

Resource Allocation  
in Time-Division-Multiplexed  
Networks on Chip



# Resource Allocation in Time-Division-Multiplexed Networks on Chip

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor  
aan de Technische Universiteit Delft,  
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,  
voorzitter van het College voor Promoties,  
in het openbaar te verdedigen

op dinsdag 24 april 2012 om 10.00 uur

door

Radu ȘTEFAN

Master of Science  
Universitatea Transilvania Brașov  
geboren te Brașov, Roemenië

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr. K.G.W.Goossens

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter  
Prof. dr. K.G.W. Goossens, promotor  
Prof. dr. H.J. Sips  
Prof. dr. K.L.M. Bertels  
Prof. dr. R.H.J.M. Otten  
Prof. dr. G.J.M. Smit  
Dr. J. Flich

Technische Universiteit Delft  
Technische Universiteit Delft  
Technische Universiteit Delft  
Technische Universiteit Delft  
Technische Universiteit Eindhoven  
Universiteit Twente  
Universidad Politécnica de Valencia

ISBN: 9789072298270

*This dissertation is dedicated to my family,  
for all their understanding and support over the years.*



# Resource Allocation in Time-Division-Multiplexed Networks on Chip

Radu STEFAN

## Abstract

---

One of the challenges of engineering is to make the best possible use of the available resources, or in other words allocating the resources in such a way as to maximize the overall profit. In the context of networks on chip the resources are represented by the communication bandwidth and the final profit is the performance of an application supported by the network on chip.

In this thesis we focus on networks on chip providing guaranteed performance, i.e. guaranteeing for each application the delivery of a requested bandwidth. In these networks, hardware resources are allocated and assigned to each application for its entire lifetime. We discuss several solutions for delivering the allocated bandwidth, and we propose models which allow us to evaluate the performance of these solutions. Starting from a general, rate-based allocation model we gradually add more architectural restrictions that lower the implementation cost, but at the same time sacrifice some performance.

NoCs with allocation based on discrete rates are very common and include priority-based, TDM, SDM, FDM, and other NoCs. They all partition the bandwidth available on the network links into discrete units. In the case of TDM NoCs these units are called time slots. The problem of resource allocation in TDM NoCs consists of finding paths through the network between the nodes that wish to communicate, and selecting along these paths a set of free time slots that is sufficiently large to fulfill the application requirements. After allocation the bandwidth is guaranteed.

In this thesis, we propose, implement and evaluate allocation algorithms for all the proposed performance models. Particular effort is dedicated to allocation algorithms for the contention-free routing model, a restrictive, but low-cost form of TDM where allocation is particularly challenging. Our allocation algorithms deal both with spatial allocation, i.e., the selection of a specific path out of the available paths through the network, and temporal allocation,

i.e., along the time axis. The latter is used for optimizing bandwidth usage and latency which we will both discuss in depth. We propose two algorithms for the allocation of slots in the time domain, both of which we show to be optimal.

We also demonstrate how the TDM schedule can be computed at run time, with low computational requirements. We demonstrate a system performing run-time allocation in FPGA and we implement hardware acceleration for the more expensive operations used by the allocation algorithm.

We propose a synthesizable NoC implementation based on the contention-free-routing model, called dAElite. Our proposal uses existing design flows but has better performance and reduced hardware cost. The network supports some of the less restrictive models that we have previously introduced thus allowing a better allocation of resources.

Finally, we present how the communication requests of the IPs are handled by the interconnect. We propose optimizations such as write coalescing and latency hiding techniques at the interface between IPs and the NoC and we demonstrate the performance benefits of the proposed approach in real applications.

The main conclusions of this thesis are that, compared to an ideal rate-based NoC offering guaranteed bandwidth, introducing fixed discrete allocation units causes a performance loss of 18% while using headers loses another 15%, under the considered, realistic scenarios. Other factors, such as topology, in-order delivery, etc. cause only a minor performance loss. We find Æthereal to lose 46% compared to an ideal rate-based network, while the dAElite network introduced here loses less than 26% and is at the same time less expensive to implement.



# Acknowledgments

I started my PhD under the supervision of Prof. Stamatis Vassiliadis who unfortunately fell ill soon afterwards and departed from among us one year later. Despite our very few encounters he had a strong influence on my development in the first years of my PhD. I would like to thank him as well as Prof. Georgi Gaydadjiev, Prof. Koen Bertels who were in charge of my supervision during that time of transition.

I would like to thank my PhD and postdoctoral colleagues for their support and exchange of ideas. I hope we will remain in close contact in the future.

I would like to thank my supervisor for his patience in reviewing the many iterations of this thesis.

Last but not least, I would like to thank you, the reader for the interest you are showing in this work.



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgments</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Design trends . . . . .	1
1.2 Overview of chip interconnect solutions . . . . .	3
1.3 Networks on Chip . . . . .	6
1.4 NoC design . . . . .	9
1.4.1 The contention-free routing model . . . . .	11
1.4.2 Thesis contribution . . . . .	12
1.5 Thesis Overview . . . . .	13
<b>2 Theoretical bounds on allocatable capacity</b>	<b>17</b>
2.1 General assumptions . . . . .	20
2.2 Architectural restrictions . . . . .	21
2.2.1 Ideal versus real topologies . . . . .	23
2.2.2 Continuous versus discrete bandwidth division . . . . .	24
2.2.3 A generic link-sharing mechanism versus contention-free routing . . . . .	27
2.2.4 Multiple paths versus a single path . . . . .	28
2.2.5 In-order versus out-of-order delivery . . . . .	29
2.2.6 Minimal versus non-minimal routing . . . . .	30
2.2.7 Header overhead . . . . .	30
2.3 Allocation methods . . . . .	31
2.3.1 Linear Programming . . . . .	33
2.3.2 Iterative, single-channel allocation using the flow algorithm . . . . .	36
2.3.3 Allocation using graph splitting in the ILP and flow methods . . . . .	38

2.3.4	Single-path exhaustive search . . . . .	40
2.3.5	Multi-path using repeated single-path exhaustive search . . . . .	42
2.4	Performance comparison of the proposed models . . . . .	43
2.4.1	Small networks with random traffic . . . . .	47
2.4.2	Uniform (random) traffic . . . . .	55
2.4.3	Permutation traffic in small networks . . . . .	57
2.4.4	Permutation traffic in large networks . . . . .	61
2.4.5	Real applications . . . . .	64
2.4.6	Performance of different topology sizes under random traffic . . . . .	69
2.4.7	Summary of experiments . . . . .	69
2.5	Related Work . . . . .	71
2.6	Conclusions and Future Work . . . . .	73
<b>3</b>	<b>Single and multi-path allocation algorithms</b>	<b>77</b>
3.1	Exhaustive search Single Path . . . . .	80
3.1.1	Proposed exhaustive path search . . . . .	82
3.2	Bandwidth allocation using Network Flow . . . . .	88
3.3	Flow algorithm for the contention-free slot model . . . . .	93
3.4	Path selection for in-order delivery . . . . .	95
3.4.1	Proof of optimality for in-order path selection . . . . .	96
3.5	Iterative maximum-bandwidth-search multi-path . . . . .	100
3.6	Related Work . . . . .	102
3.7	Conclusions and future work . . . . .	104
<b>4</b>	<b>Latency and slot selection</b>	<b>105</b>
4.1	The effect of slot selection . . . . .	106
4.2	Problem formulation . . . . .	116
4.3	The previously used algorithm . . . . .	116
4.4	Proposed algorithm for the slot selection problem . . . . .	119
4.5	Enhanced formulation of the latency problem . . . . .	124
4.5.1	Slot allocation algorithm for the enhanced formulation . . . . .	124
4.5.2	Proof of optimality of the algorithm for the enhanced formulation . . . . .	131
4.6	Algorithm complexity . . . . .	132
4.7	Experimental results . . . . .	133
4.8	Related Work . . . . .	144
4.9	Conclusions and future directions of research . . . . .	144
<b>5</b>	<b>Online allocation</b>	<b>147</b>

5.1	Channel allocation in Circuit switching networks . . . . .	148
5.2	Data structures . . . . .	149
5.3	The path finding algorithm . . . . .	150
5.4	Computation of the available bandwidth . . . . .	151
5.4.1	Exact bandwidth computation in a software loop . . .	152
5.4.2	Bandwidth approximation using lookup tables . . . .	154
5.4.3	A hardware accelerator for bandwidth computation . .	155
5.5	A blueprint for a hardware allocator . . . . .	156
5.6	Experimental results . . . . .	159
5.6.1	Memory requirements . . . . .	167
5.6.2	The speed of the algorithm completely implemented in hardware . . . . .	168
5.7	Related Work . . . . .	169
5.8	Conclusions and future directions of research . . . . .	171
<b>6</b>	<b>dAElite NoC Hardware implementation</b>	<b>173</b>
6.1	Hardware implementation . . . . .	174
6.1.1	System overview . . . . .	174
6.1.2	Configuration infrastructure . . . . .	175
6.1.3	Routers . . . . .	176
6.1.4	Network Interfaces . . . . .	177
6.2	Network configuration procedure . . . . .	179
6.2.1	Configuration packet format . . . . .	179
6.2.2	Setting up and tearing down connections . . . . .	182
6.2.3	A path set-up example . . . . .	183
6.2.4	Slot counter synchronization . . . . .	186
6.3	Multicast . . . . .	188
6.4	Performance and hardware cost . . . . .	189
6.4.1	Hardware cost . . . . .	190
6.4.2	Configuration time . . . . .	193
6.4.3	Performance . . . . .	194
6.4.4	dAElite compared to other NoCs . . . . .	196
6.5	Related work . . . . .	196
6.6	Conclusions . . . . .	200
<b>7</b>	<b>Bandwidth efficiency and Latency hiding</b>	<b>201</b>
7.1	Write coalescing . . . . .	203
7.2	Posted writes and memory consistency . . . . .	204
7.3	Software prefetch . . . . .	208
7.4	Experimental Results . . . . .	209

7.5	Related Work . . . . .	216
7.6	Conclusions . . . . .	218
<b>8</b>	<b>Conclusions</b>	<b>219</b>
8.1	Objectives of research . . . . .	219
8.2	Contributions . . . . .	220
8.3	Thesis Summary . . . . .	221
8.4	Future Directions of Research . . . . .	222
<b>A</b>	<b>Sample LP model for a 2x2 mesh network</b>	<b>223</b>
<b>B</b>	<b>Example configuration of the proposed model</b>	<b>227</b>
<b>C</b>	<b>C source of the online allocation algorithm</b>	<b>233</b>
<b>D</b>	<b>Notations</b>	<b>235</b>
	<b>Bibliography</b>	<b>239</b>
	<b>List of Publications</b>	<b>257</b>
	<b>Samenvatting</b>	<b>259</b>
	<b>Curriculum Vitae</b>	<b>261</b>

# List of Figures

1.1	Interconnect solutions. . . . .	4
1.2	An interconnect generation flow and the components of this flow addressed in this thesis . . . . .	10
1.3	End-to-end connections between IPs are sharing network links.	11
1.4	Contention free routing. . . . .	12
1.5	Thesis Overview. . . . .	14
2.1	Network models, with their corresponding architectural char- acteristics and allocation algorithms. . . . .	18
2.2	Network implementations corresponding to the different models.	19
2.3	Network on chip. . . . .	20
2.4	Ideal, all-to-all topology, a bandwidth limit only considered for the links between NIs and Routers. . . . .	23
2.5	Mesh with one NI and multiple NIs per router. . . . .	24
2.6	a) Ring, b) Torus, c) Spidergon, d) Fat tree topologies. . . . .	25
2.7	Requested bandwidth being allocated over links supporting different link division granularities. . . . .	26
2.8	Space Division Multiplexing in one router. . . . .	27
2.9	Time Division Multiplexing in one router. . . . .	27
2.10	Router in a TDM scheme using the contention-free routing model. . . . .	28
2.11	Communication channel split over multiple paths. . . . .	29
2.12	Enforcing in-order delivery in the contention-free routing model.	30
2.13	Iterative channel allocation flowchart. . . . .	36
2.14	Steps in the iterative flow allocation algorithm. . . . .	37
2.15	Network nodes (a) and graph nodes after split (b). . . . .	39
2.16	Steps in the iterative single-path allocation algorithm. . . . .	41
2.17	Multi-path using iterative exhaustive search (for a single chan- nel). . . . .	43
E2.1	4x4 mesh network, random traffic, 16 IPs, 40 connections, 16 slots . . . . .	47

E2.2	4x4 torus network, random traffic, 16 IPs, 40 connections, 16 slots . . . . .	48
E2.3	Quaternary fat tree network (2 levels), random traffic, 16 IPs, 40 connections, 16 slots . . . . .	49
E2.4	16 node spidergon network, random traffic, 16 IPs, 40 connections, 16 slots . . . . .	50
E2.5	16 node ring network, random traffic, 16 IPs, 40 connections, 32 slots . . . . .	51
E2.6	4x4 mesh network, random traffic, 16 IPs, different number of connections, direct mapping, 16 slots . . . . .	53
E2.7	4x4 mesh network, random traffic, 16 IPs, different number of connections, UMARS mapping, 16 slots . . . . .	54
E2.8	4x4 mesh network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots . . . . .	55
E2.9	4x4 torus network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots . . . . .	56
E2.10	4x4 Mesh network, permutation traffic, 16 IPs, 16 slots . . .	57
E2.11	4x4 Torus network, permutation traffic, 16 IPs, 16 slots . . .	58
E2.12	Quaternary fat tree network (2 levels), permutation traffic, 16 IPs, 16 slots . . . . .	59
E2.13	16-node Spidergon network, permutation traffic, 16 IPs, 16 slots . . . . .	60
E2.14	16-node ring network, permutation traffic, 16 IPs, 16 slots .	60
E2.15	8x8 mesh network, permutation traffic, 64 IPs, 32 slots . . .	61
E2.16	8x8 torus network, permutation traffic, 64 IPs, 32 slots . . .	62
E2.17	Quaternary fat tree (3 levels), permutation traffic, 64 IPs, 32 slots . . . . .	62
E2.18	64-node Spidergon network, permutation traffic, 64 IPs, 32 slots . . . . .	63
E2.19	4x4 mesh network, MultiMedia System [HM05] traffic, 16 IPs, 16 slots . . . . .	64
E2.20	2x4 mesh network, digital TV [HG11] traffic, 8 IPs, 60 slots	65
E2.21	3x3 mesh network, MPEG-2 [LCOM08] traffic, 9 IPs, 16 slots . . . . .	66
E2.22	4x3 mesh network, VOPD [VdTJ02], 12 IPs, 16 slots . . . .	66
2.18	Traffic types reported in the literature: a) multimedia system [HM05], b) VOPD [VdTJ02], c) MWD [VdTJ02], d) MPEG-2 [LCOM08], e) digital TV [HG11]. . . . .	67
E2.23	4x3 mesh network, MWD [VdTJ02], 12 IPs, 16 slots . . . .	68



E2.24	Network frequency versus topology size for the same use-cases (random traffic) allocated on all topologies . . . . .	70
2.19	Model performance summary. . . . .	74
3.1	One-to-one graph representation of the network topology with slot masks attached to edges. Grayed slots are occupied. . . .	78
3.2	Network topology represented through a graph with split nodes.	78
3.3	Algorithms discussed in Chapter 3 and the models they correspond to. . . . .	79
3.4	Example graph for the path finding problem, used slots are grayed out. . . . .	80
3.5	Classic path finding algorithms fail to find a solution. . . . .	81
3.6	The number of possible paths grows exponentially with length.	81
3.7	Search example. . . . .	84
3.8	Sequence of values found at the top of the stack during algorithm execution. . . . .	86
3.9	Example flow. . . . .	88
3.10	Computing flow using augmenting paths. . . . .	89
3.11	$ek[V]$ costs associated to the nodes of the graph. . . . .	93
3.12	Different delays causing out-of-order delivery (a) and deadlock situations (b, c). . . . .	94
3.13	Slot table wrap-around. . . . .	97
3.14	In-order delivery is ensured through the use of guard slots. . .	102
4.1	Header overhead varies with the distribution of slots. . . . .	107
4.2	The average latency and bandwidth delivered by allocations using different patterns of 32, 8 and 4 slots out of 32. . . . .	108
4.3	Histogram of message latency (2 data words) for various slot distributions. . . . .	109
4.4	Histogram of message latency (3 data words) for various slot distributions. . . . .	110
4.5	Histogram of message latency (10 data words) for various slot distributions. . . . .	111
4.6	Histogram of slowdowns generated by different slot selection when no latency hiding techniques are used. . . . .	113
4.7	Histogram of slowdowns generated by different slot selections, when using posted writes, write coalescing and no read prefetch.	114
4.8	Histogram of slowdowns generated by different slot selections, when using posted writes, write coalescing and read prefetch. .	115
4.9	Problem formulation. . . . .	116

4.10	Steps of the original algorithm. . . . .	118
4.11	Solutions start with one non-selected, followed by one selected slot. . . . .	119
4.12	Building partial solutions by adding a slot at the end of an existing partial solution. . . . .	122
4.13	Enhanced formulation. . . . .	124
4.14	Algorithm for the enhanced formulation. . . . .	125
4.15	Numeric example for the enhanced problem formulation . . .	126
4.16	Steps of the algorithm for the enhanced formulation, all slot possibilities for the window [2-3] are considered . . . . .	127
4.17	Steps of the algorithm for the enhanced formulation, A-sets are generated . . . . .	128
4.18	Improvement in slot utilization vs background utilization, 16 slots. . . . .	134
4.19	Improvement in slot utilization vs background utilization, 24 slots. . . . .	135
4.20	Improvement in slot utilization vs background utilization, 32 slots. . . . .	136
4.21	Improvement in slot utilization vs background utilization, 40 slots. . . . .	136
4.22	Improvement in slot utilization vs requested latency and bandwidth, background traffic 8/16 slots. . . . .	137
4.23	Improvement in slot utilization vs requested latency and bandwidth, background traffic 8/32 slots. . . . .	137
4.24	Improvement in slot utilization vs requested latency and bandwidth, background traffic 16/32 slots. . . . .	138
4.25	A bound on latency of 1 selected slot every 6 slots guarantees that 4 words can be delivered every 12 slots. . . . .	139
4.26	In some situations it is not possible to allocate 2 words every 6 slots, but it is possible to allocate 4 words every 12 slots. . . .	139
4.27	Probability of successful allocation when 4 words are required in a window of 12 slots. . . . .	140
4.28	Probability of successful allocation when 6 words are required in a window of 12 slots. . . . .	141
4.29	Probability of successful allocation when 8 words are required in a window of 12 slots. . . . .	142
4.30	Probability of successful allocation when 12 words are required in a window of 12 slots. . . . .	142
4.31	Average number of slots used by each algorithm. . . . .	143

5.1	Path reservation in a classic Circuit-Switching network. . . . .	148
5.2	All outgoing links of a node are enumerated by the path finding algorithm but the ones leading away from the solution are immediately discarded. . . . .	149
5.3	An example network and tables used to describe its topology, the algorithms accept an arbitrary directed graph, although in practice unidirectional links appear in pairs with opposite directions. . . . .	150
5.4	Header overhead in <i>Æthereal</i> . . . . .	151
5.5	Exact computation of available words. . . . .	153
5.6	Approximate computation of the available words. . . . .	154
5.7	Hardware module to directly compute the exact number of available bandwidth. . . . .	155
5.8	Operations in the algorithm that can be executed in parallel; code references point to Algorithm 5.3.1. . . . .	157
5.9	Operations in the algorithm that can be speculatively executed. . . . .	158
5.10	Hardware module for accelerated path computation. . . . .	159
5.11	Allocation time vs. path length, 8x8 mesh, 0% background traffic. . . . .	161
5.12	Allocation time vs. path length, 8x8 mesh, 10% background traffic. . . . .	162
5.13	Allocation time vs. path length and requested bandwidth, 4x4 mesh, 10% background traffic. . . . .	163
5.14	Success Rate vs. path length and requested bandwidth, 4x4 mesh, 10% background traffic. . . . .	163
5.15	Allocation time vs. path length and requested bandwidth, 4x4 mesh, 20% background traffic. . . . .	164
5.16	Success Rate vs. path length and requested bandwidth, 4x4 mesh, 20% background traffic. . . . .	164
5.17	Allocation time vs. path length and requested bandwidth, 8x8 mesh, 5% background traffic. . . . .	165
5.18	Success Rate vs. path length and requested bandwidth, 8x8 mesh, 5% background traffic. . . . .	165
5.19	Allocation time vs. path length and requested bandwidth, 8x8 mesh, 10% background traffic. . . . .	166
5.20	Success Rate vs. path length and requested bandwidth, 8x8 mesh, 10% background traffic. . . . .	166

5.21	Speed of the hardware implementation vs. software implementation with accelerated bandwidth computation, 4x4 mesh, 20% utilization. . . . .	169
5.22	Average speed of all methods, 4x4 mesh 20% utilization. . . . .	169
6.1	Example dAElite platform instance. . . . .	175
6.2	dAElite Router . . . . .	177
6.3	dAElite Network Interface . . . . .	178
6.4	Format of the configuration packets. . . . .	181
6.5	Path set-up example. . . . .	183
6.6	Path set-up, slot table registered at the first router. . . . .	185
6.7	Path set-up, slot table update takes place in R10, R11, NI11. . . . .	186
6.8	dAElite relaxed synchronous model. . . . .	187
6.9	Multicast in dAElite . . . . .	188
6.10	Hardware cost comparison. . . . .	191
6.11	Hardware cost breakdown. . . . .	192
6.12	Dependence of the hardware cost on the number of slots . . . . .	193
7.1	Correspondence of OSI layers to the Æthereal implementation. . . . .	202
7.2	NI shell supporting burst write. . . . .	204
7.3	NI shell without burst or posted write support. . . . .	205
7.4	Consistency issues raised by signals arriving at different destinations with different delays. . . . .	206
7.5	Emulation of a weak consistency model. . . . .	207
7.6	Example prefetch loop compared to original code. . . . .	208
7.7	Hardware cost of the shells in FPGA implementation as a percentage of the cost of the entire network. . . . .	210
7.8	Test setup: one MicroBlaze core is connected through two separate channels, one on the PLB bus and one on an FSL link to a remote memory. . . . .	211
7.9	Performance of Read and Write tests under different setups. . . . .	212
7.10	LL kernel 1. . . . .	213
7.11	LL kernel 6. . . . .	213
7.12	LL kernel 12. . . . .	214
7.13	LL kernel 21. . . . .	214
7.14	JPEG. . . . .	215
7.15	Average speedup across all bandwidths obtained with the different techniques in all applications. . . . .	215

# Chapter 1

## Introduction

For more than five decades, the microelectronics industry has sustained an evolution unmatched in any other field. The underlying phenomenon that has allowed this evolution is the miniaturization of silicon devices. This miniaturization enabled two secondary phenomena: integration and an increase in architectural complexity.

Integration meant that individual transistors were grouped together into logic gates, logic gates into circuits with more complex function, eventually leading to the birth of the microprocessor and today the system on chip. As a result, functions that were previously performed by separate electronic components were merged into single chips. At the same time the total number of transistors per system has increased many orders of magnitude, allowing more and more functionality to be added and increasing performance.

While miniaturization decreased the manufacturing cost per transistor, it was also necessary to decrease the design cost per transistor. This is why the evolution of design tools and methodologies played a crucial role in allowing system evolution and integration. Although there is now a strong emphasis on reusing intellectual property (IP) that has been designed, deployed in other products and verified in the past, interconnecting these already existing designs remains today's challenge.

### 1.1 Design trends

As the trend of device miniaturization continues the number of transistors per chip doubles every couple of years. The increasing density can be used

in several ways: the size of the chips can be reduced, individual processing blocks can become more complex thus providing higher processing power, more functional blocks can be integrated on the same chip. Reducing the size of chips, although beneficial from the cost point of view, cannot be done indefinitely because at a certain point the cost of packaging and terminals would become dominant. The direction that is left and is still promising is the integration functions that were traditionally performed by different devices into a single device.

Integration has several benefits: the cost of several packages is eliminated and the need for connections that would normally go to the outside of the chip is removed. Integration improves performance because communication bandwidth available on chip is significantly higher than off chip. It decreases power consumption as driving external pins uses much more energy than on-chip communication. Another important benefit is the reduced physical size of devices which is more appealing to customers.

Traditionally IP blocks were connected using a single bus or a hierarchy of buses. The parameters of these components could be manually chosen by a skilled engineer and the components themselves could be instantiated from a library to obtain a working system. However, this approach will not scale to designs having tens to hundreds of cores, because companies cannot afford increasing the engineering effort per device. Timing constraints become increasingly difficult to meet and verification becomes difficult to perform.

Analyzing the system from the performance point of view also becomes increasingly difficult. While the computation requirements for individual processors can be generally analyzed and verified for many real-life applications, the communication performance requirements are less straightforward since the interactions between different IPs need to be taken into account. If the system fails to meet the performance requirements, redesigning the interconnect (or entire SoC) may be a time-consuming and costly operation. It is therefore desirable to have automated tools to dimension and verify the interconnect.

These tools start with a high level system or application requirements and automatically generate an interconnect that the system components are attached to. This interconnect may also be verifiable by construction from the correctness and performance points of view. One such type of automatically generated interconnects are networks on chip [GDR05, MM04] which we will discuss in this thesis. They represent a promising solution for the interconnect of future designs having an increasing number of IPs.

In this thesis we study the algorithms and methods behind the interconnect

design tools. The rest of this chapter is organized as follows. Section 1.2 presents an overview of traditional and modern interconnect solutions. Section 1.3 presents an overview on networks on chip. Section 1.4.2 summarizes the thesis contributions and Section 1.5 presents the structure of this thesis.

## 1.2 Overview of chip interconnect solutions

A typical system consists of a collection of master and slave IP blocks. Master IP blocks, e.g. microprocessors, make requests, like for example modifying the contents of a memory location or of a status register inside a peripheral. Slave IP blocks, e.g. memories, receive, process and confirm the execution of these requests.

In the following paragraphs we present traditional and modern interconnect solutions. These interconnects are also represented in Figure 1.1.

A trivial solution, the **back-to-back connection** (Figure 1.1a), can be employed when a single master IP needs to be connected to a single slave IP. Ideally only wires are required for this connection, but this holds true only assuming the two IPs agree on the language used to perform the communication. A protocol defines the conventions regarding the communication between IPs: the set of physical signals, their allowed values and timing.

The requests usually have memory access semantics, i.e., memory read and memory write operations. Here the request and its response will be referred to as transactions. Several signals are characteristic to a protocol supporting these transactions. Command signals are used to perform a handshake between the two IPs. Using these signals, a master IP specifies when it presents a request and a slave IP responds when it can accept or serve the request. Address signals (usually part of the command signal group) indicate a specific memory location where the data should be stored or where it has to be read from. The actual data may be transferred either from the master to the slave IP, in which case we call the transaction a write transaction or it can be read back from the slave IP.

A more complex interconnect is needed when multiple master and/or multiple slave IPs are present. Various approaches are possible, with the main tradeoff being between performance, e.g. latency and throughput, and cost, e.g. chip area and power.

The **shared bus** approach [SLKH02] is one of the least expensive interconnects in terms of area. From a logical point of view it can be seen as a multiplexer of requests coming from the master IPs followed by a demultiplexer

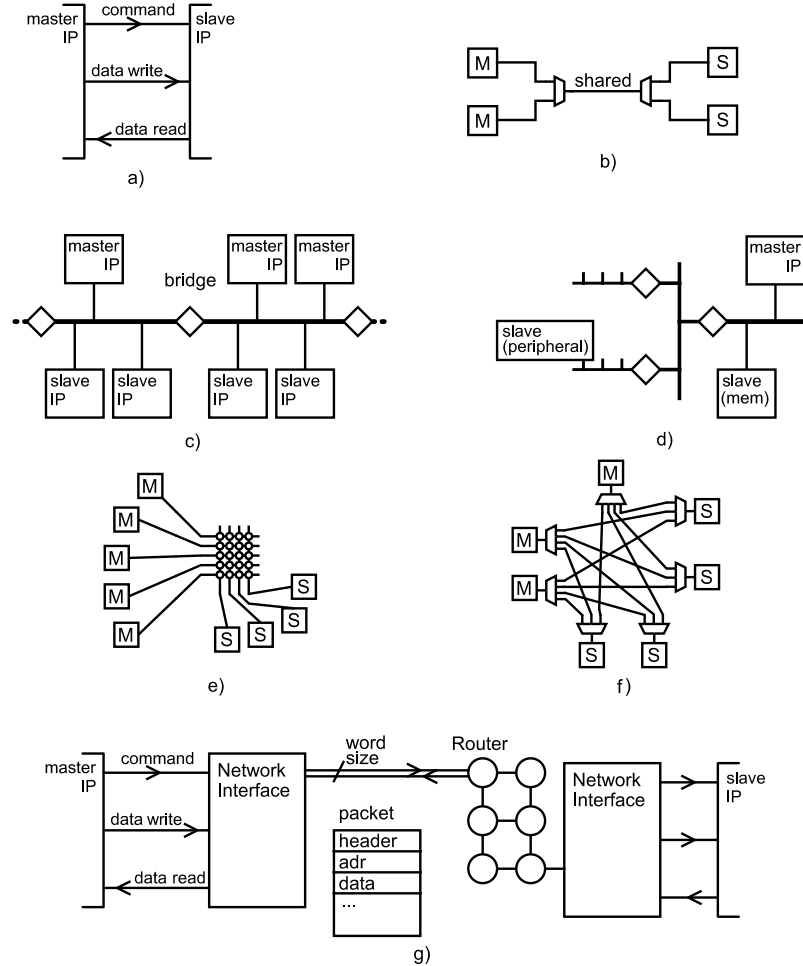


Figure 1.1: Interconnect solutions.

with outputs connected to the slave IPs (Figure 1.1b). In practice however, a “wired and” or a “wired or” was used instead of the multiplexer and the signal was distributed to all the slave IPs which decided independently whether to respond to the request or not. The slave IP block is selected based on the value of the address lines. An arbiter controls which master IP block has access to the shared bus at each moment in time. Typical example of such buses are APB [Lim08] for ARM based system or OPB [Cor01] from IBM.

The disadvantages of this approach are poor performance, low operating fre-



quency (at least in the “wired and/or” implementation), high power consumption as switching occurs on longer wires than would be needed to transmit the information between source and destination, and the fact that multiple masters cannot make requests at the same time. Another disadvantage is that slow IP blocks can block the shared portion of the interconnect for a long time preventing other IPs from performing their own operations over the bus.

Two possible improvements are the **segmented bus** [CJW<sup>+</sup>99, PSL03] (Figure 1.1c) and the **bus hierarchy** [WM88, SKL<sup>+</sup>07] (Figure 1.1d). Both these approaches attempt to mitigate the performance and power loss of the bus approach. The gain in power efficiency comes from the fact that signals are not required to propagate to all slave IP blocks, but only in the relevant segments. Multiple requests can be served in parallel in the different segments and slow IP blocks no longer slow down the entire system. It is customary to isolate slow IP blocks in slower sections of the hierarchy for example.

The design of such systems is however less straightforward. Requests may be initiated in different segments at the same time and independently of each other. When these requests need to be served outside their own segment and they furthermore require access to resources that they mutually block, a deadlock situation may occur. Special hardware may need to be introduced to avoid or resolve deadlock situations. The topology of the bus hierarchy is typically constrained to a tree to limit the possibility of such deadlocks. A typical example for this type of interconnect is IBM CoreConnect [Cor99].

The central **crossbar** (Figure 1.1e) is a high-performance interconnect solution. It also comes at a very high cost, with  $O(mn)$  hardware complexity, where  $m$  is the number of masters and  $n$  the number of slaves. However, the regularity of the circuit and careful floorplanning can make even large designs feasible [PKP10]. The crossbar does not exploit locality of communication as all data has to travel to a central location. The cost of wires that connect IPs that are spread across the surface of the chip to the central location is also high. The performance benefit comes from the fact that the number of requests being served at the same time is maximized and the latency is very low as arbitration needs to be performed only once for a request to travel from a master to a slave IP block.

**Direct connections** (Figure 1.1f), which may be all-to-all connections or only the required subset, can be regarded as the opposite of the bus. From the logical point of view they are similar to the crossbar as they allow transactions to occur simultaneously as long as they occur between different master and slave devices. They differ though in that the data does not have to travel to a central

location, but is sent over dedicated wires to its destination. This solution requires abundant wiring resources. On the other hand, the data only needs to travel the physical distance between source and destination so the switching energy is reduced. A typical implementation of this type of interconnect is ML-AHB [Lim01] from ARM. At the logic level, this type of interconnect can be seen as the opposite of the shared bus approach: a request from a master IP is demultiplexed and sent to the proper slave IP block where an arbiter and multiplexer combine the requests from multiple masters.

**Networks on chip** [DT01, BDM02, SSM<sup>+</sup>01, IHCE07, Art05] (Figure 1.1g) borrow concepts from the large-scale communication networks to create a scalable on-chip interconnect. They typically consist of network interfaces which transform the transaction requests and responses into data packets which are then transmitted over a network of routers to the destination, where they are transformed back into transactions understood by the corresponding IP. The data packets usually travel over links with lower bit-width thus reducing the needed wiring resources. The routers can be connected in an arbitrary topology. This is an advantage compared to bus hierarchies which typically support only tree topologies.

Table 1.1 gives an overview of the advantages and disadvantages of the interconnect solutions earlier presented. The biggest strength of networks on chip is scalability, NoCs can scale essentially to any number of on-chip connected components. The cost of the solutions has already been discussed in the previous paragraphs. Regarding the throughput measure we differentiate between two usage scenarios. Simple interconnects offer good performance when a single pair of IPs is communicating at any time, the more expensive interconnects are able to serve multiple communicating pairs simultaneously. What we defined as energy efficiency is the length of wire subjected to switching activity to support the communication compared to an ideal situation where data follows the shortest distance between source and destination. Finally, the number of arbitration levels indicates how many times a request is subject to arbitration before reaching the destination. A higher number of arbitration levels may indicate a higher latency, but also less complex individual arbiters.

In the next section we will have a look into the characteristics of NoCs.

### 1.3 Networks on Chip

Some of the first NoC studies [DT01, BDM02, SSM<sup>+</sup>01] pointed out that traditional interconnects are reaching their limits and a different approach

	back-to-back	shared bus	segmented bus	bus hierarchy	xbar	direct connect	NoCs
scaling	2	10	tens	tens	10-100	low tens	10+
wire cost	low	low	low	low	high	very high	average
logic/buff cost	none	very low	low	low	high	high	average
throughput	highest	very low	low	low	highest	highest	high
single conn throughput	high	low	high	high	high	high	high
energy use efficiency	good	poor	average	average	average	good	good
arbitration levels	none	one	many	many	one	one	many

Table 1.1: Interconnect advantages and disadvantages

is needed to achieve scalability in designs with potentially hundreds of IPs. Networks on Chip leverage on the experience gathered in large-scale networks and in particular in high-performance multiprocessor computers.

The typical characteristics of a NoC interconnect are:

1. **Packetization and serialization.** A network on chip translates and encodes into fixed-size words the requests and responses coming from IPs [SdWG10, RDP<sup>+</sup>05, HJK04, BM03, SBB<sup>+</sup>06]. This allows the network links to use a lower number of physical wires. In addition this encoding introduces a level of abstraction in that individual bits lose their meaning during network traversal. This means that the network logic does not need to be concerned with the semantics of the data traveling through the network, and at the same time it allows the network to act as a bridge between IPs using different bus protocols. The link width can also be arbitrarily reduced to conserve resources when the communication requirements are low.
2. **Link sharing** mechanisms are typically provided by NoCs. These allow the multiplexing of several streams of data over the same physical medium. Common schemes are space division multiplexing (SDM) [MSAA09, BWM<sup>+</sup>09, LL11, LMS<sup>+</sup>05], time division multiplexing (TDM) [GDR05, LZT04, WZLY08, ZFK<sup>+</sup>09], either in the conventional slot allocation approach or in an arbitrated (e.g. round-robin, priority) link time sharing scheme. Wavelength division multiplexing was also proposed in the context of optical NoCs [SBC07, JBK<sup>+</sup>09, KC11, CA11].
3. **Topology** [BC06, BCG<sup>+</sup>07, LGM<sup>+</sup>09] in networks on chip is typically less constrained than in bus-based interconnects. In particular, unlike a bus hierarchy, a network on chip is not required to be a tree topology.

The NoC topology can be adapted to the chip floorplan and can be optimized to take into account the communication requirements between IPs [BJM<sup>+</sup>05, OM05, Ben06, CP08, SCK06, SMBDM10].

4. **Switching model.** Networks on chip can be split in two large categories: circuit-switching and packet-switching networks [CSC06]. Circuit switching networks [GDR05, BWM<sup>+</sup>09, LST00, LWS<sup>+</sup>02] allocate relatively long-lived connections between source and destination, and provide high bandwidth and low latency over these connections. In contrast, packet switching networks [SLB07, ACG<sup>+</sup>03, BB04, BCGK04, BB04, MMV09, Bje05] use arbitration at each network node and for each data packet. Under low network load, packet switching networks provide good latency. One concern for packet switching networks is the “saturation point,” where an increase in traffic causes a disproportionate increase in latency. In this thesis we focus on circuit-switching NoCs, although some of the performance models we introduce in Chapter 2 are also applicable to packet-switching NoCs.
5. **Routing** is the process that decides the path the data takes between source and destination. Networks on chip, unlike bus hierarchies, typically offer more than one path between source and destination. This characteristic can be used to balance load [LZJ06, ACPP06, LLP05]. A large research effort was focused on finding deadlock-free routing strategies that are nevertheless able to balance traffic or deal with faults [DA93, Dua91, PHKC06, KS93, CQSD99].
6. **Flow control** is the process of managing the rate of data transmission in an interconnect, to avoid buffer overflow. Flow control in NoCs tends to be more elaborate than in bus-based systems [CMR<sup>+</sup>06, PABB05, OM06, ANM<sup>+</sup>05] because NoCs allow pipelining of transactions. Pipelining is necessary to achieve high frequency of operation and thus high communication bandwidth.
7. **Distributed operation.** A network on chip is composed of modules that operate autonomously and in parallel. This is important as it avoids a single centralized control unit such as a bus arbiter, that may represent bottleneck. However, it is not uncommon to have a single central circuit for less frequent functions, like network configuration [HG07].
8. **Quality of service (QoS)** is often defined as the ability of an architecture to guarantee performance requirements like bandwidth and

latency. Some NoC implementations offer higher priority, and thus higher performance to a subset of applications by defining traffic classes [BCGK04, BCV<sup>+</sup>05]. Others offer bandwidth and latency guarantees through resource reservation [GDR05, MNTJ04a, LST00]. Applications or communication channels are in this situation isolated from each making their performance more easily analyzable.

These characteristics also represent degrees of freedom in NoC design. Several of these parameters are explored in Chapter 2 through the proposal of network models implementing specific parameter choices. We will also return to discuss them in later chapters in the context of allocation algorithms targeting routing and link sharing or in the context of hardware implementation. Table 1.2 presents a map of the chapters that will take into discussion these parameters.

		parameter selection	resource allocation		hardware implem.		
			offline	online	NoC	IP intf.	
		Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
F-1	Serialization						v
F-2	Link Sharing	v		v			
F-3	Topology	v					
F-4	Switching Model	v	v			v	
F-5	Routing	v	v		v	v	
F-6	Flow Control					v	v
F-7	Distributed op.					v	
F-8	QoS	v		v			

Table 1.2: Chapter-parameter relation

## 1.4 NoC design

Consider the generic design flow, illustrated in Figure 1.2, which is used for the design of an on-chip interconnect, in particular a network-on-chip.

The design flow starts from a set of application domain (qualitative) requirements, e.g. optimization of average throughput in general purpose applications, worst-case guarantees for embedded real time or low-latency for control applications. These requirements specify whether QoS guarantees need to be provided [GH10], whether applications need to be isolated from each other [HGBH09, HG10, BGK<sup>+</sup>11] or how resources need to be shared. Based

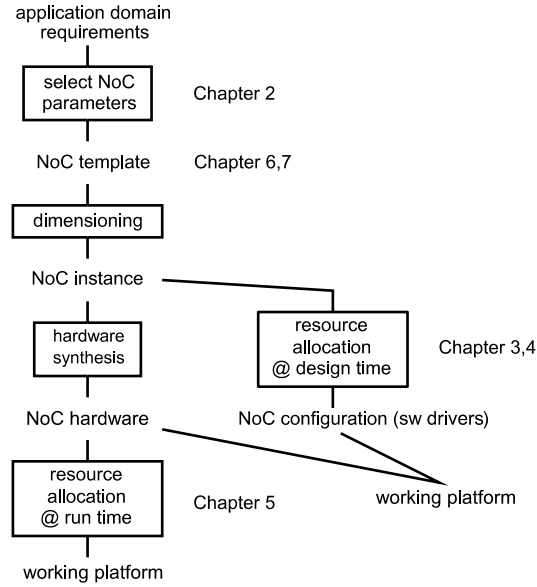


Figure 1.2: An interconnect generation flow and the components of this flow addressed in this thesis

on these requirements, the type of NoC or, as presented in Section 1.3, the parameters of the NoC can be chosen. In Chapter 2 we provide an analysis of several network models that may be used in determining some of the network parameters.

In Chapters 6 and 7 we present our proposed NoC implementation, targeted at embedded systems for real-time multimedia. Our proposal is based on parameter choices we found to be desirable for an interconnect providing QoS guarantees in the analysis in Chapter 2. We have developed a library of hardware models, that can be dimensioned and instantiated using already existing tools from the *Æthereal* tool-flow [RDP<sup>+</sup>05, HCG07b].

Even with a fixed hardware instance, there may be additional degrees of freedom in the functioning of the network. In the case of our proposed network, these degrees of freedom are represented by the selection of routes and the link sharing schedules for each of the network connections. We call the computation of these routes and schedules *resource allocation* and we dedicate a large part of the thesis to algorithms used for this computation. Chapters 3 and 4 will discuss design-time resource allocation while Chapter 5 will discuss allocation at run time.

In the following section we present the contention-free routing model, which is implemented by our proposed network. This model is a characteristic our network shares with the *Æthereal* network [HG10].

### 1.4.1 The contention-free routing model

The contention-free routing model is a model for circuit-switching networks that make use of time division multiplexing for sharing the links between multiple connections. In a circuit-switching network, long-lived connections (or circuits) are established between IPs and while established they reserve network resources on a path through the network between the IPs thus making sure that data can flow at a certain rate between source and destination. To avoid excessive blocking and possible under-utilization it is necessary that links could be shared between multiple connections (i.e., only a fraction of the link bandwidth is reserved by one connection instead of the whole link).

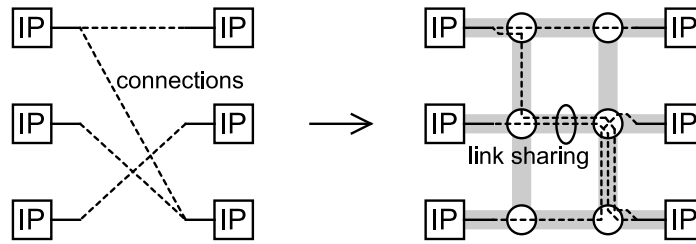


Figure 1.3: End-to-end connections between IPs are sharing network links.

A convenient mechanism for link sharing is time division multiplexing or TDM. TDM specifies that connections can take turns in using a link, according to a strict schedule. When a connection is established, one or more entries (or *slots*) in this schedule are marked as belonging to that connection. The number of slots allocated to one connection determines the bandwidth of that connection.

The contention-free routing model (as proposed in [GDR05]) however goes further than simple TDM slot reservation. It specifies that, when switching from one link to another on the path from source to destination, data is not allowed to wait. That is, when arriving at a router in one time-slot, the data must depart on the next link in the immediately next time-slot (Figure 1.4). It follows from this that connections never wait for each other, hence the name “contention-free routing.”

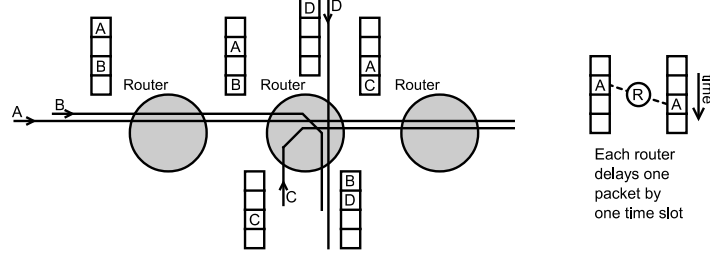


Figure 1.4: Contention free routing.

This is a very restrictive requirement, but it has significant benefits. Firstly, the network traversal latency is very low, because data does not need to wait at each router, and secondly, the buffer requirements at the router level are also very low.

The contention-free routing model unfortunately also has an unpleasant effect. The relative position of slots on all links that are used by one connection becomes “locked” and finding a schedule that avoids collisions between any connections at any point in their path becomes a very difficult problem. In this thesis, we will analyze the impact of this restriction on performance and cost and we will provide algorithms for solving the allocation problem.

### 1.4.2 Thesis contribution

This thesis has several contributions: (1) a performance analysis of network parameters through the use of several models, (2) allocation algorithms targeting (contention-free) routing and (3) an efficient hardware implementation of the dAElite network on chip that supports contention-free routing and hence offers bandwidth and latency guarantees. We detail these contributions in the following paragraphs.

To explore the effects of various *network parameters*, we have developed a range of *network models* to analyze the (guaranteed) performance. These range from generic models, applicable to any type of interconnect to specialized models that match the hardware implementation of a network based on contention-free routing. For each of the models we provide ways of measuring performance which allows us to determine tradeoffs of various interconnect choices. We evaluate the impact (on performance) of packetization, the impact of link division granularity, the impact of having aligned slots, the impact of



requiring or not requiring in-order delivery and several network topologies.

We develop and implement *algorithms* for the computation of TDM schedules (*resource allocation*) both at design time and at run time. We demonstrate several of the allocation algorithms to be optimal. For run-time allocation, we optimize our algorithms for execution on embedded processors with little memory and low processing power. We accelerate the algorithm by implementing part of its functionality in hardware and we also estimate the speed of a fully hardware-accelerated solution.

We propose a NoC *hardware implementation* supporting contention-free routing which compares favorably in terms of cost and performance to the state of the art. Our proposal features multicast and multipath routing and has a low connection set-up time. Our approach also avoids the header overhead. We design the hardware required to interface IPs to the network on chip. We propose improvements at the level of this interface that reduce the running time of applications by making better use of the network bandwidth and hiding latency.

## 1.5 Thesis Overview

In the following we detail the organization of this thesis, which is also represented in Figure 1.5. The thesis is structured around a set of network models, ranging from ideal interconnects existing only as a mathematical formulation to realistic models that allow a physical implementation.

In Chapter 2 we analyze the performance impacts of various network parameter choices. The performance variations are determined by comparing different network models that differ in these choices. We provide mathematical formulations based on linear programming for the performance of the first five models, while for the rest the performance is measured by applying allocation algorithms that will be described in the later chapters.

In Chapter 3 we discuss in depth the algorithms that are used to perform path allocation in the more restrictive models introduced in Chapter 2. Where possible we employ optimal algorithms and we demonstrate their optimality. We also give an overview of the complexity of these algorithms.

In Chapter 4 we present algorithms for slot selection which are used in conjunction with the path selection algorithms to guarantee a certain latency bound for each communication channel. We prove these algorithms to be optimal which allows us to use them as a bound in evaluating the performance of

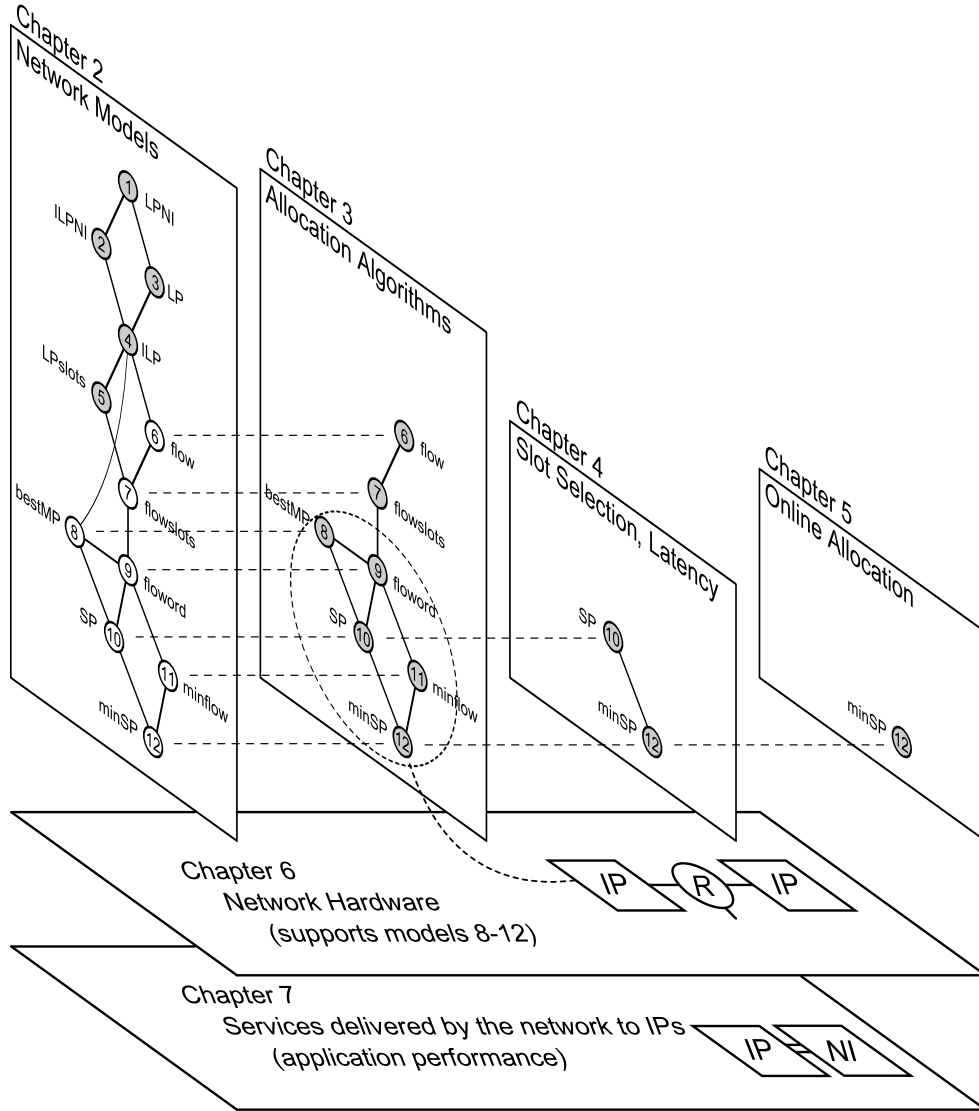


Figure 1.5: Thesis Overview.

the algorithms previously proposed in the literature. These algorithms are applicable only to the single path models. We also look into the effects of slot selection on actual application performance and we provide more elaborate definition of latency constraints.

Whereas most of the algorithms presented so far are only meant to be used at design time, in Chapter 5 we present algorithms for path and slot selection that can be executed at run time. We evaluate the speed of the algorithm versus the connection requirements and background traffic, and present different tradeoffs between the quality of the allocation, hardware resources and speed. We also evaluate the various overheads involved with keeping track of used resources and the interaction with the design-time allocation flow. We present ways to accelerate the allocation in hardware.

In Chapter 6 we propose a network on chip implementation called dAEIite that supports several of our proposed network models, offers multicast services and has very low configuration time. We implement our proposal in hardware and we show it to perform favorably in terms of hardware area and speed to other networks reported in the literature.

Chapter 7 focuses on how the raw communication services provided by the network can be translated into transaction-level services offered to the IPs. We present here optimizations regarding bandwidth use and latency hiding techniques and we analyze the overall effect on the execution time of real applications.

Finally, Chapter 8 concludes the dissertation by summarizing our contributions and presents future possible directions of research.



## Chapter 2

### Theoretical bounds on allocatable capacity

To arrive at a NoC template, we analyze in this chapter several network characteristics (Section 1.3) that affect implementation cost and guaranteed performance. Starting with a most general network, we specialize the NoC until we arrive at the contention-free routing model which can be implemented in hardware at a low cost. At the same time we design a range of resource allocation methods adapted to each network's requirements.

For this task we propose a range of interconnect models, offering different levels of abstraction. These models address the network characteristics and degrees of freedom discussed in Section 1.3. We evaluate different topology choices, different routing strategies, different granularities of link bandwidth division and a generic switching model versus the contention-free routing mechanism.

The models representing interconnects with different architectural restrictions require different algorithms for network resource allocation. For the more general models we use linear-programming-based algorithms to perform the allocation. These algorithms are able to guarantee an optimal solution globally for the entire set of communication channels. Linear programming is however too expensive to perform for the more detailed models. In particular, under the contention-free routing model, the problem of finding a globally optimal allocation was shown to be NP-complete [SBG<sup>+</sup>08]. For these cases we use algorithms that optimize the allocation of each individual channel but are not able to guarantee a global optimal solution (across the entire set of channels). We evaluate the performance of these algorithms by comparing to



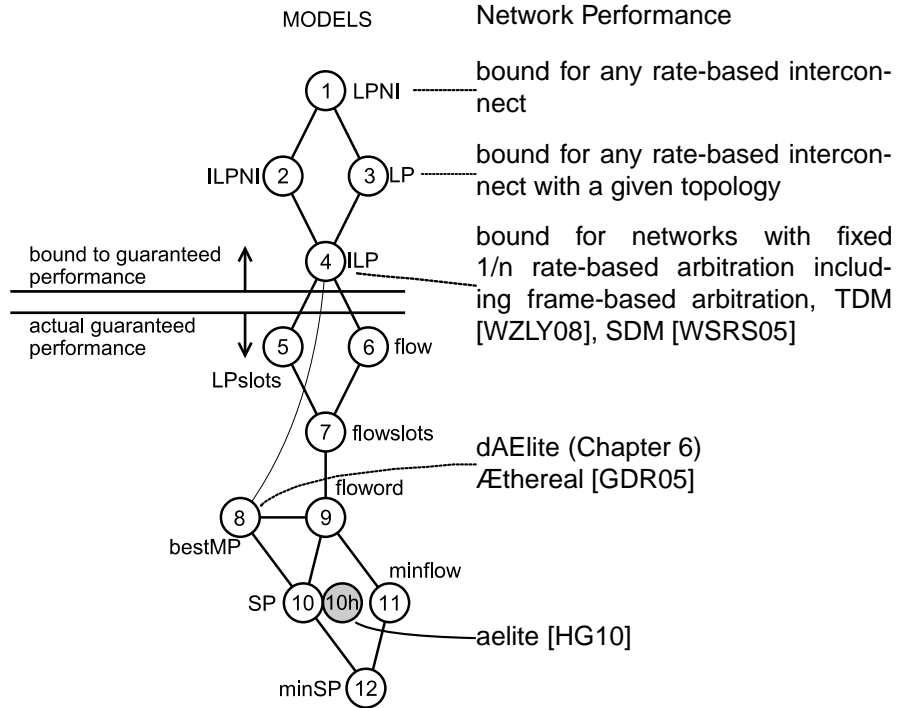


Figure 2.2: Network implementations corresponding to the different models.

resents. Models 8 and 12 represent the performance that *is* obtained (and guaranteed) by dAEIite and Æthereal respectively. Models 1-4 represent a bound on the performance that *could* be achieved by the specified networks, assuming no other technical restrictions exist, e.g. routing restrictions or limited buffering at the intermediate nodes. Furthermore, for networks using frame-based arbitration the performance guarantee represents bandwidth that is delivered in each and every frame whereas in a generic rate-based arbitration scheme the performance that can be guaranteed only concerns the average delivered bandwidth.

The rest of this chapter is organized as follows. In Section 2.1 we discuss the assumptions we make regarding the interconnect that are common to all models. Section 2.2 then details the restrictions each model successively introduces. In Section 2.3 we describe the allocation methods employed. The allocation methods based on linear programming are described here as they are the result of directly applying a linear programming optimization tool to the formal description of the problem. For the other methods we only give a brief

overview as we will provide a detailed description in the following chapter. Section 2.4 presents a performance comparison of the various models. Note that some of the models are purely theoretical and their only purpose is to provide a theoretical bound to the achievable performance regardless of the network characteristics. Section 2.5 presents related work and Section 2.6 states our conclusions.

## 2.1 General assumptions

Some of the assumptions we make are common to all models. In all our experiments IPs are assumed to communicate over connections having constant bandwidth. Bursty traffic can be dealt with by using buffers to level out the load, overallocating bandwidth or by setting up and tearing down connections to correspond with the bursts. In the latter case we only consider the time interval between any set-up or tear-down operations, i.e. the time interval when the networks is in a stable state.

Each of the models is evaluated as follows:

The network has to support a set of bidirectional connections, each of which has a required bandwidth. We emphasize the fact that the requested bandwidth represents *guaranteed bandwidth* that the network should be able to deliver. Each connection consists of two unidirectional channels, a request channel and a response channel. Latency requirements are not taken into account in this chapter as most models do not support them. Latency is considered in Chapter 4.

The NoC consists of network interfaces (NIs), routers and network links (Figure 2.3). IPs are tightly connected to a network interface and we assume there is no restriction in the speed of communication between the two. Each network interface has a single link to a router. Between routers multiple links exist, as defined by the network topology.

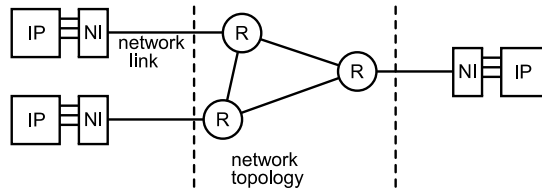


Figure 2.3: Network on chip.



The general assumptions, which are unchanged throughout the thesis, are as follows:

- GA-1 The network consists of point-to-point unidirectional communication links; bidirectional or full-duplex links are modeled as two separate unidirectional links.
- GA-2 We assume all links to have equal capacity; the models not using slots (Models 1-4 and 6) are trivial to extend to handle different-bandwidth links if desired.
- GA-3 The network does not drop or re-transmit data, for a network that uses dropping or retransmission we only model the useful throughput and ignore the dropped packets; this provides a conservative performance bound (a network using dropping cannot perform better than the network without dropping which uses the same model).
- GA-4 Our allocation algorithms assume IPs that are already mapped to network nodes; in our experiments we will use two mappings: an (essentially arbitrary) mapping with the first IP mapped to the first NI, the second IP to the second NI and so on, and the mapping produced by the UMARS algorithm [HGR07].
- GA-5 It is allowed to have multiple IPs mapped to the same NI, and it is allowed to have multiple NIs connected to the same router; both these features are used in some of our experiments.

To compare the quality of our allocation algorithms against the state of the art [Han09] we use the same conventions for bandwidth computation, link width and header overhead. Our tools also use the same input format (XML files) as the *Æthereal* toolchain [HG10] leading to potential synergies between the two flows.

## 2.2 Architectural restrictions

An ideal interconnect provides infinite bandwidth and has zero latency. The laws of physics obviously prevent us from achieving that and so do cost concerns. There are many choices in the interconnect design that balance cost against performance. We start with an idealized network model which ignores cost, though it still abides by the laws of physics. We then gradually introduce

models with more restrictions, lowering cost at the expense of performance and we evaluate the impact of each of these restrictions.

The most general model starts with the following assumptions which will then be invalidated by the more restrictive models:

- A-1 The network topology is ideal. Models 1 and 2 assume that contention only happens on the NI-router links and not on the router-router links inside the network.
- A-2 The data arrives and departs from a router in (arbitrary) infinitely divisible quantities (from a continuous range as opposed to a discrete set of values). The link bandwidth can also be divided between the connections that use it in arbitrary (continuous domain) proportions; starting with Models 2 and 4 the link bandwidth can only be divided into discrete amounts.
- A-3 No assumptions are made regarding the switching mechanism; the bounds provided by this model can be applied to any type of interconnect, including bus hierarchies; the contention-free routing model will be introduced starting with Models 5 and 7.
- A-4 In-order arrival of messages at the destination is not enforced; we will take into account the order of arrival at the destination in models starting with Model 8.
- A-5 Routing of a single communication channel over multiple paths is allowed, our only concern is that all data must eventually reach its destination; a restriction consisting of routing each channel over a single path will be introduced starting with Models 10 and 12.
- A-6 Non-minimal routing with respect to distance is allowed. The models we use optimize the path length either globally or per connection, however the successful allocation criterion takes precedence; a restriction on path length will be introduced in Models 11 and 12.
- A-7 No network header overhead is considered. Header overhead is optional starting with Model 8.

We present these assumptions in more detail in the following sections.

### 2.2.1 Ideal versus real topologies

A major factor impacting the cost of a network implementation is the network topology, as it directly affects the length and number of wires, the number of routers and the router degree. We wish to determine whether the network topologies used in practice incur a loss in terms of network performance when compared to an ideal topology.

For the ideal topology we use a model that only restricts the bandwidth of the links that connect the NIs to the core of the network (we assume all-to-all connections without a limit on bandwidth inside the network). This model is represented in Figure 2.4.

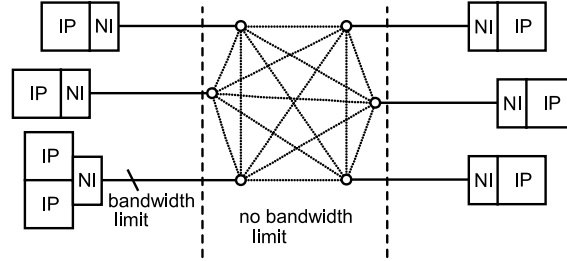


Figure 2.4: Ideal, all-to-all topology, a bandwidth limit only considered for the links between NIs and Routers.

With an ideal topology assumption, the overall performance of the network is limited by the IP or group of IPs having the largest inbound or outbound traffic. In our tests we found that in many cases the NI to router links still saturate first which means that the evaluated topology behaves like the ideal topology.

In practice the all-to-all topology is not used because of its high wiring cost. Topologies commonly used instead are the mesh and concentrated mesh, ring, torus, spidergon, and fat tree.

The mesh (Figure 2.5a) topology has the advantage of mapping very well on the 2D surface of the chip, especially in the case of homogeneous multi-cores which are composed of fixed size IPs replicated in a matrix structure. For each topology it is possible to connect multiple NIs to each router (Figure 2.5b vs. Figure 2.5a). Using multiple NIs per router decreases the size of the topology and the hop count, but increases the pressure on the router to router links.

Other commonly used topologies include: ring (Figure 2.6a), torus (Figure 2.6b), spidergon [MSVO07] (Figure 2.6c), fat tree [Lei85, GG00] (Fig 2.6d).

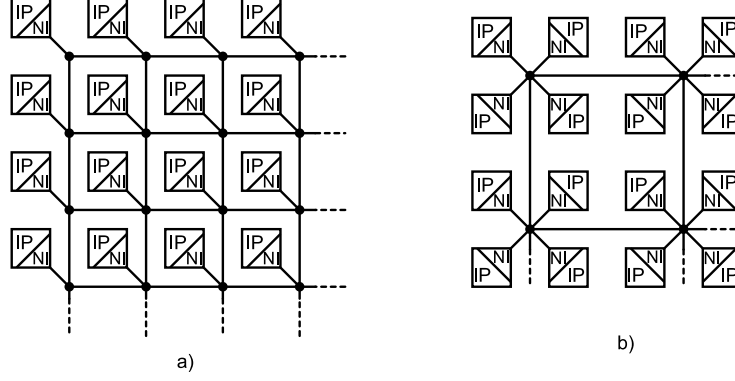


Figure 2.5: Mesh with one NI and multiple NIs per router.

For all topologies, each of the links is assumed to be bidirectional (thus modeled as two separate links in opposing directions) and the capacities of all links are assumed to be equal.

### 2.2.2 Continuous versus discrete bandwidth division

As mentioned previously, in a NoC links are typically shared. Multiple communication channels are allowed to make use of each link, and a mechanism has to be provided to divide the link bandwidth between channels.

In the more general models we assumed this division can be performed into arbitrarily fine-grained quantities (which can be represented by real numbers). In practice, this division can only be performed into discrete units dictated by the link sharing mechanism employed. In Models 2 and 4-12 we allow the link bandwidth to be split into  $n$  units, each of them equal to  $1/n$  of the total bandwidth (for simplicity, when constructing the performance models we will assume the unit of allocation to be 1 and the total link bandwidth to be  $n$ ). The value of  $n$  is constant throughout the network. Note that often, but not always,  $n$  may be made very large at low cost, so that these models approximate models 1 and 3 closely.

The use of a fixed allocation unit may be for example due to the physical link division mechanism employed, as it is the case for space-division multiplexing (SDM), or a limitation of the arbitration scheme, as it is the case with frame-based arbiters. Even when the arbitration scheme does not inherently suffer from this limitation, its practical implementation may. For example, in the

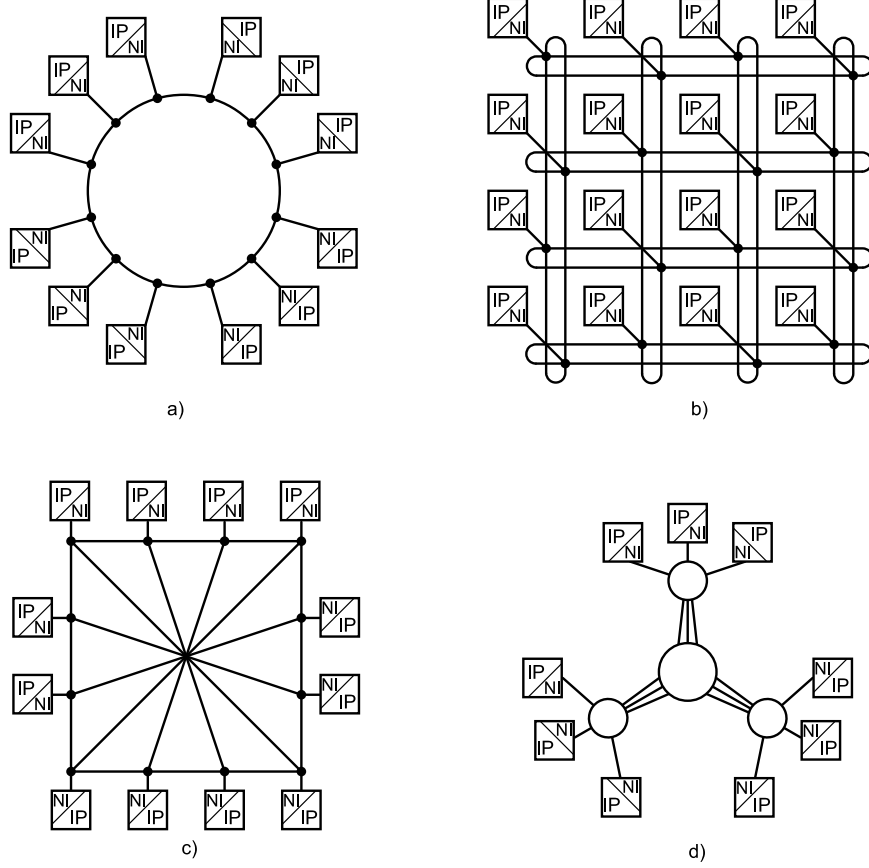


Figure 2.6: a) Ring, b) Torus, c) Spidergon, d) Fat tree topologies.

general case of rate-based arbiters, the credit counters used by the arbiter are still expressed as a binary number with finite precision. This model is valid for networks where allocation is constrained to fixed units of  $1/n$  bandwidth.

In an SDM scheme, an  $m$ -bit wide link can be divided into  $n$   $k$ -bit independent lanes, where  $nk = m$ . This imposes a restriction on the maximum value of  $n$ , since  $n$  has to be a divisor of  $m$ . The cost of routers is also affected by the choice of  $n$  as we will see shortly.

In a network using frame-based arbiters, connections are allocated a certain (integer) number of slots in a frame of size  $n$ . A particular form of frame-based arbitration is time-division multiplexing (TDM) where connections are assigned fixed slots inside the frame. The frame is called a TDM wheel to

emphasize the periodic nature of the schedule.

The size of the TDM wheel or frame has an effect on how long a connection may have to wait for its turn as well as how much buffering will be needed. The higher the  $n$  the longer the waiting period and the larger the buffers. On the other hand, a low  $n$  will result in a coarser division when allocating the link bandwidth to different connections.

A coarser division, i.e. smaller  $n$ , may cause a channel to receive more than its required bandwidth because the allocation is rounded up to the next available value (Figure 2.7). A finer division has lower overhead but higher latency and buffering cost.

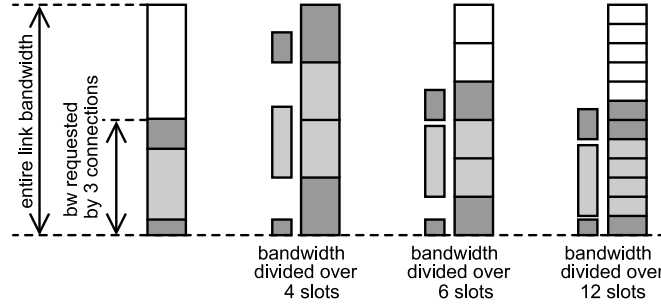


Figure 2.7: Requested bandwidth being allocated over links supporting different link division granularities.

Models starting with Model 4 and below, which enforce a certain link division granularity, are reasonably easy to implement in a classical SDM or TDM scheme. These particular implementations also guarantee that the performance bound offered by the model can be achieved, although the fact that connections can be routed over multiple paths of different lengths may introduce additional complications regarding the reassembly of data at the destination.

A router used in an SDM scheme is depicted in Figure 2.8. Each of the input and output links of the router is split into  $n$  SDM lanes. A router allowing maximum flexibility would allow routing any of the input lanes to any of the output lanes. The hardware complexity of the router is  $O(np^2)$  where  $p$  is the number of ports and a port is considered to be  $n$ -lanes wide.

A TDM router (Figure 2.9) that offers the same routing flexibility as the SDM router, would have to provide buffering space for  $n$  TDM slots for each input, for example in a circular buffer. Data from each of the input buffers may have to be forwarded to multiple output ports during one cycle, thus a multi-

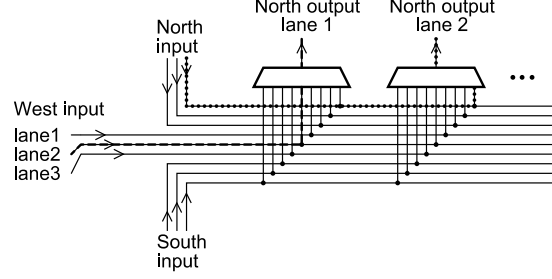


Figure 2.8: Space Division Multiplexing in one router.

port memory is required. The complexity of the hardware, ignoring physical floorplanning issues is again  $O(np^2)$ .

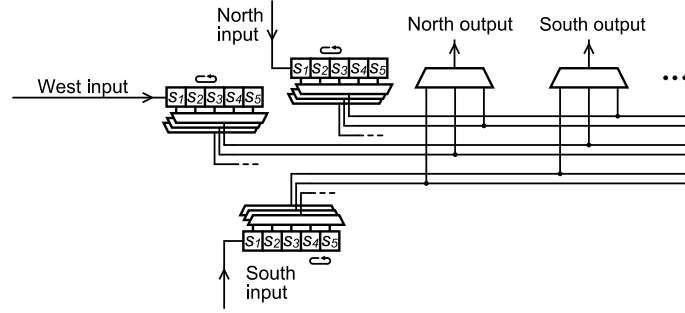


Figure 2.9: Time Division Multiplexing in one router.

### 2.2.3 A generic link-sharing mechanism versus contention-free routing

In the previous section we pointed out that providing maximum flexibility in link bandwidth allocation in a TDM scheme with  $n$  slots requires an expensive crossbar and buffering space for the data transmitted in the  $n$  slots.

It is possible to reduce the required buffer capacity and crossbar size by sacrificing some of the flexibility in choosing the slots allocated to a connection on the incoming and outgoing links. Under the contention-free routing model, the routers employ a buffer for a single data flit (the size of a TDM slot) for each link (Figure 2.10). This implies that data needs to be forwarded by each router in the next slot immediately after its arrival to make space for the next

incoming packet. The hardware complexity of the router is now  $O(p^2)$  and does not depend on the length of the TDM period, or in other words the number of lanes the link is partitioned into.

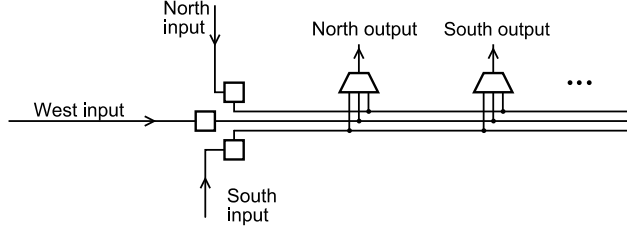


Figure 2.10: Router in a TDM scheme using the contention-free routing model.

Having the data dispatched immediately in the next slot is also beneficial in terms of reducing the worst-case network traversal latency. This very strict limitation however makes the allocation process more difficult because the position of the allocated slots is “locked” on all links on a path belonging to a communication channel.

All models starting with Model 7 use the contention-free routing model.

#### 2.2.4 Multiple paths versus a single path

Networks on chip typically provide path diversity, i.e. multiple possible paths for data to reach its destination. The method of choosing the path in a network is called routing.

Depending on the network implementation, for a single source-destination pair, data may be allowed to travel over a single path or over multiple paths. Conventionally, the first type of routing is called deterministic and the second is called non-deterministic. In [SG09] we describe nevertheless a multi-path routing method which cannot be classified as non-deterministic in the true sense of the word (the route selection is performed in a deterministic manner according to a TDM schedule). We prefer to use therefore in our terminology the notion of multi-path instead of non-deterministic.

All models except Model 10 and Model 12 impose no restrictions on the number of paths used by each communication channel. Channels may even be arbitrarily split over multiple paths, can recombine, and can take arbitrary detours (Figure 2.11).



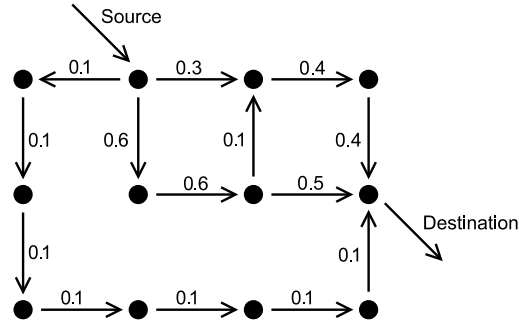


Figure 2.11: Communication channel split over multiple paths.

### 2.2.5 In-order versus out-of-order delivery

One concern when routing over multiple paths is that packets might arrive in another order than they were sent. As the data usually needs to be processed in its original sequence, expensive reordering buffers need to be employed to reorder incoming packets.

In a generic network out-of-order deliveries can be dealt with by using buffers at the destination to hold the packets that arrived too early until the packets that should have arrived before them but did not are also received. In addition to the cost of the buffers this scheme has a bandwidth overhead, because the packets need to carry ordering information.

In contrast, under the contention-free routing model, the network traversal time is known from the allocation phase, and it is possible to know in advance whether packets belonging to the same connection will overtake each other when using different paths. Because the delay per hop is fixed, paths with the same length will always deliver the packets in order, and even for paths with slightly different lengths out-of-order delivery can be avoided by not letting packets be sent on a longer path immediately before sending other packets on a shorter path (Figure 2.12). This introduces nevertheless more restrictions on the allocation process and may result in a loss in performance.

Algorithms that insure in-order delivery are discussed in more detail in Chapter 3.

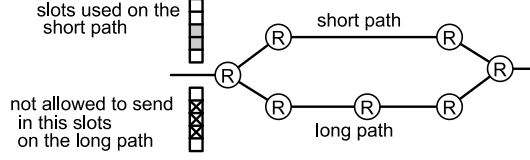


Figure 2.12: Enforcing in-order delivery in the contention-free routing model.

### 2.2.6 Minimal versus non-minimal routing

While architecturally some NoCs support routing over paths longer than the shortest distance between source and destination, it may not be desirable to allow such a routing for reasons of latency or power efficiency.

Allowing non-minimal routing also enlarges the solution space but makes the allocation process more computationally expensive, especially when exhaustive search is used. For design-time allocation we generally assume that non-minimal routing is allowed, but the online allocation algorithm that we will discuss in Chapter 5 only performs a search of minimal routes in order to minimize the allocation time.

### 2.2.7 Header overhead

Data traveling through a network often has to carry additional information like route to destination or internal link or network status which does not represent useful bandwidth. Such data is typically placed ahead of the useful payload and bears the name of header.

While this calculation is specific to *Æthereal*, a certain header overhead can be expected in other network implementations as well. In Chapter 6 we present a network implementation that avoids the header overhead.

Models 8-12 can either take headers into account or ignore the header overhead. When the header overhead is considered, we use the *Æthereal* header size model [Han09]. In *Æthereal*, a network packet occupies between one and  $s_{pkt}$  slots, each slot having  $s_{flit}$  words. Using the default values of  $s_{pkt} = 3$  and  $s_{flit} = 3$  and given that the network header size is one word, the efficiency (useful data versus the total size of the packets) is between 66.6% in the worst case (one-slot packets, 1 header word and 2 payload words) and 88.8% in the best case (with a packet size of three slots, the packet consists of 1 header word and 8 payload words).

Models 8 to 12 can either include or ignore the header overhead.

## 2.3 Allocation methods

Networks on chip offer freedom regarding the path data uses to travel to destination. This freedom can be used to balance the network load in such a way that the link capacity is not exceeded for any of the links. The contention-free routing model imposes additional restrictions regarding the arrival and departure times of packets at each hop which also need to be taken into account when computing the routing.

The allocation step assigns paths to each communication channel and under some models allocates specific TDM slots. It has an important role in the overall network performance. This section describes the allocation algorithms we have used for the different models.

- L-1 For the general models, not taking into account all network implementation details, we use *linear programming* (LP) which globally computes an optimal solution across all communication channels. In this case, the solution to the problem results directly from using a standard optimization tool on the problem formulation. LP problems can be solved efficiently in polynomial time. We will discuss the LP-formulation for Models 1 and 2 in Section 2.3.1.
- L-2 When the link bandwidth has to be allocated in discrete quantities (the invalidation of assumption A-2) the LP method has to be replaced by an *integer linear programming* (ILP) method. The problem formulation given in Section 2.3.1 still applies, but the variables are forced to integer values.
- L-3 In Section 2.3.3 we show how ILP optimization can be used to compute globally optimal slot allocation in a model using contention-free routing (Model 5). This is however extremely expensive and we could only apply it to very small topologies. Unlike LP, ILP problems cannot be solved in polynomial time (the general ILP problem is known to be NP-complete).

For detailed models where using ILP optimization is not feasible, we choose a less computationally-intensive approach. This approach consists of finding a route and schedule for one channel at the time, and marking the used resources

so that they are not used by the other channels. We call this the *iterative* approach.

This approach is not guaranteed to be globally optimal, despite the fact that some of our algorithms produce individual one-channel allocations that are optimal. The resulting performance is sensitive to the order in which the individual channels are allocated. We have found that allocating higher bandwidth channels first provides better performance, and therefore we have used this strategy in our experiments. The same iterative approach was used by the UMARS algorithm [HGR07] as well as tools used in the routing of physical wires [HS01].

For the models supporting multi-path routing the allocation of individual channels is performed using minimum-cost maximum-flow algorithms or iterative exhaustive search. For the single-path routing we use exhaustive search per communication channel.

- L-4 An *iterative* method where the *individual channels are allocated by the flow algorithm* is used for the Model 6. Iterative search using the flow algorithm is discussed in Section 2.3.2.
- L-5 The flow method can be extended to take into account the slot alignment of the contention free routing model, in the same way the Method L-3 extends L-2. The same formulation from section 2.3.3 applies here. The algorithm thus modified is applied to Model 7.
- L-6 The flow method is also used for producing multi-path solutions when in-order delivery is required. In order to cope with in-order delivery and optimize header overhead, additional heuristics, described in Chapter 3, are used. We also fall-back to single-path allocation if it produces a better result than the flow algorithm. This allocation method is used for Models 9 and 11.
- L-7 An *iterative* method where the *individual channels are allocated using single-path exhaustive search* is used for the models which support only single-path routing (Models 10 and 12).
- L-8 In Section 2.3.5, we present an alternative method for producing multi-path solutions by applying the single-path search several times for a single channel. Model 8 uses this method in addition to the standard single-path and flow methods.

In the following we discuss each allocation method in more detail.

### 2.3.1 Linear Programming

This section covers the allocation Methods L-1 and L-2.

#### Network communication formulated as a multi-commodity flow problem

The multi-commodity flow problem [Hu63, Sai68] is an optimization problem dealing with the transport of goods, or *commodities*, inside a network. Commodities can represent material goods or abstract, quantifiable entities like data flowing through a communication network. The network is expressed in the form of a multigraph (typically a graph) having a capacity attached to each of its edges. Each of the commodities has a source, a sink and a demand. The capacity of each edge is divided between the commodities using this link. This formulation applies very well to the case of a network carrying information.

We use a variation of this problem called *the maximum concurrent flow problem* [SM90]. Under this formulation, the optimization target is a factor  $r$  that all demands should be scaled by in order to make the delivery possible for all commodities. That is, the requirements are scaled while keeping their relative ratios constant, to the level where we can provide the maximum delivery. Equivalently we can keep the requirements constant, while scaling the capacity available on all links. This can be seen as an attempt to determine what is the minimum frequency the network needs to run at in order to simultaneously satisfy the bandwidth requirements of a set of applications.

The multi-commodity flow formulation makes use of a conservation law similar to Kirchhoff's current law, which states that, at each node, except the sinks and the sources, the commodity coming in must be equal to the one going out over time. In terms of networks, this is equivalent with the assumption that packets are not dropped.

We present a mathematical formulation of the transport of data through the network. This formulation represents the multi-commodity flow problem as a linear programming problem which is solved using an off-the-shelf LP solver tool, glpsol [Fou]. The LP method produces a solution for the routing problem for Models 3-4. All models using LP allocation assume multi-path routing where out-of-order delivery is allowed (see Figure 2.1).

#### LP formulation of the multicommodity flow problem

Consider the following sets used in our formulation:

- $V$  the set of network nodes (NIs and Routers)
- $E$  the set of network links

- $J$  the set of communication channels
- $R$  the set of routers  $R \subset V$
- $P_r \subset E$  the set of links arriving at router  $r$   
 $P_r = \{p_r \in E | p_r \text{ arrives at router } r\}$
- $Q_r \subset E$  the set of links departing from router  $r$   
 $Q_r = \{q_r \in E | q_r \text{ departs from router } r\}$

Each network communication channel has a source network interface producing data, a destination network interface consuming data, and may use bandwidth on any of the network links. Technically speaking, the data is produced and consumed by IPs that are mapped to the NIs, but we consider that the connection between IPs and NI never represents a bottleneck. We denote by  $y_j$  the bandwidth of data being inserted into the network by the source of channel  $j \in J$  (which is also equal to the data consumed by the destination). The  $y_j$  values are normalized to the bandwidth provided by 1 slot. The link bandwidth is the product of the link operating frequency and link width, divided by the number of slots. We always assume a link width of one word and the number of slots is only important for the ILP method.

We denote by  $z_{ej}$  with  $e \in E, j \in J$  the bandwidth used on each link  $e$  by each communication channel  $j$ , again normalized to the slot bandwidth. The paths used by each communication channel  $j$  can be extracted from the values  $z_{ej}$ .

The following equations and inequalities constrain the solution. They are both necessary and sufficient to guarantee the assumptions of Models 1-4.

Negative bandwidth values are not allowed (Equation 2.1). Some formulations use the sign to represent the direction of movement through an edge but we do not use this approach. As discussed in Section 2.1 we consider links to be unidirectional.

The bandwidth used by each communication channel on each link is always a positive amount. The transport in the opposite direction has to be represented over the return link rather than a negative value.

$$z_{ej} \geq 0, \forall e \in E, \forall j \in J \quad (2.1)$$

The total bandwidth used on one link by all communication channels taken together must be at most equal to the link bandwidth (Equation 2.2).  $s$  is the number of slots.

$$\sum_{j \in J} z_{ej} \leq s, \forall e \in E, \quad (2.2)$$

Models 1 and 2 omit the constraint 2.2 for edges not having an NI as one of their endpoints.

The total amount of data belonging to a communication channel arriving at one router must also depart from the same router, either through router-to-router links or through the links to the source and destination NIs (Equation 2.3).

$$\sum_{e \in P_r} z_{ej} = \sum_{e \in Q_r} z_{ej}, \forall r \in R, \forall j \in J \quad (2.3)$$

if  $e$  is a link originating at an NI then

$$z_{ej} = \begin{cases} y_j & \text{when the NI is the source of } e \\ 0 & \text{otherwise} \end{cases} \quad (2.4)$$

if  $e$  is a link arriving at the NI then

$$z_{ej} = \begin{cases} y_j & \text{when the NI is the destination of } e \\ 0 & \text{otherwise} \end{cases} \quad (2.5)$$

Equations 2.4 and 2.5 ensure that all data originates from the proper source and is delivered to the proper destination.

An allocation consists of finding all  $z_{ej}$  values or, in other words, the bandwidth used by each communication channel on each of the network links.

The number of equations increases with the product of the number of channels and the number of links, more precisely it is  $|J|(2|E| + |V|)$ . In practice the flow problem for an 8x8 mesh network with 200 communication channels could be solved in reasonable time (in the order of minutes) with off-the-shelf LP software glpsol.

When the result is constrained to use only integer numbers (Models 2 and 4) the problem becomes an ILP problem which is much more difficult to solve than its LP counterpart (ILP problems with bounded variables are known to be NP-hard). For our largest tests (8X8 mesh and torus, and fat-tree with 64

nodes), the running times of the glpsol tool (which is also able to solve ILP problems) were in the order of hours and even days.

In Appendix A, we present example code that describes the multi-commodity flow problem in the GNU MathProg modeling language.

### 2.3.2 Iterative, single-channel allocation using the flow algorithm

This section describes Method L-4.

Global optimization methods are generally too expensive for the computation of a channel allocation when we need to take into account all the details of the more complex models, i.e. Models 6-12. A more practical approach consists in allocating channels one by one, each channel allocation blocking its network resources from being used by subsequent allocations (Figure 2.13).

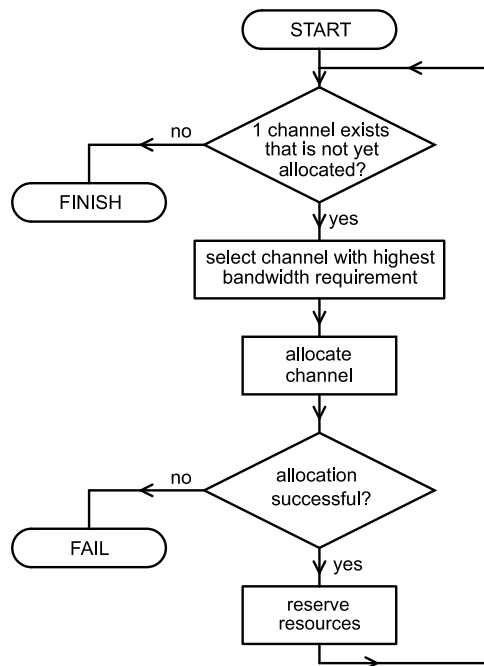


Figure 2.13: Iterative channel allocation flowchart.

Individual channels may be allocated with the flow algorithm, as we will explain in this section, or with other path-finding methods, as we will present in Section 2.3.4 and Section 2.3.5.



The iterative method starts with an empty network, in which all links have their full capacity (Figure 2.14a, for simplicity only links in one direction are represented). As channels are allocated one-by-one (Figure 2.14 b, c, d), part of the available link capacity is consumed. Each new channel allocation can only make use of the link capacity that was not yet consumed by the previous iterations. For example, the 4th channel allocation can make use only of the link capacities in Figure 2.14e.

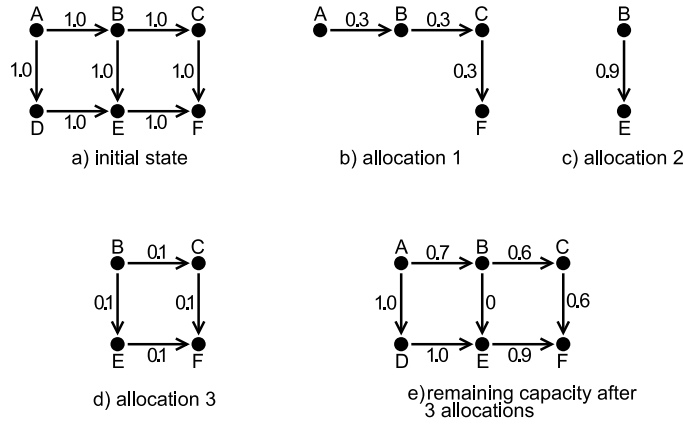


Figure 2.14: Steps in the iterative flow allocation algorithm.

We formalize the method as follows: Let  $V, E, J, R, P_r, Q_r$  have the same meaning as in section 2.3.1. Let  $c_{en}$  be the remaining capacity on link  $e \in E$  after the  $n$ -th allocation,  $c_{e0} = s$ .

The flow problem can be seen as a restriction of the multi-commodity problem in the particular case when the number of commodities equals 1. We can use the same formulation of the flow, except that Equation 2.2 is replaced by 2.6 (we assume channels  $j \in J$  are numbered  $1..n$  and allocated in that order).

$$z_{ej} \leq c_{ej} \quad (2.6)$$

We compute  $c_{e(j+1)} = c_{ej} - z_{ej}$ . The allocation of channel  $j$  consists of finding the  $z_{ej}$  values  $\forall e \in E$ . Although it is possible to use the LP solver this “single-commodity” flow allocation, it is more efficient to find a solution using flow algorithms [EK72]. Flow algorithms can perform real number as well as integer optimization in polynomial time.

Whereas in the global allocation method the distance criterion was unimportant

in deciding the success or failure of the allocation, in the iterative method it is important to select shorter paths, because in this way more resources are left available for the following allocations.

The algorithm we employ is the minimum-cost maximum flow algorithm [EK72], which is optimal in terms of both provided bandwidth and path length. The algorithm is described in more detail in Section 3.2.

The flow algorithm by its nature produces multi-path allocations.

### 2.3.3 Allocation using graph splitting in the ILP and flow methods

This section describes Methods L-3 and L-5 as extensions of the Methods L-2 and L-4.

As mentioned in section 2.2.3, the contention-free routing model imposes additional restrictions with regard to the alignment (in time) of the incoming and outgoing slots. More precisely, a packet arriving at one router has to be forwarded to its output in the immediately next time slot. This is a highly limiting restriction since it locks down the timing of all slots used by one communication channel on the path between source and destination. On the positive side, contention-free routing is very inexpensive to implement in hardware [GH10].

We attempt to determine the penalty this restriction introduces by comparing the performance models that have the restriction with models that do not. Because performance is affected by the allocation algorithm employed, we make use of equally powerful algorithms in the comparison. Model 4 can be compared to Model 5 since both guarantee globally optimal allocations. Model 6 can be compared to Model 7 as both make use of iterative channel allocation using the flow algorithm.

To model contention-free routing link sharing constrains, we use a graph splitting approach. The network is represented by a graph. Using this approach each network node is represented by  $s$  nodes in the graph (is *split* into  $s$  nodes), where  $s$  is the number of slots in the TDM table. Each network link is represented by  $s$  edges in the graph (Figure 2.15). Each graph node represents the possibility of reaching its associated network node in a certain time slot and each graph edge represents one time slot on a network link. Graph edges are connected between nodes in taking into account the hop delay of one time slot.

We model the problem of allocating all communication channels for the

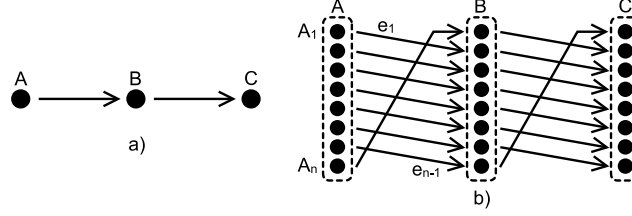


Figure 2.15: Network nodes (a) and graph nodes after split (b).

contention-free routing model using the following set of equations. These equations are similar to the ones used in Section 2.3.1 except for the introduction of third dimension  $k \in 1..s$  to the array  $z$  representing the allocation.

The bandwidth used by each communication channel on each (split) link is either 0 or 1, 1 representing the bandwidth of 1 slot (Equation 2.7.)

$$z_{ejk} \in \{0, 1\}, \forall e \in E, \forall j \in J, \forall k \in \{1..s\} \quad (2.7)$$

The sum of bandwidths used by all channels on the same slot of a link is as well 1 (Equation 2.8.)

$$\sum_{j \in J} z_{ejk} \leq 1, \forall e \in E, \forall k \in \{1..s\} \quad (2.8)$$

At each router the amount of data (belonging to a particular connection) that is incoming during one slot will leave in the consecutive slot (Equation 2.9.)

$$\sum_{e \in P_r} z_{ejk} = \sum_{e \in Q_r} z_{ej((k+1) \bmod s)} \quad \forall k \in 1..s, \forall j \in J \quad (2.9)$$

At the links connecting the NIs to the routers, a communication channel can make use of any of the available slots (Equations 2.10 and 2.11).

if  $e$  is a link originating at an NI then

$$\sum_{k \in \{1..s\}} z_{ejk} = \begin{cases} y_j & \text{when the NI is the source of } e \\ 0 & \text{otherwise} \end{cases} \quad (2.10)$$

if  $e$  is a link arriving at the NI then

$$\sum_{k \in \{1..s\}} z_{ejk} = \begin{cases} y_j & \text{when the NI is the destination of } e \\ 0 & \text{otherwise} \end{cases} \quad (2.11)$$

Performing an allocation consists of finding all  $z_{ejk}$  values. These values indicated exactly which slots on which links should be allocated to each of the communication channels. The solution generated in this way though may use multiple paths and may result in out-of-order deliveries.

The set of equations and inequalities can be solved directly by an ILP solver (Model 5), but because the number of equations is very large, and ILP problems in general cannot be solved in polynomial time, we are only able to run such tests for very small topologies (2x2 or 3x3 meshes with  $s = 8$  and a small number of connections).

A more feasible approach consists in using the iterative approach described previously. In the iterative approach we need to define  $c_{ejk}$ , the remaining capacity on slot  $k$  of edge  $e$  after allocating channel  $j$ . By extension the initial capacity is  $c_{e0k} = 1$ .

Equation 2.8 is replaced by Equation 2.12:

$$\sum_{j \in J} z_{ejk} \leq c_{ejk}, \forall e \in E, \forall k \in \{1..s\} \quad (2.12)$$

And  $c_{e(j+1)k}$  is computed as  $c_{e(j+1)k} = c_{ejk} - z_{ejk}$ .

Channel allocations are performed one-by-one, according to the flow-chart in Figure 2.13. One allocation consists of computing the set of values  $z_{ejk}$ ,  $\forall e \in E, \forall k \in \{1..s\}$  for a fixed  $j$  which is the channel being allocated.

As the flow algorithm produces multi-path allocations, further steps must be taken in order to insure in-order delivery. These steps, consisting in verifying path lengths and discarding inappropriate paths will be described in detail in Chapter 3. When minimal routing is used (Model 11), delivery is always in-order (since data has a fixed propagation delay.)

### 2.3.4 Single-path exhaustive search

This section describes Method L-7.

The single-path exhaustive search method is also an iterative method, allocating connections one by one. Starting with an empty network (Figure 2.16a), it finds paths between source and destinations for each connection in turn and every time a path is found it marks the slots used on that path so that they can not be used by subsequent connections (Figure 2.16b,c,d). Single path exhaustive search is only used for the slot-based models (Models 10 and 12 use single-path search exclusively, models 9 and 11 use it as a fall-back option, and model 8 uses it as part of repeated single-path search.)

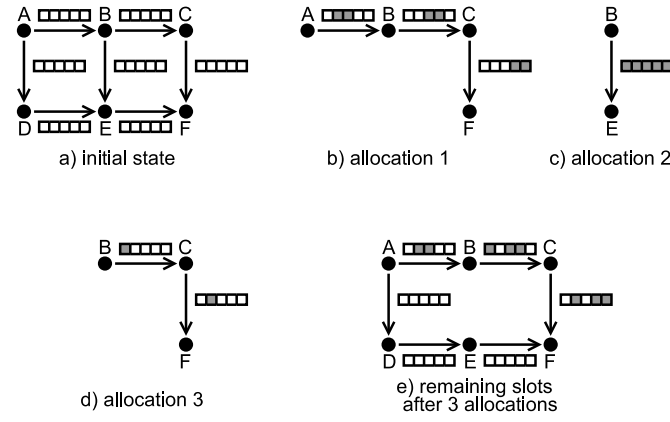


Figure 2.16: Steps in the iterative single-path allocation algorithm.

We can formalize the single path method using the same notation used in the previous section. Let  $V, E, J, R, P_r, Q_r$  have the meaning defined in the previous sections.

Let  $z_{ejk}$  be the bandwidth used on slot  $k$  of link  $e$  by channel  $j$  and  $c_{ejk}$  the remaining capacity on the same slot-link after the allocation of channel  $j$ , with  $c_{e0k} = 1$  being the initial capacity.

The problem consists in finding a path  $P = (e_1, e_2 \dots e_n)$  with  $e_x \in E$ ,  $\forall x \in \{1..n\}$ ,  $e_x$  being adjacent edges, the first one departing from the source node and the last one arriving at the destination:  $e_1 = (source_j, v_1)$ ,  $e_2 = (v_1, v_2)$  ...  $e_n = (v_{n-1}, destination_j)$ , and a set of slots  $S = \{s_{a_1}, s_{a_2} \dots s_{a_y}\}$ ,  $y > 0$  that satisfies the bandwidth requirement  $bw(S) \geq y_j$ .  $S$  represents the set of slots used by the communication channel on the ingress edge. The bandwidth provided by the set of slots  $S$ ,  $bw(S)$  is a function of the set of slots. When no header overhead is assumed, the value of the provided bandwidth is proportional to  $|S|$ . If header overhead is assumed the computation of this value is more complex and it is discussed in Chapter 4.

All the slots used by the path  $P$  must be available:

$$c_{(e_x)j((a_w+x-1) \bmod s)} = 1 \quad \forall x \in \{1..n\}, \quad \forall w \in \{1..y\} \quad (2.13)$$

The value of  $c$  after the allocation of  $j$  is computed as follows:

$$c_{e(j+1)k} = \begin{cases} 0 & \text{if } \exists x \in \{1..n\} \text{ s.t. } e = e_x \text{ and } s_{((k+x-1) \bmod s)} \in S \\ c_{ejk} & \text{otherwise} \end{cases} \quad (2.14)$$

The strategy we use to find paths is exhaustive search. Paths of minimal length are enumerated first and for each path we verify if the maximum set of available slots is sufficient to satisfy the bandwidth requirement. If none of the paths can provide sufficient bandwidth because of the previous allocations (for example if in Figure 2.16e, a channel from B to F requires 3 slots whereas only two are available), we attempt to find all paths of minimal length plus 1, then minimal length plus 2, until a solution is found or the algorithm can determine that no solution is possible. To avoid excessively long running times, the search is abandoned without completely exploring the solution space when  $10^7$  paths were examined or the path length exceeds the distance between source and destination plus 16, i.e., we allow a detour of 16 hops. The complete details of the single-path allocation algorithm are presented in Section 3.1, while the different options to compute  $S$  are covered in Chapter 4.

In all our algorithm implementations we use the same data structure to keep track of the available slots between allocations (the  $c_{ejk}$  array) and as a result it is possible to allocate some of the connections using the flow algorithm and others using the single-path approach.

### 2.3.5 Multi-path using repeated single-path exhaustive search

This section described Method L-8.

Allocating channels over multiple paths conceptually offers more freedom to routing, but the flow algorithm which produces multi-path allocations might produce in some instances worse results than the single path solution because it produces paths that are too fragmented and use more headers or it produces paths of different lengths that need to be discarded for in-order delivery. For this reason we introduce an additional method, *repeated single-path exhaustive search*, to compute multi-path solutions for Model 8.

This method computes a multi-path solution, by calling the single-path exhaustive search algorithm repeatedly for a single channel. This method, illustrated

in Figure 2.17, takes advantage of the single-path exhaustive search algorithm to minimize the header overhead.

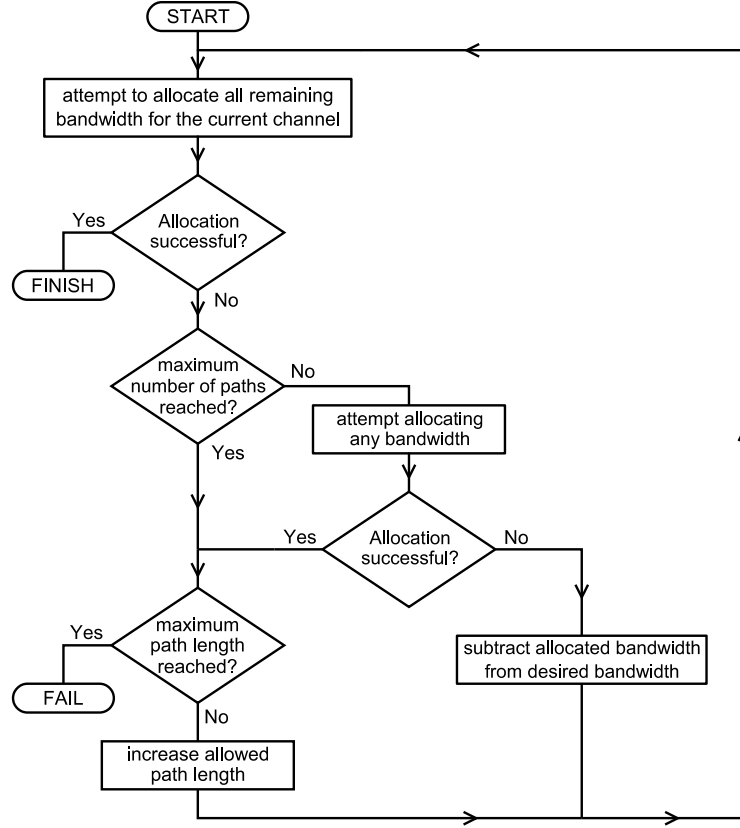


Figure 2.17: Multi-path using iterative exhaustive search (for a single channel).

The multi-path iterative exhaustive search will be described in depth in Section 3.5. Model 8 in our experiments uses both the (repeated) single-path and the flow methods for computing an allocation.

## 2.4 Performance comparison of the proposed models

In this section, we use a set of benchmarks to evaluate the performance of the various network models and of the allocation algorithms.

The basic unit of benchmarking in our evaluation is a *usecase*. By *usecase*

we mean a set of connections between specific sources and destinations, each of them having a given bandwidth requirement. Latency constraints are not included in this evaluation as they are not supported by most models. For a successful allocation the interconnect described by each model must be able to support all connections in one usecase simultaneously. The network frequency needed to support the bandwidth of all connections simultaneously is used as a figure of merit for each of the models.

As a matter of convention we use a default link width of 32-bits plus side-band information which is not available as part of the allocatable bandwidth. This matches the default parameters of the *Æthereal* network which can be used to implement Models 10 and 12, as well as our proposed hardware implementation which we will present in Chapter 6. The bandwidth specified in the models is the product of the frequency and the link width.

Note that the frequency and link width do not have meaning as absolute values. The same bandwidth can be delivered by a network with half the frequency and links twice as wide. In these experiments we are only interested in the relative performance of the models and we ensured that link widths are consistent across the models.

For the LP methods, the frequency is obtained directly by selecting the link bandwidth as the optimization target and dividing the result reported by the LP solver by the link width. For the iterative methods, it is found using binary search.

We employ three sets of benchmarks:

- random benchmarks: The IP pairs are chosen randomly with uniform probability, with one additional constraint that each IP must be at the end of at least one communication channel. The bandwidth requirement for each communication channel is also chosen randomly, with uniform probability in the interval [10..400 MByte/s].
- permutation traffic: IPs communicate in pairs, according to well defined patterns [DT03]. The required bandwidth is the same for all connections.
- task graphs of real applications reported in the literature: a multimedia system [HM05], a digital TV system [HG11], MPEG-2 [LCOM08], VOPD and MWD [VdTJ02].

Two types of mapping are used in the experiments. We call the first one *direct mapping* as it statically assigns the first IP to the first NI, second IP to the



second NI and so on. The second one is the mapping produced by the UMARS algorithm [HGR07] which also targets the contention-free routing model. The UMARS algorithm tries to reduce the distance between communicating nodes, sometimes having the adverse effect of mapping multiple IPs to the same NI when that is not required (the direct mapping will result in IPs being mapped to different NIs as long as the number of NIs is at least as large). We found that the UMARS mapping behaves worse in many cases under random traffic.

In the graphs that indicate performance, by performance we mean the inverse of the frequency of the network needed in order to support the usecase (supporting the usecase means guaranteeing the required bandwidth). High performance thus corresponds to a low required operating frequency. For each combination of topology and allocation method we compute the average frequency over 20 usecases.

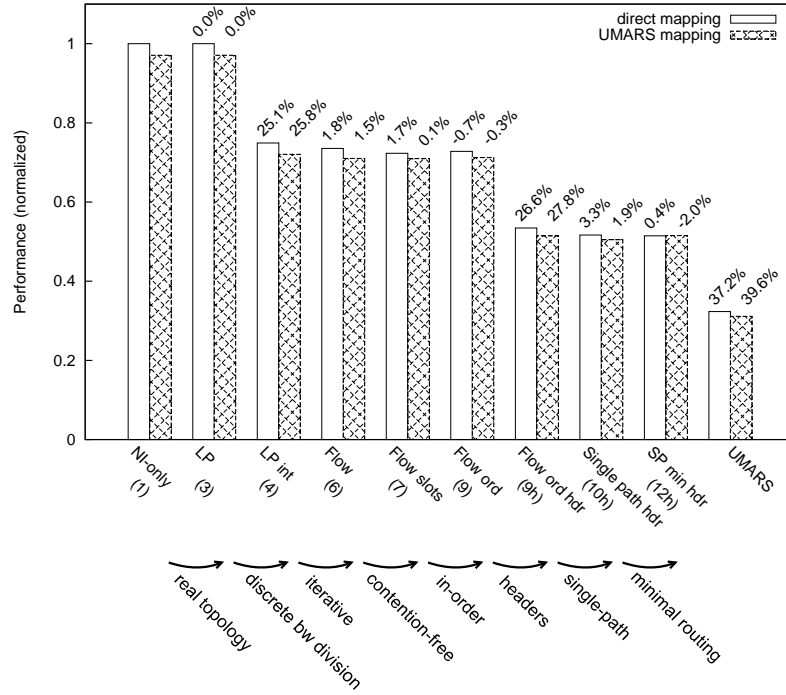
## Table of Experiments

<b>E2.1</b>	4x4 mesh network, random traffic, 16 IPs, 40 connections, 16 slots . . .	47
<b>E2.2</b>	4x4 torus network, random traffic, 16 IPs, 40 connections, 16 slots . . .	48
<b>E2.3</b>	Quaternary fat tree network (2 levels), random traffic, 16 IPs, 40 connections, 16 slots . . . . .	49
<b>E2.4</b>	16 node spidergon network, random traffic, 16 IPs, 40 connections, 16 slots . . . . .	50
<b>E2.5</b>	16 node ring network, random traffic, 16 IPs, 40 connections, 32 slots .	51
<b>E2.6</b>	4x4 mesh network, random traffic, 16 IPs, different number of connections, direct mapping, 16 slots . . . . .	53
<b>E2.7</b>	4x4 mesh network, random traffic, 16 IPs, different number of connections, UMARS mapping, 16 slots . . . . .	54
<b>E2.8</b>	4x4 mesh network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots . . . . .	55
<b>E2.9</b>	4x4 torus network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots . . . . .	56
<b>E2.10</b>	4x4 Mesh network, permutation traffic, 16 IPs, 16 slots . . . . .	57
<b>E2.11</b>	4x4 Torus network, permutation traffic, 16 IPs, 16 slots . . . . .	58
<b>E2.12</b>	Quaternary fat tree network (2 levels), permutation traffic, 16 IPs, 16 slots . . . . .	59
<b>E2.13</b>	16-node Spidergon network, permutation traffic, 16 IPs, 16 slots . . . .	60
<b>E2.14</b>	16-node ring network, permutation traffic, 16 IPs, 16 slots . . . . .	60
<b>E2.15</b>	8x8 mesh network, permutation traffic, 64 IPs, 32 slots . . . . .	61
<b>E2.16</b>	8x8 torus network, permutation traffic, 64 IPs, 32 slots . . . . .	62
<b>E2.17</b>	Quaternary fat tree (3 levels), permutation traffic, 64 IPs, 32 slots . . .	62
<b>E2.18</b>	64-node Spidergon network, permutation traffic, 64 IPs, 32 slots . . . .	63
<b>E2.19</b>	4x4 mesh network, MultiMedia System [HM05] traffic, 16 IPs, 16 slots	64
<b>E2.20</b>	2x4 mesh network, digital TV [HG11] traffic, 8 IPs, 60 slots . . . . .	65
<b>E2.21</b>	3x3 mesh network, MPEG-2 [LCOM08] traffic, 9 IPs, 16 slots . . . . .	66
<b>E2.22</b>	4x3 mesh network, VOPD [VdTJ02], 12 IPs, 16 slots . . . . .	66
<b>E2.23</b>	4x3 mesh network, MWD [VdTJ02], 12 IPs, 16 slots . . . . .	68
<b>E2.24</b>	Network frequency versus topology size for the same usecases (random traffic) allocated on all topologies . . . . .	70

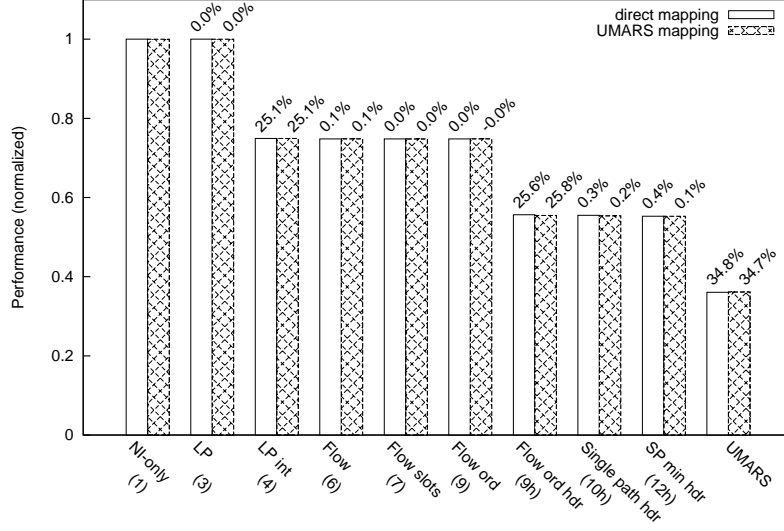
### 2.4.1 Small networks with random traffic

Experiments 2.1-2.5 present the performance of different models under random traffic on five topologies: mesh, torus, fat tree, spidergon and ring. For clarity, we selected a subset of the available models, representing a walk through the model graph, from the most to the least general of the models. Each step in this walk represents a particular NoC design choice in Figure 2.1.

In all the following experiments, the model numbers are indicated in parentheses. Model numbers suffixed with the letter “h” indicate models which include the header overhead computation. In each graph the values are normalized to the performance of Model 1, which is the most general and least restrictive. Since this model is topology agnostic the normalized values are consistent between the graphs representing different topologies, i.e. Experiments 2.1-2.5. The values represent averages over 20 randomly generated usecases.



Experiment 2.1: 4x4 mesh network, random traffic, 16 IPs, 40 connections, 16 slots.

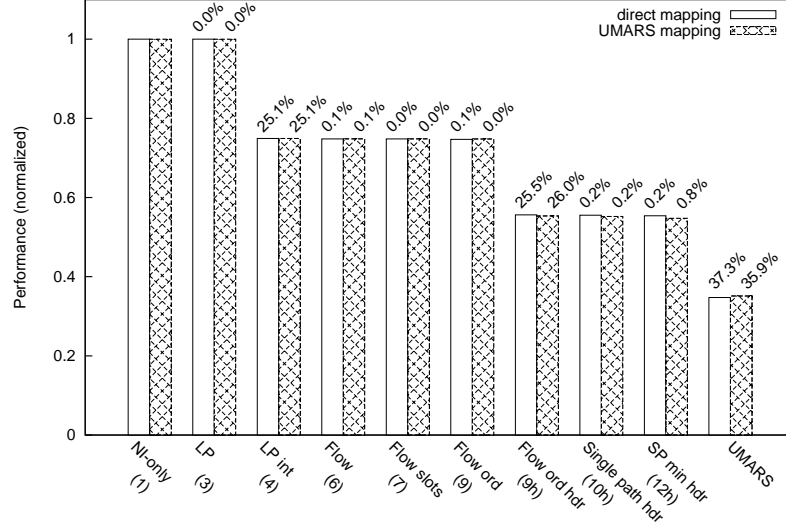


Experiment 2.2: 4x4 torus network, random traffic, 16 IPs, 40 connections, 16 slots.

In general, the less restrictive the model, the better the performance, but that does not necessarily hold true when the allocation method is not guaranteed to be optimal (LP Models 1-5 are optimal, but the iterative methods are not). The drop in performance of each model, compared to the model before it (using the same mapping) is annotated in each experiment.

The difference between Models 1 and 3 (in each graph) shows how much the network topology restricts performance compared to an ideal topology, assuming ideal switching, buffering and link sharing model. We find that topologies in general are not a large factor constraining performance, at least under random traffic, a notable exception being the ring topology. In the ring topology, mapping the communicating IPs as close as possible to each other is particularly important and the mapping produced by UMARS results this time in better performance.

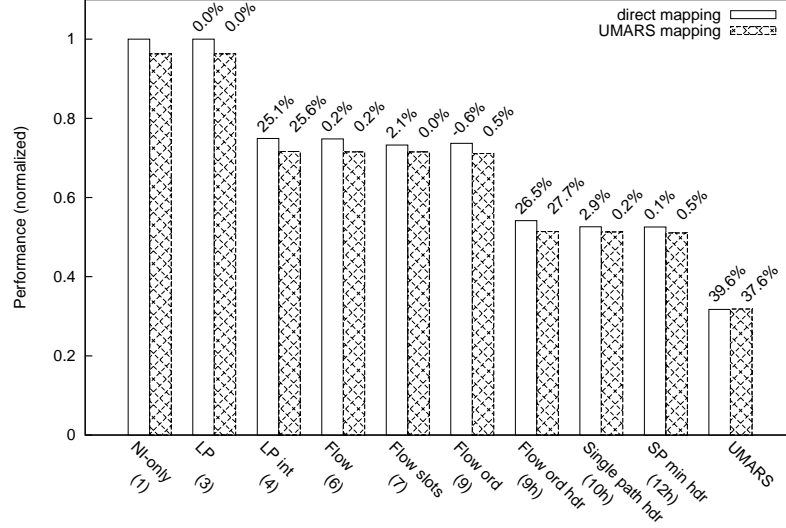
This is easily explained if we compare the average usage of NI-to-router links (we will call them *peripheral links*) to the average usage of router-to-router links (we will call them *core links*). In a mesh, all the data transmitted through *peripheral links* will be divided after the first router over several *core links* (4 such links if the source node is located at the center of the mesh, 3 links when



Experiment 2.3: Quaternary fat tree network (2 levels), random traffic, 16 IPs, 40 connections, 16 slots.

located on the edges and 2 links when located in the corners). The load being spread over multiple *core links* it is less likely that the *core links* will constitute a bottleneck. On the other hand, *core links* are shared between communication channels between different source-destination pairs. A simple estimation of the relative occupation of *peripheral links* versus *core links* can be performed in the following way. Each communication channel uses exactly 2 *peripheral links* and a variable number of *core links*, dependent on topology. In the case of the mesh and torus, the number of *core links* used by a communication channel is in the order of  $O(\sqrt{n})$ , in the case of the fat tree  $O(\log n)$  and in the case of the ring and spidergon  $O(n)$ . At the same time the mesh has slightly less than 2 *core links* for each *peripheral link*, the torus exactly 2, the fat tree  $O(\log n)$ , the spidergon 1.5 and the ring only 1. It is easy to see that contention on the *core links* of the ring and spidergon networks will quickly increase with the value of  $n$ .

One way to mitigate the saturation of *peripheral links* would be to use multiple links to connect each NI to one or more routers. One solution in this direction is proposed in [CFD<sup>+</sup>11]. This would increase the pressure on *core links* and thus making better use of the network topology.



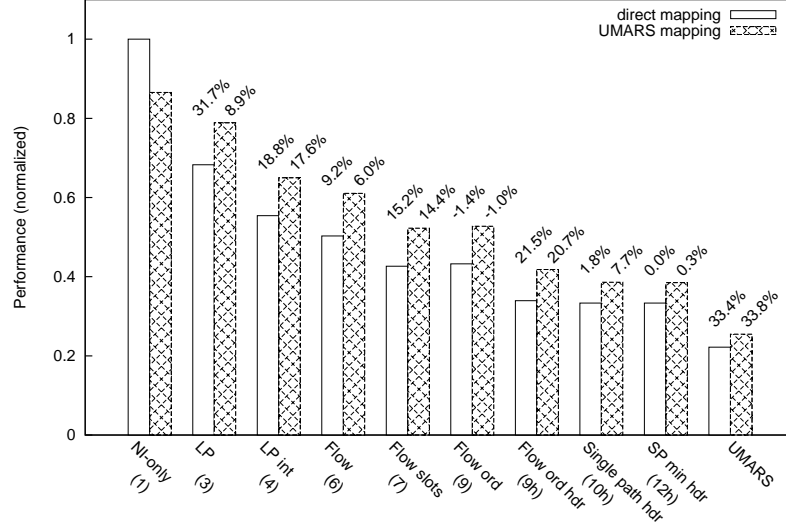
Experiment 2.4: 16 node spidergon network, random traffic, 16 IPs, 40 connections, 16 slots.

A large drop in performance is observed from Model 3 to Model 4. This corresponds to the effect of discrete link division, as explained in Section 2.2.2. We use TDM tables with 16 slots, which is in the middle of the usual range (more generally, this is the performance of any network using frame-based arbitration with a frame size of 16 or a rate-based network that enforces allocation in units of 1/16 of the link bandwidth). We make an exception for the ring network where 16 slots were insufficient for a successful allocation and a TDM table of size 32 was used instead.

The difference between Models 4 and 6 marks the switch from an optimal allocation algorithm to the iterative algorithms. The iterative algorithms exhibit very good behavior providing results almost as good as the optimal algorithm.

The effect of the contention-free routing constraints can be seen in the difference between Models 6 and 7. We find this to be most surprising and a strong argument in favor of the contention-free routing model, since the hardware cost to support this model is much lower. While Model 6 assumes optimal buffering Model 7 uses virtually no buffers inside the routers.

The effect of guaranteeing or not guaranteeing in-order delivery can be seen in the performance difference between Models 7 and Model 9. Under Model



Experiment 2.5: 16 node ring network, random traffic, 16 IPs, 40 connections, 32 slots.

9, in-order delivery is achieved by dropping paths that would cause out-of-order delivery due to difference in path length. The algorithm is analyzed in-depth in Section 3.4. From the results it would seem that there is very little to be discarded, although the difference in performance between the two models is not necessarily equal to the bandwidth discarded (this bandwidth can be allocated over other paths). In-order delivery does not incur a significant performance penalty.

The impact of header overhead can be observed in the drop in performance between Model 9 and Model 9h. The difference in performance falls in the expected range of 11.1% to 33.3% as discussed in Section 2.2.7.

The difference between the models (and allocation algorithms) supporting multi-path routing and those supporting single path routing only, can be seen in the variation in performance from Model 9h and Model 10h. Multi-path routing offers a modest performance benefit.

The difference in performance between Models 10h and 12h is due to enforcing a minimal routing strategy. This difference is also very low.

We also compare our allocation algorithms against the allocation produced

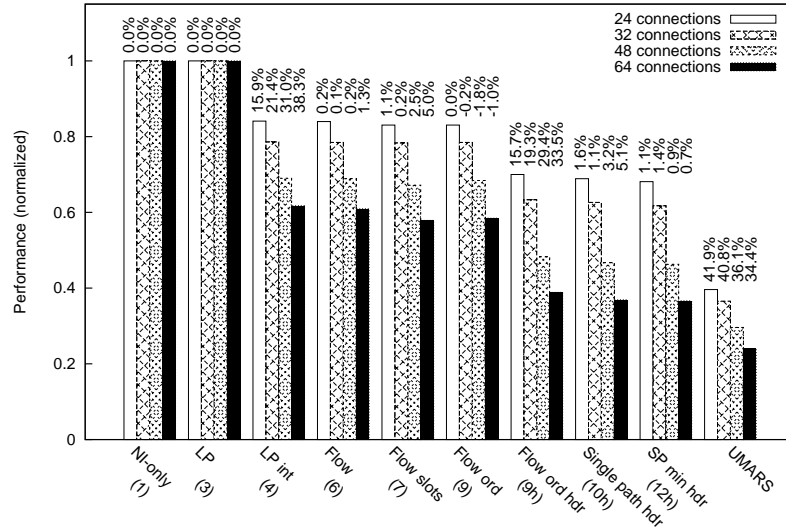
by the UMARS approach [HGR07] which is used by the *Æthereal* design flow. We are able to perform this comparison because we used the same conventions regarding the link width and bandwidth computation. Note that the *Æthereal* mapping that is used in the experiments is also computed by UMARS. From the hardware point of view *Æthereal* supports Model 10 with header overhead so the difference in performance is a result of differences in the allocation algorithms employed. These differences are discussed in more detail in Chapter 3 and Chapter 4. Our algorithms compare favorably to UMARS.



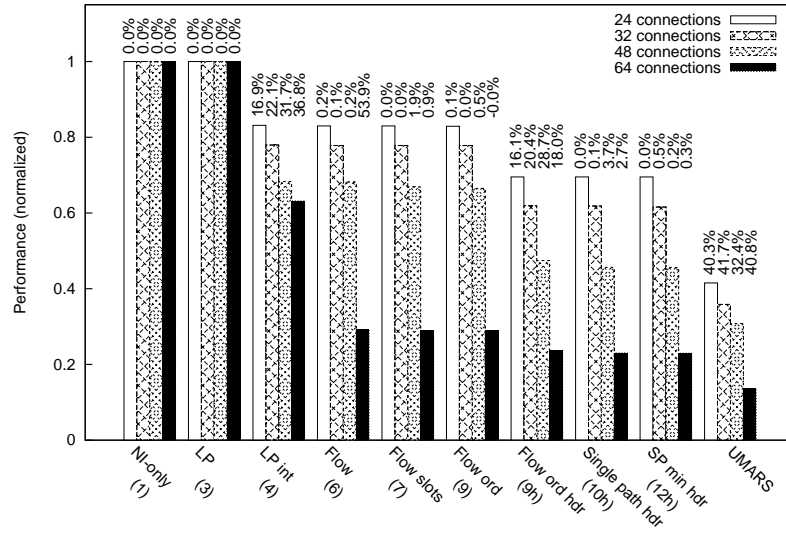
### Random traffic with different numbers of connections

We analyze the effect of varying the number of connections for a network of fixed size (Experiments 2.6 and 2.7). We present here the results for the mesh topology only, as the results for the other topologies follow the same trend.

We observe that the performance penalty related to the link division granularity (Model 3 versus Model 4) and header overhead (Model 9 versus Model 9h) increases with the number of connections, which is not unexpected given that a higher number of connections will require a finer division of the link bandwidth with fewer allocated slots per connection. When only one slot is allocated per connection, the theoretical value of header overhead is 33.3%, which is very near to the value found experimentally. An interesting observation is that the benefit of the multi-path solutions over the single-path is higher for a higher number of connections.



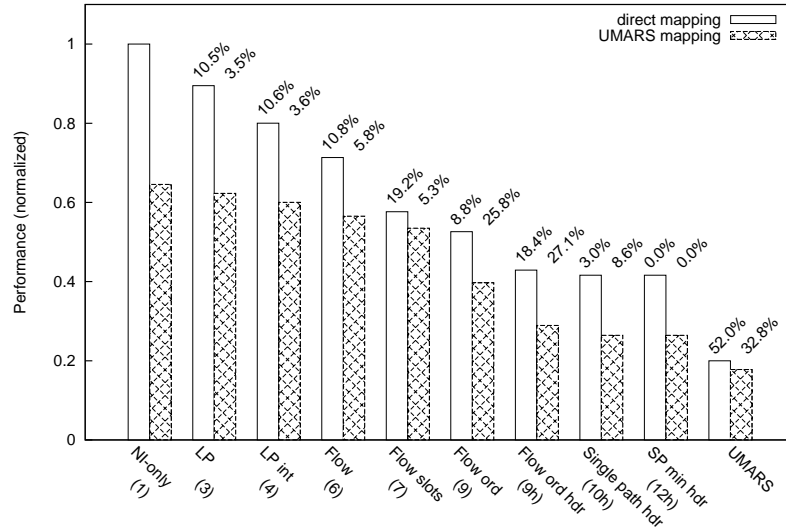
Experiment 2.6: 4x4 mesh network, random traffic, 16 IPs, different number of connections, direct mapping, 16 slots.



Experiment 2.7: 4x4 mesh network, random traffic, 16 IPs, different number of connections, UMARS mapping, 16 slots.

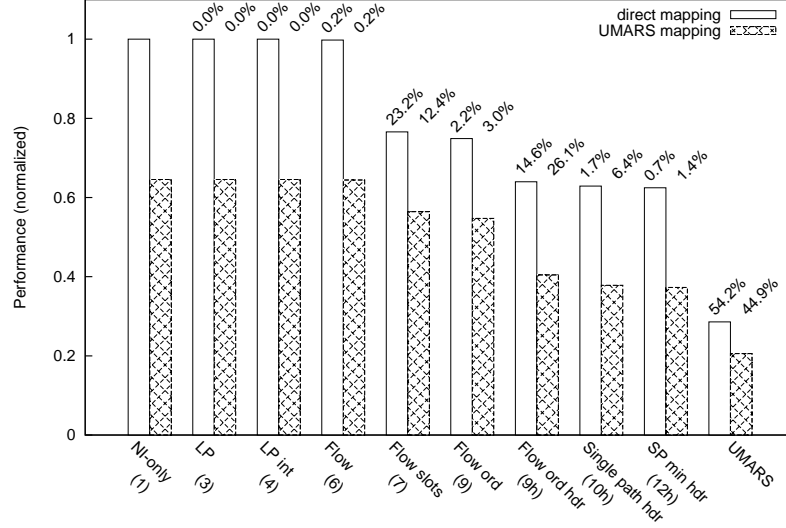
### 2.4.2 Uniform (random) traffic

Despite choosing connection destinations and requested bandwidths in a uniform manner, the random traffic used in the previous experiments is not uniform because connections are persistent and through random assignments some destinations and sources statistically receive higher bandwidths than others. If the connections were not persistent the load would be averaged over time; this is the situation described in the literature as uniform random traffic. We perform another set of experiments where we enforce equal inbound and outbound traffic from all IPs. We achieve this by requesting an equal number of incoming and outgoing connections for each IP and setting the connection bandwidth requirements (and the channel bandwidth requirements) to be equal. The results are presented in Experiments 2.8 and 2.9.



Experiment 2.8: 4x4 mesh network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots.

We observe that, compared to the plain random traffic scenario for the mesh network there is consistently a larger gap between the models. There is an unusually large gap between Model 1 and Model 3 (this only happened previously in the case of the ring network, Experiment 2.5) which means that one of the *core links* is saturated first, even with ideal routing and load balancing (Model 3 uses a globally optimal allocation algorithm). This is not



Experiment 2.9: 4x4 torus network, uniform (random) traffic, 16 IPs, 2 connections/IP, 16 slots.

entirely unexpected because no individual *peripheral link* will saturate before another (the usage on all *peripheral links* is the same), but the load of *core links* is not balanced in the same way.

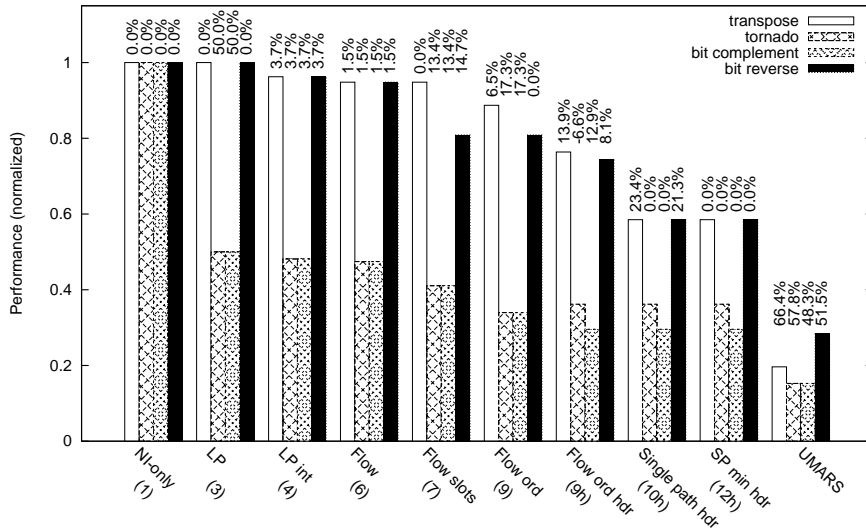
The traffic however poses no challenges to the torus network which offers better path diversity and better chances at balancing load. Because in these experiments the channels had equal bandwidth and the number of slots was divisible by the number of channels (16 slots and 2 channels) it is possible for the bandwidth division granularity to not incur any cost (Model 4 versus Model 3 in Experiment 2.9). Iterative allocation methods are also helped by the increased path diversity with little difference between the iterative and globally optimal allocation methods that operate on the same hardware architecture (Model 6 versus Model 4).

This time, the mapping produced by the UMARS algorithm behaves worse in all models. When UMARS itself is also used for the allocation the *Æthereal* mapping is still worse, but by a lower margin. The fact that the *Æthereal* mapping is worse under Model 1, indicates that UMARS attempted to map multiple IPs to the same NI, possibly trying to shorten the distance from source to destination, but this proved to be a suboptimal choice.

### 2.4.3 Permutation traffic in small networks

Another type of traffic commonly used in testing network performance is the permutation traffic [DT03]. In permutation traffic each node has a connection to only one other node. The pattern of connection is generated in a way such as to stress a particular network topology or to simulate a step in a parallel algorithm (for example the parallel fast Fourier transform). Because permutation traffic targets combinations of specific positions in the topology only the *direct mapping* is used. All connections have equal bandwidth requirements. The number of slots was set to 16.

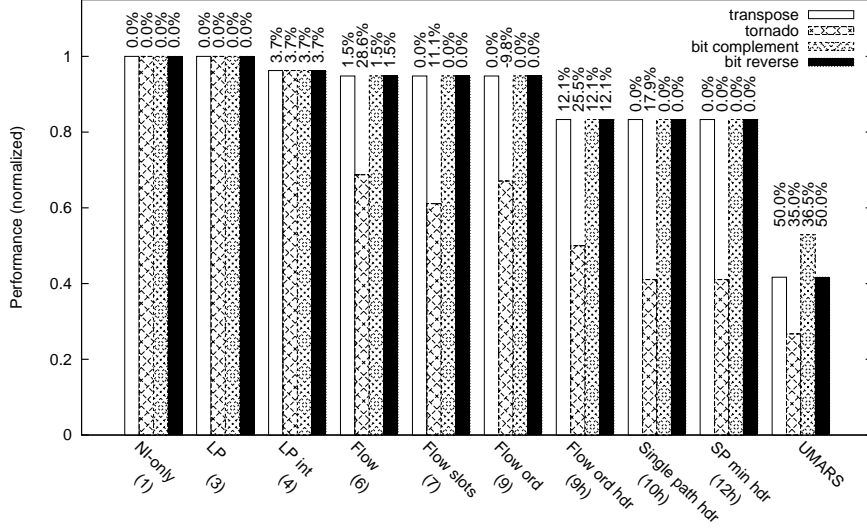
Experiments 2.10-2.14 show the performance of the proposed models under permutation traffic.



Experiment 2.10: 4x4 Mesh network, permutation traffic, 16 IPs, 16 slots.

The different permutation patterns are defined as follows (it is assumed that the source and destination are encoded as integers between 0 and  $n - 1$ ,  $n$  usually being a power of 2 and a square number):

1. for **bit complement traffic**, the destination is obtained by inverting the bits of the source. This is equivalent to saying that node  $i$  communicates with node  $n - 1 - i$ .

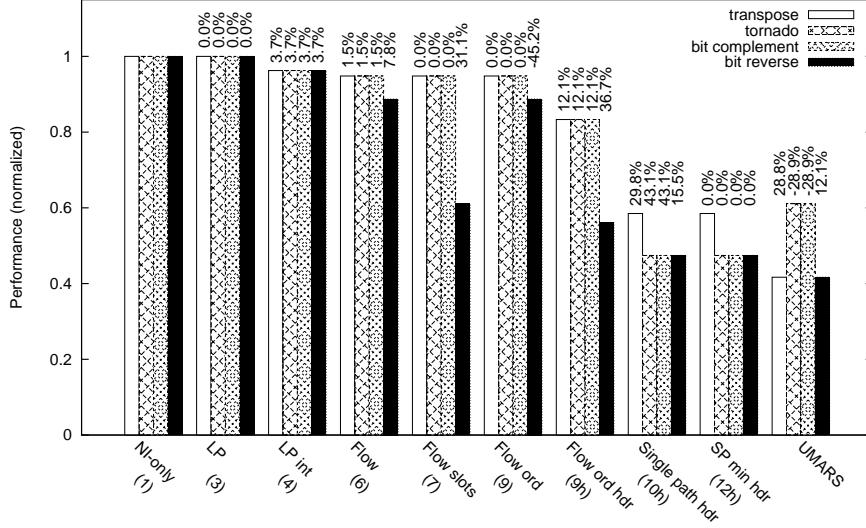


Experiment 2.11: 4x4 Torus network, permutation traffic, 16 IPs, 16 slots.

2. for **bit reverse traffic** the destination is obtained by inverting the bit positions in the encoding of the source. For example if the source node is  $s = 000110_{(2)}$  then the destination is  $d = 011000_{(2)}$ .
3. for **transpose traffic** sources and destinations are assumed to be located in a square matrix (as it is the case with the mesh and torus). Each node communicates to its symmetric over the main diagonal.
4. for **tornado traffic**, with nodes arranged in a square matrix the destination  $d$  is at a constant distance  $k = \sqrt{n}/2$  from the source on both the horizontal and vertical axes.

The definitions for transpose and tornado traffic types were adapted for 2-dimensional mesh and torus networks. The 2-d versions of the communication pattern are also used for the ring network.

Permutation tests are more demanding in terms of *core link* usage. The matrix transpose and bit complement are especially difficult cases for the mesh network. The tornado traffic was specifically designed to stress the torus and ring topologies. On the other hand, they use a lower number of connections than our random traffic tests: only one connection per IP. Consequently there

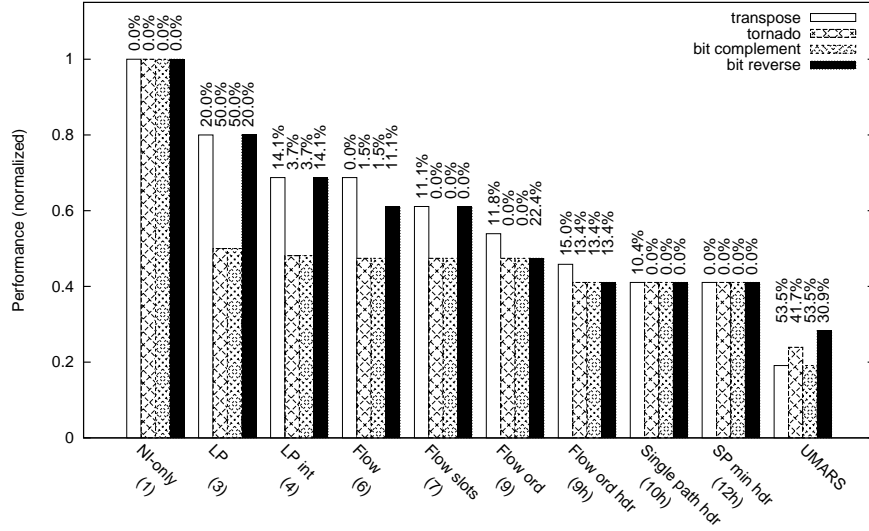


Experiment 2.12: Quaternary fat tree network (2 levels), permutation traffic, 16 IPs, 16 slots.

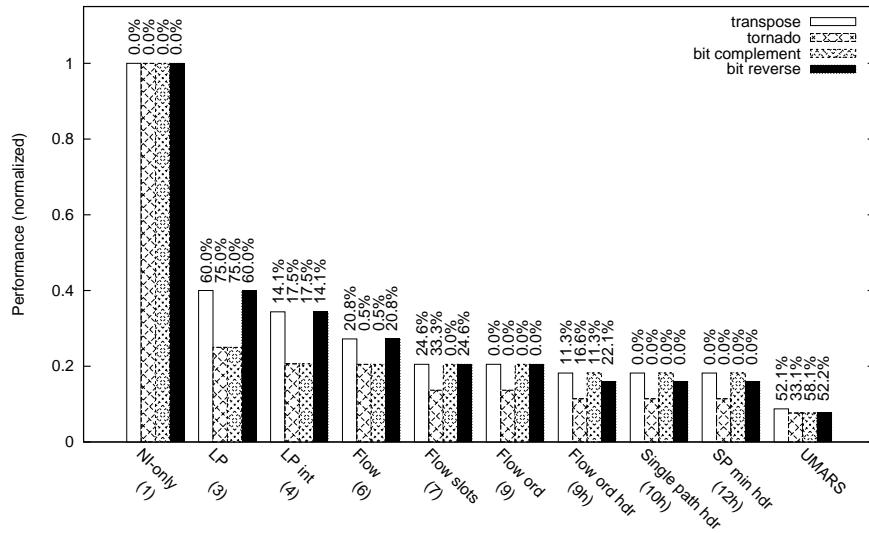
is a smaller drop in performance due to link division granularity (Model 3 versus Model 4). The header overhead is also reduced, this is because fewer connections (and as a result fewer communication channels) result in more slots allocated per communication channel and consecutive slots form larger packets with lower relative header overhead (see Chapter 4).

There is a larger difference in performance between the networks and allocation algorithms supporting multi-path and the ones allowing only single-path allocation (Model 9h versus Model 10h). Iterative methods (Model 6 versus Model 4) also display a loss that is more significant than the case of random traffic.

Although the fat tree network has ideal behavior under Models 1-4 and 6, when using the contention-free routing the performance degrades more than in the case of the torus network which provided excellent performance in all models. In general, the fat tree is considered a better topology, but apparently this is not so when using the contention-free routing. We attribute this to a mismatch in slot alignment for paths that traverse a different number of levels in the fat-tree.



Experiment 2.13: 16-node Spidergon network, permutation traffic, 16 IPs, 16 slots.



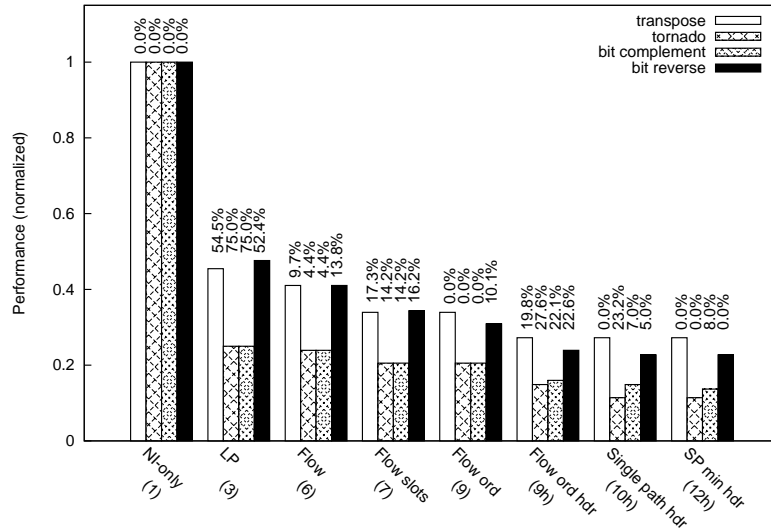
Experiment 2.14: 16-node ring network, permutation traffic, 16 IPs, 16 slots.



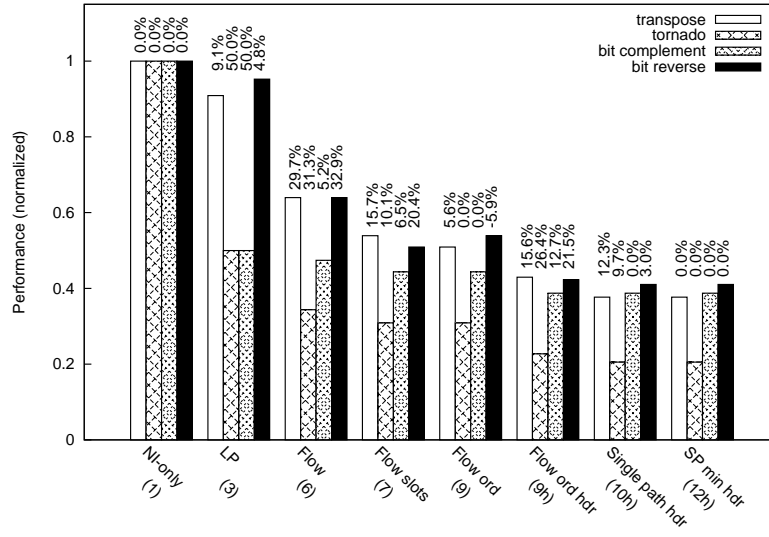
### 2.4.4 Permutation traffic in large networks

A set of tests over larger topologies (Experiments 2.15-2.18) shows the scaling properties of different topologies. The number of slots was increased to 32 to allow finer granularity in sharing *core links*. The ring topology did not allow successful allocation in the majority of cases and was omitted. The UMARS allocator could not deal with networks of this size.

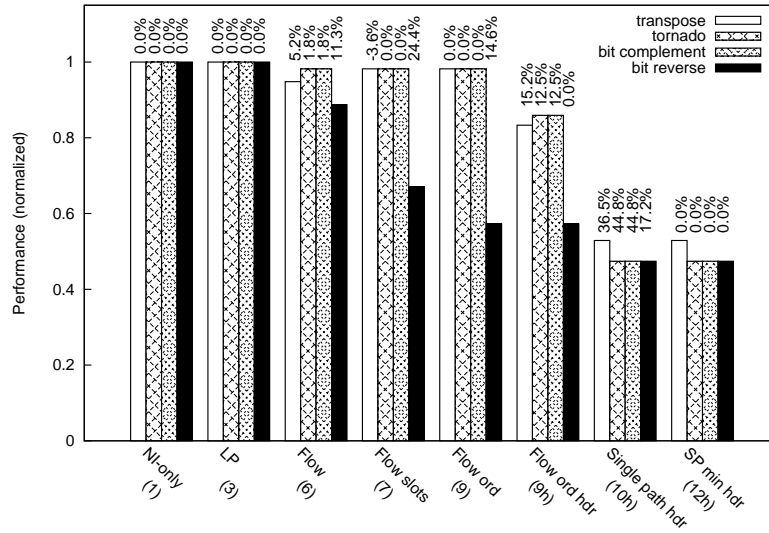
The worst scaling properties belong to the Spidergon topology. The performance of the torus network drops now below that of the fat tree which remains at levels similar to the experiments using 16 nodes. Notable observations are that the topology itself is the main cause of performance loss for the mesh and Spidergon (Model 1 versus Model 3), but the other constraints less so. The iterative allocation method performs much worse than the global optimization on the torus topology (Model 6 versus Model 3) which is something we did not find in the other experiments. Finally multi-path routing performs much better than single-path on the fat tree network (Model 9h versus Model 10h).



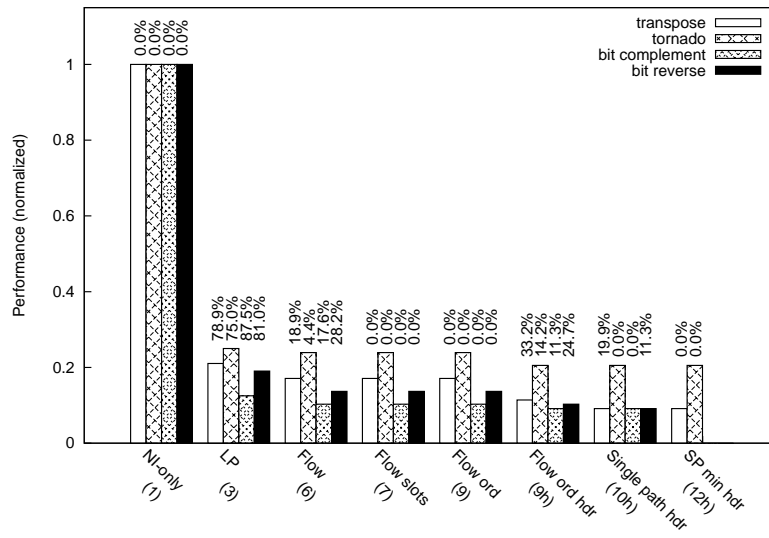
Experiment 2.15: 8x8 mesh network, permutation traffic, 64 IPs, 32 slots.



Experiment 2.16: 8x8 torus network, permutation traffic, 64 IPs, 32 slots.



Experiment 2.17: Quaternary fat tree (3 levels), permutation traffic, 64 IPs, 32 slots.

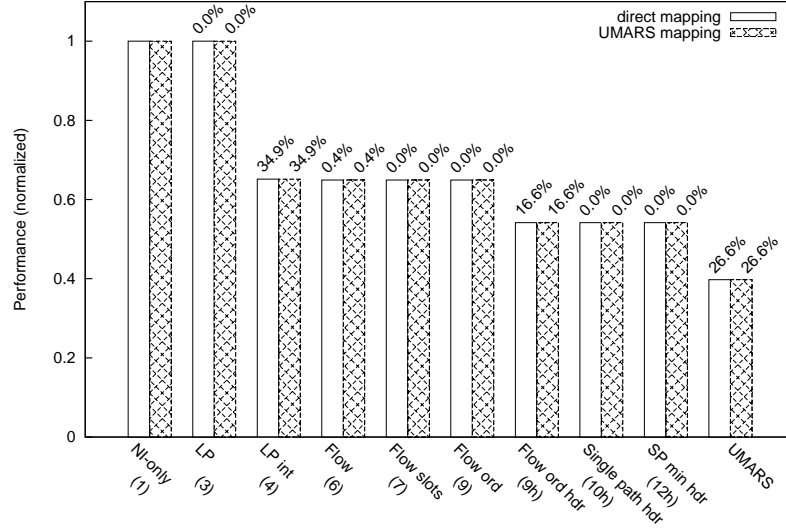


Experiment 2.18: 64-node Spidergon network, permutation traffic, 64 IPs, 32 slots.

### 2.4.5 Real applications

In the following we present the performance of the proposed network models under communication patterns extracted from real applications reported in the literature. The number of TDM slots has been set to 16 for all applications except [HG11] which used a large number of connections and required a larger number of slots. In all cases, a mesh network of the appropriate size was instantiated.

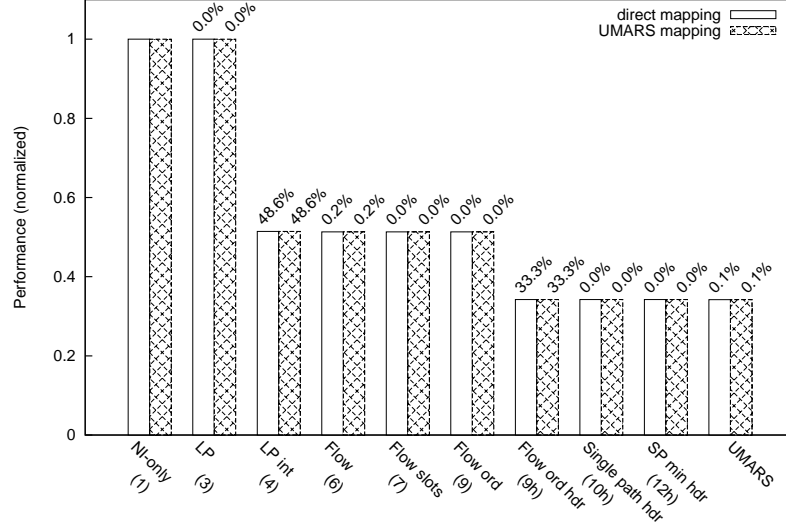
The task graphs for these applications are represented in Figure 2.18. For the MPEG-2 application [LCOM08], as the bandwidth of each connection was not specified, we assumed equal bandwidths.



Experiment 2.19: 4x4 mesh network, MultiMedia System [HM05] traffic, 16 IPs, 16 slots.

Experiment 2.19 presents the performance of the proposed models on the MultiMedia System described in [HM05] and Experiment 2.20 using the application in [HG11].

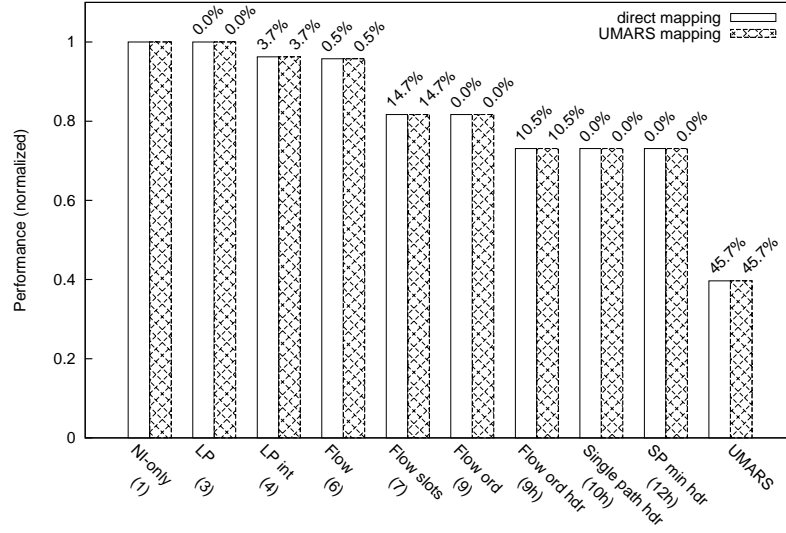
Experiment 2.21 presents the performance of the models for the MPEG-2 application described in [LCOM08]. For this investigation the necessary bandwidths were set to a constant value as they were not specified in the literature and the input and output were modeled as separate nodes.



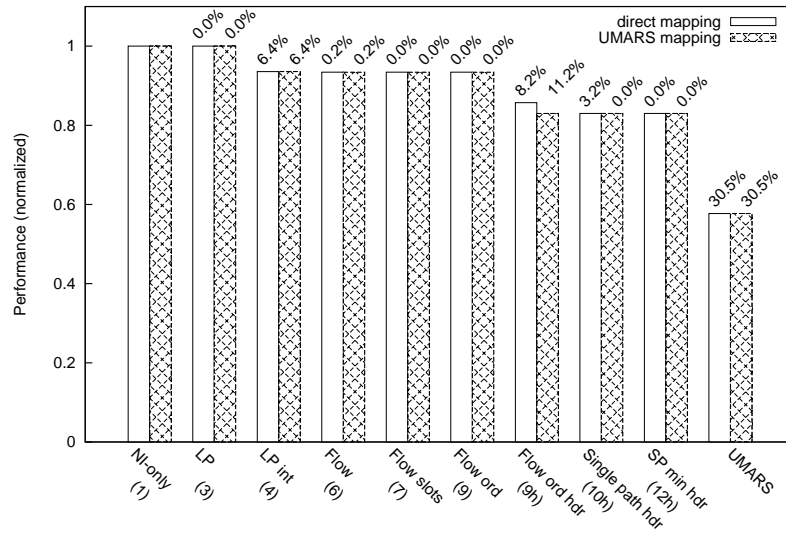
Experiment 2.20: 2x4 mesh network, digital TV [HG11] traffic, 8 IPs, 60 slots.

Experiments 2.22 and 2.23 present the performance of the models for the VOPD and MWD task graphs presented in [VdTJ02].

The applications presented very little problems, even for the simple mesh topology. The main sources of inefficiency were the discrete link division (Model 3 versus Model 4) and the header overhead (Model 9 versus Model 9h). We consider that the reason for this is that these applications involved a small number of IPs with simple communication patterns. We expect that with the emergence of many-core systems applications will exploit parallelism more than in the past and communication requirements will become more demanding.



Experiment 2.21: 3x3 mesh network, MPEG-2 [LCOM08] traffic, 9 IPs, 16 slots.



Experiment 2.22: 4x3 mesh network, VOPD [VdTJ02], 12 IPs, 16 slots.

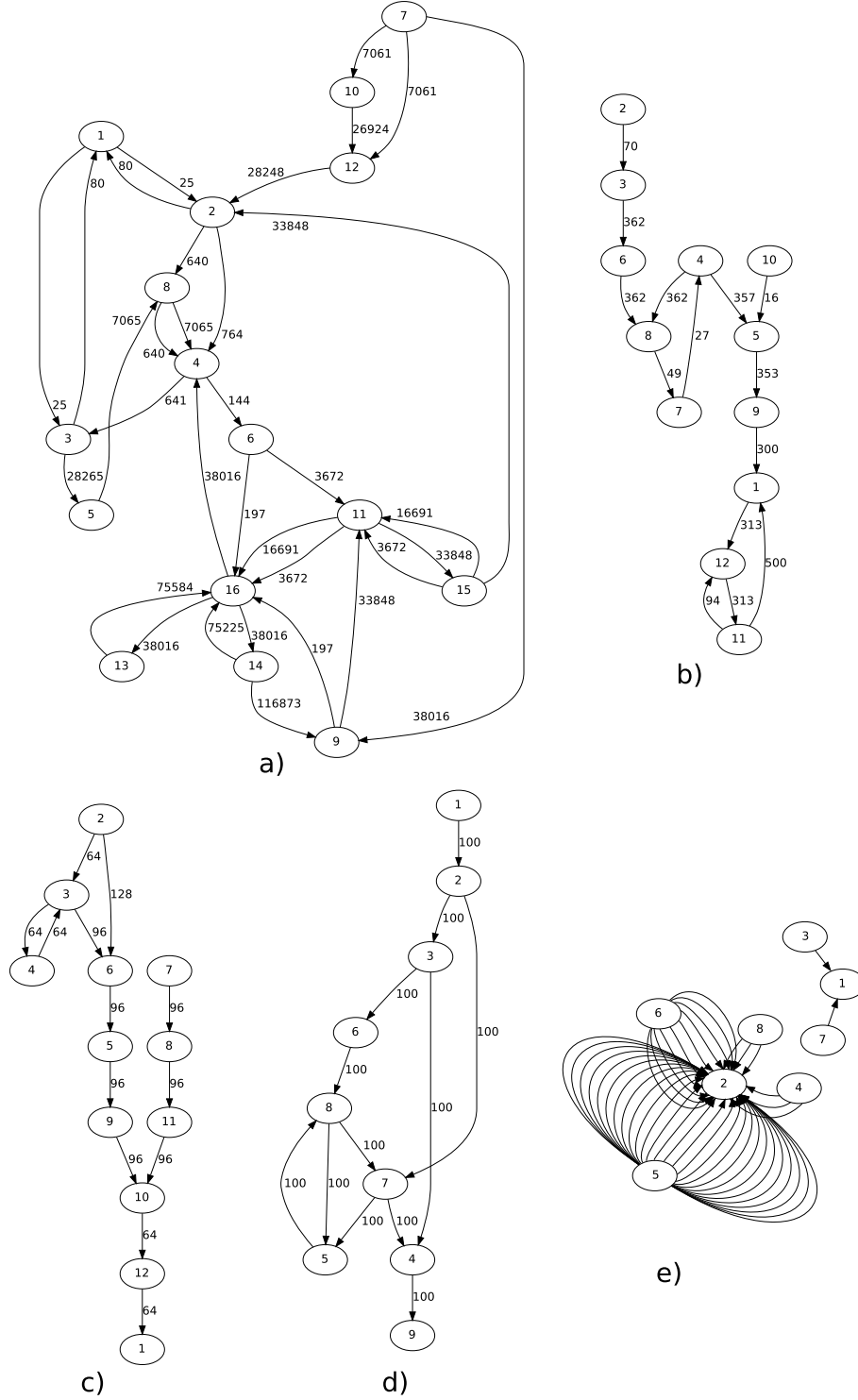
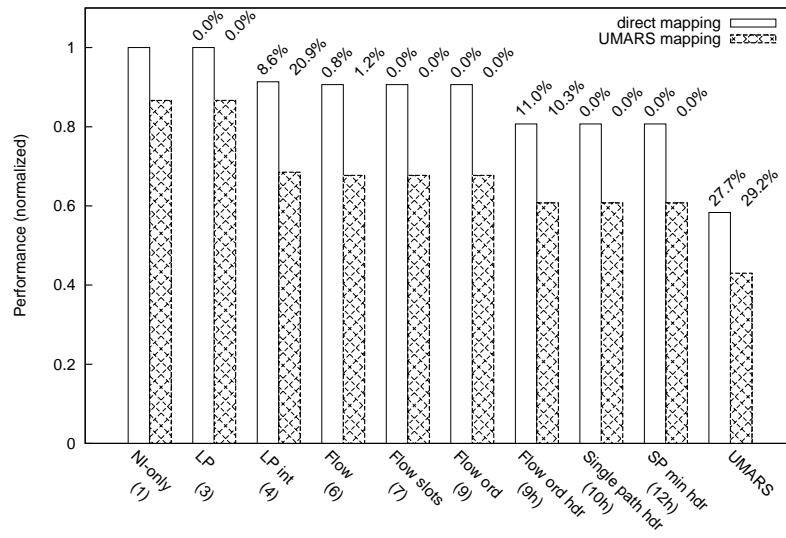


Figure 2.18: Traffic types reported in the literature: a) multimedia system [HM05], b) VOPD [VdTJ02], c) MWD [VdTJ02], d) MPEG-2 [LCOM08], e) digital TV [HG11].



Experiment 2.23: 4x3 mesh network, MWD [VdTJ02], 12 IPs, 16 slots.



### 2.4.6 Performance of different topology sizes under random traffic

We return to the random traffic model to perform a study of mapping the same communication pattern over topologies of different sizes. This time we use a system with 64 IPs and 100 connections. For the 8x8 mesh topology this corresponds to one IP for every NI, but for the smaller topologies several IPs will have to be mapped to the same NI. The *direct mapping* is used. When the number of NIs is lower than the number of IPs, more IPs are connected to the same NI by wrapping around the list of NIs. The number of slots was set to 32.

We also study topologies with up to 4 NIs connected to each router. A larger number of NIs means that *peripheral links* will be slower to saturate.

In this graph, instead of normalized performance, we present the frequency of the network that is needed to support a usecase. The result for each model and each topology is computed as an average over 20 usecases. To avoid encumbering the graphic we represent only some of the models. Models 8 and 10h are of particular interest, because Model 8 is supported by the dAElite network that we will propose in Chapter 6 and Model 10h is supported by the Æthereal network.

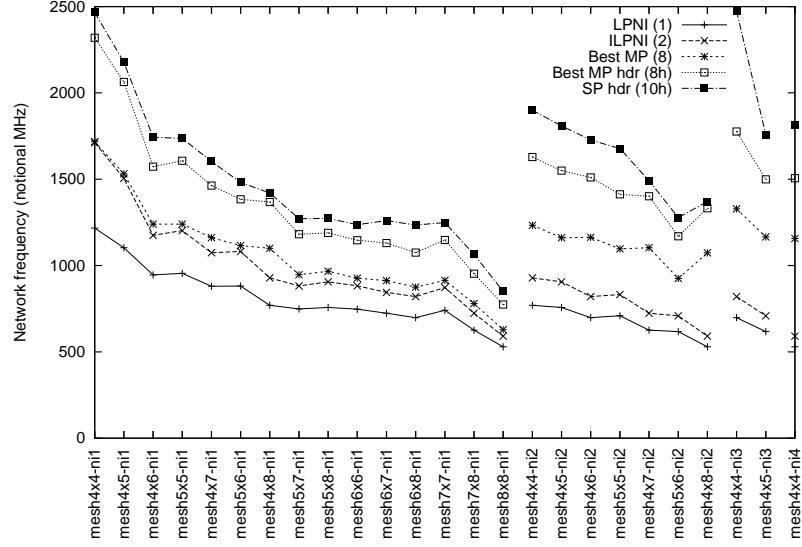
The results are presented in Experiment 2.24.

There are larger performance gaps between the ideal network and the physically realizable multi-path enabled network and between the multi-path and single path when larger topologies or topologies having more NIs per router are considered. In general, the gain of multi-path is higher when *core links* are more congested.

### 2.4.7 Summary of experiments

Two behaviors are seen in the presented experiments. As a convention we will call these scenarios *light* and *heavy load*. Whether the load is light or heavy is dependent on topology and the type of traffic.

- Under *light load*, topology is not a limiting factor and Model 1 and Model 3 exhibit the same performance. Under *heavy load*, not only do we see a drop in performance between Model 1 and Model 3, but each of the subsequent models behaves worse than the previous one does.



Experiment 2.24: Network frequency versus topology size for the same use-cases (random traffic) allocated on all topologies (lower is better).

There are topologies that cope well with any kind of traffic, or in other words, any type of traffic is *light load* for that topology. As a convention we call these *strong topologies*. We call topologies that deal badly with most kinds of traffic *weak topologies*. The *ring* topology is a weak topology, strong topologies are the *fat tree* and the *torus*, with the *mesh* and *spidergon* being somewhere in the middle.

- Small networks deal better than larger networks with most types of traffic when the traffic is scaled proportionally with the size of the network. When the amount of traffic is fixed a larger network provides a better solution.
- Bandwidth division granularity produces a sizable overhead, regardless of whether the load is *light* or *heavy*. The difference between Models 3 and 4 reaches 38% in some cases. An exception is the uniform traffic tests, when channels of equal bandwidth can match precisely the division granularity.
- Performing the allocation iteratively presents little to no performance drop under *light load*, compared to global optimization. This can be

seen in the comparison between Models 4 and 6.

- The contention-free link sharing model does not incur a large performance penalty compared to a generic link sharing model under *light load* and even under *heavy load* the drop in performance is hardly ever larger than 10%. This is an important find as the contention-free routing model is much less expensive to implement.
- In-order delivery restrictions (Model 7 versus Model 9) have hardly any effect, in some cases the model using in-order delivery even produces improvements due to the additional heuristics employed in that allocation algorithm.
- The multi-path allocation model provides improvements compared to the single-path approach, most visibly under *heavy traffic*.
- The usage of headers produces a large drop in performance under both *light* and *heavy* load, but especially when the number of connections is high.

The most restrictive model usually performs at between 40% and 60% of performance of the ideal model. We consider this performance to be quite good, considering all the restrictions that were introduced from the ideal model to the physical hardware implementation. Most of the performance loss is not due to the allocation algorithm.

## 2.5 Related Work

Ring, spidergon and mesh networks have been evaluated in [BC06] using hot-spot and random uniform traffic. In [BCG<sup>+</sup>07] an MPEG-4 task graph is used. The switching model used by the network was wormhole which is very popular among network on chip implementations because of the low cost of buffers, and the method of evaluation was simulation.

We evaluate our model using analytical methods instead of simulation. The contention-free switching model has a deterministic behavior which is easily analyzable.

A circuit-switched interconnect (crossroad interconnection architecture) is evaluated in [CSC08] using the task graphs for the VOPD and MWD applications presented in [VdTJ02]. The paper presents topology optimization techniques.

A framework for the modeling and simulation of networks on chip is presented in [CCG<sup>+</sup>04]. The proposal relies mainly on SystemC simulation.

Mesh, concentrated mesh and hypercube NoCs with 16-nodes are evaluated in [GMB<sup>+</sup>08] using RTL equivalent SystemC simulation. The authors use a manually-created high-contention traffic pattern.

Mesh and point-to-point interconnects are evaluated in [HG11]. Several interconnect options are discussed, with a focus on scalability. Buffering requirements are evaluated, the paper presents a comparison of high-level estimations of the resulting network cost to synthesis results.

Fat tree and irregular (custom generated) topologies are evaluated in [KD10] using MWD and MPEG 4 task graphs. The study also discusses traffic bursts and temporal access patterns which are not covered by our study.

A contention-free optical NoC with Spidergon topology is evaluated in [KH09]. The type of traffic used is random uniform and the evaluation method is simulation.

Energy and latency driven mapping onto a NoC are evaluated in [KMC<sup>+</sup>05] on a collection of mesh, torus and fat tree topologies. The study uses FFT, Romberg integration and Image processing applications as benchmarks for the network.

Point-to-Point, NoC and bus interconnects are evaluated in [LCOM08]. The study presents an analytic performance comparison, validated in FPGA. MPEG-2 is used as a benchmark.

Mesh, fat tree and reduced unidirectional fat tree topologies are benchmarked in [LGM<sup>+</sup>09] using uniform random traffic. The study addresses physical layout concerns for the fat tree.

An analytical model for the performance of the Spidergon network is provided in [MSVO07]. The paper presents results for network latency when using uniform random traffic.

In [MPCJ08] the authors show how NoC models can be used in the evaluation of MPSoC designs. As a proof of concept MJPEG and h264 are demonstrated in SoCs supported by 2d mesh and torus networks.

A model is introduced in [OMM<sup>+</sup>08] to evaluate NoC latency. The study proposes a technique to avoid simulating the movement of the body of packets, instead focusing on the movement of the header and trailer flits. The study uses mesh and torus networks with random uniform, normal and pareto on-off traffic.

Mesh, torus, fat tree and octagon wormhole networks are evaluated in [PGJ<sup>+</sup>05] using uniform, Poisson and self-similar random traffic.

Lossy on-chip interconnects with mesh, Spidergon and WK-recursive topologies are studied in [SBE08].

Mesh and BMIN (binary fat tree) networks were evaluated in [TM09] using simulation and analytic measures. Uniform traffic and local traffic were used.

Analytical performance models were proposed for the torus network in [SKFO08b, LO09], for mesh in [SKFO08a] and hypercube in [SKFO05, PS06].

The relation between mapping and routing is studied in [TOM<sup>+</sup>11]. While in our experiments we consider only two mappings, it would be interesting to extend this study to include different mapping strategies.

The multicommodity flow formulation is very suitable for routing problems and has been employed before both in the routing of wires in physical circuits [CILC96] and in network routing [TDB03].

Multicommodity flow is specifically applied to circuit-switching networks in [LR95].

The benefits of allowing non-minimal routing for load balancing are presented in [MKY<sup>+</sup>05].

We believe that in this chapter we offer a thorough investigation of a wide range of network models against several variables: topology, network size and traffic type.

## 2.6 Conclusions and Future Work

In this chapter we have proposed and evaluated several network models. These models allow us to determine the performance implications of various design choices, for example topology, routing and switching mechanism. The performance analysis in this chapter leads to some interesting conclusions which are useful in the context of NoC design.

We illustrate the performance difference between network models in Figure 2.19. The numbers here are based on an average over Experiments 2.1-2.23. The individual data points in each experiments were first averaged individually and each experiment contributed with equal weight in the final average.

One of the first conclusions is that topology in general is not a significant



to use computationally less expensive allocation algorithms. Such algorithms will be presented in the next chapter.

Finally, one of our goals was to determine if the contention-free routing model proposed by *Æthereal* incurs or not a large overhead (the *Æthereal* network supports models 10h and 12h from our evaluation and guarantees achieving the performance indicated by these models). We found the performance loss to be modest when considering the hardware cost advantage (a factor of 10-20 times is claimed in [GH10]) of contention-free routing. Note that of the 43% drop in performance, roughly 15% are due to header overhead and 18% due to bandwidth division granularity. Other network implementations are likely to suffer from these overheads as well.

In Chapter 6 we propose a network implementation called *dAElite* that supports a broader range of models (Models 8-12, which are all the models based on in-order-delivery contention-free routing). It also does not suffer from header overhead, thus losing on average less than 26% from the ideal NoC performance. Our proposal is also cheaper to implement than *Æthereal*.

Regarding future research, we already envision extensions to the range of network models. It would be interesting to see for example what is the effect of restrictions like single-path routing or even X-Y routing on the more general models, e.g. Model 3. It would also be interesting to compare the actual performance of classic store-and-forward, virtual-cut-through or wormhole networks to the performance bounds provided by our current models. Finally, the models presented here should be put to use in tools or toolflows for the automated design of networks on chip.





## Chapter 3

### Single and multi-path allocation algorithms

**I**n this chapter, we describe in detail the path-finding algorithms used in the experiments in Chapter 2 for Models 6-12. We also present slot-allocation algorithms that ensure in-order delivery in Models 8-9.

The problem of finding a path and slot allocation for a set of connections is similar to the problem of routing physical wires in integrated circuits. Compared to the wire routing problem, the number of connections our algorithm has to handle is several orders of magnitude lower, in the range of hundreds, but the problem formulation itself is more complex because connections do not have equal bandwidth requirements and this is reflected in the amount of resources that are allocated to each connection.

All the path-finding algorithms here are used in conjunction with the iterative method presented in Section 2.3. Instead of looking for a globally optimal solution (by global we mean a solution that optimizes all communication channel allocations simultaneously), channels are allocated one by one, earlier allocations blocking resources to the detriment of later allocations. This is similar to how physical wire routing tools operate, as well as the previous *Æthereal* tools [HGR07].

The algorithms can be split in two categories: algorithms that use slot masks and algorithms that use graph-splitting. In addition, we present the generic flow algorithm used in Model 6, which only produces path allocation and does not produce a slot allocation. This algorithm forms a base for the more complex algorithm that do perform slot allocation.

Algorithms based on the slot mask approach use a representation of the

network topology as a directed graph, each node representing one *router* or one *network interface* and each directed edge representing one *unidirectional network link*. Each directed edge has an associated table indicating which slots are available on the link it represents (Figure 3.1).

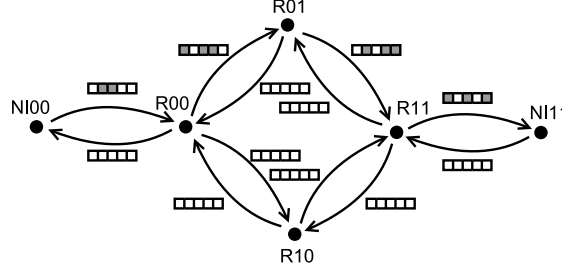


Figure 3.1: One-to-one graph representation of the network topology with slot masks attached to edges. Grayed slots are occupied.

The graph splitting algorithms use a different representation of the network topology. Each network node, i.e. *network interface* or *router*, is represented by  $s$  nodes in the graph, where  $s$  is the number of time slots (each network node is thus “split” into  $s$  different graph nodes). Each graph node represents the ability to reach its corresponding network node during a certain time slot (Figure 3.2). Each unidirectional network link is also represented by  $s$  directed graph edges. Each graph edge represents one slot on one network link.

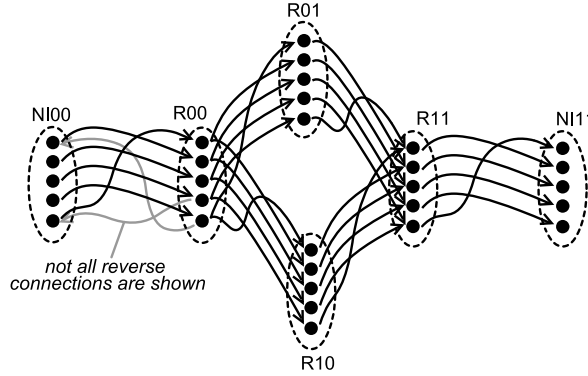


Figure 3.2: Network topology represented through a graph with split nodes.

Graph edges are connected to graph nodes in such a way as to represent the link traversal delay. Node  $A_0$  is connected to  $B_1$  and  $A_{n-1}$  back again to  $B_0$

due to wrap-around at the end of the slot table. Note that on the reverse link  $B_0$  is connected to  $A_1$  and  $B_{n-1}$  to  $A_0$  so the new graph is directed and the links do not form pairs between the same graph nodes.

The structure of the remainder of this chapter is presented in Figure 3.3.

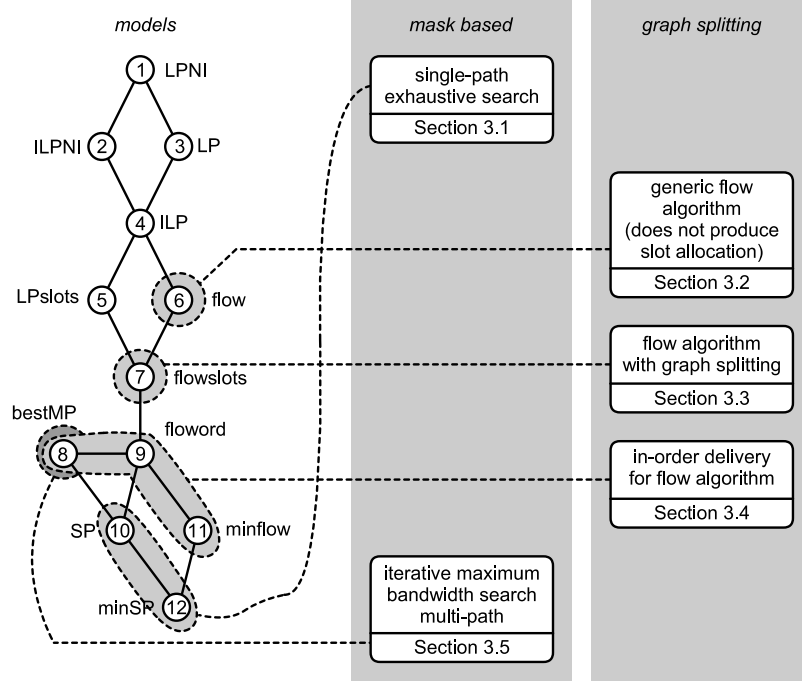


Figure 3.3: Algorithms discussed in Chapter 3 and the models they correspond to.

The allocation methods for the Models 1-5 were already discussed in Chapter 2. Because they are based on Linear Programming, they consist of simply applying an off-the-shelf LP optimization tool to the formal problem definition. In Section 3.1 we provide a description of the single-path exhaustive search algorithm which is the simplest and at the same time the one closest to the algorithm used by state-of-the-art allocator (UMARS) [HGR07]. The generic flow algorithm, used to perform allocation in Model 6 is presented in Section 3.2. The adaptation of the flow algorithm to include time slots is presented in Section 3.3. The optimal algorithm to select paths which result in in-order delivery is presented in Section 3.4. A method for iteratively computing multi-path allocations is presented in Section 3.5. Related works are presented in

Section 3.6. Section 3.7 presents our conclusions.

### 3.1 Exhaustive search Single Path

Path finding under the contention-free routing model is complicated by the fact that it needs to take into account not only the bandwidth available on each link but also the slot alignment. Consider the example in Figure 3.4.

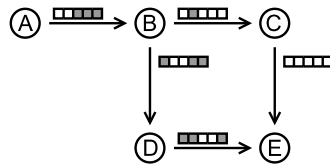


Figure 3.4: Example graph for the path finding problem, used slots are grayed out.

Assume that we want to find a path from A to E offering the bandwidth equivalent of 2 TDM slots. There are two possible routes: A-B-C-E and A-B-D-E. At first look, the A-B-C-E appears to be a better option because the links BC and DE have a lower slot occupation ratio. If we look at the slot alignment though, we observe this is not the case since link BC blocks one of the only two available follow-up slots after the traversal of link AB, and as a result the route A-B-C-E does not meet the minimum bandwidth requirement. The A-B-D-E route on the other hand does provide the needed bandwidth because, when the delay of one slot per hop is considered, the two available slots on the each of the links AB, BD, DE align perfectly.

Simple path finding algorithms like Dijkstra's algorithm [Dij59] operate by successively building partial solutions starting from nodes neighboring the initial node and working their way to more distant nodes. They use the computed path to the near nodes to find the path to more distant nodes. When multiple paths may be used to reach a certain network node, classical path-finding algorithms store only one path leading to that node, the one having minimum cost. Based on this one stored path the algorithms try to reach nodes which are further away. As it can be observed in Figure 3.5 this approach does not work when slot alignment needs to be taken into account.

Assume again a path needs to be found between node A and node E with a bandwidth corresponding to 2 TDM slots. The first portion of the path, from

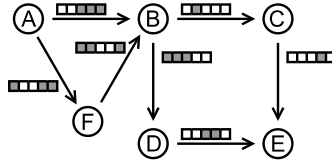


Figure 3.5: Classic path finding algorithms fail to find a solution.

node A to node B can be traversed in two ways: using the direct link A-B or through the longer route A-F-B. Each of these paths allows a different slot alignment. A classic path finding algorithm will prefer of course the shorter path and will not concern itself with the alternatives. But a path starting with slots 1-2 on link A-B does not allow any follow-up alternatives, either through B-C-E or B-D-E. The path A-F-B-D-E however provides a correct solution.

The fault of the algorithm consists in the fact that when node B is reached it is impossible to tell which way of getting there is better, as this depends on the slot alignment on future links (i.e. links that were not explored yet). The way to work around this problem is to store all possible paths to intermediate nodes.

The *Æthereal* tools employ a branch-and-bound solution [HGR07], choosing to construct and store all intermediate paths in the order of length, starting from the source node. The problem with this approach is that the number of paths explodes exponentially with the path length and memory is needed to store all paths (Figure 3.6). The problem is to some extent avoided by eliminating early the paths that do not provide sufficient bandwidth.

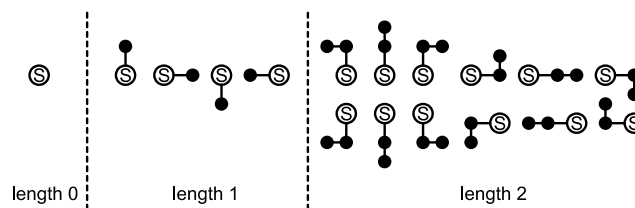


Figure 3.6: The number of possible paths grows exponentially with length.

### 3.1.1 Proposed exhaustive path search

We propose instead an exhaustive search based on the backtracking method that while still having exponential running time requires an amount of memory which only grows linearly with the number of nodes and links.

Our solution has two steps: a fast, breadth-first search to determine the distance from all nodes to destination, and then the actual exhaustive search.

The breadth-first search is a particular case of the Dijkstra algorithm where all links have equal cost. The search starts at the destination node because this provides estimate of the distance from each of the other nodes to the destination node. The estimate is optimistic because it takes into account only available bandwidth on each link and not the slot alignment.

This step allows us to quickly determine during the exhaustive search which edges are leading towards the destination and which are not. It also provides a bound on the distance from source to destination.

The algorithm, which is straightforward, is described in Algorithm 3.1.1. All distances, stored in the vector *dist*, are expressed in terms of the number of hops to destination.

We will use the following notation:

- $V$  the set of network nodes
- $E$  the set of network links (graph edges)  $e = (u, v), u, v \in V$
- $S$  the set of slots  $s_1, s_2 \dots s_n$
- $S_{u,v}$  the set of available slots on link  $(u, v) \in E$

The exhaustive search of a single path is performed by a recursive algorithm which constructs one-by-one all possible paths leading to the destination. In order to explore first shorter paths, the algorithm limits the *detour*, i.e., the difference between the length of the path and the minimum length produced by the breadth-first search. The detour is computed during the search by adding the level (the number of edges already part of the current path) to the known (bound on) distance to destination and comparing it to *maxLength* the sum of the minimum distance and the allowed *detour*. The value of *detour* is varied between 0 and 16. It is prohibitive to increase this value even further because the number of generated paths increases exponentially with their allowed length.

---

**Algorithm 3.1.1:** Breadth-first search

---

**input** :  $V, E, S_{q,w}$  according to the previously defined notation  
            $Destination \in V$  the destination node  
            $bw_r$  required bandwidth  
**output**:  $dist[.]$  a list of distances from each node to  $Destination$

```
// Q IS A QUEUE
enqueue  $Destination$  on  $Q$ ;
 $visited[V] \leftarrow false$ ;
 $visited[Destination] \leftarrow true$ ;
 $dist[V] \leftarrow \infty$ ;
 $dist[Destination] \leftarrow 0$ ;
while  $Q \neq \emptyset$  do
   $q \leftarrow headOf(Q)$ ;
  dequeue  $q$  from  $Q$ ;
  for  $w \in V$  with  $(w, q) \in E$  do
    if  $bandwidth(S_{w,q}) < bw_r \wedge \neg visited[w]$  then
       $visited[w] \leftarrow true$ ;
       $dist[w] \leftarrow dist[q] + 1$ ;
      enqueue  $w$  on  $Q$ ;
    end
  end
end
```

---

We provide a formal description of the recursive algorithm in Algorithm 3.1.2. The *rotate* function shifts the contents of a slot table by one position so as to take into account the delay of one slot per link.

For explaining the functioning of the algorithm, consider the following example illustrated in Figure 3.7. The network illustrated here has a mesh topology with a size of 4x4. Several communication channels have already been allocated and therefore some of the slots are already used. For clarity we only represent the slot tables of interest.

Consider the problem of finding a path from the local NI of Router 20 to the local NI of Router 03. Furthermore consider that links R01-R02 and R21-R22 are already loaded to the extent that the bandwidth of the current communication channel cannot be satisfied regardless of the slot alignment. This fact will be taken into account by the breadth-first search and will result in assigning a larger distance value to Router R01 for example. The distances to the destination NI, found by the breadth-first search are shown on the bottom-left side of each router in Figure 3.7.





**Algorithm 3.1.2:** Recursive backtracking exhaustive search

---

**input** :  $V, E, S_{q,w}$  according to the previously defined notation  
            $Destination, Source \in V$  the source and destination nodes  
            $dist[..]$  a list of distances from each node to  $Destination$   
            $bw_r$  required bandwidth  
            $maxLength$  the maximum length of a path to destination  
**output**: A path  $P = (p_1, p_2 \dots p_n)$  with  $n \leq maxLength, p_i \in E$  which satisfies the  
           and a set of slots  $S_{path}$  which meet the bandwidth requirements and do not  
           conflict with any of the reserved slots on the path  $P$  (when properly rotated to  
           take into account the propagation delays).

---

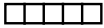
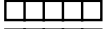









```

used[E]  $\leftarrow$  false;
recursive(Source, S, 0,  $\emptyset$ );
function recursive(node, crtS, level, path) begin
    if node = Destination then
        // SOLUTION FOUND, UPDATE IF CURRENT SOLUTION
        // IS BETTER THAN THE ONE STORED
        P  $\leftarrow$  path;
        Spath  $\leftarrow$  crtS;
        return true;
    end
    found  $\leftarrow$  false;
    for w  $\in$  V with (node, w)  $\in$  E do
        1   nxtS  $\leftarrow$  rotate(crtS)  $\cap$  S(node),w;
        2   if bandwidth(nxtS) < bwr then
            // WE MEASURE THE BANDWIDTH OF nxtS WHICH ALREADY
            // TAKES INTO ACCOUNT SLOT ALIGNMENT
            continue;
        end
        3   if level + dist[w] + 1 > maxLength then
            // WE LIMIT THE DETOUR
            continue;
        end
        4   if used[(node, w)] then
            // WE DO NOT ALLOW REVISITING EDGES
            continue;
        end
        used[(node, w)]  $\leftarrow$  true;
        found  $\leftarrow$  found  $\vee$  recursive(w, nxtS, level + 1, append(path, (node, w)));
        used[(node, w)]  $\leftarrow$  false;
    end
    return found;
end

```

---

The backtracking search starts at the source NI with a *maxLength* limit of 7, corresponding to 0 *detour*. We represent values at the top of the stack corresponding to *node*, *crtS(slots)*, *level*, *dist[node]* in Figure 3.8. The behavior is similar to that of a depth-first search, except nodes are revisited for each possible path.

<i>node</i>	<i>slots</i>	<i>distance</i>	<i>level (stack depth)</i>
source		7	0
R20		6	1
R21		5	2
R11		4	3
R12		3	4
R13		2	5
R03			
R12		3	4
R02		2	5
R03		1	6
dest		0	7

time  
↓

Figure 3.8: Sequence of values found at the top of the stack during algorithm execution.

The search follows the links that lead to nodes with a lower distance to the destination, i.e., *dist[node]* value. If the initial search with *maxLength* = *dist[Source]* does not produce results, the search will be repeated using an increased value of *maxLength* which will allow visiting nodes with higher *dist[...]* values.

During the search the *crtS* variable stores the list of slots available (with proper alignment) on all links from the source to the current node. The set of slots available at the next node to be visited *nxtS* is built equal to the intersection of the *crtS* set, rotated by one position as to take into account the delay of one slot for the next hop, with the set of slots available on the link that is being traversed.

It can happen that the new slot set *nxtS* does not meet the necessary bandwidth requirement (for example the 7th row in Figure 3.8 the delivered bandwidth is 0 as no slots are available). In the given example, when the recursive function is called with *node*=R13, the R13-R03 link is discarded by the *if* instruction in line 2, while the other links leaving from R13 will be discarded by the *if* instruction in line 3 because they lead to more distant nodes.

This causes the *for* statement in line 1 to finish without a recursive call and thus the algorithm will return (or backtrack) to the previous level in the program

stack where the *for* loop continues iterating through the neighbors of R12.

In our example the next link from R12 already leads to a solution, but the algorithm does not stop when the first solution is found and instead looks for other paths that may have better header overhead. As the set *crtS* may contain more slots than there are actually needed, when the destination is reached another algorithm is used to select the slots which are to be allocated to the given channel. The algorithms used for this purpose are presented in Chapter 4.

The search is abandoned if  $10^7$  paths have been analyzed for a single channel allocation without finding a solution. This is however a very rare occurrence in our experiments (occurred twice in a set of tens of thousands of experiments).

When the length of the allowed path is above the minimal value, it is possible for the path finding algorithm to return to a previously visited node. For example, in Figure 3.7 if *maxLength* is set to 9 instead of 7, when reaching node R13 it would be allowed to select node R12 as follow-up. This may be supported by the hardware, in fact both *Æthereal* and the *dAElite* network we propose in Chapter 6 support this feature. It should however not be allowed to reuse the R12-R13 once Router R12 is reached for a second time. This is not a hardware limitation but a limitation of the path-finding algorithm which cannot take into account the fact that slots on link R12-R13 may need to be used twice. To avoid this situation we have introduced a third check in line 4. The array *used*[] keeps track of which links are part of the current solution (the links which are found on the stack).

The algorithm is flexible and supports further adaptations, for example if desired we could deny returning to an already visited node like in the previous example, we could deny turns or introduce additional cost functions or path selection criteria.

The algorithm has a time complexity which is exponential in the distance between source and destination. The exact function depends on the arity of the network routers. For a mesh network, considering  $l$  being the distance between nodes and  $d$  the *detour* a rough estimate of the complexity would be  $O(2^{l+d} \binom{l+d-1}{\lfloor d/2 \rfloor})$ . This can be interpreted as: misroutes can take place in any of the  $l + d$  steps except for the last, hence the factor  $\binom{l+d-1}{\lfloor d/2 \rfloor}$  and at each hop, once we decided whether it is going to be a misroute or not, we have on average 2 choices.

### 3.2 Bandwidth allocation using Network Flow

The concept of flow is used to model the movement of goods through a transportation network. It can be intuitively visualized like the flow of water through series of pipes joined at nodes, or electricity through an electrical distribution network. Pipes or electrical wires have a certain capacity which can be seen for example as the physical diameter or section of the pipe or the maximum electrical current a wire supports without overheating. The same model applies well to communication networks as we will present shortly.

The network is modeled as a directed graph  $G = (V, E)$  where  $V$  is the set of vertices and  $E$  the set of links and graph-based algorithms exist that can provide an answer to whether it is possible to move a certain quantity of “goods” between given points in the network or optimize the cost of such movement from the point of view of distance traveled.

Each edge in the graph  $e \in E$  has a given capacity which we call  $c_e$ . The actual amount of goods transported through that edge we call  $f_e$  and  $f_e \leq c_e$ ,  $\forall e \in E$ . Goods are moved from a *Source* to a *Destination* node in the graph and are not lost along the way, therefore a conservation law must apply:

$$\sum_{(i,q) \in E} f_{(i,q)} = \sum_{(q,j) \in E} f_{(q,j)}, \quad \forall q \in E \setminus \{Source, Destination\}$$

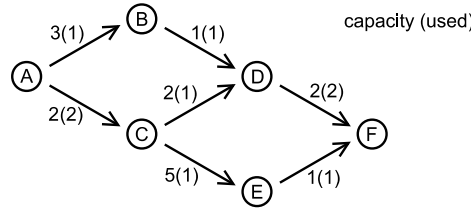


Figure 3.9: Example flow.

The flow algorithm however does not impose restrictions on how the movement of goods is distributed among the available graph edges (Figure 3.9), it may split and then recombine in arbitrary manner at intermediate nodes.

Flow networks are thus suitable for modeling network communication as long as routing over multiple paths is allowed. Graph nodes in this case would represent routers, edges network links, the goods may represent data and the

edge capacity the link bandwidth. The result of the flow algorithm will be the path or paths that data needs to be routed over in order to reach the destination.

The allocation is performed on a channel-by-channel basis, thus the iterative methods presented in Chapter 2. The initial capacities are equal to the link capacity, but as connections are allocated, their flow values  $f_e$  are subtracted from the capacities available to the following connections  $c_e$ .

The allocation of one connection starts with an “empty” flow,  $f_e = 0, \forall e \in E$  (Figure 3.10a). Remember that the previously allocated connections are already taken into account in the modified  $c$  values. The flow algorithm that we used is based on finding so called “augmenting paths” (Figure 3.10b), paths between source and destination whose links are not yet saturated  $f_e < c_e$ .

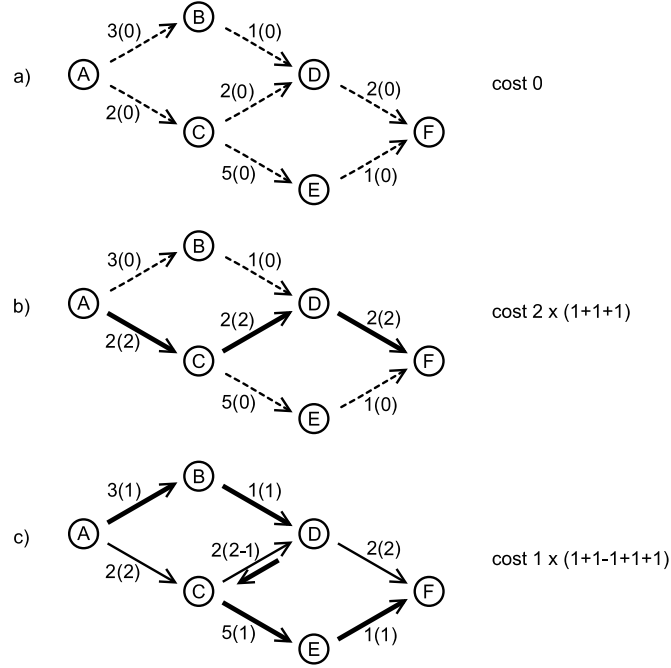


Figure 3.10: Computing flow using augmenting paths.

Because we wish to minimize the cost of communication (in terms of path length) we make use of a minimal-cost path-finding algorithm to find the augmenting paths. This approach guarantees that the overall cost of the flow ( $\sum_{e \in E} f_e * \text{edgeCost}[e]$ ) is also minimal [EK72]. Although it would be possible to assign different costs to edges in our implementation we assume

all edges have a cost equal to 1.

One advantage of the flow algorithm compared to simply applying a path-finding algorithm several times is that it is able to displace a previous unfavorable allocation to increase the flow, as shown in Figure 3.10c. In this example the first allocation (Figure 3.10b) blocked all direct ways of reaching F from A. The augmenting path finding algorithm is allowed to traverse an already used link (the C-D link) in the reverse direction, pushing back the flow on that link.

This operation introduces an additional complication as the cost of removing an unit of flow is negative and the path finding algorithm has to be able to cope with negative edge costs. We use an optimization by Edmonds and Karp [EK72] which avoids the problem of negative edges without modifying the functionality of the algorithm. This optimization consists of modifying the way the distance to a node is calculated by the path finding algorithm.

A formal description of the algorithm is given in Algorithms 3.2.1 and 3.2.2. Using the approach of Edmonds and Karp, the distance from the source node to any other node is split in two components: a *known* distance and an *incremental* distance. The *known* distance is stored in the (the  $ek[V]$  array) and is known because it was determined in a previous execution of the path-finding algorithm. The *incremental* distance is the increase in the distance to one node from one execution of the path-finding algorithm to another due to the fact that some of the network resources have already been utilized.

The path-finding algorithm only computes the *incremental* distance which is accumulated into the  $ek[V]$  array after each augmenting path is found. The *known* component of the distance is subtracted from the edge cost that is visible to the path-finding algorithm. Figure 3.11 presents the contents of the  $ek[V]$  array after the first allocation in the previous example (Figure 3.10). The path-finding algorithm determines the incremental cost of an edge according to Equation 3.1.

$$iCost_{src,dest} = \begin{cases} ek[src] - ek[dest] + 1 & \text{for forward edges} \\ ek[dest] - ek[src] - 1 & \text{for push-back edges} \end{cases} \quad (3.1)$$

From the point of view of the path-finding algorithm, the incremental cost (the only cost seen by the algorithm) of *forward* edges A-B, B-D, C-E, E-F is 0, and so is the cost of *pushing back* flow through edge C-D. The incremental cost of reaching the destination is also 0, which means that the actual cost of the new path found will be the same as that of the previous path.

---

**Algorithm 3.2.1:** Flow algorithm

---

**input** :  $V, E, c[E]$  according to the defined notation  
            $Destination, Source \in V$  the source and destination nodes  
            $bw_r$  required flow (bandwidth)  
**output**:  $f[E]$  capacity used on each edge  
 $ek[V] \leftarrow 0$ ;  
 $f[E] \leftarrow 0$ ;  
**while**  $bw_r > 0$  **do**  
      $dist[V] \leftarrow \infty$ ;  
      $maxf[V] \leftarrow 0$ ;  
      $dist[Source] \leftarrow 0$ ;  
      $Q \leftarrow \{Source\}$ ;  
     ▷ **find augmenting path** (Algorithm 3.2.2);  
     **if**  $maxf[Destination] = 0$  **then**  
         | **fail**;  
     **end**  
      $j \leftarrow Destination$ ;  
      $f_{increment} \leftarrow \min(maxf[Destination], bw_r)$ ;  
     **while**  $j \neq Source$  **do**  
         **if**  $pushback[j]$  **then**  
             |  $f[(j, predecessor[j])] \leftarrow f[(j, predecessor[j])] - f_{increment}$ ;  
         **else**  
             |  $f[(predecessor[j], j)] \leftarrow f[(predecessor[j], j)] + f_{increment}$ ;  
         **end**  
          $j \leftarrow predecessor[j]$ ;  
     **end**  
     **for**  $j \in E$  **do**  
         |  $ek[j] \leftarrow ek[j] + cost[j]$ ;  
     **end**  
      $bw_r \leftarrow bw_r - f_{increment}$ ;  
**end**

---

---

**Algorithm 3.2.2:** Flow algorithm - Main loop of augmenting path find

---

**input** : all variables of Algorithm 3.2.1**output:** updates  $cost[.]$ ,  $maxf[.]$ ,  $predecessor[.]$ ,  $pushback[.]$ 

```

while  $Q \neq \emptyset$  do
   $q \leftarrow \operatorname{argmin}_{q \in Q} dist[q];$ 
   $Q \leftarrow Q \setminus \{q\};$ 
  for  $j \in V$  with  $(q, j) \in E$  do
     $cap \leftarrow \min(maxf[q], c[(q, j)] - f[(q, j)]);$ 
    if  $cap = 0 \vee dist[j] \leq dist[q]$  then
      | continue;
    end
     $dd \leftarrow dist[q] + 1 - ek[j] + ek[q];$ 
    if  $dd > dist[j]$  then
      | continue;
    end
     $dist[j] \leftarrow dd;$ 
     $maxf[j] \leftarrow cap;$ 
     $predecessor[j] \leftarrow q;$ 
     $pushback[j] \leftarrow false;$ 
     $Q \leftarrow Q \cup \{j\};$ 
  end
  for  $j \in V$  with  $(j, q) \in E$  do
     $cap \leftarrow \min(maxf[q], f[(j, q)]);$ 
    if  $cap = 0 \vee dist[j] \leq dist[q]$  then
      | continue;
    end
     $dd \leftarrow dist[q] - 1 - ek[j] + ek[q];$ 
    if  $dd > dist[j]$  then
      | continue;
    end
     $dist[j] \leftarrow dd;$ 
     $maxf[j] \leftarrow cap;$ 
     $predecessor[j] \leftarrow q;$ 
     $pushback[j] \leftarrow true;$ 
     $Q \leftarrow Q \cup \{j\};$ 
  end
end

```

---



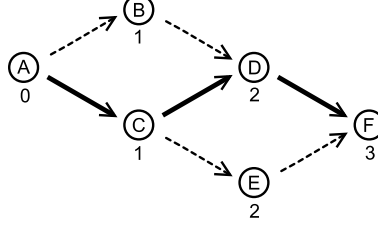


Figure 3.11:  $ek[V]$  costs associated to the nodes of the graph.

After a path is found by the path-finding algorithm, for example path A-B-D-C-E-F, the flow value is increased on the *forward* edges and decreased by the same amount on the *push-back* edges. This amount is limited by the leftover capacity of any *forward* edge along the way or the existing flow in a *push-back* edge.

When both the capacities and the requirements are integer values it follows that the result of the flow algorithm only uses integer values, thus unlike the LP formulation presented in the Chapter 2 can efficiently solve integer problems. The algorithm has polynomial running time [EK72]. Because edges have unit cost, an augmenting path can be computed with time complexity  $O(|E|)$ . The maximum number of augmenting paths is bound by  $O(|E| \cdot |V|)$  [CSRL01], but in practice, in our problem it may be bound by the granularity of network link division. The maximum number of augmenting paths is thus  $s$ , the number of TDM slots or SDM lanes, and the complexity of the algorithm is  $O(|E| \cdot s)$ .

### 3.3 Flow algorithm for the contention-free slot model

The flow algorithm can be applied to a graph in which each node represents a network router and each graph edge a network link, but it can also be applied to the “split” graph introduced in Chapter 2. In the former case, it will produce a routing function (a way of distributing connections’ load on the physical links) suitable for Model 6 of Chapter 2. In the second case it produces a contention-free slot allocation with proper alignment for each communication channel.

The produced solution nevertheless has some disadvantages. The flow algorithm does not take into account how many different physical paths are used to allocate the communication channel. We mitigate the effects of this problem by the introduction of one heuristic: for each augmenting path found by the flow algorithm, we attempt to allocate more slots, preferably consecutive to the

one just allocated, on the same physical path. This is also expected to reduce the header overhead, because consecutive slots over the same path may skip the routing header. While this heuristic may reduce the number of different paths generated for one communication channel by the flow algorithm it does not completely eliminate the problem. The result of the flow may be using multiple paths. The dAElite network that we present in Chapter 6, supports multi-path routing.

We can further reduce the total number of paths used for routing by falling back to the single-path allocation algorithm if the multi-path solution does not have an advantage in terms of either path length or number of utilized slots. Models 8 and 9 in Chapter 2 fall back to the single-path solution, while Model 7 uses a pure-flow approach.

Routing over multiple paths may result in another unwanted effect, namely out-of-order delivery. Out-of-order delivery can arise when some packets take longer paths than others (Figure 3.12a). Solutions to this problem consist of the reordering of packets at some point of convergence within the network or at destination. This approach however can result in reassembly deadlock [MS80] (Figure 3.12b) or in the case of reordering at the point of convergence may introduce circular dependencies also causing deadlock (Figure 3.12c).

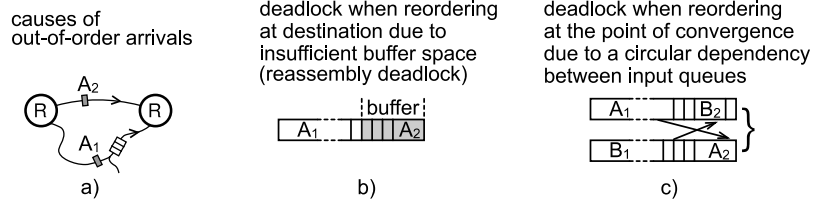


Figure 3.12: Different delays causing out-of-order delivery (a) and deadlock situations (b, c).

Using the contention-free routing model it is possible to perform multi-path routing while completely avoiding out-of-order delivery and without the threat of deadlock. The network traversal time is proportional to the number of hops on the path taken by packets and is not influenced by other factors. This allows us to determine at design time for a given set of paths and their starting slots whether they arrive in order or not. A verification step is performed in-between calls to the augmenting path finding function and paths that would produce out-of-order deliveries are discarded from the solution. Their reservation is still maintained so that the flow algorithm does not attempt to allocate the same

paths again. Although this is suboptimal it is necessary.

The flow algorithm applied to the split graph has a complexity of  $O(|E||S|^2)$  where  $E$  is the set of edges of the original (non-split) graph and  $S$  the set of slots. A single search for an augmenting path is performed with complexity  $O(|E||S|)$  because each edge has been split into  $|S|$  slots, and at most  $|S|$  searches are performed because each path will use one slot on the (non-split) link from the source node and that link only has  $|S|$  slots.

The algorithm selecting the paths that should be discarded is presented in the following section. The algorithm is optimal in that it minimizes the bandwidth of the discarded paths (and thus it maximizes the combined bandwidth of the remaining paths).

### 3.4 Path selection for in-order delivery

When a conflict exists between paths generated by the flow algorithm in the sense that they would produce out-of-order arrivals, we choose to discard as many of the paths as necessary, until the remaining ones produce only in-order deliveries. We use a deterministic algorithm based on dynamic programming to select which paths need to be discarded.

The problem can be formulated as a Monotonic Subsequence Problem [Sch61], for which optimal solutions exist with polynomial time complexity. The paths are ordered by slot departure time and the solution must comprise a subsequence with only increasing arrival times.

Further complications arise from particularities of our problem. Because consecutive slots have a different payload efficiency (consecutive slots do not need to repeat the packet header), the items in the sequence need to be weighted, and, the algorithm needs to take into account the wrap-around that occurs at the end of the slot table.

The associated weight for each path does not introduce significant changes to the algorithm, but in order to cope with the wrap-around, the algorithm will have to be applied repeatedly in a window which slides over the list of paths. A similar problem taking into account wrap-around problem is described in [AAN<sup>+</sup>07].

A formal description of the algorithm is given in Algorithm 3.4.1. The algorithm is optimal in the sense that it provides the highest possible bandwidth for the given set of paths.

**Algorithm 3.4.1:** In-order path selection

**input** : set of paths with given departure time slot, count of allocated slots and length  
**output**: reduced set of paths with in-order delivery

---

```

s ← number of time slots;
Duplicate paths  $p_1 \dots p_n$  as  $p_{n+1} \dots p_{2n}$  with delay s;
solution ←  $\emptyset$ ;
for  $\forall i \in \{1, 2, \dots, n\}$  do
    consider set  $Q = \{p_i \dots p_{i+n-1}\}$ ;
     $Q \leftarrow Q \setminus \{p_j \in Q \mid p_j \text{ arrives later than } p_{i+n-1}\}$ ;
    Q is the working window;
    initialize  $t_1 \dots t_{2n}, t_\emptyset = 0$ ;
    for all flows  $p_j \in Q$  do
        best ←  $\emptyset$ ;
        for all flows  $p_q \in Q, q < j$  do
            if  $p_q$  arrives before  $p_j$  and  $t_q > t_{best}$  then
                best ← i;
                predecessorj ← q;
            end
        end
         $t_j \leftarrow t_{best} + \text{bandwidth of } p_j$ ;
        if  $t_j$  is best solution so far then
            solution ← solution reconstructed by following the chain of
            predecessors of j;
        end
    end
end

```

---

The complexity of the path selection algorithm is  $O(n^3)$  where  $n$  is the number of paths to the destination and is given by the three nested loops in the algorithm. In practice the value of  $n$  is very small  $n < |S|$  and this algorithm does not contribute significantly to the running time of the allocation.

### 3.4.1 Proof of optimality for in-order path selection

In the following, by path we will refer to a path from source to destination and an associated set of contiguous available slots on that path.

Let  $A$  be the set of all paths  $A = \{p_i \mid p_i \text{ is a path}\}$ . In the following we will assume that  $A$  has at least one element.

Let us assume, without any loss of generality, that within a frame of the size

of the slot table  $m$ ,  $p_i$  departs no later than  $p_{i+1}$ ,  $\forall i \in \{1, 2, \dots, n-1\}$ . We say without loss of generality because the indices  $i$  of  $p_i$  can be chosen in such a way that the departure times of paths  $p_1 \dots p_n$  are chronologically ordered. Let  $b_i > 0$  be the bandwidth delivered by path  $p_i$  and  $r_i$  the arrival time of path  $p_i$  when considering a particular slot table revolution starting at  $p_1$  (Figure 3.13). The bandwidth takes into account the header overhead.

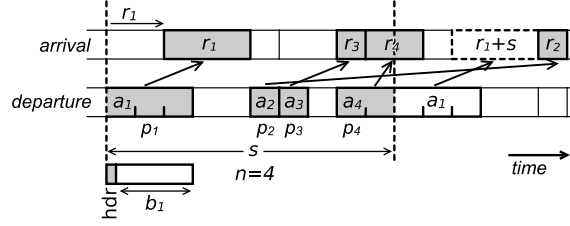


Figure 3.13: Slot table wrap-around.

**Definition 1.** A solution to the problem is a non-empty set  $X \subseteq A$  which ensures in-order delivery.

We formalize the requirement for in-order delivery as:

$$\forall p_i, p_j \in X \text{ with } i < j \rightarrow r_i < r_j, \quad r_j < r_i + s$$

where  $s$  is the duration of one slot table revolution. Note that all sets containing a single path, that is, all sets of the form  $X = \{p_q\}$  are solutions because a single path cannot produce out-of-order deliveries. Using the previous formalization we observe that the implication is always true since there is no  $i < j$  among the valid indices.

Let  $\xi_X$  be the set of all solutions.

Let us denote 
$$\mathcal{B}(X) = \sum_{\substack{p_i \in X \\ \forall i \in \{1..n\}}} b_i$$

**Definition 2.** We call an optimal solution  $\mathcal{A} \in \xi_X$ , a solution that maximizes  $\mathcal{B}(\mathcal{A})$

$$\mathcal{B}(\mathcal{A}) = \max_{X \in \xi_X} \mathcal{B}(X)$$

Let  $\xi_{\mathcal{A}}$  be the set of all optimal solutions.

Let  $X_i \in \xi_X$  be a solution with the property that  $p_i \in X_i$ , let  $\xi_{X_i}$  be the set of all solutions containing  $p_i$ , and  $\mathcal{A}_i$  an optimum over the set  $\xi_{X_i}$ .

**Lemma 1.**

$$\mathcal{B}(\mathcal{A}) = \max_{i \in \{1..n\}} \mathcal{B}(\mathcal{A}_i)$$

*Proof.* We use the following property of the maximum:

$$\max_{x \in A \cup B} f(x) = \max(\max_{x \in A} f(x), \max_{y \in B} f(y))$$

We show that  $\xi_X = \bigcup_{i \in \{1..n\}} \xi_{X_i}$ . Obviously  $\xi_{X_i} \subseteq \xi_X, \forall i$  since  $\xi_X$  is the set

of all solutions, and also  $\bigcup_{i \in \{1..n\}} \xi_{X_i} \subseteq \xi_X$ . For the reverse implication, if

$X \in \xi_X$ , according to Definition 1, there is at least one element  $p_j \in X$ , but then  $X \in \xi_{X_j}$  because  $\xi_{X_j}$  is the set of all solutions that contain  $p_j$ .  $\square$

This implies that by finding the maximum in each set  $\xi_{X_i}$  and selecting the highest value found, we obtain the global maximum. It also implies that the element in  $\xi_{X_i}$  for which this maximum is achieved is a global optimum.

*Explanation:* In algorithm 3.4.1, the outer loop iterates over the local optima  $\mathcal{A}_i$ .

We show how the optimum can be found over the set  $\xi_{X_1}$ . The solution can be easily generalized since the table of slots is periodic and a period with the length of one slot table revolution can be chosen so that it starts with a slot associated with any of the paths  $p_j$ .

*Explanation:* In the implementation of the algorithm, this is achieved by duplicating the list of paths with the proper increment in arrival time and selecting  $n$  paths starting at position  $i$ . As an optimization, paths that are already known to conflict with path  $p_i$  in terms of order of arrival are already discarded at this point.

Let  $Q_{1,j}$  be a subsets of  $A$  so that  $Q_{1,j} = \{p_i \in A \mid 1 \leq i \leq j\}$ .

Let  $X_{1,j}$  with  $j \in \{1..n\}$  be a solution with the property that  $X_{1,j} \subseteq Q_{1,j}$  and  $p_1 \in X_{1,j}$  and  $p_j \in X_{1,j}$ . Let  $\xi_{X_{1,j}}$  be the set of all such solutions.

Note that an  $X_{1,j}$  does not necessarily always exist, as  $p_1$  and  $p_j$  may produce out-of-order deliveries thus  $\xi_{X_{1,j}}$  may be the empty set, but a solution exists at least for  $j = 1$  which is  $X_{1,1} = \{p_1\}$ .

We define  $\mathcal{A}_{1,j}$  as an optimal solution within the subset  $\xi_{X_{1,j}}$ , thus

$$\mathcal{B}(\mathcal{A}_{1,j}) = \max_{Y \in \xi_{X_{1,j}}} \mathcal{B}(Y)$$

Since  $\xi_{X_{1,1}}$  has only one element, which is  $\{p_1\}$ ,  $\mathcal{A}_{1,1} = \{p_1\}$ .

**Lemma 2.** *For any  $k > 1$ , if  $\mathcal{A}_{1,k}$  exists, we can compute  $\mathcal{A}_{1,k} = \mathcal{A}_{1,j} \cup \{p_k\}$  by selecting  $j < k$  which maximizes  $\mathcal{B}_{\mathcal{A}_{1,j}}$  with the restriction that  $p_j$  and  $p_k$  produce in-order delivery.*

*Proof.* We first prove that the in-order delivery condition for  $p_j$  and  $p_k$  is sufficient to ensure in-order delivery for the entire set  $\mathcal{A}_{1,k}$ . If  $\mathcal{A}_{1,j}$  is a solution,  $r_j > r_i, \forall i < j \Rightarrow r_k > r_j > r_i, \forall i$  with the property that  $p_i \in \mathcal{A}_{1,j}$ . Since we are only interested in the case where  $\mathcal{A}_{1,k}$  exists,  $r_k < r_1 + s$ , but  $r_1 < r_i, \forall i$  with the property that  $p_i \in \mathcal{A}_{1,j} \Rightarrow r_k < r_i + s$ , for the same values of  $i$ , which is the second property required by definition 1.

We prove by mathematical induction that this method of constructing  $\mathcal{A}_{1,k}$  ensures the optimality criterion.  $\mathcal{A}_{1,1} = \{p_1\}$  is obviously optimal, since when a single path is available no more bandwidth can be delivered than that provided by the path itself.

We prove the induction step by contradiction. Assume  $\mathcal{A}_{1,1} \dots \mathcal{A}_{1,k-1}$  are optimal sub-problem solutions as earlier described. If  $\mathcal{A}_{1,k}$  were not an optimal solution, there exists  $\mathcal{A}'_{1,k}$  so that  $\mathcal{B}(\mathcal{A}'_{1,k}) > \mathcal{B}(\mathcal{A}_{1,k})$ .

$\mathcal{A}'_{1,k}$  contains at least one element  $p_j$  where  $j < k$ . Let  $p_j$  be the element with the highest index  $j < k$ . Let  $\mathcal{A}'_{1,j} = \mathcal{A}'_{1,k} \setminus \{p_k\}$ .  $\mathcal{A}'_{1,j}$  is a set containing only elements from  $Q_{1,j}$  and provides in-order delivery because its superset  $\mathcal{A}_{1,k}$  provides in-order delivery.

$\{p_j, p_k\}$  provides in-order delivery for the same reason, which implies that  $\mathcal{A}_{1,j} \cup \{p_k\}$  respects the requirements for in-order delivery. It follows that  $\mathcal{A}_{1,k}$  is at least as good a solution as  $\mathcal{A}_{1,j} \cup \{p_k\}$  and as a result  $\mathcal{B}(\mathcal{A}_{1,k}) \geq \mathcal{B}(\mathcal{A}_{1,j}) + b_k$ , but  $\mathcal{B}(\mathcal{A}'_{1,k}) > \mathcal{B}(\mathcal{A}_{1,k}) \Rightarrow \mathcal{B}(\mathcal{A}'_{1,j}) + b_k > \mathcal{B}(\mathcal{A}_{1,k}) \geq \mathcal{B}(\mathcal{A}_{1,j}) + b_k \Rightarrow \mathcal{B}(\mathcal{A}'_{1,j}) > \mathcal{B}(\mathcal{A}_{1,j})$  which is impossible, because  $\mathcal{A}_{1,j}$  was already assumed to be an optimal solution to the  $\xi_{X_{1,j}}$  subproblem (from a previous induction step, as  $j < k$ ).  $\square$

**Lemma 3.**

$$\mathcal{B}(\mathcal{A}_1) = \max_{i \in \{1..n\}} (\mathcal{B}(\mathcal{A}_{1,i}))$$

*Proof.* We again use the property that:  $\max_{x \in A \cup B} f(x) = \max(\max_{x \in A} f(x), \max_{y \in B} f(y))$ . We show that  $\xi_{X_1} = \bigcup_{i=1..n} \xi_{X_{1,i}}$ . It is first of all obvious that  $\xi_{X_{1,i}} \subseteq \xi_{X_1}$  as any element of  $\xi_{X_{1,i}}$  is a solution and it contains  $p_1$ , therefore  $\bigcup_{i=1..n} \xi_{X_{1,i}} \subseteq \xi_{X_1}$ . For the proving the reverse implication  $\xi_{X_1} \subseteq \bigcup_{i=1..n} \xi_{X_{1,i}}$ , consider an element  $X_1 \in \xi_{X_1}$ . Since  $n$  is a finite value, we can find  $j \leq n$  so that  $j$  is the highest index of an element  $p_j \in X_1$ , that is  $\nexists k > j, p_k \in X_1$ . It results that  $X_1 \subseteq Q_{1,j}$ , but at the same time we also know that  $p_1 \in X_1$ ,  $p_j \in X_1$  and  $X_1$  is a solution, therefore  $X_1 \in \xi_{X_{1,j}}$ .  $\square$

This implies that by finding the maximum in each set  $\xi_{X_{1,j}}$  and selecting the highest value found, we obtain the maximum over  $\xi_{X_1}$ . It also implies that the element in  $\xi_{X_{1,j}}$  for which this maximum is achieved is optimum over  $\xi_{X_1}$ , which based on Lemma 1 also a global optimum.

### 3.5 Iterative maximum-bandwidth-search multi-path

As mentioned in Section 3.3 the flow algorithm may generate solutions with a less than optimal header overhead. In addition some of the paths may need to be discarded due to out-of-order deliveries. On the other hand the single-path approach is more restrictive in terms of routing and may not always find a solution, but it does optimize slot arrangement for maximum useful throughput.

We study another method of computing multi-path allocations based on the single path algorithm. This method is used in Model 8 in addition to the other methods presented in this chapter. Compared to the flow approach, the advantage is that this method makes use of the header overhead optimization and the number of different paths can be limited to a given value. The disadvantage is that unlike the flow algorithm it cannot displace unfavorable paths once they are allocated.

A formal description of the algorithm is given in Algorithm 3.5.1. The *allocate* function implements the functionality of Algorithm 3.1.2 with the difference that *crtS* is initialized to  $S \setminus \text{slotMask}$  such as to deny the usage of slots in *slotMask*. The function is assumed to return a list of the allocated slots.



**Algorithm 3.5.1:** Iterative Maximum Bandwidth search algorithm

---

```

allowedPathLength  $\leftarrow$  length of shortest path from Source to Destination;
pathLengthLimit  $\leftarrow$  allowedPathLength + 16;
bwneeded  $\leftarrow$  bwr;
slotMask  $\leftarrow$   $\emptyset$ ;
paths  $\leftarrow$  0;
while bwneeded > 0 do
    bwrequest  $\leftarrow$  bwneeded;
    S  $\leftarrow$  allocate(Source, Destination, bwrequest, allowedPathLength, slotMask);
    if success then
        finish;
    else if paths  $\neq$  maxPaths then
        bwrequest  $\leftarrow$  any;
        S  $\leftarrow$ 
            allocate(Source, Destination, bwrequest, allowedPathLength, slotMask);
        if success then
            bwneeded  $\leftarrow$  bwneeded -  $\mathcal{B}_S$ ;
            slotMask  $\leftarrow$  slotMask  $\cup$  S;
            paths  $\leftarrow$  paths + 1;
            continue;
        end
    end
    allowedPathLength  $\leftarrow$  allowedPathLength + 1;
    if allowedPathLength > pathLengthLimit then
        fail;
    end
    for i where si  $\in$  slotMask do
        slotMask  $\leftarrow$  slotMask  $\cup$  {si+n-1 mod n};
    end
end

```

---

The algorithm executes up to *maxPaths* iteration. In every iteration it first attempts to allocate all remaining bandwidth using one path of minimal length. If the allocation is unsuccessful, the algorithm has two options: to increase the allowed path length or to allocate part of the bandwidth using a minimal length path. The second option is preferred unless the maximum number of paths has already been reached. When no bandwidth at all can be allocated for the given path length the algorithm increases the allowed path length. The same *pathLengthLimit* of length of the minimum path plus 16 is used as in the case of the single-path search as well as the limit of  $10^7$  explored paths.

To avoid out-of-order deliveries, the algorithm uses a slot-masking technique (Figure 3.14). A slot mask is updated with all slots that were allocated

(Algorithm 3.5.1, Line 1) but in addition, every time the allowed path length is increased one guard slot is added before a previously allocated slot or a previously marked guard slot. If a path of length  $n$  is allocated, before a path of length  $n + m$  is allocated  $m$  guard slots will have been inserted in front of the slots used by the path of length  $n$ . This ensures that packets traveling over the path of length  $n + m$  will have enough time to reach the destination and will not be overtaken by packets traveling over the shorter path.

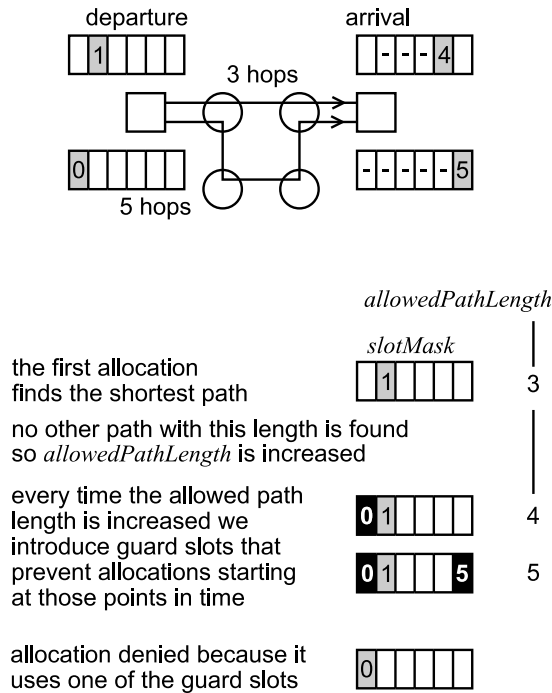


Figure 3.14: In-order delivery is ensured through the use of guard slots.

The complexity of the algorithm is linear in the number of allowed paths, but the single path allocation which is a sub-step of the algorithm has exponential complexity in the length of the paths.

### 3.6 Related Work

A solution which performs mapping, single path routing and slot allocation in the *Æthereal* network is presented in [HGR07]. While we do not perform

mapping, our solution is sufficiently similar to be applicable in a combined mapping-routing flow.

Enhancements to the solution in [HGR07] were proposed in [HCG07b] to deal with multiple use-cases and in [HCG07a] to share allocated timeslots between multiple connections. These enhancements do not directly concern the algorithm allocating individual channels but rather the relation between the different allocations.

A graph-splitting approach was employed in [MBD<sup>+</sup>05] and [MMB07] to provide online allocation. The algorithms based on graph splitting have the advantage of running in polynomial time, but they can only produce one-slot allocations.

A solution for path and slot allocation in the Nostrum [MNTJ04b] network but also applicable to *Æthereal* is presented in [LJ08]. The solution is a technique equivalent to the graph splitting approach entitled “logical networks.” The algorithm is also part of an iterative approach.

A rip-up strategy to deal with multiple channel allocations is used in [SBG<sup>+</sup>08]. Individual allocations are still performed by enumerating all feasible paths and selecting the best path found. The paper implies that storage memory for all paths is required, which we avoid in our backtracking solution. A detour of maximum 2 is allowed, which is much lower than the value we allow in our experiments. The rip-up approach could also be employed in conjunction with our algorithms.

We have previously proposed a multi-path allocation algorithm producing disjoint paths for security applications in [SG11]. Multi-path routing with disjoint paths has been previously studied in [Kol05, LG01]. Applications of multi-path routing for security, load balancing and fault tolerance are discussed in [Rab89].

Multi-path routing in NoCs has been previously proposed in [MABDM07], the method presented there requires a complex mechanism to ensure in-order delivery but it does not require complex calculations to find the paths like our solution does. In-order delivery in larger (system-level) networks was studied in [KMF<sup>+</sup>05].

The problem of multi-path routing in networks with resource reservation i.e. asynchronous transfer mode or ATM was studied by Cidon et al. [CRS99, CRS97], and was shown to provide a benefit in terms of connection establishing time, while having mixed results from the bandwidth point of view.

A close equivalent to our proposed multi-path routing scheme in large scale networks is non-redundant dispersity routing [Max07].

An overview of routing algorithms for QoS NoCs is presented in [CN98]. The algorithms referenced here however do not need to take into account the complexity of performing the allocation in the time domain as well (slot allocation).

Our exhaustive search algorithms improve upon the state of the art in terms of size of the network they are able to deal with and the maximum allowed detour. To our knowledge we are the first to propose multi-path routing in the context of networks based on the contention-free routing model.

### **3.7 Conclusions and future work**

In this chapter we have presented in detail the allocation algorithms used for the experiments in Chapter 2. In general we strive to achieve optimality in our solutions, either by performing an exhaustive search of the solution space or by designing algorithms that find an optimum in more efficient ways.

We accomplish this to a large extent, but even though the results of independent steps in the allocation process are known to be optimal the overall result of the allocation is not necessarily so. With the addition of heuristics we produce results very close to the theoretical bounds, as attested by the experimental results in Chapter 2.

While our methods compare favorably to the state of the art in terms of quality of the results we do not yet support all the features of the state of the art. The ability to perform mapping, simultaneously deal with multiple usecases [HG07] and support for channel trees [HCG07a] are subjects for future work.

## Chapter 4

### Latency and slot selection

In the previous chapters we have focused entirely on allocating paths through the network that can supply a given bandwidth. In this chapter we shift our attention to the latency of network communication. In *Æthereal* the network latency has two components: the network traversal latency, which depends only on the path length, and the scheduling latency, the time a connection has to wait for its turn in the TDM table. While in terms of network traversal latency not much can be done assuming a minimum length route is already selected, we explore how latency is affected by the selection of different patterns of slots in the TDM schedule. We also determine the effect of network latency on the execution time of actual applications under different optimization scenarios.

The operation of slot selection is performed after a path between source and destination has been found, and consists of selecting a subset of the slots available on that path, subset which preferably has a minimal number of elements and necessarily meets the bandwidth and latency requirements. Slot selection is performed for each of the candidate paths found by the path-finding algorithm described in Chapter 3.

The problem can be formulated as follows: along a given candidate path, a set of slots with proper alignment (available slots) is found. A minimal subset of this set should be selected, which provides a required bandwidth and latency.

A greedy algorithm to solve this problem was proposed in [Han09]. This algorithm was however not optimal, in that it did not minimize the resources that were allocated to satisfy the constraints. We propose an algorithm based on dynamic programming that we show to be optimal. The new algorithm presents savings in terms of used resources averaging 6% at given utilization

ratios while at certain design points (particular combinations of latency and bandwidth requirements) it can provide an improvement as high as 33%.

The algorithms we present here are applicable both the *Æthereal* network, and the *dAElite* network that we propose in Chapter 6. Some of the optimizations in these algorithms specifically target the header overhead of *Æthereal* while our network does not have a header overhead. We will indicate at the proper time what changes need to be introduced in the algorithm to support the different models. When comparing to the algorithm in [Han09], we use the model with headers, as this is the one targeted by the competing algorithm.

We also develop a more powerful model to express latency constraints which allows setting bounds on the latency of longer messages. We provide an algorithm for optimally solving problems that use this formulation as well.

This chapter is organized as follows: Section 4.1 presents the effect of slot selection on communication and application performance. In Section 4.2, we formalize the problem definition. Section 4.3 presents the previously used algorithm for the problem of slot selection. Our proposed algorithm and a proof of its optimality is presented in Section 4.4. An enhanced problem formulation and its solution are presented in Section 4.5. The complexity of the proposed algorithms is analyzed in Section 4.6. Experimental results are presented in Section 4.7. Related works are presented in Section 4.8. Section 4.9 presents our conclusions.

## 4.1 The effect of slot selection on communication and application performance

In this section we study the effect of slot selection on communication parameters and the performance of applications that are using the NoC to communicate with a remote memory. For the communication performance we will use analytical measures, while application performance will be evaluated on an FPGA system based on the *Æthereal* NoC and Microblaze processors. The analytical model will assume the presence of headers.

Under the contention-free routing model link bandwidth is divided into a discrete number of TDM slots. Each communication channel is allotted a number of slots that determine the delivered bandwidth in an almost linear fashion. The distribution in time of these slots has an impact on the time a connection has to wait for its turn during a TDM wheel revolution and hence is important when considering latency restrictions.

Even when latency is not critical from the point of view of the applications, due perhaps to latency hiding techniques, lower latencies may have a beneficial effect on the size of the buffers used in communication as for optimal operation buffer sizes are proportional to the round trip delay.

Particularities of the *Æthereal* implementation regarding the header overhead make slot selection a more difficult problem. *Æthereal* employs a header carrying routing information in the first slot out of a sequence of consecutive slots belonging to the same connection and also repeats the header every three slots in longer such sequences to allow transmitting credits.

The result of this is that a sparse distribution of slots provides a better latency but has a worse header overhead while a dense (grouped) distribution has worse latency but better payload efficiency (Figure 4.1).

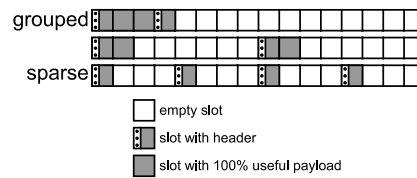


Figure 4.1: Header overhead varies with the distribution of slots.

Many combinations, regular and irregular are possible. In the experiments we present here we will use the patterns in Figure 4.2 which offer various latency - header overhead trade-offs. The header overhead is always between 1 word in 3 (the size of a slot is 3 words) and 1 word in 9, thus between 11.1% and 33.3%. The reported latency represents the average waiting time of a two-word message (thus a message that only requires one slot) before it can be transmitted. The efficiency is the ratio of useful payload to the total amount of data transmitted on a link.

The data present in all these figures was computed using an analytical model, which, due to the predictable nature of the network is completely accurate.

The average latency is not necessarily the only concern, in particular for applications with real-time requirements the highest latency may be of more importance. Figure 4.3 presents a histogram of the latencies for 2 word messages of the allocation schemes in Figure 4.2 that use 8 slots. As can be expected though, the result is very much in line with the average latency value, and the distribution of latency values is flat.

For messages up to two words in length a single slot is sufficient to transmit

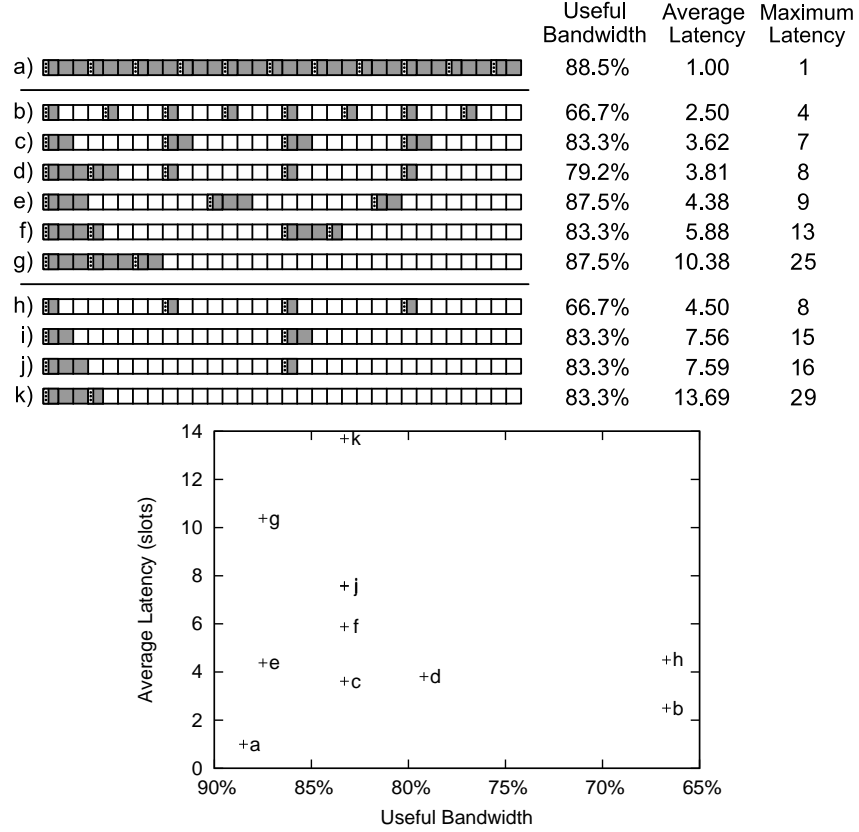


Figure 4.2: The average latency and bandwidth delivered by allocations using different patterns of 32, 8 and 4 slots out of 32.

the entire payload. The maximum latency is thus the longest delay between two allocated slots. When we consider larger message sizes, computing the latency becomes more difficult. Figure 4.4 shows a histogram of latencies for 3-words messages.

It can be easily seen in this case that the sparse distribution of slots (a) is no longer the most efficient one, being surpassed by (b) and on average also by (d). A 3-word messages size is typical for a single word write transactions: one word represents transaction qualifiers, one is the address and one the data to be written.

When the message size is even further increased, for example a 10-word message which would correspond to a 8-word write transaction for example the



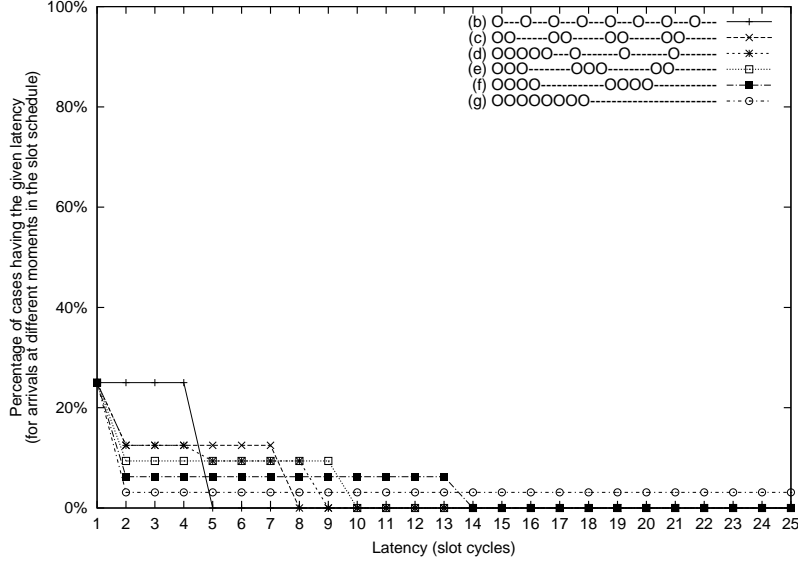


Figure 4.3: Histogram of message latency (2 data words) for various slot distributions.

update of a line of cache, the balance is shifted even further to the advantage of grouped slot patterns and the disadvantage of sparse patterns. This can be seen in Figure 4.5. Slot distributions that for short message sizes were surpassed in terms of both bandwidth and latency: (c) and (f), now provide the best and second-best average latency.

In Section 4.4 we propose an algorithm for slot allocation that considers only the maximum 2-words latency problem, while in Section 4.5 we will introduce an algorithm that optimizes the slot allocation for the latencies of longer messages.

### Measurements of the effect of latency in a real system

We would like to determine if the previous analytical measures of latency have a corresponding effect on the performance of real-life applications. We use an FPGA prototype supported by the  $\mathcal{A}$ ethereal network on chip in which a MicroBlaze processor running various applications performs accesses to a remote memory, i.e., a memory to which it only has access through the network. The test setup will be presented in detail in Chapter 7.

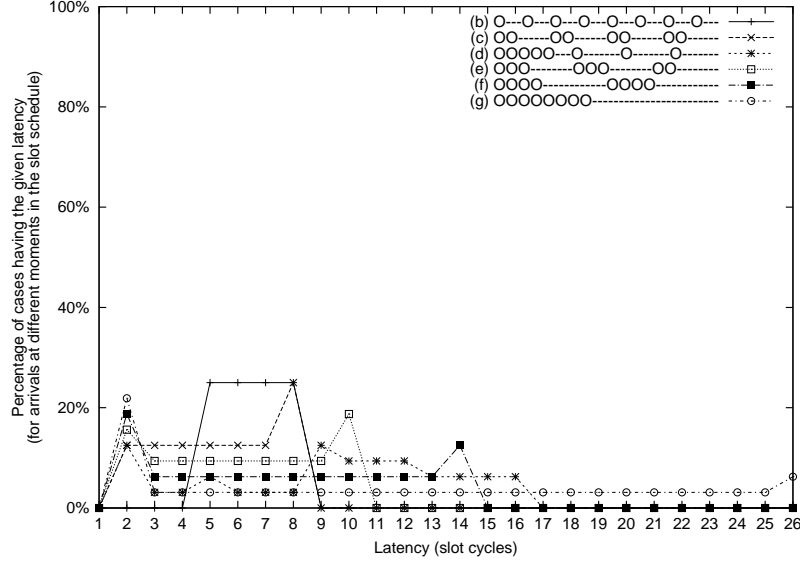


Figure 4.4: Histogram of message latency (3 data words) for various slot distributions.

As benchmark applications we use from the most to the least demanding in terms of memory bandwidth: remote memory read and write loops, a small subset of the Livermore Loops kernels [McM86], and a JPEG decoding application. For each of these applications, each of the slot patterns in Figure 4.2 is employed on both the request and response path (always with the same number of slots both in both directions), and with all possible alignments between the request and response slots. We evaluate both the cases where latency hiding techniques were used and where they were not.

In Figures 4.6-4.8, we present histogram of the relative increase in application running time compared to the case where the entire link bandwidth was dedicated to the communication channel. A slot table of 32 slots was used in the experiments, thus  $1/4$  of the link bandwidth corresponds to 8 allocated slots and  $1/8$  of the link bandwidth to 4 allocated slots.

Figure 4.6 represents experiments where no latency hiding techniques were used. As the system is most sensitive to increases in latency, it displays the highest variations in application running time.

It is important to note that there is significant overlap between the scenarios using 4 and 8 slots, that is, for some patterns of allocated slots, same or better

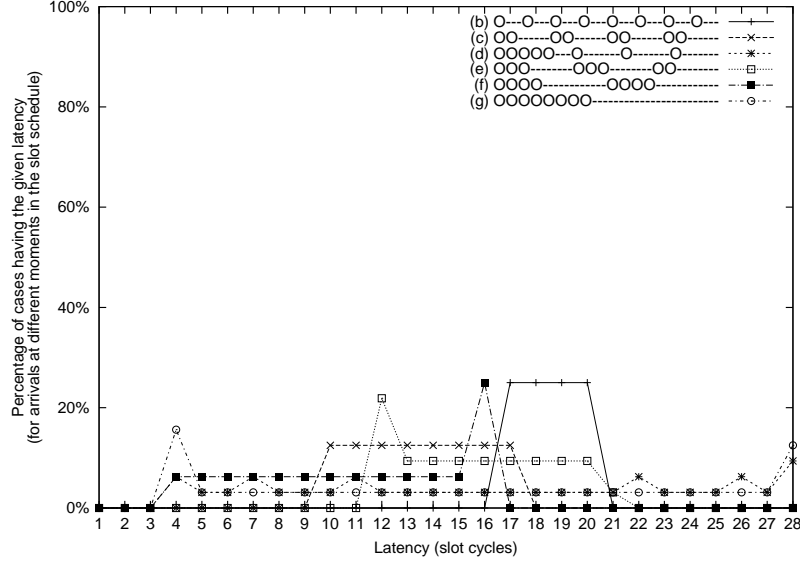


Figure 4.5: Histogram of message latency (10 data words) for various slot distributions.

performance can be obtained by a 4 slot allocation compared to an 8 slot allocation which essentially uses double the network resources. Also some of the 8 slot allocation perform (almost) same as well as the full bandwidth allocation. The gap in performance between the most and least efficient allocation is significant, the performance in the synthetic benchmarks varying by a factor of almost 3x. The JPEG application is computationally intensive and only uses the remote memory to read the input data and produce the output result. Some variation in performance is observed nevertheless, but not to the same extent as in the case of the synthetic benchmarks.

The second round of experiments involves a latency hiding technique for the write transactions, that is posted writes (with a bandwidth optimization to group consecutive write operations into bursts when the addresses are consecutive). The same type of histograms as in the previous case are presented in Figure 4.7. There is now a more clear separation between the allocations using 4 and 8 slots. The spread of results is also diminished (note the different X axis.)

With latency hiding techniques for both the read and write operations (Figure 4.8) there is a clear separation based on delivered bandwidth, visible especially

in the cases of the read and write loops. There is still some spread between the best and worst case for the same number of slots, but much reduced when compared to the previous scenarios. The JPEG application, with its very low average bandwidth requirement is virtually unaffected by both the pattern and the number of allocated slots.

Our conclusion is that, if latency hiding techniques are not used, the distribution of slots has a very important effect on application performance.

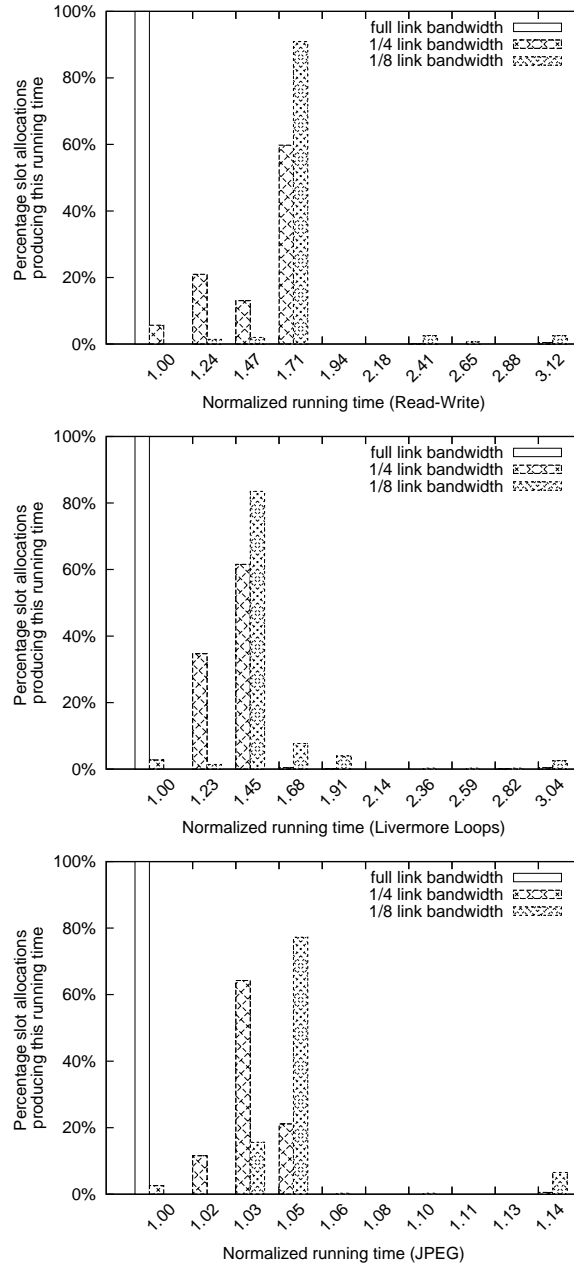


Figure 4.6: Histogram of slowdowns generated by different slot selection when no latency hiding techniques are used.

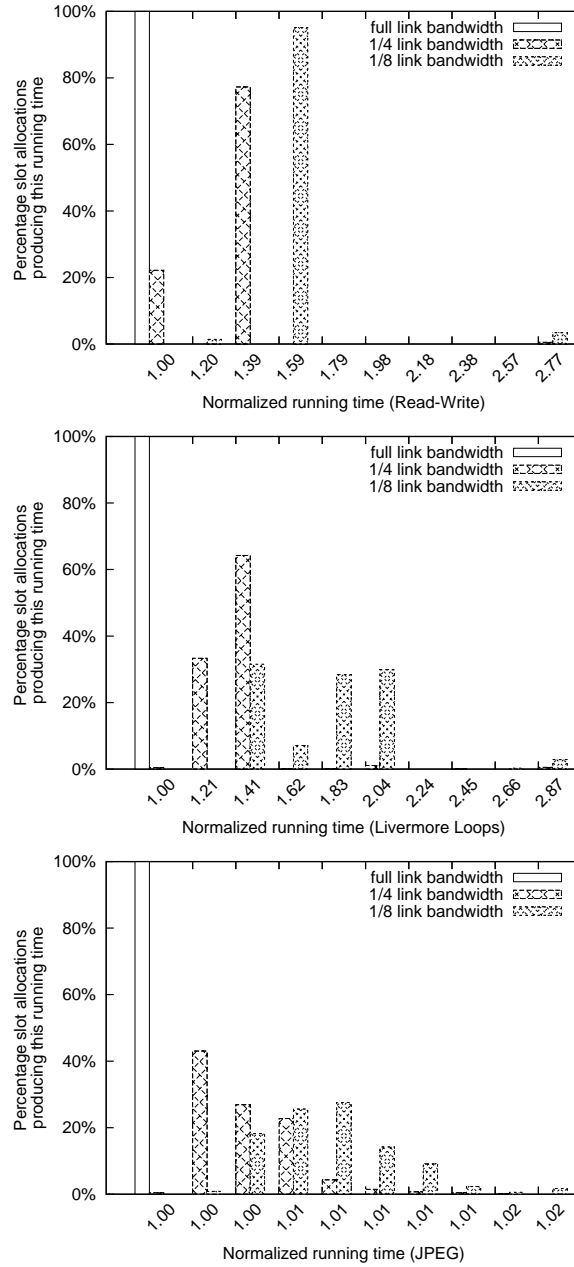


Figure 4.7: Histogram of slowdowns generated by different slot selections, when using posted writes, write coalescing and no read prefetch.

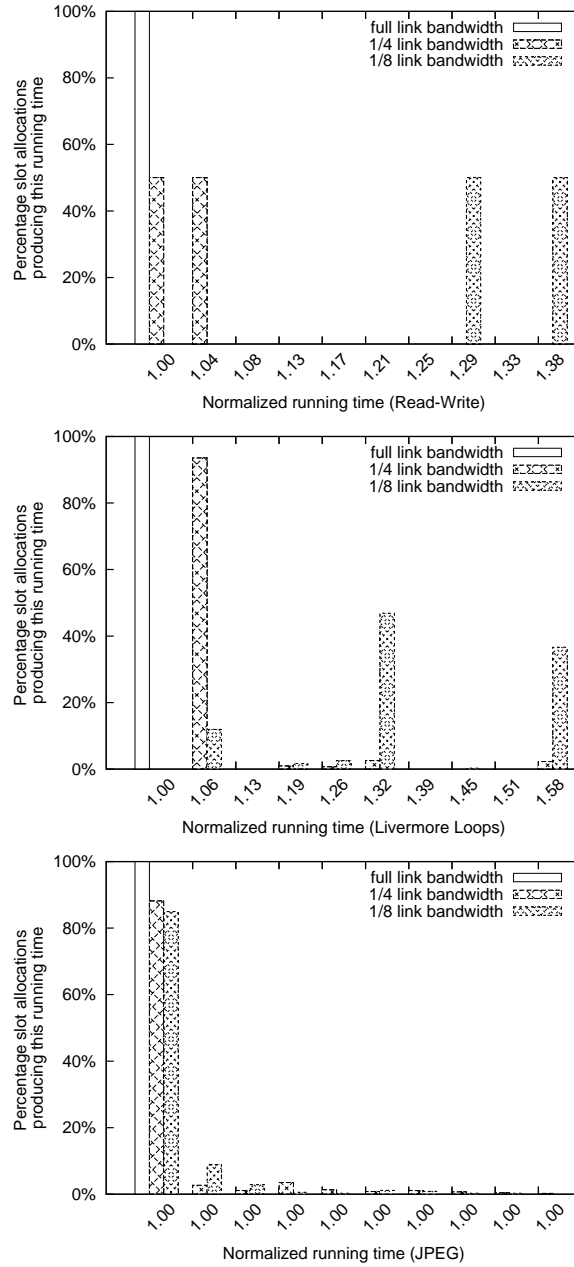
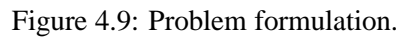


Figure 4.8: Histogram of slowdowns generated by different slot selections, when using posted writes, write coalescing and read prefetch.

It is relatively easy to verify whether a given set of slots provides the desired bandwidth and latency. It is sufficient to determine the maximum distance (in time) between two slots the number of slots and which slots present a header overhead. Comparatively, given a set of slots which provides more than the necessary bandwidth and/or better latency, it is much more difficult to determine a minimal subset of slots which provides the minimum requirements. Finding this subset is the problem that we would like to solve.

The worst-case scheduling latency for a communication channel  $l$  is equal to the maximum distance in time between two allocated slots. It is expressed in slots. The latency for the transmission of one word of data over the network is equal to the scheduling latency plus the network traversal latency which is trivial to compute since it is equal to the path length.



The algorithm previously used to solve this problem was proposed in [Han09]. It is a greedy algorithm which does not guarantee an optimal solution. The steps of the algorithm are illustrated in Figure 4.10 and a formal description is given in Algorithm 4.3.1.

The algorithm always allocates the first available slot, and then attempts to find available slots situated at the maximum distance imposed by the latency limit.



---

**Algorithm 4.3.1:** Slot selection algorithm presented in [Han09]

---

**input** : set of available slots  $A \subseteq S = \{s_1..s_n\}$   
            $l$  maximum allowed latency measured in slot periods  
            $B_r$  required bandwidth  
**output**:  $\mathcal{A} \subseteq A$  which satisfies the bandwidth and latency constraints

```

firstFreeSlot  $\leftarrow$  1;
while firstFreeSlot  $\leq n \wedge s_{\text{firstFreeSlot}} \notin A$  do
  | firstFreeSlot  $\leftarrow$  firstFreeSlot + 1;
end
if firstFreeSlot  $> l$  then
  | fail;
end
 $\mathcal{A} \leftarrow \{s_{\text{firstFreeSlot}}\}$ ;
lastAllocatedSlot  $\leftarrow$  firstFreeSlot;
while firstFreeSlot +  $n$  - lastAllocatedSlot  $> l$  do
  | slot  $\leftarrow$  min(lastAllocatedSlot +  $l$ ,  $n$ );
  | while slot  $>$  lastAllocatedSlot  $\wedge s_{\text{slot}} \notin A$  do
  | | slot  $\leftarrow$  slot - 1;
  | end
  | if lastAllocatedSlot = slot then
  | | fail;
  | end
  | lastAllocatedSlot  $\leftarrow$  slot;
  |  $\mathcal{A} \leftarrow \mathcal{A} \cup \{s_{\text{slot}}\}$ ;
end
slot  $\leftarrow$  firstFreeSlot;
while  $B_{\mathcal{A}} < B_r$  do
  | if slot  $> n$  then
  | | fail;
  | end
  | if  $s_{\text{slot}} \in A$  then
  | |  $\mathcal{A} \leftarrow \mathcal{A} \cup \{s_{\text{slot}}\}$ ;
  | end
  | slot  $\leftarrow$  slot + 1;
end

```

---

If these slots are occupied, the algorithm retreats to an available slot farthest from the current slot but within the latency limit.

Slots found in this way are repeatedly added to the solution until the wrap-around latency is satisfied or the algorithm fails to find a slot that can be

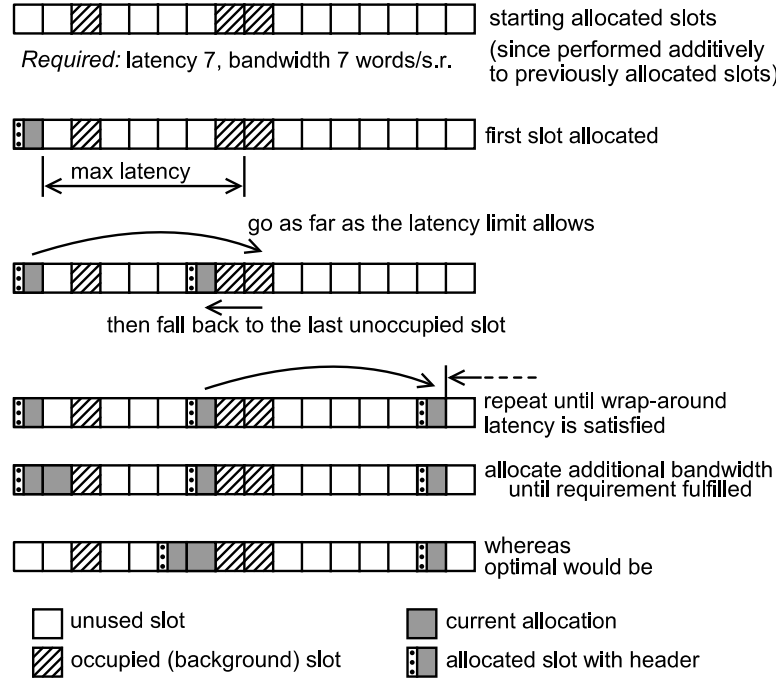


Figure 4.10: Steps of the original algorithm.

allocated.

In the end, the algorithm adds available slots as necessary until bandwidth requirement is satisfied.

As illustrated in Figure 4.10 the algorithm does not always produce an optimal result. Some of the causes may be:

- The original slot choice leads to a suboptimal solution. This could be avoided by running the algorithm  $n$  times starting at each slot.
- Successive allocations do not take into account the fact that multiple consecutive slots have to be allocated anyway because of the bandwidth requirement.
- The final allocation for bandwidth does not guarantee a maximum efficiency in terms of headers.

#### 4.4 Proposed algorithm for the slot selection problem

We propose an algorithm for slot selection based on dynamic programming. We show this algorithm to be optimal in that it uses a minimal number of slots to meet the latency and bandwidth constraints.

A formal description of our algorithm is given in Algorithms 4.4.1 and 4.4.2. We denote  $r$  as the number of slots after which a header needs to be repeated,  $\text{slotSize}$  the size of a slot in words,  $\text{headerSize}$  the size of the header in words. If the network does not make use of headers the value of  $\text{headerSize}$  can be set to 0 and  $r$  has a value of 1.  $n$  is the number of slots in the TDM table. We assume a  $\text{Latency}(A)$  function is defined which computes the latency provided by a set of slots  $A$ .

Our algorithm first restricts solutions to a list of slots starting with one non-selected slot followed by one selected slot (Figure 4.11). By iterating over all rotations of the slot table, with wrap-around, we ensure the coverage of the entire solution space, one exception being a table with all slots selected, which is treated as a separate case.

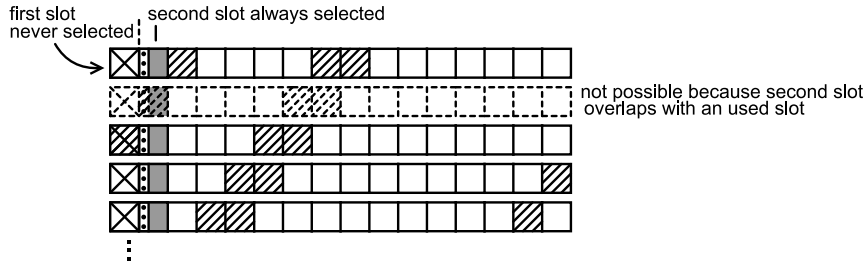


Figure 4.11: Solutions start with one non-selected, followed by one selected slot.

We then build a set of optimal partial solutions  $A_{k,i,j}$ . Partial solutions are constructed by adding slots to other partial solutions until the set of partial solutions is sufficient to guarantee it contains an optimum over the entire solution space.

In the following we give a formal description and in-depth explanation of our algorithm. We will assume, without reducing generality that all solutions and partial solutions start with a non-selected slot followed by one selected slot.

Let  $S$  be the set of all slots  $\{s_1, s_2, \dots, s_n\}$ , and let  $A \subseteq S$  be the set of available slots  $A = \{s_i \in S \mid s_i \text{ is not occupied}\}$ .

**Algorithm 4.4.1:** Optimal slot selection

---

**input** : set of available slots  $A \subset S = \{s_1..s_n\}$   
 $l$  maximum allowed latency measured in slot units  
 $B_r$  required bandwidth  
**output**:  $\mathcal{A}$  which minimizes  $|\mathcal{A}|$  while satisfying bandwidth and latency constraints

**if**  $B_r > B_A \vee \text{Latency}(A) > l$  **then**  
| fail;  
**end**  
 $\text{hdrSlotSize} \leftarrow \text{slotSize} - \text{headerSize};$   
solution  $\mathcal{A} \leftarrow A;$   
 $B_{\mathcal{A}} \leftarrow B_A;$   
**for**  $\forall i \in \{1, 2, \dots, n\}$  **do**  
| reorder original  $A, S$  to start with slot  $s_i$ ,  $s_i$  becomes  $s_1$ ;  
| **if**  $s_2 \in A$  **then**  
| |  $\mathcal{A}_{2,1,1} \leftarrow \{s_2\};$   
| |  $B_{\mathcal{A}_{2,1,1}} \leftarrow \text{short};$   
| | **for**  $\forall k \in \{3, 4, \dots, n\}$  **do**  
| | | **for**  $\forall i \in \{0, 1..r-1\}, \forall j \in \{1..k-1\}$  **do**  
| | | |  $\mathcal{A}_{k,i,j} \leftarrow \emptyset;$   
| | | |  $B_{\mathcal{A}_{k,i,j}} \leftarrow 0;$   
| | | **end**  
| | |  $i=1;$   
| | | **for**  $\forall j \in \{2..k-1\}$  **do**  
| | | | **for**  $\forall x \in \{0, 1..r-1\}, \forall y \in \{\max(2, k-l)..k-2\}$  **do**  
| | | | | **if**  $B_{\mathcal{A}_{k,i,j}} < B_{\mathcal{A}_{y,x,j-1}} + \text{hdrSlotSize}$  **then**  
| | | | | |  $B_{\mathcal{A}_{k,i,j}} \leftarrow B_{\mathcal{A}_{y,x,j-1}} + \text{hdrSlotSize};$   
| | | | | |  $\mathcal{A}_{k,i,j} \leftarrow \mathcal{A}_{y,x,j-1} \cup s_k;$   
| | | | | **end**  
| | | | **end**  
| | | **end**  
| | **end**  
| | **for**  $\forall i \in \{0, 1..r-1\}, \forall j \in \{2..k-1\}$  **do**  
| | |  $x = (i-1) \bmod r;$   
| | |  $\text{gain} \leftarrow \text{slotSize};$   
| | | **if**  $i = 1$  **then**  
| | | |  $\text{gain} \leftarrow \text{hdrSlotSize};$   
| | | **end**  
| | | **if**  $B_{\mathcal{A}_{k,i,j}} < B_{\mathcal{A}_{k-1,x,j-1}} + \text{gain}$  **then**  
| | | |  $B_{\mathcal{A}_{k,i,j}} \leftarrow B_{\mathcal{A}_{k-1,x,j-1}} + \text{gain};$   
| | | |  $\mathcal{A}_{k,i,j} \leftarrow \mathcal{A}_{k-1,x,j-1} \cup s_k;$   
| | | **end**  
| | **end**  
| | **end**  
| **end**  
|  $\triangleright$  search for best solution (Algorithm 4.4.2);  
**end**  
**end**

---

---

**Algorithm 4.4.2:** Search for optimal selection

---

**input** : all local data structures in Algorithm 4.4.1**output:**  $\mathcal{A}$  which minimizes  $|\mathcal{A}|$  while satisfying bandwidth and latency constraints

```

for  $\forall k \in \{n - l + 1, \dots, n\}$  do
    // LIMITING THE SEARCH TO  $n - l + 1$  ENSURES THAT THE LATENCY LIMIT
    // IS OBEYED AT WRAP-AROUND
    for  $\forall i \in \{0, 1 \dots r - 1\}, \forall j \in \{1 \dots k - 1\}$  do
        if  $\mathcal{B}_r \leq \mathcal{B}_{\mathcal{A}_{k,i,j}}$  then
            if  $j < |\mathcal{A}| \vee (j = |\mathcal{A}| \wedge \mathcal{B}_{\mathcal{A}} < \mathcal{B}_{\mathcal{A}_{k,i,j}})$  then
                // NOTE THAT  $|\mathcal{A}_{k,i,j}| = j$ 
                 $\mathcal{A} \leftarrow \mathcal{A}_{k,i,j};$ 
                 $\mathcal{B}_{\mathcal{A}} \leftarrow \mathcal{B}_{\mathcal{A}_{k,i,j}};$ 
            end
        end
    end
end
end

```

---

Let  $A_{k,i,j}$  be a subset of  $\{s_2, s_3, \dots, s_k\} \cap A$  with  $s_k \in A_{k,i,j}$ , having exactly  $j$  elements, i.e.,  $|A_{k,i,j}| = j$ , and ending with  $q$  selected slots, where  $q$  is any natural number so that  $q \bmod r = i$  or in other words  $s_{k-q} \notin A_{k,i,j}$  and  $\{s_{k-q+1}, s_{k-q+2}, \dots, s_k\} \subset A_{k,i,j}$ . Obviously  $j \leq k - 1$ .

Furthermore, partial solutions are required to obey the latency limit on the interval  $2..k$ , i.e., there should be no gap larger than  $l$  in the set of slots. They are however not required to obey the latency limit at the wrap-around of the TDM table. Whether they obey or not the latency limit at wrap-around is only a function of  $k$  because we know that  $s_k$  is the last slot belonging to the set and the first slot is  $s_2$ .

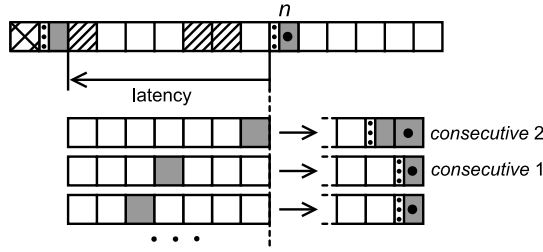


Figure 4.12: Building partial solutions by adding a slot at the end of an existing partial solution.

The reason behind the classification of partial solutions by their modulo  $r$  number of ending slots is that it enables us to compute the bandwidth obtained by attaching one additional slot at the end of the partial solution (Figure 4.12), that is, if the bandwidth delivered by  $A_{k-1,i,j-1}$  is  $\mathcal{B}_{A_{k-1,i,j-1}}$ , the bandwidth delivered by  $A_{k-1,i,j-1} \cup s_k$  is:

$$\mathcal{B}_{A_{k,(i+1) \bmod r,j}} = \begin{cases} \mathcal{B}_{A_{k-1,i,j-1} \cup s_k} + \text{slotSize-headerSize} & \text{when } i = 0 \\ \mathcal{B}_{A_{k-1,i,j-1} \cup s_k} + \text{slotSize} & \text{otherwise} \end{cases} \quad (4.1)$$

When attaching one slot to a solution in which the last slot is not selected (a non-consecutive slot, line 2 in Algorithm 4.4.1), we always pay the header penalty and the number of slots at the end of solution becomes 1.

$$\mathcal{B}_{A_{k,1 \bmod r,j}} = \mathcal{B}_{A_{k-m,i,j-1} \cup s_k} + \text{slotSize-headerSize} \quad (4.2)$$

where  $m > 1$

It is easy to see that any solution having at least two slots  $A_{k,i,j}; j \geq 2$  can be built from a solution  $A_{x,y,j-1}$  by adding a new slot on the position  $k$ , and the delivered bandwidth can be computed using one of the Equations 4.1 and 4.2. To obey the latency limit it is mandatory that  $k - x \leq l$ .

The global optimum is found by enumerating all the generated  $A_{k,i,j}$  sets and selecting the one which:

1. Provides the necessary bandwidth
2. Obeys wrap-around latency requirement
3. Has the lowest  $j$  value (number of used slots)
4. For the lowest  $j$  value has the highest  $\mathcal{B}_{A_{k,i,j}}$

### Proof of optimality

Let us denote  $\mathcal{A}_{k,i,j}$  a set  $A_{k,i,j}$  that is optimal in that for the given  $k, i, j$  it provides the largest bandwidth. We argue that  $\mathcal{A}_{k,i,j}$ , if it exists can only be obtained by adding a slot  $k$  to one optimal set  $\mathcal{A}_{x,y,j-1}$ . Indeed, if that was not the case, then  $\mathcal{A}_{k,i,j}$  would be obtained from a non-optimal  $A_{x,y,j-1}$  as  $A_{x,y,j-1} \cup \{s_k\}$  and  $\mathcal{B}_{\mathcal{A}_{k,i,j}} = \mathcal{B}_{A_{x,y,j-1}} + q$  where  $q$  is a constant dependent only on  $x$  and  $k$ , derived from Equations 4.1, 4.2. Note that  $k - x < \text{latency}$  to obey the latency requirement.

But since  $A_{x,y,j-1}$  is not optimal  $\exists A_{x,y,j-1}$  so that  $\mathcal{B}_{A_{x,y,j-1}} > \mathcal{B}_{A_{x,y,j-1}} \Rightarrow \mathcal{B}_{A_{x,y,j-1} \cup \{s_k\}} + q > \mathcal{B}_{\mathcal{A}_{k,i,j}}$  and  $\mathcal{A}_{k,i,j}$  is not optimal, which would contradict the hypothesis.

It results from here that if we generate all feasible  $A_{x,y,j-1}$  sets (or at least one set for each  $(x, y, j - 1)$ ) we can generate, if it exists, any  $\mathcal{A}_{k,i,j}$  set.

An optimum to our original problem, that is, a subset of  $S$  which satisfies the latency bound  $l$ , and has a minimum required bandwidth  $bw$ , can always be expressed as a set  $A_{k,i,j}$ , by properly selecting values for  $k, i$  and  $j$ , but since  $\mathcal{A}_{k,i,j}$  uses the same number of slots,  $j$  and it provides bandwidth as high as any of the  $A_{k,i,j}$  sets,  $\mathcal{A}_{k,i,j}$  is also an optimal solution, with the added benefit that among the solutions that use  $j$  slots it also provides the highest possible bandwidth, which goes beyond the original problem requirements.

## 4.5 Enhanced formulation of the latency problem

The previous formulation only guarantees a bound on the time a connection has to wait for its next allocated slot. This translates directly into the latency of sending one slot's worth of data. Some network messages though are longer than one slot. A single write operation for example has 3 words: a command word, an address word and a data word. Messages encoding read or write bursts may be even longer.

Obviously an implicit guarantee exists that a 4-word transaction will take at most twice as long as the 2-word transaction, a 6-words transaction 3 times as long and so on, all because once a connection got access to one slot the latency bound applies to the arrival of the next slot. A straightforward way to impose a maximum latency  $l$  for the duration of an  $n$ -word transaction is to impose a latency of  $\frac{l}{\lceil n/2 \rceil}$  to the 2-word transaction and use the previous algorithm.

This approach however may result in over-constraining the solution, as we will show in section 4.7. As an alternative we present a second slot allocation algorithm that can produce an optimal allocation with the constraint that within any window of  $w$  slots,  $n$  words of data can always be delivered (Figure 4.13). By optimal allocation we mean here that a minimal number of slots is used.

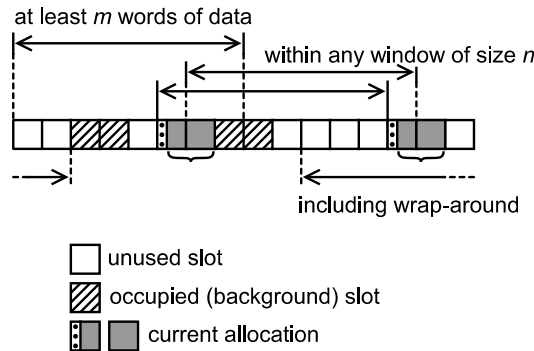


Figure 4.13: Enhanced formulation.

### 4.5.1 Slot allocation algorithm for the enhanced formulation

The algorithm we propose in Algorithm 4.5.1-4.5.2 is based on dynamic programming. Its complexity is polynomial in the size of the table of slots, but exponential in the size of the considered window.



Let  $w$  be the size of the window. The algorithm starts rotating the slot table

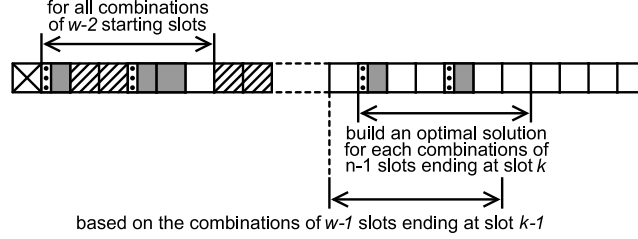


Figure 4.14: Algorithm for the enhanced formulation.

such that the first slot is an occupied slot. In the case that all slots are available, either the solution consists of all slots, which is trivial, or one slot can be arbitrarily excluded from the solution and marked as occupied.

In a top-level loop, the algorithm iterates through all possible sets of the slots  $s_2..s_{w-1}$  provided these slots are available and separately runs a search for a solution constrained to using these starting sets.

Each iteration of the top loop consists of exhaustive searches in a window of size  $w - 1$  which slides over the entire slot table.

We introduce here a formal notation and describe the algorithm steps, after which we will provide a full formal description in Algorithms 4.5.1, 4.5.2.

Let  $S$  be the set of all slots  $\{s_1, s_2, ..s_n\}$ , and let  $A \subseteq S$  be the set of available slots  $A = \{s_i \in S | s_i \text{ is not occupied}\}$ .

Let  $A_{w-1, START} \subset A \cap \{s_2..s_{w-1}\}$  be the starting set. The top loop of the algorithm iterates over all  $A_{w-1, START}$ .

We consider the property  $\mathcal{P}(A, i)$  to be true if  $A$  provides the necessary bandwidth over the window  $\{s_i..s_{i+w-1}\}$  or  $\{s_i..s_n, s_1..s_{w+i-n-1}\}$  when the window wraps-around.

We use the notation of  $A_{k, c_{k-w+2}, c_{k-w+3}, .., c_k} \subset \{s_1..s_k\}$  or  $A_{k, C_k}$  for sets having the following relation between the values of  $c_i$  and the member slots:

$$s_q \in A_{k, c_{k-w+2}, c_{k-w+3}, .., c_k} \Leftrightarrow c_q = 1, \quad \forall q \in \{k - w + 2..k\}$$

Notice that  $A_{k, c_{k-w+2}, c_{k-w+3}, .., c_k}$  may still contain elements  $s_i$  for  $i < k - w + 2$  as only the membership of elements  $s_{k-w+2}..s_k$  is enforced by the  $c$ -values.  $A_{k, C_k}$  sets divide the solution space in regions with a certain property. For different values of  $k$  these sets may overlap.

We name  $A_{k,C_k}$  a valid partial solution if:

$$A_{k,C_k} \cap \{s_1 \dots s_{w-1}\} = A_{w-1,START}$$

and

$$A_{k,C_k} \text{ has } \mathcal{P}(A_{k,C_k}, i) \forall i \in \{1 \dots k - w + 1\}$$

A set  $A_{k+1,C_{k+1}} = A_{k+1,C_{k-w+3} \dots C_k,1}$  can be obtained from either  $A_{k,0,C_{k-w+3} \dots C_k}$  or  $A_{k,0,C_{k-w+3} \dots C_k}$  by adding element  $s_{k+1}$ . A set  $A_{k+1,C_{k-w+3} \dots C_k,0}$  is equal to a set  $A_{k,C_k}$  with the proper corresponding  $c$  values. In order to verify the validity of  $A_{k+1,C_{k+1}}$  as a solution as long as it was obtained from a set  $A_{k,C_k}$  it is enough to verify  $\mathcal{P}(A_{k+1,C_{k+1}}, k - w + 2)$ . Conversely, if  $A_{k+1,C_{k+1}}$  is a valid partial solution,  $A_{k+1,C_{k+1}}/\{s_{k+1}\} = A_{k,C_k}$  is also a valid partial solution as  $A_{k,C_k}$  only needs to verify  $\mathcal{P}(A_{k+1,C_{k+1}}, 1) \dots \mathcal{P}(A_{k+1,C_{k+1}}, k - w + 1)$  which  $A_{k+1,C_{k+1}}$  already does and the absence of slot  $s_{k+1}$  has no influence since it does not belong to any of the windows.

We define an optimal valid partial solution  $\mathcal{A}_{k+1,C_{k+1}}$  a valid partial solution  $A_{k+1,C_{k+1}}$  which minimizes  $|A_{k+1,C_{k+1}}|$  (the number of used slots). We compute  $\mathcal{A}_{k+1,C_{k+1}}$  from either  $\mathcal{A}_{k,0,C_{k-w+3} \dots C_k}$  or  $\mathcal{A}_{k,1,C_{k-w+3} \dots C_k}$ , selecting the  $\mathcal{A}_{k,C_k}$  with a minimal  $|\mathcal{A}_{k,C_k}|$  and ensuring  $\mathcal{P}(\mathcal{A}_{k+1,C_{k+1}}, k - w + 2)$ . A proof the optimality of this method is presented in section 4.5.2.

We use the following numeric example to illustrate the functioning of the algorithm (Figure 4.15). In a TDM schedule with 6 slots, the slots  $s_3$  and  $s_5$  have already been reserved by other connections. We wish to provide a communication channel that can deliver 4 words of data with a maximum waiting delay of 4 time slots.

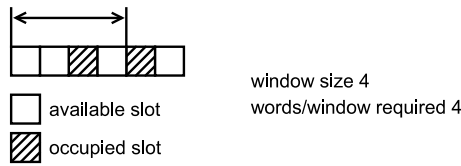


Figure 4.15: Numeric example for the enhanced problem formulation

The algorithm begins by rotating the slot table with an occupied slot in the position  $s_1$  (Figure 4.16). Since the TDM schedule is periodic this operation has no effect on the communication parameters.

The algorithm will then iterate through all combinations of the slots  $s_2-s_3$  (actually we are interested in the window  $s_1-s_3$  which is needed by the wrap

around windows, but we already know the slot  $s_1$  cannot be selected). Because slot  $s_3$  is also not available, we have only two choices for the starting set  $A_{w-1,START}$ , using the previous notation we will denote them  $A_{w-1,START} = A_{3,000} = \emptyset$  and  $A_{w-1,START} = A_{3,010} = \{s_2\}$ .

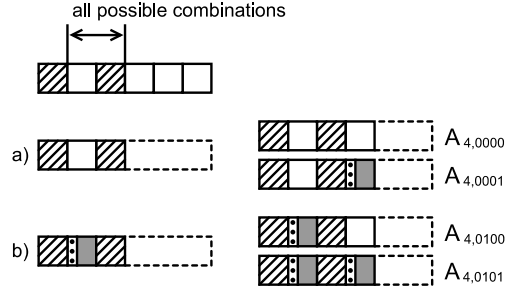


Figure 4.16: Steps of the algorithm for the enhanced formulation, all slot possibilities for the window [2-3] are considered

The rest of the algorithm will be run once for each of these sets, with the solution constrained to start with the given selection of slots. It is observed though that the starting set  $A_{3,000}$  (Figure 4.16a) does not produce viable solutions even for the first window  $s_1$ - $s_4$ , so we focus on solutions with  $A_{w-1,START} = A_{3,010}$ .

The algorithm successively generates  $A_k$  sets for values of  $k$  increasing from  $w$  to  $n$ . The algorithm does not allow  $w > n$  which in fact would not even be very useful as we could request a desired bandwidth for an entire slot table revolution instead. There are two  $A_{4,C_4}$  sets that can be generated with the starting restrictions  $A_{4,(0)100}$  and  $A_{4,(0)101}$  (Figure 4.16b) (we represented in brackets the  $c_1$  value which is known during set generation).  $A_{4,(0)100}$  does not have  $\mathcal{P}(A_{4,C_4}, 1)$  and therefore it is not a valid solution and will not be used in constructing  $A_{5,C_5}$  sets.

$A_{4,101}$  will be used to construct two  $A_{5,C_5}$  sets (Figure 4.17),  $A_{5,(1)010}$  and  $A_{5,(1)011}$ , both of which provide the necessary bandwidth. Whenever we build these sets we keep track of their cost. In general each  $A_{k,C_k}$  set can be used to build two  $A_{k+1,C_{k+1}}$  sets assuming such sets are valid solutions and correspondingly, each  $A_{k+1,C_{k+1}}$  set can be obtained from either of two  $A_{k,C_k}$  sets. For example  $A_{5,010}$  could also have been obtained from  $A_{4,001}$  if such a solution were valid. When two possibilities exist to construct a set we prefer of course the least expensive one. For convenience, in our

algorithm implementation we use a cost of  $\infty$  (actually a very high integer value in C) to denote an invalid solution. When building the  $A_{k+1,C_{k+1}}$  sets we verify  $\mathcal{P}(A_{k+1,C_{k+1}}, k - w + 2)$ .  $A_{k+1,C_{k+1}}$  is guaranteed to also verify  $\mathcal{P}(A_{k+1,C_{k+1}}, i)$ ,  $\forall i < k - w + 2$  because  $A_{k,C_k}$  already does

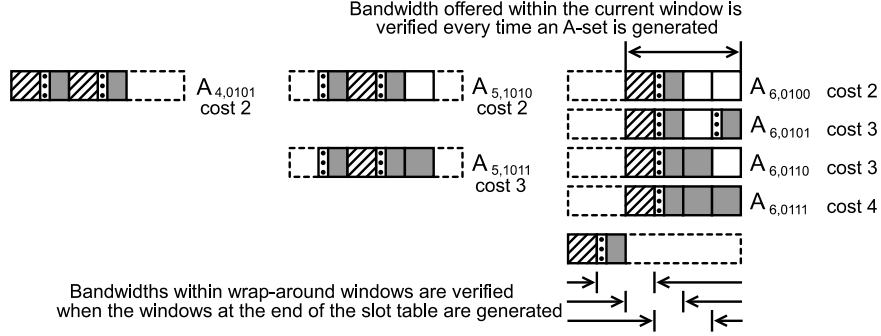


Figure 4.17: Steps of the algorithm for the enhanced formulation, A-sets are generated

The number of  $A_k$  combinations as presented in Figure 4.17 is increasing exponential and limited by  $2^{w-1}$ , where  $w$  is the size of the window. When the  $A_{n,C_n}$  sets are built, in addition to verifying  $\mathcal{P}(A_{n,C_n}, n - w + 1)$  we verify  $\mathcal{P}(A_{n,C_n}, q)$  wrap-around windows  $q \in \{n - w + 2..n\}$ . The wrap-around bandwidth can be verified just by using the value of  $A_{w-1,START}$  and  $c_{n-w+2}..c_n$  and is thus independent of the other elements in  $A_{n,C_n}$  (the elements not specified by the  $c$ -values or  $A_{w-1,START}$ ).

A global optimum to the slot selection problem is also a set of the form  $A_{n,C_n}$  for particular values of  $A_{w-1,START}$  and  $c_{n-w+2}..c_n$ . By iterating over all possible combinations of  $A_{w-1,START}$  and  $c_{n-w+2}..c_n$  and finding optimal solutions inside the partial solution spaces  $A_{n,C_n}$  we find the global optimum solution to the problem.

In Algorithm 4.5.1 we give a formal description of the algorithm.

**Algorithm 4.5.1:** Outer loop of the algorithm for the enhanced formulation

**input** : set of available slots  $V \subset S = \{s_1..s_n\}$   
            $w < n$  size of a window  
            $bw_r$  required bandwidth within each window of size  $w$   
**output**:  $\mathcal{A}$  which minimizes  $|\mathcal{A}|$  while satisfying bandwidth constraint

```

 $W \leftarrow \text{rotate}(V)$  so that  $s_1 \notin W$ ;
 $best \leftarrow \emptyset$ ;
 $bestCost \leftarrow \infty$ ;
for  $d_2d_3..d_{w-1} \in \{0,1\} \times \{0,1\} \times \dots \{0,1\}$  do
    if  $\exists i, d_i = 1, s_i \notin W$  then
        | continue;
    end
     $cost[w..n, \{0,1\}^w] \leftarrow \infty$ ;
    if  $bandwidth(0d_2d_3..d_{w-1}1) < bw_r$  then
        | continue;
    end
     $cost[w, 0d_2d_3..d_{w-1}1] \leftarrow 1 + \sum_{i=2}^{w-1} d_i$ ;
    if  $bandwidth(0d_2d_3..d_{w-1}0) \geq bw_r$  then
        |  $cost[w, 0d_2d_3..d_{w-1}0] \leftarrow \sum_{i=2}^{w-1} d_i$ ;
    end
     $\triangleright$  build partial solutions (Algorithm 4.5.2);
    for  $c_1c_2..c_w \in \{0,1\}^w$  do
        if  $cost[n, c_1c_2..c_w] < bestCost$  then
            |  $bestCost \leftarrow cost[n, c_1c_2..c_w]$ ;
            | // FOLLOW PREDECESSORS AND ADD INITIAL SLOTS
            |  $best \leftarrow \emptyset$ ;
            |  $e_{n-w+1}..e_n \leftarrow c_1..c_w$ ;
            | for  $j \leftarrow n-w$  to  $w$  do
            | |  $e_j = pre[j+1, e_{j+1}e_{j+2}..e_{j+w}]$ ;
            | | end
            |  $e_2..e_{w-1} \leftarrow d_2..d_{w-1}$ ;
            | for  $j \leftarrow 2$  to  $n$  do
            | | if  $e_j = 1$  then
            | | |  $best \leftarrow best \cup \{s_j\}$ ;
            | | end
            | end
        end
    end
end

```

---

**Algorithm 4.5.2:** Build partial solutions in the algorithm for the enhanced formulation

---

```

for  $q \leftarrow w + 1$  to  $n$  do
  for  $c_1 c_2 \dots c_w \in \{0, 1\}^{w-1}$  do
    if  $\text{bandwidth}(c_1 c_2 \dots c_w) < bw_r$  then
      | continue;
    end
    if  $c_w = 1 \wedge s_q \notin W$  then
      | continue;
    end
     $wrapOk \leftarrow \text{true};$ 
    if  $q = n$  then
      for  $j \leftarrow 2$  to  $w$  do
        if  $\text{bandwidth}(c_j c_{j+1} \dots c_w 0 d_2 d_3 \dots d_{j-1}) < bw_r$  then
          |  $wrapOk \leftarrow \text{false};$ 
        end
      end
    end
    if  $\neg wrapOk$  then
      | continue;
    end
    if  $\text{cost}[q - 1, 0 c_1 c_2 \dots c_{w-1}] < \text{cost}[q - 1, 1 c_1 c_2 \dots c_{w-1}]$  then
      |  $\text{cost}[q, c_1 c_2 \dots c_w] \leftarrow c_w + \text{cost}[q - 1, 0 c_1 c_2 \dots c_{w-1}];$ 
      |  $\text{pre}[q, c_1 c_2 \dots c_w] \leftarrow 0;$ 
    else
      |  $\text{cost}[q, c_1 c_2 \dots c_w] \leftarrow c_w + \text{cost}[q - 1, 1 c_1 c_2 \dots c_{w-1}];$ 
      |  $\text{pre}[q, c_1 c_2 \dots c_w] \leftarrow 1;$ 
    end
  end
end

```

---

### 4.5.2 Proof of optimality of the algorithm for the enhanced formulation

We prove by mathematical induction that the sets  $\mathcal{A}_{k,C_k}$  generated using the algorithm previously described are optimal solutions to the sub-problem  $|A_{k,C_k}| = \text{minimum}$ ,  $A_{k,C_k}$ ,  $k \geq w - 1$  is a valid partial solution.

The base for the induction is at  $k = w - 1$  ( $A_{k,C_k}$  is not defined for  $k < w - 1$ ). For  $k = w - 1$  a single valid partial solution exists  $A_{k,C_k} = A_{w-1,START}$  (from the definition of a valid partial solution), therefore this solution is optimal.

For  $k > w - 1$  we prove that  $\mathcal{A}_{k+1,C_{k+1}}$  can only be obtained from  $\mathcal{A}_{k,C_k}$  sets using the method described in the algorithm. We demonstrate this using a proof by contradiction.

Consider a set  $\mathcal{A}_{k+1,C_{k+1}}$  which is an optimal valid partial solution. If this set could be written as  $A_{k,C_k} \cup \{s_k + 1\}$  where  $A_{k,C_k}$  is not an optimal solution, then  $\exists \mathcal{A}_{k,C_k}$  with  $|\mathcal{A}_{k,C_k}| < |A_{k,C_k}|$  (the existence of  $A_{k,C_k}$  implies the existence of a  $\mathcal{A}_{k,C_k}$ ). If  $A_{k,C_k} \cup \{s_k + 1\}$  verifies  $\mathcal{P}(A_{k,C_k} \cup \{s_k + 1\}, k - w + 2)$  then  $\mathcal{A}_{k,C_k}$  verifies  $\mathcal{P}(\mathcal{A}_{k,C_k} \cup \{s_k + 1\}, k - w + 2)$  because in the window that needs to verify the bandwidth requirement property the slots in the two sets are completely determined by the values  $c_{k-w+2} \dots c_{k+1}$ . if  $B_{k+1,C_{k+1}} = \mathcal{A}_{k,C_k} \cup \{s_k + 1\}$ ,

$$|B_{k+1,C_{k+1}}| = 1 + |\mathcal{A}_{k,C_k}| < 1 + |A_{k,C_k}| \Rightarrow |B_{k+1,C_{k+1}}| < |\mathcal{A}_{k+1,C_{k+1}}|$$

which is a contradiction, because we assumed that  $\mathcal{A}_{k+1,C_{k+1}}$  was an optimal valid partial solution.

Furthermore, using the same reasoning, if one optimal set  $\mathcal{A}_{k,C_k}^{(1)}$  leads to a optimal partial solution  $\mathcal{A}_{k+1,C_{k+1}}^{(1)}$ , and multiple  $\mathcal{A}_{k,C_k}^{(i)}$  exist, they all lead to optimal partial solutions  $\mathcal{A}_{k+1,C_{k+1}}^{(i)}$ . It is therefore sufficient to store one  $\mathcal{A}_{k,C_k}$  for each value of  $k$  and the vector  $C_k$  like our algorithm does.

Analogously we can prove that  $\mathcal{A}_{k+1,C_{k+1}}$  can only be obtained from a  $\mathcal{A}_{k,C_k}$  set.

Consider an optimal global solution  $\mathcal{A}$ . For one iteration of the top-level loop of the algorithm,  $A_{w-1,START} = \mathcal{A} \cap s_1 \dots s_{w-1}$ . During this iteration,  $\mathcal{A}$  matches the definition of  $A_{n,C_n}$  for a certain value of the vector  $C_n$  and furthermore, because it minimizes  $|\mathcal{A}|$  is also an optimal partial solution of the type  $\mathcal{A}_{n,C_n}$ . Because the algorithm iterates over all possible values of vector  $C_n$  it will also find an optimal solution  $\mathcal{A}_{n,C_n}$  with the  $C_n$  vector of  $\mathcal{A}$  the same as the  $C_n$  vector of  $\mathcal{A}_{n,C_n}$  and  $|\mathcal{A}_{n,C_n}| = |\mathcal{A}|$ . Because the verification of the bandwidth

requirement in wrap-around windows  $\mathcal{P}(A_{n,C_n}, q)$ ,  $q \in \{n - w + 2..n\}$  only depends on the  $C_n$  vector and  $A_{w-1,START}$ , since  $\mathcal{A}$  is a valid solution  $\mathcal{A}_{n,C_n}$  is also a valid solution and thus a global optimum.

## 4.6 Algorithm complexity

Consider the following notation for the calculation of complexity:

- $n$  is the number of slots in the slot table
- $m$  is the number of free slots  $m \leq n$ , worst case  $t = n$
- $l$  is the maximum distance between slots
- $r$  is the maximum number of consecutive slots after which the header needs to be repeated
- $w$  is the size of the window for the enhanced algorithm

**The original algorithm** performs a linear search for the first available slot, in complexity  $O(n)$ . This is followed by a search for free slots at intervals of length  $l$ , which imply testing of at most  $n - 1$  slots, which is again  $O(n)$  complexity. Another linear  $O(n)$  complexity search is sufficient to determine the additional slots necessary to satisfy the bandwidth constraint, therefore the total complexity of the original algorithm is  $O(n)$ . No special memory structures are used except for the list of slots which has a memory complexity of  $O(n)$ .

**The dynamic programming algorithm** is run  $t$  times for each rotation of the slot table in which the second slot is available (as explained in Section 4.4). Each of these runs involves building a table of partial solutions ( $A_{k,i,j}$ ) of size  $n^2 * r$ . When slot  $k$  is available (which happens for  $m$  of the slots)  $A_{k,1,j}$  is computed based on  $l * r$  other values, and  $A_{k,i,j}$  with  $i \neq 0$  is computed based on one other value. The complexity to compute the table is  $O(m^2 * l * r)$  and the total algorithm complexity  $O(m^3 * l * r)$  which is in the worst case  $O(n^3 * l * r)$ . The memory complexity is dominated by the size of the tables mentioned and is  $O(n^2 * r)$ .

**An exhaustive search** which we used to verify our solutions can be performed by testing all  $2^m$  combinations of the free slots, for each combination having to determine whether it satisfies the latency and bandwidth requirements which is typically done in complexity  $O(n)$ . This can be easily achieved with



$O(n * 2^m)$  complexity, but it can also be optimized making the latency and bandwidth computation incremental (readjusting bandwidth and latency values with the decision of including or excluding each independent slot), resulting in a complexity of  $O(2^m)$ . In the worst case, the complexity is  $O(2^n)$ , but it should be noticed that the exhaustive search is practical even for large slot table sizes if the occupancy ratio is high. For example, on a completely empty table of size 32, the exhaustive search would have to iterate through approximately 4 billion solutions, however, if the table is 50% full, it would only have to iterate through 65536 solutions. The memory complexity is  $O(n)$ .

**The enhanced algorithm** iterates in its outer loop through up to  $2^{w-2}$  conditions for the initial slots. For each of these a table of size  $(n - w) * 2^w$  needs to be constructed, each element in the table being computed based on two other elements. Removing the constant factors, the complexity is  $O((n - w) * 4^w)$ . The number of actual valid states is also affected by the  $m/n$  ratio, but in a relatively complex manner, therefore we only provide a worst-case complexity measure. The advantage compared with the exhaustive search is that the complexity does not increase exponentially with the size of the slot table. The memory complexity is  $O((n - w)2^w)$ .

## 4.7 Experimental results

In this section we compare our proposed algorithm with the original greedy algorithm. For performing the comparison we use slot tables with a certain percentage of slots marked as occupied (background traffic) and we request both algorithms to provide an allocation for every feasible combination of latency and bandwidth. For slot table sizes up to 16 we use as background traffic all the combinations of occupied and unoccupied slots (65536 combinations in total). For the larger slot table sizes, as this method becomes unfeasible we produce 1000 samples at each background traffic ratio (the number of occupied slots divided by the total number of slots).

We additionally verify the dynamic programming against an exhaustive search algorithm for slot table sizes up to 24. It is generally not possible to run the exhaustive search on the larger tests because of its exponential running time.

Our proposed algorithm produces in many cases a better solution than the greedy algorithm. We represent the improvement for slot table sizes of 16, 24, 32, 40 in Figures 4.18-4.21. The improvement can take two forms: in some cases, the required bandwidth and latency can be delivered using fewer slots. In the graphs, this is called “slot improvement.” When not producing slot

improvement, there is still a chance that the dynamic programming algorithm delivers more bandwidth than is required due to the granularity of slots and that bandwidth is higher than the one of the greedy algorithm. In the graphs this is called “bandwidth improvement.”

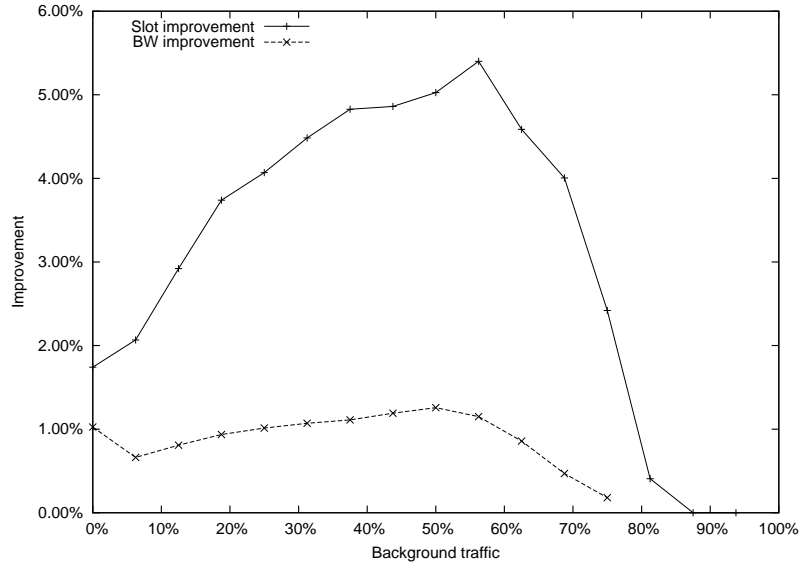


Figure 4.18: Improvement in slot utilization vs background utilization, 16 slots.

The plots represent an average over all requested combinations of latency and bandwidth versus the amount of background traffic.

Obviously little gain can be obtained when the slot table is essentially empty or when it is completely full. Some improvement exists though for a completely empty table because the original algorithm does not properly take into account bandwidth gain at wrap-around. The largest gains are in the middle section of the interval, corresponding to average background utilization. In practice, when allocating a usecase, the first channel allocation will be performed on an empty slot table, thus zero background utilization while further allocations will encounter some background utilization. If high utilization is never reached it means the network was probably over-provisioned; if it is reached too early there is a good chance the allocation will not succeed at all.

Focusing the region of the data with average background utilization (25% and 50%), we analyze the improvement with respect to the requested latency and

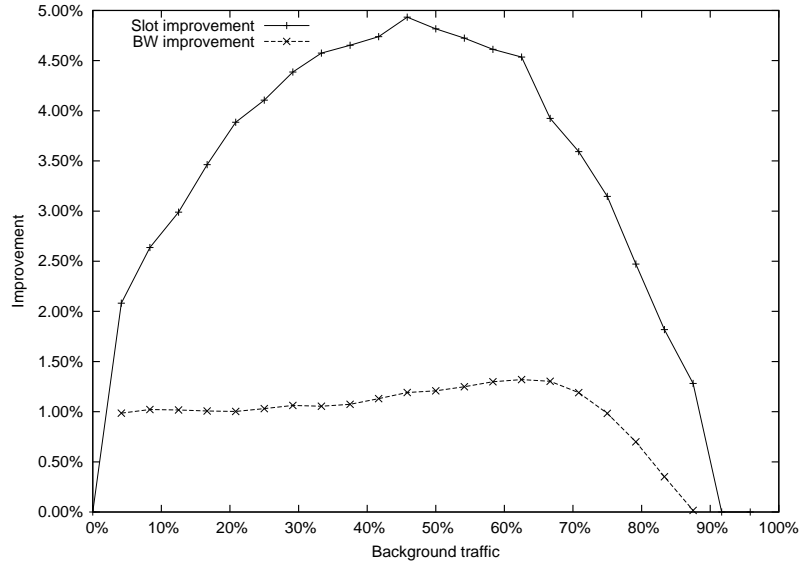


Figure 4.19: Improvement in slot utilization vs background utilization, 24 slots.

bandwidth. We can see that for very tight latency constraints there is hardly any gain since there is very little flexibility in choosing the needed slots. This effect is more pronounced with a small slot table. Also very little gain is made when the required bandwidth is very low or very high. For the first case it means that for latency-only constraints the initial greedy algorithm is performing very well, for the latter obviously when the entire bandwidth needs to be allocated only one solution exists and that consists of allocating all the available slots.

There is also some periodicity visible in the graphs along the required latency axis. This is influenced by the modulo of the slot table size (minus the number of additional slots due to the bandwidth requirement) to the required latency. The optimal algorithm works better when this modulo is closer to zero which means there is little slack in satisfying the requirement.

It is also noticeable that for the lower bandwidths the graph curve is independent of bandwidth. This is because even with just the latency constraint a certain bandwidth is provided anyway.

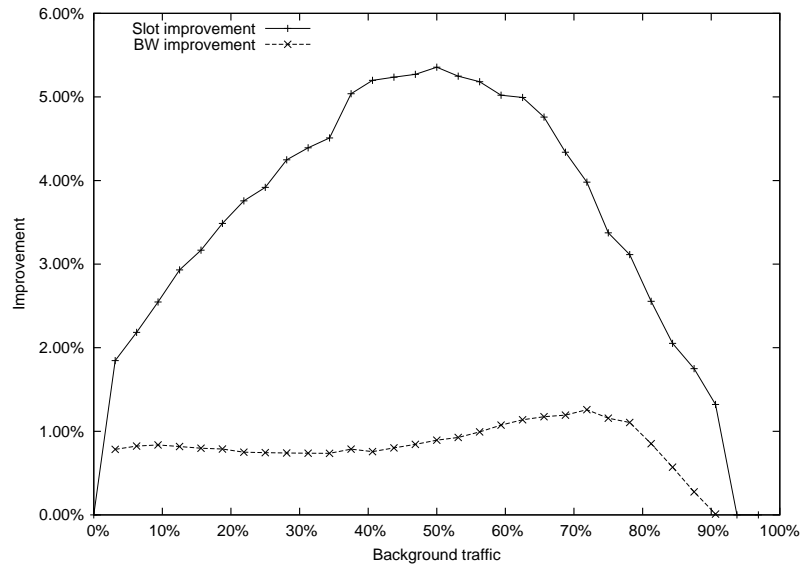


Figure 4.20: Improvement in slot utilization vs background utilization, 32 slots.

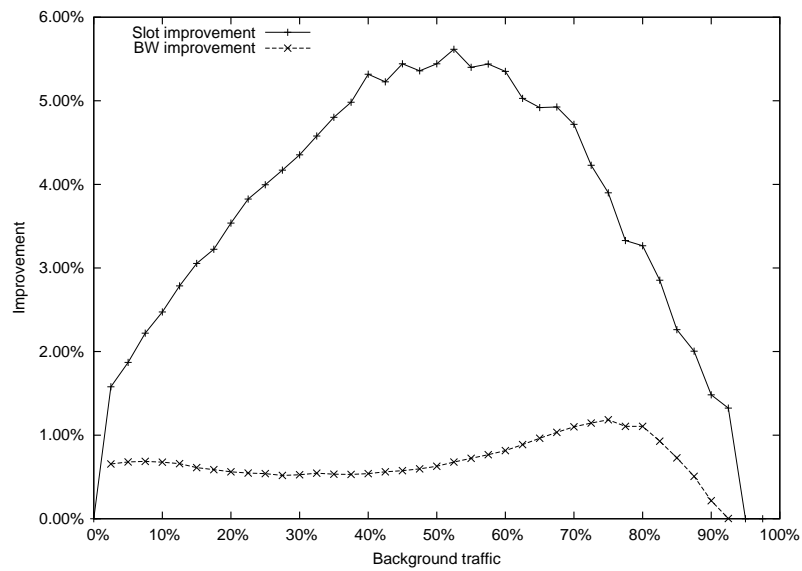


Figure 4.21: Improvement in slot utilization vs background utilization, 40 slots.

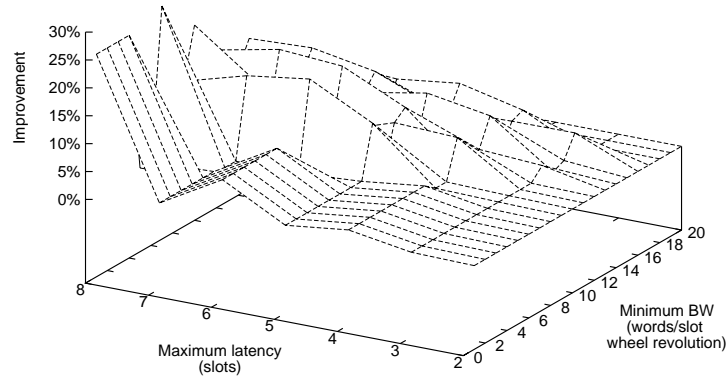


Figure 4.22: Improvement in slot utilization vs requested latency and bandwidth, background traffic 8/16 slots.

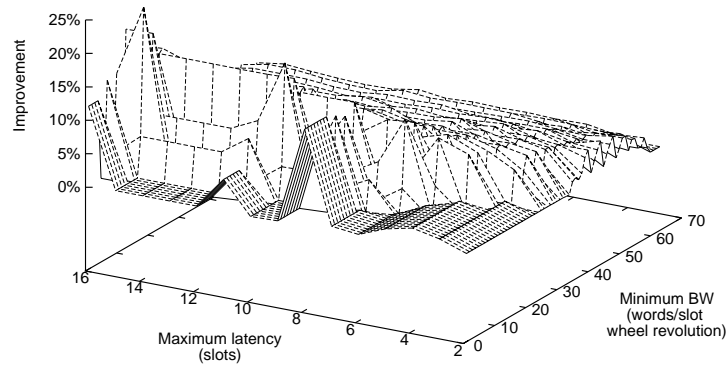


Figure 4.23: Improvement in slot utilization vs requested latency and bandwidth, background traffic 8/32 slots.

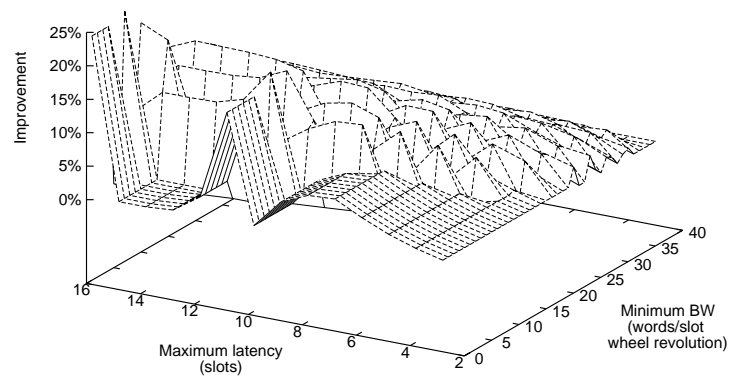


Figure 4.24: Improvement in slot utilization vs requested latency and bandwidth, background traffic 16/32 slots.

### Performance results for the enhanced formulation

The enhanced formulation allows us to provide latency guarantees for longer messages without over-constraining the solution. Consider the following example:

We wish to provide a bound on the latency of a 4-word message to a maximum of 12 slots. With the original algorithm, the only way to achieve this is to set the one-slot (2 words) latency to half the value of the bound, as represented in Figure 4.25.

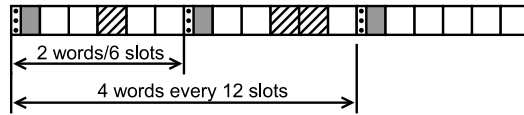


Figure 4.25: A bound on latency of 1 selected slot every 6 slots guarantees that 4 words can be delivered every 12 slots.

This method of reducing the problem to a more simple formulation however will fail to find solutions in some instances. For example, with the same requirements used above, if 6 consecutive slots are occupied and cannot be used a solution with latency 6 does not exist but it is possible to find a solution that provides 4 words within any window of 12 slots (Figure 4.26).

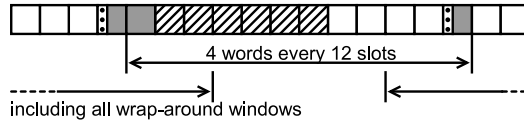


Figure 4.26: In some situations it is not possible to allocate 2 words every 6 slots, but it is possible to allocate 4 words every 12 slots.

The enhanced algorithm in Section 4.5 can provide an optimal slot selection whenever one exists directly for the words/window problem formulation. The algorithm nevertheless has some practical limitations in that its complexity is exponential in the size of the window. While this is a limitation of the algorithm, the same method that is used for extending the scope of the basic algorithm to messages of longer size can be applied here at a loss in terms of solution quality (although we did not investigate this here.) In our experiments we used a window size of 12 and a slot table size of 32, which allow computation times in the order of 1 second.

We perform a series of tests to determine the success rate of the enhanced (optimal) algorithm against the method of reducing the problem to the original formulation. We plot the success rate against the number of occupied slots (background traffic). For each value of the background traffic we have generated 1000 random sets of occupied slots. The results are shown in Figures 4.27-4.30.

We perform the tests for words/window requirements that have a direct equivalent in the original formulation: 4 words/12 slots is equivalent to 1 slot out of 6 (due to header overhead), 6 words/12 slots to 1 slot out of 4, 8 words/12 slots to 1 slot out of 3 and 12 words/12 slots to 1 slot out of 2. Using values that are not divisors of 12 would only exaggerate the disadvantages of the original algorithm which would need to round the value to the nearest integer latency.

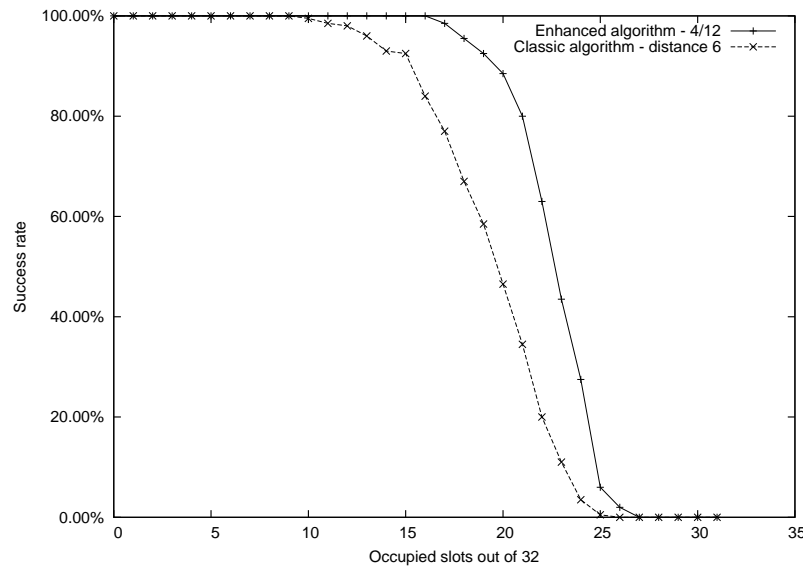


Figure 4.27: Probability of successful allocation when 4 words are required in a window of 12 slots.

With an empty slot table obviously both algorithms have no trouble finding a solution but as the number of available slots decreases finding a solution becomes more difficult. The optimal algorithm is of course more efficient than the original algorithm which may miss some solutions. For low bandwidth (words/window) requirements the optimal algorithm can provide a solution under roughly 10% higher load. Under higher bandwidth requirements, the



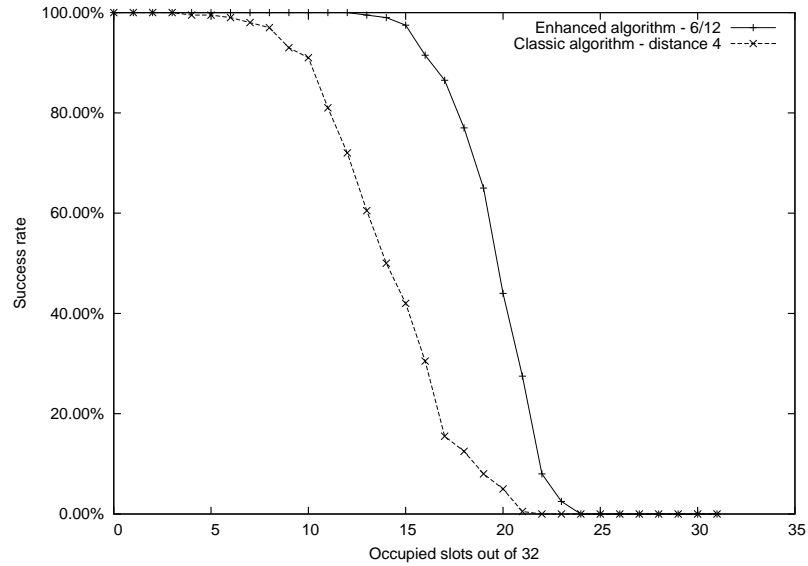


Figure 4.28: Probability of successful allocation when 6 words are required in a window of 12 slots.

gap increases to 30% .

In addition to being able to provide the requested bandwidth in more situations, the enhanced algorithm can provide a more efficient solution, in that it requires fewer slots to satisfy the requirement. The original algorithm always produces sparse slot allocations which have a disadvantage in terms of header overhead. In contrast, the enhanced algorithm always produces an optimal sequence of slots, taking into account the header overhead.

The average number of slots used in the allocations that were successful using both methods is represented in Figure 4.31. The average is performed across all loads. It is apparent that for low bandwidth requirements the difference is minimal however the situation changes for the high bandwidth requirements which have solutions with more densely packet slot tables.

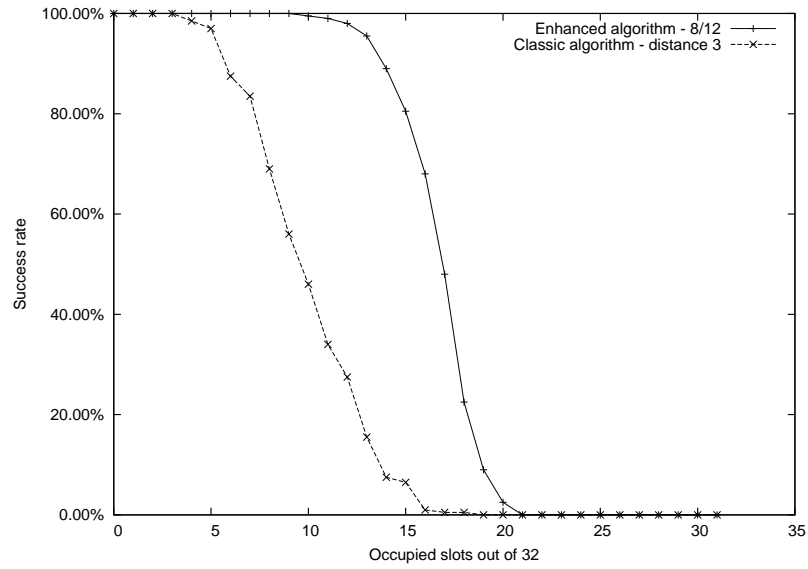


Figure 4.29: Probability of successful allocation when 8 words are required in a window of 12 slots.

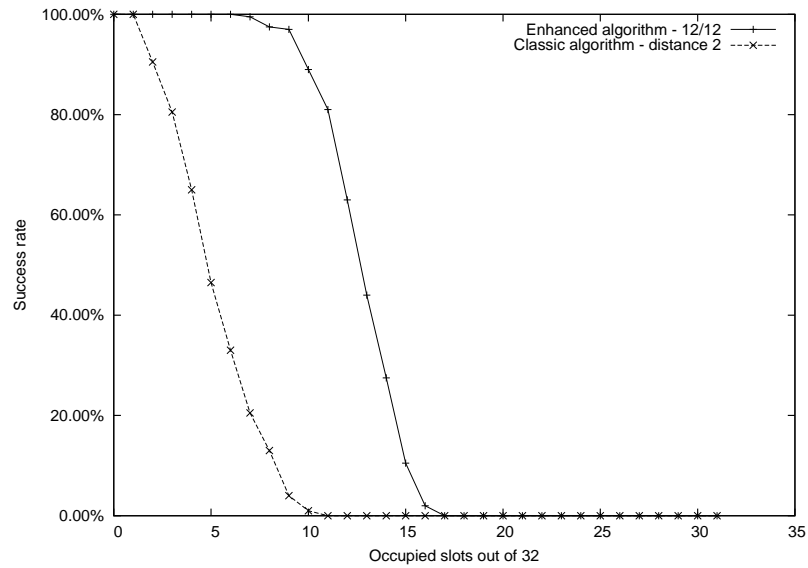


Figure 4.30: Probability of successful allocation when 12 words are required in a window of 12 slots.

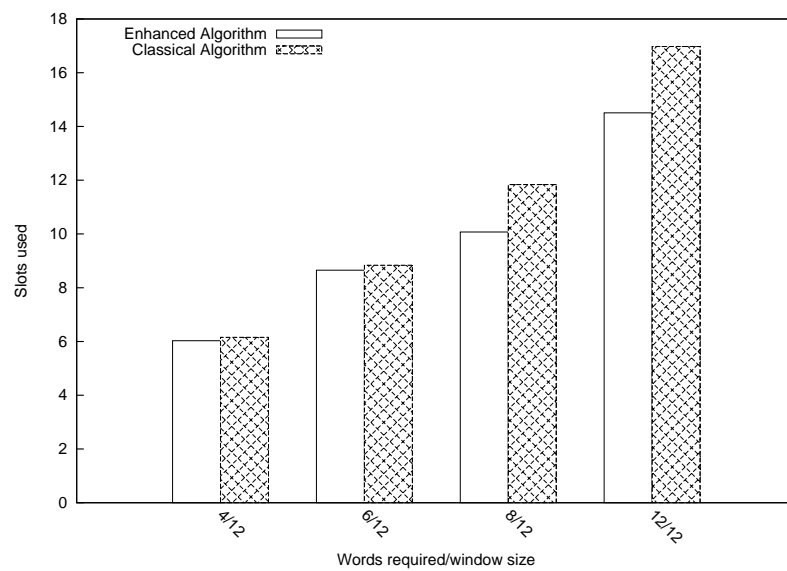


Figure 4.31: Average number of slots used by each algorithm.

## 4.8 Related Work

The problem of routing in NoCs has been widely studied [ACPP06, DA93, HM03, HM04]. Our algorithm is related to the routing problem but instead of spatial allocation it deals with allocation along the time axis. It is expected to be used in conjunction with an algorithm for route selection in a TDM network for latency optimization.

Our technique applies in general to NoCs using TDM, like Nostrum [MNTJ04b], aSoC [LST00] and the TDM network in [WZLY08]. Although some implementation details like the computation of header overhead are specific to the *Æthereal* implementation we believe that similar problems arising in other implementations could be solved by the same algorithm.

A related algorithm achieving both path and slot allocation is presented in [TLZ09]. Different slot assignments schemes are evaluated, namely distributed, random and consecutive. However the method does not provide latency bounds and does not target complex header overhead scheme found in *Æthereal*.

The algorithm previously used to solve this problem is presented in [Han09], we use this to compare the performance of our algorithm. An algorithm improvement involving path rip-up and reallocation on top of the normal *Æthereal* algorithms is presented in [SBG<sup>+</sup>08]. While we do not directly investigate this technique we consider our algorithm to be compatible with it.

An analysis of communication latency after the slots have been selected is provided in [NHCG10], while [HWM<sup>+</sup>09] details the relation between communication performance and the overall application behavior.

In a similar network implementation with more relaxed constraints on slot alignment, a graph coloring algorithm is proposed by [LZT04] to solve a slot allocation problem. Our algorithm is to a large extent motivated by the restriction that in *Æthereal* slots need to be forwarded on the next link without delay. However other TDM networks may choose to do this even when it is not mandatory, in order to improve latency.

## 4.9 Conclusions and future directions of research

In this chapter we have analyzed the effect of slot selection on communication and application performance and we have proposed optimal algorithms for the

problem of slot selection. Because our algorithms are optimal they can be used as an absolute measure for the performance of previously proposed algorithms.

Our algorithms offer gains in terms of allocation efficiency and thus network performance at negligible cost (the only cost is an increased computational load at design time.) If allocation is performed at run time, the previous algorithm may still be preferable.

The slot selection algorithm is currently only targeting the single-path search. In the iterative multi-path search it can be used without modifications to provide optimal slot selection with regard to bandwidth. If any of the multi-path algorithms is used, but restricted to only use same-length paths, it is possible to use the same algorithm with only one modification in Equation 4.1 to take into account the fact that consecutive slots on different paths need to repeat the header. It might however be possible to integrate latency-based decisions deeper into the path search algorithm, leading to further improvements.



## Chapter 5

### Online allocation

As the computation of the slot allocation in networks using the contention-free routing model is fairly complex, it is customary to perform this computation at design time. Connections can be set up dynamically at run time as needed based on pre-computed tables. All usage scenarios consisting need to be taken into account at design time when computing the allocation. This is suitable for streaming applications, applications that have well-defined communication behavior or applications that require guarantees for the communication performance.

There are however classes of problems that do not have such a predefined behavior, where communication is dynamic or dependent on the input data, or do not require bandwidth and latency guarantees. These applications may receive then network resources on demand, in the form of best-effort services. However, supporting best-effort services in the form of a packet-switching network implementation was shown to be very expensive [GH10], and packet-switching in general is known to be less power efficient than circuit-switching [KSWJ06, BWM<sup>+</sup>09].

We propose the implementation of Best-Effort like services over the existing Guaranteed Throughput Circuit Switching network by performing the allocation of channels online, at run time. Channels allocated in this way do not have the advantage of receiving guarantees from design time but they can make use of whatever network resources happen to be available when they are requested at run time. The allocation may also fail and the connection will have to wait until more network resources become available.

In this chapter we present algorithms and a system setup for online channel allocation. We demonstrate our method in an FPGA prototype based on

the *Æthereal* network. The allocation algorithm is implemented in software written in the C language. We measure its performance when run in FPGA on a Microblaze embedded processor. We also demonstrate a hardware accelerator that implements some of the more time-consuming computations and we offer a blueprint for a fully hardware accelerated allocator.

This chapter is organized as follows: Section 5.1 presents general considerations regarding channel allocation in Circuit Switching networks. The data structures required by the allocation algorithm are presented in Section 5.2. The pathfinding algorithm is presented in Section 5.3. Various options for bandwidth computation are presented in Section 5.4. A blueprint for a fully hardware accelerated allocator is presented in Section 5.5. Experimental results detailing the speed of the allocator are presented in Section 5.6. Related works are presented in Section 5.7. Section 5.8 presents our conclusions.

## 5.1 Channel allocation in Circuit switching networks

In a typical circuit switching network resources are allocated in a distributed manner, using probes or setup packets. Each router has knowledge of the state of the links it is connected to. A set-up packet travels through the network incrementally, allocating the required bandwidth on each link it visits (Figure 5.1). The disadvantage of this method is that it is not possible to know beforehand if the entire path to the destination is available, the path currently being set up blocks the already allocated links until failure is discovered and a mechanism to tear down the path must be provided. Measures also need to be taken to avoid deadlock situations, which may result in routing restrictions.

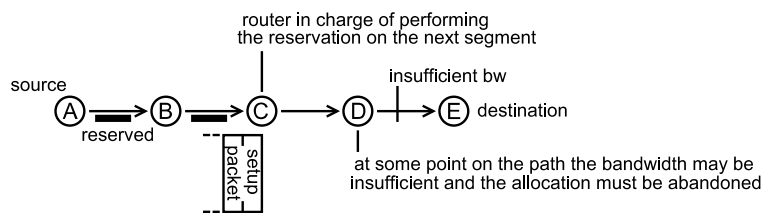


Figure 5.1: Path reservation in a classic Circuit-Switching network.

By comparison, in our solution, the entire allocation is first simulated in a processing node which has knowledge of the entire state of the network and only then the path is reserved (hence clients must contact the central node to request a connection). When a path is not found by the first attempt, the



algorithm searches alternatives using a backtracking method until a path is found or all options are exhausted. The algorithm running on the central node keeps track of the network state, including all connections allocated using the design-time allocation method as well as the ones allocated using online allocation.

## 5.2 Data structures

A routing algorithm requires knowledge of the underlying network topology and available network resources. In this section we describe the data structures used to represent this information. In all situations we opted for simple data structures with minimal memory footprint and we selected data types with the minimum bit-width that allows storing the necessary values.

The developed algorithms can be applied to any network topology by only changing the static arrays that describe the topology. Although topology specific optimizations would be possible we preferred using topology agnostic algorithms at some cost in execution speed. We have chosen this option because it does not affect actual algorithm complexity, but it only adds a small overhead to computation due to unnecessary look-up operations (Figure 5.2).

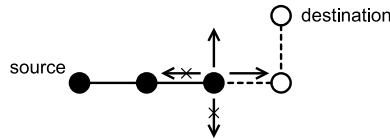


Figure 5.2: All outgoing links of a node are enumerated by the path finding algorithm but the ones leading away from the solution are immediately discarded.

We provide complete integration between the on-line allocation mode and the classic design-time allocation flow used by *Æthereal*. We use the same files and file-formats at design time to generate the C code of the online allocator and a compatible scheme of identifiers for the network interfaces and ports.

Both links and network nodes are identified by numeric IDs. Network nodes are sorted by type, first network interfaces then routers. Links are considered to be unidirectional (bidirectional links are stored as two separate unidirectional links). Links are sorted by their source node. One table (*dest* in Figure 5.3) stores the destination of each link. Another table (*start*) stores the first link

in the table of links that belongs to each IP. The last entry in this table marks the end of the links table. This corresponds to the Compressed Row Format [Pis84] for sparse matrix representation.

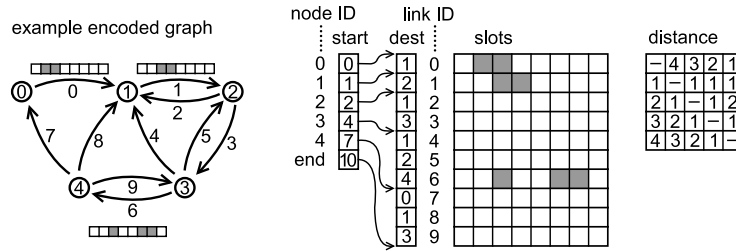


Figure 5.3: An example network and tables used to describe its topology, the algorithms accept an arbitrary directed graph, although in practice unidirectional links appear in pairs with opposite directions.

The same numeric link ID used as an index in the *dest* table is also used as index in the *slots* table (Figure 5.3). Depending on the usage scenario, we can have multiple slot tables, for example for offline (guaranteed) allocated slots and for best-effort channels.

We also store a symbolic list of identifiers for debugging purposes. In our FPGA prototype we can obtain debug information about the state of the network and the allocation process, directly over a serial link connected to a PC.

### 5.3 The path finding algorithm

The path finding algorithm we use is essentially the same as the Single Path allocation described in Chapter 3 with several modifications meant to improve performance:

- the recursive backtracking implementation has been replaced by a finite-state-machine like implementation which does not make use of recursive calls and thus avoids function calling overhead.
- for now, latency restrictions are ignored.
- the first solution found is accepted, without looking for better solutions.

- the final stage of the algorithm, the slot selection is performed using a simple greedy algorithm which selects the first available slots instead of the more complex algorithms described in Chapter 4.

A formal description of the algorithm is given in Algorithm 5.3.1. A stack is simulated by the *level* variable. The algorithm makes use of the following tables: the list of link destinations  $dest[linkId]$ ; a table indicating the index of the first link departing from a node  $firstLink[node]$  and the last link  $lastLink[node]$  (in fact the two are implemented by a single table,  $start[] = firstLink[]$ , because in the list of links nodes receive consecutive ranges so  $lastLink[n] = firstLink[n+1]-1$ ; the two separate names were kept for clarity); an array of distances from each node to *destination*,  $dist[nodeId]$ ; a table of available slots on each link  $slots[linkId]$ ;

Exiting the search once a solution has been found or the search is to be abandoned is also less costly using non-recursive approach. Overall we found the non-recursive implementation to perform better than the recursive one.

## 5.4 Computation of the available bandwidth

Within the path finding, the most time-consuming operation is checking whether enough words of data can be delivered by the available slots. This is complicated by the *Æthereal* header usage scheme (Figure 5.4).

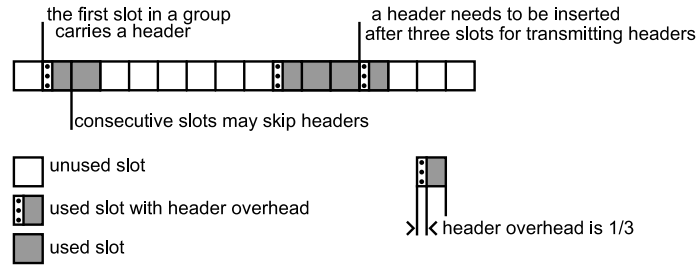


Figure 5.4: Header overhead in *Æthereal*.

In the following sections we propose and evaluate several methods of computing the bandwidth delivered by a set of slots. The first method performs an exact computation using a loop executed in software, the second method is based on look-up tables and when headers are involved provides only an approximation.

**Algorithm 5.3.1:** Non-recursive exhaustive pathfinding**input** : *source* and *destination* nodes*requiredBw* the required Bandwidth**output:** Path from *source* to *destination* which satisfies the bandwidth constraints  
will be found stored in *solLink*[1..*level*]

```

1  level  $\leftarrow$  1;
2  crtNode  $\leftarrow$  source;
3  crtSlots  $\leftarrow$  S;
4  crtLink  $\leftarrow$  start[crtNode];
5  while level > 0 do
6      if crtNode = destination then
7          process solution;
8          break;
9      end
10     nxtSlots  $\leftarrow$  shift(crtSlots) and not slots[crtLink];
11     crtDest  $\leftarrow$  dest[crtLink];
12     slotsOK  $\leftarrow$  bw(nxtSlots)  $\geq$  requiredBw;
13     if dist[crtDest]  $\leq$  allowedDistance - level  $\wedge$  slotsOK then
14         solution[level]  $\leftarrow$  crtNode;
15         solLink[level]  $\leftarrow$  crtLink;
16         avSlots[level]  $\leftarrow$  crtSlots;
17         level  $\leftarrow$  level + 1;
18         crtSlots  $\leftarrow$  nxtSlots;
19         crtNode  $\leftarrow$  crtDest;
20         crtLink  $\leftarrow$  firstLink[crtDest];
21         continue;
22     end
23     crtLink  $\leftarrow$  crtLink + 1;
24     if crtLink  $\geq$  lastLink[crtNode] then
25         level  $\leftarrow$  level - 1;
26         crtNode  $\leftarrow$  solution[level];
27         crtLink  $\leftarrow$  solLink[level] + 1;
28         crtSlots  $\leftarrow$  avSlots[level];
29     end
30 end

```

**5.4.1 Exact bandwidth computation in a software loop**

The exact computation requires sequentially testing the slot table twice. On the first pass we determine the number of usable slots at the end of the slot table. We need this in order to determine the position of headers in the first

**Algorithm 5.4.1:** Exact computation of the available bandwidth

---

**input** :  $n$  size of slot table  
 $s[0..n-1]$  list of available slots,  
a value of 1 represents an available slot

**output**: Bandwidth in terms of number of data words per slot table revolution *result*

```

result  $\leftarrow$  0;
count  $\leftarrow$  0;
for  $i \leftarrow n-1$  to 0 do
    if  $s[i] \neq 1$  then
        | break;
    end
    count  $\leftarrow$  count + 1;
end
if count =  $n$  then
    // WHEN ALL SLOTS ARE AVAILABLE WE CONSIDER THE FIRST ONE TO HAVE
    A HEADER
    count  $\leftarrow$  0;
end
count  $\leftarrow$  count mod 3;
for  $i \leftarrow 0$  to  $n-1$  do
    if  $s[i] = 1$  then
        | result  $\leftarrow$  result + slotSize;
        | if count = 0 then
        | | result  $\leftarrow$  result - hdrOverhead;
        | end
        | count  $\leftarrow$  (count + 1) mod 3;
    else
        | count  $\leftarrow$  0;
    end
end

```

---

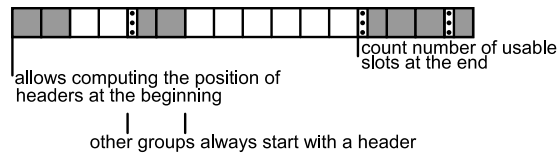


Figure 5.5: Exact computation of available words.

group of slots (Figure 5.5). The method is described in Algorithm 5.4.1.

On the second pass, the groups of consecutive available slots are identified and

the deliverable bandwidth is calculated. Because decision logic is necessary in both passes the program is slow.

### 5.4.2 Bandwidth approximation using lookup tables

If the size of the slot table was very small it would be feasible to use a look-up table to immediately determine the number of available words. The size of the lookup table increases exponentially with the number of slots  $2^n$ . This is a much less computationally intensive solution as it can provide the result with a single memory access, but its disadvantage is a higher memory requirement. For a realistic size of the slot table of 16 slots, the cost of the table would be prohibitive.

A less memory intensive solution would be to split the slot table into groups of slots of reasonably small size and perform a lookup operation for each group. The difficulty is that in the networks that employ headers (Æthereal) we cannot know for certain which of the slots have headers and which do not.

It is possible to assume conservatively that the all slots have headers. The allocation algorithm would produce correct results, but it may fail to find some solutions which were feasible. We have computed that as an average over all combinations of used and unused slots in a table of size 16, assuming that all slots have a header underestimates the bandwidth provided by one slot table by 15.5%. A more accurate solution is to assume that the first slot in a group always has a header (Figure 5.6). This never causes overestimating the total bandwidth provided by the set of slots and the difference to the real value is only 3.3% on average when the size of a group is 6. A group size of 6 requires a lookup table of only 64 entries.

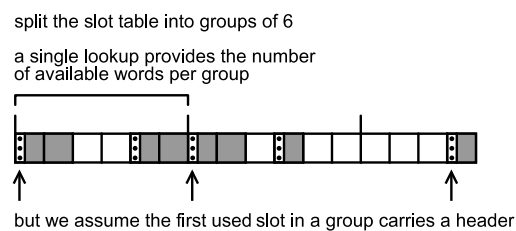


Figure 5.6: Approximate computation of the available words.

In networks which do not have a header overhead, like the one we propose in Chapter 6 the group-based lookup produces the exact result.

### 5.4.3 A hardware accelerator for bandwidth computation

Most of the computations performed by Algorithm 5.4.1 are performed with very small operands, for example *count* can be represented with only 2 bits, while for the variable *result* up to 6 bits may be necessary. These computations can be efficiently performed in a dedicated hardware unit having very low cost.

We propose and implement a module which performs the bandwidth computation in hardware. The module is described in Figure 5.7. It follows the same logic as the software implementation, but it replaces most of the computations with look-up operations.

For example the following strategy is used to compute the value of *count* at the end of the first loop in Algorithm 5.4.1. Slots are organized in groups of three. A block called *phase* indicates how many slots are available at the end of a group and whether or not all slots are available. When all slots are available the value of *count* modulo 3 is taken from the previous block, if that block also happens to have all slots available from the one before it and so on, using a ripple type of logic.

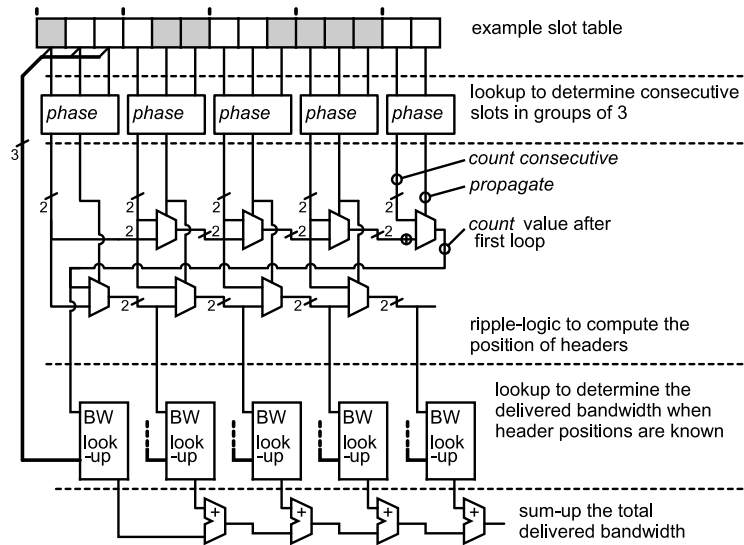


Figure 5.7: Hardware module to directly compute the exact number of available bandwidth.

The bandwidth offered by each group of three slots is also computed using look-up tables, but unlike the approximate computation previously described

we also take into account the *count* of slots module 3 before each block, thus performing an exact computation. At the end, the looked up values are added together.

FPGA synthesis in the Xilinx Virtex-6 technology of a module performing the computation on a table of 16 slots indicates a working frequency of 288 MHz and an area utilization of 108 LUTs and 59 registers. The design is interfaced with the Microblaze processor using the low-latency FSL links. The design is pipelined, with 2 pipeline stages for the actual computation in addition the ones necessary to interface it with the microprocessor.

## 5.5 A blueprint for a hardware allocator

In this section we provide a blueprint for a circuit implementing the entire allocation algorithm described in Algorithm 5.3.1. While we do not offer a synthesizable hardware description we model this circuit in enough detail to allow us to estimate its speed. Being a circuit with a specialized function its cost will be lower than the cost of a general purpose processor and its program memory running the same algorithm in software.

We start with the following observations:

1. The state of the program is stored in 4 variables *level*, *crtNode*, *crtSlots*, *crtLink* and a stack.
2. The algorithm has one loop in which the entire processing is performed.
3. Inside the loop, the program may branch on 4 possible paths, of which one is only take once, when a solution is found.

The first way in which the hardware accelerated implementation can improve on the performance of its software counterpart is by executing several operations in parallel.

In Figure 5.8 we represent in vertical columns the operations that can be executed in parallel grouped by the branch they belong to, and from left to right the scheduling of these operations based on their dependencies on previous operations.

There are several operations that can be parallelized:

1. arithmetic operations: increment and decrement and comparison are all



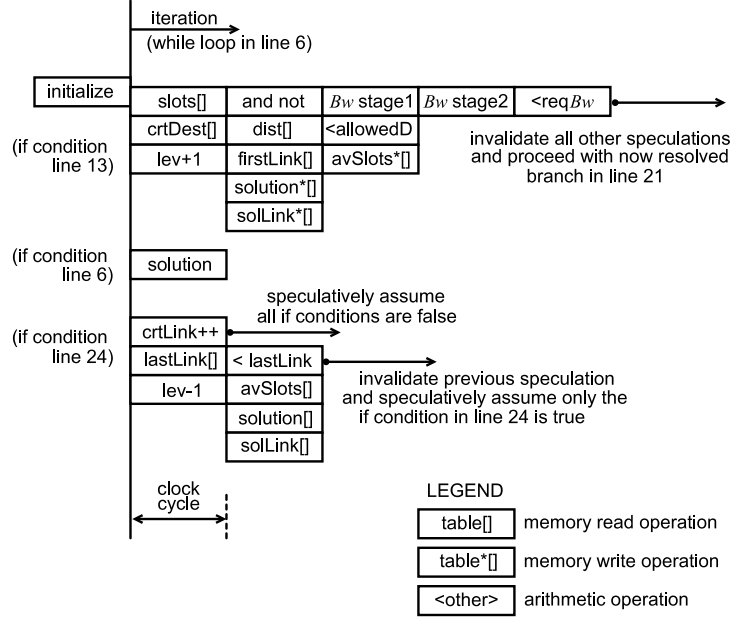


Figure 5.8: Operations in the algorithm that can be executed in parallel; code references point to Algorithm 5.3.1.

performed on low bit-width operands, therefore having a dedicated unit for each instruction would not represent a large cost

2. we use of course for the bandwidth computation the accelerated module presented in section 5.4 which is pipelined and has a throughput of one computation per cycle
3. arrays are independent and we assume they can be accessed in parallel; no array is written in more than one place and we assume the stack is implemented as a dual-ported memory which allows simultaneous read and write access.

To further improve performance we make use of speculation. Speculation is in general expensive, if it is used inside microprocessors, but in our case it has almost no cost. We observe that the two longer branches rewrite the for variable states anyway, so we could speculatively proceed with the shorter branch and, if it turns out the decision was wrong, override the state variables with the ones from the longer paths. The stack is only updated by the longest

path and it updates a stack element that is above the ones employed by the other paths.

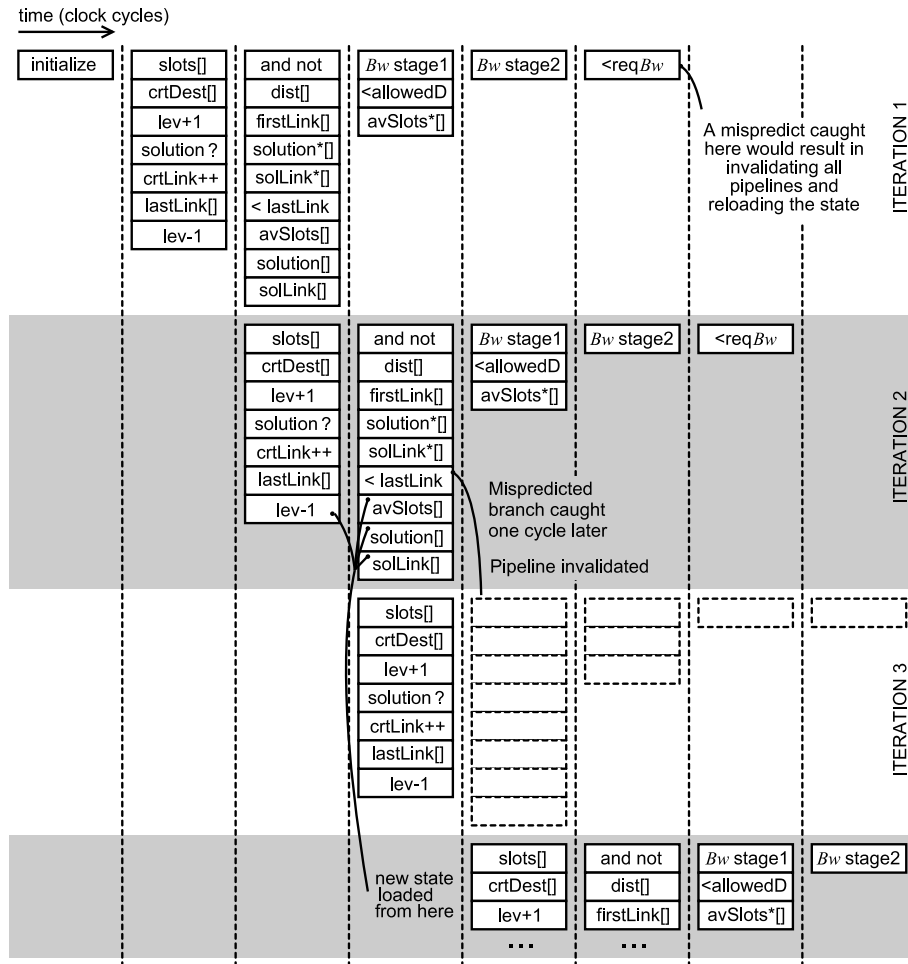


Figure 5.9: Operations in the algorithm that can be speculatively executed.

Figure 5.9 presents the operations that can be executed in parallel if speculation is used. The number of instructions executing in parallel increases to roughly 20, although some of them will have to be discarded because of wrong speculations. It can be observed that none of the operations appears twice within the same column.

The most convenient way to implement a circuit to achieve this parallelism is

by having a separate pipeline for each execution path, as illustrated in Figure 5.10. Each level of speculation will occupy a different stage in each of the pipelines.

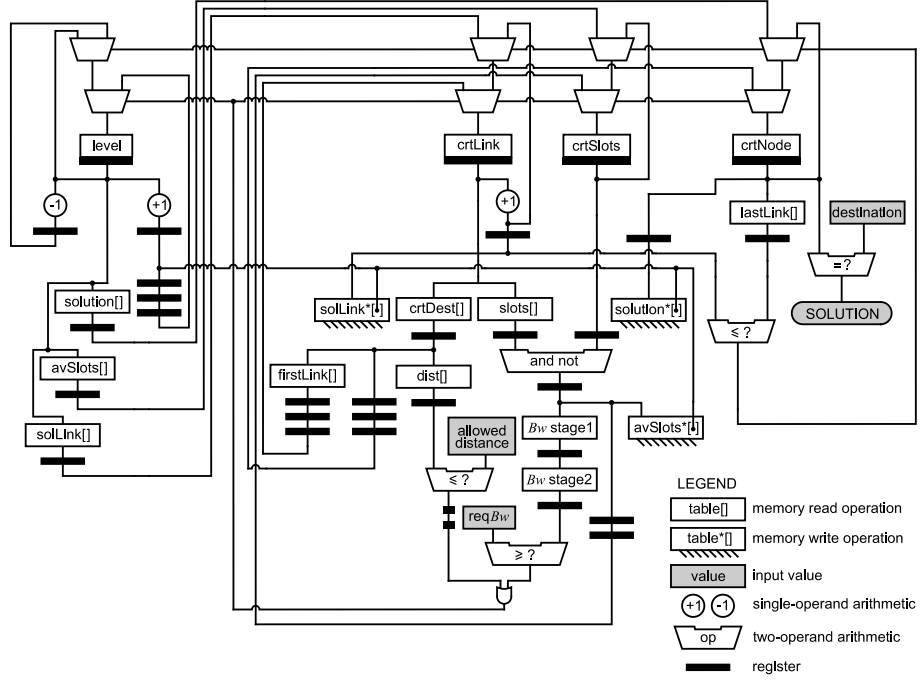


Figure 5.10: Hardware module for accelerated path computation.

The decision logic at the end of each execution path controls how the state variables are to be updated. We did not represent here the pipeline stage invalidation logic nor the initialization logic. The performance of the hardware allocator will be estimated in Section 5.6.2 by running traces of the software algorithm execution on a model of the pipeline.

## 5.6 Experimental results

In this section we evaluate the performance and memory requirements of our online allocation algorithms. We evaluate both the software-only and the software combined with the hardware computation of bandwidth proposed in Section 5.4.3 by executing the program on an embedded Microblaze processor in an FPGA prototype. The hardware accelerator for bandwidth computation

was connected to the Microblaze processor through an FSL link. For the high-level model of the fully hardware accelerated algorithm we perform a performance estimation by simulating the delays of the internal pipeline on a trace obtained from the software implementation.

We have evaluate the performance of the allocation algorithm on mesh network topologies of size 4x4, 6x6 and 8x8. The number of slots we use in the experiments is 16, the maximum number of requested words per slot table revolution thus being 40. When the bandwidth computation is performed in hardware, the speed of the algorithm does not depend on the number of slots, but the number of slots is limited to 32 because this is the width of the FSL link.

Our performance measure consists in the number of cycles the algorithm needs before it can either find a solution or determine that an allocation is not possible. We perform exhaustive search of minimal paths as described in Algorithm 5.3.1. It is possible to perform a search of longer paths by increasing the *allowedDistance* variable but that may lead to an unacceptable increase in the running time. It would also be possible to bound the computation time by requiring the algorithm to give up after a certain number of attempted paths. This can be achieved by forcing an exit out of the loop 5-30 in the same algorithm.

The main factor affecting the duration of path computation is the distance between the source and destination. When the network load is zero or close to zero it is expected that the first path found gives a successful allocation. The algorithm running time will then increase linearly with the path length. This behavior is confirmed by the experimental results in Figure 5.11.

One interesting fact is that the running time levels off or even decreases for the larger path lengths. This is because the only nodes with such a high distance are in the corners of the mesh. The path finding algorithm, not encountering any obstructions chooses a path along the edge of the network where routers have a lower number of ports. Since the algorithm enumerates all ports regardless of whether they are useful or not (Figure 5.2), fewer router ports means a reduced running time.

It is expected that under network load, the running time of the algorithm would increase. We generate network load (named here background traffic) by allocating random connections until a certain average load is achieved. The average behavior of the three methods with 10% background load is presented in Figure 5.12. This represents an average over all requested bandwidths.

It can be seen that the running time has significantly increased and furthermore

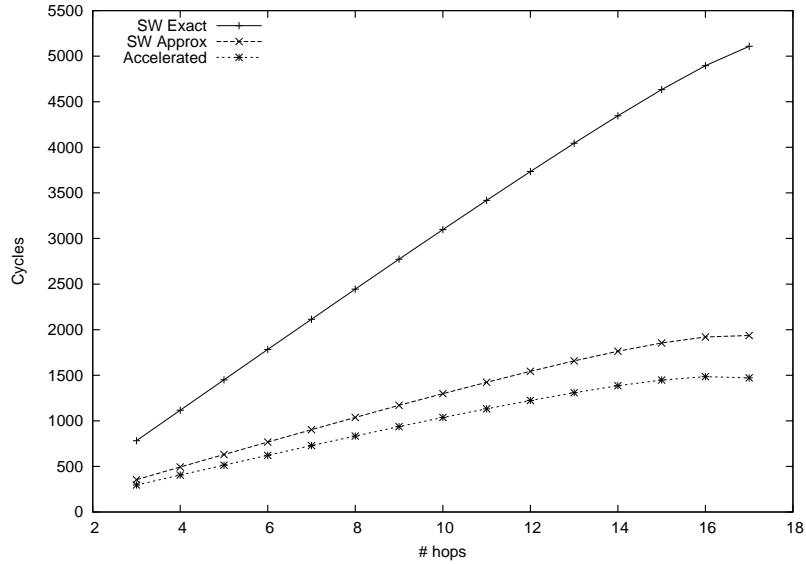


Figure 5.11: Allocation time vs. path length, 8x8 mesh, 0% background traffic.

it is exponentially rather than linearly dependent on the distance. It may be desirable in this case to limit the number of considered paths. If the algorithm considers only one path instead of exhaustive search, the running time is expected to return to the values in the previous graph, but the success ratio will be severely impacted.

The combination of background traffic and requested bandwidth also has an important effect on the running time. When the requested bandwidth is very low and the background traffic is not extremely high, a path can be easily found, almost as easily as in the case with no background traffic. If the requested bandwidth is much higher than the one that could be accommodated by the network, the algorithm will determine quickly that no route is possible. The longest running times are obtained when the nearest the maximum allowed for a successful allocation.

This relation between success rate and running time is confirmed by the experiments in Figures 5.13-5.20 which present the running time and the success rate as a function of requested bandwidth and distance for different values of background traffic.

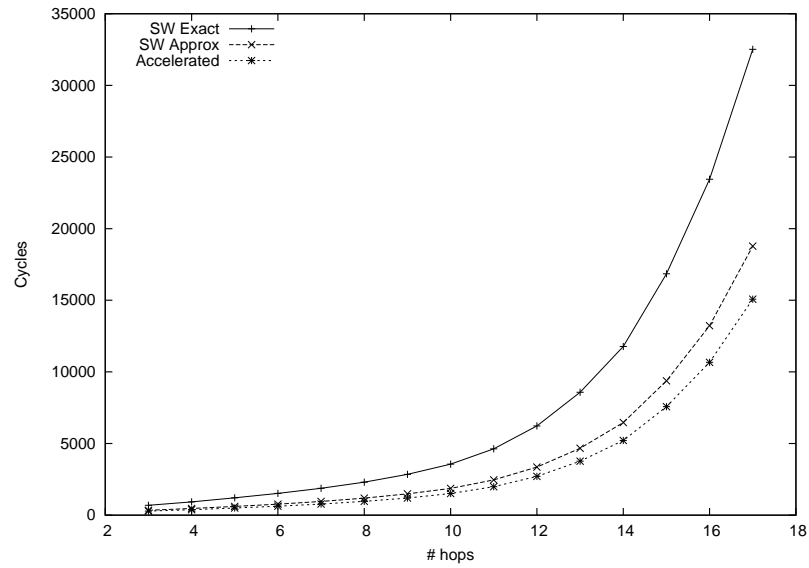


Figure 5.12: Allocation time vs. path length, 8x8 mesh, 10% background traffic.

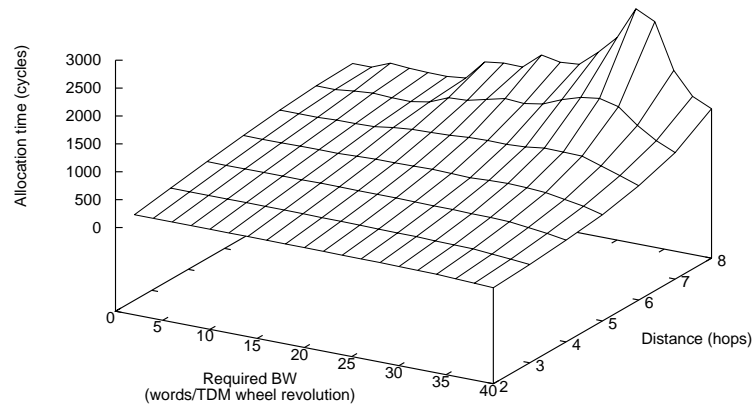


Figure 5.13: Allocation time vs. path length and requested bandwidth, 4x4 mesh, 10% background traffic.

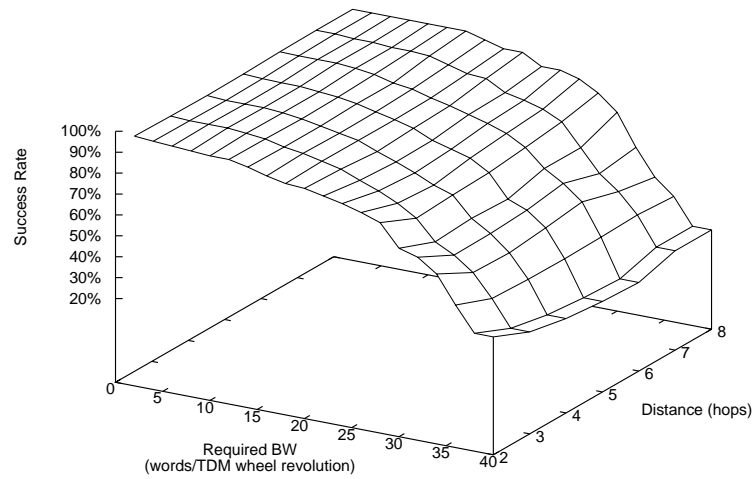


Figure 5.14: Success Rate vs. path length and requested bandwidth, 4x4 mesh, 10% background traffic.

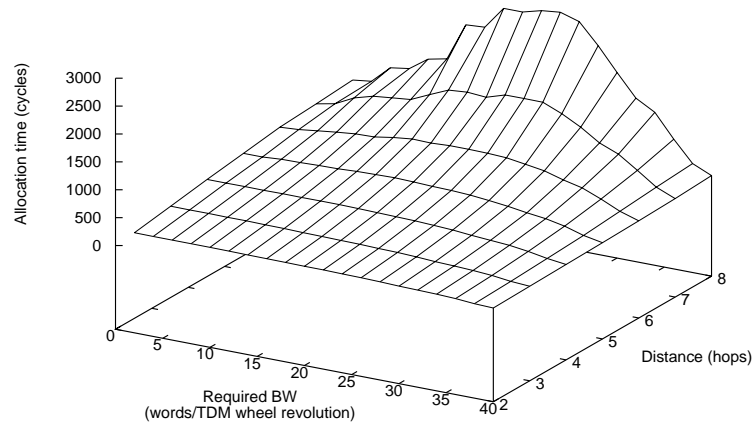


Figure 5.15: Allocation time vs. path length and requested bandwidth, 4x4 mesh, 20% background traffic.

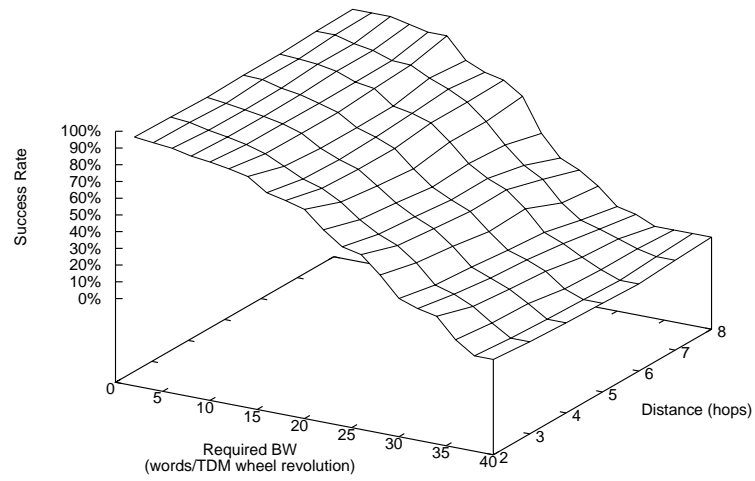


Figure 5.16: Success Rate vs. path length and requested bandwidth, 4x4 mesh, 20% background traffic.



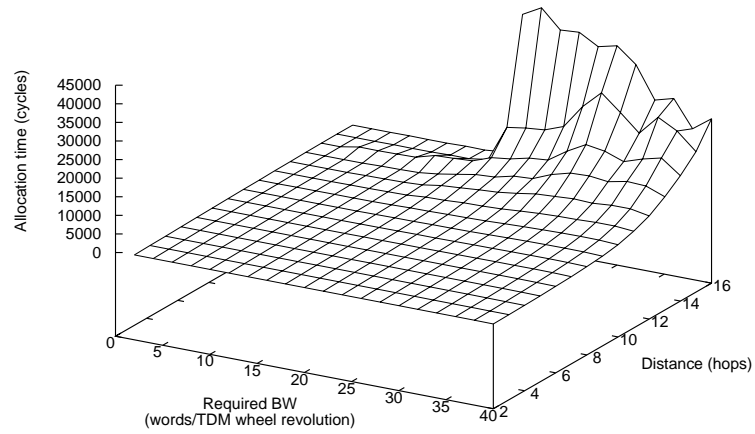


Figure 5.17: Allocation time vs. path length and requested bandwidth, 8x8 mesh, 5% background traffic.

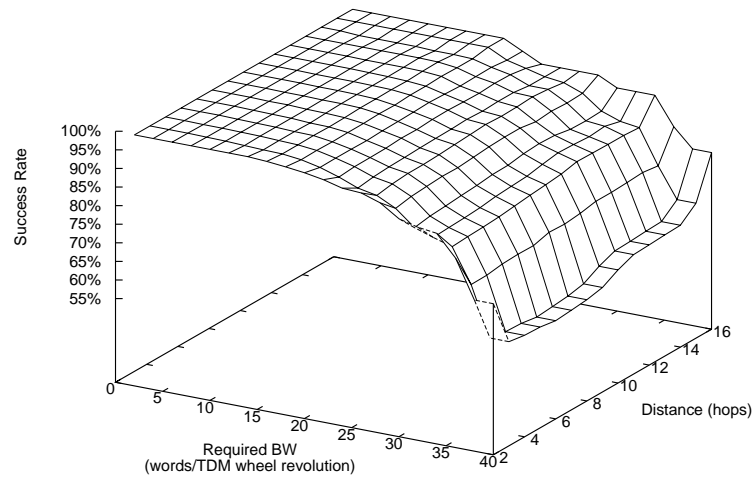


Figure 5.18: Success Rate vs. path length and requested bandwidth, 8x8 mesh, 5% background traffic.

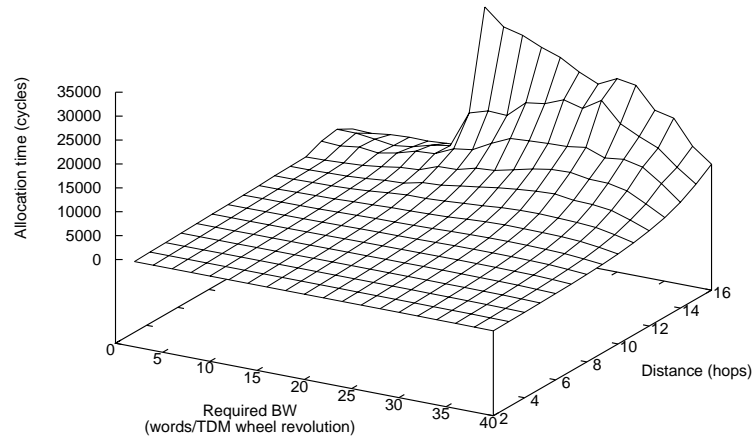


Figure 5.19: Allocation time vs. path length and requested bandwidth, 8x8 mesh, 10% background traffic.

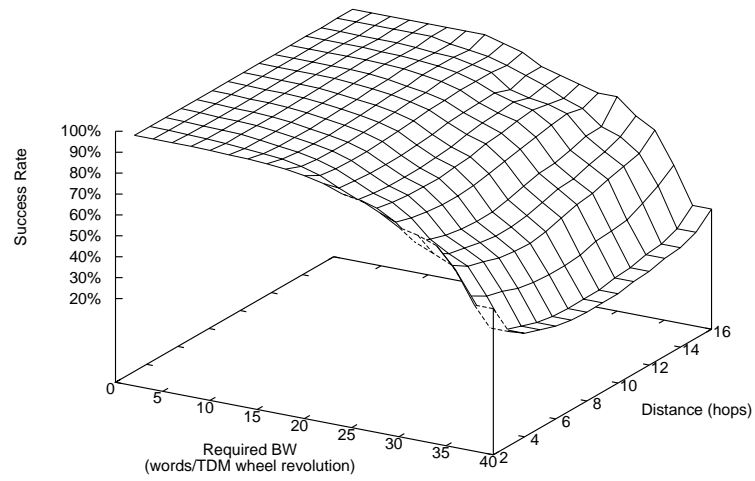


Figure 5.20: Success Rate vs. path length and requested bandwidth, 8x8 mesh, 10% background traffic.

It can be observed how the peak of the running time graph is shifted from the requested bandwidth of 34 words per slot table revolution in Figure 5.13 to 18 words per slot table revolution in Figure 5.15. This is caused by the increase in background traffic which decreases the chances of successful allocation.

For the 4x4 network, path-finding computation remains in the order of hundreds up to thousands of cycles. To that is added the time needed for book-keeping (keeping track of which slots are in use and which are not), which is also in the order of 100 cycles. As the time needed by *Æthereal* to set up paths is also in the order of hundreds of cycles [HG07], the overhead of performing allocation at run time is not particularly high.

For the 8x8 network the number of explored paths may need to be limited in order to allow reasonable running times, especially when communicating nodes are distant.

### 5.6.1 Memory requirements

The compact representation presented in Section 5.2 allows us to provide a complete description of the network topology with memory complexity  $O(n + m)$  where  $n$  is the number of nodes and  $m$  the number of links.

For a 4x4 mesh network the memory size used by the topology description is:

$$\begin{array}{rcl}
 & 80 \text{ links} \times 1 \text{ byte/link} & \\
 + & 32 \text{ IPs} \times 1 \text{ byte/IP} & \\
 + & 13 \text{ bytes (scalar data)} & \\
 \hline
 = & 125 \text{ bytes} & 
 \end{array}$$

For an 8x8 mesh network the size of the tables is:

$$\begin{array}{rcl}
 & 352 \text{ links} \times 1 \text{ byte/link} & \\
 + & 128 \text{ IPs} \times 2 \text{ bytes/IP} & \\
 + & 14 \text{ bytes (scalar data)} & \\
 \hline
 = & 622 \text{ bytes} & 
 \end{array}$$

An array of distances (Figure 5.3) is used by the path-finding algorithm. This array is particularly problematic as its size is  $O(nm)$  ( $n$  rows and  $m$  columns) where  $n$  is the number of network interfaces and  $m$  is the number of network nodes. Each row  $i$  in this array represents the distance from all nodes to network interface  $i$ . Only one row at a time is necessary during the path-finding process and this row could be recomputed as needed in order to save

memory. When the topology of the network is regular, for example mesh, it is also straightforward to compute these values on-the-fly instead of storing the table. For a 4x4 mesh network the size of this table is 512 bytes while for 8x8 it is 8 Kbytes.

The set of available slots is stored in a packed format with one integer for each link, one bit of that integer representing one slot on the link. This provides a straightforward representation for up to 64 slots (the largest integer size allowed by the MicroBlaze compiler). The advantage, in addition to the low memory requirements is that up to 32 slots can be manipulated at once (the microblaze is a 32-bit processor, although the compiler allows 64 bit arithmetic, operations will be split into multiple 32-bit instructions). Finding a common subset of slots between two links can be done using an “and” operation while advancing time by one slot can be performed with a shift operation.

The size of a slot table in the 8x8 mesh scenario with 32 slots is 1408 bytes.

### 5.6.2 The speed of the algorithm completely implemented in hardware

We estimate here the speed of the fully hardware accelerated implementation of the algorithm presented in Section 5.5. For this purpose we extract traces from the algorithm running in software, traces indicating all taken branches, and we use the model in Figure 5.10 to compute the number of cycles required by the computation.

There is an almost linear dependence between the speed of the software implementation and the estimated speed of hardware, affected only by the ratio of execution of different branches and initialization times. Individual channel allocation times using the hardware implementation (estimated) and the measured software implementation with bandwidth computation accelerated one are represented in Figure 5.21.

On average, the fully accelerated algorithm performs 19 times faster than the one with accelerated bandwidth computation and 42.4 times faster than the one without any acceleration (Figure 5.22).

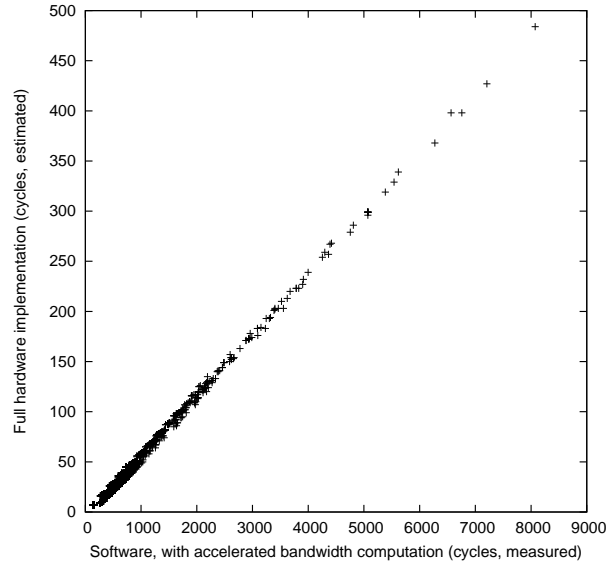


Figure 5.21: Speed of the hardware implementation vs. software implementation with accelerated bandwidth computation, 4x4 mesh, 20% utilization.

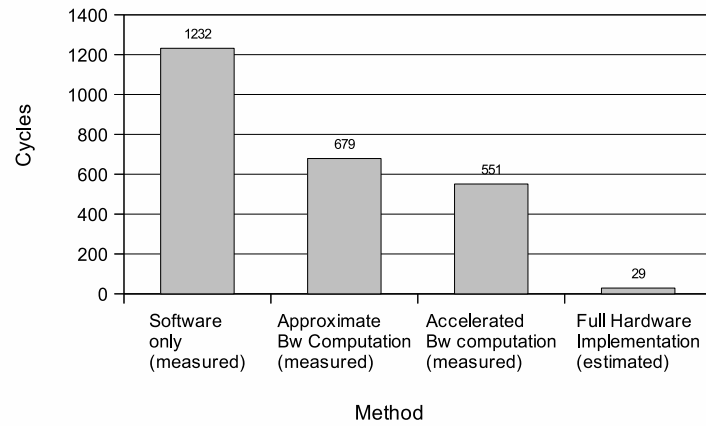


Figure 5.22: Average speed of all methods, 4x4 mesh 20% utilization.

## 5.7 Related Work

Dynamic time slot allocation in a TDM NoC has been studied before in [MBD<sup>+</sup>05]. The paper reports allocation times in the order of 1000 cycles per

hop on an ARM processor. This is deemed sufficient for an augmented reality 3d rendering application. The path-finding algorithms use a graph-splitting approach like the one described in Chapter 3 and are only able to deal with a bandwidth requirement of one slot. The reported size of the network is a few tens of routers.

In [MMB07], the authors propose runtime mapping of applications on a multi-core design also supported by the *Æthereal* NoC. The path finding algorithm employs as well a graph-splitting method. The algorithm running time is not presented.

A hardware accelerated NoCManager to perform path-finding and allocation is presented in [WF08]. The network architecture employed in that case is simpler, not making use of TDM (the equivalent *Æthereal* configured with only 1 TDM slot). The NoCManager was found to require 10 to 15 cycles for the allocation of one channel.

In [tBHK<sup>+</sup>10], the authors discuss routing in the context of run-time application mapping. Their experiments show that the failure in assigning tasks to specific locations in the system has a high probability of resulting from a failure in computing proper routing.

The assignment of virtual channels (or VCs) at run time in a NoC is discussed in [KSWJ06]. Although the platform is different, the approach is similar: a central authority assigns network resources (this time virtual channels instead of time slots) to connections requiring service guarantees. The problem is solved using a simple path-finding algorithm as it does not have to deal with the more complex time-domain allocation we encounter in *Æthereal*.

The hardware acceleration of graph algorithms including the problem of the shortest path is discussed in [Hue00]. The graph representation allows changing the graph connections at run time which can be used to account for the reservation and the release of links. While the shortest path problem formulation cannot be applied directly to produce allocations that require multiple slots for the same connection, it can be used for single-slot allocations.

A hardware accelerated solution for the routing wires in FPGA designs is presented in [DHW02] with reported speed-ups of 10x-1000x. This approach could be used as well to compute single-slot allocations.

An FPGA acceleration of the reachability and shortest path problems is also presented in [MHH02].

## 5.8 Conclusions and future directions of research

In this chapter we have presented an online implementation of the allocation algorithm which can be used at run time to dynamically compute the communication channel allocations. Our implementation is optimized for low resource usage and is able to properly take into account and avoid collision with previously allocated communication channels.

We also present methods to accelerate in hardware one of the more expensive operations of the allocation algorithm and we present a blueprint for a full hardware allocator. We have found hardware acceleration to provide a significant advantage in terms of speed.

Our method currently targets only the single-path allocation algorithm. We regard multi-path allocation as a promising direction for future research as the allocation algorithm is capable of executing in polynomial time.





## Chapter 6

### **dAEIite NoC Hardware implementation**

In this chapter, we propose a network on chip implementation based on the contention-free routing model. Based on the models introduced in Chapter 2 our network can implement Models 8-12 without a header overhead. The closest implementation to our proposal is aelite [HSG09] (a light-weight version of Æthereal) which supports Models 10 and 12 with a header overhead. We call our proposal dAEIite, as it uses distributed routing instead of the source routing used by aelite.

The study of various network models in Chapter 2 indicates that less restrictive models offer potentially better performance. Our network improves performance compared to aelite in three significant ways: it allows a finer granularity of bandwidth division without increasing latency, it removes the header overhead, and it allows multi-path routing at no additional cost. In addition it supports multicast, which aelite does not.

The first performance gain comes from a finer granularity of the link bandwidth division. In a TDM network, the granularity of allocation is given by the size of the TDM table. A large TDM table results in more efficient allocation, but it may increase the scheduling latencies as connections have to wait longer for their allocated slot(s) in the TDM table. It would be possible to reduce the scheduling latency by reducing the slot size, but in aelite this would increase the header overhead. dAEIite does not suffer from header overhead and as a result can reduce the slot size, offering a better link division granularity for the same absolute duration of a slot table revolution.

The second gain comes from the removal of the header overhead. aelite uses

source routing for packets, which means the path taken by each packet has to be encoded into the header of that packet. This results in a header overhead between 11.1% and 33.3% as explained in Chapter 4. Not only that, but to support long paths the header size or the link width may need to be increased. dAElite uses a distributed routing model which does not require the presence of headers.

The third performance gain is related to routing flexibility. The experiments in Chapter 2 and Section 6.4 show that the allocation can be performed better if communication channels are allowed to be routed over multiple paths. This is somewhat analogous to adaptive routing in packet switching networks which may provide better load balancing by dynamically switching paths to use less congested links.

In this chapter we will present the proposed NoC hardware architecture and we will evaluate its performance. The rest of this chapter is organized as follows. Section 6.1 presents the hardware implementation details the dAElite network. The network configuration process is presented in Section 6.2. Section 6.3 presents how multicast is achieved in dAElite. The hardware cost of and performance of the proposed solution is evaluated in Section 6.4. Related works are presented in Section 6.5. Section 6.6 presents our conclusions.

## 6.1 Hardware implementation

In this section we present the hardware implementation of our proposed NoC. We start by presenting an overview of a typical system based on dAElite, after which we will give the details of the configuration infrastructure, the router and NI architecture. All these elements were implemented and tested in FPGA.

### 6.1.1 System overview

A typical SoC platform based on dAElite is exemplified in Figure 6.1. dAElite is a connection based network. For a master IP to communicate to a slave IP over the network, a connection is set up between two network interface shells, one connected to the master, the other connected to the slave. The network interface shells have the role of translating the request between the bus protocol spoken by the IPs and the packet format used by data while traversing the network. This setup is similar to that of aelite (Figure 2.1 of [HG10]).

IPs are connected to the NI shells by lightweight local buses which have the

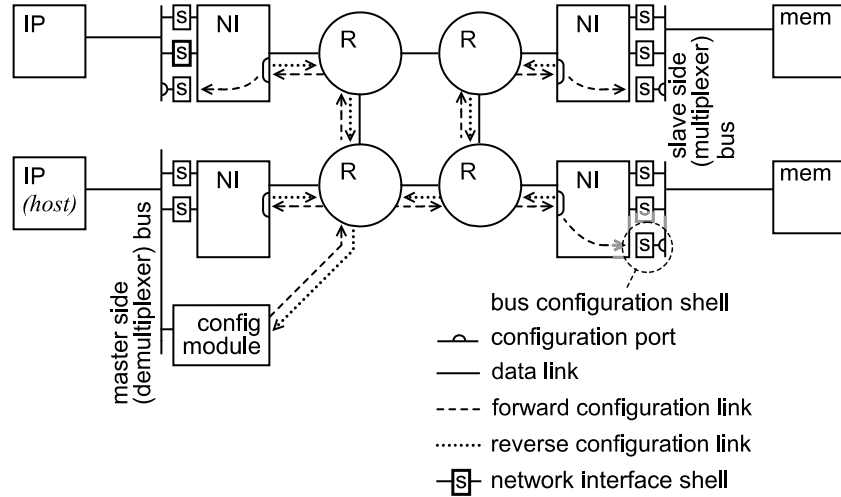


Figure 6.1: Example dAElite platform instance.

role of either demultiplexing requests to different network connections at the master side, or multiplexing requests from different network shells (and thus network connections) at the slave side. This is because the network connections are long-lived and an IP can have several connections simultaneously open to multiple other IPs.

The buses may be configurable, on the master side to select address ranges corresponding to each connection, on the slave side to select arbitration schemes and priorities.

A typical usage scenario is that the required connections are set up before starting an application or an execution phase of an application. The application can use the configured connections during that execution phase without further intervention to the network configuration. The connections are torn down once they are no longer needed. Setting up and tearing down connection can be done dynamically without affecting the normal operation of the system, i.e., an application can use existing open connections while others are being set up and torn down.

### 6.1.2 Configuration infrastructure

The configuration infrastructure is used to set-up and tear-down network connections by updating the contents of the slot tables inside routers and NIs, to

set and read back flow control information for each connection, to perform the synchronization of the slot counters inside NIs and routers and to configure buses adjacent to the network.

We implement the configuration logic as a dedicated broadcast network with a tree topology, with links running in parallel to a subset of the normal data network links. This subset is chosen in such a way as to minimize the distance from the configuration node to any of the network nodes.

One IP, by convention called *host*, has exclusive access to the configuration logic. The host performs write operations to a configuration module. These writes typically have a wide data width, e.g., 32-bit, compared to the width of the configuration links. The configuration module thus serializes the data words received from the host into several, smaller bit-width *configuration words* which are inserted at the root of the broadcast tree.

If the host does not need to send at one time as many configuration words as are contained in a data word, it can perform “0-padding.” The configuration module will also send “0” values into the configuration broadcast tree when it has no data to send.

The configuration tree provides of forward and a reverse paths. The forward configuration path is of broadcast type. Each non-leaf node (always a router), forwards all the configuration data it receives to all its downstream neighbors. The NIs, being leaves in the tree, do no forward the incoming configuration messages. They do however produce messages for the configuration of buses using a different type of link. The configuration payload is deserialized into wider 37-bit words which are then translated by an NI shell into the proper bus standard transactions (DTL in our case) used by the configuration ports of the buses.

On the reverse path in the configuration tree, messages converge toward the configuration module. To avoid arbitration on the response path, the host only issues one message requiring response at a time. In our case, the requests requiring a response are read operations directed at the state tables of the NIs.

The requests and responses traveling through the configuration network take the form of packets, the format of which will be presented in Section 6.2.1.

### 6.1.3 Routers

The structure of network routers is presented in Figure 6.2. Because we are using a distributed routing mechanism each router contains a slot table to

store the TDM schedule. Incoming packets are “blindly” routed (switched) based on this schedule. A high operating frequency can be achieved because no arbitration is required and the router does not need to examine the packet contents.

The schedule for packet destinations, or more precisely the source of each of the outputs during each slot, is contained in a slot table. A counter iterates circularly through this slot table and the selected row is used to control the router crossbar. A configuration submodule, implemented as a state machine is used for setting the initial value of the counter, as we will discuss in Section 6.2.4 and to update the contents of the slot table.

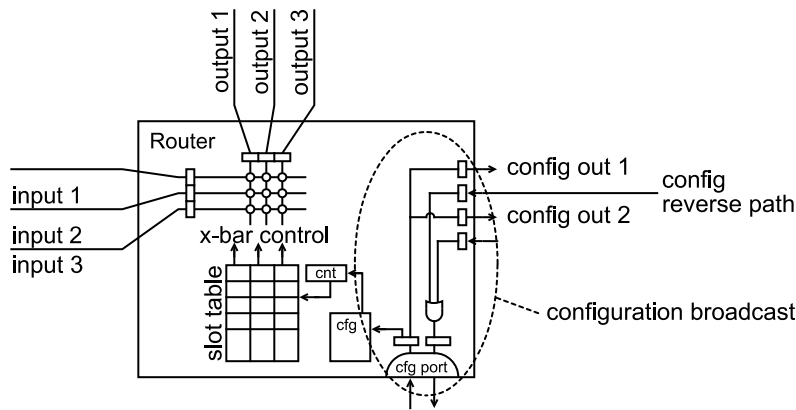


Figure 6.2: dAElite Router

On the configuration connections, the router simply copies the input value to all of the outputs on the forward path and performs an “or” operation between all inputs writing the result to the output on the reverse path.

Data is thus buffered twice inside the router: once after link traversal, and once after crossbar traversal. The latency per hop is thus fixed to two cycles. To simplify the network design, configuration data is also buffered twice at each hop in the configuration tree. This allows the configuration links to be treated in the same way as the data links when dealing with timing constraints.

#### 6.1.4 Network Interfaces

Network interfaces have the role of providing connections over the network. A network interface (NI) is connected using a network data link to a network

router, and one or more links to network interface shells. Each link to an NI shell supports a connection to another NI shell at the other side of the network. NIs thus multiplex several communication channels having NI shells as endpoints to a single network data link.

As specified by the contention-free routing model, the packets belonging to different connections are inserted into the network only at specific times. The arriving packets are also forwarded to the proper NI shell based on their arrival time, according to a strict schedule. The departure and arrivals schedule is stored inside a slot table which is part of the NI. The slot table controls the multiplexer and demultiplexer in the same way the router slot table controls the router crossbar.

Figure 6.3 presents a diagram of the network interfaces. The network slot table, same as the one of the router, is indexed with the value of a circular counter and is programmed by a configuration submodule. For each of the data connections there are input and output FIFOs, credit counters for end-to-end flow control and decision logic for enabling or disabling the sending of data from an input FIFO to the network.

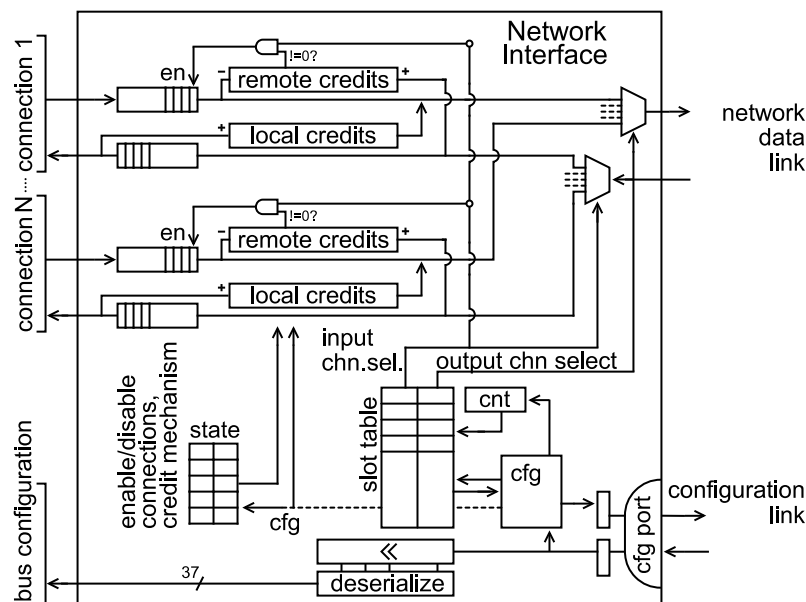


Figure 6.3: dAElite Network Interface

Credit-based flow control is provided by the NI for each of the one-to-one

connections going over the network (optionally flow control can also be disabled on a per-connection basis). Two credit counters are used for each connection. One credit counter keeps track of how many words of storage space are available in the output FIFO at the other end of the network. An input FIFO is only allowed to send data into the network if the value of this credit counter is different than 0 and the slot table indicates it is the connections' turn to send data. A second credit counter accumulates the number of words that were delivered to the destination from the output FIFO. The value of this counter is sent back to the other end of the connection in every slot allocated on the return path and the counter is reset.

As connections are bidirectional, credits for one direction are sent on separate bit-lines alongside data in the opposite direction. The separate credit lines and data obey the same TDM scheme and there is actually no distinction between the two at the router level. Other networks, like aelite [HSG09] send the credits inside packet headers, but that approach is not viable here as dAEIte does not employ packet headers. The number of bit-lines transporting credit information is configurable. To make better use of these lines, the value of the credit counters is sent serialized, over the 2 cycles of a TDM slot. In our test design, 3 wires dedicated to sending credit data are sufficient for sending the value of a 6-bit credit counter during each slot cycle.

The configuration submodule is responsible for updating other network state information like enabling or disabling connections, enabling or disabling flow control for each connection, and reading or writing flow control information to credit counters. It also identifies special messages which are destined to the buses and deserializes them to the bus configuration shell.

## **6.2 Network configuration procedure**

This section describes in detail the network configuration tasks. Special attention is given to the connection set-up and tear-down which is the most complex operation. We describe as well as packet format for the other operations and we present our proposed method for synchronizing the slot counters inside routers and network interfaces.

### **6.2.1 Configuration packet format**

The configuration is performed using configuration packets, consisting of one or more words, transmitted one per cycle over the configuration links.

The configuration links have small bit-width that is equal to the size of the configuration words. To optimize the logic of the state machines governing the configuration process, this width is selected in such a way that each of the parameters listed below can be encoded in a single configuration word in addition to one bit marking the command headers.

- a value uniquely identifying a specific router or NI, of size  $(\log_2 N)$  where  $N$  is the number of network elements;
- two values identifying one input and one output port of a router or a null entry, of size  $(2 \times \log_2(p + 1))$  where  $p$  is the maximum number of ports on any router;
- an NI port (connection) number or null value, and an additional bit to indicate whether the input or the output slot table is to be updated, of size  $(1 + \log_2(c + 1))$  where  $c$  is the largest number of connections supported by an NI; and
- the value of a credit counter plus one indicator bit (which marks valid response data), of size  $(1 + \log_2(b + 1))$  where  $b$  is the size of the largest buffer the counter must keep track of.

A configuration word size (including a one-bit marker for packet type flits) of 7 bits is sufficient for a network with 64 network elements (routers and NIs), routers with an arity of 7, network interfaces supporting up to 31 channels and buffer sizes of up to 31. From here onwards, we denote as *configuration word* a 7-bit word transmitted on a configuration link.

The format of the configuration packet is illustrated in Figure 6.4. An end-of-packet is implicitly marked by the beginning of a new packet, but can also be marked explicitly. The configuration mechanism supports the writing of slot tables, reading and writing credit information, writing status flags governing NI behavior, and resetting internal TDM counters.

The first word in each packet indicates the type of configuration command. The first bit is always zero for a command word. The packet formats for the various operations are as follows (Figure 6.4):

For set-up and tear-down operations, the command word is followed by a list of slots used at the destination NI, represented as a bit-mask with one bit per slot. A list of the traversed Routers and NIs follows, each with a corresponding input and output descriptor. Each affected NI/Router recognizes its own identifier (a constant parameter defined at circuit design time) from the list and modifies its



PACKET TYPES	NOTES
network reset [ 0 - - - - 1	initializes the NI and Router timers
read credits from NI credit counters [ 0 - - 0 1 1 0 1 NI id 1 port	
write credits or status flags to NI [ 0 - - 1 1 1 0 1 NI id 1 sel port value 1 sel port 1 value	select between credits/flags it is possible to perform multiple write operations to credit counters or flags of the same NI
write local bus configuration [ 0 - - 1 0 1 0 1 NI id 1 value 1 value	serialized data MSB first
connection setup or teardown [ 0 - - 0 1 0 0 1 slots 1 slots 1 1 NI id 1 1 port 1 n id 1 port dest 1 0 NI id 1 1 port	list of affected slots target NI intermediate router source NI
SPECIAL FLITS	
empty flit [ 0 0 0 0 0 0 0	this value can be used as padding inside any packet
end-of-packet [ 0 - - 1 0 0 0	this is an optional flit used to mark the end of a packet

Figure 6.4: Format of the configuration packets.

internal slot table. The path set-up procedure will be described in detail in the following sections.

Comparatively, the other configuration operations are simpler. For the reading of credits, a header, the NI identifier (ID), the port (connection) number are broadcast into the network and the addressed NI will recognize its own identifier from the packet and will reply with the counter value over the reverse configuration link. The response consists of the credit counter value and a bit (an MSB of 1) that marks the presence of data.

Writing of credits is similar, except after the port selector the new value of the counter is sent. The addressed NI, once selected will keep watching for port/value pairs. In this way, writing multiple credit counters of a single NI can be done in a single configuration packet.

Writing to a bus programming port is again performed with a packet marked by a distinctive header. The addressed NI will deserialize the received data to the shell connected to the configuration port of the bus. We used this approach for compatibility reasons with existing configurable bus implementations.

### 6.2.2 Setting up and tearing down connections

For setting up or tearing down a network connection, several operations have to be performed:

1. setting up network paths;
2. initializing credit counters; and
3. initializing bus address decoders.

Steps (1) and (2) have to be performed for the both the request and response channels. Step (3) is the last one performed as it signals to the bus that it can start using the connection for transferring data.

Connection tear-down can be performed in the following way:

1. the bus address decoders are reset;
2. the credit values are read back from the credit counters to check if all data items have reached their destination; and
3. the request and response paths are torn down.

Step (1) ensures that no more requests will be pushed over the connection which is torn down. Step (2) is necessary to make sure that no packets belonging to the connection that is being torn down are still in-flight through the network. Step (3) performs the actual path tear-down. As an additional safety measure to prevent packets being sent over half-torn-down paths, the connection can be disabled using by setting a (per-connection) configuration bit inside the NI.

6.2.3 A path set-up example

We illustrate in this section step-by-step how a path set-up is performed. Consider the system in Figure 6.5. We analyze the operation of setting up a path from port 0 of NI10 to port 0 of NI11. The IP which has access to the network configuration, which we call the *host IP*, writes data words to the configuration module. As the bit width of the dedicated configuration links is typically lower than the data width of the bus, a single write from the host may contain multiple configuration words (for this system 4), which are sent over the configuration link by the configuration module.

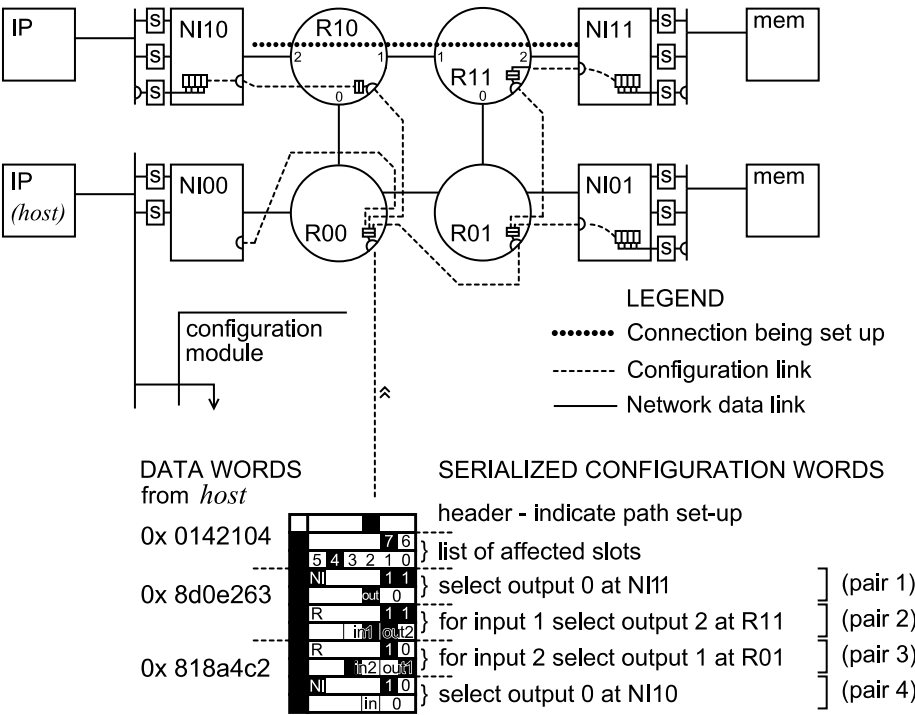


Figure 6.5: Path set-up example.

The first configuration word is a header that informs all the network elements that a path setup sequence will follow. It is uniquely identified by a most significant bit with a value of 0 and the decimal value of 4 in the last 4 bit positions. The next two configuration words contain a table of slots affected by this path set-up operation. We assume here a slot table size of 8. The two bits set to one in this example identify slots 7 and 4.

Three cycles after starting the path set-up process (Figure 6.6a), allowance being made for whatever pipeline stages the configuration logic may contain, the complete slot table has been registered at router R00, while routers R01 and R10 have just recognized the path set-up header and entered set-up mode. This is because each router has two pipeline stages on the configuration path, thus delaying the configuration by two cycles for the next routers in the broadcast tree. The configuration module, having emptied its serialization register is able to accept one more data word.

The configuration words after this point are organized in pairs. The first word in a pair identifies a network element while the second describes modifications to the slot table of the identified element. The MSB of each word is set to identify a valid configuration word. For the first word, a value of 1 in the MSB-1 position identifies an NI while a value of 0 identifies a router. The ID is stored in the least significant bits. For NIs the least significant bits identify the port (connection) number and the following bit identifies the input or output slot table. For routers, the least significant bits identify the input port and the immediately higher bits the output port.

The meaning of the configuration words in Figure 6.6a is the following: for the defined slots, the input of the Network Interface NI11 should be forwarded to channel 0. On the previous path segment, but one time slot earlier, router R11 should forward data from input port 1 to output port 2.

After two more clock cycles (Figure 6.6b), the path set-up header reaches R11 and NI10, the slot table is registered in R10 and R01, and the first network element identifier pair reaches R00. Because R00 does not recognize its own ID in this pair, it ignores the pair but at the same time it rotates (bitwise) its table of affected slots by 1 position.

Four cycles later (Figure 6.6b), after another 4 configuration words have been transmitted, the table of affected slots is finally stored by all network elements, rotated by a different number of positions. So far, the network element IDs in the configuration packets never matched the element IDs of the traversed routers. The number of rotations in the slot table seems to depend so far on the distance from the root of the configuration tree, but this is misleading. The number of rotations actually depends on the number of path elements in the received list that did not match the local network element ID.

Eventually a configuration word carrying the proper ID reaches the network elements that need to be configured. In our case that happens simultaneously for R10, R11, NI11, as shown in Figure 6.7.

In order to avoid configuring upstream nodes before downstream nodes, we

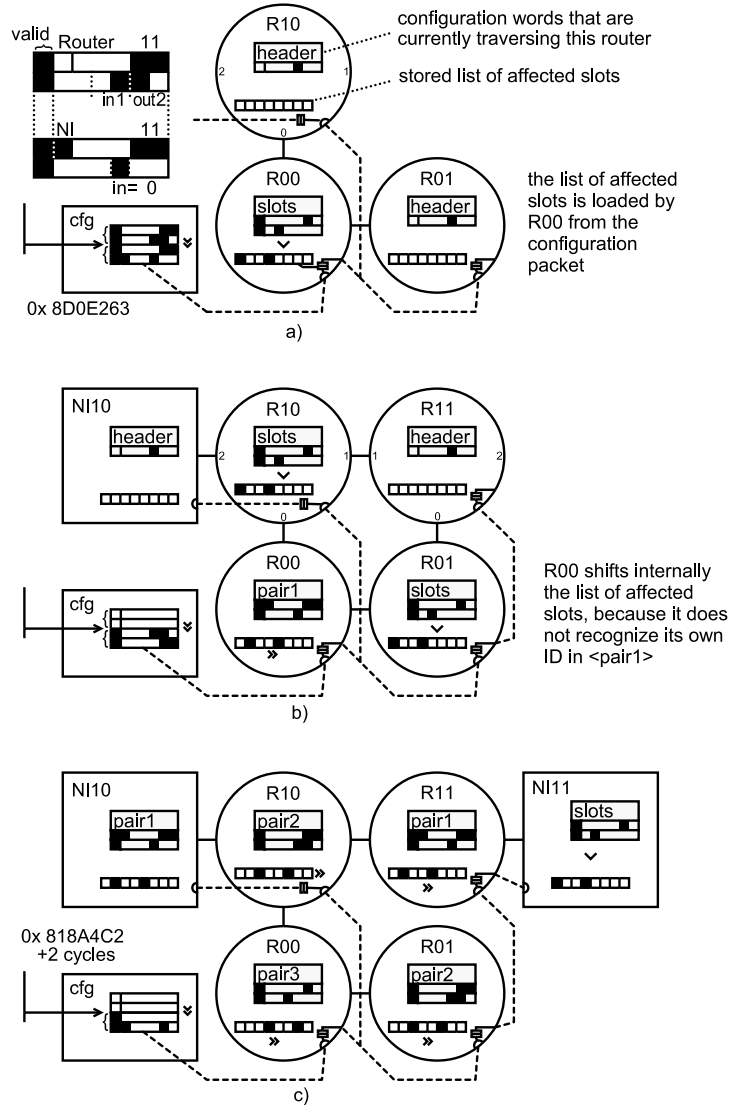


Figure 6.6: Path set-up, slot table registered at the first router.

had to take an additional precaution: we need to verify that no path for data exists that is shorter than the difference in configuration path length from the root of the configuration tree. Optionally the enable-disable mechanism per connection can be employed for the same purpose.

The update of the slot table is not performed instantaneously but through the

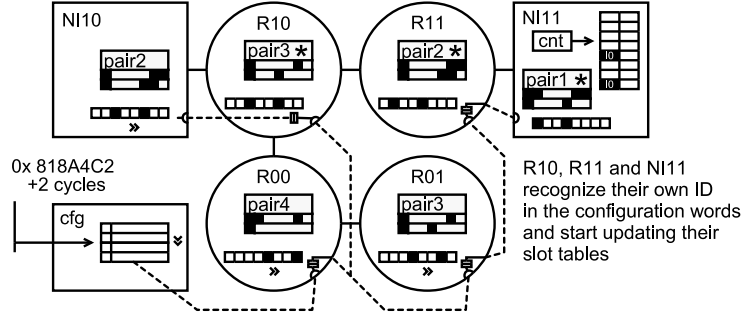


Figure 6.7: Path set-up, slot table update takes place in R10, R11, NI11.

use of a counter, one slot table element is initialized each cycle. We preferred this approach to reduce the cost of FPGA implementation. This implies that a cool down period is necessary before another path set-up or teardown may take place. A hardware cool down timer inside the configuration module enforces this policy by delaying channel set-up packet headers as long as it is different from 0. Other configuration packets are allowed during this time so the initialization of credit counters of bus address decoders can overlap with the initialization of slot tables.

The list of traversed routers/NIs begins at destination to ensure that downstream routers are initialized before the upstream NI and routers start sending packets. For each item in the list, the slot table which was sent in the beginning of the packet is rotated by one slot so that all routers along the path, when they recognize their ID in the list, already have the properly aligned table. It is not mandatory that a packet contains a complete source-to-destination NI path, independent path segments can be initialized as well. This can be used to set up broadcast trees, for example.

Appendix B provides an example configuration program.

#### 6.2.4 Slot counter synchronization

The dAElite network is at the logical level a synchronous network implementation, which furthermore relies on a notion of global time. This may in practice be difficult to achieve [LSGB11] due to clock skew issues in large designs. Nevertheless, studies have shown [BKVW03] that clock skew values in the order of tens of picoseconds are achievable. Other approaches exist [HSG09] that avoid the problem by offering synchronous behavior at the logical level

while relying on a mesochronous or asynchronous implementation at the physical level.

In this section, making abstraction of how the synchronization issue is solved at the physical level, we show how to solve the issue of synchronizing the slot counters at the logical level. To illustrate the synchronization problem, consider the following scenario, illustrated in Figure 6.8. We use the following simplification which allows us to explain the synchronization mechanism: assume that the clock skew between neighboring nodes (nodes that are connected by a network link) is sufficiently small to allow correct data transmission between the nodes and allow the node to agree on a common value of the current time slot when they exchange data. Over large distances between nodes that are not connected using a link that is not required.

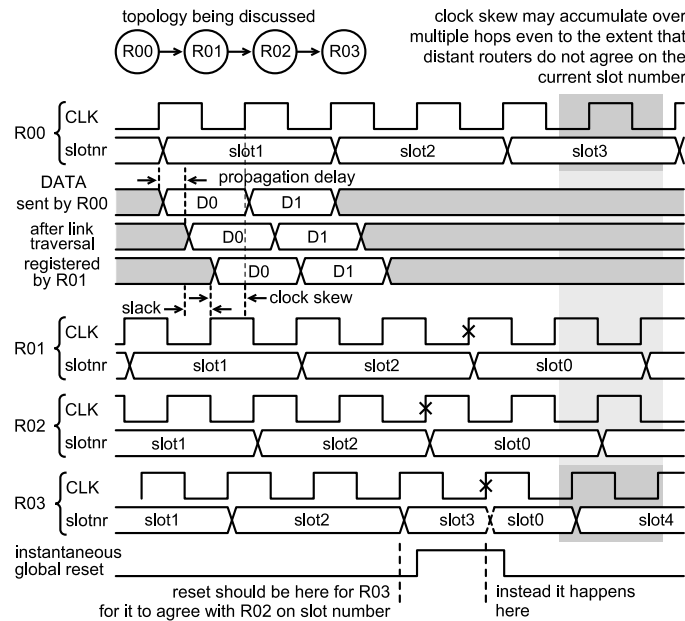


Figure 6.8: dAElite relaxed synchronous model.

This may be a reasonable assumption for a mesh network where links are short and connect only physically neighboring nodes. The important aspect here is that each router agrees on the value of the slot counter and is able to transfer data to its neighbors. It is sufficient if this happens only from the logical point of view, regardless of the physical implementation.

If the slot counter values were reset using a global reset signal, and that reset

signal did not follow the same skew pattern as the clock, the reset signal will inevitably be sampled on the wrong clock edge in one of the various clock skew domains. (Figure 6.8 shows how an instantaneous reset signal which does not follow the pattern of clock skews causes R02 and R03 to disagree on the current slot number).

To solve this problem we provide a reset mechanism which follows the logical view of network time. The reset signal is transmitted as a special packet through the configuration network. This packet will arrive at the different nodes in different clock cycles, but the logical delay is known (two cycles per hop) and the nodes compensate for it by initializing their slot counters to the distance to the root of the configuration tree. This solution also allows for pipelined links which can be useful in achieving high frequency of operation.

Another advantage of this approach is that it allows resynchronizing the NoC slot counters after a partial power down.

### 6.3 Multicast

dAElite offers a mechanism to achieve multicast that is both simple and efficient. The TDM schedule in a dAElite router is implemented as a table that specifies for each output port which input port should the data be taken from during each cycle. Two (or more) output ports are allowed to use the same input port as a source (Figure 6.9).

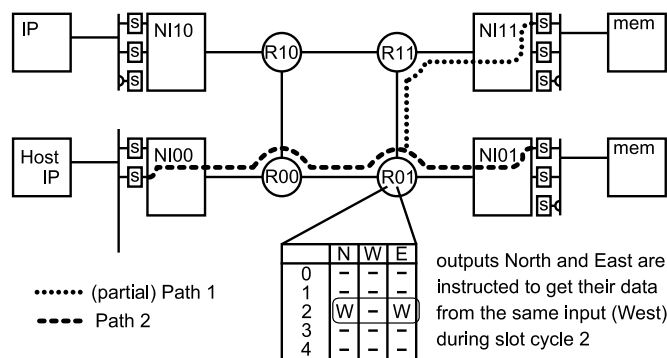


Figure 6.9: Multicast in dAElite

The multiple paths to the different destinations form a tree, rooted at the source NI. This is more efficient and offers higher performance than having separate



connections from the source NI to all destinations because in the latter case the bandwidth on output link of the source NI would need to be divided between all the connections.

The initialization of multicast trees is made possible by allowing a configuration packet to set up partial paths; i.e., paths that start at a router instead of a source NI. In the tree, partial branches should be set-up first, before the “main branch” which goes to the root of the tree. This is done to avoid the source NI starting to transmit before the entire multicast tree has been set up. In the example in Figure 6.9, the partial path R01-R11-NI11 should be set up first and the path NI00-R00-R01-NI01 should be set up second.

All multicast destination shells will receive the same stream of messages and will translate them into the same write commands on the destination buses. In fact, depending on the IP/bus protocol, any kind of transaction or message can be transported to all destinations.

One potential problem when using multicast is that the default flow-control mechanism cannot be used (the source NI only has one credit counter for each communication channel). The least expensive solution to this problem is to guarantee that the destination bus can process the received memory transaction at the same rate that they are transmitted. In our scheme guaranteeing this is made easier by the fact that the connection bandwidth can be set to a desired value (with a certain granularity) by allocating more or fewer TDM slots to it.

## 6.4 Performance and hardware cost

Comparing NoCs is not straightforward as the services provided by two different NoCs may be different. Furthermore many of the solutions presented in the literature only give details of the network routers which makes it difficult to assess the cost of an entire network able to deliver the service level demanded by an application.

The obvious target for comparison is represented by the *Æthereal* network, with which *dAElite* shares the contention-free-routing based TDM model. Several implementations of the *Æthereal* network exist, some of them supporting best-effort (BE) services in addition to the guaranteed services. Supporting BE services was found to be much more expensive [GH10]. *aelite* [HSG09] inherits the GS-only model from *Æthereal*, and introduces the possibility of using asynchronous and mesochronous links. From here onwards, we will refer to the GS-only version of *Æthereal* as *aelite*, without any implications

to a particular asynchronous or mesochronous link implementation scheme. We dedicate the largest part of the cost and performance evaluation to the comparison with aelite.

We have verified our design using Microblaze processors as the IPs in Figure 6.1, both by running in FPGA and visually inspecting the waveforms in simulation. We have specifically verified the proper functioning of the following operations:

- Network configuration, link set-up and tear-down, setting status flags, setting and reading credit information
- Normal reads and writes from the Host IP to the remote memories
- Broadcast writes from the Host to both remote memories
- Configuration of a remote bus
- Read and write from secondary IP after its bus has been configured

#### 6.4.1 Hardware cost

The hardware cost of dAElite compared aelite is presented in Table 6.1 and Figure 6.10. Both implementations consist of 2x2 mesh networks, with 4 NIs and 4 connections. The sizes of the FIFOs implementing the queues inside NIs have been set to 16 words except for the aelite configuration channels which use the minimum required size of 3 words on the forward path with 8 words on the return paths. dAElite uses the dedicated configuration infrastructure. We believe the FPGA implementation can be further improved by taking advantage of FPGA specific structures to implement slot tables and FIFOs.

For the FPGA implementation using the Xilinx tools, we performed runs with both area and speed optimizations. For the ASIC implementation we did not perform time-constrained synthesis as timing is more likely to be dictated by floorplaning which we did not address. We performed instead two synthesis runs, one with preserving the individual components and one with design flattening and high optimization settings. The cost of each hardware component is presented in Figure 6.11. The bulk of the cost is in the network interfaces due to the relatively large FIFOs.

For the routers, the gain of our implementation is on account of the reduction in flit size to 2 which eliminates one register per router channel. A multiplexer per channel per router is also eliminated as we do not need to shift routing information in packet headers. We add instead more complex configuration logic and a slot table.

	Xilinx ISE Virtex 6 - Area	Xilinx ISE Virtex 6 Constrained 200MHz	Synopsys TSMC 65nm	Synopsys TSMC 65nm flatten, high effort
aelite (8 slots)	5,500 slices 7,665 LUTs 15,444 registers 8.379 ns	5,393 slices 9,403 LUTs 15,840 registers 4.986 ns	182,419 $\mu\text{m}^2$ total 44,419 combinational 137,992 noncomb. 1.44 ns	171,820 $\mu\text{m}^2$ total 39,563 combinational 132,261 noncomb. 1.13 ns
dAEIite (8 slots)	3,098 slices 10,026 LUTs 12,323 registers 8.19 ns	3,655 slices 12,470 LUTs 12,973 registers 4.968 ns	155,509 $\mu\text{m}^2$ total 39,185 combinational 116,317 noncomb. 1.36 ns	141,233 $\mu\text{m}^2$ total 29,541 combinational 111,693 noncomb. 1.08 ns
dAEIite (32 slots)	3,483 slices 10,903 LUTs 13,533 registers 8.84 ns	4,425 slices 14,335 LUTs 14,191 registers 4.85 ns	171,476 $\mu\text{m}^2$ total 43,611 combinational 127,865 noncomb. 1.36 ns	156,639 $\mu\text{m}^2$ total 33,410 combinational 123,230 noncomb. 1.08 ns

Table 6.1: Hardware cost comparison.

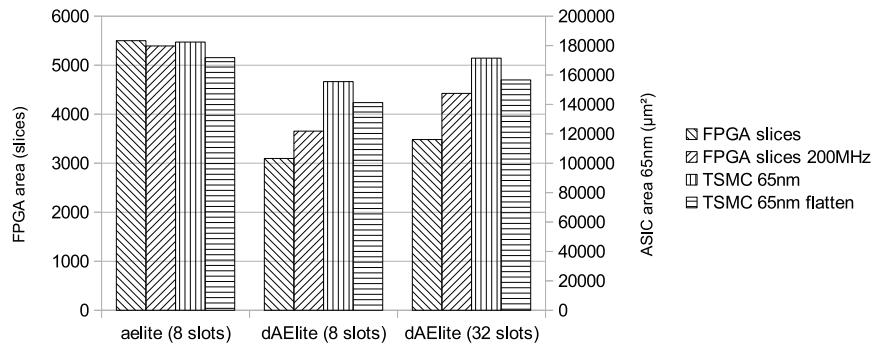


Figure 6.10: Hardware cost comparison.

Compared to aelite, our NIs benefit from a simpler configuration mechanism which uses a state machine connected directly to the configuration infrastructure, instead of interpreting requests on a DTL bus (aelite uses a regular network channel for configuration, followed by a shell translating the configuration messages into DTL bus transactions.) Further area gains originate from the removal of the table of paths inside the NI, and some of the configuration buses.

The cost of dAEIite is more sensitive to the size of the TDM wheel. The slot tables inside the NIs need to contain entries for both departures and arrivals

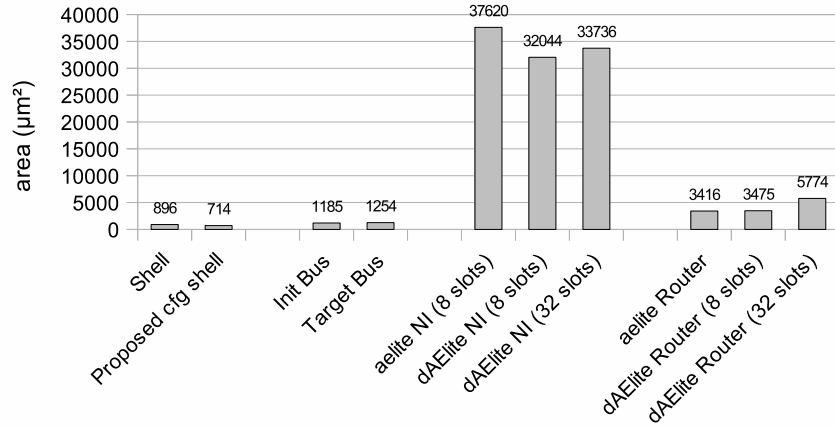


Figure 6.11: Hardware cost breakdown.

whereas in aelite they only contain entries for departures. Additionally, slot tables are present in each router. We expect, and we confirm experimentally that the cost increases linearly with the number of slots. It is expected therefore that as the number of slots is increased the hardware cost of dAElite will increase relatively to aelite. We performed additional synthesis runs to determine the point where the cost of dAElite becomes higher than that of aelite. We found that point to correspond to a TDM table size of 70 (Figure 6.12). In practice we do not expect to use TDM table sizes larger than 32 slots.

On the other hand, the cost of dAElite increases less than that of aelite with the number of connections, because a path per connection is not stored inside the NIs (the path is stored in a distributed manner inside the router slot tables). Our setup, which uses a relatively low number of connections provides a conservative estimate of the hardware area benefit of dAElite.

dAElite has one disadvantage, namely a 20.8% increase in the number of link wires, in part due to the configuration network, and in part because of the separate wires for end-to-end credit communication. There is a slight variation in overhead across topologies and sizes, because the configuration network is only a subtree of the original topology. For example, on a 4x4 mesh the wire overhead stands at 19.1%. The relative overhead would decrease if wider links were to be considered.

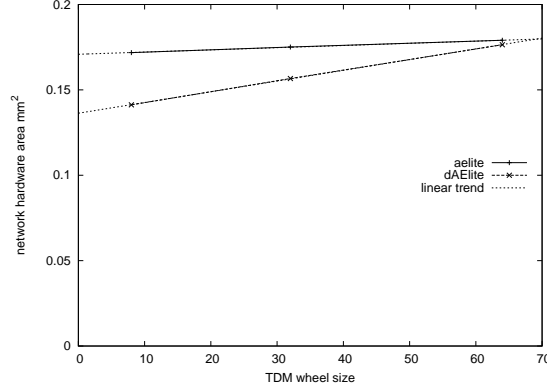


Figure 6.12: Dependence of the hardware cost on the number of slots

### 6.4.2 Configuration time

In our proposal, the path set-up time is only dependent on path length as all slots reserved for a connection are configured at once, for each one of the routers on the path.

We present the results of experiments in Table 6.2. All results are expressed in cycles. In both cases the configuration code is written in C and compiled with maximum optimization. The ideal value reported for aelite is taken from [HG07] and represents the configuration delay without taking into account processor execution time of the configuration code, but only the actual read and writes. In aelite, the path set-up time has a small dependence on the distance from the host to the source and destination of the path, and to a larger extent to the number of slots reserved on that path, as the slots in the source and destination NIs are reserved separately.

The ideal value for our proposal is computed analytically from the number of configuration words that are being written in each case to which the cooldown latency was added.

The improvement in set-up time compared with aelite is due to the following:

- overheads are reduced: the original implementation used DTL transactions encoded into packets;
- we avoid transmitting redundant information, the slot tables at the intermediate nodes are generated by rotating the slot table at destination;

	aelite		dAElite	
	ideal	measured	ideal	measured
6 hops	246	822-1555	60	81
8 hops			68	94
10 hops			76	119
12 hops			84	133

Table 6.2: Configuration times

and

- the configuration bandwidth is not dependent on the slot table size and the number of slots allocated to configuration. A dedicated infrastructure is less expensive than implementing the same functionality over a generic network.

Our FPGA experiments indicate that dAElite configuration is roughly one order of magnitude faster than aelite.

### 6.4.3 Performance

dAElite has several advantages in terms of performance compared to aelite. The primary sources of performance increase are the ones described in the beginning of this chapter:

- Bandwidth division granularity can be improved without increasing the slot wheel revolution time. In the current implementation (a slot size of 2) the ratio between the granularities of the two solutions is  $2/3$ , thus a slot table of 24 slots in dAElite will have the same period as a table of 16 slots in aelite. It is also possible to decrease the slot size to 1 resulting in a factor of 3 improvement.
- The header overhead is eliminated. Sending headers with each packet when the schedule was in fact static is indeed wasteful both in terms of bandwidth and energy. The header overhead in aelite is between 11.1% and 33.3% depending on whether consecutive slots are used by the same communication channel or not.
- dAElite allows multi-path routing without any additional hardware overhead on any number of paths. Multi-path offers more freedom to the routing computation and may allow a more efficient allocation.

The total improvement, and the improvement due to each of the three sources is quantified in Table 6.3. The experiments setup used here is the same as the one presented in Chapter 2. The traffic type is random traffic, the slot table sizes are 16 and 24, the number of connections is 1.5 per IP and the number of IPs matches the number of network interfaces which in turn matches the number of routers.

The three sources of improvement may interact with each other. We analyze all combinations of the three sources and report the minimum and maximum (from all combinations, average value per topology). More explicitly, the improvement of A is measured in the presence and absence of B and/or C and we report the minimum and maximum out of all combinations of B and C.

	Total Bandwidth Gain	(A) Granularity	(B) Header Overhead	(C) Multipath
4x4 mesh	28.8%	5.8% - 7.6%	17.1% - 19.2%	1.1% - 3.1%
6x6 mesh	44.6%	5.7% - 13.8%	20.7% - 29.9%	5.3% - 7.4%
8x8 mesh	53.3%	7.0% - 17.4%	20.7% - 30.6%	7.1% - 13.1%

Table 6.3: Performance improvement in dAElite compared to aelite

The proposed implementation presents other performance benefits as well.

- D) In aelite the configuration takes place over the normal network connections and a certain bandwidth over these connections is “consumed” by the configuration channels. At least one slot over each NI-to-Router and Router-to-NI link needs to be dedicated to configuration which results in an overhead of 6.25% assuming a slot table size of 16. Because dAElite uses a dedicated configuration network, this overhead does not exist in our case.
- E) Router traversal time is decreased in dAElite, as the dAElite router has only two pipeline stages (one after the link traversal and one after the crossbar traversal). Performance is not degraded as header parsing is also eliminated from the router, eliminating the need for the extra pipeline stage.
- F) Our proposal also provides a mechanism to synchronize the TDM wheel clock across Routers and NIs based on our broadcast configuration mechanism. This can be used for example to wake up from sleep states.

All these improvements stack up, making dAElite a more attractive solution.

#### 6.4.4 dAElite compared to other NoCs

We compare here the cost of the dAElite to the cost of other networks on chip reported in the literature. A conclusive comparison is difficult to perform because the different NoCs support different features and furthermore most publications only report the cost of the routers while our proposal consists of an entire NoC, including the interface to IPs. dAElite compares favorably in terms of router hardware area cost but the bulk of the cost is concentrated in the NIs. Nevertheless we present here a comparison using the available data.

We have synthesized our design in TSMC 65 nm, 90 nm and 130 nm. We report the area after synthesis in Table 6.4, compared to other values of the router area reported in the literature. We do not report timing as we expect it to be more affected by other factors like length of the links. We expect dAElite to perform well in terms of operating frequency as it allows arbitrary pipelining of links and it lacks complex decision logic like arbiters.

A solution similar in concepts and functionality to dAElite is the one proposed in [BWM<sup>+</sup>09]. Same as dAElite, it makes use of a separate configuration network but it is based on a SDM scheme instead of TDM. The result is roughly 6.7 times more expensive but it offers more routing flexibility. (in SDM any of the 4 lanes of an input port can be forwarded to any of the 4 lanes of an output port, but in our TDM scheme, one TDM time slot can be forwarded only to the immediate next time slot).

### 6.5 Related work

Many NoC implementations, either connectionless or connection-oriented, have been proposed in the literature. These networks may offer both Best-Effort (BE) and Guaranteed Services (GS). Networks on chip as SPIN [ACG<sup>+</sup>03], xPipes [BB04], qNoC [BCGK04], SoCIN [ZS03], artNoC [SLB07], Quarc [MMV09] and [SHG10], implement a connectionless packet switching approach. QNoC implements quality-of-service through the means of prioritized traffic classes, but the guarantees offered are at best statistical. ArtNoC has support for multicast but only from one node at a time. Support for multicast is also provided in [MMV09, MNTJ04b] and [SHG10]. Another approach is BENoC [WCK08], which uses a bus to complement the services of the NoC. While the NoC would provide high data throughput, the bus would provide low latency messaging, multicast and broadcast. Compared to BENoC the advantage of our approach is that we can provide high-throughput multicast



Table 6.4: Cost of a dAElite router compared to other implementations

<i>16-bit 5-port router, 130 nm technology:</i>	
artnoc [SLB07] 2-flit buffers, 4 Virtual Channels	$0.060 \text{ mm}^2$
Wolkotte [WSRS05] circuit switched	$0.050 \text{ mm}^2$
Wolkotte [WSRS05] packet switched	$0.180 \text{ mm}^2$
dAElite router	$0.016 \text{ mm}^2$
<i>16-bit 4-port router:</i>	
Mango [BS05] 120 nm, 8 Virtual Channels	$0.188 \text{ mm}^2$
dAElite router 130 nm	$0.020 \text{ mm}^2$
<i>32-bit 8-port router, 130 nm technology:</i>	
Quarc [MMV09] (not full 8x8 crossbar)	$0.063 \text{ mm}^2$
dAElite router	$0.053 \text{ mm}^2$
<i>36-bit 8-port router, 130 nm technology:</i>	
SPIN [AG03] (not full 8x8 crossbar)	$0.240 \text{ mm}^2$
dAElite router	$0.057 \text{ mm}^2$
<i>5-port router, 90 nm technology:</i>	
Banerjee and Wolkotte [BWM <sup>+</sup> 09], 4 SDM lanes 16 bit/lane	$0.108 \text{ mm}^2$
dAElite router 64 bit links, 4 TDM slots	$0.016 \text{ mm}^2$
<i>32-bit 4-port router, 130 nm technology:</i>	
xpipes lite [SAC <sup>+</sup> 05], 4 stages output buffer	$0.091 \text{ mm}^2$
dAElite router	$0.020 \text{ mm}^2$

and more multicast connections operating in parallel.

Connectionless packet switching NoCs typically do not offer latency and bandwidth guarantees, thus we do not discuss them further. In the following we comment on the connection-oriented, circuit-switching NoCs, as they are similar to dAElite. Among these we give special attention to [GDR05], as it is the closest approach to ours.

Æthereal [GDR05] is a hybrid network offering both Best-Effort and Guaranteed Services. Æthereal supports three routing models, distributed routing with BE configuration, source routing with BE configuration and source routing with GS configuration. More recent studies [GH10] suggest that the BE

Network	Link sharing	Routing	Connection Setup	End-to-End Flow Cont	Connection types
DAElite	TDM	distributed	dedicated	separate wire, TDM	1-1, multicast
aelite [HSG09]	contention-free TDM	source	GS	separate wire, TDM	1-1, channel trees
TTNoC [PK08]	contention-free, pulsed	source	GS, over data network	headers	1-1, multicast
TTNoC [Sch07]	contention-free, pulsed	distributed	unspecified	none	1-1, broadcast
Aetheral w/BE [GDR05]	contention-free TDM	source/distributed	GS/BE, guaranteed	headers	1-1, multicast*
Kavalidjev [KSJW06]	VCs	source	packet, BE <sup>†</sup>	none	1-1
Wolkotte [WRS05]	SDM	distributed	separate BE	separate wire	1-1
Nostrum [MNTJ04a]	TDM, looped	unspecified <sup>‡</sup>	container <sup>‡</sup>	none	1-1, multicast
SocBUS [LWS <sup>+</sup> 02]	none	distributed	packet, BE	none	1-1

Table 6.5: Comparison with network implementations using similar concepts

\* The distributed version of Aetheral could in theory support multicast at network level, although a solution for configuring the network for this scenario was not proposed; multicast was proposed using separate connections for each target

<sup>†</sup> Guaranteed connections have preallocated VCs and setup is assumed to always succeed

<sup>‡</sup> The paper only mentions that routes are decided at run time, possibly they are stored in a distributed fashion inside the routers

<sup>‡</sup> No explicit connection setup is required, containers can be added and removed at will at runtime by any of the nodes on the route but lack of conflicts must be ensured

versions are not very cost-effective. For guaranteed services, *Æthereal* makes use of a routing model called contention-free routing in which each connection may use a link in a given timeslot. Channel trees [HCG07a] enhance the performance of this basic scheme, by allowing sharing of timeslots between channels, i.e., connections. This sharing may render invalid the service guarantees per connection, thus are not discussed further.

[RDP<sup>+</sup>05] proposed the implementation of multicast in *Æthereal* using separate connections. *dAElite* uses instead a broadcast/multicast tree to achieve the same result. Our solution is more efficient since it avoids both using separate channels inside the NI and using the link bandwidth  $n$  times, one for each of  $n$  destinations. Compared to *Æthereal*, we also use a more efficient, low-cost connection set-up mechanism. The connection state is stored inside all network elements in a distributed manner and the network configuration mechanism is centralized. Moreover, *aelite* requires a separate data connection over the network to configure the buses around the NoC. *dAElite* programs these buses through a broadcast mechanism, leading to faster configuration.

A network very similar to *aelite* is *TTNoC* [PK08] which also uses contention-free routing but claims to offer more freedom than a fixed, periodic TDM table. The network supports multicast but because source routing is used we expect a significant overhead in encoding the multicast trees in the packet headers. An earlier version of *TTNoC* [Sch07] used distributed routing, but was tied to a ring topology. This implementation supported a broadcast operation.

Another network that uses a TDM scheme to provide guaranteed bandwidth is *Nostrum* [MNTJ04a]. *Nostrum* does not have a fixed TDM wheel size, but instead, the TDM period is linked to the length of looping connections. Multicast is supported by adding more receiver nodes to a closed loop. *Nostrum* also offers BE communication using deflection routing. One disadvantage of *Nostrum* is that routing paths, and consequently multicast node sets, must be decided at design time.

The network proposed in [KSJW06] uses per-connection virtual channels (VCs) and round-robin arbitration to provide communication guarantees. VCs are in general expensive as they require buffers, multiplexers, demultiplexers and separate flow control. The number of VCs per router suggested by the authors due to cost concerns is of only 4 which may limit the number of simultaneously supported connections.

*aSOC* [LST00, LLTB03] implements the same type of static TDM schedule found in *Æthereal*, but it does not implement the actual end-to-end connections, leaving this task to the IPs.

MANGO [Bje05] is an asynchronous network implementation that uses, as [KSJW06], per-connection virtual channels. Since the network is clockless, there is no actual TDM table. Like *Æthereal*, connection setup is provided by using a Best-Effort network.

Another possibility for link sharing is SDM, used by [WSRS05]. Like our implementation, it makes use of an external network for route configuration, but it does not explicitly specify how this network is implemented. Reported configuration times are higher than those of dAElite.

Some implementations like SoCBUS [LWS<sup>+</sup>02] do not share the link between connections. This approach has a very low cost but it may result in excessive blocking.

Table 6.5 summarizes the related approaches to several aspects of the NoC implementation. One key differentiator is the type of routing employed which also has implications on the location where the connection state is stored. Source routing encodes the packet path in the header of the packet while distributed routing relies on separate routing decisions at each hop. We consider source routing to be too expensive for multicast and broadcast especially if small packets are considered, thus dAElite utilizes distributed routing.

Our proposal provides a unique set of features, namely multicast, multi-path routing, a low cost contention-free routing model and distributed routing, along with an improved performance/cost ration compared to the state of the art.

## 6.6 Conclusions

In this chapter we have proposed a network implementation which supports some of the less restrictive models that we have proposed in Chapter 2, in this way allowing better performance.

Our proposal improves upon the state of the art in terms of cost and performance. In particular, compared to *aelite*, which is the closest model we have the advantage that our proposal supports multi-path routing and multicast, it does not have a header overhead and it allows a finer-grained link bandwidth division without increasing the scheduling latency. We have achieved this by storing the slot tables in a distributed manner inside routers as well as network interfaces and by employing a lightweight and efficient configuration mechanism which also significantly improves connection set-up time.

## Chapter 7

# Bandwidth efficiency and Latency hiding

While in the previous chapters we have focused on maximizing the raw bandwidth provided by the network, in this chapter we analyze how this bandwidth is translated into services offered to the the end-user or more precisely to the IPs connected to the network.

The services provided to the IPs are built on top of the network services in a layered approach similar to the one of the OSI Protocol Stack [HG09, HG10]. We illustrate this in Figure 7.1. The network provides pure transport of data between different locations on the chip, but it is not concerned with the meaning of this data. On the other hand IPs expect services with memory operation semantics, i.e., memory read and write operations. In-between is found a layer which translates the memory operations into messages.

The modules performing this translation are called (Network Interface) shells. This is a common feature of aelite and our network proposed in Chapter 6, dAElite. Connections are set up between a pair of Network Interface shells, one translating the memory request to a network message, the other translating the message back into a memory request. Connections have a long lifetime, a connection is used to perform many memory operations before it is torn down.

Optionally, in-between NI-shells and IPs, a demultiplexer bus sends the request to the NI shell corresponding to the desired destination. This bus exists if an IP can communicate to multiple destinations over the network. At the other side of the network, an optional multiplexer bus arbitrates between incoming request from different sources.

In this chapter we focus on the translation layer between IPs and the network.

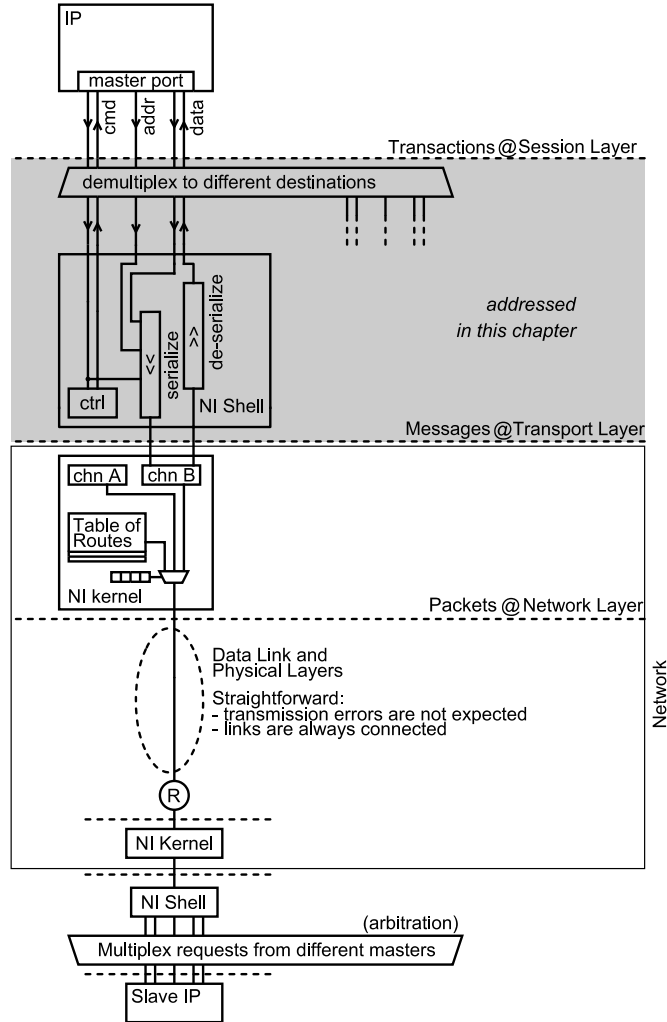


Figure 7.1: Correspondence of OSI layers to the Æthereal implementation.

We propose three optimizations that target the translation process taking place inside the NI shells. Two of the optimizations are transparent from the point of view of the IP while another one requires the explicit changes to the software.

The first optimization consists of performing automatic write coalescing. Automatic write coalescing improves network bandwidth utilization, as it will be shown in Section 7.1. The second optimization, which we detail in Section 7.2, consists of a mechanism to perform posted writes while preserving memory

consistency. The third provides a hardware mechanism to support software prefetch in an IP where this functionality was absent. The prefetch operation is also supported at the level of NI shells and it is presented in Section 7.3.

We also demonstrate interoperability between different bus standards at the different ends of the network. Network interface shells translate memory requests received from a master IP into an internal common message format. After traversing the network, the messages are translated by network interface shells into the specific format (bus protocol) understood by the slave IP, regardless of whether this protocol is the same one used by the master IP or not.

All our proposals were implemented in VHDL and tested in FPGA. Our prototype used an instance of the *Æthereal* network with our own custom designed network shells a Microblaze processor as a master IP and a memory as a slave IP. Interfacing was performed on the busses using the Processor Local Bus (PLB) and Fast Simplex Link (FSL) protocols. Section 7.4 presents the experimental results showing the performance improvement of each of our proposed techniques.

## 7.1 Write coalescing

Memory transactions, encoded as messages, traverse the network in a serialized fashion, with headers, addresses and data sharing the same bandwidth. Longer messages, with multiple words of data for a single header/address pair would thus have better payload efficiency.

Burst transactions on the bus side correspond to such messages inside the network, however, not all IPs have the capability of generating burst transactions. The instruction set of the MicroBlaze soft-core does not provide an instruction to write to memory more than one word at a time, and this is true for many other simple IPs also.

We automatically identify sequences of write operations to consecutive addresses and combine them into a single message for the purpose of traversing the network. At the destination NI shell these messages can then be split again into individual write operations or optionally they can be served directly to the destination bus if burst transactions are supported.

We perform this write coalescing as long as the addresses are consecutive, the burst length has not reached the maximum value, 32 for the message format we are currently using, and there is data in the outgoing network queue. This last condition is to ensure that we are not unnecessarily delaying messages and

to avoid deadlock.

A diagram of the NI shell we implemented is found in Figure 7.2. The shell uses several independent queues to store the data, the address and the headers. Data is copied from the PLB bus to the send queue whenever a write transaction is accepted, new headers and address are copied to the header queue whenever a transaction cannot be merged with the previous transaction or the network is idle and the current transaction can be processed immediately.

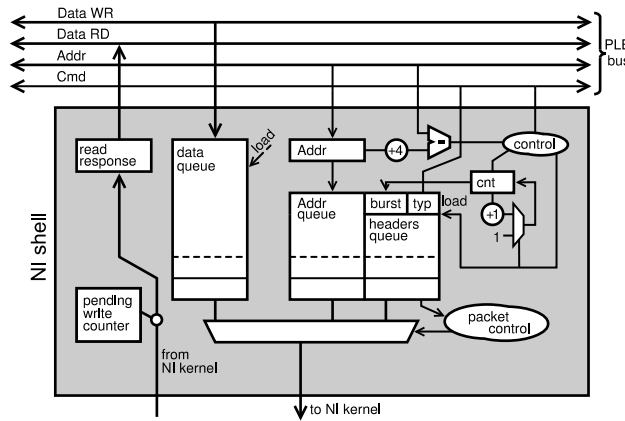


Figure 7.2: NI shell supporting burst write.

It would also be possible to implement the same functionality for read transactions, however the Microblaze processor always stalls until a read transaction is complete thus making this feature useless.

For comparison, the NI shell without burst or posted write support (Figure 7.3) consists of a simple serializer shifter. The entire request message is generated in a single cycle when accepting a request from the bus and is sent to the network word-by-word in the following cycles. Because all requests are blocking, additional queues would not provide any benefit.

## 7.2 Posted writes and memory consistency

Memory consistency is a term used to describe the expected system behavior with regard to the order in which memory writes are visible to the different IPs in the system. In general a stricter memory model provides more guarantees regarding the order of memory operations thus making the programmer's job



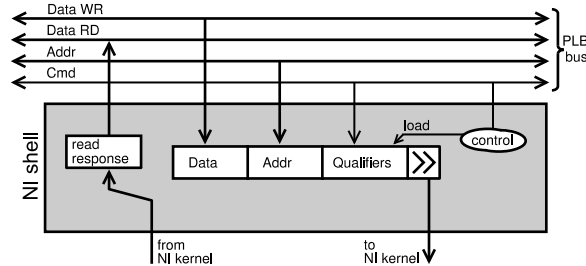


Figure 7.3: NI shell without burst or posted write support.

easier, but makes the hardware implementation more difficult and less efficient.

The strictest consistency model in use is the sequential consistency [Lam79], which requires that the memory operations of each individual processor, as seen by the other processors in the system appear to execute in the order of specified by the processor's program. Any particular interleaving between the instructions of different programs is allowed, but the same interleaving shall be seen by all processors or processes. This is essentially the same result that can be expected from multiple threads running on a single processor, and studies indicate that this is the behavior programmers expect from the machine [AG96, Hi198].

We first present a basic hardware implementation that would provide sequential consistency, then show how the requirements can be relaxed to allow performance optimizations. We assume from the beginning an architecture where memory requests, both read and writes can be pipelined, but the ordering of requests, even between reads and writes is preserved within the pipeline. It must be mentioned that allowing reads to bypass writes is sometimes accepted as an optimization [Goo89], reads being considered more important, as the reader process was likely stalled waiting for data. Although we do not accept reordering of normal read operations with respect to write operations, we allow it for prefetch reads as it will be explained in the following section.

Despite the fact that our architecture does not employ caches, and the system seems to maintain ordering of requests, consistency problems may still arise. Consider the following scenario involving a transfer between a producer and a consumer of data, represented in Figure 7.4. The producer (node A) generates data items and places them in some memory location, for example in external memory. Upon completion, it signals to node B that the data is ready to be processed.

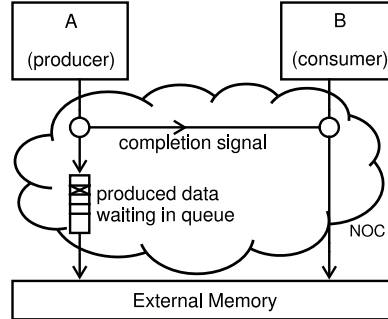


Figure 7.4: Consistency issues raised by signals arriving at different destinations with different delays.

Because the NoC allows messages with different destinations to travel independent of each other, it may be possible for the confirmation message from A to arrive while the data is still queued waiting to be written to the memory [LVG10]. An attempt by B to read the data through its own connection to the memory would return stale values. Not only it is easy to see that this produces an erroneous program behavior, but it does that by breaking the assumptions of sequentiality, which required that the confirmation message, for example a write operation to a specific flag in memory, would be seen by B strictly after the write operation to main memory was completed.

One possible solution would be that all write operations need to be confirmed (non-posted writes), preventing A from sending a message to B before the previous operation has been completed. Unfortunately, this would completely prevent pipelining, resulting in a severe performance penalty.

Our solution consists of performing posted writes (write operations which are acknowledged at the source without waiting for confirmation from the target), but keeping track in the NI shell of which write operations have been confirmed and which have not been. Future writes to different destinations will be stalled until all pending writes have been confirmed, in other words, consecutive writes to the same destination are pipelined like posted writes but writes to different destinations behave like non-posted ones.

Let us analyze why this ensures sequentiality. The sequential consistency model required that from each process's point of view, all memory accesses in the system seem to take place in the same order, with an arbitrary interleaving of accesses belonging to different processes, but maintaining the program order for accesses of each individual process. This order is not necessarily the order

given by the physical time of issue of each access, but in our case it can be chosen as the order of physical time of completion of each access (the physical read or write to each memory). Because we did not allow the reordering of write operations in each pipeline and all messages between two nodes travel on the same path, writes to one memory from one specific processor will occur in program order. Because our system waits for completion confirmation when switching between different targets, writes to memory B that occur in the program after writes to memory A will also take place physically later in time. It is necessary for the same to happen for read requests so our mechanism also enforces this.

It is possible to further relax these restrictions to allow higher system performance. For example when one memory is not read by any other process, like it might be the case of a video frame-buffer, it is not necessary to order the accesses to that memory with respect to accesses to other memories. It is also possible to emulate the behavior of other consistency models by only partially connecting the command signals used to block some memory accesses until accesses to other modules have completed and mapping synchronization variables to specific memories, Figure 7.5. We can for example implement the weak consistency model [DSB98] by mapping synchronization variables in one memory and enforcing sequentiality between that memory and each of the data memories, but not among data memories. A consistency model similar to the Release/Acquire model [GLL<sup>+</sup>90] could be implementing by splitting the synchronization memory in two separate memories and enforcing only one way ordering between accesses to these memories and the data memories, for example an access to Acquire must complete before an access to Data, but not the other way around.

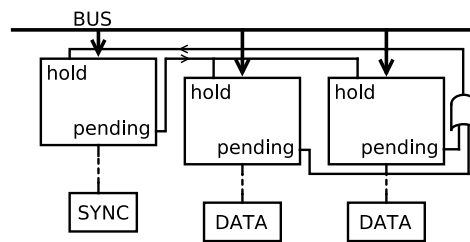


Figure 7.5: Emulation of a weak consistency model.

### 7.3 Software prefetch

The previous proposals are transparent from the point of view of the software developer. Except for the variation in performance, the system behaves no differently from a system with only local memory, even the introduction of the NoC between an IP and its memory is completely transparent. The following proposal however introduces a feature that needs to be explicitly used by the programmer, and in some cases requires a significant rewrite of the software.

Networks on chip and large scale interconnects in general have a higher latency than back-to-back connections. Given that, it is natural that we look for ways to cope with that latency. While posted writes provide an efficient method of dealing with write transactions, for read transactions we do not have a similar solution. Processors providing out-of-order execution mitigate the problem to some extent by allowing other instructions to continue while data is being fetched from memory. This type of approach though is expensive and not often predictable.

In our system we opt for introducing an explicit command to bring data from memory some time in advance before actually being needed (prefetch). Note that, unlike other prefetch implementations, the data is not used to update a local cache from which it can be later read using a normal read instruction, but instead is deposited in a queue from where it needs to be explicitly read by the program. The technique can be seen as a software split transaction, where the request for data is decoupled from the receiving of data. Example code is provided in Figure 7.6.

```
// original loop
1 for (i=0; i<n; i++)
2 {
3     v=a[i];
4     // use value v
5 }

// modified loop
6 prefetch(a[0]);
7 for (i=0; i<n; i++)
8 {
9     if (i<n-1) prefetch(a[i+1]);
10    v=read_prefetch_fifo();
11    // use value v
12 }
```

Figure 7.6: Example prefetch loop compared to original code.

In the modified code, one or more data items are requested prior to entering the

main processing loop (line 6). In each loop iteration, more data is requested in advance (line 9) with care being taken to not exceed the limits of the actual input. Data from the previous requests which should have already arrived in the buffers is then retrieved in line 10.

The hardware necessary to support the prefetch mechanism consists of an NI shell connected to the FSL bus of the MicroBlaze processor (Figure 7.8).

The shell behaves as a processing FIFO, accepting at the input port memory read requests and delivering at the output port the retrieved data. The shell accepts additional commands to configure the size of burst reads.

All prefetch operations are under explicit control of the program, which may also have to ensure ordering with respect to the normal read and write operations. Currently the code must be manually edited by the software developer, which is also what we did in our experiments. Although in principle it might be possible to offload this task to the compiler for example, this task is far from trivial and is complicated by consistency issues especially in multi-core systems [SKKC09].

All our proposals were implemented and tested in FPGA. The hardware cost of the implemented shells relative to the size of the entire system is presented in Figure 7.7. In our test system, the prefetch module was connected to a dedicated NI kernel, thus doubling the size of the interconnect, however in practice this would not be necessary. The optimized shell implements both the burst and posted writes.

## 7.4 Experimental Results

We have performed our tests on three similar systems, all having the structure presented in Figure 7.8. For the main PLB target we have substituted three different NI shells, one performing only non-posted operations, one capable of performing posted writes with the described safety mechanism, and one performing both burst coalescing and posted writes. The FSL interface was always present, but since its use is always explicit, we specify through the MicroBlaze program whether it should be used or not.

Our tests involved only one processor, however we simulate the effects of having multiple processors by allocating only a fraction of the link bandwidth inside the network. For a fair comparison, in the tests where both the FSL and PLB link are used, we restrict the total bandwidth for both links to the bandwidth offered to the PLB alone in the non-prefetch scenario. Another

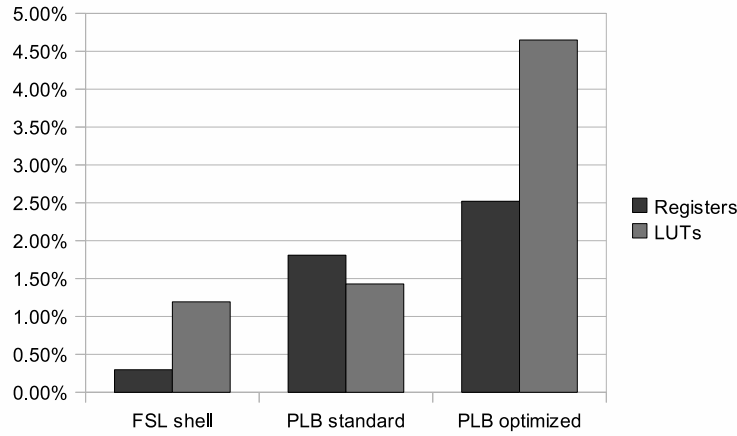


Figure 7.7: Hardware cost of the shells in FPGA implementation as a percentage of the cost of the entire network.

approach would have been to use a combined PLB and FSL shell that would multiplex prefetch reads and standard memory transactions over the same network connection thus sharing the bandwidth of that connection.

The most efficient way of distributing the bandwidth between the PLB and FSL connection was found using exhaustive search with increments of 25%. We also perform an additional test, where two separate full-bandwidth links are provided, one for each link (marked B in the result graphs, Figures 7.9 - 7.14).

For the software we have chosen several benchmarks, ranging from synthetic to real applications. In short, these are: read and write loops, several kernels with a relatively high amount on memory activity chosen from the Livermore Loops [McM86], and a JPEG decoding application.

Our first set of tests consists of a read and a write loop for an array of 16 K words. Due to their intensive use of memory transactions they show the highest performance variations among our tests. The results are presented in Figure 7.9. As the number of processors in the system is increased, there are two factors causing performance degradation: an increase in latency and a reduction in the available bandwidth. The posted write and the prefetch read tests, which are largely immune to the increase in latency provide a good indication of the point when bandwidth becomes the limiting factor. The use of burst read and write operations alleviates the effect of the bandwidth reduction.

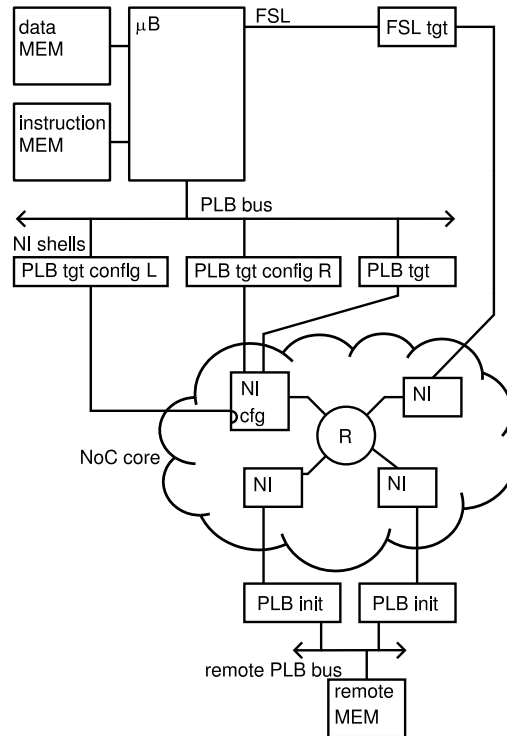


Figure 7.8: Test setup: one MicroBlaze core is connected through two separate channels, one on the PLB bus and one on an FSL link to a remote memory.

Not using posted write and prefetch read incurs a large penalty even for the one processor configuration.

The results show that the largest performance gain is obtained by simultaneously using prefetch reads and posted writes regardless of the available bandwidth. This follows directly from the fact that the network latency is added to every memory operation. Using the burst facility of both read and write operations shows a benefit mainly when the bandwidth is severely limited.

Some result artifacts in the tests dominated by the read latency are produced by the read loop locking onto the pattern of allocated time slots, preventing the use of some of the slots, even when they are allocated. This results in an above trend deterioration in performance of the 8 processors test case, which produces the same result as the 16 processors case.

The Livermore Loops are small kernels that were used to evaluate the perfor-

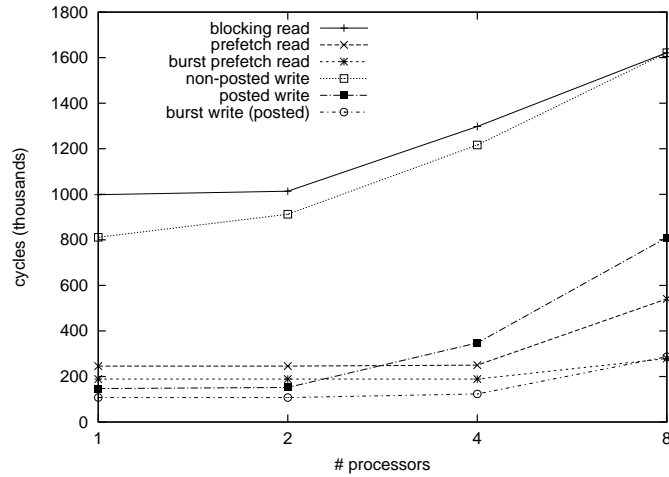


Figure 7.9: Performance of Read and Write tests under different setups.

mance of supercomputers. They are representative for scientific applications but we have chosen to use them as they provide a variety of memory access patterns. The versions we employed were translated in C and were set to use only integer operands. The loops had to be manually modified to perform software prefetch, and thus we only used four of them in our tests, more specifically kernels 1 (hydro), 6 (linear recurrence), 12 (first difference), and 21 (matrix multiplication).

The results of the Livermore Loops tests are shown in Figures 7.10-7.13. While the latency hiding techniques provide a significant advantage, the burst optimization does not always produce an improvement as it depends on the patterns of memory access. The numerical results for all tests can be found in Table 7.1.

The JPEG application is a complex program with a high computation-to-communication ratio. It requests data from an external memory, performs calculations on the retrieved data and writes back the result to the external memory in small bursts which are essentially memory copy operations on the already decoded data. The read operations are almost always sequential with the exception of a few initial headers, while the write operations follow an access pattern characteristic for accessing a sub-matrix out of a larger matrix.

The results are presented in Figure 7.14. The difference is more pronounced when the available bandwidth is severely limited as it is the case with the 8



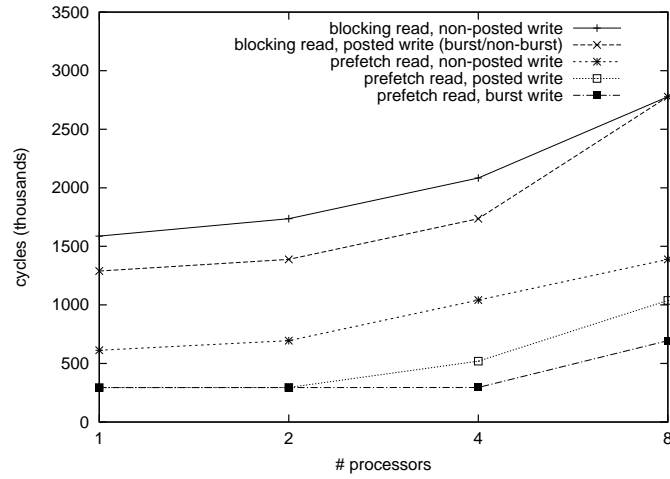


Figure 7.10: LL kernel 1.

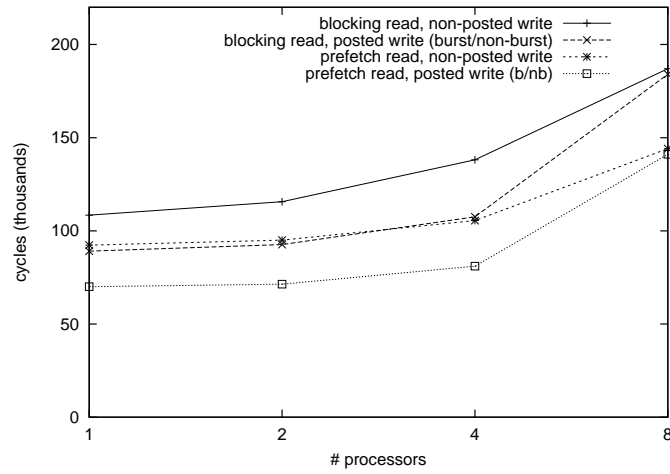


Figure 7.11: LL kernel 6.

processors scenario. The improvement in the case of the JPEG application is a modest 6 to 12%, however this represents an increase in the performance of the entire application, while the optimization targeted a single component of the system.

Figure 7.15 shows the average performance increase over all bandwidths

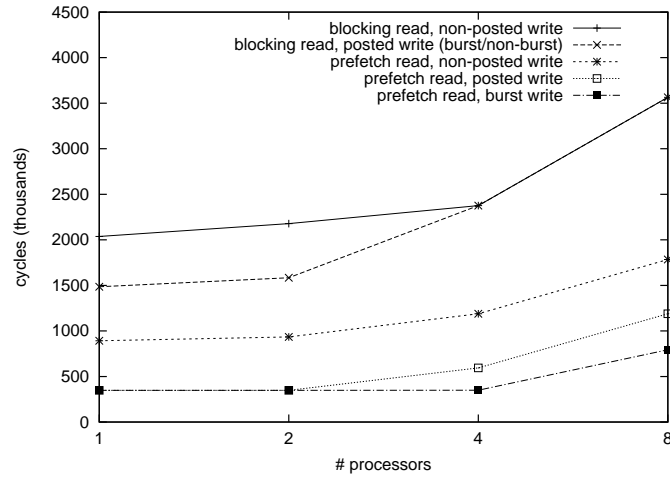


Figure 7.12: LL kernel 12.

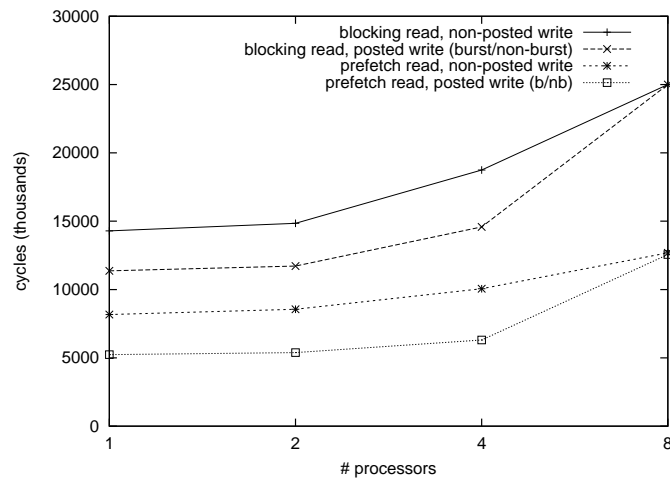


Figure 7.13: LL kernel 21.

obtained in all applications. The performance increase due to particular optimizations is represented separately. The performance increase provided by the posted writes is presented in two separate scenarios, when prefetch read is not used and when it is. The performance of burst posted writes is presented in comparison to normal posted writes. The overall speedup is the speedup for the

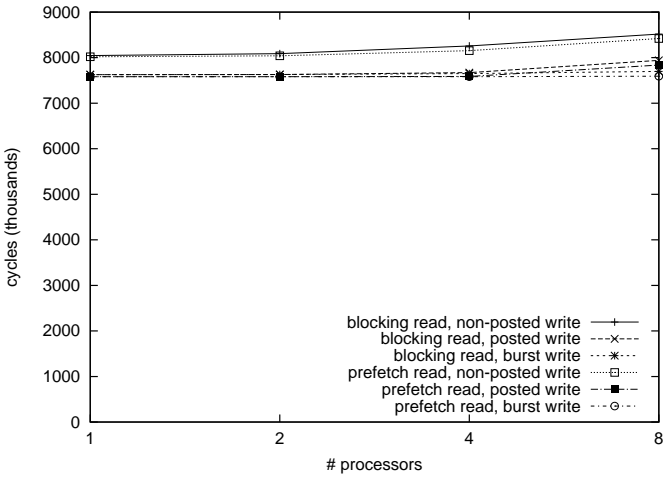


Figure 7.14: JPEG.

entire system with all optimizations enabled. Numerical results are presented in Table 7.1.

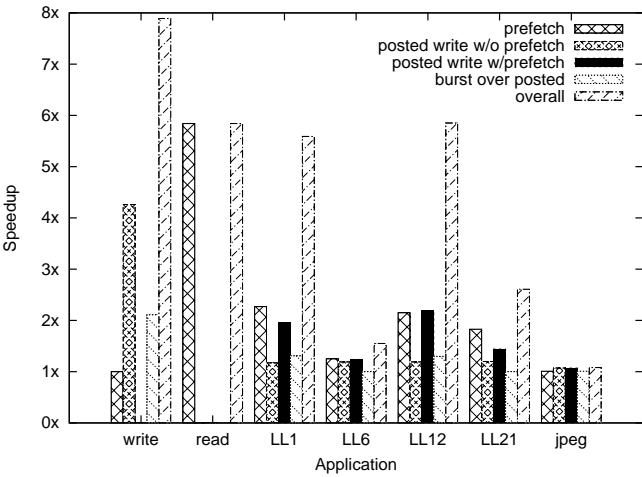


Figure 7.15: Average speedup across all bandwidths obtained with the different techniques in all applications.

test	prefetch read over baseline	posted wr. no prefetch over baseline	pfetch & posted over pfetch only	burst in addit. to prefetch& posted	all techniques over baseline
write-1p	-	5.511	-	1.369	7.542
write-2p	-	6.007	-	1.412	8.484
write-4p	-	3.505	-	2.820	9.883
write-8p	-	2.003	-	2.824	5.655
read-1p	5.297	-	-	-	5.297
read-2p	5.379	-	-	-	5.379
read-4p	6.885	-	-	-	6.885
read-8p	5.817	-	-	-	5.817
LL1-1p	2.590	1.231	2.081	-	5.388
LL1-2p	2.498	1.250	2.359	-	5.892
LL1-4p	2.001	1.200	2.005	1.762	7.068
LL1-8p	2.000	-	1.338	1.494	3.996
LL6-1p	1.174	1.217	1.319	-	1.550
LL6-2p	1.217	1.249	1.331	-	1.620
LL6-4p	1.309	1.285	1.303	-	1.704
LL6-8p	1.298	1.016	1.021	-	1.325
LL12-1p	2.284	1.371	2.560	-	5.846
LL12-2p	2.330	1.375	2.683	-	6.252
LL12-4p	1.998	-	1.999	1.707	6.817
LL12-8p	1.997	-	1.501	1.499	4.494
LL21-1p	1.750	1.258	1.559	-	2.728
LL21-2p	1.735	1.267	1.590	-	2.758
LL21-4p	1.865	1.286	1.593	-	2.971
LL21-8p	1.971	-	1.010	-	1.990
jpeg-1p	1.003	1.055	1.058	-	1.061
jpeg-2p	1.006	1.060	1.060	-	1.066
jpeg-4p	1.012	1.076	1.075	-	1.089
jpeg-8p	1.011	1.073	1.074	1.032	1.122

Table 7.1: Speedup obtained using the different techniques for the considered applications and bandwidths

## 7.5 Related Work

Many network on chip implementations already exist, some providing mature flows with the possibility of generating a hardware description of the entire infrastructure [GDR05, BCGK04, BB04, PAM<sup>+</sup>07]. The authors often provide

complete solutions covering all aspects of interconnect design including the interface to the IPs.

Many studies also exist covering the optimization of different aspects of the interconnect, like routing [HM04, FMLD07] and network topology [SCK05, OM05], however we have found that little effort was dedicated to optimizing the interfaces between traditional IPs, unaware of the existence of the NoC and the NoC itself.

Of the literature dedicated specifically to network interfaces we mention [RDP<sup>+</sup>05] which is a precursor of the architecture we use in our current research proposed separating the networking function of the NI from the actual interface to the IP. The implied benefit is the possibility of easily reusing part of the design when developing interfaces to other types of IPs. The same idea is also present in [HJK04].

Options for connecting IPs to NoC are explored by [BM03]. The work considers solutions based on both software and hardware, with an approach focused on modifying the IP building wrappers around it. By comparison we choose to leave the IP unaltered and only interface with it based on its existing connections to standard buses.

Wrappers are also used by [SBB<sup>+</sup>06], with the advantage of both hiding the implementation details of the interconnect from the IP and avoiding modifications of the IPs and the network internals.

None of the works previously mentioned suggests performance optimizations at the level of the NI.

In the domain of high-performance cache-enabled processors, write coalescing is a function commonly performed by write buffers [SC97] or by the cache itself, as in the case of write-back caches. Prefetching has also been studied extensively [fCB94]. Our work is focused on less costly solutions which do not assume the presence of caches and integrate the functionality of the write buffer into the communication infrastructure.

Despite the fact that the optimizations presented in this paper are generally well known, we believe to be the first to propose and analyze them in the context of networks on chip.

## 7.6 Conclusions

In this chapter we have studied optimizations that target the process of translating the requests coming from IPs into network packets. This translation layer allows the IPs to perform requests using memory transaction semantics while being unaware how these requests travel to their destination and how responses return.

We have found that while the latency introduced by the network-on-chip and the bandwidth limitations due to sharing resources between multiple processors can reduce performance, using simple optimizations can restore the lost performance. The optimizations that we studied here consist of latency hiding techniques, namely read prefetch and posted write and a bandwidth optimization technique: write coalescing. We have found that prefetch read and posted write are especially important for applications with a high communication-to-computation ratio while burst transfers can offer an additional performance benefit when the available link bandwidth is severely constrained.

## Chapter 8

### Conclusions

In this chapter we explain the position this thesis occupies in the context of NoC research. Section 8.1 explains the main objectives of our research and summarizes our findings, Section 8.2 indicates our contributions, Section 8.3 summarizes the contents of each chapter and Section 8.4 presents future directions of research.

#### 8.1 Objectives of research

In the electronics industry of the last several decades, miniaturization has an exponential decrease in the manufacturing cost per transistor. This has allowed an increase in the complexity and computational power especially in the case of digital circuits. To deal with this complexity it was also necessary to decrease the design cost per transistor. This was achieved thorough the use of automated design tools and improved design methodologies.

One of the areas these automated tools target is the generation of an on-chip interconnect. This is becoming more and more important as the number of on-chip components that need to be connected to each other increases. The work presented in this thesis allows us to evaluate different choices in the design of NoCs which may be useful in the context of automated design tools. We also develop and implement a series of algorithms to allocate network resources and we provide a template for the NoC hardware.

In our analysis of NoC choices we found that topology is not in general a limiting factor (a performance loss of only 8%, as shown in Figure 2.19), with the simple mesh and torus topologies presenting very good performance. The

ring and Spidergon topologies have lower performance. We have found the discretization of bandwidth division for link sharing purposes to introduce a non-negligible performance loss (18%). On the other hand we have found the performance loss introduced by the contention-free routing model to be small (5%) especially when considering the cost benefit which is a factor of 10 to 20 [GH10]. In terms of allocation efficiency, the difference between using global optimization algorithms and heuristics is small, as is the impact of enforcing in-order delivery.

Based on these findings, we have designed a NoC template that offers a good performance-cost ratio. This network template can be used to create network instances with any desired topology and connecting a customizable number of IPs. We have also developed improved hardware for the layer connecting IPs to the NoC. We have shown these improvements to mitigate the effect of latency and bandwidth limitations on the performance of applications running on the IPs connected to the network.

The largest part of the thesis though is dedicated to algorithms for the allocation of network resources to the applications requesting them. Our algorithms improve upon the state of the art in terms of the efficiency of allocating resources. For several algorithms we have offered a proof of their optimality. We have shown that allocation can be performed at run-time with minimal computation and memory requirements. We have also proposed hardware accelerators that allow an even faster computation of the allocation.

## 8.2 Contributions

In this section we discuss the major contributions of this thesis.

- **We propose network models** that allow us to determine achievable network performance both in an ideal scenario and under the constraints of real hardware. We evaluate these models for a wide range of network topologies and under several types of traffic. We determine the effect of various design choices including the contention-free routing model which is supported by low-cost network implementations. Some of the models are supported by optimal, linear-programming based allocators that allow establishing bounds on the performance of any type of interconnect.
- **We design allocation algorithms**, some for generic networks and some targeting specifically the contention-free routing model. Our algorithms



perform spatial as well as temporal allocation. Spatial allocation consists of selecting a specific path, or in the case of multi-path allocation several paths, that a specific connection should use through the network. Temporal allocation selects the slots in the TDM schedule to be used by each connection. We provide algorithms both for design time and run-time allocation. Our algorithms improve upon the state-of-the art in terms of performance and features.

- **We provide hardware implementations** of the network on chip and the interfaces between the network on chip and IPs. Our NoC proposal is circuit-switched and uses the contention-free routing model. It compares favorably to other proposals presented in the literature. Our interface to the IPs presents several optimizations: write coalescing, posted writes and read prefetch. We also present hardware acceleration modules that are useful for increasing the performance of the online allocation algorithm.

### 8.3 Thesis Summary

We began this thesis by introducing the problems related to the design of on-chip interconnects and presenting traditional as well as modern solutions.

In Chapter 2 we proposed a series of network models which we then used to evaluate the performance implications of several interconnect design choices. We have also offered a mathematical solution based on linear programming to the allocation problem in some of these models.

In Chapter 3 we proposed path allocation algorithms for the more restrictive models introduced in Chapter 2. Where possible we made use of optimal algorithms and we have demonstrated their optimality.

In Chapter 4 we proposed algorithms for slot selection. These algorithms are used to improve or guarantee a certain latency bound for each communication channel. We have proven these algorithms to be optimal and we have compared their performance to the performance of previously proposed algorithms. We have also looked into the effect of slot selection on actual application performance.

In Chapter 5 we demonstrated route and slot selection performed at run time. We have provided a memory and computationally efficient implementation of the allocation algorithm and we have implemented hardware acceleration of the algorithm.

In Chapter 6 we proposed a hardware implementation of a network on chip based on the contention-free routing model. Our network offers multi-path routing and multicast, avoids header overhead, and has very low configuration time. We evaluated our proposal in FPGA and ASIC synthesis and we found it to compare favorably in terms of hardware area and speed to other networks reported in the literature.

In Chapter 7 we looked at how the raw communication services provided by the network can be translated into transaction-level services offered to the IPs. We presented optimizations regarding bandwidth use and latency hiding techniques and we analyzed the overall effect on the execution time of real applications.

## 8.4 Future Directions of Research

We present here new potential avenues for exploration having this work as a starting point. In this thesis we have discussed many aspects of an on-chip interconnect. It would be possible to combine some of the techniques described here to achieve further improvements.

For example it may be possible to combine multi-path routing with run-time allocation. The multi-path allocation algorithms have polynomial running time which is an advantage, although they may be slower on small topologies and have slightly higher memory requirements. It would also be possible to evaluate the more complex slot allocation algorithms in the context of run-time allocation.

It would be possible to further improve the allocation algorithms, for example using a rip-reroute technique which was not one of the targets of this study. It would also be possible to explore different path orderings during offline allocation.

It would be possible to combine our NoC hardware implementation supporting fast connection set-up with a hardware accelerated allocation algorithm to create a network which provides on-demand connections with very low latency. This would allow networks based on the contention-free routing model with a centralized allocator to support a wider range of applications.

Most importantly, the network and algorithms should be put to actual use, because through actual use we discover the limitations and directions in which we need to improve.

## Appendix A

### Sample LP model for a 2x2 mesh network

```
set I;  
/* R-R links */  
  
set R;  
/* routers */  
  
set O;  
/* NI io */  
  
set J;  
/* connections */  
  
param a{j in J, r in R};  
/* comm incidence at routers */  
  
param b{j in J, o in O};  
/* comm incidence at NI */  
  
param c{j in J};  
/* comm weight */  
  
param d{i in I, r in R};  
/* link incidence at routers */  
  
var x >= 0;  
var y{j in J} integer;
```

```

var z{i in I, j in J} integer;
maximize tput: x;

s.t. wt{j in J}: y[j] >= x*c[j];
s.t. pv{i in I, j in J}: z[i,j] >= 0;
s.t. bwl{i in I}: sum{j in J} z[i,j] <= 16.0;
s.t. nbw{o in O}: sum{j in J} y[j]*b[j,o] <= 16.0;
s.t. csv{r in R, j in J}: a[j,r] * y[j] + sum{i in I} d[i,r] * z[i,j] = 0;
/* conservation of each flow at each router */

data;
# *** this begins the data section

set I := R0to1 R0to2 R1to0 R1to3 R2to0 R2to3 R3to1 R3to2;
set R := R0 R1 R2 R3;
set O := NI0 NI1 NI2 NI3 NI4 NI5 NI6 NI7 NO0 NO1 NO2 NO3 NO4 NO5
NO6 NO7;
set J := C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16
C17
C18 C19 C20 C21 C22 C23;

param a : R0 R1 R2 R3 :=
C0  1  -1  0  0
C1  -1  1  0  0
C2  1  0  -1  0
C3  -1  0  1  0
C4  0  1  -1  0
C5  0  -1  1  0
C6  0  0  0  0
C7  0  0  0  0
C8  -1  0  1  0
C9  1  0  -1  0
C10 -1  0  1  0
C11  1  0  -1  0
C12  0  0  0  0
C13  0  0  0  0
C14  0  0  -1  1
C15  0  0  1  -1
C16 -1  0  1  0
C17  1  0  -1  0
C18  0  1  -1  0
C19  0  -1  1  0
C20  0  0  0  0
C21  0  0  0  0
C22  0  0  1  -1

```

```

C23  0  0  -1  1
;
param b : NI0 NI1 NI2 NI3 NI4 NI5 NI6 NI7 NO0 NO1 NO2 NO3
        NO4 NO5 NO6 NO7 :=
C0    1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0
C1    0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0
C2    0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
C3    0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
C4    0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0
C5    0  0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0
C6    0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0
C7    0  0  1  0  0  0  0  0  0  0  0  0  1  0  0  0  0
C8    0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  0
C9    0  1  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
C10   0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0
C11   0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
C12   0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  1
C13   0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0
C14   0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0
C15   0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  1
C16   0  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0
C17   1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
C18   0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0
C19   0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  0
C20   0  0  0  0  1  0  0  0  0  0  0  0  0  0  1  0  0
C21   0  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0  0
C22   0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1
C23   0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0
;
param c :=
C0    0.686
C1    0.001
C2    1.681
C3    0.001
C4    0.625
C5    0.001
C6    1.008
C7    0.001
C8    1.549
C9    0.001
C10   0.620
C11   0.001
C12   1.553
C13   0.001
C14   1.012

```

```
C15  0.001
C16  0.915
C17  0.001
C18  0.752
C19  0.001
C20  1.016
C21  0.001
C22  1.333
C23  0.001
;
param d :      R0 R1 R2 R3 :=
R0to1      -1  1  0  0
R0to2      -1  0  1  0
R1to0       1 -1  0  0
R1to3       0 -1  0  1
R2to0       1  0 -1  0
R2to3       0  0 -1  1
R3to1       0  1  0 -1
R3to2       0  0  1 -1
;
end
```

## Appendix B

### Example configuration of the proposed model

```
/* the mapped address of the configuration shell */  
volatile int * cfg= (volatile int *) 0x80000000;  
volatile int * mem0= (volatile int *) 0x90000000;  
  
const int SETUP=4;  
  
const int I0= MSG | 0;  
const int I1= MSG | 4;  
const int I2= MSG | 8;  
  
const int O0= MSG | 0;  
const int O1= MSG | 1;  
const int O2= MSG | 2;  
  
const int R00= MSG | 0;  
const int R01= MSG | 1;  
const int R10= MSG | 2;  
const int R11= MSG | 3;  
  
const int NI00= NI_BASE | 0;  
const int NI01= NI_BASE | 1;  
const int NI10= NI_BASE | 2;  
const int NI11= NI_BASE | 3;  
  
const int CMD_SETUPCHANNEL=4;  
const int CMD_READCREDITS=6;  
const int CMD_WRITECONFIG=10;
```

```

const int CMD_WRITECREDITS=14;
const int ENDOFPACKET=8;

inline int conf(int a,int b,int c,int d)
{
    return a+(b<<7)+(c<<14)+(d<<21);
}

/* this function is declared inline,
 * it will translate into 3 writes to
 * the configuration port consisting
 * preferably of constant values */
inline void writecfg(int ni, unsigned int adr, unsigned int val)
{
    /* each configuration word consists of 37/6=7 parts
     * a packet consists of header=0, address, data
     *
     * a write transaction consists of 3 words
     * a header with a value of 0, the address and data
     *
     * it takes 24 cycles to serialize a write operation
     * to a remote configuration port, 2 such writes
     * are required for configuring one channel
     */

    cfg[0]=conf(CMD_WRITECONFIG, ni, MSG, MSG);
    cfg[0]=conf(MSG, MSG, MSG, MSG);

    cfg[0]=conf(MSG,
        /* address, start with MSB*/
        MSG,
        MSG | ((adr>>29) & 0x3f),
        MSG | ((adr>>23) & 0x3f)
    );

    cfg[0]=conf(
        MSG | ((adr>>17) & 0x3f),
        MSG | ((adr>>11) & 0x3f),
        MSG | ((adr>>5) & 0x3f),
        MSG | ((adr<<1) & 0x3f)
    );

    cfg[0]=conf(
        /* data, start with MSB */
        MSG | ((val>>31) & 0x1),

```



```

        MSG | ( (val>>25) & 0x3f) ,
        MSG | ( (val>>19) & 0x3f) ,
        MSG | ( (val>>13) & 0x3f)
    );

    cfg[0]=conf(
        MSG | ( (val>> 7) & 0x3f) ,
        MSG | ( (val>> 1) & 0x3f) ,
        MSG | ( ( (val<< 5)+31) & 0x3f) ,
        ENDOFPACKET
    );
}

void setup_connection_1 (
{
    /* the channel setup command, followed by the slot table
    * as two-6-bit words (the slot table has 8 entries),
    * followed by one padding (0) 6-bit word
    *
    * this identifies a path using slot 6 at destination
    */
    cfg[0]=conf(CMD_SETUPCHANNEL, MSG+1,MSG,0);

    /* the path starting with the destination NI and going
    * backwards to the source NI
    * two hops can be configured per 32-bit word
    * (4 6-bit configuration words)
    *
    * the route used is NI00 - R00 - R10 - R11 - NI11
    */
    cfg[0]=conf(NI11, MSG | 4,0,0);
    cfg[0]=conf(R11, I1 | O2, R10, I0 | O1);
    cfg[0]=conf(R00, I2 | O1, NI00, MSG | 0);

    /* the response channel
    * a hardware cooldown timer inside the
    * configuration shell will make sure that
    * routers have enough time to process
    * one configuration request before
    * the second one begins
    */
    cfg[0]=conf(CMD_SETUPCHANNEL,MSG+1,MSG,0);
    cfg[0]=conf(NI00, MSG | 4,0,0);
    cfg[0]=conf(R00, I1 | O2, R10, I1 | O0);
    cfg[0]=conf(R11, I2 | O1, NI11, MSG | 0);
}

```

```

/* provide credits to the channel to allow it to send data
 * we need to identify the NI which has the channel
 * the number of the channel and the number of credits
 *
 * for simplicity, each of these pieces of information
 * is contained in one configuration word
 *
 * both the request and response channel need to be
 * initialized
 */
cfg[0]=conf(CMD_WRITECREDITS, NI00, MSG ,MSG | 15);
cfg[0]=conf(CMD_WRITECREDITS, NI11, MSG ,MSG | 15);
/* the same command can be used to write status flags
 * for example we could disable a channel using the
 * following command
 */
//cfg[0]=conf(CMD_WRITECREDITS, NI00, MSG — 4,MSG — 2); disable
channel

/* credits can also be read back, for example when we
 * intend to tear-down a connection we should first make
 * sure that all packets have drained from the given channel
 */
cfg[0]=conf(CMD_READCREDITS, NI00, MSG ,ENDOFPACKET);
u=cfg[0];
}

void setup_connection_2( )
{
/* forward path:
 * NI00(slot 3)
 * -> R00(slot 4)
 * -> R10(slot 5)
 * -> NI10(slot 6)
 */
cfg[0]=conf(CMD_SETUPCHANNEL, MSG | 1, MSG, 0);
cfg[0]=conf(NI10, MSG | 4, R10, I0 | 02);
cfg[0]=conf(R00, I2 | 01, NI00, MSG | 1);

/* reverse path:
 * NI10(slot 2)
 * -> R10(slot 3)
 * -> R00(slot 4)
 * -> NI00(slot 5)

```

```

    */
    cfg[0]=conf(CMD.SETUPCHANNEL, MSG,MSG|32,0);
    cfg[0]=conf(NI00, MSG | 5, R00, I1 | O2);
    cfg[0]=conf(R10, I2 | O0, NI10, MSG | 0);

    /* credits
    * NI00 - channel 1 given 15 credits
    * NI10 - channel 0 given 15 credits
    */
    cfg[0]=conf(CMD.WRITECREDITS, NI00, MSG|1 ,MSG | 15);
    cfg[0]=conf(CMD.WRITECREDITS, NI10, MSG      ,MSG | 15);
}

void config_programmable_bus ( )
{
    /* configuration of a remote bus,
    * as performed by AEthereal
    *
    * we identify the NI that the bus
    * configuration port is connected to
    * and we write the address mask and value
    * to specific configuration addresses
    */
    writecfg(NI01,0x00000d00,0xf0000000);
    writecfg(NI01,0x00000c00,0x80000001);

    /* for a second channel */
    writecfg(NI01,0x00004504,0xf0000000);
    writecfg(NI01,0x00004404,0x90000001);
}

int main(void)
{
    int i;
    /* configure a remote bus */
    config_secbus ( );
    /* setup connections */
    setup_connection_1 ( );
    setup_connection_2 ( );
    /* perform write and read operations
    * to remote meories to verify proper
    * operation
    */
    for (i=0; i<256; i++) mem0[i]=i+256;
    printf("%d",mem0[10]);
}

```

```
    return 0;  
}
```

## Appendix C

### C source of the online allocation algorithm

Some function names and variables were renamed to better correspond with the algorithm in chapter 5. The main loop body has been reorganized to achieve faster execution.

```
void norec(int source)
{
    int crtLink, crtDest, level, allowedDistance;
    level=0;
    allowedDistance=dist[ source ];
    slotmask tmp;
    solution[0]=source;
    solLink[0]=crtLink=firstLink[ source ];
    avSlots[0]=ALLSLOTS;
    goto nrl2;
    while ( 1 ) // main non-recursive loop
    {
        sl=avSlots[level];
        nrl1:
        crtLink=solLink[level]+1;
        nrl2:
        solLink[level]=crtLink;
        node=solution[level];
        if ( crtLink>=lastLink[node] )
        {
            if ( !level ) break;
            level--;
            continue;
        }
    }
}
```

```

    }
    crtDest=dest[crtLink];
    tmp=sl & ~ slots[crtLink];
    if (crtDest==dest)
    {
        if (compute_bw(tmp)<requiredBw)
        {
            goto nrl1;
        }
        solution[level+1]=dest;
        avSlots[level+1]=tmp;
        solution_found(level);
        break;
    }
    if (dist[crtDest]>(allowedDistance-level))
    {
        goto nrl1;
    }
    if (compute_bw(tmp)<requiredBw)
    {
        goto nrl1;
    }
    level++;
    solution[level]=crtDest;
    avSlots[level]=sl=shift(tmp);
    crtLink=firstLink[crtDest];
    goto nrl2;
}
}

```

## **Appendix D**

### **Notations**

Table D.1: Table of Notations

$V$	the set of network nodes, can be interpreted as vertices in a graph
$R$	the set of routers, $R \subset V$
$r$	one router, $r \in R$
$J$	the set of communication channels
$j$	one communication channel, $j \in J$
$E$	the set of network links, can be interpreted as edges in a multigraph
$e$	one edge $e \in E$
$P_r$	the set of links arriving at router $r$
$Q_r$	the set of links departing from router $r$
$y_j$	the bandwidth requirement of channel $j$
$z_{ej}$	the bandwidth used on link $e$ by channel $j$
$z_{ejk}$	the bandwidth used in slot $k$ on link $e$ by channel $j$
$z_{ejk}$	the remaining capacity of slot $k$ on link $e$ after the $j$ th allocation
$P$	a path in the network (a list of network links or edges)
$S$	the set of all slots
$s_i$	the slot with number $i$ , numbering starts from 1
$S_{q,w}$	the set slots available on the edge $(q, w)$
$S_{path}$	the set slots available a <i>path</i> , considering proper alignment
<i>Source</i>	the source node during a path-finding operation
<i>Destination</i>	the destination node during a path-finding operation
$bw_r$	the requested bandwidth
$c_e$	the capacity of an edge (flow algorithm)
$\mathcal{B}(S_x)$ or $\mathcal{B}_{S_x}$	the bandwidth offered by a set of slots $S_x$



Table D.2: Table of Notations for the in-order path selection algorithm

$p_i$	a path (represented by path length and set of slots)
$r_i$	the arrival time of path $p_i$
$A$	set of all paths
$X$	a solution to the in-order slot selection problem
$\xi_X$	the set of all solutions
$\mathcal{A}$	an optimal solution
$X_i$	a solution with the property that $p_i \in X_i$
$\xi_{X_i}$	the set of all solutions of the form $X_i$
$\mathcal{A}_i$	an optimum over the set $\xi_{X_i}$
$Q_{1,j}$	a subset of $A$ , $\{p_1, p_2, \dots, p_j\}$
$X_{1,j}$	a solution which is a subset of $Q_{1,j}$
$\xi_{X_{1,j}}$	the set of all solutions of the form $X_{1,j}$
$\mathcal{A}_{1,j}$	an optimum over the set $\xi_{X_{1,j}}$
$\mathcal{A}'_{1,j}$	an optimal solution like $\mathcal{A}_{1,j}$ used for proof by contradiction

Table D.3: Table of Notations for the slot selection algorithms

$A$	the set of available slots
$A_{k,i,j}$	subset of $\{s_2 \dots s_k\}$ with $s_k \in A_{k,i,j}$ , $ A_{k,i,j}  = j$ and ending with $q$ selected slots where $q \bmod 3 = i$
$\mathcal{P}(A, i)$	the property that set $A$ obeys the bandwidth requirement over the window $\{s_i \dots s_{i+w-1}\}$
$\mathcal{A}$	a minimal subset of $A$ which satisfies $\mathcal{P}(A, i)$ , $\forall i$
$\mathcal{A}_{k,i,j}$	a minimal set of the form $A_{k,i,j}$ that satisfies $\mathcal{P}(A, x)$ , $\forall x \in \{1 \dots k - w + 1\}$
$w$	the size of the window for the enhanced slot selection algorithm
$C_k$	a list of values $c_{k-w+2} \dots c_k \in \{0, 1\}$
$A_{k,C_k}$	a subset of $A$ with the property that $s_q \in A_{k,C_k} \Leftrightarrow c_q = 1, \forall q \in \{k-w+2 \dots k\}$
$A_{w-1,START}$	a particular case of $A_{k,C_k}$ where $k = w - 1$
$\mathcal{A}_{k,C_k}$	a valid, optimal partial solution of the form $A_{k,C_k}$



## Bibliography

- [AAN<sup>+</sup>07] M. H. Albert, M. D. Atkinson, Doron Nussbaum, Jörg-Rüdiger Sack, and Nicola Santoro. On the longest increasing subsequence of a circular list. *Information Processing Letters*, 101(2), 2007.
- [ACG<sup>+</sup>03] A. Adriahtenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2003.
- [ACPP06] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. A new selection policy for adaptive routing in network on chip. In *Proceedings of the 5th WSEAS International Conference on Electronics, Hardware, Wireless and Optical Communications (EHAC)*, 2006.
- [AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.
- [AG03] A. Andriahtenaina and A. Greiner. Micro-Network for SoC: implementation of a 32-Port SPIN network. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2003.
- [ANM<sup>+</sup>05] P. Avasare, V. Nollet, J-Y. Mignolet, D. Verkest, and H. Corporaal. Centralized end-to-end flow control in a best-effort network-on-chip. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT)*, pages 17–20, 2005.
- [Art05] Arteris. A comparison of network-on-chip and busses. White paper, Arteris, 2005.
- [BB04] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 2004.
- [BC06] Luciano Bononi and Nicola Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE): Designers' forum*, pages 154–159, 2006.

- [BCG<sup>+</sup>07] Luciano Bononi, Nicola Concer, Miltos Grammatikakis, Marcello Coppola, and Riccardo Locatelli. NoC topologies exploration based on mapping and simulation models. In *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD)*, pages 543–546, 2007.
- [BCGK04] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 2004.
- [BCV<sup>+</sup>05] 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 the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*., pages 54 – 63, March 2005.
- [BDM02] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1), January 2002.
- [Ben06] L. Benini. Application specific NoC design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, volume 1, pages 1 –5, March 2006.
- [BGK<sup>+</sup>11] Saddek Bensalem, Kees Goossens, Christoph M. Kirsch, Roman Obermaisser, Edward A. Lee, and Joseph Sifakis. Time-predictable and composable architectures for dependable embedded systems. In *Proceedings of the ninth ACM international conference on Embedded software (EMSOFT)*, pages 351–352, 2011.
- [Bje05] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2005.
- [BJM<sup>+</sup>05] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. NoC synthesis flow for customized domain specific multiprocessor Systems-on-Chip. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):113–129, 2005.
- [BKVW03] N. Bindal, T. Kelly, N. Velastegui, and K.L. Wong. Scalable sub-10ps skew global clock distribution for a 90nm multi-ghz ia microprocessor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, volume 1, pages 346–498, 2003.
- [BM03] Praveen Bhojwani and Rabi Mahapatra. Interfacing cores with on-chip packet-switched networks. In *Proceedings of the 16th International Conference on VLSI Design (VLSID)*, page 382, 2003.
- [BS05] T. Bjerregaard and J. Sparso. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2005.

- [BWM<sup>+</sup>09] Arnab Banerjee, Pascal T. Wolkotte, Robert D. Mullins, Simon W. Moore, and Gerard J. M. Smit. An energy and performance exploration of Network-on-Chip architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):319–329, 2009.
- [CA11] Mark J. Cianchetti and David H. Albonesi. A low-latency, high-throughput on-chip optical router architecture for future chip multiprocessors. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 7:9:1–9:20, July 2011.
- [CCG<sup>+</sup>04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Giuseppe Maruccia, and Francesco Papariello. OCCN: a Network-On-Chip modeling and simulation framework. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2004.
- [CFD<sup>+</sup>11] J. Camacho, J. Flich, J. Duato, H. Eberle, and W. Olesinski. Towards an efficient NoC topology through multiple injection ports. In *Euro-micro Conference on Digital System Design (DSD)*, pages 165–172, 2011.
- [CILC96] R. C. Carden IV, J. Li, and C. K. Cheng. A global router with a theoretical bound on the optimal solution. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(2):208–216, 1996.
- [CJW<sup>+</sup>99] J. Y. Chen, W. B. Jone, J. S. Wang, H. I. Lu, and T. F. Chen. Segmented bus design for low-power systems. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 7(1):25–29, 1999.
- [CMR<sup>+</sup>06] Martijn Coenen, Srinivasan Murali, Andrei Ruadulescu, Kees Goossens, and Giovanni De Micheli. A buffer-sizing algorithm for networks on chip using TDMA and credit-based end-to-end flow control. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 130–135, 2006.
- [CN98] S. Chen and K. Nahrstedt. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. *Network, IEEE*, 12(6):64–79, 1998.
- [Cor99] IBM Corporation. The CoreConnect bus architecture. White paper, 1999.
- [Cor01] IBM Corporation. On-chip peripheral bus architecture specification, 2001.
- [CP08] Jeremy Chan and Sri Parameswaran. NoCOUT: NoC topology generation with mixed packet-switched and point-to-point networks. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 265–270, 2008.

- [CQSD99] R. Casado, F. Quiles, J. Sanchez, and J. Duato. Deadlock-free routing in irregular networks with dynamic reconfiguration. *Network-Based Parallel Computing. Communication, Architecture, and Applications*, pages 165–180, 1999.
- [CRS97] I. Cidon, R. Rom, and Y. Shavitt. Multi-path routing combined with resource reservation. In *Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 1, pages 92–100 vol.1, April 1997.
- [CRS99] Israel Cidon, Raphael Rom, and Yuval Shavitt. Analysis of multi-path routing. *IEEE/ACM Transactions on Networking*, 7(6), 1999.
- [CSC06] Kuei-Chung Chang, Jih-Sheng Shen, and Tien-Fu Chen. Evaluation and design trade-offs between circuit-switched and packet-switched NOCs for application-specific SoCs. In *Proceedings of the 43rd annual Design Automation Conference (DAC)*, pages 143–148, 2006.
- [CSC08] Kuei-Chung Chang, Jih-Sheng Shen, and Tien-Fu Chen. Tailoring circuit-switched network-on-chip to application-specific system-on-chip by two optimization schemes. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13:12:1–12:31, 2008.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [DA93] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions Parallel and Distributed Systems (TPDS)*, 4(4), 1993.
- [DHW02] A. DeHon, R. Huang, and J. Wawrzynek. Hardware-assisted fast routing. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 205–215, 2002.
- [Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1959.
- [DSB98] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 320–328, 1998.
- [DT01] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the annual Design Automation Conference (DAC)*, 2001.
- [DT03] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

- [Dua91] J. Duato. On the design of deadlock-free adaptive routing algorithms for multicomputers: design methodologies. In *Proceedings on Parallel architectures and languages Europe (PARLE)*, 1991.
- [EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2), 1972.
- [fCB94] Tien fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st annual international symposium on Computer architecture (ISCA)*, 1994.
- [FMLD07] J. Flich, A. Mejia, P. Lopez, and J. Duato. Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips. In *Proceedings of the First International Symposium on Networks-on-Chip (NOCS)*, pages 183–194, 2007.
- [Fou] Free Software Foundation. GNU linear programming kit. <http://www.gnu.org/software/glpk>.
- [GDR05] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5), 2005.
- [GG00] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, 2000.
- [GH10] Kees Goossens and Andreas Hansson. The æthereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proceedings of the annual Design Automation Conference (DAC)*, 2010.
- [GLL<sup>+</sup>90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, 1990.
- [GMB<sup>+</sup>08] Francisco Gilabert, Simone Medardoni, Davide Bertozzi, Luca Benini, Mara Engracia Gomez, Pedro Lopez, and Jos Duato. Exploring High-Dimensional topologies for NoC design through an integrated analysis and synthesis framework. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 107–116, 2008.
- [Goo89] James Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.
- [Han09] Andreas Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, Eindhoven University of Technology, 2009.

- [HCG07a] Andreas Hansson, Martijn Coenen, and Kees Goossens. Channel trees: reducing latency by sharing time slots in time-multiplexed networks on chip. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.
- [HCG07b] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pages 954–959, 2007.
- [HG07] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proceedings of the First International Symposium on Networks-on-Chip (NOCS)*, pages 233–242, 2007.
- [HG09] Andreas Hansson and Kees Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES-ISSS)*, pages 99–108, 2009.
- [HG10] Andreas Hansson and Kees Goossens. *On-Chip Interconnect with aelite: Composable and Predicable Systems*. Embedded Systems. Springer, 2010.
- [HG11] A. Hansson and K. Goossens. A quantitative evaluation of a network on chip design flow for multi-core consumer multimedia applications. *Journal of Design Automation for Embedded Systems (DAEM)*, 15(2): 1–32, June 2011.
- [HGBH09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14:2:1–2:24, January 2009.
- [HGR07] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI Design*, 2007.
- [Hil98] Mark D. Hill. Multiprocessors should support simple Memory-Consistency models. *Computer*, 31(8):28–34, 1998.
- [HJK04] R. Holsmark, A. Johansson, and S. Kumar. On connecting cores to packet switched on-chip networks: A case study with Microblaze processor cores. In *Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, April 2004. Slovakia.



- [HM03] Jingcao Hu and Radu Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2003.
- [HM04] Jingcao Hu and Radu Marculescu. DyAD: Smart routing for networks-on-chip. In *Proceedings of the annual Design Automation Conference (DAC)*, pages 260–263, 2004.
- [HM05] Jingcao Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(4):551–562, 2005.
- [HS01] J. Hu and S. S Sapatnekar. A survey on multi-net global routing for integrated circuits. *Integration, the VLSI Journal*, 31(1):1–49, 2001.
- [HSG09] A. Hansson, M. Subburaman, and K. Goossens. aelite: A flit-synchronous network on chip with composable and predictable services. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2009.
- [Hu63] T. C. Hu. Multi-Commodity network flows. *Operations Research*, 11(3):344–360, 1963.
- [Hue00] Lorenz Huelsbergen. A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph algorithms. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, pages 105–115, 2000.
- [HWM<sup>+</sup>09] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET, Computers Digital Techniques (IET-CDT)*, 3(5), September 2009.
- [IHCE07] D.A. Ilitzky, J.D. Hoffman, A. Chun, and B.P. Esparza. Architecture of the scalable communications core’s network on chip. *IEEE Micro*, 27(5):62–74, September-October 2007.
- [JBK<sup>+</sup>09] Ajay Joshi, Christopher Batten, Yong-Jin Kwon, Scott Beamer, Imran Shamim, Krste Asanovic, and Vladimir Stojanovic. Silicon-photonics networks for global on-chip communication. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 124–133, 2009.
- [KC11] Yu-Hsiang Kao and H. Jonathan Chao. BLOCON: a bufferless photonic cros network-on-chip architecture. In *Proceedings of the 5th ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 81–88, 2011.

- [KD10] Gul N Khan and Victor Dumitriu. A modeling tool for simulating and design of on-chip network systems. *Microprocessors & Microsystems*, 34:84–95, 2010.
- [KH09] Somayyeh Koochi and Shaahin Hessabi. Contention-free on-chip routing of optical packets. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 134–143, 2009.
- [KMC<sup>+</sup>05] M. Kreutz, C. Marcon, L. Carro, N. Calazans, and A. A Susin. Energy and latency evaluation of NoC topologies. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 5866– 5869 Vol. 6, May 2005.
- [KMF<sup>+</sup>05] Michihiro Koibuchi, Juan C. Martinez, Jose Flich, Antonio Robles, Pedro Lopez, and Jose Duato. Enforcing in-order packet delivery in system area networks with adaptive routing. *Journal of Parallel and Distributed Computing*, 65(10):1223 – 1236, 2005.
- [Kol05] S. G Kolliopoulos. Edge-disjoint paths and unsplittable flow. *Draft chapter from the forthcoming Handbook of Approximation Algorithms and Metaheuristics*, 60, 2005.
- [KS93] J. Kim and K. G Shin. Deadlock-free fault-tolerant routing in injured hypercubes. *IEEE Transactions on Computers*, 42(9):1078–1088, 1993.
- [KSJW06] N. Kavaldjiev, G.J.M. Smit, P.G. Jansen, and P.T. Wolkotte. A virtual channel Network-on-Chip for GT and BE traffic. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*, 2006.
- [KSWJ06] N. Kavaldjiev, G. Smit, P. Wolkotte, and P. Jansen. Providing QoS guarantees in a NoC by virtual channel reservation. *Reconfigurable Computing: Architectures and Applications*, pages 299–310, 2006.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computers*, 28 (9):690–691, 1979.
- [LCOM08] Hyung Gyu Lee, Naehyuck Chang, Umit Y Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12:23:1–23:20, May 2008.
- [Lei85] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34:892–901, 1985.

- [LG01] S. J Lee and M. Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *Proceedings of the IEEE International Conference on Communications, (ICC)*, volume 10, pages 3201–3205, 2001.
- [LGM<sup>+</sup>09] D. Ludovici, F. Gilabert, S. Medardoni, C. Gmez, M. E Gmez, P. Lpez, G. N Gaydadjiev, and D. Bertozzi. Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 562–565, 2009.
- [Lim01] ARM Limited. Multi-layer AHB overview, 2001.
- [Lim08] ARM Limited. AMBA APB protocol specification v2.0, 2008.
- [LJ08] Zhonghai Lu and Axel Jantsch. TDM Virtual-Circuit configuration for Network-on-Chip. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 16:1021–1034, August 2008.
- [LL11] Angelo Kuti Lusala and Jean-Didier Legat. Combining SDM-based circuit switching with packet switching in a NoC for real-time applications. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2505 –2508, May 2011.
- [LLP05] Il-Gu Lee, Jin Lee, and Sin-Chong Park. Adaptive routing scheme for NoC communication architecture. In *Proceedings of the 7th International Conference on Advanced Communication Technology (ICACT)*, volume 2, 2005.
- [LLTB03] A. Laffely, Jian Liang, R. Tesseir, and W. Burleson. Adaptive system on a chip (ASOC): a backbone for power-aware signal processing cores. In *Proceedings of the International Conference on Image Processing (ICIP)*, 2003.
- [LMS<sup>+</sup>05] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest. Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 81–86, 2005.
- [LO09] S. Loucif and M. Ould-Khaoua. Performance analysis of deterministically-routed bi-directional torus with non-uniform traffic distribution. *Future Generation Computer Systems*, 25:489–498, May 2009.
- [LR95] F. T. Leighton and S. Rao. Circuit switching: a multicommodity flow based approach. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

- [LSGB11] D. Ludovici, A. Strano, G. N. Gaydadjiev, and D. Bertozzi. Mesochronous NoC technology for power-efficient GALS MPSoCs. In *Proceedings of the Fifth ACM Interconnection Network Architecture, On-Chip Multi-Chip Workshop (INA-OCMC)*, pages 27–30, January 2011.
- [LST00] Jian Liang, Sriram Swaminathan, and Russell Tessier. aSOC: A scalable, single-chip communications architecture. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 37–, 2000.
- [LVG10] E. Larsson, B. Vermeulen, and K. Goossens. A distributed architecture to check global properties for post-silicon debug. In *Proceedings of the 15th IEEE European Test Symposium (ETS)*, pages 182 –187, 2010.
- [LWS<sup>+</sup>02] D. Liu, D. Wiklund, E. Svensson, O. Seger, and S. Sathe. SoCBUS: the solution of high communication bandwidth on chip and short TTM. In *Proceedings of the Real-Time and Embedded Computing Conference (RTECC)*, 2002.
- [LZJ06] Ming Li, Qing-An Zeng, and Wen-Ben Jone. DyXY: A proximity congestion-aware deadlock-free dynamic routing method for network on chip. In *Proceedings of the 43rd annual Design Automation Conference (DAC)*, 2006.
- [LZT04] Jian Liu, Li-Rong Zheng, and Hannu Tenhunen. Interconnect intellectual property for network-on-chip (NoC). *Journal of Systems Architecture (JSA)*, 50(2-3), 2004.
- [MABDM07] S. Murali, D. Atienza, L. Benini, and G. De Micheli. A method for routing packets across multiple paths in NoCs with in-order delivery and fault-tolerance guarantees. *VLSI-Design Journal*, 2007.
- [Max07] N. F. Maxemchuk. Dispersity routing: Past and present. In *Proceedings of the IEEE Military Communications Conference, (MILCOM)*, pages 1–7, 2007.
- [MBD<sup>+</sup>05] T. Marescaux, B. Bricke, P. Debacker, V. Nollet, and H. Corporaal. Dynamic time-slot allocation for QoS enabled networks on chip. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA)*, pages 47–52, 2005.
- [McM86] F. H. McMahon. The Livermore Fortran kernels: a computer test of the numerical performance range. 1986.
- [MHH02] O. Mencer, Z. Huang, and L. Huelsbergen. HAGAR: efficient multi-context graph processors. *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 915–924, 2002.

- [MKY<sup>+</sup>05] H. Matsutani, M. Koibuchi, Y. Yamada, A. Jouraku, and H. Amano. Non-minimal routing strategy for application-specific networks-on-chips. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW)*, pages 273–280, 2005.
- [MM04] Srinivasan Murali and Giovanni De Micheli. SUNMAP: a tool for automatic topology selection and generation for NoCs. In *Proceedings of the 41st annual Design Automation Conference (DAC)*, pages 914–919, 2004.
- [MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proceedings of the ACM symposium on Applied computing (SAC)*, pages 1557–1564, 2007.
- [MMV09] M. Moadeli, P. Maji, and W. Vanderbauwhede. Quarc: A High-Efficiency network on-Chip architecture. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, 2009.
- [MNTJ04a] 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 Conference on Design, Automation and Test in Europe (DATE)*, 2004.
- [MNTJ04b] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2004.
- [MPCJ08] Edson Ifarraguirre Moreno, Katalin Maria Popovici, Ney Laert Vilar Calazans, and Ahmed Amine Jerraya. Integrating abstract NoC models within MPSoC design. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, pages 65–71, 2008.
- [MS80] Philip Merlin and Paul Schweitzer. Deadlock avoidance in Store-and-Forward Networks–II: other deadlock types. *Communications, IEEE Transactions on*, 28(3):355–360, 1980.
- [MSAA09] Mehdi Modarressi, Hamid Sarbazi-Azad, and Mohammad Arjomand. A hybrid packet-circuit switched on-chip network based on SDM. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 566–569, 2009.
- [MSVO07] M. Moadeli, A. Shahrabi, W. Vanderbauwhede, and M. Ould-Khaoua. An analytical performance model for the Spidergon NoC. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, pages 1014–1021, 2007.

- [NHCG10] A. Nelson, A. Hansson, H. Corporaal, and K. Goossens. Conservative application-level performance analysis through simulation of MPSoCs. In *Proceedings of the 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 51–60, 2010.
- [OM05] Umit Y. Ogras and Radu Marculescu. Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 352–357, 2005.
- [OM06] U.Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. In *Proceedings of the annual Design Automation Conference (DAC)*, pages 839–844, 2006.
- [OMM<sup>+</sup>08] Luciano Ost, Fernando G Moraes, Leandro Mller, Leandro Soares Indrusiak, Manfred Glesner, Sanna Mtt, and Jari Nurmi. A simplified executable model to evaluate latency and throughput of networks-on-chip. In *Proceedings of the 21st annual symposium on Integrated circuits and system design (SBCCI)*, pages 170–175, 2008.
- [PABB05] Antonio Pullini, Federico Angiolini, Davide Bertozzi, and Luca Benini. Fault tolerance overhead in network-on-chip flow control schemes. In *Proceedings of the 18th annual symposium on Integrated circuits and system design (SBCCI)*, pages 224–229, 2005.
- [PAM<sup>+</sup>07] Antonio Pullini, Federico Angiolini, Paolo Meloni, David Atienza, Srinivasan Murali, Luigi Raffo, Giovanni De Micheli, and Luca Benini. NoC design and implementation in 65nm technology. In *Proceedings of the First International Symposium on Networks-on-Chip (NOCS)*, pages 273–282, 2007.
- [PGJ<sup>+</sup>05] Partha Pratim Pande, Cristian Grecu, Michael Jones, Andre Ivanov, and Resve Saleh. Performance evaluation and design Trade-Offs for Network-on-Chip interconnect architectures. *IEEE Transactions on Computers*, 54:1025–1040, 2005.
- [PHKC06] M. Palesi, R. Holsmark, S. Kumar, and V. Catania. A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, page 147, 2006.
- [Pis84] Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press, 1984.
- [PK08] Christian Paukovits and Hermann Kopetz. Concepts of switching in the Time-Triggered Network-on-Chip. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 120–129, 2008.

- [PKP10] G. Passas, M. Katevenis, and D. Pnevmatikatos. A 128 x 128 x 24Gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6% of their area. In *Proceedings of the 4th ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 87–95, 2010.
- [PS06] A. Patooghy and H. Sarbazi-Azad. Analytical performance modelling of partially adaptive routing in wormhole hypercubes. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, page 7 pp., 2006.
- [PSL03] J. Plosila, T. Seceleanu, and P. Liljeberg. Implementation of a self-timed segmented bus. *IEEE Design & Test of Computers*, 20(6):44–50, 2003.
- [Rab89] M. O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2): 335–348, 1989.
- [RDP<sup>+</sup>05] Andrei Radulescu, John Dielissen, Santiago Gonzlez Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *Transactions on CAD of Integrated Circuits and Systems (TCAD)*, 2005.
- [SAC<sup>+</sup>05] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. D Micheli. Xpipes lite: a synthesis oriented design library for networks on chips. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2005.
- [Sai68] Romesh Saigal. *Multicommodity flows in directed networks*. PhD thesis, University of California, Berkeley, 1968.
- [SBB<sup>+</sup>06] Sanjay Singh, Shilpa Bhoj, Dheera Balasubramanian, Tanvi Nagda, Dinesh Bhatia, and Poras Balsara. Generic network interfaces for plug and play NoC based architecture. In *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 287–298. 2006.
- [SBC07] Assaf Shacham, Keren Bergman, and Luca P. Carloni. The case for low-power photonic networks on chip. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, pages 132–135, 2007.
- [SBE08] S. Suboh, M. Bakhouya, and T. El-Ghazawi. Simulation and evaluation of On-Chip interconnect architectures: 2D mesh, spidergon, and WK-Recursive network. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 205–206, 2008.

- [SBG<sup>+</sup>08] Sander Stuijk, Twan Basten, Marc Geilen, Amir Hossein Ghamarian, and Bart Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on NoC. *Journal of Systems Architecture*, 54:411–426, March 2008.
- [SC97] Kevin Skadron and Douglas W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 144–155, 1997.
- [Sch61] C Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 1961.
- [Sch07] M. Schoeberl. A Time-Triggered Network-on-Chip. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 377–382, August 2007.
- [SCK05] K. Srinivasan, K. S. Chatha, and G. Konjevod. An automated technique for topology and route generation of application specific on-chip interconnection networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 231–237, 2005.
- [SCK06] K. Srinivasan, K.S. Chatha, and G. Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 14(4): 407–420, April 2006.
- [SdWG10] R.A. Stefan, J de Windt, and K.G.W. Goossens. On-chip network interfaces supporting automatic burst write creation, posted writes and read prefetch. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 185–192, July 2010.
- [SG09] Radu Stefan and Kees Goossens. Multipath routing in TDM NoCs. In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2009.
- [SG11] Radu Stefan and Kees Goossens. Enhancing the security of time-division-multiplexing networks-on-chip through the use of multipath routing. In *Proceedings of the International Workshop on Network on Chip Architectures (NoCArc)*, pages 57–62, 2011.
- [SHG10] F.A. Samman, T. Hollstein, and M. Glesner. Adaptive and deadlock-free tree-based multicast routing for networks-on-chip. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 18(7):1067–1080, 2010.
- [SKFO05] F. Safaei, A. Khonsari, M. Fathy, and M. Ould-Khaoua. Performance modelling and analysis of pipelined circuit switching in hypercubes with faults. In *Proceedings of the Eighth International Conference*



- on *High-Performance Computing in Asia-Pacific Region (HPCASIA)*, pages 265–, 2005.
- [SKFO08a] F. Safaei, A. Khonsari, M. Fathy, and M. Ould-Khaoua. Modeling and predicting point-to-point communication delay of circuit switching in the mesh-connected networks. In *Proceedings of the 9th international conference on Distributed computing and networking (ICDCN)*, 2008.
- [SKFO08b] F. Safaei, A. Khonsari, M. Fathy, and M. Ould-Khaoua. Pipelined circuit switching: Analysis for the torus with non-uniform traffic. *Journal of Systems Architecture: the EUROMICRO Journal*, 54:97–110, 2008.
- [SKKC09] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 209–218, 2009.
- [SKL<sup>+</sup>07] Erno Salminen, Tero Kangas, Vesa Lahtinen, Jouni Riihimki, Kimmo Kuusilinna, and Timo D. Hmlinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *Journal of Systems Architecture*, 53(8):477–488, 2007.
- [SLB07] C. Schuck, S. Lamparth, and J. Becker. artNoC - a novel Multi-Functional router architecture for organic computing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2007.
- [SLKH02] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen. Overview of bus-based system-on-chip interconnections. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages II–372–II–375 vol.2, 2002.
- [SM90] Farhad Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, 1990.
- [SMBDM10] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. Sunfloor 3d: A tool for networks on chip topology synthesis for 3-d systems on chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(12):1987–2000, December 2010.
- [SSM<sup>+</sup>01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th Conference on Design Automation (DAC)*, 2001.
- [tBHK<sup>+</sup>10] Timon D. ter Braak, Philip K. F. Holzenspies, Jan Kuper, Johann L. Hurink, and Gerard J. M. Smit. Run-time spatial resource management for real-time applications on heterogeneous MPSoCs. In *Proceedings*

- of the *Conference on Design, Automation and Test in Europe (DATE)*, pages 357–362, 2010.
- [TDB03] B. Towles, W. J. Dally, and S. Boyd. Throughput-centric routing algorithm design. In *Proceedings of the 15th annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 200–209, 2003.
- [TLZ09] Li Tong, Zhonghai Lu, and Hua Zhang. Exploration of slot allocation for On-Chip TDM virtual circuits. In *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 127–132, 2009.
- [TM09] Dietmar Tutsch and Mirosław Malek. Comparison of network-on-chip topologies for multicore systems considering multicast and local traffic. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools)*, pages 23:1–23:9, 2009.
- [TOM<sup>+</sup>11] R. Tornero, J. Ordua, A. Mejia, J. Flich, and J. Duato. A communication-driven routing technique for application-specific NoCs. *International Journal of Parallel Programming*, 39:357–374, 2011.
- [VdTJ02] E.B. Van der Tol and E.G.T. Jaspers. Mapping of MPEG-4 decoding on a flexible architecture platform. *Media Processors*, 4674, 2002.
- [WCK08] I. Walter, I. Cidon, and A. Kolodny. BENoC: A bus-enhanced network on-chip for a power efficient cmp. *Computer Architecture Letters*, 7(2):61–64, 2008.
- [WF08] Markus Winter and Gerhard P. Fettweis. A Network-on-Chip channel allocator for Run-Time task scheduling in Multi-Processor System-on-Chips. In *Proceedings of the 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 133–140, 2008.
- [WM88] D. C. Winsor and T. N. Mudge. Analysis of bus hierarchies for multiprocessors. *ACM SIGARCH Computer Architecture News*, 16: 100–107, May 1988.
- [WSRS05] P.T. Wolkotte, G.J.M. Smit, G.K. Rauwerda, and L.T. Smit. An Energy-Efficient reconfigurable Circuit-Switched Network-on-Chip. In *Proceedings of the 19th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2005.
- [WZLY08] Yu Wang, Kai Zhou, Zhonghai Lu, and Huazhong Yang. Dynamic TDM virtual circuit implementation for NoC. In *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 1533–1536, December 2008.

- [ZFK<sup>+</sup>09] Linlin Zhang, Virginie Fresse, Mohammed Khalid, Dominique Houzet, and Anne-Claire Legrand. Evaluation and design space exploration of a Time-Division multiplexed NoC on FPGA for image analysis applications. *EURASIP Journal on Embedded Systems*, 2009: 1–15, 2009.
- [ZS03] C.A. Zeferino and A.A. Susin. SoCIN: a parametric and scalable network-on-chip. In *Proceedings of the 16th annual symposium on Integrated circuits and system design (SBCCI)*, 2003.



# List of Publications

## *Journals*

- **R.A. Stefan**, K. Goossens, “A TDM slot allocation flow based on multipath routing in NoCs”, *Journal of Microprocessors and Microsystems*, September 2010

## *Peer-reviewed International Conferences and Workshops*

- **R. A. Stefan**, S. D. Cotofana, “Bitstream Compression Techniques for Virtex 4 FPGAs”, *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 323-328, Heidelberg, Germany, September 2008
- **R. Stefan**, K. Goossens “Multipath Routing in TDM NoCs”, *Proceedings of the International Conference on VLSI and System-on-Chip (VLSI-SoC)*, Florianopolis, Brazil, October 2009
- A. Molnos, J. A. Ambrose, A. Nelson, **R. Stefan**, S. Cotofana, K. Goossens, “A Composable, Energy-Managed, Real-Time MPSOC Platform”, *Proceedings of the International Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, Brasov, Romania, May 2010
- **R. A. Stefan**, J. de Windt, K. Goossens, “On-chip Network Interfaces supporting automatic burst write creation, posted writes and read prefetch”, *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 185-192, Samos, Greece, July 2010
- **R. Stefan**, K. Goossens “An Improved Algorithm for Slot Selection in the Æthereal Network-on-Chip”, *Proceedings of the HiPEAC Workshop on Interconnection Network Architecture On-Chip, Multi-Chip (INA-OCMC)*, pp. 7-10, Heraklion, Crete, Greece, January 2011

- **R. Stefan**, K. Goossens “Enhancing the Security of Time-Division-Multiplexing Networks-on-Chip through the Use of Multipath Routing”, Proceedings of the International Workshop on Network on Chip Architectures (NoCArc), Porto Alegre, Brazil, December 2011
- **R. Stefan**, A. Beyranvand Nejad, K. Goossens “Online allocation for contention-free-routing NoCs”, Proceedings of the HiPEAC Workshop on Interconnection Network Architecture OnChip, MultiChip (INA-OCMC), Paris, France, January 2012
- **R. Stefan**, A. Molnos, A. Ambrose, K. Goossens “A TDM NoC supporting QoS, multicast, and fast connection set-up”, Proceedings of the Conference on Design, Automation and Test in Europe (DATE), Dresden, Germany, March 2012

#### *Local Conferences*

- **R. Stefan**, K. Goossens, “NoC Security using multipath routing”, Proceedings of ProRISC, pp. 522-525, Veldhoven, The Netherlands, November 2009
- **R. Stefan**, K.L.M. Bertels, “A hardware implementation of the UniSim pipeline model”, Proceedings of ProRISC, Veldhoven, The Netherlands, November 2007

#### *Deliverables and Technical Reports*

- **R. Stefan**, I. Sourdis, G. N. Gaydadjiev, K.G.W. Goossens, “Comparison of custom topology networks against rigid interconnects”, CE-TR-2008-01, TU Delft, February 2008

## Samenvatting

**I**N DIT proefschrift behandelen wij het probleem van de verdeling van netwerkbronnen in het kader van op connectie-gebaseerde netwerken op een chip die gegarandeerde prestaties moeten leveren.

Deze verdeling moet aan de bandbreedte- en doorlooptijdvereisten van alle connecties voldoen, en ook aan de beperkingen van het netwerkmodel met betrekking tot de mogelijke verdeling van tijdsloten.

We bieden een theoretisch model van de prestaties die bereikt kunnen worden door het meest algemene netwerk, en we analyseren de vermindering van die prestatie als gevolg van specifieke implementatiekeuzes, zoals topologie, discrete allocatie-eenheden, enzovoort.

Verder behandelen we verdelingsalgoritmen die kenmerkend zijn voor het contentievrije-routerings model, en we stellen verbeterde algoritmen voor om paden en tijdssloten te vinden. Voor sommige van deze algoritmen bewijzen we optimaliteit.

We tonen aan hoe het vinden van paden en tijdsloten plaats kan vinden terwijl het systeem actief is, in plaats van dat tijdens het ontwerp van het systeem te doen.

De resultaten laten zien dat ten opzicht van een ideaal netwerk, praktische netwerkimplementaties die gebruik maken van zogenaamde time-division multiplexing, gemiddeld tot 43% van hun prestaties verliezen, waarvan de grootste gedeelten door discrete allocatie-eenheden en zogenaamde pakketheaders. We stellen een nieuw netwerk voor, dAElite genaamd, dat geen pakketheaders gebruikt, en een verbetering in prestatie tegen lagere implementatiekosten laat zien ten opzichte bestaande van netwerken.





## Curriculum Vitae



**Radu Stefan** was born in Braşov, Romania on the 6<sup>th</sup> of May 1981. He obtained his Bachelor of Engineering degree from the Faculty of Electronics and Computers at the Transilvania University of Braşov in 2005 with a thesis on FPGA compression techniques and a Master of Science degree at the same university with a final dissertation on training neural networks using genetic algorithms. He has worked at Splash Software Braşov on high-end video codecs performing development of rate-control algorithms and codec parallelization. He participated in the Scientific Committee of the Romanian National Olympiad in Informatics in 2001 and 2004, and the Balkan Olympiad in Informatics in 2011.

Copyright © 2012 R. Stefan

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.