; //the FSM updates this tab when the operation is finished
else
  $bankToPrepare \leftarrow bank_0$;
end if
if ($bankToPrepare < 0$) **then**
  for ($i = 0; i < lookAheadWindow; i++$) **do**
    $bank_n \leftarrow queueOfCommands.getNthElementPointer(i).getBank()$;
    $row_n \leftarrow queueOfCommands.getNthElementPointer(i).getRow()$;
    if ($row_n \neq$ **openRowInBank**$[bank_n]$ **and isThisBankRequested**$[bank_n] = false$) **then**
      $bankToPrepare \leftarrow bank_n$;
      isThisBankRequested$[bank_n] \leftarrow true$;
      $break$;
    end if
  end for
end if
return $bankToPrepare$;

down. The only task that impacts the efficiency of the memory during a normal operating is the *refreshing* task. For this reason, we omit the other tasks of the memory manager in our model.

DDR3 SDRAM requires refresh cycles at an average periodic interval of *tREFI* to retain the stored data. This operation lasts for *tRFC* clock cycles. When a refresh command should be sent to the memory, the memory manager sends a signal to the commands generator to create and send a *refresh* command to the memory. The *refresh* command can be sent either for one bank or for all banks.

3.8.3 Data handler

DDR3 SDRAM standard requires a Write Latency delay (tWL) between the moment when a write command is sent and the moment when the write data are sent. Our back-end model simulates this delay and sends write data twice each clock cycle. The Read Latency delay (tRL) represents the delay in the memory device between the reception of a read command and the issuing of the read data. This delay is modelled in our DDR3 SDRAM which is the subject of the next section.

3.9 DDR3 SDRAM model

The real and simple function of this model is to simulate accurately the memory access latency in case of read operation. Indeed, the write operation latency is already computed by the back-end as it respects the memory timing constraint when it sends a write command followed by the write data after *tWL* clock cycles.

When the memory receives a read command, it will return the data after *tRL* clock cycles. The sent data is burst oriented as we mentioned earlier. The possible burst lengths are 4 and 8 transfers, with a rate of 2 transfers per clock cycle.

3.10 Conclusion

In this chapter, we introduced our design of a totally customizable memory controller based-on fully configurable building components. This design is a high-level abstraction and cycle approximate model, it can accurately simulate the memory access delays during a normal operating regime.

Our components library covers both parts of the memory controller, i.e. the front-end and the back-end. The front-end building components are easy to interface with each other, which gives the designer of the memory system a high degree of freedom in designing and exploring the memory controller architecture. The back-end is DDR3 SDRAM technology compatible, and respects all DDR3 SDRAM timing constraints.

As the memory system performance became a key factor in the design of modern systems-on-chip, modifying and exploring the shared memory system architecture became vital to determine the best configuration according to the whole system requirements. Our totally customizable memory controller meets the needs of the designers and offers the necessary configurable components to build and develop high-level and cycle approximate model for memory controller.

CHAPTER 4

Extreme End to End Flow Control Protocol for SDRAM Access

Contents

4.1 Introduction 65

4.2 Credit-based flow control 66

4.2.1 Analytical model for the *end-to-end* credit-based flow control 67

4.3 End to end flow controls 68

4.4 Pressure on the memory system in modern MPSoCs 69

4.5 Guaranteed service traffic in the memory controller 71

4.6 Saturation risk of the requests queue 71

4.6.1 Problem description 71

4.6.2 Possible solutions 72

4.7 EEEP: Extreme End-to-End Protocol 73

4.7.1 Novel system approach 73

4.7.2 EEEP principle 73

4.7.3 EEEP mechanism 75

4.7.4 *Requests queue* sizing method 75

4.7.5 EEEP guarantees and limitation 77

4.7.6 System modifications to support EEEP 78

4.8 Conclusion 79

MPSoC platforms face an increasing diversity of traffic requirements due to the number of applications run by the user, which leads to the coexistence of the *best effort* traffic and the *guaranteed service* traffic in the same platform. In this chapter, we propose an *Extreme End-to-End Protocol* (EEEP) as a new end-to-end flow control protocol to access SDRAMs through a multi-port memory controller in NoC-based MPSoCs. Our protocol considers the memory access with a *system approach*. It smartly exploits the occupancy rate of the *requests queue* in the memory controller within the policy of the traffic injection at the master network interfaces level. By controlling the *best-effort* traffic injection, EEEP guarantees the bandwidth and the latency of the *guaranteed service* traffic while improving them.

4.1 Introduction

Although the bandwidth problem has been solved inside MPSoC with the introduction of Networks on Chip (NoCs), it has indeed led to a growing pressure on off-chip SDRAM accesses that must provide higher bandwidth while keeping latencies low. An additional burden is that current MPSoCs concurrently execute different applications coming from different application classes. This imposes new challenges to the NoC, as it must accommodate applications traffics with very different characteristics and requirements [23].

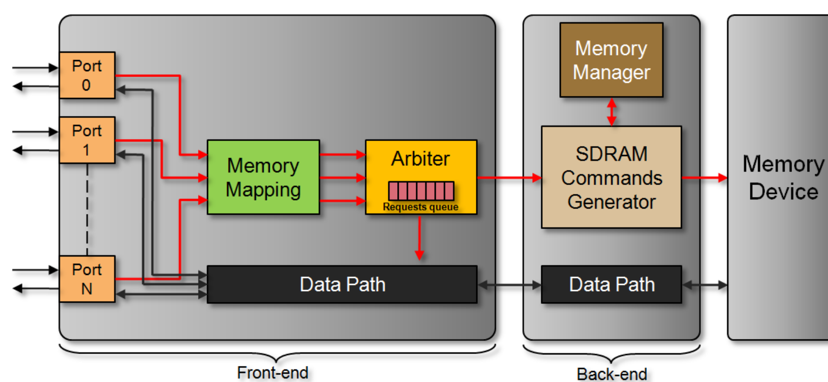


Figure 4.1: The front-end and the back-end of a memory controller

The memory controller is the interface between the interconnect and the memory module. Figure 4.1 depicts a general architecture of a memory controller as introduced earlier. The *front-end* is aware of the supported QoS in the interconnect, it schedules the requests in a way to satisfy the masters requirements. The *back-end* is memory-technology-dependent, it deals with the memory device and converts the interconnect requests into memory commands.

In this chapter, we focus on the memory controller front-end to exploit the occupancy of its *requests queue* in the implementation of our end-to-end flow control protocol. In section

(4.2), we provide on overview of the credit-based flow control with its two approaches, link level and end-to-end. In section (4.4), we show the high pressure problem on memory sub-system in today's MPSoCs, and then we explain in section (4.6) the saturation problem of the requests queue in the memory controller front-end. Section (4.7.2) introduces our extreme end-to-end protocol as a solution to this problem. We present then a method for the sizing of the requests queue, and we finally show the minor modifications in the system to support our protocol.

4.2 Credit-based flow control

With the credit-based flow control, the sender keeps a count of the number of free queue slots in the receiver [20]. This mechanism can be used either at the physical link level in a NoC or with an end-to-end approach between two network interfaces.

The *link level* credit-basic flow control can be used between a network interface (NI) and a router, or between two routers. The sender NI (or router) does not ever send more flits than a receiver router (or NI) can receive in its input queue. Figure 4.2 shows an example of the link level credit based flow control. The sender block maintains a detailed knowledge of the number of queue slots that the receiver block still has available through the exchange of credits. The sender block keeps track of the storage capacity in the receiver block with a credit counter that is initialized with a value equal to the size of the corresponding queue, and it is dynamically updated to track the number of available slots in the queue. Hence, the sender block continuously transmits only a subset of the message packets that is guaranteed to arrive inside the receiver block.

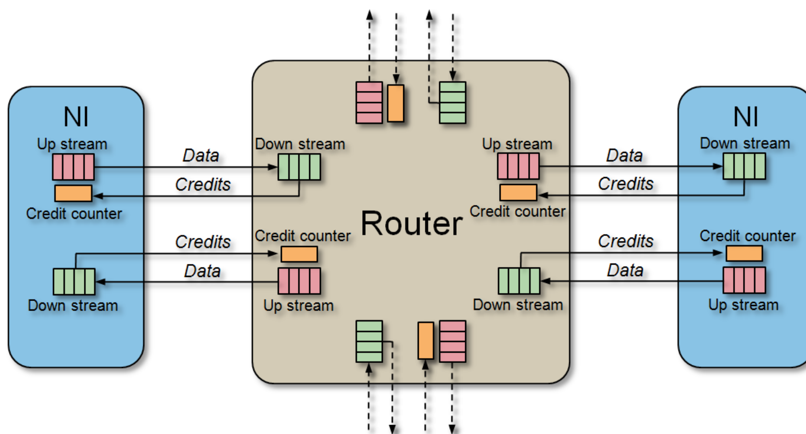


Figure 4.2: *Link level* credit-based flow control

Differently from the credit-based flow control that operates at the link level between a pair of interconnected routers, the *end-to-end* credit-based protocol operates between two NIs separated by multiple hops in the network. Figure 4.3 shows an example of NI architec-

ture that supports the end-to-end credit-based flow control. With the end-to-end approach, the sender NI also maintains a detailed knowledge of the number of queue slots that the receiver NI still has available through the exchange of the *end-to-end* credits. A credit can be associated to either a packet or to a packet flit depending on the desired level of granularity. The sender NI keeps track of the storage capacity in the receiver NI with a credit counter that is initialized with a value equal to the size of the corresponding queue, and it is dynamically updated to track the number of available packet slots in the queue. This protocol guarantees that no fragment of a message can be blocked in the network due to the lack of space in the receiver NI input queues. The receiver NI sends a credit back to the sender network interface upon the generation of an empty slot in an input queue. Note that an end-to-end credit-based flow control should be used in addition to a basic link-level flow control.

Note that for a given system, a NI that may send messages to N different NIs needs N credit counters while if it can receive messages from M different NIs it needs M different queues. This has a negative impact on the NI area overhead.

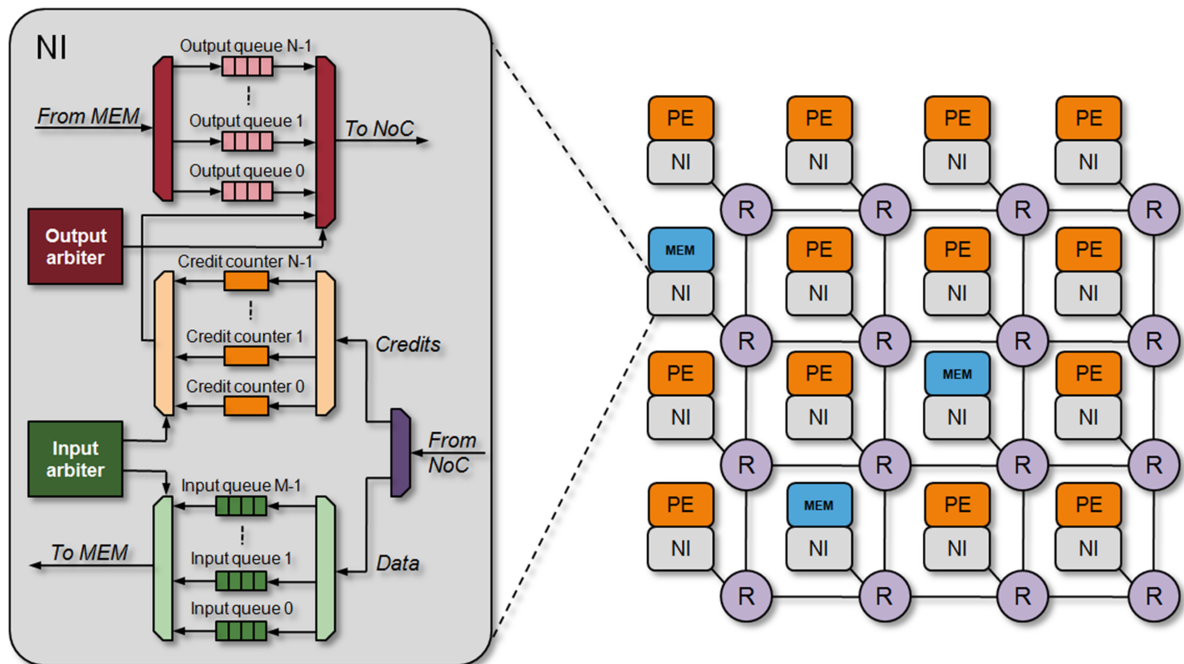


Figure 4.3: Simplified NI architecture that supports the *end-to-end* credit-based flow control

4.2.1 Analytical model for the *end-to-end* credit-based flow control

The sizing of network interface queues is one of the important issues when we use the *end-to-end* credit-based flow control. Here, we will present a method to size the input queues in a network interface that supports the end-to-en credit-based flow control.

To minimize the duration of a transaction between NIs, a sender NI_s should be able to

generate a continuous flow of data flits, and the receiver NI_r should be able to absorb and process this flow. Hence, it is important to guarantee that the source NI_s never runs out of end-to-end credits. This performance requirement can be satisfied by properly sizing the input data queues of the the receiver NI_r. This requires to account for the round trip time between a receiver NI_r and all its possible peer sender network interfaces. The zero-load latency θ measured in clock cycles that is taken by a flit to traverse the NoC from a given source NI_s to reach a given receiver NI_r is equal to

$$\theta(NI_s, NI_r) = \Delta(NI_s, NI_r).R + \sigma(NI_s, NI_r) \quad (4.1)$$

Where $\Delta(NI_s, NI_r)$ is the distance in hops between the two NIs, R is the number of clock cycles used by routers to process and forward a flit, $\sigma(NI_s, NI_r)$ captures the aggregate number of wire pipeline stages across all links on the path between NI_s and NI_r. The zero-load *round-trip* time taken by a flit to traverse the NoC from NI_s to NI_r and back is given by

$$T_{rt}(NI_s, NI_r) = \theta(NI_s, NI_r) + \varepsilon_r + \theta(NI_r, NI_s) \quad (4.2)$$

Where ε_r is the delay in clock cycles between the moment when the NI_r queues an arriving flit, and the moment when the NI_r forwards it to its final destination, freeing therefore a queue slot.

Putting all together, the input queue Q_{in_s} of a given network interface NI_r, receiving flits from NI_s, should be sized according to the equation

$$Q_{in_s} = C + T_{rt}(NI_s, NI_r) \quad (4.3)$$

Where C is the number of credits carried by each credit packet.

Note that this queue size can ensure continuous transfer in case of zero-load network. When the network becomes loaded, the transfer between two NIs may be discontinuous.

4.3 End to end flow controls

The **ÆTHEREAL [28]** and **FAUST [25]** NoCs use *credit-based end-to-end flow control protocols.* The mechanism is similar to the one detailed in section (4.2).

The *Connection Then Credit* (CTC) flow control protocol proposes a micro-architecture of the network interface (NI) that decreases the number of credit counters [15]. It uses a single credit counter together with an output queue to send all the possible outgoing messages, and a single pair of data-request queues that is shared across all possible incoming messages. However, CTC requires the completion of a handshake procedure between any pair of cores that want to communicate before the actual message transfer starts. This procedure increases the total latency of the transactions.

Radulescu *et al.* [66] present an end-to-end flow control for guaranteed service in addition to the basic link level flow control. **Jafari *et al.* [39]** propose a flow regulation to reduce the delay and backlog bounds in SoCs. A prediction-based flow control is presented by **Ogras and Marculescu [60]**, it predicts the cases of possible congestion in the network, and controls the packet injection rate at the sources of the traffic in order to regulate the total number of packets in the NoC.

As the network-on-chip and the memory controller become correlated with each other in most SoCs, several researchers propose micro architectures and methods to optimize the performance of the memory subsystem, and thus the system performance. **Paganini *et al.* [63]** develop a decentralized control system, where the sources adjust their traffic generation rates based on the feedback received from the bottleneck links.

All previous flow controls we have seen until here consider only the state of the interconnect. The range of these flow controls is limited at the network interfaces level, without tackling at all the state of the targets. Furthermore, none of them can ensure the continuity of the services between the network and the main memory system.

4.4 Pressure on the memory system in modern MPSoCs

Today's memory systems suffer from a very high pressure because of several factors. The multi-threading technique used in MPSoCs increases the contention on the main memory and demands memory systems with more complex architecture and higher performance. The SDRAM system is often unique in the system-on-chip because of cost reasons. Even if several memory systems exist in the same chip, the ratio between the number of cores and the number of memory systems is at least 10. Figure 4.4 shows an overview of the multi-core chip TILE*pro*64 [78], which has 4 memory controllers. The frequency gap between the CPU

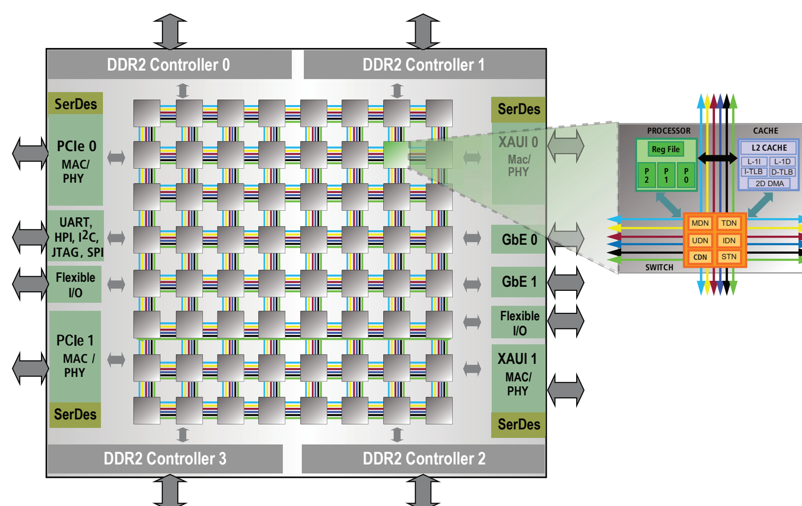


Figure 4.4: Overview of TILEpro64 that includes 4 memory controllers, source [78]

and the memory is still increasing. The classical CPU-DDR SDRAM case shows that the frequency gap between CPU and main memory eventually offsets most performance gains from further improvements on the CPU speed. For instance, a cache miss is equivalent to hundreds cycles for today's CPUs, a time long enough for the processor to execute hundreds of instructions. While the DDR SDRAM IO frequency has been improving by 37% per year since 2001, the CAS[8] Latency of SDRAM that fundamentally determines its overall performance has been only improving by 5% per year [75; 76; 77].

Another factor that emphasizes the pressure on the memory is the number of cores in future systems-on-chip. If we have a look at the graph in Figure 4.5, we will see that the number of processing engines in the SoC consumer portable designs is going to reach 1000 processing engines in 2019. The increasing number of processing engines in SoCs increases the number of applications run by the user in parallel, which consequently leads to the co-existence of several classes [8; 70]. *This co-existence of several classes of traffic in the same memory system complicates the task of the memory controller, especially that their memory access patterns cannot be known in a predictable way, and they must be dynamically computed in the memory controller.*

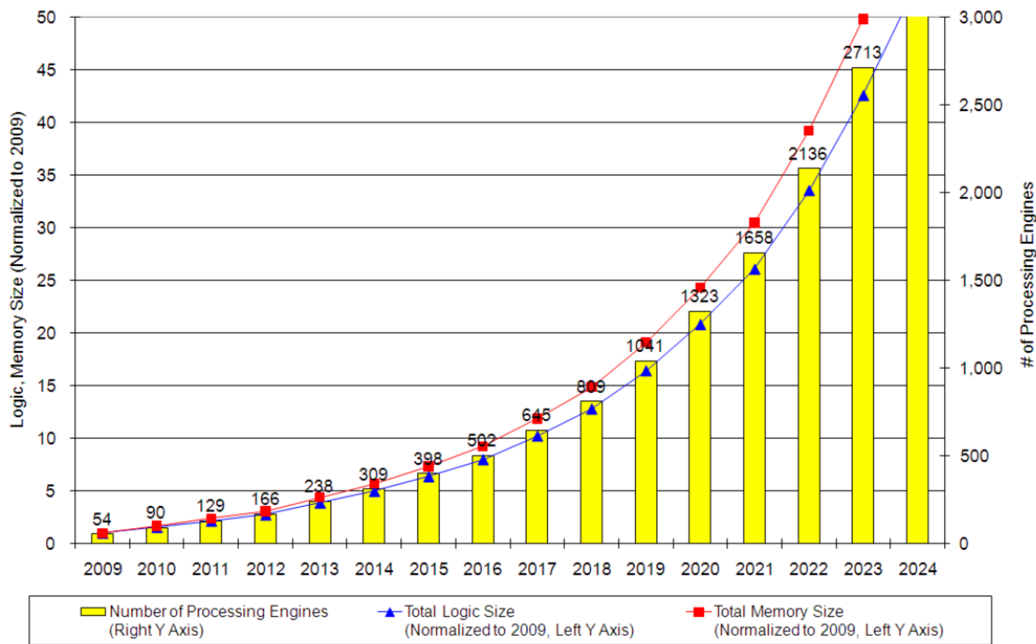


Figure 4.5: SoC consumer portable design complexity trends, source ITRS [37]

[8]Column Access Strobe

4.5 Guaranteed service traffic in the memory controller

In this framework, we mean by *guaranteed service* (GS) traffic, the traffic that has requirements in terms of bandwidth and latency. This traffic can be subdivided in two separate categories: *latency-sensitive* traffic such as CPU traffic, and *bandwidth-sensitive* traffic such as display controller traffic. The bandwidth-sensitive traffic can also have requirements in term of latency jitter, so the latency variation of its transactions must remain controlled. We define the *best effort* (BE) traffic as all other traffic that does not have severe requirements, and can be always served after the *guaranteed service* traffic.

We assume that the network-on-chip provides the necessary services to meet the requirements of the GS traffic. As for the memory controller, most designs provide several levels of priority to distinguish the traffic classes. So the highest priority should be attributed to the *guaranteed service* traffic. Quite the contrary, the BE traffic has the lowest priority. Several levels of priority can be devoted to the GS traffic. In order to avoid starvation situations, additional mechanism can be implemented. A Round Robin intra-scheduling policy can be used to arbitrate the requests in case of conflict between requests that have the same priority level. The *ageing* mechanism is also a solution to avoid starvation problems. It consists in increasing the priority level of the low-priority requests every *maxAge* clock cycles. So that these requests do not stall for very long time behind the high-priority requests.

In order to be able to provide services to the GS traffic in the memory controller, the requests should obviously be buffered in the *requests queue*. One additional guarantee that the memory controller must provide to the GS traffic *in absence of BE traffic* is the appropriate number of available slots in the requests queue. This number of slots should be determined according to the number of outstanding requests of the GS traffic. When a new BE traffic is accepted in the memory controller, we should guarantee that this traffic has the lowest priority level. However, the number of available slots in the requests queue for GS traffic can no longer be guaranteed in presence of BE traffic.

4.6 Saturation risk of the requests queue

Shared resources pose a significant resource management problem in designing MPSoCs. SDRAM is always accessed through a memory controller front-end, which has in most architectures a large queue to buffer the memory requests. The storage of the requests is necessary for an efficient arbitration in order to increase the memory subsystem performance and respect the *quality-of-service* (QoS) requirements of each master.

4.6.1 Problem description

Running several applications in parallel involves the generation of several categories of traffic, which can interfere with each other while accessing the shared resources. A typical technique

to control the traffic is to give an appropriate QoS to each class of traffic according to the applications requirements. In this framework, we focus on two classes of traffic: *guaranteed service* traffic (GS) and *best effort* traffic (BE). During several timing windows, the network grants the BE traffic, and allows it to access the shared resource. If the shared resource is an SDRAM, the BE traffic can use the whole *requests queue* in the memory controller as long as none of the GS traffic accesses the SDRAM. As the SDRAM commands execution latency is considerable, the saturation of the requests queue can subsequently prevent the GS traffic from being received by the memory controller during dozens or even hundreds of clock cycles. Therefore, the bandwidth and the latency of the GS traffic that accesses the SDRAM will be dramatically impacted. Figure 4.6 describes the case where the *best effort* can saturate the requests queue in a memory controller.

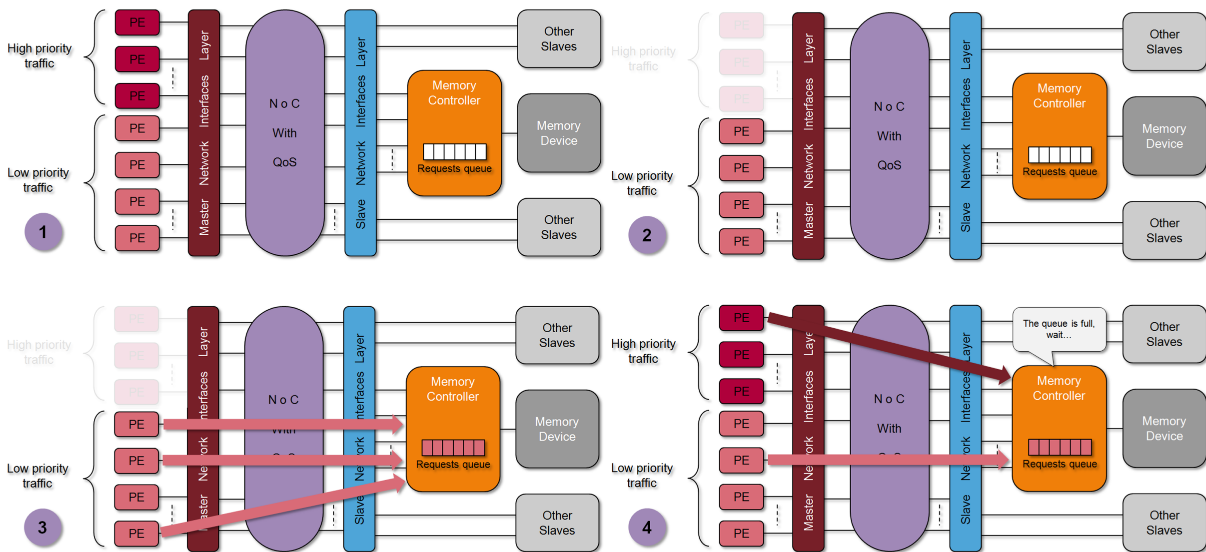


Figure 4.6: Saturation risk of the requests queue in a memory controller

4.6.2 Possible solutions

Several solutions to this problem do exist. However, they cannot all respect the VLSI[9] constraints.

One of the obvious solutions to this problem is the extra-sizing of the *requests queue* in order to be able to buffer every time all requests whatever their origin. In other words, this solution considers the *worst case* as the normal operating regime. However, this solution is not efficient in term of silicon area overhead as the worst case may be close for infinity.

Separating the BE traffic queue from the GS traffic queue could also be a solution. Nevertheless, the dynamic sharing between the queues is not possible in this case, leading to a waste of hardware resources.

[9]Very Large Scale Integration

Our innovative solution consists in using one queue for requests and dynamically share it between the BE traffic and the GS traffic. This dynamic sharing will be insured by applying a new BE traffic injection policy in the NoC, which is based on the use of information about the memory controller state.

4.7 EEEP: Extreme End-to-End Protocol

We build on the credit-based approach to develop our *Extreme End-to-End Protocol* (EEEP). EEEP regulates the packets injection rate of BE traffic in the network when the packets are sent to the memory controller.

4.7.1 Novel system approach

Our approach is different from the previously mentioned work in a number of ways:

First, we consider at once the NoC and the memory controller within a *system approach*: from the master network interfaces to the last *requests queue* in the memory controller front-end. We exploit the occupancy rate of this queue within the policy of the traffic injection at the master network interfaces level. We prevent the BE traffic that addresses the memory system from being injected in the network if there are no available slots for it in the requests queue. Thus, we ensure that the BE traffic will not occupy more slots than necessary to obtain the required average bandwidth, and therefore, we guarantee that the GS traffic is always received by the memory controller.

Second, unlike the end-to-end credit-based protocol, EEEP needs neither additional queues nor counters to be implemented in the slave network interfaces (see Figure 4.3 for more details about the number of queues and counters in a NI that supports an end-to-end credit based). Indeed, all the slave network interfaces which are connected to the multi-port memory controller interface target the requests queue. This convergence of paths of requests allows us to cross the network boundary, and move the credits management from the slave network interfaces to the memory controller. Consequently, only a few modifications in the slave network interfaces are required to support EEEP.

Figure 4.7 shows a comparison between the extend of EEEP and other end-to-end flow controls.

4.7.2 EEEP principle

The buffering of the commands in the *requests queue* in the memory controller front-end is inevitable for efficient arbitration. As we mentioned earlier, EEEP uses the occupancy rate of the requests queue within the SDRAM access policy to modify the traffic injection policy of the masters that generate *best effort* traffic (BE).

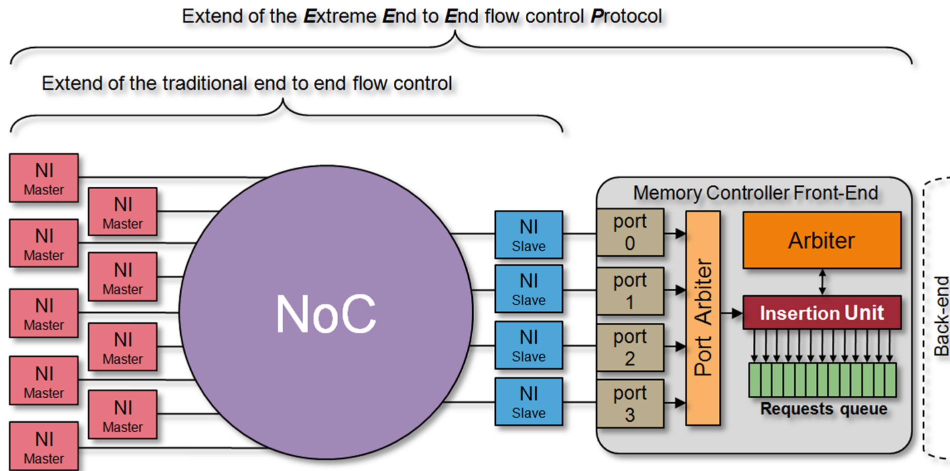


Figure 4.7: EEEP extend versus other flow controls extend

The idea behind EEEP is the dynamic sharing of the requests queue between the BE and GS traffic. Let us suppose that the GS traffic needs N slots, and the BE traffic uses the remaining M slots (the queue depth is $N + M$). Whatever the load of the memory controller, EEEP ensures that the GS traffic can always use *at least* M slots of the requests queue, and the BE traffic can use *at most* M slots.

EEEP is used in addition to a basic *link-level* flow control. Our novel protocol guarantees the availability of the necessary part of the requests queue in the memory controller front-end to the GS traffic by tuning the packets injection rate of the BE traffic. The NIs that inject BE traffic keep track of the remaining slots in the requests queue, and do inject any request in the network before verifying that this request will not use any slot reserved for the GS traffic.

For each NI that injects BE traffic, a maximum number of slots in the memory controller requests queue can be used. Even if this NI requires more slots during a period of time, EEEP blocks its traffic until the release of one slot (which had been allocated to it) in the requests queue. This mechanism is based on the end-to-end credits exchanging: the BE traffic NI consumes *EEEP credits* when it targets the shared memory, and the memory controller sends *EEEP Credits* back upon the release of one or more slots in the requests queue. Note that EEEP is mono-directional, it can only be used on the *request path* of the BE traffic between the NIs and the memory controller. Regarding the GS traffic, it only uses the link-level flow control.

To summarize, EEEP ensures that the BE traffic does not use more queue slots than those allocated to it and, consequently, guarantees the receiving of the packets of the GS traffic in the memory controller. Moreover, if the GS traffic requires extra queue slots for a period of time, it is allowed to use the available queue slots for the BE traffic. Figure 4.8 shows an overview of the EEEP implementation in a NoC-based system.

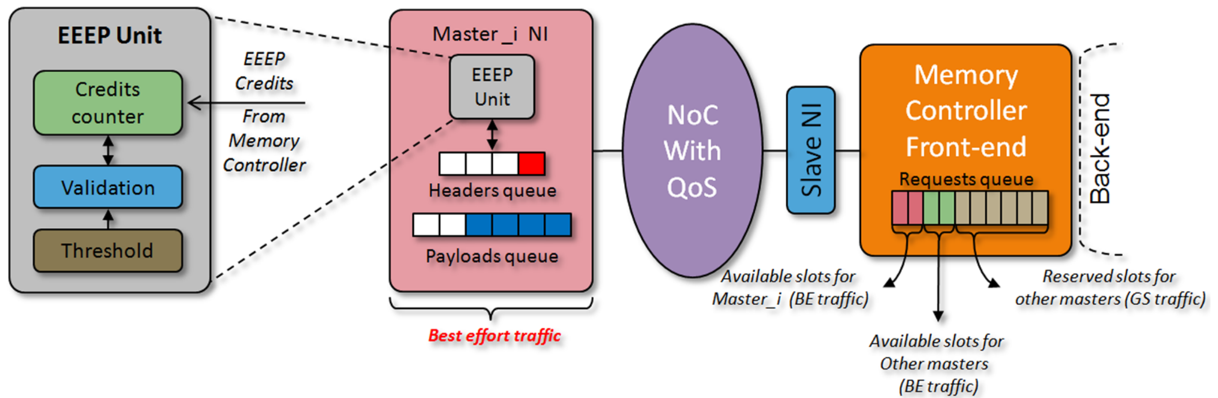


Figure 4.8: An overview of the EEEP implemented in a system

4.7.3 EEEP mechanism

We have to implement an *EEEP Unit* in each network interface (NI) that injects BE traffic. This unit will count the number of available slots in the *requests queue* in the memory controller. We have also to implement a *Credits Generation Unit* in the memory controller front-end to generate the EEEP credits and send them back to the concerned NIs.

When a NI that uses EEEP has to send a BE traffic packet to the SDRAM, it checks first the value of the *EPPP Credits* counter, and then the value of *EEEP Credits Threshold*. If the *EPPP Credits* are validated (credits>threshold), the master NI checks the link-level flow control before it starts sending the packet flits to the router. The *EEEP Credits* counter is decremented by one just after the packet header is sent. The utility of the threshold is to accumulate the credits and to consume them in a bursty way. This allows to regulate the shape of BE traffic. Figure 4.9 describes this mechanism in details for the BE request packets.

When the memory controller sends a response back through a slave NI, it sends an *EEEP Credit* which will be written in the response packet header. This credit is going to inform the master NI that it has one more available slot in the requests queue. As soon as the master NI receives the response packet from the memory controller, its *EEEP Credits* counter is incremented by one.

4.7.4 *Requests queue* sizing method

Unlike the end-to-end credit-based flow control, EEEP does not aim at ensuring continuous data flow for BE traffic. In contrary, it tries to slow down the requests of the BE traffic with the goal of guaranteeing the continuity and extending the services for GS traffic.

EEEP only controls the request path of the BE traffic towards the memory controller. When the memory controller returns responses back with EEEP credits, it uses the link level flow control that the network supports.

The sizing of the *requests queue* in the memory controller front-end is the most important

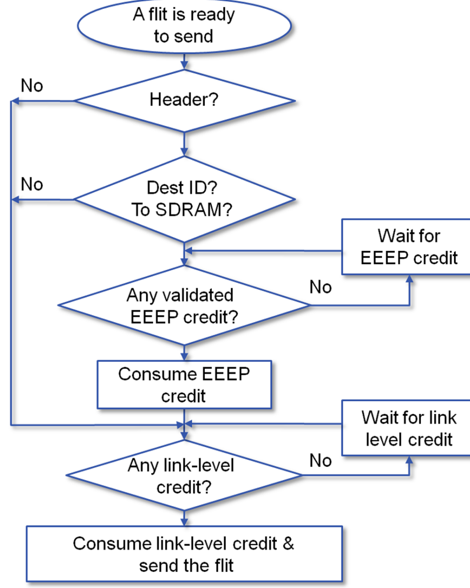


Figure 4.9: EEEP diagram for *best effort* traffic request packets

issue when we use the *extreme end-to-end protocol.* More precisely, the part of the requests queue that should contain BE traffic requests. Remember that the requests queue is shared between the GS traffic and the BE traffic, and we want to prevent the BE traffic requests from using more queue slots than necessary to maintain its *average* bandwidth.

Based on the method introduced in section (4.2.1), we will present our method to size *the part of the requests queue which can be used by BE traffic requests.* The zero-load latency θ measured in clock cycles that is taken by a flit to traverse the network from a given source of BE traffic NI_{S_n} to reach the memory controller network interface NI_{mc} is equal to

$$\theta(NI_{S_n}, NI_{mc}) = \Delta(NI_{S_n}, NI_{mc}).R + \sigma(NI_{S_n}, NI_{mc}) \tag{4.4}$$

Where $\Delta(NI_{S_n}, NI_{mc})$ is the distance in hops between the two NIs, R is the number of clock cycles used by router to process and forward a flit, $\sigma(NI_{S_n}, NI_{mc})$ captures the aggregate number of wire pipeline stage across all links on the path between NI_{S_n} and NI_{mc}. The zero-load round trip latency taken by a flit to traverse the network from NI_{S_n} to NI_{mc} and back is given by

$$T_{rt}(NI_{S_n}, NI_{mc}) = \theta(NI_{S_n}, NI_{mc}) + \theta(NI_{mc}, FE_{rq}) + \varepsilon_{mc} + \theta(FE_{rq}, NI_{mc}) + \theta(NI_{mc}, NI_{S_n}) \tag{4.5}$$

Where $\theta(NI_{mc}, FE_{rq})$ is the zero-load latency between the memory controller network interface and the front-end requests queue. The ε_{mc} parameter represents the delay between the moment when a request is placed and the moment when its response is sent back with credit. Since ε_{mc} is unpredictable because it depends on the dynamic access patterns to the memory,

we will fix it at tRL which corresponds to the optimal delay of a read operation.

We call DTI_n the Diverted best effort Traffic Indicator which is the average number of requests towards other targets in the system than the memory controller inside a given time window $(AVG_{otherReq_n})$, multiplied by the outstanding transactions number $(outStand_n)$ that this source can support, divided by the average number of requests (AVG_{Req_n}) in the same time window. So the diverted BE traffic indicator is given by

$$DTI_n = \lfloor outStand_n \cdot \frac{AVG_{otherReq_n}}{AVG_{Req_n}} \rfloor \tag{4.6}$$

As BE traffic does not need to be fully smooth, a degree of freedom is added. We call DC_{be_n} the flow discontinuity factor that we associate with the BE traffic flow coming from the NI_{S_n}.

Putting all together, the part of the requests queue where the BE traffic requests coming from *one* NI_{S_n} can be placed is given by

$$QS_n = C_n + T_{rt}(NI_{S_n}, NI_{mc}) - DTI_n - DC_{be_n} \tag{4.7}$$

Where Cn is the number of EEEP credits carried by the memory controller response packets.

Finally, the number of slots in the requests queue that can contain BE traffic requests for all BE NIs in a system is given by

$$QS_{be} = \sum_{n=1}^{N} C_n + T_{rt}(NI_{S_n}, NI_{mc}) - DTI_n - DC_{be_n} \tag{4.8}$$

Where N is the number of network interfaces in the system that inject BE traffic.

The number of queue slots determined by the equation (4.8) is a maximum limit. That means the BE traffic requests can use *at most* this number of queue slots. If the GS traffic requests need some extra queue slots during a time window, they can use the slots dedicated to the BE traffic requests.

To summarize, if we have N master network interfaces that inject BE traffic, we will need to implement N *EEEP Units* in these NIs, and only one EEEP *Credits Generation Unit* in the memory controller.

4.7.5 EEEP guarantees and limitation

Whatever the load of the memory controller and the ratio between the BE and GS requests, EEEP guarantees that the GS requests are always received and buffered in the *requests queue* without stalling in the memory controller NIs because of lack of space in the requests queue.

The services provided by the memory controller to a kind of GS traffic can only be effective when its requests are stored in the requests queue. Therefore, the continuity of services between the network and the memory subsystem can only be ensured by guaranteeing the *immediate* admission of the requests in the memory controller. This is how EEEP guarantees the *service continuity* for GS traffic between the master NIs and memory controller.

Note that we use the granularity of the packet for the EEEP, which corresponds to one memory request. The longer the burst length of a request is, the more the request uses the shared buffers in the memory controller. Indeed, a write request uses shared resources on the request path in the memory controller (*requests queue* and *write-data* buffers), while a read request only uses the *requests queue* on the *request path*, and the *read-data* buffer on the response path. This version of EEEP does not take into account the BE requests size in the credit allocation policy.

4.7.6 System modifications to support EEEP

A few modifications in the network-on-chip and the memory controller are required to support EEEP.

Master network interface

We add in the master network interface, which injects BE traffic, a counter to buffer the number of *EEEP Credits* that indicates the available slots for this master in the *requests queue* of the memory controller front-end. We must also add a register to buffer the *EEEP Credits Threshold* that is used to valid the *EEEP Credits*. If the *EEEP Credits* number is less than the *EEEP Credits Threshold*, the *EEEP Credits* cannot be used.

Memory controller

We add a *Credits Generation Unit* in the memory controller front-end. This unit creates one credit when a slot in the requests queue becomes vacant after the sending of a BE traffic request.

Slave network interface

We add a side signal between the memory controller interface and the slave network interface to transport the EEEP Credits. A simple mechanism will be required to add the EEEP credit(s) in the response packet header (see Figure 4.10).

NoC protocol

We must reserve a field in the response packet header in order to send the EPPP Credits back to the master network interface. This field is only one bit in most cases.

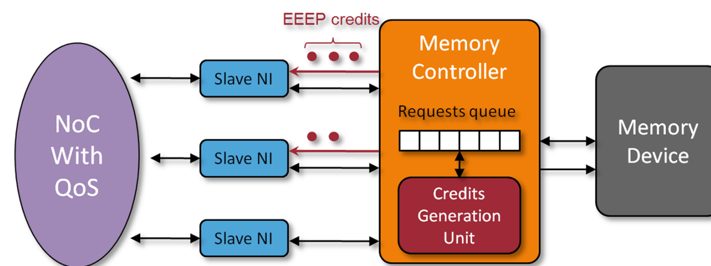


Figure 4.10: Memory controller modification to support EEEP

According to our estimation, the implementation of this protocol costs less than 4% of the total silicon area of a given memory controller, and less than 1% for the NIs that support EEEP.

4.8 Conclusion

We presented in this chapter an *Extreme End to End Protocol* to access the memory subsystem in MPSoCs through a multi-port memory controller. EEEP should be used for *best effort* (BE) traffic in addition to a link-level flow control. By controlling the injection of the BE traffic in the network, EEEP increases the performance of *guaranteed service* traffic in terms of bandwidth and latency, while maintaining the average bandwidth of the BE traffic. This flow control protocol handles the SDRAM access within a *system approach* by considering the memory controller state before injecting requests packets in the network. EEEP requires neither additional queues nor counters in the slave network interface, because it is based on the available slots in the *requests queue* in the memory controller front-end.

The novelty of this protocol consists of exploiting information coming from the memory controller within the quality of service in the network-on-chip. Unlike other end-to-end protocols, EEEP crosses the boundary of the network and guarantees the continuity of services from the master network interfaces to the memory devices.

CHAPTER 5

Implementation of the Customizable Memory Controller Architecture

Contents

5.1 Development environment **83**

5.2 NED language overview **84**

5.3 Model structure in OMNeT++ **85**

5.4 General description of a building component **86**

5.5 Memory controller building components parameters **87**

5.5.1 Memory mapping parameters 88

5.5.2 Generic queue parameters 88

5.5.3 Capture unit parameters 88

5.5.4 Insertion unit parameters 88

5.5.5 Generic arbiter parameters 89

5.5.6 Re-ordering unit parameters 89

5.6 EEEP components parameters **89**

5.7 Traffic generator **90**

5.8 Conclusion **92**

IN ORDER to evaluate architectures making use of our generic memory controller, we must provide virtual prototypes within a simulation environment. We thus rely on a high-level cycle approximate simulator to implement our totally customizable memory controller architecture. The development environment should allow the implementation through an object oriented language for development flexibility reasons. We are going to show how to implement our customizable architecture.

5.1 Development environment

We use OMNeT++ as development environment to implement our customizable memory controller. OMNeT++ is a discrete event simulation environment. Its primary application area is the simulation of communication networks, but because of its generic and flexible architecture, it is successfully used in other areas like the simulation of complex IT systems, queuing networks or hardware architectures as well [62]. We opt for OMNeT++ as a development environment for its performance and flexibility compared to other simulators from its category such as NS2 and OPNET [81]. A quick overview of the simulation with OMNeT++ is given below:

1) An OMNeT++ model is build from components (*modules*) which communicate by exchanging messages. Modules can be grouped together to form a compound module. When creating the model, we need to map the system into a hierarchy of communicating modules.

2) We define the model structure in the NED language. We can edit NED files in a text editor or in the graphical editor of the Eclipse-based OMNeT++ Simulation IDE.

3) The active components of the model (*simple modules*) have to be programmed in C++, using the simulation kernel and class library.

4) We provide a suitable file to hold OMNeT++ configuration and parameters for the model. A configuration file can describe several simulation runs with different parameters.

5) We build the simulation program and run it. We will link the code with the OMNeT++ simulation kernel and one of the user interfaces that OMNeT++ provides. There are command line (*batch*) and interactive-graphical user interfaces.

6) The simulation results are written into output vector and output scalar files. We can use the Analysis Tool in the Simulation IDE to visualize them. Result files are text-based, so we can also process them *Perl* or other ways.

5.2 NED language overview

The user describes the structure of a simulation model in the NED language. NED stands for NEtwork Description. NED lets the user declare simple modules, and connect and assemble them into compound modules. Channels are another component type, whose instances can also be used in compound modules. The NED language has several features which let it scale well to large projects:

Hierarchical: the traditional way to deal with complexity is by introducing hierarchies. In OMNeT++, any module which would be too complex as a single entity can be broken down into smaller modules, and used as a compound module.

Component-based: simple modules and compound modules are inherently reusable, which not only reduces code copying, but more importantly, allows component libraries to exist.

Interfaces: module and channel interfaces can be used as a place-holder where normally a module or channel type would be used, and the concrete module or channel type is determined at network setup time by a parameter.

Inheritance: modules and channels can be sub-classed. Derived modules and channels may add new parameters, in-out ports (called gates in NED language), and (in the case of compound modules) new sub-modules and connections. They may set existing parameters to a specific value, and also set the gate size of a gate vector.

Packages: the NED language features a Java-like package structure, to reduce the risk of name clashes between different models. NEDPATH (similar to Java's CLASSPATH) was also introduced to make it easier to specify dependencies among simulation models.

Inner types: channel types and module types used locally by a compound module can be defined within the compound module, in order to reduce name-space pollution.

Metadata annotations: it is possible to annotate module or channel types, parameters, gates and sub-modules by adding properties. Meta-data are not used by the simulation kernel directly, but they can carry extra information for various tools, the runtime environment, or even for other modules in the model. For example, a module's graphical representation (icon, etc) or the prompt string and measurement unit (nano second, etc) of a parameter are already specified as meta-data annotations.

5.3 Model structure in OMNeT++

An OMNeT++ model consists of hierarchically nested modules which communicate with messages. OMNeT++ models are often referred to as networks. The top level module is the system module. The system module contains sub-modules, which can also contain further sub-modules, Figure 5.1 shows an example. The depth of module nesting is not limited, this allows the user to reflect the logical structure of the actual system in the model structure.

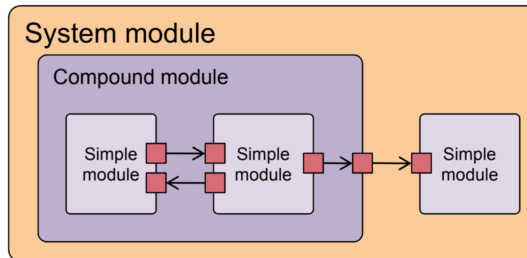


Figure 5.1: Model structure in OMNeT++: compound and simple modules, gates, connections

Modules that contain sub-modules are termed compound modules, as opposed simple modules which are at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model and they are implemented by the user. Both simple and compound modules in a given network are instances of module types. While describing the model, the user defines module types and uses them to define more complex module types. Finally, the user creates the system module as an instance of a previously defined module type. When a module type is used as a building block, there is no distinction whether it is a simple or a compound module. This allows the user to split a simple module into several simple modules (embedded into a compound module), or vice versa, aggregate the functionality of a compound module into a single simple module, without affecting existing users of the module type.

Modules communicate by exchanging messages. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queuing network or other types of mobile entities. Messages are sent out and arrive through gates, which are the input and output interfaces of a module. Input and output gates of different modules can be interconnected. Each connection is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two sub-modules, or a gate of one sub-module and a gate of the compound module.

Modules can have parameters. Parameters are used for two main purposes: to customize simple module behaviour, and to parametrize model topology. Compound modules can pass parameters or expressions of parameters to their sub-modules. Figure 5.2 provides the NED

description of a simple module (the generic arbiter).

The behaviour of each simple module is programmed in C++. This description could contain several basic components coming from the simulator library. Interested readers may have further information on this library in [61].

```
simple GenericArbiter
{
    parameters:
        int inputsNumber;
        int additiveLatency;
        int arbitrationScheme;

        @display("bgb=155,124;i=block/join");

    gates:
        input requests_in[NUM_INPUTS];
        output request_out;
}
```

Figure 5.2: The NED description of the generic arbiter

5.4 General description of a building component

All building components of the memory controller are based on the **cSimpleModule** class of OMNeT++. This class provides what is necessary for exchanging and handling messages.

A typical C++ description of a building component consists in 3 principal parts: the *initialization*, the *run-time behaviour*, and the *finishing*. These parts are respectively represented by the *initialize()*, *activity()* and *finish()* functions. The headers of the previous functions are provided in the cSimpleModule class. However, we still have to specify the content of each of them.

The ***initialize*** method is invoked after OMNeT++ has set up the system (i.e. created modules and connected them according to the definitions), and provides a place for initialization code. The initialization code deals with the module parameters reading (from the NED file to the executable file), and the global variables assignment. Figure 5.3 shows the initialization method of the generic arbiter.

```
void cGenericArbiter::initialize(void){
       NUM_INPUTS = par("inputsNumber");
       ADDITIVE_LATENCY = par("additiveLatency");
       TIME_UNIT = simulation.getModule(1)->par("timeUnit");
       ARBITRATION_SCHEME = par("arbitrationScheme");
       selected_port=0;
       last_port = 0;
}
```

Figure 5.3: The initialization method of the generic arbiter

The ***activity*** method is run in a coroutine. Coroutines are similar to threads, but are

scheduled non-preemptively (this is also called cooperative multitasking). From one coroutine you can switch to another coroutine by a *transferTo*(otherCoroutine) call. Then this coroutine is suspended and (otherCoroutine) will run. Later, when (otherCoroutine) does a *transferTo*(otherCoroutine) call, execution of the first coroutine will resume from the point of the *transferTo*(otherCoroutine) call. The full state of the coroutine, including local variables are preserved while the thread of execution is in other coroutines. This implies that each coroutine must have its own stack, and *transferTo()* involves a switch from one process to another. Coroutines are at the heart of OMNeT++, and the simulation programmer does not ever need to call *transferTo()* or other functions in the coroutine library, nor does he need to care about the coroutine library implementation. However, it is important to understand how the event loop found in discrete event simulators works with coroutines. The clock of each module is modelled in the *activity* method. The time unit is one of the module global variable which are assigned during the initialization process. Figure 5.4 depicts the *activity* method of the generic arbiter.

```
01: void cGenericArbiter::activity(){
02:  int portIndex;
03:  cMessage *msg;
04:  mFlit *flit;
05:  cMessage *clockEvent = new cMessage("Clock Event genericArbiter");
06:
07:  for (;;) {
08:     scheduleAt( simTime() + TIME_UNIT, clockEvent );
09:     while((msg=receive()) != clockEvent) {
10:       flit = msg;
11:       portIndex = flit->getArrivalGate()->getIndex();
12:       inputQueues->storeElement(portIndex, flit);
13:     }
14:     arbitrateInputs();
15:  }
16:}
```

Comments:
Line 08: The *scheduleAt()* function schedules the *clockEvent* message which will serve as a clock for this module. The clock period is *TIME_UNIT*. This function delivers the *clockEvent* back at simulation time (*simTime()+TIME_UNIT*).
Line 09: The received message can be either the clock event (self-message) or an arriving message from another module. We verify that it is not a self-message.
Line 11: We get the arrival input index for the reveived *flit* in order to know in which *inputQueue* we must store it.
Line 14: We call the *arbitrateInputs()* function which will schedule, according to the chosen policy, the inputQueues.

Figure 5.4: The activity method of the generic arbiter

The ***finish*** method is called when the simulation has terminated successfully, and its recommended use is the recording of summary statistics.

5.5 Memory controller building components parameters

Each building component of our customizable memory controller receives its parameters through a NED file. The following tables show the parameters of each building component with its description and values range.

5.5.1 Memory mapping parameters

Hereafter the parameters of the memory mapping unit. More details about this block are given in section (3.6.1) on page 42.

Table 5.1: Memory mapping parameters

Parameter name	Type	Range	Description
timeUnit	double	$1 \rightarrow 10$	relative clock cycle period
memAddressWidth	int	$32 \rightarrow 128$	Memory address space width in bits
bankBits	int	$2 \rightarrow 4$	Bank bits in the memory address
rowBits	int	$8 \rightarrow 12$	Row bits in the memory address
columnBits	int	$12 \rightarrow 18$	Column bits in the memory address
mappingScheme	int	$0 \rightarrow 2$	RBC, RCB and BRC mapping schemes
additiveLatency	int	$0 \rightarrow 999$	An additive delay that a request spends in this unit

5.5.2 Generic queue parameters

Below the parameters of the generic queue. More details about this block are given in section (3.6.2) on page 43.

Table 5.2: Generic queue parameters

Parameter name	Type	Range	Description
timeUnit	double	$1 \rightarrow 10$	relative clock cycle period
numOfQueues	int	$1 \rightarrow 16$	Number of queue inside the generic queue unit
maxLength	int	$1 \rightarrow 64$	Length of each queue in the generic queue unit
ageing	bool	true, false	Activation of the ageing mechanism
maxAge	int	$1 \rightarrow 500$	Threshold in clock cycles from which the priority of each element in the queue is incremented by 1
additiveLatency	int	$0 \rightarrow 16$	An additive delay that an element spends in the queue

5.5.3 Capture unit parameters

Table 5.3 shows the parameters of the capturing unit. More details about this block are given in section (3.6.3) on page 44.

Table 5.3: Capturing unit parameters

Parameter name	Type	Range	Description
lookAheadWindow	int	$1 \rightarrow 64$	The look-ahead window depth in the queue. This reflects the number of queue slots that are observable by the capturing unit
capturingRule	int	$1 \rightarrow 4$	Several rules could be used consecutively

5.5.4 Insertion unit parameters

Hereafter the parameters of the insertion unit. More details about this block are given in section (3.6.4) on page 47.

Table 5.4: Insertion unit parameters

Parameter name	Type	Range	Description
lookAheadWindow	int	$1 \to 64$	The look-ahead window depth in the queue. This reflects the number of queue slots that are observable by the insertion unit
insertionRule	int	$1 \to 5$	Several rules could be used consecutively

5.5.5 Generic arbiter parameters

Below the parameters of the generic arbiter. More details about this block are given in section (3.6.5) on page 51.

Table 5.5: Generic arbiter parameters

Parameter name	Type	Range	Description
timeUnit	double	$1 \to 10$	relative clock cycle period
inputsNumber	int	$2 \to 16$	Inputs to be arbitrated
arbitrationScheme	int	$1 \to 5$	Scheduling algorithm to be used
additiveLatency	int	$0 \to 16$	An additive delay that a request spends to be scheduled

5.5.6 Re-ordering unit parameters

The parameters of the re-ordering unit are shown in Table 5.6. More details about this block are given in section (3.6.7) on page 55.

Table 5.6: Re-ordering unit parameters

Parameter name	Type	Range	Description
timeUnit	double	$1 \to 10$	relative clock cycle period
portsNumber	int	$1 \to 16$	Memory controller interface ports number
outstandingRequests	int	$1 \to 64$	Number of outstanding requests in the memory controller through the same port
respQueueLength	int	$1 \to 32$	Maximum capacity of the response queue
additiveLatency	int	$0 \to 16$	An additive delay that a request spends to be scheduled

5.6 EEEP components parameters

The EEEP unit in the master network interface and the *Credit Generation Unit* in the memory controller also need to be parametrized. Table 5.7 summarizes the parameters that the master network interface and the memory controller when the *Extreme End to End Protocol* is enabled for the *best effort* traffic (BE).

The initial credits number of each master network interface depends on the requirements of the concerned master in term of bandwidth. We explicitly omit the latency requirement for those masters because they already generate BE traffic. The memory controller is aware of the master network interfaces that inject BE traffic, and consequently sends the EEEP credits back to these network interfaces.

Table 5.7: EEEP parameters

Parameter name	Type	Range	Description
eeepEnabled	bool	true, false	Enable or disable the EEEP mechanism for this NI
eeepInitialCredit	int	$1 \rightarrow 16$	The number of the initial credits for this NI. This value is related to the number of queue slots allocated to each *best effort* NI. See equation (4.7) on page 77

5.7 Traffic generator

Mapping multiple applications on available computational resources leads to interaction and contention at various network resources. Consequently, taking into account the traffic characteristics becomes of crucial importance for performance analysis and optimization of the communication infrastructure, as well as proper resource management [9; 14; 46].

We build on the statistical distributions provided by OMNeT++ in order to create our traffic generator. This generator has two modes to create stimuli. The first one is *constrained random* traffic generation, and the second one is *back-annotated* traffic generation.

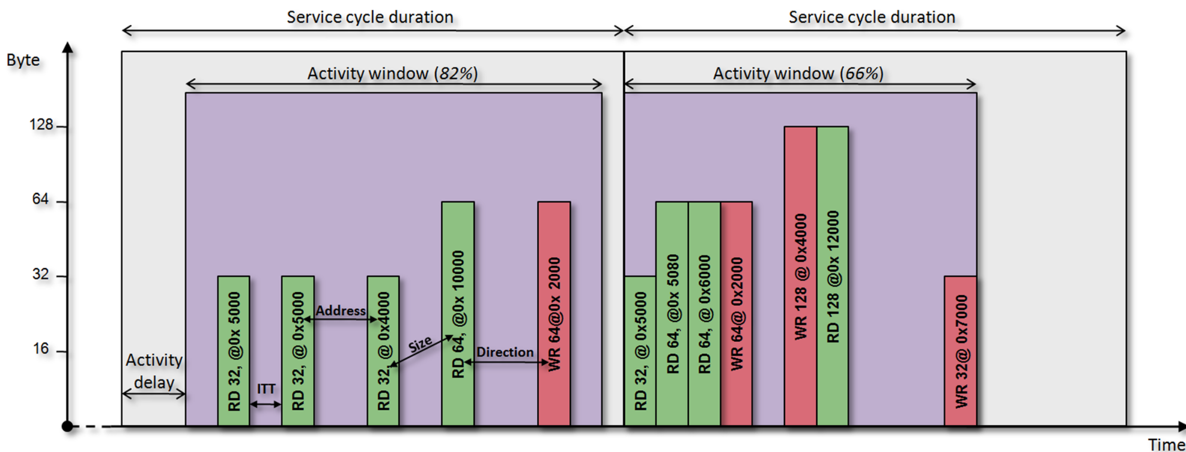


Figure 5.5: The activity method of the generic arbiter

In the ***constrained random*** mode, we have seven parameters to define in order to configure the shape of the generated traffic:

1) **Service cycle duration**, the repetitive period of time in which we shape the traffic.

2) **Activity window**, a time window inside the *service cycle*, in which the traffic generator is allowed to issue requests.

3) **Activity delay**, the delay time between the beginning of the *service cycle* and the beginning of the *activity window*.

4) **ITT**, *Inter Transaction Time*, a delay between the current transaction and the next one.

5) **Address**, the address range(s) that the requests want to access.

6) **Size**, the size of the transactions in bytes.

7) **Direction**, the direction of the requests (read/write).

Only the Service cycle duration parameter has a hard value that remains the same during the simulation. All other parameters are based on statistical distributions supported by the simulator[10]. Figure 5.5 shows the signification of the previous parameters in the traffic shape. Table 5.8 shows an example of the *constrained-random* configuration.

Table 5.8: Traffic generator, example of the *constrained random* configuration

Parameter	Description
service cycle duration = 1000	The service cycle duration in clock cycles
activity window = uniform(100,500)	For each service cycle, the activity window will have have a random value between 100 and 500 clock cycles. This value is generated according to the uniform distribution
activity delay = histogram((5,25),(20,75))	For each service cycle, the activity delay will be 25% 5 clock cycles, and 75% 20 clock cycles
ITT = uniform(1,50)	The ITT inside the activity window will be between 1 and 50 clock cycles
address = histogram((uniform(1000,2000),40),(uniform(5000,9000),60)	40% between 1000 and 2000, and 60% between 5000 and 9000
size = histogram((8,20),(16,20),(32,50),(64,5),(128,5))	20% 8 bytes, 20% 16 bytes, 50% 32 bytes, 5% 64 bytes, 5% 128 bytes
direction = fixed(1)	Only write requests

In the ***back-annotated*** mode, we use a stimuli file which includes the requests to generate. The requests represent the real behaviour a processing units if the stimuli file contains the trace of an emulation system. In this mode, we keep the service cycle duration, active window, activity delay and ITT parameters within the generation policy, and we replace the address, size and direction parameters with values provided in the stimuli file. The following figure depicts an example of a stimuli file for the *back-annotated* generation mode.

```
service cycle width = 1000
activity window = uniform(100,500)
activity delay = uniform(1,30)
ITT = fixed(0)
RD 22004FD0  8
RD 2654A9C0  8
RD 2654A9A0  16
RD 22004FD0  16
RD 24C97F10  8
WR 21F6EBC8  8
WR 25FFA880  32
RD 240AEBA8  32
RD 24CA3120  64
. . . . . . . .
. . . . . .
. . . .
. .
.
```


Figure 5.6: The header of a stimuli file for the *back-annotated* generation mode

[10] Among the statistical distributions we note: Uniform, Exponential, Normal, Erlang, Student, Cauchy, Bernoulli, Binomial, Poisson and Histogram.

5.8 Conclusion

This chapter was devoted to the description of the implementation of our totally customizable memory controller architecture in addition to the *Extreme End to End Protocol*. We chose OMNeT++ as working environment not only because it can be used as high level and cycle approximate simulator, but also because of its rich library of basic components on which we built our customizable architecture. Furthermore, we use the statistical distribution functions provided by the simulator to build our traffic generator which can nearly simulate the real behaviour of several processing units.

The implementation of the building components under OMNeT++ makes them easy to instantiate and to interface with each other. This makes the model flexible and allows the memory system designer to explore the architecture in order to find the configuration that meets the requirements of the concerned processing engines.

CHAPTER **6**

Experiments and Results

Contents

6.1 Memory system **95**
6.1.1 Memory controller architecture 95
6.2 Standalone tests **96**
6.2.1 Memory controller configuration for standalone tests 96
6.2.2 Memory timing tests 97
6.2.3 Priority and ageing mechanism test 99
6.2.4 Summary 102
6.3 EEEP tests **102**
6.3.1 Traffic modelling 103
6.3.2 EEEP in a Spidergon NoC-based SoC 103
6.3.3 EEEP in a 2DMesh NoC-based SoC 107
6.3.4 EEEP in an irregular NoC-based SoC 107
6.3.5 Analysis 111
6.4 Conclusion **112**

EVALUATING the performance of MPSoCs became a hard task with the increasing complexity of their architectures. The memory subsystem performance has rapidly attracted the designer interest as most processing engines access it. In this chapter, we are going to analyse the performance of a memory subsystem made up of a multi-port memory controller with a DDR3-800 SDRAM model. We will start the evaluation by verifying the memory timing constraints with several standalone tests. Then we will move on to test our novel *Extreme End-to-End Protocol* within modern multimedia SoCs based-on different network-on-chip topologies.

6.1 Memory system

We need a multi-port memory controller to evaluate the performance of our *extreme end to end protocol*. This is why we opt for the memory controller *Beta* we showed in section 3.7.2 on page 56. Depending on the type of experiment we will be working on, we will have to change the configuration of the memory controller to meet the system requirements. This will be notified, if necessary, in each subsection of this chapter.

6.1.1 Memory controller architecture

We remove the first arbitration stage between read and write requests inside each port because the interface we chose mixes together the read and write requests in the same channel. So the only arbiter we use is the inter-port arbiter, which is a priority-based arbiter (see Figure 6.1). Each port has its own priority value, which ranges from the highest priority (*pr0*) to the lowest priority (*pr3*).

When a request is selected by the port arbiter, it is forwarded to the placement unit to be placed in the *requests queue*. The insertion rules are respectively: (1) system data consistency; (2) master data consistency; (3) priority; (4) direction grouping. Once placed into the *requests queue*, the relative order of the requests is constant.

When the status of the back-end allows to receive a new request, the front-end forwards the request which is in the head of the *requests queue* to the back-end. Upon the receiving of a request, the back-end converts it into DDR3 SDRAM commands, and buffers them inside a *memory commands queue*. These commands will sequentially be sent to the memory device according to the DDR3 SDRAM timing. Note that this memory controller uses an *early response* mechanism, which consists of sending the write response back to the master when the write request is scheduled. It does not wait until the end of the write operation to send the response back.

Figure 6.1 depicts the architecture of the memory controller we use for our experiments.

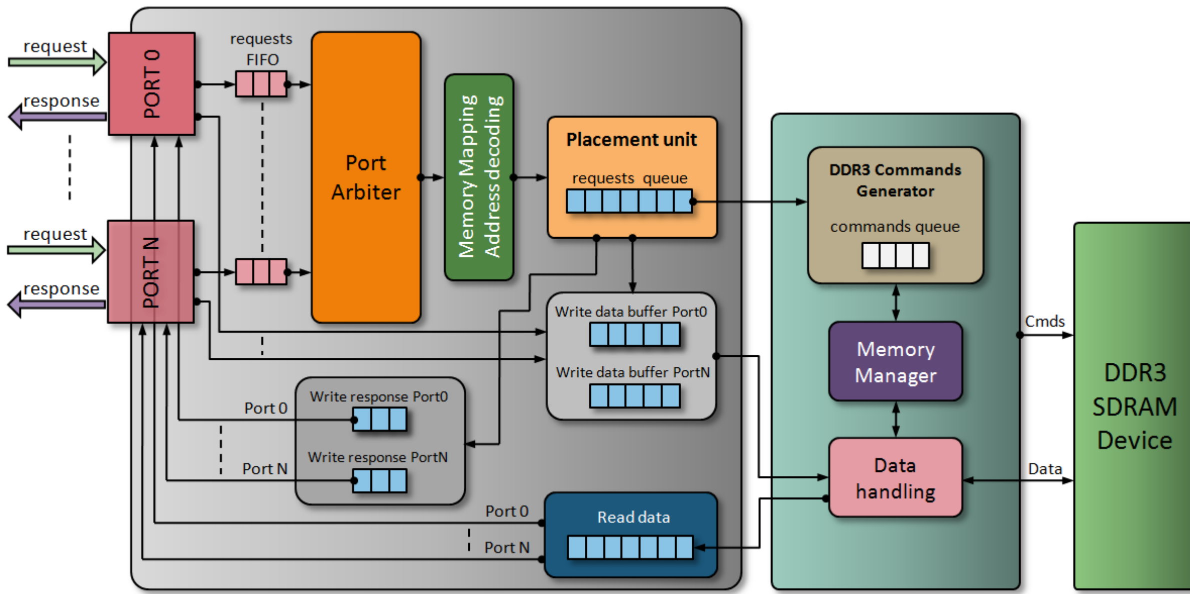


Figure 6.1: Architecture of the multi-port memory controller

6.2 Standalone tests

The goal of these tests is to verify most important timing parameters of the memory controller. We also show the impact of other parameters on the system performance such as *port priorities* and *ageing* mechanism. We start these tests by configuring the memory controller.

6.2.1 Memory controller configuration for standalone tests

Table 6.1 shows the configuration of the memory. The choice of the queues depth is tightly correlated with the kind of traffic that the memory controller will receive. For this set of standalone tests, we use two traffic generators that have an outstanding[11] value of 4. So the maximum number of outstanding requests that we can have in the memory controller is 8 requests. The write-data queue length should also have the capacity to store at least half of the longest write request that the memory controller may receive. The longest write requests in these preliminary tests is 128 bytes.

We configure the back-end in order to support DDR3-800 SDRAM timings. We use the timing values of the Samsung DDR3-800 K4B4G0446A [69], which are summarized in Table 6.2. The definitions of these timing parameters have been presented in Table 3.2 on page 40.

[11]The outstanding parameter in a traffic generator represents the maximum number of requests that the generator can issue without receiving the response of the first request

Table 6.1: Memory controller configuration for standalone tests

Parameter	Value
Number of ports	2
Front-end data path width	8 bytes
Back-end data path width	4 bytes
Memory data bus width	4 bytes
Front-end & back-end & memory device frequency	400 MHz
Port FIFOs depth	2 slots
Arbitration scheme	Priority then round-robin
Memory address space	32 bits
Memory mapping	Row(17 bits) / Bank (3 bits) / Column (12 bits)
Placement unit requests queue depth	8 slots
Back-end ddr3 commands queue depth	4 slots
Write data queues depth	8 slots
Read data queue depth	4 slots
Write reponse queues depth	2 slots

Table 6.2: Samsung DDR3-800 SDRAM timing parameters

Timing parameter	Value [clock cycle]	Timing parameter	Value [clock cycle]	Timing parameter	Value [clock cycle]
tRAS	14	tRC	21	tRCD	6
tCCD	4	tRP	6	tRTP	4
tREFI	1560	tRFC	64	tRL	5
tWL	5	tWR	15	tWTR	4
tFAW	20	-	-	-	-

6.2.2 Memory timing tests

We focus now on the duration of the back-end operations to execute the requests that the memory controller receives. We are especially interested in: 1) the time penalty of *row miss* and bus direction switching, 2) the bank interleaving mechanism, 3) the requests format.

Direction switching test

The goal of this test is to verify the access delays which are due to the memory bus switching direction between read and write.

We send a set of stimuli through one traffic generator to *Bank 0*. The requests include all cases of row hits/misses and read/write switching. In order to simplify the explanation of the back-end delays, we only send read and write requests of 32 bytes. The left column in Figure 6.2 represents the request of the traffic generator. By following the sequences in the right column of the same figure, we verify that the back-end respects the timing constraints of the DDR3-800 SDRAM.

Bank interleaving test

The idea behind this test is to verify and show that the back-end can interleave the bank preparation commands in order to hide at maximum the bank preparation delays, i.e. the

Traffic generator

Memory Requests
RD 00000000 32
RD 00008000 32
RD 00008000 32
WR 00008000 32
RD 00008000 32
WR 00000000 32
RD 00008000 32
WR 00008000 32
WR 00008000 32
WR 00000000 32

Memory controller back-end

Clock cycle	Memory command	Bank number	Row number
12:	ACT	Bank 0	Row 0
18:	RD_8	Bank 0	Row 0
26:	PRCH	Bank 0	Row 0
33:	ACT	Bank 0	Row 1
39:	RD_8	Bank 0	Row 1
43:	RD_8	Bank 0	Row 1
49:	WR_8	Bank 0	Row 1
61:	RD_8	Bank 0	Row 1
61:	PRCH	Bank 0	Row 1
67:	ACT	Bank 0	Row 0
73:	WR_8	Bank 0	Row 0
97:	PRCH	Bank 0	Row 0
103:	ACT	Bank 0	Row 1
109:	RD_8	Bank 0	Row 1
115:	WR_8	Bank 0	Row 1
119:	WR_8	Bank 0	Row 1
143:	PRCH	Bank 0	Row 1
149:	ACT	Bank 0	Row 0
155:	WR_8	Bank 0	Row 0

Memory transition time
tRCD = 6
tRTP = 4 (ACT 2 PRE =14 => +4)
tRP = 6
tRCD = 6
tCCD = 4
tRL+tCCD+2tCK-tWL = 6
tWR+tCCD+tWTR = 5+4+4 = 13
tRTP = 4
tRP = 6
tRCD = 6
tWL +tCCD + tWR = 5+4+15 = 24
tRP = 6
tRCD = 6
tRL+tCCD+2tCK-tWL = 6
tCCD=4
tWL+tCCD +tWR = 5+4+15 = 24
tRP = 6
tRCD = 6

Figure 6.2: Back-end log file: direction switching and bank preparation delay

Traffic generator

Memory Requests
RD 00000000 32
RD 00001000 32
RD 00010000 32
RD 00011000 32

Memory controller back-end

Clock cycle	Memory command	Bank number	Row number
12:	ACT	Bank 0	Row 0
14:	ACT	Bank 1	Row 0
18:	RD_8	Bank 0	Row 0
22:	RD_8	Bank 1	Row 0
26:	PRCH	Bank 0	Row 0
28:	PRCH	Bank 1	Row 0
32:	ACT	Bank 0	Row 2
34:	ACT	Bank 1	Row 2
38:	RD_8	Bank 0	Row 2
42:	RD_8	Bank 1	Row 2

Figure 6.3: Back-end log file: bank interleaving mechanism

the delays due to *Prechage* and *Activate* commands. Figure 6.3 shows a sequence of read requests that access *bank 0* and *bank 1* and create in both banks row misses. Note that at the beginning, all banks are in the *Idle* state. For more details about the banks finite state machines, please refer to Figure 3.2 on page 39.

Requests format

Here, we would like just to show how the memory controller interprets the cores requests and formats them to fit in the DDR3-SDRAM commands granularity. Figure 6.4 depicts a set of write requests ranging from 8 bytes to 128 bytes. These requests do not present any bank conflict. According to the *length* of the request, the back-end will divide it into a set of *WR_4* and *WR_8* memory commands.

Traffic generator

Memory Requests
WR 00000000 8
WR 00011000 16
WR 00022000 32
WR 00033000 64
WR 00044000 92
WR 00055000 128

Memory controller back-end

Clock cycle	Memory command	Bank number	Row number
13:	ACT	Bank 0	Row 0
17:	ACT	Bank 1	Row 2
19:	WR_4	Bank 0	Row 0
23:	WR_4	Bank 1	Row 2
24:	ACT	Bank 2	Row 4
25:	ACT	Bank 3	Row 6
30:	WR_8	Bank 2	Row 4
31:	WR_8	Bank 3	Row 6
34:	ACT	Bank 4	Row 8
35:	WR_8	Bank 3	Row 6
40:	WR_8	Bank 4	Row 8
44:	WR_8	Bank 4	Row 8
45:	ACT	Bank 5	Row 10
48:	WR_8	Bank 4	Row 8
52:	WR_8	Bank 5	Row 10
56:	WR_8	Bank 5	Row 10
60:	WR_8	Bank 5	Row 10
64:	WR_8	Bank 5	Row 10

Figure 6.4: Back-end log file: requests format

Four-activation window

As the row activation process correspond to copying an entire row from the memory matrix in the row buffer in a bank, this operation consumes a lot of power, leading to a power consumption peak in the memory module. For this reason, the DDR3 SDRAM standard [77] limits the number of activation commands to 4 with a tFAW time window (time Four-Activation Window). As the DDR3-800 has a tFAW of 20 clock cycles, none of the examples shown in Figure 6.2, Figure 6.3, and Figure 6.4 violates this timing constraint.

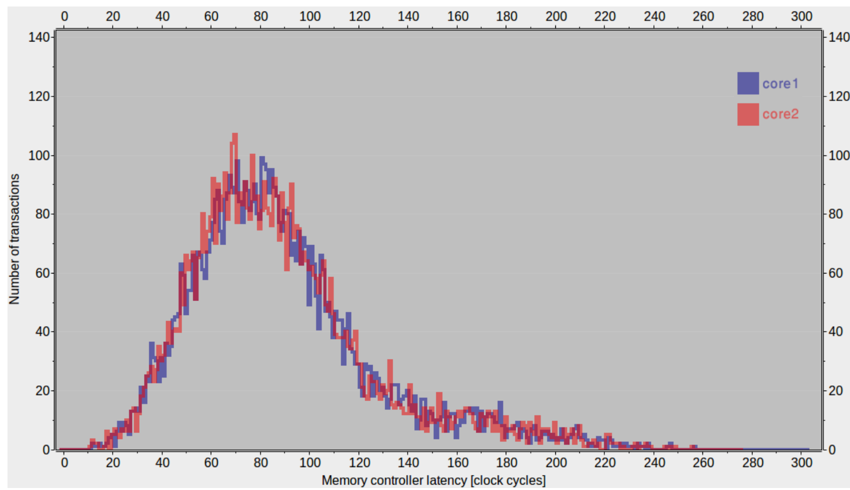
6.2.3 Priority and ageing mechanism test

Here, we would like to highlight the impact of the port priority on the latency and bandwidth of the initiator that accesses the memory controller through this port.

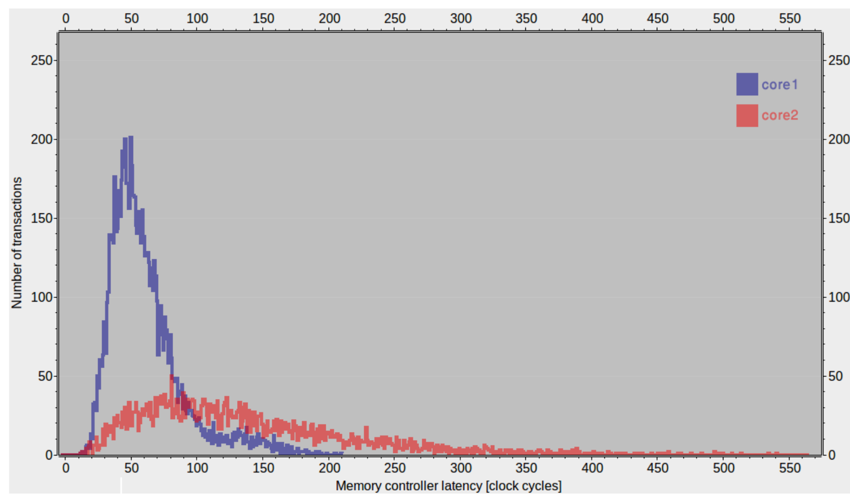
We keep the configuration of the memory controller as shown in Table 6.1, but we use two traffic generators that represent two initiators. Both traffic generators use the *constrained random* generation mode, and they have the same configuration (see Table 6.3).

We ran 3 simulations during 200k clock cycles, a time long enough to cover several refresh intervals (more than 128 refresh intervals). In the first simulation, *port1* and *port2* have the same priority. In the second simulation, we give to *port1* the highest priority level *pr0*, and the lowest priority level *pr3* to *port2*. The *ageing* mechanism is disabled in the first two runs. In the last simulation, we keep the port priorities as described previously, and we activate the *ageing* mechanism with a *maxAge* of 10 clock cycles for the queued requests. Figures 6.5 and 6.6 show the latency histogram and the average bandwidth for both initiators.

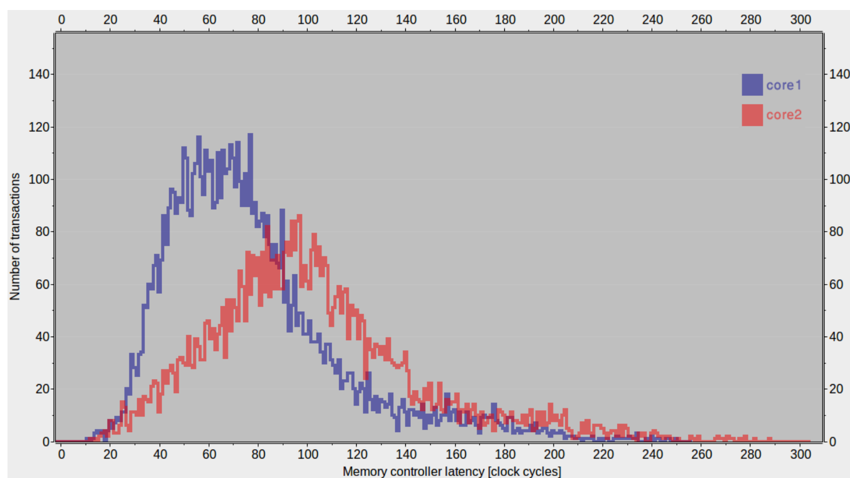
The simulation results show that the memory controller latency and the average band-



(a) Both cores have the same priority

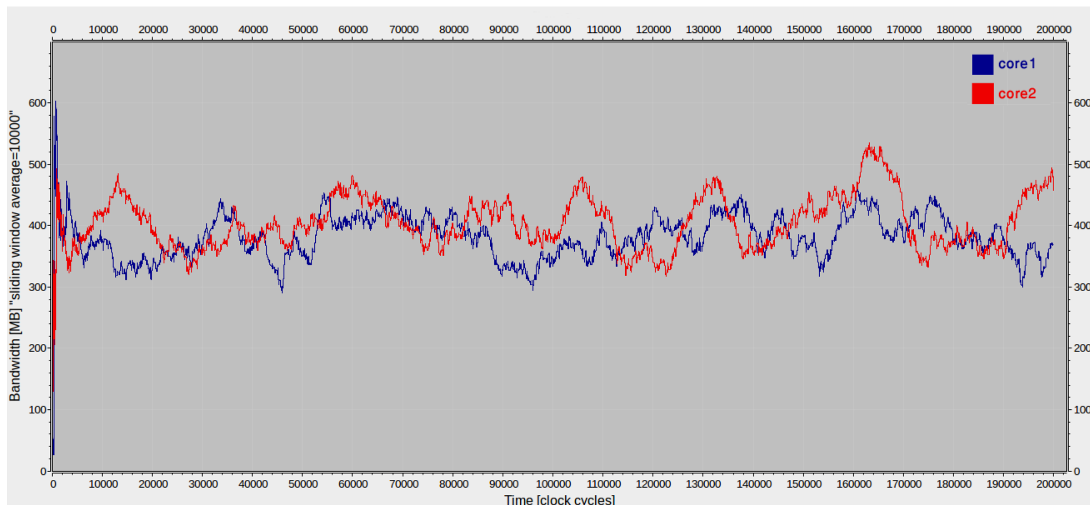


(b) core1 has the highest priority (pr=0), and core2 has the lowest priority (pr=3)

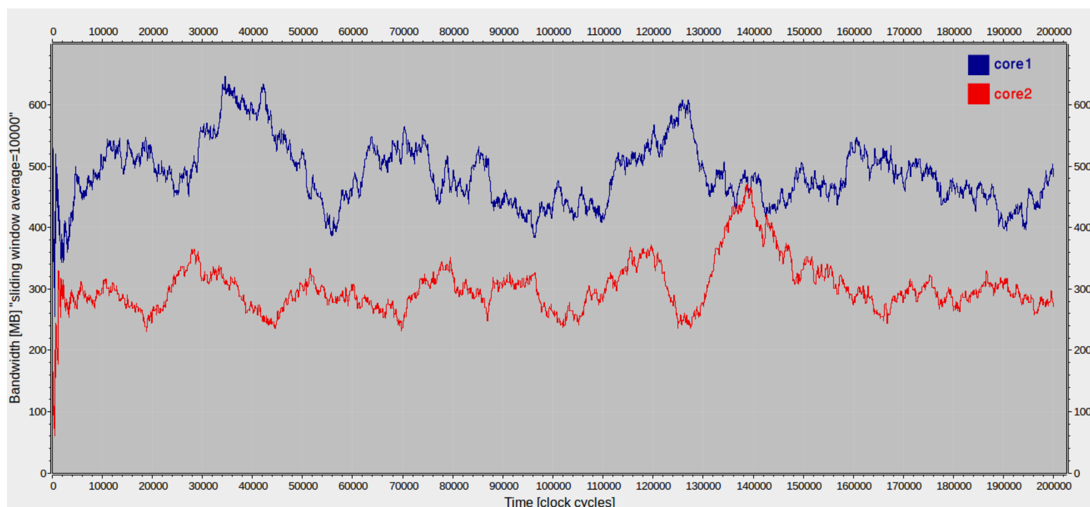


(c) core1 has the highest priority (pr=0), and core2 has the lowest priority (pr=3). The ageing mechanism is activated with *maxAge=10*

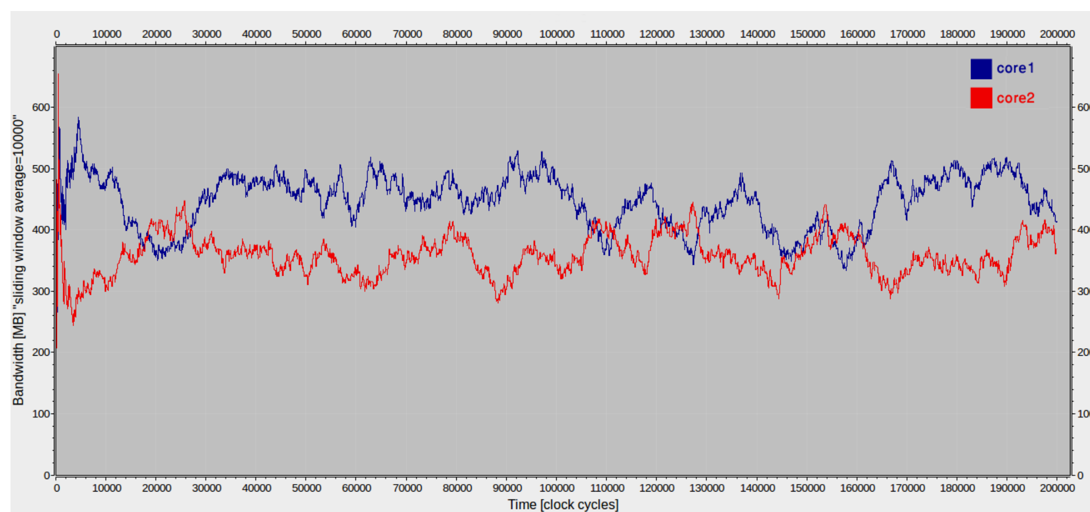
Figure 6.5: Memory controller latency histogram for core1 and core2



(a) Both cores have the same priority



(b) core1 has the highest priority (pr=0), and core2 has the lowest priority (pr=3)



(c) core1 has the highest priority (pr=0), and core2 has the lowest priority (pr=3). The ageing mechanism is activated with *maxAge=10*

Figure 6.6: Moving average bandwidth for core1 and core2

Table 6.3: Traffic generators configuration in *constrained random* mode

Parameter	Value	Comments
outstanding	4	-
service cycle duration	1000	in clock cycles
activity window	1000	the whole service cycle
activity delay	fixed(0)	-
ITT	fixed(1)	No additive delay between requests
address	uniform(0x00000000,0x00004000)	It covers all banks
size	histogram((8,33),(32,33),(64,17),(128,17))	33% 8 B, 33% 32 B, 17% 64 B, 17% 128 B
direction	histogram((0,66),(1,33))	66% read, 33% write

width are approximately the same for both initiators when we give the same priority level to *port1* and *port2*. When the memory controller ports have different priority levels, the requests of *core2* are always placed behind the requests of *core1*. This is why *core1* has a tight and picky latency histogram with an average latency of 70 clock cycles, and *core2* has a large and flat latency histogram with an average latency of 142 clock cycles (see figure 6.5b). A trade-off between the performance of *core1* and *core2* can be obtained by using the ageing mechanism. This mechanism increases the priority level of *core2* requests every 10 clock cycles. When the priority level of these requests become similar to *core1* requests, all new requests are placed behind them. This explains the difference between Figure 6.5b and Figure 6.5c.

6.2.4 Summary

In this section, we have verified most important timing parameters of the memory controller we use in this chapter. We showed that these timing values are totally matched with the DDR3 SDRAM constraints. We tested then the influence of the port priority, coupled with the ageing mechanism. We have shown that the ageing mechanism can avoid starvation situations for the masters that have low priority. The rest of this chapter is devoted to the evaluation of our *Extreme End-to-End Protocol*.

6.3 EEEP tests

We evaluate now the system performance variation when the *Extreme End-to-End Protocol* is used for *best effort* traffic in multimedia SoCs. We use a set of traffic models for CPU, display controller, video decoder, GPU and blitter. We map the system on three different NoC topologies: Spidergon 16, 2DMesh 4x4 and irregular. One main memory system is used for each topology. It consists of the memory controller that we have shown in Figure 6.1, connected to a model of the Samsung DDR3-800 SDRAM whose the parameters are shown in Table 6.2. We show in Table 6.4 the configuration of the memory system we use in this section.

Table 6.4: Memory controller configuration for EEEP tests

Parameter	Value
Number of ports	4
Front-end data path width	16 bytes
Back-end data path width	8 bytes
Memory data bus width	8 bytes
Front-end & back-end & memory device frequency	400 MHz
Port FIFOs depth	4 slots
Port arbitration policy	Priority then round-robin
Memory address space	32 bits
Memory mapping	RBC, Row(17 bits) / Bank (3 bits) / Column (12 bits)
Placement unit requests queue depth	32 slots for Spidergon and 2DMesh, 20 slots for irregular
Back-end ddr3 commands queue depth	6 slots
Write data queues depth	24 slots
Read data queue depth	24 slots
Write response queues depth	4 slots

The sizing of the queues in the memory controller is based-on the kind of traffic that accesses the SDRAM. We take into account the maximum number of outstanding requests that the memory controller can have, and the maximum length in bytes of each request when we size the queues.

6.3.1 Traffic modelling

Based on the methodology presented by **Srinivasan and Salminen [73]**, we model the traffic of the system in order to test the extreme end-to-end protocol. We use our traffic generator in *constrained random* mode (see section 5.7 for more details) to model the traffic of several IPs such as CPU, GPU, video decoder, display controller and blitter. We summarize the configuration of these traffic generators in Table 6.5.

Note that 80% of the system traffic is accessing the main memory subsystem, and 20% is accessing other targets in the system.

6.3.2 EEEP in a Spidergon NoC-based SoC

We build a simulation platform made up of 4 CPUs, 3 GPUs, 2 display controllers, 4 video decoders, and 3 blitters. These IPs are connected to a main memory system and other targets through an ST Spidergon Network-on-chip. Figure 6.7 shows the simulation platform. The description of the traffic injected by the initiators is given in Table 6.5. The configuration of the memory system is shown in Table 6.4. We consider the blitters traffic as *best effort* traffic (BE), and all other traffic as *guaranteed service* traffic (GS). We give the blitter traffic the lowest priority in the network and the memory controller.

We activate the EEEP unit only in the blitter network interfaces, and allocate 3 slots for each blitter in the memory controller *requests queue*. We test the protocol with 2 different credit thresholds, 1 and 3. We compare the system performance when EEEP is enabled with

Table 6.5: Traffic generators configuration in *constrained random* mode for EEEP tests

IP name	Parameter	Value
CPUs *5-10% of system BW*	outstanding	2
	service cycle duration	1000
	activity window	1000
	activity delay	fixed(0)
	ITT	uniform(20,60)
	address	uniform(0xC0000000,0xFFFFFFFF)
	size	fixed(64)
	direction	histogram((0,70),(1,30))
GPUs *10-15% of system BW*	outstanding	2
	service cycle duration	4000
	activity window	2000
	activity delay	uniform(0,1999)
	ITT	uniform(1,100)
	address[c]	uniform(0x00000000,0x2FFFFFFF)
	size	histogram((128,50),(256,50))
	direction	histogram((0,66),(1,33))
Video decoders *15-20% of system BW*	outstanding	2
	service cycle duration	3000
	activity window	1000
	activity delay	uniform(0,999)
	ITT	uniform(1,80)
	address[a]	uniform(0x30000000,0x5FFFFFFF)
	size	histogram((128,50),(384,50))
	direction	histogram((0,66),(1,33))
Display controllers *20-30% of system BW*	outstanding	3
	service cycle duration	35000
	activity window	10000
	activity delay	uniform(0,9999)
	ITT	uniform(1,25)
	address[b]	uniform(0x60000000,0x8FFFFFFF)
	size	histogram((128,50),(384,50))
	direction	histogram((0,66),(1,33))
Blitters *15-20% of system BW*	outstanding	6
	service cycle duration	4000
	activity window	1000
	activity delay	uniform(0,1999)
	ITT	uniform(1,160)
	address[c]	uniform(0x90000000,0xBFFFFFFF)
	size	histogram((128,50),(256,50))
	direction	histogram((0,66),(1,33))

[a] Read addresses per row are correlated. Write addresses per row are correlated
[b] Read addresses for the display controller are correlated
[c] Read addresses are correlated. Write addresses are correlated

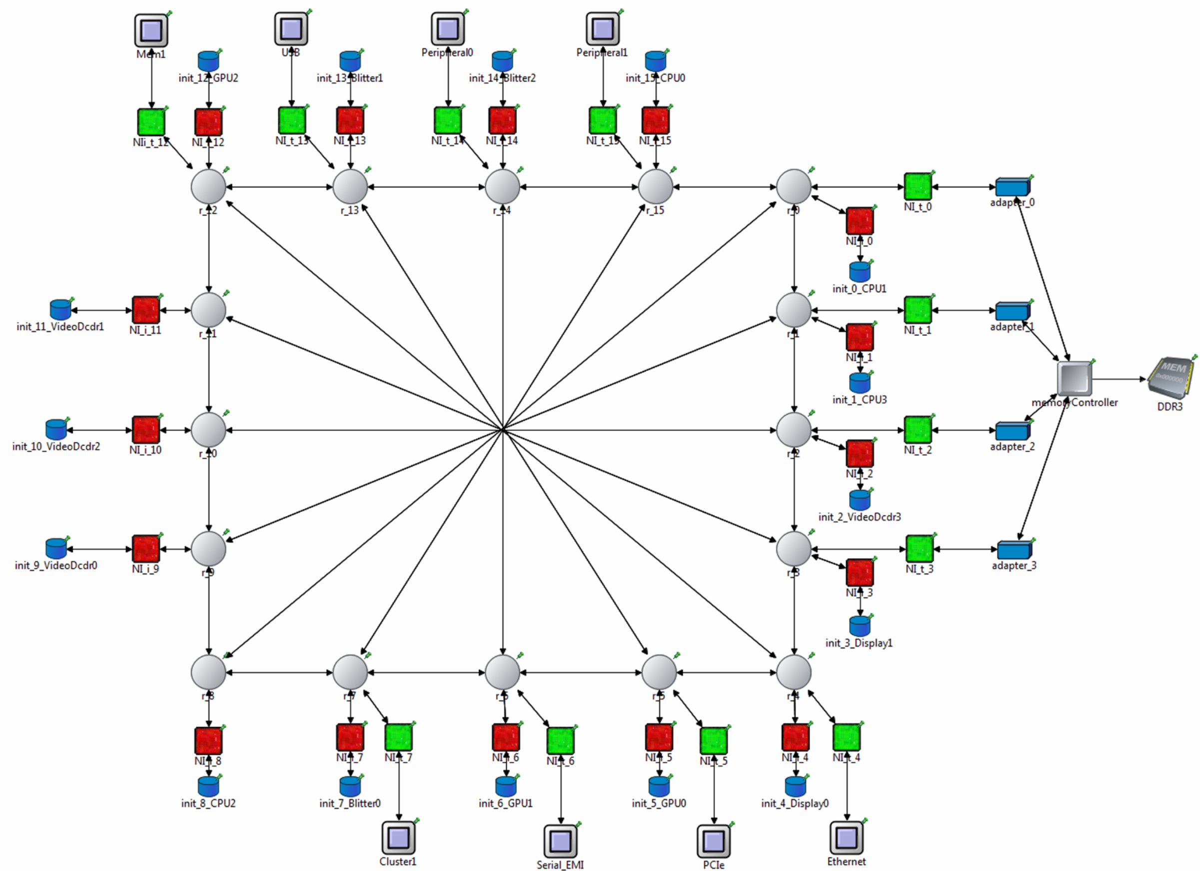
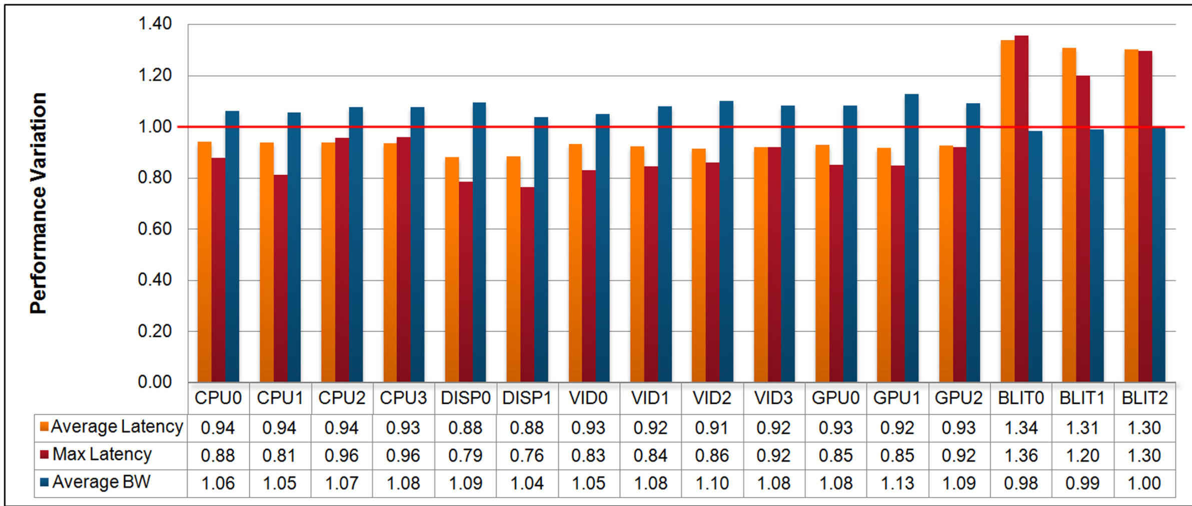
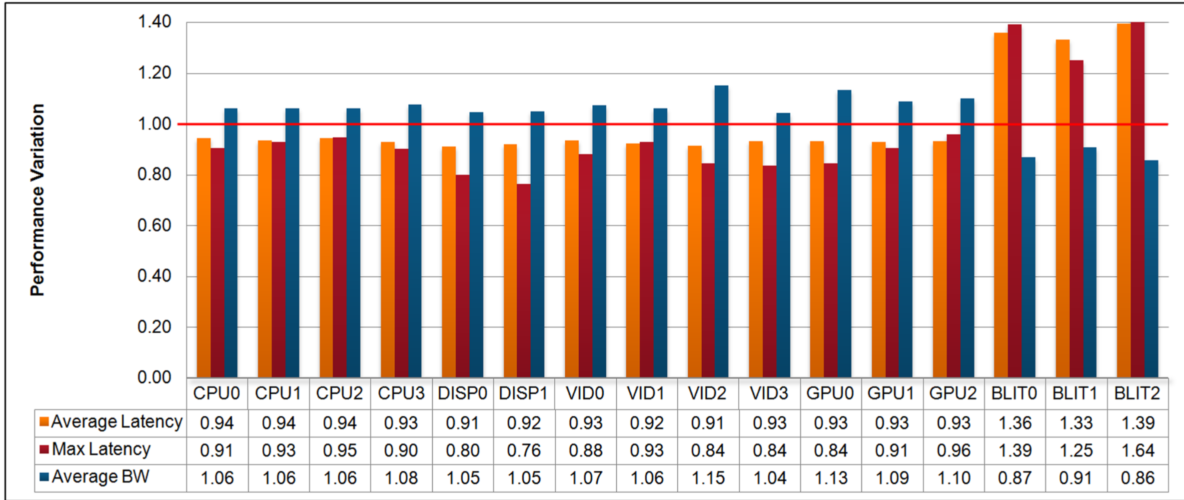


Figure 6.7: Spidergon NoC-based simulation platform (across last routing)



	CPU0	CPU1	CPU2	CPU3	DISP0	DISP1	VID0	VID1	VID2	VID3	GPU0	GPU1	GPU2	BLIT0	BLIT1	BLIT2
Average Latency	0.94	0.94	0.94	0.93	0.88	0.88	0.93	0.92	0.91	0.92	0.93	0.92	0.93	1.34	1.31	1.30
Max Latency	0.88	0.81	0.96	0.96	0.79	0.76	0.83	0.84	0.86	0.92	0.85	0.85	0.92	1.36	1.20	1.30
Average BW	1.06	1.05	1.07	1.08	1.09	1.04	1.05	1.08	1.10	1.08	1.08	1.13	1.09	0.98	0.99	1.00

(a) EEEP credits=3, EEEP credit threshold=1 for the Blitters NIs



	CPU0	CPU1	CPU2	CPU3	DISP0	DISP1	VID0	VID1	VID2	VID3	GPU0	GPU1	GPU2	BLIT0	BLIT1	BLIT2
Average Latency	0.94	0.94	0.94	0.93	0.91	0.92	0.93	0.92	0.91	0.93	0.93	0.93	0.93	1.36	1.33	1.39
Max Latency	0.91	0.93	0.95	0.90	0.80	0.76	0.88	0.93	0.84	0.84	0.84	0.91	0.96	1.39	1.25	1.64
Average BW	1.06	1.06	1.06	1.08	1.05	1.05	1.07	1.06	1.15	1.04	1.13	1.09	1.10	0.87	0.91	0.86

(b) EEEP credits=3, EEEP credit threshold=3 for the Blitters NIs

Figure 6.8: Performance variation when EEEP is activated in a Spidergon NoC-based SoC

the case where only the link level credit-based flow control is used. Figure 6.8a and Figure 6.8b depict the simulation results.

The horizontal axe represents the processing engines in the platform with their measured values for average latency, maximum latency, and average bandwidth. The vertical axe represents the variation of the bandwidth and latencies (average and maximum) between the case when only the link-level flow control is used, and the case when EEEP is enabled for the blitters traffic. The horizontal red line in the figures show the reference when only the link-level flow control is used. Let us take the case of CPU0 in figure 6.8a as an example. Its average latency is 0.94, which means that its average latency has been decreased by 6% when EEEP is enabled for BE traffic. Its average bandwidth is 1.06, which means that its average bandwidth has been decreased by 6% when EEEP is enabled.

When the EEEP threshold is 1, EEEP reduces the average latency of the *guaranteed service* traffic by 8% on average and the maximum latency by 14% on average, while increasing the bandwidth by 8% on average. This performance improvement is done while maintaining the average bandwidth of the *best effort* traffic. The performance improvement of the GS traffic is still guaranteed when the EEEP threshold is 3. However, the BE traffic bandwidth is no longer guaranteed (loss of 12% on average). This is due to increasing average latency of the BE traffic as it must wait for all EEEP credits before issuing any request packet.

6.3.3 EEEP in a 2DMesh NoC-based SoC

We keep here the configuration of the memory system as shown before, and we change the topology of the network to 4x4 2DMesh. Figure 6.9 shows the simulation platform. Here also we give the lowest priority to the blitter traffic in the network as well as in the memory controller.

Figure 6.10a and Figure 6.10b show that EEEP decreases the average latency and the maximum latency of the GS traffic by 8% and 20% on average when the credit threshold is 1, and by 7% and 17% respectively when the credit threshold is 3. It also increases the GS traffic bandwidth by 8% and 6% when the credit threshold is 1 and 3 respectively. Similarly to the Spidergon case, the threshold variation does impact the bandwidth of the BE effort traffic. The BE traffic bandwidth is only maintained when the credit threshold is 1.

6.3.4 EEEP in an irregular NoC-based SoC

Now we change the *requests queue* size in the memory controller to 20 as the number of initiators has been decreased. The system is mapped on an irregular network, which is shown in Figure 6.11. We also keep the service level for GS traffic, and BE traffic in the network and the memory controller.

Figure 6.12a and Figure 6.12b show the performance variation for all masters. When EEEP is enabled with a threshold of 1, it decreases the average and the maximum latency of the GS traffic by 7% and 17% on average and increases the average bandwidth by 8%. When

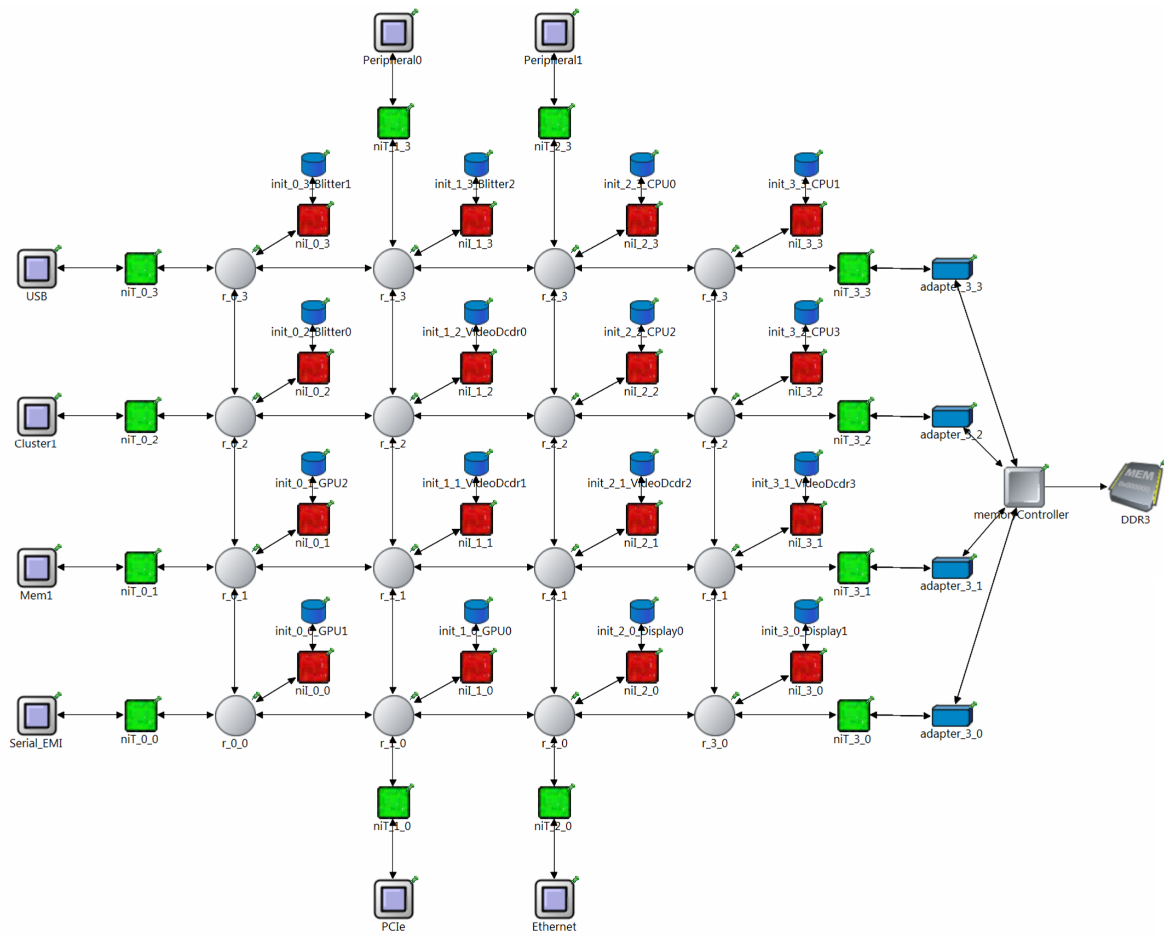
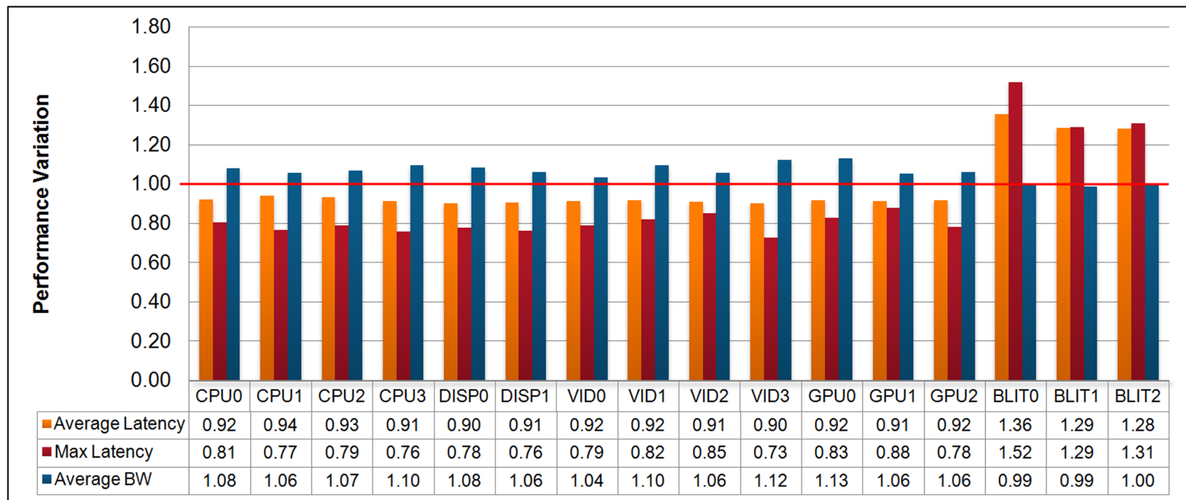
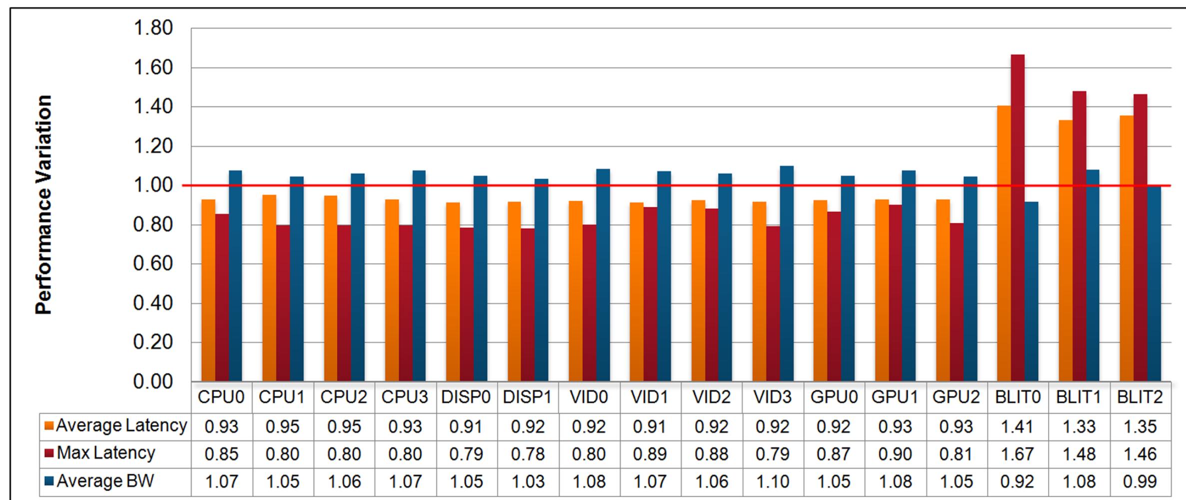


Figure 6.9: 2DMesh NoC-based simulation platform (XY routing)



(a) EEEP credits=3, EEEP credit threshold=1 for the Blitters NIs



(b) EEEP credits=3, EEEP credit threshold=3 for the Blitters NIs

Figure 6.10: Performace variation when EEEP is activated in a 2DMesh NoC-based SoC

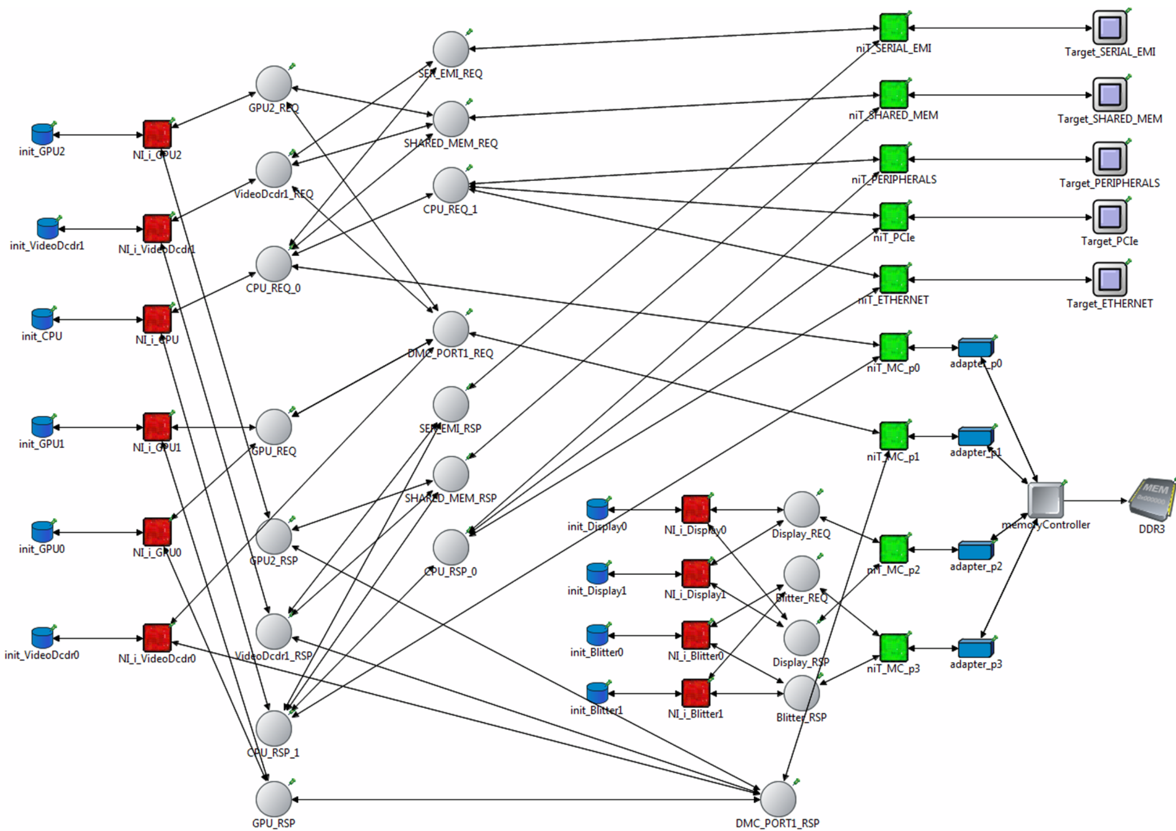
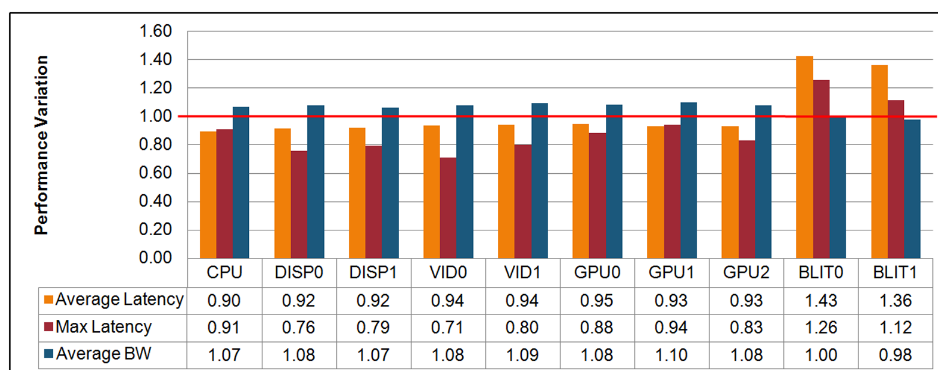
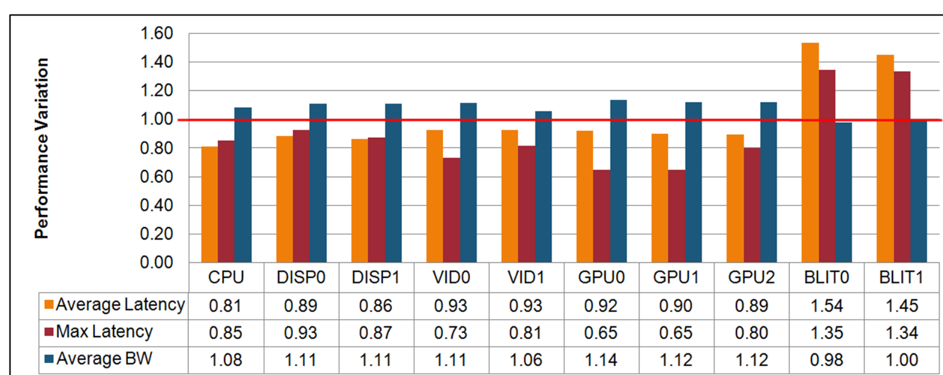


Figure 6.11: irregular NoC-based simulation platform (source routing)



(a) EEEP credits=3, EEEP credit threshold=1 for the Blitters NIs



(b) EEEP credits=3, EEEP credit threshold=3 for the Blitters NIs

Figure 6.12: Performace variation when EEEP is activated in an irregular NoC-based SoC

the threshold is 3, the average and maximum latency are decreased by 11% and 21%, and the average bandwidth is increased by 11%. Contrary to the previous cases, the threshold variation does not impact the bandwidth of the BE effort traffic.

6.3.5 Analysis

We have shown that our *Extreme End-to-End Protocol* improves the performance of GS traffic in several NoC topologies. The average bandwidth of the BE traffic depends of two factors, the EEEP credit threshold, and the number of hops between the blitters NIs and the memory controller. In the Spidergon and the Mesh NoC platforms, there are several routers separating the blitters NIs from the memory subsystem. Increasing the credit threshold means that the blitters requests have to spend more time in the blitters NIs, waiting for the EEEP credits, to be sent in a bursty way. This stalling time does not influence the GS traffic, however it could have an important impact on the BE traffic bandwidth when the number of hops between the NIs and the memory controller increases. The bigger the number of hops on the BE traffic path, the less the BE traffic shape is peaky when it arrives at the memory controller. This is due to the fact that BE traffic has the lowest service level in the network, so it could be split at each

arbitration stage in the routers. To summarize, cumulating the EEEP credits in order to send bursts of requests to the memory controller does not maintain the bandwidth of the BE traffic when the number of hops on its path to the memory increases.

In an irregular topology, we have the choice to isolate the path of the BE traffic from the GS traffic, minimizing thus the number of hops between the masters and the memory controller. This is the case of the third simulation platform where the blitter traffic accesses the memory controller through a devoted router. Here the BE traffic shape at the blitter NIs level and the memory controller NI is almost the same. Consequently, the average bandwidth of the BE traffic is still maintained when the credit threshold increases.

6.4 Conclusion

In this chapter, we first evaluated a memory subsystem made up of a multi-port memory controller with a model for DDR3-800 SDRAM. We verified that the memory controller back-end accurately simulates all memory device latencies. We tested then the whole memory system and showed how the port priority can impact the bandwidth and latency of the requests coming through it.

We evaluated then the performance of our novel *extreme end-to-end protocol* in MPSoC platforms based-on three different topologies of network-on-chip. We proved that EEEP improves the performance of the *guaranteed service* traffic (GS), while maintaining the average bandwidth of *best effort* traffic. The simulation results show that EEEP reduces the average latency of the GS traffic by 8% on average (14% at best), and increases its average bandwidth by 8% on average (11% at best). These results prove that EEEP can guarantee the services for high-priority traffic in any network-on-chip topology.

EEEP is the first end-to-end protocol that deals the memory access with a *system approach,* and uses information about the memory subsystem status in the traffic injection policy at the master network interface level.

CHAPTER 7

Conclusion and Perspectives

THE REQUIREMENTS of MPSoCs for high bandwidth and low latency makes the access to the external DDR SDRAM become a bottleneck. The increasing number of processing units in these systems, in addition to the multi-threading technique used nowadays, increases the contention on the main memory system and demands memory systems with more complex architecture and higher performance. Recognizing the importance of high performance off-chip SDRAM communication as a key to a successful system design, we have focused on the configurability of the memory controller architecture, and proposed a novel protocol for DDR SDRAM access through networks-on-chip. Here are presented a brief summary of the entire dissertation and a list of some potential future directions of the work.

7.1 Conclusion

The task of a memory controller is complex because it has not only to obey all SDRAM timing constraints to provide correct functionality, but also to satisfy the initiators requirements in terms of bandwidth and latency. This puts a lot of constraints on the design and makes the architecture exploration of the memory controller very difficult. From a system perspective, the impact of the memory controller architecture on the memory subsystem performance, and consequently on the system performance, is very important.

Advanced memory controllers and scheduling policies are presented in [2; 10; 47; 56; 57; 84]. Even if some of these designs present configurable memory controllers, the architecture exploration is restricted to limited sets of parameters such as FIFOs depth, data bus size, QoS level and bandwidth distribution. Moreover, none of the previous work presents a totally configurable architecture to give the designer the liberty of exploring and adapting the memory controller architecture. Though, the exploration of the memory controller architecture is essential to measure its impact on the overall system performance.

Being able to explore the architecture of the memory controller and its arbitration algorithms is essential to find an optimized architecture. This emphasizes the importance of having a memory controller with a flexible and configurable architecture. We introduced in Chapter [3] our design of a *totally customizable memory controller* based-on fully configurable building components. This design is a high-level abstraction and cycle approximate model, it can accurately simulate the memory access delays during a normal operating regime. Our components library covers both parts of the memory controller, i.e. the front-end and the back-end. The modelled front-end reflects all delay cycles that are due to the buffering elements number, buffers depth, number of arbiters and scheduling policies. Moreover, the front-end building components are easy to interface with each other, which gives the designer of the memory system a high degree of freedom in designing and exploring the memory controller architecture. We show at the end of Chapter [3] three front-end architectures modelled with our building components. Our customizable architecture is not restricted to front-end part, we also introduce a back-end model, which is DDR3 SDRAM technology com-

patible, and respects all DDR3 SDRAM timing constraints. It can be easily adapted to support previous generations of DDR SDRAM, i.e. DDR1 and DDR2. These building components have been designed to build all architectures of industrial memory controllers we came across.

The continuity of the guaranteed service between the NoC and the memory subsystem can only be ensured by the joint use of architectural and protocol mechanisms. However, these mechanisms remained to be defined in the VLSI context within its constraints in terms of area and power consumption. Many networks-on-chip provide guaranteed service to traffic classes [24; 64; 7; 16; 30; 54; 6]. A few flow controllers and arbitration schemes take into consideration the specificity of the SDRAM as a target [13; 41; 40; 72]. However, these solutions predict the state of the SDRAM, and require heavy arbitration schemes in the routers. None of them use information on the real memory state neither within its arbitration algorithms nor within the flow control. We should know what information the memory controller has to share with the NoC in order to enhance the network performance within the process of SDRAM request scheduling. We introduce in Chapter [4] the *Extreme End-to-End Protocol* (EEEP) as new flow control protocol between the network and the memory controller. EEEP should be used for *best effort* (BE) traffic in addition to a link-level flow control. By controlling the injection of the BE traffic in the network, EEEP increases the performance of *guaranteed service* (GS) traffic in terms of bandwidth and latency, while maintaining the average bandwidth of the BE traffic. This flow control protocol handles the SDRAM access within a *system* approach by considering the memory controller status before injecting request packets in the network. EEEP requires neither additional queues nor counters in the slave network interface, because it is based on the available slots in the *request queue* of the memory controller front-end. The novelty of this protocol consists in exploiting information coming from the memory controller within the quality of service in the network-on-chip. Unlike other end-to-end protocols, EEEP crosses the boundary of the network and extends the quality of service to cover both network-on-chip and memory subsystem. We evaluated the performance of our novel protocol in multimedia SoCs based-on three different topologies of network-on-chip. We proved that EEEP improves the performance of the GS traffic, while maintaining the average bandwidth of BE traffic. The simulation results show that EEEP reduces the average latency of the GS traffic by 8% on average (14% at best), and increases its average bandwidth by 8% on average (11% at best).

7.2 Future work directions

This section discusses interesting future work and open issues in the context of this work.

7.2.1 3D stacking - wide I/O memories

3D integration enables stacking SDRAM on top of one or more logic layers and connecting them with vertical wires called through-silicon-vias (TSVs) [29], thus removing the need to go off-chip to access the memory. with the TSV technology, the number of connections to the SDRAM can significantly increase. Removing the pin constraint has many benefits for memory efficiency, since sharing wires between memory banks can be reduced (or removed).

Interesting future work involves investigating the benefits of the 3D stacking to the increase of signals on the memory interface. The impact of 3D integration may go well beyond the memory devices themselves and change the architecture of contemporary systems. Increasing the number of connections to memory enables wider memory interfaces and higher peak bandwidths.

We believe that extending our *customizable architecture of memory controller* to model any future architecture of memory system is important future work. However, we still have to define how to adapt the existing building components and what kind of new components to introduce.

7.2.2 More memory system information exploitation

We have shown earlier that exploiting the memory system status can help to access the shared memory efficiently, and to extend the *quality of service* between the network-on-chip and the memory subsystem.

Most memory controller architectures contain a *request queue* to schedule the requests efficiently. We decided to exploit the *request queue* occupancy rate in the injection policy of the *best effort* traffic in the network to guarantee and improve the services for the *guaranteed service* traffic. However, we believe that other information on the memory status can be used by the network-on-chip to improve and extend the network services in case of shared memory access. The future work in this direction is to specify what information about the memory system should be shared with the network, and how the network will use these pieces of information to better access the main and shared memory.

7.2.3 *Extreme End-to-End Protocol* evolution

EEEP only exploits the *request queue* occupancy rate to control the injection of the best effort traffic requests in the network. We use the granularity of the packet for the EEEP, which corresponds to one memory request. We are aware that the longer the burst length of a request, the more the request uses the shared buffers in the memory controller. And this is available for both paths, i.e. for write-data buffers and read-data buffer. One of the future work directions is to determine how EEEP can include the occupancy rate of the shared data buffer in the memory controller front-end.

We developed EEEP under the assumption that the memory controller front-end has only one *requests queue*. However, few industrial memory controllers can have several *requests queues*, which is the case of the memory controller *Gamma* we have shown in section[3.7.3]. This memory controller buffers the requests in several queues according to the memory bank they are addressed to. Nevertheless, we wait for the future work to define how to adapt such a protocol in order to cover all memory controller architectures.

Bibliography

[1] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis. CODES+ISSS '07*, pages 251–256, 2007.

[2] Benny Akesson. *Predictable And Composable System-on-Chip Memory Controllers*. PhD thesis, 2010.

[3] ARM. Primecell dynamic memory controller pl340. Technical reference manual, 2007.

[4] Arteris. A comparison of network-on-chip and busses - white paper, 2005.

[5] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *Micro, IEEE*, 22(5):16–23, 2002.

[6] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous noc architecture providing low latency service and its multi-level design framework. In *Proceedings of Asynchronous Circuits and Systems. ASYNC'05*, pages 54 – 63, 2005.

[7] T. Bjerregaard and J. Sparso. Implementation of guaranteed services in the mango clockless network-on-chip. In *Proceedings of Computers and Digital Techniques*, volume 153, pages 217–229, July 2006.

[8] Paul Bogdan and Radu Marculescu. Workload characterization and its impact on multicore platform design. In *CODES+ISSS*, pages 231–240, 2010.

[9] P. Bogdan and R. Marculescu. Non-stationary traffic analysis and its implications on multicore platform design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30, April 2011.

[10] Artur Burchard, Ewa Hekstra-Nowacka, and Atul Chauhan. A real-time streaming memory controller. In *Proceedings of the conference on Design, Automation and Test in Europe. DATE '05*, pages 20–25, 2005.

[11] Everton Carara, Gabriel Marchesan Almeida, Gilles Sassatelli, and Fernando Ghem Moraes. Achieving composability in NoC-based MPSoCs through QoS management at software level. In *Proceedings of the conference on Design, Automation and Test in Europe. DATE '11*, March 2011.

[12] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: building a smarter memory controller. *Proceedings of the Fifth International Symposium On High-Performance Computer Architecture*, pages 70–79, Jan 1999.

[13] Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proceedings of the conference on Design, Automation and Test in Europe, DATE'10*, pages 39–44, 2010.

[14] Roopesh Chuggani, V. Laxmi, M. S. Gaur, Pankaj Khandelwal, and Prateek Bansal. A traffic model for concurrent core tasks in networks-on-chip. In *Proceedings of the Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)*, 2011.

[15] Nicola Concer, Luciano Bononi, Michael Soulie, Riccardo Locatelli, and Luca P. Carloni. Ctc: An end-to-end flow control protocol for multi-core systems-on-chip. In *Proceedings of the Third IEEE International Symposium on Networks-on-Chip, NoCs'09*, pages 193–202, 2009.

[16] Marcello Coppola, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia, and Lorenzo Pieralisi. *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*. CRC Press, Inc., Boca Raton, FL, USA, 2008.

[17] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 222–233, 1999.

[18] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor socs. In *Proceedings of the 21st International Conference on Computer Design*, pages 536–539, 2003.

[19] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, 2001.

[20] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[21] Masoud Daneshtalab, Masoumeh Ebrahimi, pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. A low-latency and memory-efficient on-chip network. In *Proceedings of the Fourth*

ACM/IEEE International Symposium on Networks-on-Chip, NoCs'10, pages 99–106, May 2010.

[22] Jonas Diemer and Rolf Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *Proceedings of the Fourth ACM/IEEE International Symposium on Networks-on-Chip, NoCs'10*, pages 155–162, May 2010.

[23] J. Diemer, R. Ernst, and M. Kauschke. Efficient throughput-guarantees for latency-sensitive networks-on-chip. In *Proceedings of ASP-DAC*, pages 529–534, 2010.

[24] R. Dobkin, R. Ginosar, and I. Cidon. Qnoc asynchronous router with dynamic virtual channel allocation. In *Proceedings of the First International Symposium on Networks-on-Chip, NoCs'07*, pages 218–218, May 2007.

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

[26] F. Feliciian and S.B. Furber. An asynchronous on-chip network router with quality-of-service (qos) support. In *Proceedings. IEEE International SOC Conference*, pages 274 – 277, 2004.

[27] Sahar Foroutan. *An Analytical Method for Performance Evaluation of Networks-on-Chip*. PhD thesis, CEA-LETI, September 2010.

[28] Om Prakash Gangwal, Andrei Radulescu, Kees Goossens, Santiago Gonzalez Pestana, and Edwin Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the Æthereal network on chip. In *Dynamic and Robust Streaming In And Between Connected Consumer Electronics Devices, Philips Research Book Series*, pages 1–36, 2005.

[29] Philip Garrou, Christopher Bower, and Peter Ramm. *Handbook of 3D Integration: Technology and Applications of 3D Integrated Circuits*. Wiley-VCH, 2008.

[30] Kees Goossens and Andreas Hansson. The aethereal network on chip after ten years: Goals, evolution,lessons, and future. In *Proceedings of the Design Automation Conference, DAC'10*, June 2010.

[31] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, Sept-Oct 2005.

[32] Boris Grot, Stephen W. Keckler, and Onur Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, Micro-42*, pages 268–279, 2009.

[33] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, Automation and Test in Europe, DATE '00*, pages 250–256, 2000.

[34] S. Heithecker and R. Ernst. Traffic shaping for an fpga based sdram controller with complex qos requirements. In *Proceedings of the Design Automation Conference, DAC'05*, pages 575–578, June 2005.

[35] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2006.

[36] Saied Hosseini-Khayat and Andreas D. Bovopoulos. A simple and efficient bus management scheme that supports continuous streams. *ACM Trans. Comput. Syst.*, 13(2):122–140, 1995.

[37] International Technology Roadmap for Semiconductors, 2010.

[38] E. Ipek, O. Mutlu, J.F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of 35th International Symposium on Computer Architecture, ISCA '08*, pages 39–50, June 2008.

[39] F. Jafari, Zhonghai Lu, A. Jantsch, and M.H. Yaghmaee. Optimal regulation of traffic flows in networks-on-chip. In *Proceedings of the conference on Design, Automation and Test in Europe, DATE'10*, pages 1621–1624, 2010.

[40] Wooyoung Jang and D.Z. Pan. An sdram-aware router for networks-on-chip. In *Proceedings of the Design Automation Conference, DAC'09*, pages 800–805, July 2009.

[41] Wooyoung Jang and D.Z. Pan. Application-aware noc design for efficient sdram access. In *Proceedings of the Design Automation Conference, DAC'10*, pages 453–456, 2010.

[42] S. Jayadevappa, R. Shankar, and I. Mahgoub. A comparative study of modelling at different levels of abstraction in system on chip designs: a case study. In *Proceedings. IEEE Computer society Annual Symposium on VLSI*, pages 52–58, 2004.

[43] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *Micro, IEEE*, 22(5):36–45, sep/oct 2002.

[44] Dongki Kim, Sungjoo Yoo, and Sunggu Lee. A network congestion-aware memory controller. In *Proceedings of the Fourth ACM/IEEE International Symposium on Networks-on-Chip, NoCs'10*, pages 257–264, May 2010.

[45] T.S.R. Kumar, C.P. Ravikumar, and R. Govindarajan. Memory architecture exploration framework for cache based embedded soc. In *Proceedings of the 21st International Conference on VLSI Design*, pages 553 –559, jan. 2008.

[46] V. Laxmi, R. Chuggani, M.S. Gaur, P. Khandelwal, and P. Bansal. Traffic characterization for multicasting in noc. In *Proceedings of NORCHIP*, 2010.

[47] Hyuk-Jun Lee and Eui-Young Chung. Scalable qos-aware memory controller for high-bandwidth packet memory. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 16, pages 289–301, March 2008.

[48] Kun-Bin Lee, Tzu-Chieh Lin, and Chein-Wei Jen. An efficient quality-aware memory controller for multimedia platform soc. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):620–633, May 2005.

[49] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *SIGARCH Comput. Archit. News*, volume 36, pages 89–100, 2008.

[50] A. Lines. Asynchronous interconnect for synchronous soc design. *Micro, IEEE*, 24(1):32–41, 2004.

[51] Dake Liu, Daniel Wiklund, Erik Svensson, Olle Seger, and Sumant Sathe. Socbus: The solution of high communication bandwidth on chip and short TTM, 2003.

[52] C. Macian, S. Dharmapurikar, and J. Lockwood. Beyond performance: secure and fair memory management for multiple systems on a chip. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 348–351, 2003.

[53] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual channels in networks on chip: Implementation and evaluation on hermes noc. In *Proceedings of the 18th Symposium on Integrated Circuits and Systems Design*, pages 178–183, 2005.

[54] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Date'04*, volume 2, pages 890–895, 2004.

[55] Fernando Gehm Moraes, Ney Laert Vilar Calazans, Aline Vieira de Mello, Leandro Heleno Möller, and Luciano Copello Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. Technical report, 2003.

[56] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-40*, pages 146–160, Dec. 2007.

[57] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th International Symposium on Computer Architecture, ISCA '08*, pages 63–74, 2008.

[58] Chitra Natarajan, Bruce Christenson, and Fayé Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Proceedings of the 3rd workshop on Memory performance issues, WMPI'04*, pages 80–87, 2004.

[59] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-39*, pages 208–222, Dec. 2006.

[60] U.Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. In *Proceedings of the Design Automation Conference, DAC'06*, pages 839–844, 2006.

[61] OMNeT++ API Reference, http://www.omnetpp.org/doc/omnetpp41/api/, 2011.

[62] OMNeT++, http://www.omnetpp.org/documentation, 2011.

[63] F. Paganini, J. Doyle, and S. Low. Scalable laws for stable network congestion control. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 1, pages 185–190, 2001.

[64] Ivan Miro Panades. *Design and Implementation of a Network-on-Chip with Guaranteed Service*. PhD thesis, Pierre et Marie Curie University - Paris VI, May 2008.

[65] John Probell. Routing congestion: The growing cost of wires in systems-on-chip. Technical report, Arteris, 2011.

[66] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4–17, jan. 2005.

[67] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture, ISCA '00*, pages 128–138, 2000.

[68] I. Saastamoinen, D. Siguenza-Tortosa, and J. Nurmi. Interconnect IP node for future system-on-chip designs. In *Proceedings of The First IEEE International Workshop on Electronic Design, Test and Applications*, pages 116–120, 2002.

[69] Samsung. Samsung ddr3-800 K4B4G0446A. Technical report, 2010.

[70] A. Scherrer. *Analyses statistiques des communications sur puce*. PhD thesis, 2006.

[71] Jun Shao and B.T. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture. HPCA'07*, pages 285–294, 2007.

[72] Sonics. Sonics sx smart interconnect solution. Datasheet, 2008.

[73] Krishnan Srinivasan and Erno Salminen. A methodology for performance analysis of network-on-chip architectures for video socs. *OCP IP*, April 2009.

[74] Krishnan Srinivasan and Erno Salminen. A memory subsystem model for evaluating network-on-chip performance, White Paper. 2010.

[75] Double data rate (ddr) sdram specification, May 2002.

[76] Double data rate (ddr2) sdram specification, January 2005.

[77] Double data rate (ddr3) sdram specification, April 2008.

[78] TILERA. Tilepro64. Available on line: http://www.tilera.com/sites/default/files/product-briefs/pb019_tilepro64_processor_a_v3.pdf. Technical report, 2010.

[79] W.-D. Weber, J. Chou, I. Swarbrick, and D. Wingard. A quality-of-service mechanism for interconnection networks in system-on-chips. In *Proceedings of the conference on Design, Automation and Test in Europe, DATE'05*, pages 1232–1237, March 2005.

[80] S. Whitty and R. Ernst. A bandwidth optimized sdram controller for the morpheus reconfigurable architecture. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008*, pages 1–8, 2008.

[81] Xiaodong Xian, Weiren Shi, and He Huang. Comparison of OMNET++ and other simulator for WSN simulation. In *Proceedings of the 3rd IEEE Conference on Industrial Electronics and Applications, ICIEA*, pages 1439–1443, 2008.

[82] C.A. Zeferino and A.A. Susin. Socin: a parametric and scalable network-on-chip. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pages 169 – 174, 2003.

[83] Fucen Zeng, Lin Qiao, Mingliang Liu, and Zhizhong Tang. A novel memory subsystem evaluation framework for chip multiprocessors. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, pages 231–238, 2010.

[84] Hongzhong Zheng, Jiang Lin, Zhao Zhang, and Zhichun Zhu. Memory access scheduling schemes for systems with multi-core processors. In *Proceedings of the 37th International Conference on Parallel Processing, ICPP'08*, pages 406–413, Sept. 2008.

[85] Zhichun Zhu and Zhao Zhang. A performance comparison of dram memory system optimizations for smt processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA-11*, pages 213–224, Feb. 2005.

[86] Zhichun Zhu, Zhao Zhang, and Xiaodong Zhang. Fine-grain priority scheduling on multi-channel memory systems. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 107–116, Feb. 2002.

[87] Jiayi Zhu, Peilin Liu, and Dajiang Zhou. An sdram controller optimized for high definition video coding application. In *IEEE International Symposiumon Circuits and Systems. ISCAS'08*, pages 3518–3521, May 2008.

APPENDIX A

Problem Definition: *Simulation Platform*

A.1 Spidergon STNoC building blocks

The Spidergon STNoC contains four different types of building blocks, which are:

- The network interface (NI), provides a hardware access point to external IP or processor cores and the necessary hardware to implement a set of communication primitives and low-level platform services.
- The router, responsible for implementing the network layer of Spidergon STNoC protocol stack. It must ensure a reliable packet transfer trough the on-chip network, according to a proper QoS policy. From a very high-level perspective, a router is based on a crossbar switch with a given number of input and output ports.
- The network plug switch (NPS), used to aggregate several NIs for accessing the network. This component enables the connection of several network interfaces to the NI port of a router.
- The physical link implements the physical layer of the Spidergon STNoC protocol. It is responsible for connecting routers to each other, and also router to NIs. There are several possible ways of implementing physical links, including combinations of synchronous / asynchronous and serial / parallel links. In fact, the choice of physical link technology involves trade-offs between many issues, such as clock distribution, amount of on-chip wiring, and required chip area.

A.2 Platform composition

Figure A.1 shows a simplified architecture of the simulation platform. It is made up of:

- 4 traffic generators representing two cache controller ports; one DMA and an ARM processor.
- 4 SRAMs and 1 ROM.
- 2 SDRAM DDR subsystems, made up of memory controller and Micorn DDR SDRAM modules [**?**].
- A Spidergon STNoC, composed of two separated and symmetric networks, one for requests and one for responses. Both networks contain 6 routers.

Table A.1: Routing table of both request and response networks

	DMC 0	DMC 1	SRAM 0	SRAM 1	SRAM 2	SRAM 3	SRAM 4
Cache Ctrl 0	0	0-2	0-1	0-2-3	0-2-3	0-5-4	0-5
Cache Ctrl 1	2-0	2	2-1	2-3	2-3	2-3-4	2-0-5
DMA	1-0	1-2	1-0-5	1-2-3	1-2-3	1-4	1-4-5
Streaming IP	4-1-0	4-1-2	4-1	4-3	4-3	4	4-5

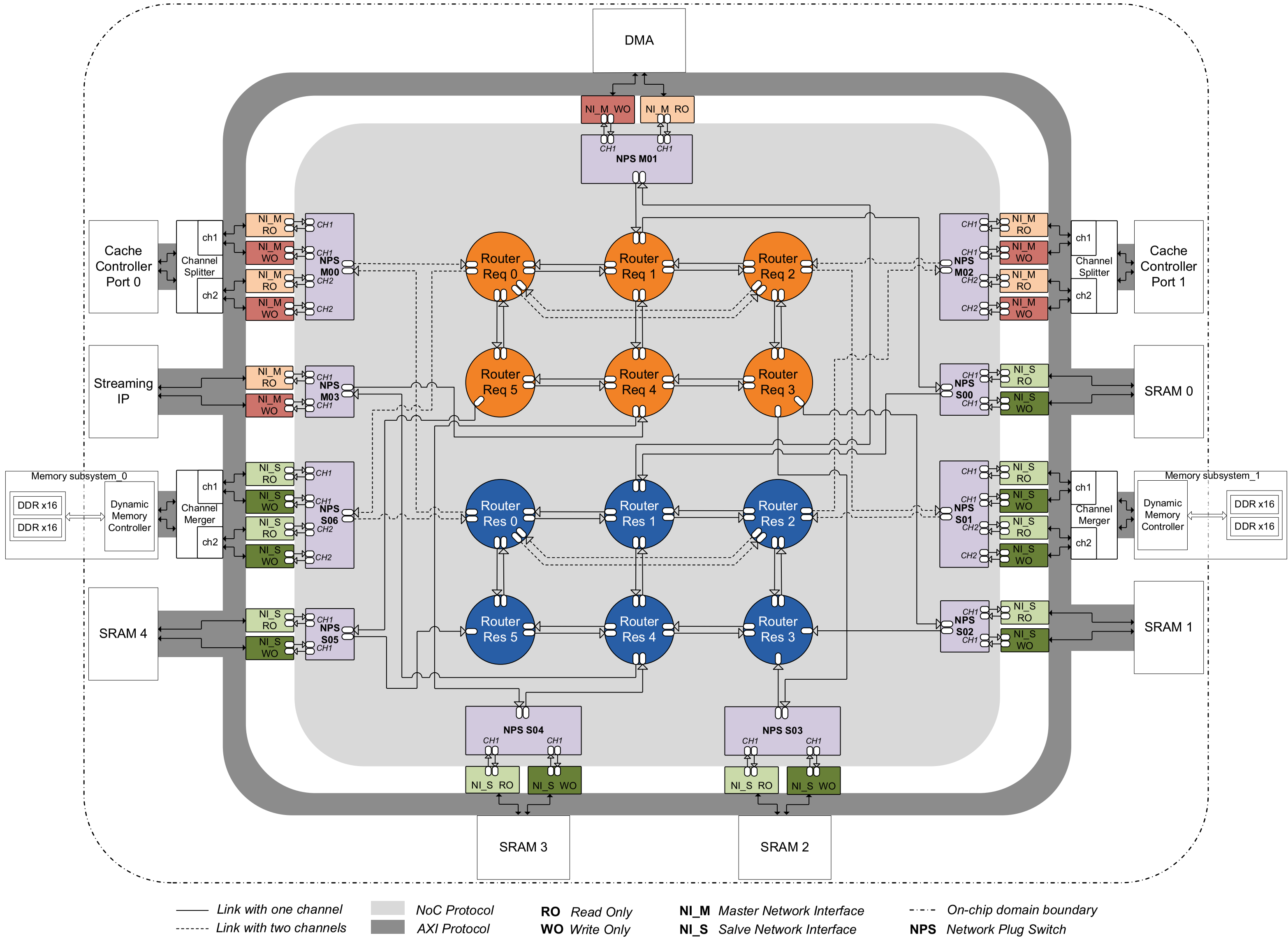


Figure A.1: Simplified architecture of the simulation platform

APPENDIX B

Memory Controller Scheduling Algorithms

Algorithm 10 Highest priority capture

local variables $priority, L, selectedRequestIndex, highestPriority$;
$L \leftarrow$ **genericQueue**$[queueIndex].getLength()$;
$selectedRequestIndex \leftarrow 0$;
$highestPriority \leftarrow 0$;
for $(i = (L-1); i \geq 0; i--)$ **do**
  $priority \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getPriority()$;
  if $(priority \geq highestPriority)$ **then**
    $highestPriority \leftarrow priority$;
    $selectedRequestIndex \leftarrow i$;
  end if
end for
return $selectedRequestIndex$;

Algorithm 11 Row hit opposite direction capture

local variables $bankIndex, row, lastRow, L, selectedRequestIndex$;
$L \leftarrow$ **genericQueue**$[queueIndex].getLength()$;
$selectedRequestIndex \leftarrow 0$;
rowHitOppositeDirection $\leftarrow false$
for $(i = 0; i < L; i++)$ **do**
  $bankIndex \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getBank()$;
  $row \leftarrow$ **genericQueue**$[queueIndex].getRequestPointer(i).getRow()$;
  $lastRow \leftarrow$ **bankStatus**$[bankIndex].getLastRow()$;
  if $(row = lastRow$ **and rowHitSameDirection** $= false)$ **then**
    rowHitOppositeDirection $\leftarrow true$
    $selectedRequestIndex \leftarrow i$;
    $break$;
  end if
end for
return $selectedRequestIndex$;

Algorithm 12 Global system data consistency insertion

```
local variables existingBank, newBank, existingRow, newRow, existingColumn, newColumn;
local variables L, insertionPosition, lowerBoundary;
systemDataConsistencyLimit ← 0;
lowerBoundary ← 0;
L ← genericQueue[queueIndex].getLength();
newBank ← newRequest.getBank();
newRow ← newRequest.getRow();
newColumn ← newRequest.getColumn();
insertionPosition ← L;
if (L = 0) then
    systemDataConsistencyLimit ← 0;
else
    for (i = (L − 1); i ≥ lowerBoundary; i − −) do
        existingBank ← genericQueue[queueIndex].getRequestPointer(i).getBank();
        existingRow ← genericQueue[queueIndex].getRequestPointer(i).getRow();
        existingColumn ← genericQueue[queueIndex].getRequestPointer(i).getColumn();
        if (newBank = existingBank and newRow = existingRow and newColumn = existingColumn) then
            systemDataConsistencyLimit ← i;
            insertionPosition ← (i + 1);
            break;
        end if
    end for
end if
return insertionPosition;
```

Algorithm 13 Priority-based insertion

```
local variables existingPr, newPr, L, insertionPosition, lowerBoundary;
priorityLimit ← 0;
lowerBoundary ← 0;
L ← genericQueue[queueIndex].getLength();
newPr ← newRequest.getPriority();
insertionPosition ← L;
if (L = 0) then
    priorityLimit ← 0;
else
    for (i = lowerBoundary; i < L; i + +) do
        existingPr ← GenericQueue[queueIndex].getRequestIndex(i).getPriority();
        if (newPr > existingPr) then
            priorityLimit ← i;
            if (i = lowerBoundary and lowerBoundary ≠ 0) then
                insertionPosition ← (i + 1); //insert after
            else
                insertionPosition ← (i); //insert before
            end if
        end if
        break;
    end for
end if
return insertionPosition;
```

Algorithm 14 Direction grouping insertion

```
local variables existingdirection, newDirection, L, lowerBoundary;
lowerBoundary ← 0;
L ← genericQueue[queueIndex].getLength()
newOpcode ← newRequest.getDirection();
insertionPosition ← L;
if (L ≠ 0) then
    for (i = lowerBoundary; i < L; i++) do
        existingOpcode ← genericQueue[queueIndex].getRequestPointer(i).getDirection();
        if (newDirection = existingDirection) then
            insertionPosition ← (i + 1);
            break;
        else
            insertionPosition ← (L);
        end if
    end for
end if
return insertionPosition;
```

Algorithm 15 Round-robin scheduling

```
local variables inIndex, selectedInput;
inIndex ← 0;
for (in = 0; in < inputsNumber; in++) do
    inIndex ← (lastSelectedInput + 1 + in);
    if (inIndex > inputsNumber) then
        inIndex ← (inIndex − inputsNumber);
    end if
    if (inputValid[inIndex] = true) then
        selectedInput ← inIndex;
        break;
    end if
end for
lastSelectedInput ← selectedInput;
return selectedInput;
```

Algorithm 16 Initialize least-recently-used

```
for (i = 0; i < inputsNumber; i++) do
    for (j = 0; j < inputsNumber; j++) do
        if (i ≤ j) then
            LRUtab[i][j] ← 0;
        else
            LRUtab[i][j] ← 1;
        end if
    end for
end for
```

Algorithm 17 Least-recently-used scheduling

local variables $sum, max, inputIndex, selectedInput$;
$max \leftarrow 0$
for ($i = 0$; $i <$ **inputsNumber**; $i + +$) **do**
  $sum \leftarrow 0$;
  if (**inputValid**$[in] = true$) **then**
    for ($j = 0$; $j <$ **inputsNumber**; $j + +$) **do**
      $sum \leftarrow (sum +$ **LRUtab**$[i][j])$;
    end for
    if ($sum > max$) **then**
      $max \leftarrow sum$;
      $selectedInput \leftarrow i$;
    end if
  end if
end for
return $selectedInput$;

Algorithm 18 Least-recently-used update

for ($i = 0$; $i <$ **inputsNumber**; $i + +$) **do**
  if ($selectedInput = i$) **then**
    LRUtab$[selectedInput][i] \leftarrow 0$;
  else
    LRUtab$[selectedInput][i] \leftarrow 0$;
    LRUtab$[i][selectedInput] \leftarrow 1$;
  end if
end for

Algorithm 19 Priority scheduling

local variables $inIndex, selectedInput$;
for ($pr =$ **maxPriorityLevel**; $pr \geq$ **minPriorityLevel**; $pr - -$) **do**
  for ($in = 0$; $in <$ **inputsNumber**; $in + +$) **do**
    if (**inputPriorityMatrix**$[pr][in] = 1$) **then**
      $selectedInput \leftarrow in$;
      $break$;
    end if
  end for
end for
return $selectedInput$;

APPENDIX C

List of Publications

Paper:
Khaldon Hassan, Frédéric Pétrot, Riccardo Locatelli and Marcello Coppola,
EEEP: an Extreme End to End flow control Protocol for SDRAM Access Through Networks on Chip,
In the proceedings of the Fifth International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip,
ACM-Jan 2011.

Book contribution:
Khaldon Hassan and Marcello Coppola,
Off-Chip SDRAM Access Through Spidergon STNoC,
VLSI 2010 Annual Symposium,
Springer-Aug 2011.

APPENDIX D

About the Author

Khaldon HASSAN was born in Damascus, Syria in 1982. He received the M.Sc. degree in Microelectronics from Grenoble Institute of Technology, France in 2007. The master project was carried out at STMicroelectronics in Grenoble, France on the topic of a comparative study of STMicroelectronics interconnection systems from the synthesis point of view. In March 2008, Khaldon HASSAN started the journey towards a Ph.D. degree at the Grenoble Institute of Technology in collaboration with STMicroelectronics. He joined Schlumberger in 2011, and works as Electrical Engineer with the Technology Group in Riboud Production Center in Clamart, France.

Abstract

The ongoing advancements in VLSI technology allow System-on-Chip (SoC) to integrate many heterogeneous functions into a single chip, but still demand, because of economical constraints, a single and shared main off-chip SDRAM. Consequently, main memory system design, and more specifically the architecture of the memory controller, has become an increasingly important factor in determining the overall system performance.

Choosing a memory controller design that meets the needs of the whole system is a complex issue. This requires the exploration of the memory controller architecture, and then the validation of each configuration by simulation. Although the architecture exploration of the memory controller is a key to successful system design, state of the art memory controllers are not as flexible as necessary for this task. Even if some of them present a configurable architecture, the exploration is restricted to limited sets of parameters such as queue depth, data bus size, quality-of-service level, and bandwidth distribution.

Several classes of traffic co-exist in real applications, e.g. *best effort* traffic and *guaranteed service* traffic, and access the main memory. Therefore, considering the interaction between the memory subsystem and the interconnection system has become vital in today's SoCs. Many on chip networks provide guaranteed services to traffic classes to satisfy the applications requirements. However, very few studies consider the SDRAM access within a system approach, and take into account the specificity of the SDRAM access as a target in NoC-based SoCs.

This thesis addresses the topic of dynamic access to SDRAM in NoC-based SoCs. We introduce a totally customizable memory controller architecture based on fully configurable building components and design a high level cycle approximate model for it. This enables the exploration of the memory subsystem thanks to the ease of configuration of the memory controller architecture. Because of the discontinuity of services between the network and the memory controller, we also propose within the framework of this thesis an *Extreme End to End* flow control protocol to access the memory device through a multi-port memory controller. The simple yet novel idea is to exploit information about the memory controller status in the NoC. Experimental results show that by controlling the best effort traffic injection in the NoC, our protocol increases the performance of the guaranteed service traffic in terms of bandwidth and latency, while maintaining the average bandwidth of the best effort traffic.

Keywords: memory controller, SDRAM, NoC, MPSoC, performance analysis, traffic classes, end-to-end protocol, modeling.

ISBN 978-2-84813-172-6