Department of Information Engineering and Mathematical Sciences (DIISM)
University of Siena, Italy

Computer Architecture Group
PhD program of Information Engineering and Science (IES)
CYCLE: XXIX

An Efficient NoC-based Framework To Improve Dataflow Thread Management At Runtime

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Doctoral Candidate:

Somnath Mazumdar

Director of the PhD Program in IES:

Prof. Antonio Vicino

July 2017

"Not everything that can be counted counts. Not everything that counts can be counted."– William Bruce Cameron [Informal Sociology: A Casual Introduction to Sociological Thinking]

Acknowledgements

It was indeed an amazing journey. Similar to other PhD students, there are many times where the academic grind and struggle of completing this doctoral degree seems overwhelming. However, I must thank Prof. Antonio Vicino, Prof. Stefano Maci, Prof. Marco Maggini and Prof. Chiara Mocenni for putting me in the right place to keep me going and giving me the peace of mind to complete this doctoral work. I also have to say thank you to Alberto Scionti, Prof. Marco Pranzo and Prof. Anoop S. Kumar allowing me to explore interesting research works, and most importantly supporting me through this doctoral work. It was invaluable to have such supporters who supported my ideas and worked with me to solve them.

From the personal note, I have to thank my mother and my father for their consistent emotional support. These people are the ones more excited than me about this PhD!! I also offer sincere thanks to Mr. Prem G. Krishnan, Choti, and Bachcha for their uncountable supports in the crucial time of my PhD.

ABSTRACT

This doctoral thesis focuses on how the application threads that are based on dataflow execution model can be managed at Network-on-Chip (NoC) level. The roots of the dataflow execution model date back to the early 1970's. Applications adhering to such program execution model follow a simple producer-consumer communication scheme for synchronising parallel thread related activities. In dataflow execution environment, a thread can run if and only if all its required inputs are available. Applications running on a large and complex computing environment can significantly benefit from the adoption of dataflow model.

In the first part of the thesis, the work is focused on the thread distribution mechanism. It has been shown that how a scalable hash-based thread distribution mechanism can be implemented at the router level with low overheads. To enhance the support further, a tool to monitor the dataflow threads' status and a simple, functional model is also incorporated into the design. Next, a software defined NoC has been proposed to manage the distribution of dataflow threads by exploiting its reconfigurability.

The second part of this work is focused more on NoC microarchitecture level. Traditional 2D-mesh topology is combined with a standard ring, to understand how such hybrid network topology can outperform the traditional topology (such as 2D-mesh). Finally, a mixed-integer linear programming based analytical model has been proposed to verify if the application threads mapped on to the free cores is optimal or not. The proposed mathematical model can be used as a yardstick to verify the solution quality of the newly developed mapping policy. It is not trivial to provide a complete low-level framework for dataflow thread execution for better resource and power management. However, this work could be considered as a primary framework to which improvements could be carried out. [303 words]

Contents

List of Figures **ix**

List of Tables **xiii**

1 Introduction **1**

1.1 Thread management issues . . . 3
1.2 Research problem and associated solutions . . . 4
1.3 Thesis structure . . . 7

2 Background **9**

2.1 Dataflow threads . . . 9
2.2 Hardware overview . . . 14
2.3 Interconnection subsystem . . . 20
2.4 Summary . . . 28

3 Thread Distribution **29**

3.1 Introduction . . . 29
3.2 DF-Threads and its scalability . . . 31
3.3 Program Execution Model (PXM) . . . 33
3.4 Proposed Architecture . . . 35
3.5 Hash Scheduling Function . . . 38
3.6 Evaluation . . . 39
3.7 Summary . . . 41
3.8 Acknowledgement . . . 41

4 Monitoring **43**

4.1 Introduction . . . 44
4.2 System model . . . 44
4.3 RADA's implementation . . . 46
4.4 Dealing with heterogeneity . . . 48
4.5 Evaluation . . . 52
4.6 Summary . . . 55
4.7 Acknowledgement . . . 56

5 Thread Management at Software defined NoC **57**

5.1 Introduction . . . 58
5.2 System overview . . . 59
5.3 NoC software interface . . . 61

5.4 Proposed Network-on-Chip architecture 62
5.5 Evaluation 65
5.6 Summary 68
5.7 Acknowledgement 68

6 Customised NoC Architecture 69
6.1 Introduction 70
6.2 System overview 72
6.3 Proposed Network-on-Chip architecture 73
6.4 Evaluation methodology 80
6.5 Applicability and future improvements 90
6.6 Summary 93
6.7 Acknowledgement 94

7 Analytical Model 95
7.1 Introduction 95
7.2 Problem description and assumptions 97
7.3 Mathematical formulation 101
7.4 Simulation results 103
7.5 Summary 114
7.6 Acknowledgement 115

8 Conclusion and Future Work 117
8.1 Contribution 117
8.2 Future work 119

Bibliography 121

List of Figures

1.1 Solution overview: Target chip overview for the manycores clustered processor. Cores are organised into clusters (i.e., nodes) connected each other through a 2D mesh. Within each node, cores and other shared resources are linked via a ring. Each node has a dedicated node manager that controls the runtime and also monitor the system. Each core has a local scheduling unit to distribute the threads. Blocks that are addressed in the thesis are highlighted via dotted line. 8

2.1 Block diagram of parallelism: application level parallelism (left), thread level parallelism (middle) and instruction level parallelism (right) 10
2.2 Thread execution using control and data signals 10
2.3 Programming models classification 11
2.4 Some well-known topology for interconnect subsystem 21
2.5 XY DoR routing direction (left) and block diagram of a mesh router architecture is presented (right) 23

3.1 Instruction count normalised to the matrix size 256. 32
3.2 Speedup of user cycles count normalised to the matrix size 256. 32
3.3 Scaling of read and write operations for DF-Threads. 33
3.4 A simple kernel application adhering with the proposed PXM and a possible mapping of threads on the PEs. 35
3.5 Chip organization: tiles contain a PE (white box) and router (gray box). The scratchpad substitutes the traditional L1-data cache. 36
3.6 Thread Dispatcher module organization (left) with the internal structure of the $H(\cdot)$ function (right). 37
3.7 NoC performance: distribution of threads on the PEs (a), average throughput (b), and power consumption (c). 40

4.1 System overview: the abstract machine model used to managing execution of dataflow threads. 45
4.2 Implementation of the proposed system. 47
4.3 Negative feedback closed-loop based task scheduling system of the RADA. 48
4.4 A code snippet of the recursive Fibonacci kernel: the code highlight the software interface exposed by RADA, which simplifies the amount of code needed to synchronise threads' activities. 51
4.5 Recursive Fibonacci sequence: evaluation of the RADA and OpenMP execution. 52

4.6 Block matrix: evaluation of the RADA and OpenMP execution. 53
4.7 Number of requests issued to the SU by the recursive Fibonacci kernel (single node, 8 cores). 53
4.8 Score function obtained from the execution of the BMM kernel running on host CPU and Intel Xeon Phi accelerator. 55

5.1 Mapping between the physical network with a 2D-mesh topology, and a multi-level virtual topology. Links to the physical network are organised into local rings (blue lines) and a global 2D-mesh among rings (purple lines). 59
5.2 The lightweight router microarchitecture. Ring stations (RSs) have injection and ejection ports, and bypass (blue squares) and power-gating (red squares) bits. Inter-ring switches are power-gated depending on the state of the RS (grey circles are OR/AND gates). 63
5.3 The internal structure of a RS with BP/PG bits and the link counter in the network interface (dashed lines represent selection signals for multiplexers/demultiplexers). 64
5.4 An example of virtual topology mapping: Grey structures represent components (i.e., interconnections, RSs or inter-ring switches) of the router that are power-gated. Red lines correspond to active links used to build local rings, while green lines show links of the mesh. Furthermore, green boxes represented components set in bypass mode and used to construct the mesh among the virtual rings correctly. 65
5.5 Distribution of random traffic over 1024-based CMP. 66

6.1 An instantiation of the proposed scalable NoC: 256 PEs organised into 4×4 block units, each connecting four ringlets. 74
6.2 Modified 2D-mesh router microarchitecture: two groups of local/global channels are used to manage traffic within the 2D-mesh and traffic exchange with local ringlets. 75
6.3 Timing: a) best-case (success) and b) worst-case (failure) of pre-arbitration. 76
6.4 The microarchitecture of the RS of the ringlet's master: horizontal dimension is used to create the bidirectional ring connection, while vertical dimension connects the mesh router and local PE of the ringlet. 77
6.5 Packet header organisation. 79
6.6 Static and dynamic power distribution 82
6.7 Total power consumption with increasing network size. 83
6.8 Average packet latency in uniform random traffic pattern. 85
6.9 Average packet latency in bit-reversal traffic pattern. 85
6.10 Average packet latency in transpose traffic pattern. 86

6.11 Average network throughput in uniform-random traffic pattern. 87
6.12 Average network throughput in bit-reversal traffic pattern. 87
6.13 Average network throughput in transpose traffic pattern. 88
6.14 Average packet latency with increasing network size. 89
6.15 Average network throughput with increasing network size. 90
6.16 Comparing average network throughput and average packet latency with increasing network size. 91
6.17 Internal organisation of the morph control packet (left), and the corresponding control structures in the mesh router (right). 92

7.1 Representation of multiple application running on a NoC 98
7.2 NoC model representation: via architectural model (left) and also via theoretical model (right) 98
7.3 Basic system graph representation including application threads and associated single memory controller 100
7.4 T2C mapping process in zig-zag heuristic algorithm for NoC size 4×4 . . 103
7.5 Relative average traffic performance 110
7.6 MC influence on four algorithms while objective function is energy cost and latency 111
7.7 Comparing latency with energy cost with different network sizes 111
7.8 Performance comparison between 5×6 and 4×8 113
7.9 Comparing the queue length for network size 4×8: (left) MC=1, (right) MC=2 113
7.10 Comparing the queue length for network size 5×6: (left) MC=1, (right) MC=2 114

List of Tables

2.1 Theoretical performance of 8x8 mesh for six synthetic traffic patterns . . 25

4.1 The execution trace (first 10 entries) for the RFS kernel with the input size set equals to $n = 10$. 54
4.2 The execution trace (first six entries) captured from one PE when running RFS kernel with size $n = 10$. 54

6.1 Mesh-router: main microarchitectural parameters. 76
6.2 Area and power comparison between a standard router architecture and the proposed mesh router. 81
6.3 Relative resource utilisation in Vivado (values are in percentage). 81

7.1 Experiment configuration 104
7.2 Problem size for the MILP_0 model 105
7.3 Performance of MILP_0 model while the objective function is minimisation of energy cost 106
7.4 Performance of MILP_0 model while the objective function is minimisation of latency 107
7.5 Performance of MILP model while the objective function is minimisation of energy 107
7.6 Performance of MILP model while the objective function is minimisation of latency 108
7.8 Gap from the LB(MILP) which is the best attainable for the three algorithms (MILP, ZZ, RND) while minimising the latency 109
7.7 Gap from the LB(MILP) which is the best attainable for the three algorithms (MILP, ZZ, RND) while minimising the energy cost 109

1 Introduction

In recent years, applications have evolved from simple, monolithic, centralised execution model to highly agile, distributed and dynamic model. These transformations have forced to make changes in the hardware microarchitecture for better support at the software level. The functional part of computation unit has also evolved a lot due to the changing nature of user applications. Some well-supported features of the current applications are: *i)* dynamic scaling (up and/or down) of number of processing cores as the execution time progresses (e.g., MapReduce programming framework); *ii)* requirement of heterogeneous processing cores (supported by e.g., asymmetric CMPs (chip multiprocessors), Cell broadband engine processor, ARM's bigLITTLE technology). Today's applications also started to become communication-centric. A huge part of the processor chip's power budget is used to transport the data packets from/to cores and also to the memory modules. Current and future multi-/many-cores will be hosting multiple applications to improve the performance per unit energy. The primary aim of sharing the CMPs by multiple applications is to improve the overall resource utilisation of the system. However, improving the resource utilisation without creating the resource contention and also the workload imbalance is a very critical task. The solution should be holistic and must follow Hardware/Software Codesign approach. Here, the software could be used for flexibility, and hardware for higher performance.

The nature of the execution flow in the applications can differ significantly. Thus, the proposed software solutions cannot be generic. The higher proportion of today's software application follows traditional control-flow approach. As a consequence, the state-of-the-art hardware is also tailored to that. There is another execution model that execute threads based on the availability of input data called *Dataflow execution model*. The advantage of this model is that independent portion of the application can be executed in parallel when their input data are available. In this doctoral work, dataflow based execution model is used. Parallelism can be achieved at different levels. Applica-

tion threads allow parallelizing code inside the applications to reduce overall execution time. Most common form of parallelism are *i)* instruction-level-parallelism, *ii)* data level parallelism (DLP) and *iii)* thread-level-parallelism (TLP).

Dataflow architecture was initially introduced for ILP [Den80]. It was also initially implemented in the form of "restricted dataflow" in a superscalar processor [HP86]. Later, a similar concept had been investigated by [AN90] also to support threads. Soon this effort led to an explicit token-store (ETS) architecture based machine known as *Monsoon* [PC90]. The dataflow execution model also supports DLP, but this work is focused on the TLP support of dataflow. During TLP, the total execution time would be equal to the execution time of the longest-executing thread if all threads in an application started to execute independently. In applications, TLP can also be exploited using other execution models (such as fork-join, master/slave, divide and conquer). Current CMPs can run a variety of complex applications consisting of a massive number of concurrent threads. Today, a single core can support concurrent execution of multiple threads (known as *simultaneous multithreading* (SMT)). Well-known hardware such as Intel's core architecture supports two threads each core and Xeon Phi co-processor that can support four threads per core.

Today's applications are long running, multithreaded and have different resource requirements. Only the microarchitectural advancements are not enough to execute applications in the best possible way along with proper programming model. Both hardware and software support for threads is also necessary. Hardware-based support for explicit multithreading can facilitate smooth execution of fine-grain threads and offer advantages over managing them at the software level. Conversely, software based thread supports can lead to higher performance loss (as the number of thread increases). The situation may get worst for large, complex and long running parallel applications. In general, dedicated hardware module offers low overhead and faster execution time, but requires altering existing hardware. On the other hand, software components are easier to add and manage, while new hardware support could increase performance. Before proposing a dedicated or new hardware module into the existing system, we should consider the associated design time, also the increased area and power cost.

The functionalities and also the complexities of manycore chips are increasing due to the immense evolution of chip's microarchitecture. Different applications have different resource requirements. A typical hardware execution unit consists of various dependent components (such as cores, local memory, DRAM controller, I/O controllers) and software plays a crucial part to tie them together for executing applications. With increasing core counts, contention reduction among shared resources is a challenge. The interconnection subsystem is becoming a vital component for better performance for manycore chips. Stable performance becomes dependent on the contention of other cru-

cial functional components (such as memory hierarchy and interconnections) as the core count of the chips are increasing. Today's, popular interconnection subsystem for CMPs is known as Network-on-chip (NoC) which is mainly an embedded switching network to interconnect the processing cores inside a CMP.

Multiple topologies are available or proposed from the scientific community. Two-dimensional mesh (2D-Mesh) is the most popular topology. In recent years, the NoC research domain has matured enough. Managing threads at low-level have several advantages (such as fewer hotspots, better performance, efficient power utilisation are few among others) unfortunately managing the threads at NoC level has not gained more attention. NoC is not just a topological layout of processing cores inside a silicon die, but also an effective network to route the data packets between the cores and also between the primary memory. Here, we advocate for the adaptation of NoC level thread management. This approach can facilitate the adaptation of low-level thread management for better control of data packets and also the floor planning for energy efficient execution. This method could also adapt itself to operational conditions (such as hotspots). In this work, a dataflow-based program execution model (PXM) has been used to map and distribute threads. Next, a software defined NoC, and also a hybrid NoC topology has been proposed keeping in mind to be used with the proposed thread distribution mechanism.

1.1 Thread management issues

The application threads are executed in a dynamic environment. For the smooth and efficient execution of applications, the system architecture needs to dynamically adapt the "ever changing" situations with minimal commotion to the functionality they offer. The change can come from various sources (such as resource contention, failure of hardware components, or may be a specific requirement of running applications). The known issues can be broadly grouped into two groups. They are:

- **Non-uniformity and in-elasticity** Each application (no matter whether they are a regular type of application or irregular type of application) may be composed of several hundreds of concurrent threads. A single solution may not be appropriate for all types of application threads, so customisation is needed for better performance. The main issue of today's applications is that their natures are not predictable easily. Managing threads at runtime using available resources are not easy. The traditional approaches to tackling the runtime thread management limit the overall flexibility of the underlying hardware.

- **Runtime prediction** Runtime prediction for resource requirements by threads is very complex to envisage. It is very complex to forecast the influence of the appli-

cation on the current system. There also exists I/O stalls or hardware component failures to make the situation more complicated.

Resource sharing opportunity arises when multiple computing elements connect inside a system. A proper resource management must be in place to harness the inherent facilities in a distributed computing platform where opportunities [MHH+15, CGMP10] of sharing processor core's content, hardware resources [GAD+13], and other services exists. However, developing an effective resource management ecosystem for "opportunistic" computing environment is not easy. An application is a collection of multiple sub-tasks which can communicate each other and also may be dependent on each other. These scenarios are ideal for I/O stalls or memory stalls. Furthermore, this situation can lead to reduced performance. Thus, we need to make a (low-level) ecosystem which is simple, scalable and yet efficient enough for executing applications at higher performance and with lower energy cost. In general, multiple efficient software, as well as the hardware based solution, have been proposed for addressing the issues belonging to the groups as mentioned above. Some of the recently proposed solutions also consider the current hardware microarchitectural changes (such as increased complexity at last level cache (LLC), prevailing heterogeneity at core level).

1.2 Research problem and associated solutions

Recent times power aware computing has gained importance. As a consequence, the microarchitectures are becoming more energy efficient. Power-aware approach somehow leads to multicores concept and later to the manycores. Apart from the increased core count, this approach also helps to steer the microarchitecture from homogeneous to heterogeneous. The increased core count offer the higher degree of parallelism. Interconnection and memory modules are the critical subsystems which are shared by all cores for data exchange. NoC is a scalable communication architecture that offers advantages compared to other alternatives (such as bus, ring) because both topologies suffer from high energy consumption, low scalability, and low bandwidth for connecting a large number of processing cores. Apart from that, NoC also offers better scalability, productivity and more deterministic performance [DT01].

For better energy efficiency and scalability, this research work advocate for an appropriate means of distributing and monitoring dataflow based application threads at NoC level. However, the proposed approach is also flexible enough to be used for control-driven applications. Results also show the efficiency of the simple hash based policy proposed for concurrent, large thread distribution model. A hybrid NoC architecture has been developed on the state-of-the-art FPGA device for underlying support. The pro-

posed hybrid NoC is application-aware by providing the support to scale (up or down) its size as per applications requirement. The primary research question that has been addressed in this work is:

How a NoC-based framework composed of multiple components (mainly for thread execution, distribution and monitoring) can improve the runtime adaptability of dataflow program execution models?

In other words, the goal of this research is to enhance the runtime adaptability of dataflow thread management policy by providing a scalable and efficient hybrid NoC topology. The framework can manage and also monitor the threads while lowering the energy consumption. The proposed hybrid NoC design is also flexible enough to adapt to the changing resource requirements of applications at runtime. The doctoral work solves the research problem via five steps. They are:

Step: 1 (Aim: Thread Distribution and Execution): How to easily distribute the massive number of concurrent threads at low levels so that it become energy efficient?

Article I (Chapter 3) gives the detailed the hash-based thread distribution scheme that can be very efficient with simple hardware modification. The proposed scheme targets dataflow execution model and offers abstraction and flexibility.
Based on the Paper: "*Enabling Massive Multi-Threading with Fast Hashing*" by Alberto Scionti, Somnath Mazumdar and Stéphane Zuckerman.
Status: Accepted for publication at IEEE Computer Architecture Letters (CAL).

Step: 2 (Aim: Thread Monitoring): How can we monitor the multithreaded dataflow applications?

The main aim of Article II (Chapter 4) is to propose a simple tool to analyse dataflow-based applications at runtime. It aims at a faster evaluation of hierarchical dataflow execution model. The output provided by the tool can be of great help in analysing the traffic generated by the scheduling activity.
Based on the Paper: "*Analysing Dataflow Multi-Threaded Applications at Runtime*" by Somnath Mazumdar and Alberto Scionti.
Status: Published at the 7th IEEE Advance Computing Conference (IACC-2017).

Step: 3 (Aim: Thread Execution Support): How to manage threads at Software-defined NoC level?

Article III (Chapter 5) discusses this challenge by proposing a scalable *Software defined NoC* (SDNoC) architecture for future manycores. The proposed interconnect allows mapping different types of topologies (*virtual topologies*). In this work, the software layer can directly control the network topology to accommodate different application requirements and communication patterns.
Based on the Paper:"*Software defined Network-on-Chip for scalable CMPs*" by Alberto Scionti, Somnath Mazumdar and Antoni Portero.

Status: Published in IEEE International Conference on High Performance Computing & Simulation (HPCS), (pp. 112-115), 2016.

Step: 4 (Aim: NoC microarchitecture): Provide an efficient NoC microarchitectural design to exploit the network traffic localisation for better traffic management at low energy cost.

Article IV (Chapter 6) gives a detailed account of a hybrid, scalable and efficient NoC microarchitecture that is designed to support future manycores chips. Similar to Article III, it also fuses ring and 2D-Mesh topology to provide high-performance while processing local (rings) and global (mesh) traffic efficiently. The results show that it is indeed efficient compared to the traditional 2D mesh topology.
Based on the Paper: "*A High-Performance Interconnect for Future Scalable Manycore CMPs*" by Somnath Mazumdar and Alberto Scionti.
Status: Under review at Journal of Parallel and Distributed Computing, Elsevier.

Step: 5 (Aim: Thread-to-Core Mapping): Provide an efficient analytical mapping model to certify the mapping of threads onto the cores are optimal or not.

Article V (Chapter 7) proposes a mixed integer linear program based formulation to map threads on cores at worst-case scenario by keeping into account the spatial topology of 2D-mesh NoC. The proposed analytical model is general enough to consider a different optimising policy (either optimising energy or latency) together with a variable number of memory controllers.
Based on the Paper: "*An Analytical Model for Thread-Core Mapping for Tiled CMPs*" by Marco Pranzo and Somnath Mazumdar.
Status: Under review at IEEE Transactions on Computers.

1.2.1 Solution overview

The proposed framework incorporates a flexible dataflow thread management mechanism that can efficiently distribute dataflow threads across the available processing cores. The proposed hybrid NoC topology can manage the traffic inside the NoC first by dividing them as local and global. Apart from that, the topology exploits the idea of "traffic localisation" to improve the overall energy cost to transport data packets. The contribution on the main research issues related to the problem is described in figure 1.1. The figure presents the target architecture with its basic functional block diagram, and the dotted line represents the contribution of this work. The proposed framework relies on three main components:

- **Dataflow based approach:** is used to enhance the degree of parallelism available in a dataflow application. Dataflow mainly explores the "spatial parallelism" upon the input available. Spatial parallelism is based on the concept that a thread

can execute in parallel if there exist enough space (resources or processing cores) and the associated input is available. We consider the dataflow model as it has fewer management issues and the available general purpose hardware can be used to exploit the inherent parallelism of the dataflow applications.

- **Thread distribution and monitoring:** is a critical feature while managing a huge amount of concurrent threads at runtime. In the framework, hashing based an effective thread distribution policy is used with a small amount of HW overheads. This work also proposes an analytical model for evaluating the solution quality of mapping threads on the free cores (in a 2D-mesh NoC). It provides a (theoretical) optimum solution for variable NoC sizes. Finally, a simple, yet powerful tool has also been proposed to monitor the threads at runtime.

- **Hybrid NoC topology:** takes into account the infrastructural maintenance issues during the thread execution. To counter the runtime management issues, a hybrid NoC architecture works in such a way so that with increasing data traffic, the system is stable with the continuous rise in throughput and also with better power consumption compared to the traditional 2D-mesh topology. The current state-of-the-art NoC subsystem is complex enough to maintain the threads at such low level with efficient topology layout and smart thread management but with a few overheads for implementing hardware-based thread distribution policy.

1.3 Thesis structure

The thesis is organised as:

- Chapter 2 presents **state-of-the-art** of the related works. It presents the basic information about the used programming model, with its execution models along with its related hardware supports. The brief information about some current manycores and the NoC has also been given.

- Chapter 3 presents **hash-based thread distribution** scheme for manycores. It describes how this distribution scheme can solve massive concurrent thread management issues at the hardware level with little overheads.

- Chapter 4 details a **runtime analysis tool** that can be used to evaluate the dataflow applications. It aims to be a simple simulation tool for fast evaluation of applications adhering to dataflow PXMs.

- Chapter 5 presents a scalable **Software defined Network-on-Chip (SDNoC)** architecture to provide the execution support for threads based on dataflow PXMs.

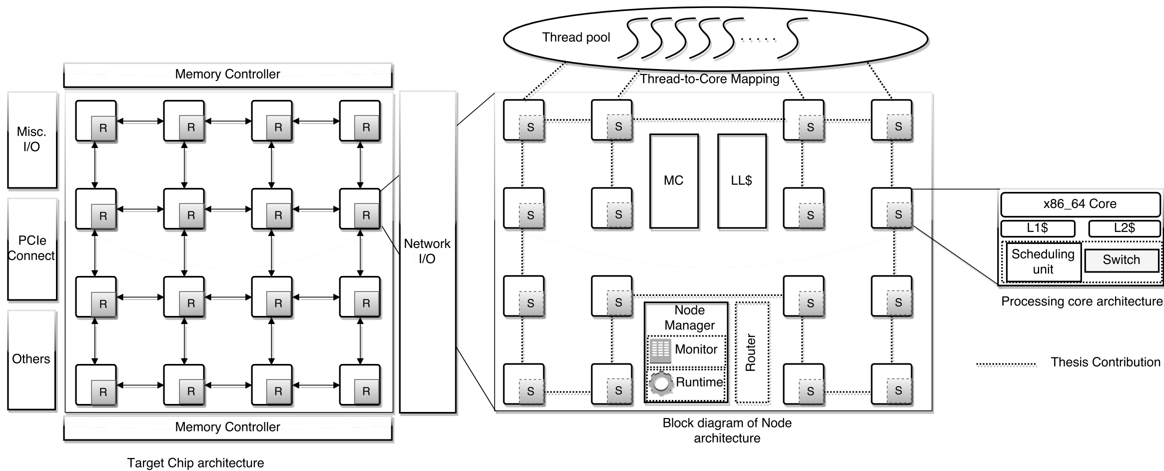


Figure 1.1: Solution overview: Target chip overview for the manycores clustered processor. Cores are organised into clusters (i.e., nodes) connected each other through a 2D mesh. Within each node, cores and other shared resources are linked via a ring. Each node has a dedicated node manager that controls the runtime and also monitor the system. Each core has a local scheduling unit to distribute the threads. Blocks that are addressed in the thesis are highlighted via dotted line.

- Chapter 6 details a proposed **hybrid NoC implementation** that have been developed to support both control-driven and data-driven execution model with better scalability, throughput and improved latency along with lower power and area cost.

- Chapter 7 presents an **analytical model** to map application threads on to the free cores to achieve optimal performance (such as energy cost and latency).

- Finally, Chapter 8 summarises the chapters and also discusses some perspectives for future research works.

2 Background

This chapter discusses the dataflow threads and also provides a generic overview of current hardware together with the relevant description about Network-on-Chip (NoC). The discussion is presented in three parts: Section 2.1 describes the brief summary of dataflow threads, including its program execution models (PXMs) and associated dataflow languages. Next, Section 2.2 discusses the current multicore domain from general purpose architecture (mainly focusing on the manycore architecture and the heterogeneity) perspective and also about the dataflow based hardware support. In this chapter, a brief overview of the FPGA device has been given as the proposed customised NoC design has been implemented on FPGAs. Finally, in Section 2.3, we discuss the NoC including its traditional topologies and the related works that are relevant to this doctoral work.

2.1 Dataflow threads

Today's multi-/many-cores support massive level of (thread) parallelism. For better support, the functional components and its capabilities are changing. Current systems now wrap hundreds of processing cores in a single silicon die to execute a huge number of concurrent threads. Simultaneous execution of multiple threads reduces the latency occurred in the system so that performance can be improved. An application consists of a collection of tasks which further can be abstracted into multiple threads. Multiple threads can run in parallel if they do not become dependent on each other. It is always worth to be noted that more the parallelism exists in the application code, better the application would be parallelized. In Figure 2.1, we have shown the canonical way to achieve parallelism at different levels from an application. In coarse grain, an application can be divided into multiple tasks which work on particular data sets. Next, Each task can be further split into multiple threads known as thread-level-parallelism

(TLP, see Figure 2.1 (middle)). In Figure 2.1(right) a thread is divided into a set of instructions (blocks) to provide instruction-level-parallelism (ILP). With large core counts,

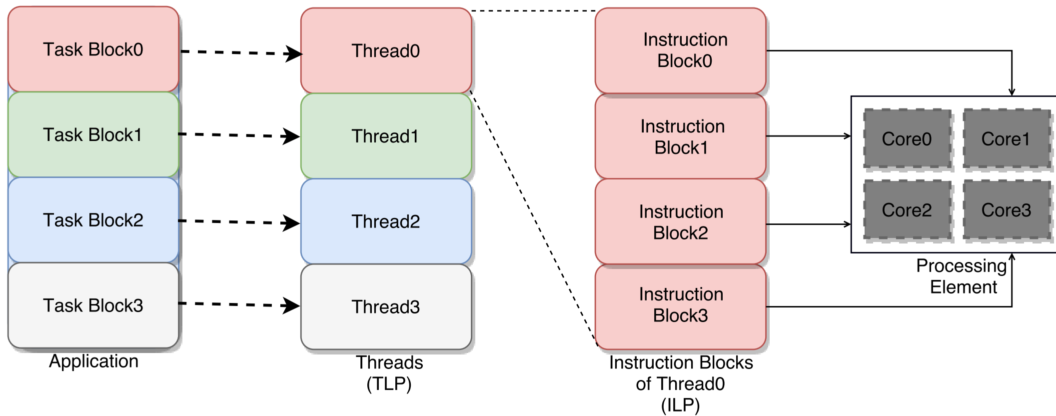


Figure 2.1: Block diagram of parallelism: application level parallelism (left), thread level parallelism (middle) and instruction level parallelism (right)

each core can generate an enormous amount of data traffic inside the chip. It can further lead to a resource contention (such as I/O stalls or memory stalls). Hence, it is needed to efficiently manage the increased level of parallelism to avoid performance degradation. In general, the program execution flow is governed by either control dependency or data dependency. In control dependency based execution, threads are instantiated when some conditions are met. In data dependency model, threads are started to run when needed input data are available (see Figure 2.2). Most of the conventional (high-level)

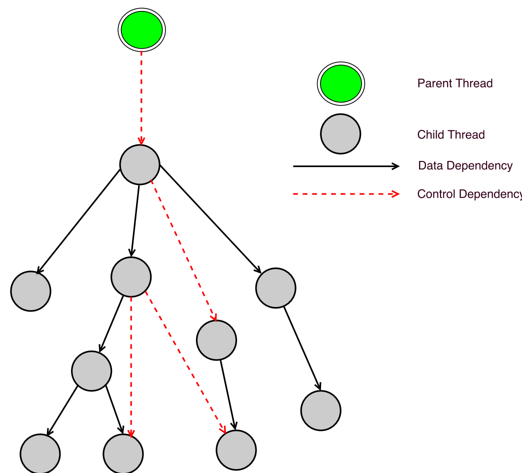


Figure 2.2: Thread execution using control and data signals

programming languages are control driven, and dataflow programming languages are mainly data driven. The available programming models can be broadly classified into five groups. They are: *i)* shared memory based programming model, *ii)* distributed

memory based programming model, *iii)* partitioned global address space (PGAS) programming model, *iv)* dataflow programming model and *v)* heterogeneous programming model (displayed in the Figure 2.3). Until now, the most popular program execution

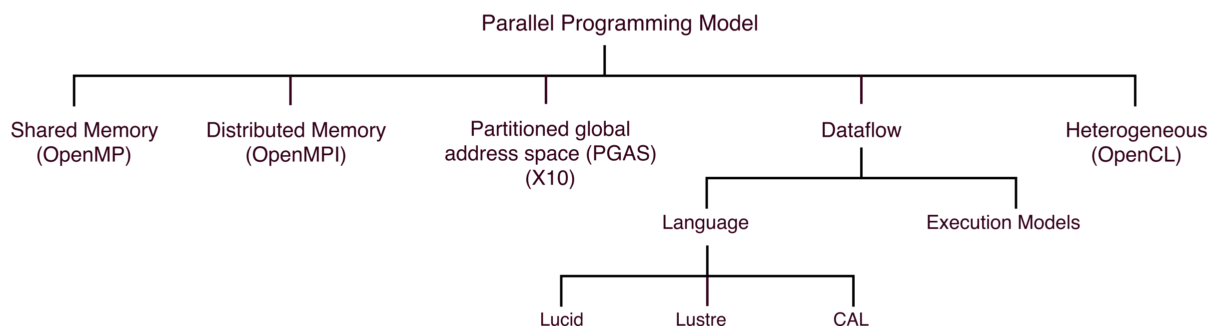


Figure 2.3: Programming models classification

models are based on von Neumann architecture. Due to its limitations (such as memory latency, synchronisation) [I+88], dataflow model of computation has gained popularity. However, the roots of the dataflow execution model date back to the early 1970's. Dataflow [Den80, Den86] is an asynchronous as well as synchronous [KGA01] distributed computation model. Dataflow threads are set of synchronised and scheduled instructions. It can support non-preemptive execution, but the compiler controls thread granularity. In general, dataflow model can be classified either as static [DM75] or dynamic [GKW85]. Applications adhering to such program execution models (PXMs) follow producer-consumer model, which offers a natural way for synchronising parallel activities. By allowing threads to schedule the execution of other consumer threads, the synchronisation mechanism set to the required number of input data to an appropriate consumer thread. The dynamic form of dataflow can support a higher level of parallelism by supporting repetitions [GP+77]. An application written based on dataflow language creates a directed acyclic graph (DAG). In DAG, each node represents a thread, and the arc is the data path to other threads. Computation occurred when the needed data arrived at the nodes, and dependent threads will not resume until their data arrives at the nodes via the arcs. All the nodes that have input data available can immediately start their execution. In this doctoral work, the employed dataflow threads exhibit some features such as *i)* a dataflow thread triggers its execution only when all input data are available, *ii)* it does not support any jump, *iii)* simultaneous read-write is not permitted, and *iv)* supports shared memory model. During the execution, a dataflow thread can have any one of the following states:

- Waiting: when all the inputs are not available.
- Ready: when all the inputs are available.

- Execution: thread is assigned to a specific core for its execution.

Yazdanpanah et al. in their paper [YAMJGE14] classify hybrid dataflow/von-Neumann models compared to block level execution model, control and dataflow execution models. Nowatzki et al. in their work [NGS15] claim that dataflow model can achieve good performance by reducing energy up to 40%. In this paper, authors also argued that if a core provides support for both dataflow and out-of-order execution mode, then applications can achieve better performance at lower cost (by automatically switching from one mode to another).

2.1.1 Dataflow execution model

Unlike conventional control driven execution, in the dataflow execution a DAG is used to represent the data dependencies among the threads. It allows the consumer thread to run once all the required inputs are available. This way of triggering the execution helps to exploit the TLP because it reduces the amount of traffic generated by the synchronisation activities on multicores. Dataflow-based PXMs offer a lower synchronisation cost with a better use of processor resources. Hardware support for such PXMs are implemented in various early architectures ([CGSVE95, $N^{+}90$]). Both High-Performance Computing (HPC) and High Throughput Computing (HTC) applications, as well as Cloud Computing, can significantly benefit from the adoption of dataflow PXMs. Some dataflow based PXMs are: DF-Threads [GF14], Codelets [SZG13], Data triggered threads (DTT) [TT14, TT12, TT11], Data driven multithreading (DDM) [KET06] and Scheduled Dataflow (SDF) [KGA01].

- DF-Threads is a variant of dynamic dataflow that can be easily interfaced to libraries and hardware implementations.

- Codelet is a fine-grained dataflow model for supporting multiple nodes connected via an interconnection subsystem. A codelet represents a non-preemptive, single unit of computation. Codelet relies on an abstract machine model (AMM). It provides an abstraction of the features which is required by the hardware processors to support the thread execution. The two levels adopted in the Codelet model allows better use of the data principle of locality. For efficient execution, multiple codelets are connected to form a codelet graph (CDG).

- Data triggered threads (DTT) proposed a dataflow-inspired execution model called *data-triggered thread execution*. It has been suggested for CMP and SMT domain. Software supports for DTT can run on any parallel machines. CDTT is a compiler framework which supports C/C++ and generates data-triggered thread executable (which can run by the runtime). It adds four new instructions for runtime support

and an extra hardware support for house-keeping jobs. DTT tries to exploit the redundant computation to speed up the execution.

- Data-driven multi-threading (DDM) execution model has inherited data availability based on execution feature from dynamic dataflow model (more precisely from decoupled data-driven (D3) graphs [EG90]). In DDM, applications are partitioned into a data-driven synchronisation graph and threads at the compile time. DDM effectively separates threads that are data dependent and independent. DDM model requires storing the application dataflow graph (DFG) locally to select ready threads and scheduling new ones. Similar to conventional dataflow, this non-blocking mode of execution is another way to hide synchronisation and communication latencies by running other independent threads in a shared memory address space.

- Scheduled dataflow (SDF) architecture represents a decoupled memory/execution implementable on multithreaded architecture using non-blocking dataflow threads. Both SDF and Codelet supports self-scheduling property where it is not required to store the application DFG, and the threads are dynamically scheduled at runtime. However, the Codelet model differs from the SDF for the hierarchical organisation of the threads.

Recently some of these PXMs also receive some hardware supports ([STE06, TT11, GS15]).

2.1.2 Dataflow languages

Some of the well-known dataflow languages are Lucid [AW77], Lustre [HCRP91], actor model based (such as CAL [EJ03]) (see figure 2.3).

- Lucid programming language is a non-procedural/functional language. Dataflow computation can be achieved using its temporal operators. In Lucid code, the statements are mainly equations. It is described in such a way so that a calculation of rational numbers can be done.

- Similar to Lucid, Lustre is also an another functional language. It is a synchronous dataflow programming language. The Lustre code has three main parts: clocks, flows and nodes. The code can be compiled to multiple target languages (such as C).

- CAL is a dataflow-inspired actor based programming language. CAL describes modular, non-shared dataflow components called *Actors*. Actors are threads running C or C++ language and can communicate with thread-safe software FIFO

buffers. The FIFO's are protected by the mutex, but it suffers from high latency issues. Actors only can communicate via I/O ports. Actor intakes token, changes its state and produces another token. During execution, CAL produces a network of actors. Reading a thread will suspend until a producer thread does not produce all needed tokens/inputs for input port (data-driven approach).

2.2 Hardware overview

Today's HPC applications have two main requirements: *i)* higher performance and *ii)* better energy efficiency. In recent years due to the immense growth in silicon industry, the microarchitecture also started to become complex and powerful (e.g., the prototype of a Kilocore which connects 1024 cores in a single system [BSP+16b]). The changes can mostly be grouped into two classes: increased core count (lead to multicore and now manycore chips), and different processing capability fused inside a single die (heterogeneity). The increased core count provide immense computing power and to support them multiple programming models also started to develop. The system with increased core counts are also known as *accelerator* or *co-processor* (such as Intel Xeon-Phi). However, in recent time, building low powered single board computer with very good amount of computational power is getting popular, and few of them are Parallela, Kalray's MPPA 256. The growth in core count will help to share the chip area and resources to run multiple applications concurrently but the concurrent execution of multiple threads will immensely stress the system. So proper resource management together with heat removal mechanism must be incorporated into the system. In this section, the discussions will be focused on some (co)-processors which either supports control-driven execution or data-driven execution.

2.2.1 Hardware support for control-driven execution

2.2.1.1 Intel Xeon-Phi co-processor

Many integrated core (MIC) architecture is an HPC accelerator built upon X86 architecture and manufactured using 22-nanometer lithography. Xeon Phi has processor cores that can run its own operating system. Each core has an L1 cache (of 32 KB for data and instruction), L2 cache (of 512 KB for both instruction and data) but not L3 cache. All L2 cache have their *tag directories* and translation lookaside buffers (TLBs). These distributed tag directories are used to provide uniform access and to track cache lines in all L2 caches. It supports MESI (Modified-Exclusive-Shared-Invalid) protocol for cache coherency and memory coherency. Each core is multithreaded (up to four threads)

and follows in-order execution. Cores are interconnected by a bidirectional ring topology (dual-ring). MIC supports single instruction, multiple data (SIMD) taxonomy and also supports specialised SIMD instruction sets (such as advanced encryption standard new instructions (AES-NI), MMX, or streaming SIMD extensions (SSE) (extension of MMX)). Apart from that, the accelerator card only understands X87 instructions. Phi micro-architecture is based on scalar pipeline and vector processing. Each core has a dedicated 512-bit wide vector floating point unit (VPU) and is crucial for the higher performance. It also supports fused multiply-add (FMA) operations. If FMA is not used, then performance will be turned to half. Each core can execute two instructions (one on U-Pipe and another on V-Pipe) in its core pipeline in every clock cycle. Only one floating point or vector instruction can be executed in any one cycle. The instruction decoder is a two-cycle unit and hence, at least two threads are needed to attain maximum core utilisation.

Recently, commercial manufacturing of single board computers with multicore chips and an accelerator (on the same circuit board) have started. Below two are the example of such systems.

2.2.1.2 Parallella

Parallella [Ada13] is a scalable, open-source, superscalar, RISC (Reduced instruction set computing) based manycore architecture which relies on MIMD (multiple instructions, multiple data)-based execution model. It provides scalability and low power consumption (e.g., using 5 volts 16-core based Parallella board can attain 32 GFLOPS as peak performance). Parallella is based on Epiphany system-on-chip (SoC) that wraps Zynq with dual-core ARM Cortex-A9 as host processing core. ARM cores have its kernel and follow master-slave execution model. The Epiphany architecture has been designed to extend the floating point computing power with very low energy consumption. Each co-processor is of 32-bit wide with the max clock speed of 1 GHz (average 500 to 600 MHz) and has variable-length instruction pipeline. Each core mainly has six components one each for integer operation, floating point operation, 64-bit general purpose registers, program sequencer, interrupt controller and debugging unit. Similarly, to Xeon Phi, it also supports FMA operations and can perform two floating point operations at every clock cycle. Parallella supports 32-bit wide little-endian based flat shared-memory architecture, and each core has unprotected, a local memory (both for data and instruction) of size 32KB that are further divided into four sub-banks. Each co-processor has a unique global address and can transfer 8-bytes of data or instructions at every clock cycle. Each core connected by 2D low-latency NoC. It provides three lines for reading and two for writing (one for off-chip and on-chip) operations. It is based on ANSI-C/C++ and OpenCL programming environments. The read operations are non-blocking and the

read-write operation for local memory follows *strong memory model*, but the read-write operation for non-local memory does not follow any strict order execution. However, memory constraints and memory access issue (such as concurrency or memory quirks) are big problems for it's performance in real applications.

2.2.1.3 MPPA-256 processor

Kalray's MPPA-256 is a multi-purpose processor array (MPPA) based single-chip many-core processor that is built using 28nm CMOS lithography [dDdML+13]. The MPPA includes quad-core CPUs coupled with the manycores. Each MPPA core is based on a 32-bit very long instruction word (VLIW) architecture and also comes with an FMA functionality. Each core has it's L1 instruction and data cache. It wraps 256 processing cores and 32 *system cores* on a chip (thus many counts total 288 cores). Each compute cluster consists of 16 identical cores with own standard FPU and memory management unit (MMU). The cluster cores work non-preemptively, which is ideal for very specialised computation, but not suitable for the high-level application. This MPPA architecture consists of an array of clusters linked by two 2D torus-based NoC (one for data movement and the other for control). The NoC provides a full duplex communication between clusters and a lightweight POSIX kernel is running inside each cluster. In every clock cycle, it can support up to five 32-bit RISC-like integer operations. The processing cores in the array used their local memories and dedicated DMA to perform their global memory addressing. It seamlessly supports C based dataflow and pthread based execution models and is also well-suited for applications such as image, audio, signal processing.

2.2.2 Hardware support for dataflow execution

The hardware support for dataflow execution has started from early 1990's. Dataflow inspired architectures are meant to offer low cost, efficient and flexible platform for dataflow computational model. Dataflow approach can offer the highest level of parallelism if the application has a huge number of concurrent independent threads. The dataflow systems support both static and dynamic dataflow based execution and also the explicit token-store architecture (such as Monsoon). Few known dataflow supported hardware are Monsoon [PC90], Efficient Architecture for Running THreads (EARTH) [The99], WaveScalar [SMSO03] and the latest Maxeler processing platform [PL13].

2.2.2.1 Monsoon

Monsoon is an explicit token-store (ETS) based architecture prototype for dataflow based execution. In ETS architecture, token carry pointers and each token descriptors are addressed in a global memory which is partitioned among the cores. In Monsoon, each

core supports the eight-stage pipeline. In instruction fetch (IF) stage, an explicit token address is computed from the frame address and an offset. Next, the availability of the operands is checked. When data arrived, it is stored in the frame slot of the frame memory. When all the data arrived, the execution stage starts.

2.2.2.2 WaveScalar

WaveScalar is a tile-based processor architecture for executing conventional pthreads, and also dataflow threads. To support multithreading, WaveScalar extended its instruction set. Its thread spawning mechanism tries to parallelize small loops inside the application. WaveScalar supports threads that do not have their stack and cannot make functions call but has it's thread identification. WaveScalar provides each thread with a consistent memory view and lets application to manage the memory ordering directly. To support the concurrent thread execution, it supports multiple, independent sequence of ordered memory access.

2.2.2.3 Efficient Architecture for Running THreads (EARTH)

EARTH belongs to the group of hybrid dataflow/von Neumann execution model. The PXM of EARTH was customised to support dataflow threads. For efficient runtime execution, programs are divided into *threaded procedure* (similar to functions, but differ on frame allocation, thread invocation, scheduling and parameter passing) and *fiber*. Fiber supports three states during its lifecycle such as enabled, active, and dormant. Fiber supports sequential execution of its instruction sets and non-preemptive execution. When a thread is ready, the system enables the fiber, and it executes the threaded procedure. Procedures are invoked automatically by the application but can be terminated explicitly. After invoking the procedure, the system creates a context for the procedure and executes other housekeeping jobs. Finally, when execution is complete, fiber is removed from the processor, but the associated threaded procedure may stay alive.

2.2.2.4 Maxeler

Maxeler is an FPGA-based platform which could be a promising candidate for accelerating the HPC applications. It is a combination of synchronous dataflow, vector, and array processors. This platform has PCIe-based connectors to connect X86 processors, and FPGA to offload the task. The Maxeler ecosystem maps the application on the FPGA using static graph. Maxeler processing platform comprises of multiple dataflow engines (DFEs) with their local memories connected to the host CPU via interconnect. Multiple DFEs are linked by a high-bandwidth interconnect called MaxRing. DFEs execute the

tasks in a dataflow fashion and supports both fast and large memory. DFE can manage one or multiple kernels (for main execution) and a manager for data movement within DFEs. Maxeler has its compiler to generate dataflow implementations which can then be called from the CPU via a special interface called SLiC (Simple Live CPU). Together with the compiler, it also has it's software middleware between the Linux and DFEs for runtime data transfer and optimisation. Maxeler is an FPGA-based accelerator which comes with it's programming methodology to help to develop a customised energy efficient (runtime) system.

2.2.2.5 Transport Triggered processor

Transport triggered processors (TTPs) are mainly based on the scalable transport triggered architectures (TTAs) where operations occur as side effects of data movements [Cor97]. TTP is an evolution of VLIW processor while TTP directly sends data from one fetch unit (FU) to another without involving the registers or memory [Cor94]. TTPs (are of single instruction multiple transports type) support ILP and can also be used as application-specific architectures. Unlike others, TTAs do not include any instruction set. Thus the programmer defines the data movements between FUs. The memory access speed in TTP is low, and data transport defines the cycle time of the processor. Adding new FU to TTA is easy, in turn, addition also linearly increases the complexity.

2.2.3 Heterogeneity

Increased core count does not only offer a higher number of cores but also pose challenges (such as concurrent programming issues, controlled power consumption, and improved scalability). The heterogeneous system architecture (HSA) is an approach where it tries to achieve better parallelism using low-power cores with different capabilities [KFJ+03]. Asymmetric CMPs (ACMPs) [SPS+07, KTJR05] are one of the first successful approaches to embed heterogeneity into the CMPs. The heterogeneity does not only improve the power but may also successfully satisfy the application requirement. Generally, ACMPs contain one or multiple large, powerful, out-of-order cores and also few small, simple and power efficient cores on the same die. ACMPs can effectively accelerate both fine grains and coarse grain thread execution.

A popular heterogeneous computing model is created when a host processor (such as general purpose core) combines with an accelerator (such Xeon Phi). In this ecosystem, host CPU works as a master while Xeon Phi is used as an accelerator to speed up the whole process (slave). However, the offloading cost of computation must be computed before initiating the task off-loading. Based on the amount of task, the offloading may

become costly. Super computer (such as *Tianhe-2*) is also built upon CPU-Xeon Phi ecosystem.

Commercial product such as ARM's heterogeneous multicore big.LITTLE [ARM13] is another example of multicore ACMPs. It wraps a powerful application processor (Cortex-A15) with simpler cores (Cortex-A7). This architecture gives a trade-off between high performance and low power consumption. To schedule the jobs on these cores, two possible ways are CPU migration and task migration. In CPU migration, processes are mapped first on the simpler core and later moved to a larger core. In real time execution, the scheduler only sees one logical core for each set of big.LITTLE cores. However, in task migration, all the cores are exposed to the scheduler. The main task started to execute on the bigger core and simpler tasks are performed on the simpler cores. The cost of migration between core happens only at a coarse granularity. In general, the energy consumed by an instruction is partially related to the number of pipeline stages it traverses.

2.2.3.1 FPGA

Field-Programmable Gate Arrays (FPGAs) offer a quick prototype of hardware designs and also re-designing features via reconfiguration capabilities. FPGAs can be programmed via various ways such as high-level synthesis (HLS), circuit schematics or by using a hardware description languages (such as VHDL, Verilog).

The two most important components of FPGA are configurable logic blocks (CLBs) or logic array block (LAB) and look-up-tables (LUTs). Every CLB consists of multiple LUTs, a configurable switch matrix, selection circuitry (MUX), registers and flip-flops (FFs). LUT is nothing but a hardware implementation of a truth table. LUTs are used to implement any arbitrarily defined Boolean functions in CLBs. LUTs are also used as small memories or small RAM. In general, there exist two types of FPGA: one-time programmable (OTP) FPGAs and widely used SRAM-based (can be reprogrammed as the design evolves). SRAM blocks are interspersed in the fabric and can be chained together to build deeper wider memories or RAMs. For processing data, FPGA also has hard IPs (such as a multiplier, DSP, processors–e.g., ARM Cortex-A9 dual-core MCU is used in Zynq-7000 SoC from Xilinx). Apart from that, there is also software-based core or softcore which can be a simple microcontroller or a full-fledged microprocessor. It has less clock speed compared to hard IPs. However, it can be easily modified and tuned to specific requirements, custom instructions (e.g., OpenRISC is a softcore). For a faster communication, FPGA also offers high-speed serial I/Os and all state-of-the-art FPGAs incorporate multi-gigabit transceivers (MGTs) for very high-speed communication.

2.3 Interconnection subsystem

Broadly, the research topics discussed in [SC13, MOP+09, OHM05] could be classified into multiple research streams: *(i)* microarchitectural domain (mainly deals with network topology, architecture, capacity management); *(ii)* the communication infrastructure (mainly proposing the models, switching techniques, congestion control, power management, fault tolerance); *(iii)* analytical methods for evaluating proposed NoC's performance; and *(iv)* mapping applications on the processing core.

For large core counts, NoC is a critical component responsible for better performance. NoC's architectural components such as channel width, buffer size, routing algorithms are very critical for better flow of the data packets inside the chip. Latency improvement, hotspot mitigation or deadlock free traffic movements are the main research issues in interconnection subsystem. During thread execution, all the core communication and the data transfer is done using interconnection subsystem. Inside the chip, last-level caches (LLCs), the bandwidth of interconnect as well as memory have become very critical for better performance. For better throughput and lower latency, the on-chip interconnect bandwidth should be high. Inefficient interconnection subsystem may lead to reducing the overall system performance and consume a significant portion of the area and power budget of the chip [HVS+07].

Apart from that, application mapping on manycore processors is not easy when performance constraints (such as power consumptions, latency, throughput) must be satisfied. Multiple processing cores are connected by NoC, which provides the shared communication medium for information exchange. Typically, a processing core of a tiled CMP consists of first level cache, last level cache, a network interface (NIC) and a router. The NIC manages the cache level information and breaks the information into the flow control units. Next, the messages are composed of one or multiple packets. The elements that mainly characterise a NoC are the topology, routing algorithm, flow control, and crossbar based router microarchitecture. Below we are describing them in brief (for more details, please refer to [DT04a]).

2.3.1 Topology

A key aspect of NoC-based interconnections is the topology, i.e., the way routers are connected to each other. The topology mainly describes the floor planning of routers to use the interconnect subsystem efficiently. It defines the hop count to refer to physical immediacy between the cores; more the hop count higher the latency for the packets are. The interconnection subsystem does not only connect cores, but also other functional components such as LLC, memory and DMA controller. Hierarchical or hybrid topolo-

gies started getting introduced, when designers started to face performance challenges for connecting multiple cores using simple topologies (such as bus or ring). In general, some of the well-known topologies are bus, ring, 2D-mesh, torus, flattened butterfly (discussed below (see figure 2.4)).

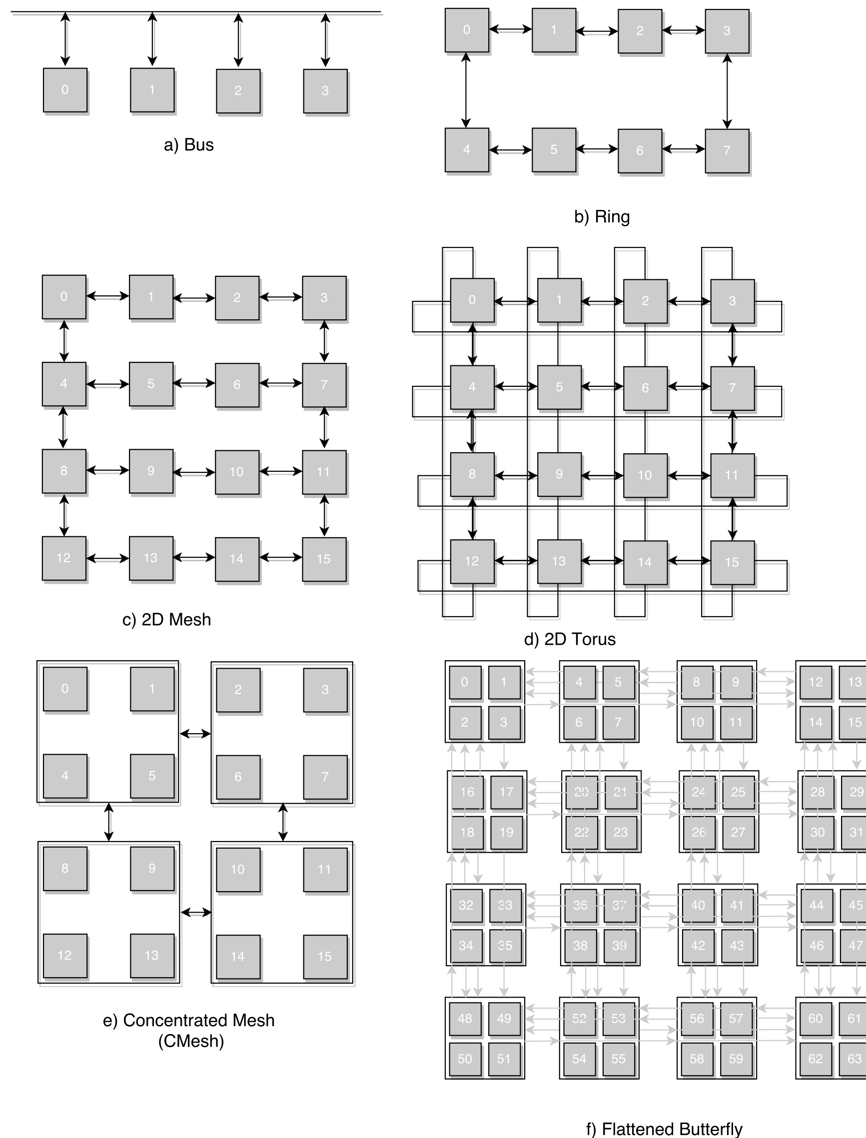


Figure 2.4: Some well-known topology for interconnect subsystem

- Bus topology is a simple, inexpensive topology that connects cores along a single connection wire. *Ethernet 10Base2* was a practical example of the first bus topology in action. However, the bus topology started to suffer from high energy consumption, low scalability, and low bandwidth for connecting a large number of processing cores. The main reason was the physical capacitance of bus wires

which started to grow with the number of connected modules, thus resulting in a growing wire delay.

- Ring topology connects multiple cores along a single wire in such a way it forms a closed loop. The ring topology is also a simple and inexpensive topology. It can continue to serve increased core count after exceeding its capacity but with slow speed. In general, scaling up or down the ring capacity can affect the network service. Similarly, for the ring, the average latency started to increase with the proportional number of cores inside the chip and making the bandwidth a potential bottleneck.

- In direct topology, cores are connected in a two-dimensional space. This kind of organisation helps to remove the routing overheads with increased core counts using the low radix routers. The 2D-mesh is an example of direct network topologies. In a 2D-mesh, all the core connecting wire length are equal in length. Thus the area and power consumption grow as the number of connecting core grows.

- Similar to the 2D-mesh, a 2D-torus is an another example of direct topology. Torus is also known as *k-ary n-cube*. It means that k number of cores can be connected in the n-dimensional space (in each dimension). It is interesting to note that ring is an example of *one dimensional torus*. Unlike 2D-mesh, the 2D-torus can lead to higher latency because the each edge cores are further connected to routers of the opposite edge via wrap around wires. This feature has alleviated in folded 2D-torus where the wires are of the same length.

- Concentrated mesh or C-Mesh network [BD06] is a modification of 2D-mesh to a radix-4 mesh where each router maintains four processing core of the network. It helps to reduce the latency (hops) proportional to the concentration degree of the network, while providing few number of channels with higher bandwidth. The C-Mesh network uses dimension-order routing with the express links for packet transfer.

- Flattened butterfly [KBD07] is another improved layout of mesh to reduce the latency for better communication among the cores. It is based on the high-radix routers (radix=10) and non-minimal global adaptive routing. It provides a maximum of two hops but with longer wires. Though the longer wire increases cost, it also provides a better way to store temporary packets intermediately.

For smaller core count, topologies such as bus, ring were very popular, but as the core counts inside the chip started to increase we need newer topologies such as the C-Mesh or flattened butterfly. However, for very high core counts hierarchical topologies

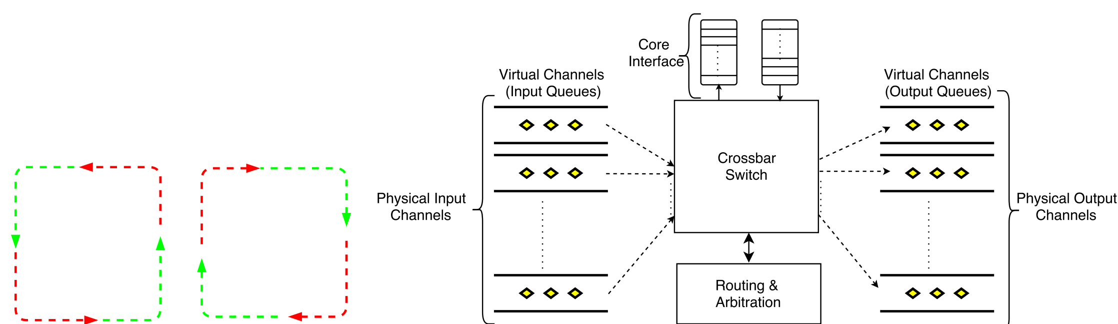


Figure 2.5: XY DoR routing direction (left) and block diagram of a mesh router architecture is presented (right)

have also been proposed (such as bus and 2D-mesh based [DEM+09], mesh and ring based [AFY+16] are few of them).

2.3.2 Routing

The routing algorithm selects the links that a packet must follow to reach its destination from the source. Two most common class of routing algorithms for NoC are adaptive routing algorithm and deterministic routing algorithm. In adaptive routing, packets are moved from source to destination using different links. However, when the links are busy, different links can be selected based on user defined metrics. A cyclic link is not created to avoid deadlocks. In the deterministic scheme, the same set of paths is always selected for the same set of source and destination cores. In this work, the XY dimension-ordered routing (XY DoR) (see figure 2.5 (left)) has been used. In this routing scheme, all the packets always traverse first in the X direction (i.e., east or west) and then turns towards Y direction (i.e., north or south) in the figure red colour is the forbidden path, and green colour shows the allowed path.

2.3.2.1 Flow control

Flow control (FC) controls the flow of packet by allowing it to flow along its allowed path and also to stay temporary in some buffers when the link is busy. FC optimises the packet latencies and also the throughput at higher loads. It works in close collaboration with the routing scheme to make sure that packets reach its destination. There are two types of flow control policy: circuit switching and packet switching. In this work, packet switching has been used. The packet switching follows store and forward mechanism. In this policy, resources are allocated to the whole packet. The entire packet is stored before forwarding to the next link. It can increase latencies when packet consists of multiple flits. Packet-switched flow control can be classified into store-and-forward (used in this

work), virtual cut-through and wormhole (mainly for flits) which are further explained in short below.

- As the name suggests, in *store-and-forward* (SAF) flow control, an entire packet must be received completely in the router before being forwarded to next router. For efficient storing, buffers and link bandwidth are allocated for the packet (based on packet size). It offers higher per-hop latency, but still, large buffers are needed for large packet size.

- Unlike SAF, *virtual cut-through* (VCT) flow control reduces per-hop latency for large packets but still need large buffers. It reduces the per-hop serialisation delay by forwarding some flits of a packet before receiving all flits of the packet.

- *Wormhole flow control* mainly works at flit level granularity. It is an improved control mechanism compared to SAF and VCT. Similar to VCT, it allows the flits serially to move to the next router given there is a space for it to store. It improves the buffer utilisation but suffers from the head of line (HOL) blocking. HOL refers to the problem where a flit of a packet at the FIFO queue gets blocked due to the congestion at its next router, then other packets behind it also get blocked.

2.3.3 Router microarchitecture

A basic microarchitecture of a crossbar switch based mesh router has been shown in figure 2.5 (right). The input channels of a mesh router have input buffers. Each input channel connects to a crossbar switch and then connect to any output channel. The input buffers consist of multiple virtual channels (VCs). In general, routers are pipelined to improve the packet latency. Logic (router wraps I/O ports, route compute unit, an arbitration logic, and VC status table) are used at every step to make sure that packet arrives at it's destination based on the header information. There are three main operations performed by the router. They are route computation, switch allocation (SA) and VC allocation (VCA).

In route computation, the packets are routed based on the routing algorithm. In SA process, packets are arbitrated to access the crossbar switch. It is mainly a mapping issue between the VCs of the router to the free output port of the router. VCA makes sure that the whole packet gets a VC at the next router. Round-robin arbitration is widely used arbiter and has been used in this work because it provides better fairness. Allocator maps incoming requests to available resources (VCs and crossbar switch ports), while arbiter matches requests to a single free resource.

Table 2.1: Theoretical performance of 8x8 mesh for six synthetic traffic patterns

Traffic Pattern	**Avg Hop Count**	**Throughput (flits/nodes/cycle)**
Uniform Random	5.25	0.50
Bit-Reverse	5.25	0.14
Transpose	5.25	0.14
Bit-Complement	8.00	0.25
Tornado	3.75	0.33
Shuffle	8.00	0.25

2.3.3.1 Traffic patterns

There are six well-knows traffic patterns to represent the real-time application's traffic behaviour. The patterns, their theoretical latency (average hop count) and theoretical throughput for 8x8 mesh router using with XY-DoR routing are presented in table 2.1. It is worth to note that the injection rate was one packet per cycle (worst case scenario). From the table, we can see that the mesh has performed best in the uniform random traffic pattern, but uniform random traffic pattern does not identify the load imbalance of the design because in this pattern every source with equal probability can send packets to every destination. Hence it is always recommended to analyse the design using multiple traffic pattern. In this work, first three traffic patterns are used.

2.3.4 Hybrid NoC architectures

In the past years, NoCs received much attention from the research community [ODH+07, BM06a]. Where some of the works focused on proposing low latency router microarchitectures ([KPKJ07, HVS+07]) and power efficient microarchitectures ([WPM03, MCM+04]), other researchers focused on proposing different topologies. For instance, Dragonfly [KDSA08] and Flattened butterfly [KBD07] are few among others. Other works (such as [AFY+16, DEM+09, BZ07]) tried to improve the performance-power consumption trade-off through the introduction of hierarchical NoC topologies.

2.3.4.1 Ring and mesh-based approaches

HiRD [AFY+16] is a hierarchical ring-based NoC design for improved energy efficiency, where buffers within individual rings are not used. It provides buffer support between different levels of the ring hierarchy, and upon the saturation of buffers, flits are deflected in the rings. It needs four levels of hierarchy to connect 256 PEs. CSquare [ZGHC15] proposes a way of clustering routers so that clusters adopt an internal tree-like organisation. It is a topology with clusters forming a global parallel structure to provide high

scalability. The authors also showed that this topological design improves throughput, while lowers the average latency over mesh-like topologies under the uniform traffic pattern. Transportation network inspired NoC (tNoC) [KKM+14] is another proposed hierarchical ring topology. It employs hybrid packet-flit, credit-based flow control mechanism for better scalability, as well as priority-based arbitration for achieving better performance. tNoC allocates channels with a flit granularity, while buffers are allocated with a packet granularity for reducing buffer counts. Koohi et al. [KAH11] proposed 2D-HERT, a two-dimensional hierarchical expansion of a ring topology focusing on optical NoCs. Kilo-NoC [GHKM11] is a topology-aware QoS-oriented architecture, adopting a low-diameter topology. It provides a service guarantee for applications with reduced power and area costs. It reduces the extent of hardware support to portions of the die, which in turn reduces router complexity to support large core counts. In [BZ07] authors present a hybrid architecture where a large 2D-mesh is partitioned into several smaller sub-meshes. Next, the sub-meshes are connected using a hierarchical ring interconnect for delivering global traffic. In this work, a bridge module is used for driving traffic to the different levels of the hierarchy. The addressing and routing scheme has also been modified to support the proposed topology.

2.3.4.2 Other approaches

In [DEM+09], a two-tier hierarchical topology consisting of local networks managed through a bus, and a global network controlled by a low-radix mesh router has been proposed. Authors showed that proposed topology could reduce the latency, power consumption and energy-delay product only for localised communication-based applications. Apart from hybrid ring-mesh or bus-mesh topologies, concentrated mesh (CMesh) [BD06] is a modified mesh architecture with replicated sub-networks where express channels are used to incorporate the second network without increasing the die area and wire length. This approach aims at reducing the hop count and load imbalance. Channel lengths are kept short to reduce energy dissipation, while express channels are used to improve energy efficiency.

2.3.5 Application Thread-to-Core Mapping policies

In general, the problem of mapping application threads onto a NoC is a graph embedding problem [AR82] and also the mapping problem is an instance of NP-hard problems [GJ79]. Further mapping a graph onto another graph is an example of quadratic assignment problem (QAP) [GJ79]. Several different approaches are mentioned in the literature to map the application threads to free processing cores, and they can roughly be classified into exact and heuristic approaches.

2.3.5.1 Exact approaches

In [SBSK12], multi-commodity flow (MCF) based integer linear programming (ILP) has been proposed to derive optimal static schedule tables for calculating upper bounds of the worst-case execution time. The model was employed on a time-division-multiplexed (TDM) NoC meant for hard real-time systems, and the model also considers different topologies. In [Tos11] author proposes a cluster-based ILP formulation for application mapping problem for 2D-mesh NoC. Both the application and the mesh are represented as graphs and further partitioned into smaller sub-graphs. The proposed ILP is used to map each sub-graph onto the corresponding sub-mesh. In [OB04] the authors, suggest an MCF based ILP formulation provide optimal routing and wavelength assignment for optical networks. The cost function is based on a piecewise linear, monotonically increasing, link cost function with a penalty term for the constraints violations. Among the exact approaches, authors in [HM05] proposed a branch-and-bound based algorithm for both application mapping and path allocation problem for 2D mesh NoC. It maps the cores to tiles and generates a suitable deadlock-free routing function to optimise the total communication energy cost by bandwidth reservation.

2.3.5.2 Heuristic approaches

Among the heuristic approaches for the thread-to-core mapping problem (T2CMP), Sorensen et al. [SSPH14] propose a metaheuristic scheduler for inter-processor communication in multicore platforms using TDM NoCs. The scheduling problem has been modelled as a fixed-flow, minimum-time integer MCF problem. In [HZC$^+$06], authors, propose a polynomial time approximation algorithm for MCF based formulation to minimise the power consumption of a NoC. It's constraints are to satisfy the global communication latency while optimising network topologies and wire styles. [SC05] also presents a polynomial time heuristic for application mapping on mesh-based NoC to minimise the communication energy. In this proposed solution, bandwidth, as well as latency constraints, are also satisfied. Authors in [MBDM05] proposed a unified design approach for building application specific NoCs to automate application mapping operations onto cores. It uses a tabu search algorithm for mapping and MILP for physical planning. The author claims that the model guarantees QoS by satisfying constraints (such as the delay/jitter, real-time constraints) of the traffic streams. Multi-objective genetic algorithm based heuristics are also used for application-core mapping mainly to optimise performance and power consumption [ACP04, HAQT$^+$04, LK03].

In [EF15], a fractional MCF based algorithm has been proposed to design NoCs with guaranteed QoS. It determines the widths of the interconnections as well as the routes of the flits by giving topology, mapping of tasks, and traffic pattern. [MDM04a] pro-

pose a mapping algorithm for 2D mesh-based NoC architecture to minimise the average communication delay by satisfying the bandwidth constraints. The proposed algorithm is based on fractional MCF. It is developed for both single minimum-path routing and split traffic routing. There are also research works (such as SUNMAP [MDM04b] a tool for automating the topology selection and generation process, a link speed assignment algorithm [SK04] with voltage scalable links to minimise the energy cost, or an application-to-core mapping algorithm [DAM+13] to place bandwidth-intensive applications closer to the MCs) that are worth to mention.

2.4 Summary

In this chapter, the background of the relevant domains has been presented. It mainly follows the top-down approach. The discussion first started with the programming environment mainly focused on dataflow programming model, then move to the dataflow-inspired program execution model and some of the proposed runtimes from academia is also presented. In general, there is a vast set of programming models that are available to use, and the mentioned background does not intend to be an exhaustive background. However, the basic information about dataflow execution model, then its languages and some proposed dataflow based PXMs are discussed.

Next, the discussion focuses on the hardware level that is relevant to the current time and also with the used programming models. Here, the provided background of the targeting hardware or the dataflow based hardware is not only proposed from academia, but also from the industry. The proposed hardware was supposed to offer dataflow model, but unfortunately, they do not become a mainstream model because of the lack of an integrated approach towards dataflow computation based ecosystem (which consider the different levels of computation from hardware to programming model to programming languages with proper compiler support and active dataflow community). Finally, an overview on the interconnection subsystem is given. In the next chapter, a hashing based dataflow thread distribution mechanism is presented.

3

Thread Distribution

This chapter lay the first brick of this doctoral thesis work and is mainly divided into two segments. In the first part, the chapter discusses the scalability features of dataflow threads (using DF-Threads). Next, it has been shown how a hash-based distribution scheme can be very efficient with simple hardware modification. As current chips are becoming more powerful and getting more complex, so it is also needed to provide an infrastructure to harness their capability. The proposed distribution scheme offers abstraction, structure and flexibility and targets dataflow execution model. The data-driven runtime or the languages can be benefited using the proposed thread distribution policy. The chapter is organised as follows: Section 3.1 gives the generic overview of the problem in the context of multi-/many-cores. In Section 3.2 a brief overview of DF-Threads (with its APIs) is given and also results are presented to show its scalability. Section 3.3 provides the description of the implemented (another) dataflow PXM to be used with the proposed hardware extension for distributing and managing the threads. Section 3.4 provide the complete overview of the proposed architecture, while Section 3.5 explains the hashing mechanism. Section 3.6 mentions some of the primary results to show the efficacy of the proposed hardware extension using some synthetic applications. Finally, section 3.7 summarises the contribution of the chapter.

3.1 Introduction

Exascale machines are expected to execute large, multiple applications at higher speed. To meet this goal, these machines have to manage a number of threads that is orders of magnitude greater than in current petascale machines, with more stringent power and resiliency constraints [D+11]. Recent CPU designs favour the integration of a vast number of simple, single-issue, in-order cores [LKGF+12] to increase the number of threads that can be executed in parallel. However, traditional program execution mod-

els (PXMs) derived from the von Neumann model and used by such systems, exhibit a large thread synchronisation overheads. Their inherent sequential nature makes it tough to guarantee correctness and race condition freedom in multithreaded program executions [Lee06]. Furthermore, when fine-grain threads are exploited, their synchronisation activity quickly becomes the main performance limiting factor [GZM+17], also contributing to energy waste. Instead, PXMs which rely on explicit producer-consumer semantics and are self-scheduled [Den74] can drive the design of efficient and less power hungry chip architectures [YAMJGE14].

The eXplicit Multi-Threading (XMT) architecture [VDBN98] introduces an abstract execution model, where switching from serial to parallel execution is made through explicit spawn/join instructions. Specifically, such instructions create a group of concurrent threads executing the same code block, while microarchitectural support remains generic. DDM proposed a scalable architecture which, however, requires a large amount of storage to maintain a local copy of the threads' dependency graph, while a flat thread distribution model is applied. Recently, TERA-FLUX [GBB+14] proposed a chip architecture for the explicit exploitation of a dataflow PXM, where cores are organised into fixed-size nodes. Although it was proposed as a scalable solution, many drawbacks remain: locality of computations is not guaranteed (threads cannot explicitly restrict execution within a node), and an efficient selection of the target execution cores is not described. In recent years, GPUs are emerged as the preferable platform to accelerate computations [SIL+15], thanks to their capability of running hundreds of fine-grain threads in parallel. However, their architecture is optimised for regular applications and does not adapt well to irregular data and control problems.

In the first part of the chapter, the scalability features of a dataflow PXM (such as DF-Threads) are shown. Later, in the chapter, a Data-Enabled muLti-Threaded Architecture (DELTA) is proposed – which attempts to: *i*) implement an effective mechanism to select target execution cores, as well as to guarantee locality of computations. *ii*) supporting the execution of a large number of concurrent threads with a lightweight synchronisation mechanism, and *iii*) provide a simple programming interface. Starting from a manycore tiled chip, we augment the NoC router structure with a hardware unit responsible for the threads' creation and distribution over the application lifetime (we are agnostic with respect to the processing element architecture). Also, a fast hash-based mechanism allows the system to efficiently distribute the threads among the available processing resources, leading to more dynamic scaling-up capabilities and less power consumption.

3.2 DF-Threads and its scalability

Synchronisation and distribution of data can be managed efficiently by reorganising the execution in such a way that the threads follow more closely the data flow of the program (such as with DF-Threads). DF-Threads can be efficiently implemented by a distributed hardware thread scheduler [GS15] which support fault tolerance at the hardware level and efficient fine grain dataflow thread distribution. To reduce the thread management overhead, the scheduling needs to be accelerated in hardware, by mapping its structure into the FPGA. A DF-Thread is defined as a function that expects no parameters and returns no parameters. The body of this function can refer to data which reside at the memory location for which it has got the pointer. The DF-Threads API's [Gio12] are summarised below:

- `void *DF_TSCHEDULE(bool cnd, void *ip, uint64_t sc)`: Allocates the resources (a DF-frame of size `sc` words and a corresponding entry in the distributed thread scheduler or DTS) for a new DF-Thread and it returns a frame pointer `fp`. The `ip` is the instruction pointer of DF-Thread. The allocated DF-Thread is not executed until its `sc` reaches `0` and together also satisfy the boolean condition `cnd`.
- `void DF_DESTROY()`: To release allocated resources held by current DF-Thread.
- `uint64_t DF_TREAD(uint64_t offset)`: Loads the data indexed by `offset` from the current thread of DF-frame.
- `void DF_TWRITE(uint64_t val, void *fp,uint64_t off)`: The data `val` is stored into the DF-frame pointed to by `fp` at the specified offset `off`.
- `void *DF_TALLOC(uint64_t size, uint_8 type)`: Allocates a block of memory of `size` words and returns the pointer (or null) while `type` specifies the special purpose memory type.
- `void DF_TFREE(void *p)`: Frees memory pointed to by `p`.

The scalability of the DF-Threads has been tested using the HP-Labs COTSon simulator [AFF+09] which uses "functional-directed" approach. The simulator can perform a full-system simulation. We have reported the experimental results (in Figure 3.1, 3.2, and 3.3) consisting 1, 2 or 4 nodes while the employed execution model is based on DF-Threads. We use well-known blocked matrix multiplication as application benchmark. The scalability of DF-Threads is tested using three matrix sizes: `n=256, 512, 1024` while the block size is fixed to four `b=4`. The parallelization is based on the ratio between the matrix size `n` and the block size `b` (i.e., the expected number of DF-Threads is

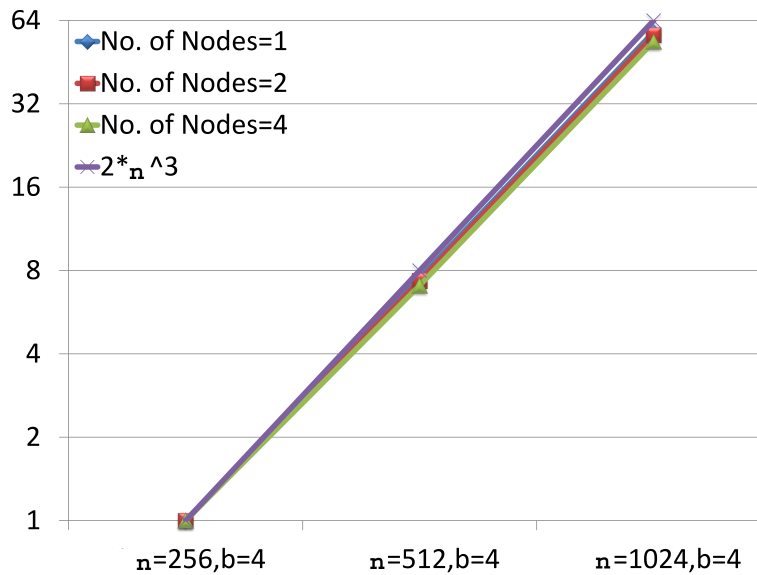


Figure 3.1: Instruction count normalised to the matrix size 256.

n/b). In the experiments, the number of DF-Threads are 64, 128, 256 respectively. The interesting result is related to the total number of instructions. We can see from Figure 3.1, for each matrix size the instruction count has almost the same value once we vary the node sizes from 1 to 4 (three superposing lines). The reason for that is due to the small overhead to manage DF-Threads across nodes. Moreover, the number of instructions follow the theoretical increase (i.e., the number of instructions increases as $O(n^3)$) in the case of a classical block-matrix multiplication closely. We normalised the total number of instructions for each curve to the case of matrix size n=256 to compare the three experimental cases and the theoretical $O(n^3)$ line in Figure 3.1.

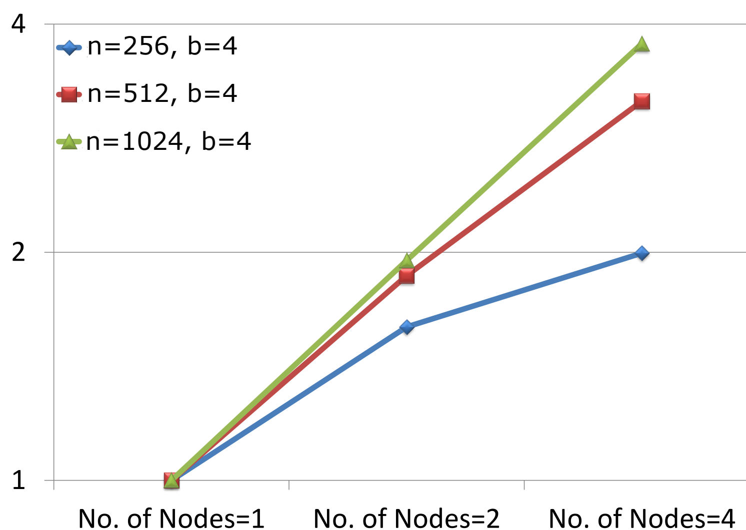


Figure 3.2: Speedup of user cycles count normalised to the matrix size 256.

Next, the scalability improves significantly when there are a larger number of threads (see Figure 3.2). The speedup is almost ideal (for four nodes the speedup is almost `4`) in the case of `n=1024, b=4`. The effect of different block sizes are not reported, but for smaller block sizes typically it achieves better scalability [Gio15]. It is worth to note that there is a possibility to scale performance across nodes which have separate address spaces.

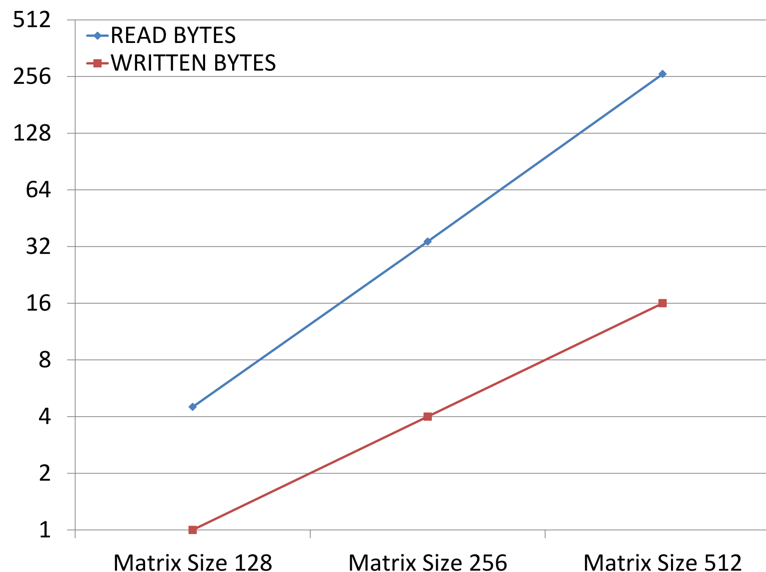


Figure 3.3: Scaling of read and write operations for DF-Threads.

Finally, in Figure 3.3, the total amount of read and write data set size (bytes) are also reported. However, unlike the previous cases, the matrix size is `n=128, 256, 512` while the block size is fixed to eight `(b=8)`. In the figure, 128x128 is used as the baseline with increasing node counts. The read and write data set size follows a linear trend and is also not creating a saturation effect with increasing core count.

3.3 Program Execution Model (PXM)

PXMs define how a computation must be carried on a target machine, on concurrency (how threads are created, scheduled, and destroyed), memory behaviour (how memory is addressed, and what ordering rules it obeys), and synchronisation (how threads can synchronise/wait for each other). Contrary to programming models, which describe *what* high-level action should be done (and when), PXMs describe *how* such an action is carried in the system. For example, OpenMP's programming model allows a programmer to define a region of code as *parallel* but makes no explicit mention of threads. However, OpenMP's PXM specifies that when a parallel region is encountered, a team of threads has to be created and must also be destroyed when the end of the region is reached.

Here, we use an execution model directly derived from the Codelet Model [SZG13], where assisting hardware provides large performance improvements over a pure software implementation. The applications are divided into a set of fine-grain threads, each totalling no more than a few tens or hundreds of instructions. Fine-grain threads are exploited in a way that allows maximising the utilisation of the system.

Threads represent the quantum of execution: they exchange data with each other by resorting to an explicit producer-consumer scheme. It allows the construction of a dataflow graph (DFG) at compile time, which explicitly shows data dependencies among threads. Each thread holds a local storage space (*frame*) used to receive input data from producer threads, as well as to write intermediate results. It also contains a *scheduling slot* (SS) counting the number of inputs still required for the execution. A thread context contains the frame data and a unique thread identifier. To preserve locality and allow for better latency hiding, threads are grouped into *asynchronous functions* (AFs). Similarly, DELTA provides a mechanism for dynamically grouping processing elements (PEs) to form virtual nodes (VNs) as part of the hashing mechanism, so that threads within an AF are forced to be executed on the same VN. With the aim of exposing these characteristics at the programming level, the proposed architecture extends the PE ISA with a reduced set of dedicated instructions (eventually wrapped by high-level programming language functions, *e.g.*, C/C++) as follows:

- `CreateThread(*code,SS,frame)`: creates a new thread context, *i.e.*, the SS, the type of the thread, a unique identifier, and the required space to hold the thread's data frame and allows to schedule the execution of threads belonging to the same TP within the same VN;
- `CreateAF(*code,SS,frame,TT)`: as the above instruction, but it allows a new asynchronous function (*i.e.*, a new thread spawned outside the VN);
- `ReadData(offset)`: reads data from a thread's frame, at a specific offset within the frame;
- `WriteData(TID,frame,offset,data)`: writes data to a thread's frame, at a specific offset within the frame (both within and outside the current VN);
- `DecreaseSS(TID,dep_cnt)`: allows to decrease the SS of a thread by the number of resolved dependencies;
- `DeleteThread()`: removes the context of a thread that has completed the execution;
- `SetVN(N_pe)`: sets the number of tiles of each VN and sends a broadcast message to all the tiles indicating the number of PEs composing each VN.

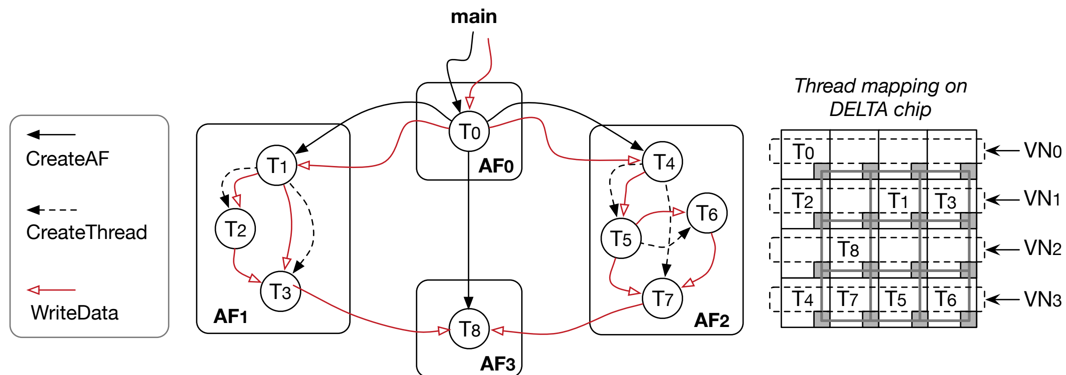


Figure 3.4: A simple kernel application adhering with the proposed PXM and a possible mapping of threads on the PEs.

- `ConfigRouter(*config,Rd,B)`: allows to configure routers, by specifying the memory address where the configuration is stored. Destination router identifier is contained in the Rd variable, while flag B indicates if the configuration is broadcast to all routers.

With the aim of further optimising the execution, the compiler can aggregate multiple writes (e.g., dealing with large loops) and use a single `DecreaseSS` signalling operation to update the corresponding SS field.

Every time a new thread is spawned, a PE within the current VN is automatically selected and signalled. Similarly, every time a thread creates a new AF, the destination PE is selected within the whole chip. Hence, the creation of a new AF is led back to the scheduling of the root thread of the DFG contained in the AF. Figure 3.4 shows an example of a simple kernel application consisting of 4 asynchronous functions, each with its DFG. Both asynchronous functions and threads are directly managed by the compiler, which is responsible for mapping high-level programming constructs (*e.g.*, `#pragma omp for` when using OpenMP) with the correct sequence of `CreateAF` and `CreateThread` instructions. Figure 3.4 also shows that AF scheduling requests and write operations remain well confined on the local VN. By monitoring the scheduling slots, the hardware unit automatically fires threads became runnable, without the need of executing an explicit instruction. On the contrary, the execution completion is signalled by the `DeleteThread` instruction that allows freeing resources held by the thread.

3.4 Proposed Architecture

Figure 6.1 shows the proposed chip organisation: a dedicated Network-on-Chip connects a large group of tiles covering the entire chip area. Each tile contains a PE coupled with

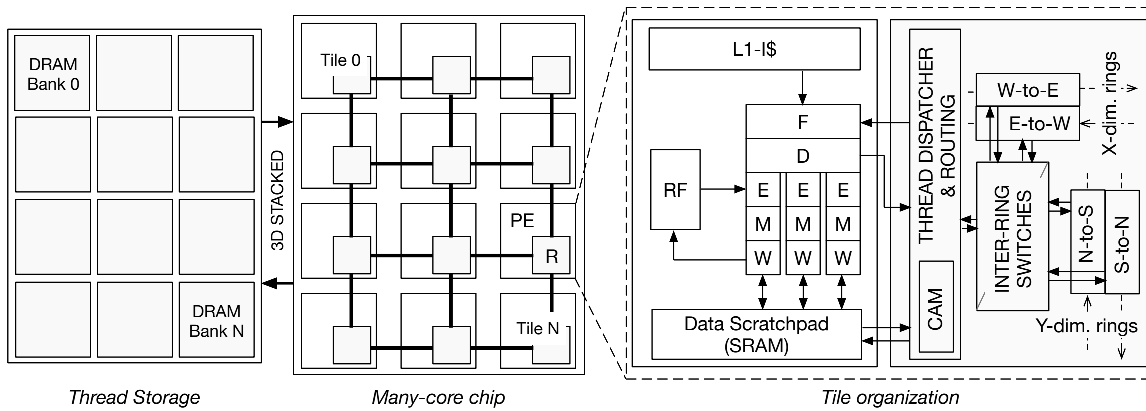


Figure 3.5: Chip organization: tiles contain a PE (white box) and router (gray box). The scratchpad substitutes the traditional L1-data cache.

a lightweight router supporting a 2D-mesh topology. Routers are augmented with our fine-grain thread hardware support: a local unit called *Thread Dispatcher* (TD) manages the threads during their lifetime. In particular, it allocates internal space for storing thread contexts every time new threads are created, removes previously allocated resources whenever threads complete, reads from (respectively writes to) associated frames. Since the software controls the behaviour of threads through the extended instruction set, the TD directly interact with the decoding stage of the PE. We assume that only one thread at a time can be executed in each PE, although our approach can benefit from the implementation of a form of local simultaneous multithreading.

Chang et al. already showed how to apply a dynamic hashing mechanism to distribute the workload in a peer-to-peer system [CCLL08]. Here, we propose to apply hashing for thread allocation in a hardware system, in a much more constrained environment. On the one hand, the hashing mechanism must avoid the creation of hot-spots in the chip (*i.e.*, areas of the chip particularly stressed). In fact, an imbalance in the load distribution quickly and significantly increase the power and temperature of the more stressed portion of the chip and thus contributing to decrease the overall reliability (*e.g.*, device ageing is accelerated). Load imbalance can also create congestion in the network since some of the links drive more traffic than others. On the other hand, the hashing mechanism used to distribute the threads must guarantee locality of computations. To this end, our hashing scheme allows placing a group of dependent threads (asynchronous function) on the same group of PEs (VN), while still preserving a fair thread distribution within the VN by selecting PEs in a random fashion. Similarly, the hashing scheme allows to randomly schedule asynchronous functions on different VNs across the whole chip. It is worth noting that choosing PEs that minimise the communication distance between producer and consumer threads in a reasonable amount of clock cycles is not a trivial problem. In fact, our PXM implies more than one producer

can generate input data for a single consumer, as well as producers can be scheduled and executed at different points in time.

Each thread in the system is identified by a unique *thread identifier* (T_{id}) over the whole application execution. It is composed of three main fields: the *source field*, the *destination field*, and a local counter (*CNT*). The source and destination fields are in turn formed by two sub-fields representing the *core identifier* of the PE (C_{id}), and the *VN identifier* (N_{id}). While the content of the source field is fixed for each tile (once the number of cores in each VN has been selected), the content of the destination field is produced at run-time by the hashing function $H(\cdot)$. These fields allow the system to generate unique identifiers that are conveniently stored in a 64-bits register. The organization of the T_{id} is illustrated in figure 3.6. By passing to the $H(\cdot)$ module both the size of VNs (N_{pe}) and the indication of the executed instruction – I_{ex} (*i.e.*, the `CreateThread` or the `CreateAF` instruction), it uniquely identifies the PE responsible for the execution of the newly generated thread (*i.e.*, $\langle N_{id}, C_{id} \rangle_{dst} = H(N_{pe}, I_{ex})$). Once selected, the PE is signalled by sending a message over the network. Since the destination is encoded in the T_{id}, any subsequent operation on the thread can easily be forwarded to its corresponding PE, without any further calculation. This contributes to the speedup of the system.

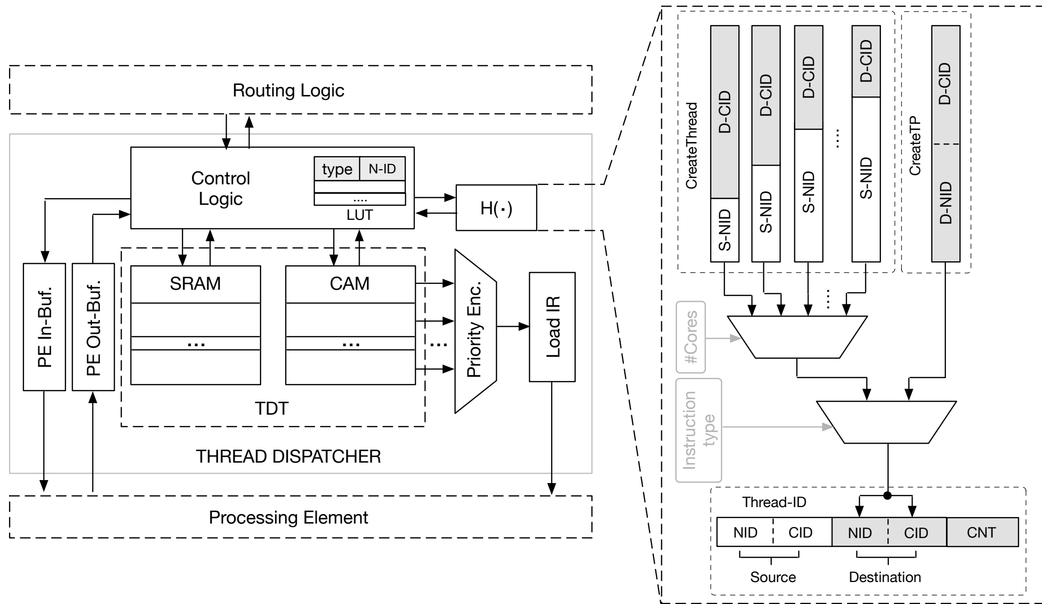


Figure 3.6: Thread Dispatcher module organization (left) with the internal structure of the $H(\cdot)$ function (right).

3.4.1 Thread descriptor table

Threads are managed through a data structure called *Thread Descriptor Table* (TDT) that is organised in two fixed-size local memory arrays (their total area is comparable

with that of an L1 data cache). Input and intermediate data are stored in the scratchpad memory. Every time the context of a thread is updated, the T_{id} is used as a search key within the CAM. In case of a match, the returned base address of the frame F_b is added to an offset F_o to determine the location access (*i.e.*, $l = F_b + F_o$). Finally, a priority encoder selects the thread with the lowest T_{id} among those runnable ($SS = 0$ – see figure 3.6). Every time the selected PE is devoid of free resources, it can access to a larger but slower memory area called *Thread Storage*, implemented as a 3D stacked DRAM layer. It is organised in banks (one for each PE) representing a larger TDT structure. When a PE receives a new thread, it first selects the entry in the local TDT and compares the SS value of the new thread and the one currently stored. The thread with the highest SS will be swapped on the DRAM memory bank.

3.4.2 Network-on-Chip Architecture

The basis of our system is a 2D-mesh NoC implemented with lightweight ring-based routers [SMP16]. Such kind of routers allows implementing a flexible 2D-mesh topology on top of four unidirectional rings. The internal crossbar switch is substituted with four ring stations, each capable of driving the network traffic in the same direction or steering it to the opposite dimension. The ring stations are coupled with two additional modules (*inter-ring switch* – IRS) that are responsible for ejecting traffic travelling in one dimension or to inject traffic in the opposite dimension. An internal table describes how traffic flows in the links.

3.5 Hash Scheduling Function

The purpose of the hash scheduling function $H(\cdot)$ is to map new threads to PEs for their efficient execution. In our case, the hardware module assigns to the newly created thread the tuple $\langle N_{id}, C_{id} \rangle_{dst}$ depending on the VN size and the executed instruction. To be effective, the scheduling function $H(\cdot)$ has to distribute `CreateAF` and `CreateThread` requests among the available resources fairly. The effectiveness of the hashing function derives from the ability to limit the number of times two distinct input values result in the same output value for the hashing. In our distributed scheduling scheme, this translates in avoiding different PEs selecting the same destination, given two different T_{id}. In that case, the PEs' load (*i.e.*, the number of threads to execute) is balanced, thus avoiding the formation of hot-spots and increasing the overall system reliability. To this end, we found that the following scheme provides very good results while maintaining a low area overhead and preserving the capability of dynamically changing the size of VNs. Another important aspect of our scheme is that it works in

a completely distributed fashion, meaning that single-point-of-failure is not present, as desired in a system equipped with thousands of PEs possibly. The $H(\cdot)$ module contains a set of maximum-length linear-feedback shift registers (LFSRs), each providing a pseudo-random sequence with a different length L_{rnd}. Let n be the number of bits composing the LFSR, the length of the sequence is given by $L_{rnd} = 2^n - 1$, *i.e.*, the register cycles through all the 2^n configurations except for the configuration containing all 0s. The structure of the LFSR is thus modified, in such way all the 2^n configurations can be generated. The use of LFSRs allows the $H(\cdot)$ module to select every time a different PE in a round-robin fashion (although it is a random sequence). Compared to simple counters, LFSRs guarantee a homogeneous spatial distribution of the threads among the PEs in a VN, over the time (since the LFSRs are initialised with a different seed the generated sequences are different over the time). It again preserves the chip from the emerge of local hot-spots and contributes to reducing pressure on the NoC links, reducing link contention and thus improving performance. Depending on the executed instruction (`CreateThread` or `CreateAF`), LFSRs are used to generate the destination C_{id} or both the C_{id} and N_{id}. In the case of the `CreateThread`, the destination VN is the same of the PE spawning the new thread. The $H(\cdot)$ module computes the destination PE for different VN sizes in parallel; while the selection of the actual one depends on the effective VN size, and it is performed by a multiplexer. For instance, a VN containing 64 PEs requires an LFSR 6-bits long. The executed instruction also controls which tuple (*i.e.*, the one formed by both the newly generated C_{id} and N_{id}, or the one formed by only newly generated C_{id}) to copy in the destination field of the T_{id} through a second multiplexer. This mechanism is shown in figure 3.6, right side. Note our scheme guarantees that locality of computations: threads belonging to the same asynchronous function are kept close to each other since they are executed on PEs placed closely in the chip albeit randomly selected (see also figure 3.4). At the same time, asynchronous functions are homogeneously distributed among the VNs. Finally, to deal with space limitation for thread management, each tile can eventually reschedule a thread on a different PE for a limited number of time.

3.6 Evaluation

We evaluated the proposed architecture regarding scalability and power consumption, as well as we evaluated the capability of the hashing function of well-distributing input requests. The simulated manycore design comprises up to 256 PEs implementing a 5-stage RISC-V compliant in-order execution pipeline (16 KiB I-cache + 16 KiB scratchpad memory), supporting our proposed instruction set extension, and integrating a 2-stage lightweight router [SMP16]. We generated network traces using an in-house simulator

and monitoring the set of requests send to the TD units. We performed the scalability and power consumption measurements implementing the NoC infrastructure on an Altera Stratix III-based device.

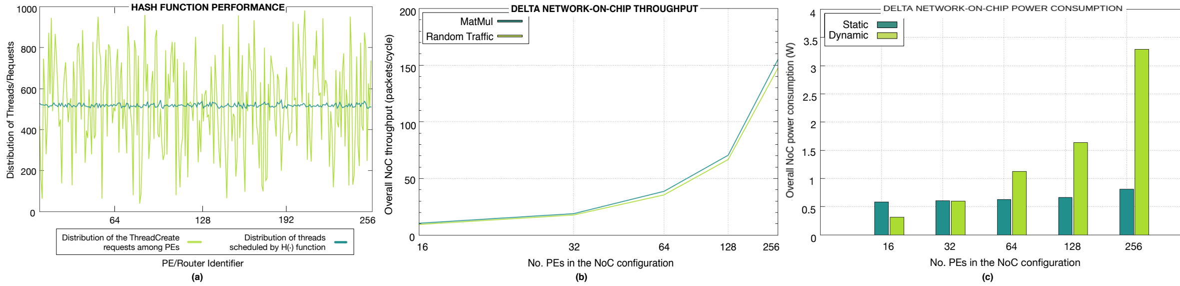


Figure 3.7: NoC performance: distribution of threads on the PEs (a), average throughput (b), and power consumption (c).

Figure 3.7 (a) shows the performance provided by the proposed hashing function implementation. The purpose of this experiment was to show how a huge number of input keys for the $H(\cdot)$ module (*e.g.*, thread scheduling requests, write operations) were distributed among the PEs. To that end, we simulated a random traffic pattern in the NoC by allowing each tile to randomly injecting a schedule request (injection rate was 1.0) towards a randomly selected VN and PE. This kind of pattern is more effective in showing the capability of the hashing mechanism since the traffic cannot be predicted. The green line represents the initial distribution among the PEs of the `CreateThread` requests, while the blue line shows the effective distribution of threads as they have been scheduled by the $H(\cdot)$ modules. The high fairness in the assignments of the threads to different PEs greatly contributes to the high overall performance of the network. Similar results have been obtained simulating the traffic pattern generated by a block matrix multiplication kernel. The same traffic patterns have also been used to assess the NoC throughput and power consumption. Such traffic patterns are effective in showing the capability of our hashing scheme to balance threads' requests, thus avoiding particular links to be overloaded. Figure 3.7 (b) shows the average throughput obtained for different configurations. Irrespective of the growing of the number of PEs in the system, the throughput grows almost linearly. Finally, figure 3.7 (c) demonstrates the other advantage of the proposed thread distribution approach: power consumption is relatively low if compared with an implementation based on traditional crossbar switch routers. In general, the area and power consumption for the scheduling logic are very low, while that of the TDT is in line with that of an L1-data cache (it is worth noting that the scratchpad memory substitutes the L1 data cache, and represents the main data-exchange point between routers and PEs).

3.7 Summary

Improper management of huge amount of concurrent threads can create resource contention, leading to overall degraded system performance. Execution models and manycore chips are expected to overcome limitations of current systems, by leveraging more effective approaches to distributing threads on the available resources. In this chapter, first, the scalable feature of a data-driven PXM is shown and later a distributed thread management mechanism is proposed. A manycore tiled architecture where NoC routers are extended to support the proposed execution model has been considered. The efficacy of hash function for distributing threads with low overheads has also been shown here. By extending the structure of NoC routers in such way, a fair thread distribution policy can be achieved. The proposed extension is analysed, as well as a set of specialised instructions, are also proposed for supporting thread management. Simulation results confirm that the proposed design can manage a large number of simultaneous threads, relying on a simple hardware structure and also its capability to effectively scale with an increased number of PEs, while future investigations are needed to port complex applications to the proposed PXM and also extended some support if needed.

3.8 Acknowledgement

First segment of this chapter (Section 3.2) contains the selected results from two published papers entitled **AXIOM: A Hardware-Software Platform for Cyber Physical Systems** by Somnath Mazumdar, and others, at Euromicro Conference on Digital System Design 2016, and **Modeling Multi-Board Communication in the AXIOM Cyber-Physical System** by Roberto Giorgi, Somnath Mazumdar and others at Ada User Journal, vol. 37, no.4, 2016.

The second part of the chapter (Section 3.3-Section 3.6) is an edited version of the accepted paper entitled **Enabling Effective Hash for Massive Multi-Threading** by Alberto Scionti, Somnath Mazumdar and Stéphane Zuckerman at IEEE Computer Architecture Letters (CAL).

4

Monitoring

In the previous chapter, how to distribute dataflow threads has been discussed, and now the main aim of this chapter is to provide an overview for monitoring the dataflow threads at runtime. In recent years, dataflow based execution models have started to gain popularity. The popularity mainly is driven due to its better support for concurrent, multithreaded applications. Some dedicated hardware (see Chapter-2) also been proposed to harness the capability of dataflow-based execution models. However, understanding the behaviour of a huge number of concurrent threads, and also the best achievable performance on a given hardware platform, remain a big issue. Moreover, the evaluation process of running applications often requires the usage and modification of complex simulation frameworks. To counter these challenges, a runtime analysis tool (RADA–Runtime Analysis of Dataflow-based Applications) for dataflow-based applications has been proposed. It is a simple simulation tool for fast evaluation of applications adhering to a hierarchical dataflow execution model. It provides an abstract model for different hardware platforms by hiding the unnecessary hardware details, and also allow to optimise the application for the execution on heterogeneous systems. In particular, it integrates an efficient scheduling mechanism, in the form of a runtime library, which allows the system to distribute the workload efficiently while minimising the synchronisation overhead. The primary output provided by the tool are of great help in analysing the traffic generated by the scheduling activity. Preliminary evaluation results show the significant benefit in adopting this kind of tool for the assessing the dataflow based applications and its execution models.

The chapter is organised as follows: Section 4.1 introduces the issue and in Section 4.2, the abstract machine model has been introduced and also the dataflow threading model that is used by the proposed tool. Section 4.3 describes the actual implementation of the simulation environment, while Section 4.4 describes extensions enabling the evaluation of application's execution on a heterogeneous system. Finally, Section 4.5

presents experimental results, while Section 4.6 summarises the chapter with the findings and also the future works.

4.1 Introduction

Performance scaling remains a concern despite multiple achievements in manufacturing technology and architectural designs. Harnessing built-in parallelism from current multi-/many-core processors requires the adoption of effective program execution models (PXMs). In general, effective PXMs limit the communication and synchronisation overheads experienced by concurrent threads. Until now, the most well-known execution models for executing massively parallel applications are mainly based on the von Neumann architecture. Due to its limitations [I^{+}88], in recent times, dataflow model of computation has gained popularity. However, the roots of the dataflow execution model dated back to early of 1970's [RRB69]. Applications adhering to dataflow PXMs follow a simple producer-consumer communication scheme for synchronising parallel thread related activities. In dataflow execution environment, a thread can run if and only if all its required inputs are available. By allowing threads to schedule the execution of other consumer threads, the synchronisation mechanism is set to the required number of input data to the appropriate consumer thread. Applications running on a complex and large computing domain (such as high-performance computing (HPC), high throughput computing (HTC) and Cloud computing) can significantly benefit from the adoption of dataflow PXMs. In this chapter, a tool called RADA (RADA–Runtime Analysis of Dataflow-based Applications) for runtime analysis of dataflow-based applications has been proposed. The contributions of the proposed tool are three-fold: *i)* it enables the execution of a multithreaded dataflow application on a highly parallel system, exploiting the advantage of modern multi-/many-core processors. *ii)* it allows the system designer to assess the performance of the given application. *iii)* it allows evaluating the effect of the scheduling policy on the performance and the traffic generated by the interconnection subsystem. This tool allows integrating the scheduling features directly inside the applications, while an abstract execution model allows running the application effectively on different hardware platforms. It is worth to note that the latter feature becomes essential to evaluate the impact of scheduling algorithms on heterogeneous systems. Preliminary results are presented to show its potential, and the advantages carried out by the dataflow execution.

4.2 System model

Abstract machine models (AMMs) provide the abstraction for the class of multi-/many-core processors and software systems that implement the actual execution substrate.

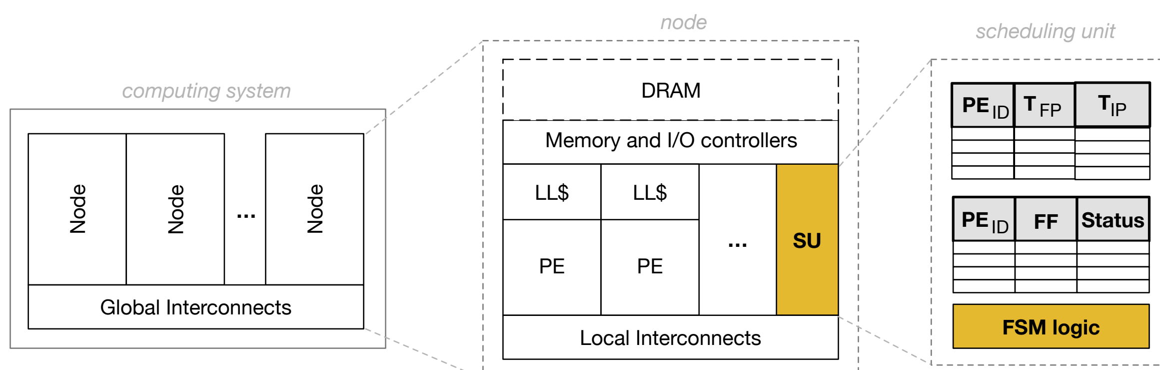


Figure 4.1: System overview: the abstract machine model used to managing execution of dataflow threads.

The tool has been designed taking into consideration an AMM providing the set of features needed to execute fine-grain threads in a data-driven fashion (supporting the dataflow execution environment). Figure 4.1 shows the architecture of the proposed AMM. Current hardware can also provide performance enhancement through heterogeneity [MV15]. When a particular hardware platform is not available, the only way of assessing the performance of an application is to use a simulation environment. Execution-driven simulators emulate the behaviour of the real platform to execute the application unmodified. Conversely, in the trace-driven approach, the application is extended with additional information. Such information allows the application to extract values from a set of monitored parameters (e.g., the number of jumps, executed floating point operations, cache misses) [UM97]. From this viewpoint, RADA offers an efficient *trace-driven approach* for assessing the performance of a dataflow application following a hierarchical PXM and executing on modern multi-/many-cores. The system consists of a set of computing nodes which are connected each other through a global interconnects. Such interconnection subsystem provides the set of resources, as well as it exposes the set of features required to allow dataflow threads to communicate each other. Nodes resemble general purpose multi-/many-cores. Each node contains several processing elements (PEs) equipped with their cache hierarchy. Figure 4.1 highlights the last level cache module (LL$) which interfaced to the rest of the main memory through a memory controller module (external DRAM memory modules can be used to build a shared memory environment). The latter also embeds the logic to manage I/O operations (memory and I/O controllers). A dedicated local interconnection provides resources to manage the communication among the PEs, as well as to interface with the global interconnects. The scheduling unit (SU) is a dedicated core, which is responsible for managing the lifecycle of dataflow threads within the node (i.e., keep track of the resources allocated to the threads, update the threads state). It is also responsible for migrating threads ev-

ery time internal node resources are exhausted. Whenever a thread is created, a private memory region is assigned to it. This region referred to as the *frame block*, and it can be written by producer threads (but can be read only by the owner thread (consumer)). A table in the SU tracks the list of frame blocks assigned to each PE, and their status (i.e., assigned or free).

A second table is used to keep track of threads allocated to each PE (scheduled for the execution). The table contains the link to the assigned frame block, and the pointer to the code to execute as well. A PE locally maintains look-up data structures. In particular, each assigned thread can be in one of three possible states: (i) *waiting* – when the input dependencies have not completely satisfied, (ii) *ready* – all the inputs have been produced and the thread can be executed, (iii) *running* – one thread at a time can be in this state, meaning that PE executes the thread. *Synchronisation counter* (SC) is a counter that is set to the number of required inputs. PEs can track the input dependencies for each thread, by using the SC. Every time a new input is received, the corresponding SC is decremented by one. Threads become ready once the SC is reduced to zero. Each PE also maintains the list of pointers to pre-assigned frame blocks (used to speed up scheduling), the list of waiting threads (WT), and the list of ready ones (RT). WT list is implemented as a small look-up table containing SCs.

4.3 RADA's implementation

RADA is meant to be an effective tool for assessing the performance of dataflow applications. To achieve this goal, it has been implemented in such way so that it can run on commodity multicore machines while providing all the characteristics of the AMM (described in section 4.2). The tool has been implemented as a runtime library which provides scheduling and synchronisation features to application threads. Applications are written using conventional high-level imperative languages (C and C++), while the access to scheduling/synchronisation features is mediated by a set of dedicated functions. It is worth noting that the behaviour and semantic of such functions can be implemented as processor instructions allowing to access dedicated hardware modules. In the beginning, a set of working processes (*workers*) is created to emulate the structure of a node. Workers keep track of all assigned dataflow threads. The worker with the smallest process ID (PID) in the group is designated as the SU. Shared memory in the host platform provides the substrate for the communication among the PEs. Mechanisms for managing communications in a distributed memory environment (e.g., using MPI) can be applied to allow the inter-node communication. Fast look-up of internal data structures is ensured by the usage of red-black trees, with a computational complexity equals to $O(\log n)$. Figure 4.2 shows the implementation of the proposed system.

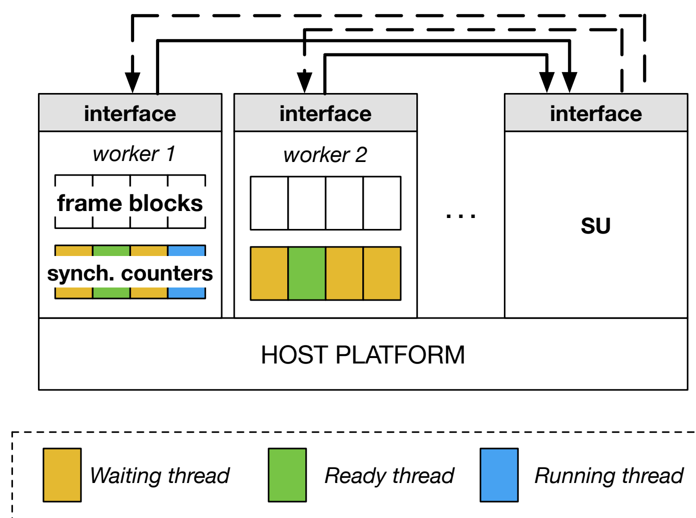


Figure 4.2: Implementation of the proposed system.

4.3.1 Software interface

To support the thread management, a minimalistic set of functions for scheduling, dynamically allocating/de-allocating private memory blocks and removing threads, as well as to read and write from/to frames has been provided. Such functions represent the RADA's software interface, specifically:

- `ThreadCreate()`: schedules a new dataflow thread by allocating resources to it, and generating the thread context (i.e., instruction pointer to the thread code, the pointer to the frame block, and the SC). The actual worker is selected by the SU, which receives the request from the scheduling worker.
- `ThreadRemove()`: releases all the resources held by the thread.
- `ThreadMemAlloc()`: allows the threads to dynamically allocate a private memory block of a given size (the size is specified in terms of number of words).
- `ThreadMemFree()`: frees a memory block previously allocated through the ThreadMemAlloc function.
- `ThreadMemRead()`: allows the dataflow thread to read a value stored in its frame memory block.
- `ThreadMemWrite()`: is invoked by a producer dataflow thread to write a value in the frame of a consumer thread. Every time a ThreadMemWrite is executed, the corresponding SC entry is decremented by one.

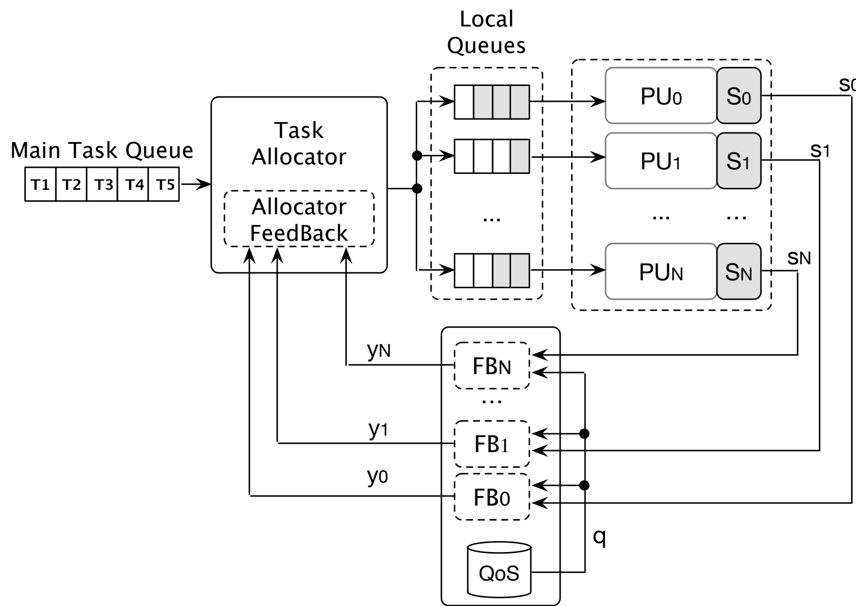


Figure 4.3: Negative feedback closed-loop based task scheduling system of the RADA.

4.4 Dealing with heterogeneity

Programmers can take advantage from running their applications in a heterogeneous environment since portions of the application code can be better executed on specific hardware. The tool supports heterogeneity by two-ways. First, nodes can be hosted on different machines equipped with processors exposing different features but using the same instruction set architecture (ISA). Second, by allocating a variable number of workers, it is possible to emulate the behaviour of multi-/many-cores, (i.e., systems equipped with a small number of cores, as well as an enormous number of cores). The last case is interesting because some distributed programming framework (such as MapReduce) support variable workers at different time of task's life cycle. Dataflow applications can use an extended version of the `ThreadCreate()` function to support such features, which allows scheduling the thread execution also on a different node. This support can be accomplished at the software interface level by adding a function call parameter. Threads that run on the same node form a *task*. Moreover, PXMs (such as the Codelet model) provide a hierarchical organisation of threads, that can be exploited to define tasks at the application level. Figure 4.3 shows the task scheduler implemented by the proposed tool. Task scheduling is more complex than thread scheduling since it deals with executing application code blocks exposing different characteristics regarding memory, processing and I/O resource usage. The tool allocates an individual node (actually, it is a dedicated process on the host machine) that is responsible for schedul-

ing tasks across nodes exposing a different set of resources. Such differences also reflect differences in the host microarchitectures. In fact, nodes can be mapped on different host platforms (e.g., a general-purpose X86 CPU and an Intel Xeon Phi accelerator).

In general, the feedback control is used when a system must be controlled for better stability, improved performance and robustness (or need to guide the runtime system behaviour). Any change in the system output should trigger a reaction of the controller for which feedback control is used. Feedback control is reactive or error-driven, while the feedforward is a proactive model. The feedback controller uses the measured errors, compute changes to the inputs, and sends those inputs to the system. In feedback control, the outputs of the system are measured, and the controller has feedback from the output of the system which quantifies "how far" is it from the desired state. RADA can take fast decisions by implementing a negative feedback control loop for the task scheduler, which exploits both information gathered, at runtime, from performance counters, and a linear power model used to build a ranking function.

4.4.1 Execution efficiency model

Migration of tasks from host core to co-processor is a popular approach to speed up the execution process, but during task migration, we also need to consider other parameters (such as input data transfer). In general, the decision of migrating a certain task on a given node is taken at runtime and based on the measured execution efficiency. At the basis of the proposed approach, there is the following observation: energy consumed per amount of work done is directly proportional to the effectiveness of the system. Thus, a simple yet powerful energy-based cost metric to evaluate the efficiency has also been proposed. As a single instruction also consumes energy, the proposed tool takes into account the overall energy used by the instruction to be executed (EPI – energy per instruction). An energy model has been developed and integrated, which uses several microarchitectural parameters (such as the pipeline depth, registry delay, and clock skew of the simulated system). Memory and I/O stalls can affect the total energy consumed by a single instruction. So, EPI has been assigned an average value obtained from several executions, each having the system in a different workload condition. For simplicity, a small group of instructions containing the one under evaluation is isolated and executed them directly on the host platform. The information that is gathered from such assessments is integrated into a cost function.

Equation 4.1 provides the overall offloading cost for a given task (C_T). An empirical tunable parameter called δ_T is introduced which combines the different (micro-)architectural parameters that may affect the execution of the instructions (e.g., effective bandwidth for transferring data, the number of pipeline stages, number of cache

levels). The another element of the cost function derives from the execution cost (C^{th}_{exe}) of each thread on the selected node (K is the total number of threads of the task). This cost is multiplied by τ, which counts the execution time for the given set of threads, to obtain a similar value for the overall cost function. In fact, C_T corresponds to the overall energy spent to execute all the threads.

$$C_T = \{\delta_T \cdot \sum_{th=1}^{K} C^{exe}_{th}\} \cdot \tau \quad (4.1)$$

The execution cost is computed according to equation 4.2, where N is the total number of instructions in the thread code, and EPI_{ist} measures the EPI. According to the equation 4.1, the summation of all the EPI values is multiplied by the clock frequency ϕ to obtain the power consumed by each instruction.

$$C^{exe} = \phi \cdot \sum_{ist=1}^{N} EPI_{ist} \quad (4.2)$$

It is interesting that energy consumption for a running application can be gathered at runtime by measuring different parameters through performance counters, which are commonly available in modern processing architectures. By measuring the cost function for each task separately, it is possible to build a small in-memory database (QoS-DB) containing the expected execution efficiency (best case).

4.4.2 Task scheduling

A task scheduler (TS) is responsible for distributing the workload among the other nodes. TS may also be easily bound to a core of the host system, which is not used to execute worker threads, to not limit performance. TS reads incoming tasks from a central task queue, which is created at the application launch time. Each entry contains the memory reference to a task. Every time TS makes an assignment, it moves the task reference to a task queue associated with the specific node. To be effective, the negative feedback control loop needs to sample the execution of the task. The result of such operation is a feedback value summarising the node state, which can be used to modify the task allocation eventually. To this end, the cost function (best case) for the task is compared with the cost function obtained at runtime. Every time the task efficiency is lower than the expected one, the task is moved to a more powerful node. Runtime information on the node state is collected through performance counters S_i. Instead of monitoring and adjusting the execution of a single task at a time, the system collects and processes values from several nodes. To this end, a vector s_i is compared with the expected vector y_i obtained from the QoS-DB, as follows:

$$\forall i : y_i = q_i - s_i \quad (4.3)$$

```
1   void fibonacci (void)
2   {
3       int n = …
4       // spawning new thread
5       int fib_thread_1 = ThreadCreate(…);
6       int fib_thread_2 = ThreadCreate(…);
7
8       // passing n-1 input value
9       ThreadMemWrite(…);
10      // passing n-2 input value
11      ThreadMemWrite(…);
12
13      // spawning new thread
14      int fib_thread_add = ThreadCreate(…);
15
16      ThreadMemWrite(fib_thread_add, …);
17      ThreadMemWrite(fib_thread_1, …);
18      ThreadMemWrite(fib_thread_2, …);
19
20      // terminate current thread
21      ThreadRemove(…);
22  }
```

Figure 4.4: A code snippet of the recursive Fibonacci kernel: the code highlight the software interface exposed by RADA, which simplifies the amount of code needed to synchronise threads' activities.

All the y_i values form a vector that is used by TS to trigger modifications in the allocation of incoming tasks. On the other hand, the assignment of tasks to the nodes is based on the values provided by a *score* function. These values are used by the TS to rank the nodes depending on their affinity with the incoming tasks. For each node, the TS calculates the value of the score function, which is described by equation 4.4. α and β are tunable parameters, while P_{util} and I/O_{util} represent the percentage (i.e., values normalised on a scale of 0 to 1) of utilisation by the task of processing core and I/O activities.

$$score = \left(1 - \frac{P_{util}}{0.8}\right)^{\alpha} \cdot \left(1 - \frac{I/O_{util}}{0.9}\right)^{\beta} \tag{4.4}$$

P_{util} is mainly based on the CPU utilisation which is also a good estimator of the system power consumption [FWB07, CHL+08]. In particular, a linear power model [KKH+09, CHL+08, FWB07] associated with the CPU usage is considered. Such a model assumes that power consumption of a system increases linearly as the workload increases. Although not used in this work, the power consumption of DRAM memory can be easily integrated into the model (e.g., using RAPL mechanism [DGH+10], it is possible to measure and set limits to the power consumption).

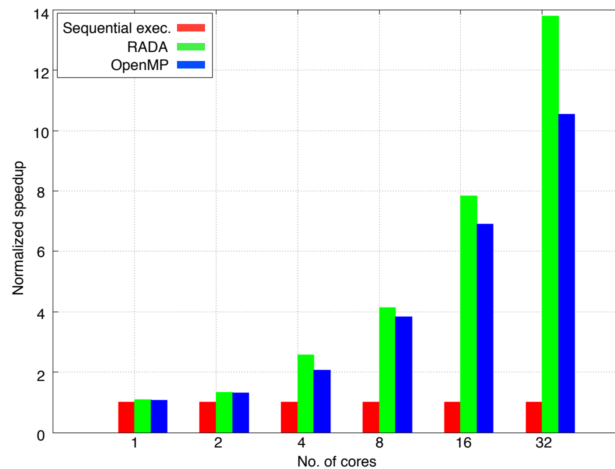


Figure 4.5: Recursive Fibonacci sequence: evaluation of the RADA and OpenMP execution.

4.5 Evaluation

A set of experiments has been carried out to evaluate both the capability of RADA to export useful information to drive the designer in tuning hardware resources, as well as software applications. The benefit regarding scalability and performance of adopting a dataflow execution model has also been demonstrated. For experiments, two popular applications are used: the recursive Fibonacci sequence (RFS), and the block matrix multiply (BMM). Figure 4.4 shows a small code snippet related to the RFS kernel: recursion is simply expressed by dynamically scheduling new threads. Thread interaction is mediated by the set of functions described in Section 4.3.1. The runtime library has been integrated into both the applications, allowing the creation of a single node multi-core execution system. The host platform was a server machine equipped with the Intel Core^{TM} i7-6700K running at 4.00GHz. The host server was also equipped with 32GiB of main memory and run the Ubuntu Linux distribution (version 14.04 LTS). Also, an Intel Xeon Phi P5110 (clock speed at 1.053GHz) accelerator board connected to 8GiB of local GDDR memory, was used to assess the capability of the proposed task scheduler. Figure 4.5 shows the performance of the RFS kernel with a fixed input size ($n = 35$), varying the number of available PEs (i.e., according to the implementation of the AMM described in Section 4.3, each PE is mapped to a worker process) in the system from 1 to a maximum of 32. During the experiment, it has been observed that further increasing the number of workers did not increase the performance. Thus, in experiments, it has been fixed to 32 workers per node (in this case, it equals up to 4 processes per physical core). The speedup to the behaviour of the sequential execution is normalised and compared it by a standard OpenMP implementation. The results clearly

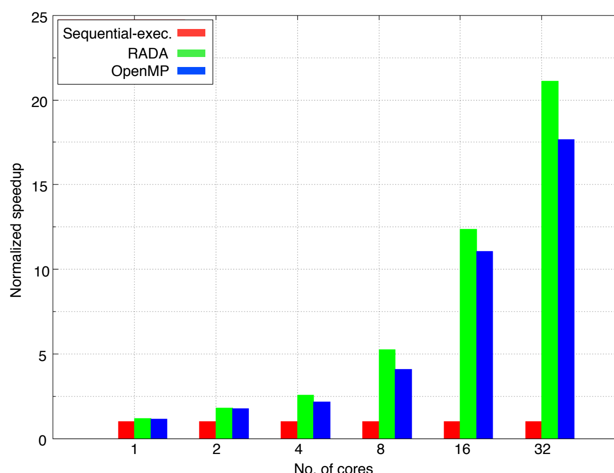


Figure 4.6: Block matrix: evaluation of the RADA and OpenMP execution.

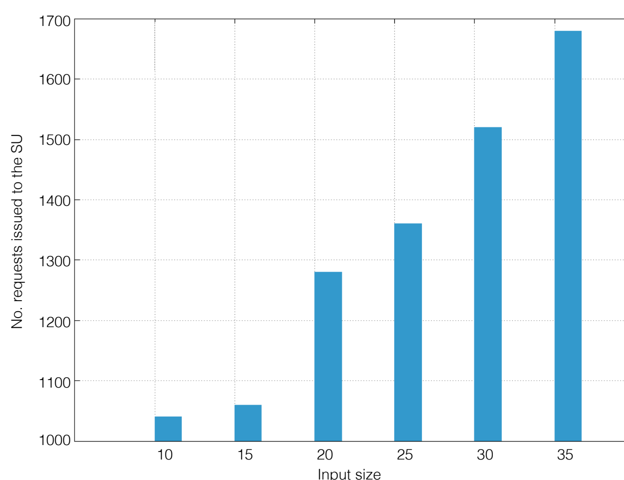


Figure 4.7: Number of requests issued to the SU by the recursive Fibonacci kernel (single node, 8 cores).

show that the dataflow version is more efficient than its OpenMP counterpart and that the dataflow execution can scale better when the number of running threads increases, this behaviour also in-line with the finding reported in the paper [SZG13]. Similarly, executing the dataflow version of the BMM kernel provides more performance and better scalability compared to the OpenMP version (see Figure 4.6). Figure 4.7 presents the number of requests generated by the execution of the RFS kernel on a single node. The number of requests issued to the SU remains low by increasing the input size of the application kernel. So, the adoption of a dataflow execution model allows limiting the synchronisation traffic. It also helps to explain the good performance of the RFS and BMM kernels on their OpenMP versions.

In table 4.1 and table 4.2, runtime execution parameters are captured which are respectively associated with the data structures held by the SU and by the PEs. The

Table 4.1: The execution trace (first 10 entries) for the RFS kernel with the input size set equals to $n = 10$.

Action	Frame	IP	FP	SC
RADA launched	–	–	–	–
Thread Scheduled	0	0x4039d7	0x1845780	1/1
Thread Scheduled	1	0x401820	0x18457e0	0/2
Thread Write	1	0x401820	0x18457e0	1/2
Thread Write	1	0x401820	0x18457e0	2/2
Thread Read	1	0x401820	0x18457e0	2/2
Thread Scheduled	2	0x401820	0x1845840	0/2
Thread Scheduled	3	0x401820	0x18458a0	0/2
Thread Write	2	0x401820	0x1845840	1/2
Thread Write	3	0x401820	0x18458a0	1/2

Table 4.2: The execution trace (first six entries) captured from one PE when running RFS kernel with size $n = 10$.

FF		WT			RT	
FP	Status	FP	IP	SC	FP	IP
0x1fc4010	Alloc.	0x1fc4010	0x402de8	1/1	0x1fc4010	0x402de8
0x1fc4070	Alloc.	0x1fc4070	0x4011f9	2/2	0x1fc4070	0x4011f9
0x1fc4070/0	Alloc.	0x1fc4070	0x4011f9	1/2	0x1fc4070/0	0x4011f9
0x1fc40a1	Free.	–	–	–	–	–
0x1fc4070/1	Alloc.	0x1fc4070/1	0x4011f9	0/2	0x1fc4070/1	0x4011f9
0x1fc4070/1	Alloc.	0x1fc4070/1	0x4011f9	0/2	0x1fc4070/1	0x4011f9

tables show the main parameters related to the dataflow execution (such as the frame memory block identifier (Frame), the instruction pointer to the thread code (IP), the pointer to the frame block (FP), and SC). The column SC represents the number of received inputs over the number of required ones. For instance, the fourth entry in table 4.1 indicates that the thread requires two inputs and only one has been received. The information provided by the captured traces is presented in a structured way (see table 4.2) and it is very helpful for the designer to understand the interaction among the threads, as well as the allocated resources. It is worth to mention that the set of the parameter that can be captured is larger, and it can be selected by the user when the RADA is launched. These two tables also show the flexibility of the proposed tool.

Finally, figure 4.8 shows the score function obtained from the execution in a heterogeneous environment. The internal performance counters are used to generate the score function while running on Intel Xeon Phi accelerator. This feature has been used effectively to schedule the execution of the BMM kernel on both the host server CPU and the accelerator. Matrices have been partitioned into blocks, and parallel tasks performed the multiplication of such blocks. Due to the dataflow execution and the larger

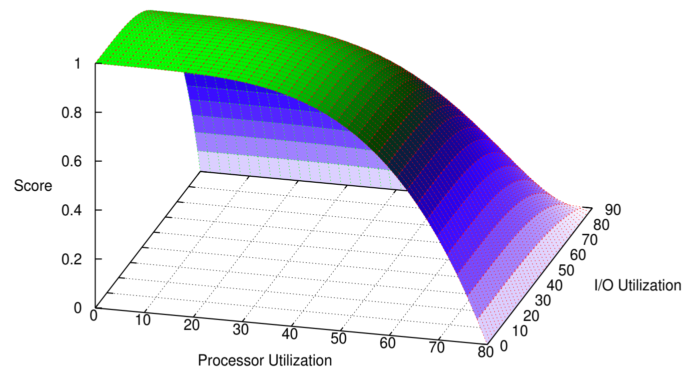


Figure 4.8: Score function obtained from the execution of the BMM kernel running on host CPU and Intel Xeon Phi accelerator.

parallelism offered by current accelerators, a significant speedup in the execution has been obtained.

4.6 Summary

This chapter presents RADA, a tool for a fast and comprehensive analysis of dataflow-based applications. RADA provides a simple software interface that dataflow applications can exploit to execute and gather information at runtime. Information captured with RADA are helpful for the designer to tune correctly the application and the hardware resources allocated to it. Initial results show the significant benefit of using the dataflow execution model on two popular applications, as well as the capability of RADA to manage heterogeneous execution platforms. However, the presented work is not complete, and it still needs many further improvements.

As future work, the support of feedback unit of the model will be improved with robust policy and also the software interface support will be increased from its initial proposal. The proposed cost model will also be improved with more certain system variables. The current traffic generation is not complete, thus in future, the traffic generation in memory hierarchy levels will also be considered. An accurate timing model is needed to extend RADA for robust analysis of the applications. Finally, an area estimation model needs to be integrated to assess critical design parameters related to new thread schedulers. Finally, the mentioned list of modifications will be added into the model before making the source code public to the users. In the next chapter, a platform based on software defined NoC has been proposed for managing the dataflow threads.

4.7 Acknowledgement

This chapter is an edited version of the accepted paper entitled **Analysing Dataflow Multi-Threaded Applications at Runtime** by Somnath Mazumdar and Alberto Scionti at the 7th IEEE Advance Computing Conference (IACC-2017).

5

Thread Management at Software defined NoC

How to monitor the dataflow threads using a simple tool has been proposed in Chapter-4. Now, this chapter presents the mechanism to manage data-driven threads at Software-defined NoC (SDNoC) level efficiently. Moving from Petascale to Exascale computing, there is an assorted effort in microarchitectural domain aiming at increasing the performance/power ratio of multicores. Future manycore processors will contain thousands of low-powered processing elements to support the execution of a large number of concurrent threads. Albeit data-driven program execution models (PMXs) are gaining popularity due to their effective support for thread communication, frequent data exchange among many concurrent threads put stress on the underlying interconnect subsystem, which results in hotspots and high latency for data packets.

In this chapter, this challenge has been addressed by proposing a scalable *Software defined Network-on-Chip* (SDNoC) architecture for future manycore processors. The proposed microarchitecture tries to merge the benefits of ring-based NoCs (i.e., performance, energy efficiency) with those brought by dynamic reconfiguration (i.e., adaptation, fault tolerance) while keeping the hard-wired topology (2D-mesh) fixed. To possibly accommodate different application requirements and communication patterns, the proposed interconnect maps different types of topologies (*virtual topologies*). Few customised instructions are added to the core ISA to allow the software layer to control and monitor the NoC subsystem. For supporting a data-driven PXM, a set of instructions are also designed. In experiments, the lightweight reconfigurable architecture with a conventional 2D-mesh interconnection subsystem has been compared.

The chapter is organised as follows: Section 5.1 provides the relevance of the problem and Section 5.2 discusses the problem and provide the proposition for the proposed design. Section 5.3 briefly explain the proposed software interface for managing threads,

while Section 5.4 describes extensions enabling the evaluation of application's execution on a NoC system. Finally, Section 5.5 presents experimental results, while Section 5.6 summarises the chapter with the findings and also with the future works.

5.1 Introduction

Modern silicon technology allows integrating billions of transistors into a single die, making possible to wrap thousands of processing elements (PEs) on a chip (kilo-core CMPs) [BC11]. Such massive number of PEs provide the substrate for effectively executing an enormous number of concurrent threads (up to hundreds per PE in future Exascale machines). To alleviate limitations in the way power is dissipated [EBA+11], current designs tend to favour the use of a large number of single-issue, in-order cores, which consumes less energy than their multi-issue, out-of-order counterparts [CAB+13, Jac]. Another source of improvement derives from the exploitation of hardware-software resource usage monitors, which are very helpful when considering parallel applications that can vary their size during the execution. Albeit these techniques contribute to exploiting the parallelism of kilo-core CMPs fully, the use of classical von Neumann program execution models (PXMs) makes multithreaded applications suffering from a large synchronisation overhead, while it becomes difficult to guarantee their correct and race condition free execution [DG08]. Conversely, data-driven PXMs (e.g. the Codelet model [SZG13]) allow the implementation of self-scheduling and isolation of execution properties, which limit the synchronisation overhead and ease the application to grow and shrink over time. However, the support from these new PXMs to concurrency imposes a major challenge to the interconnect subsystem, compared to the cores. Software-defined networking (SDN) is getting popular by its flexibility to separate the hardware packet forwarding mechanism from the control logic [NMN+14]. SDN allows us to integrate network control mechanisms with the application software, thus facilitating the application execution with a better networking support.

Starting from these premises, this chapter focuses on the NoC-based interconnection subsystem by proposing a scalable NoC architecture, which merges the advantages of ring-based interconnects, with those carried by the reconfiguration. Instead of relying on a fixed topology, an effective way for dynamically changing the topology of the network itself is introduced. A reduced set of instructions is added to cores' ISA to allow the software to control the topology of the network during execution directly. The proposed architecture can be referred as a *Software defined Network-on-Chip* (SDNoC), thanks to the flexibility in managing the underlying interconnection subsystem at the application level [SAPMVA+16]. Targeting applications where threads are hierarchically organised and dynamically scheduled [SZG13], adaptation allows for a better exploitation of the

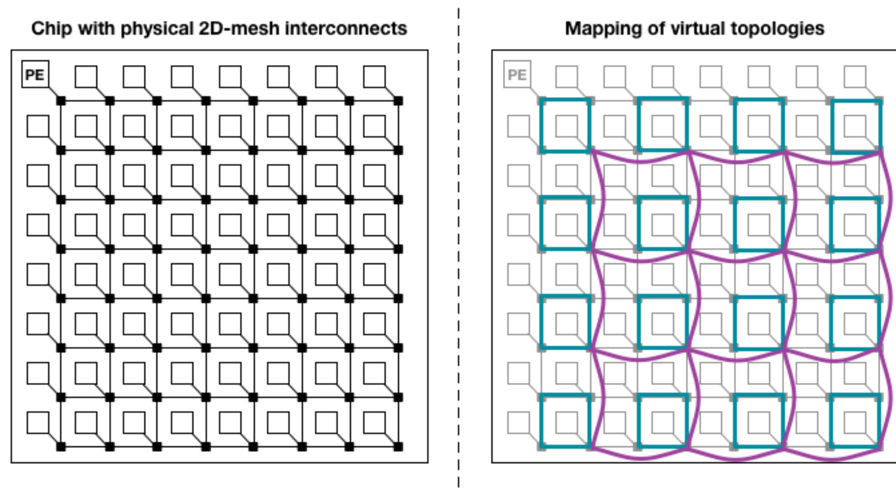


Figure 5.1: Mapping between the physical network with a 2D-mesh topology, and a multi-level virtual topology. Links to the physical network are organised into local rings (blue lines) and a global 2D-mesh among rings (purple lines).

network resources and a greater power saving. This architecture allows the software to control each single link with a fine granularity using ring-based interconnects as the building block for implementing the physical 2D-mesh topology. As a result, physical links can be switched off when not used. Link usage is monitored as well using internal hardware counters and made accessible through dedicated instructions. This information can be exploited by the programmer and the optimisation tools or compiler to better adapt the virtual topology to the communication patterns of application.

5.2 System overview

The proposed system is encompassing a large number of computing tiles. Within a tile, a PE directly communicates with a router (R), which is responsible for delivering data towards the other tiles. Routers are interconnected through a physical 2D-mesh network, which provides a scalable support for chip-level communications. The 2D mesh topology has become very popular in complex many core designs (e.g., TILE [EWB^{+}07]) because it offers advantages compared to other alternatives regarding wiring area, power cost, and fault tolerance [BCGK04]. The left side of Figure 5.1 depicts the organisation of the target system: white squares (PEs) communicate with routers (black squares) which are connected each other through the 2D-mesh network (black lines). To exploit a massive parallelism of such kind of architecture, an explicit data-driven PXM has been considered. This execution model assumes that the application is divided into a set of fine-grain threads, each composed of up to few hundreds of instructions. Threads are

assumed to exchange data with others through an explicit producer-consumer scheme. To preserve locality and allow a better latency hiding, threads can be grouped within the same group (*virtual node*), and threads can also be scheduled for the execution on PEs that are closed to each other.

5.2.1 Problems and its solutions

The adoption of a standard 2D-mesh physical interconnection presents three major challenges that can seriously limit scalability. First, with the increase in the number of connected tiles, the NoC quickly consumes a large portion of the chip power budget. Secondly, with the increase in the number of concurrent threads, the network bandwidth is rapidly consumed. Finally, fixed network topology does not support the principle of locality exploited by a data-driven PXM, as well as it does not adapt to the changes in the communication patterns. Overcoming these criticalities requires the capability of the network to be dynamically configured. Being able to switch off not used links selectively, allows the network to save power, while still supporting local communication among threads (belonging to the same virtual node). To provide a solution, several works focused only on specific aspects. Kim et al. [KK09] propose the use of ring-based routers for implementing a scalable 2D-mesh topology. Thanks to a simpler microarchitecture, routers can scale better and consume less energy than conventional ones. However, the topology of the network is fixed, thus lightly used links cannot be excluded. On the contrary, Panthre [PDB14] integrates reconfiguration features in the router microarchitecture. Although it allows a significant power saving, routers are based on the conventional microarchitecture organisation which is power hungry (i.e., the majority of the dissipated power is due to the crossbar switch and link drivers).

The presented work tries to merge the benefit of lightweight and scalable router microarchitecture [KK09, LNLK13] with those of dynamic reconfiguration. Four separate rings allow the communication is flowing in the north, south, east and west directions, while specific bits control the status of each link. During the configuration phase, the links can be set in three different modes: *normal*, *bypass* (BP), or *power-gated* (PG). Bypass and power-gated modes disable partially or fully the links, thus allowing the network to save power and to implement different topologies at the same time. The right side of Figure 5.1 shows, an example of virtual topology mapping, with groups of adjacent PEs communicating through virtual rings (blue), which are in turns connected through a virtual mesh (purple). For instance, routers of PEs 3 and five can disable respectively south and north links, while the router of PE 6 can configure north link in bypass mode so that its traffic is directly injected in the south link towards router of PE 8. In this configuration, about 37.5% of the links can be switched off, while still

preserving communications at the chip level. Performance is ensured by the possibility of using high-speed clocks, thanks to the simplified router microarchitecture. Furthermore, each of the links is equipped with a counter for tracking traffic statistics whose value is exported to the software layer through a very small instruction set extension (ISE).

5.3 NoC software interface

The ISA of PEs is extended with few instructions to manage the reconfiguration phases, to monitor the links' traffic, and to support thread operations. Since the proposed interconnection architecture is agnostic with regards to the PEs' architecture, here, a generic interface (for the ISE) is reported. Each instruction can be conveniently wrapped by a function in standard high-level languages (e.g. C/C++) as follows:

- `SetRouterCfg($RD,$RS,B)`: sends a configuration request to the routers, by specifying the memory address where the configuration is stored. Parameter \$RD is the destination register of the destination router, \$RS is the source register, and flag B (unsigned immediate value) indicates if the request is actually sent in broadcast to all routers ($B>0$), or not ($B=0$).
- `ReadCounter($RD,$RS1,$RS2)`: reads the content of a link's counter, by specifying the link to read (one of the four bits starting from the LSB position in the \$RS1 register must be set), the destination router (source register \$RS2), and the register where the counter content will be stored (\$RD).
- `ResetCounter($RD,$RS,B)`: allows to reset traffic statistics, by specifying which links' counters to reset (four bits starting from the LSB position in the source register \$RS are set if the corresponding links' counter must be reset), the destination router (destination register \$RD), and a flag (B) that indicates if the request is actually sent in broadcast to all routers ($B>0$), or not ($B=0$).
- `CreateThread($RD,$RS1,$RS2)`: schedules a new thread, by specifying the code pointer (\$RS1), if it belongs to the current virtual node (\$RS2>0) or not (\$RS2=0), and where the unique thread identifier is stored (\$RD).
- `ReadData($RD,$RS1,$RS2)`: reads data (\$RD) from a local memory block (\$RS1) associated to the thread \$RS2.
- `WriteData($RD,$RS1,$RS2)`: writes data (\$RS1) to a local memory block (\$RD) associated to the thread \$RS2.

- `DeleteThread($RD)`: marks the thread ($RD) as completed, allowing associated resources to be released.

Routers are equipped with a simple hardware logic function that is in charge of scheduling a new thread whenever the `CreateThread` instruction is executed, as well as to forward data to other PEs when executing `ReadData` and `WriteData` instructions. In addition, each router is equipped with a 16×1 SRAM containing the *switch table* (ST). It describes how traffic entering a link can flow in other links. More precisely, the memory content is logically arranged as a 4×4 matrix. Each row of the matrix corresponds to an input link, while the four columns in a row represent output links. An element $s_{i,j} \in ST$ is set to 1 if the traffic travelling in the direction i (e.g., west) can be forwarded on the direction j (e.g. north). The content of the ST, as well as the values of BP and PG bits, are stored in memory in the form of a bitstream. Every time the `SetRouterCfg` instruction is executed, it generates a corresponding message sent to a specific router. The destination router directly accesses to the memory location (actually the PE performs this operation) where the configuration is stored, by adding a fixed offset to the basic address ($RS). The operation can be parallelized if multiple routers need to configure.

5.4 Proposed Network-on-Chip architecture

The basis of the proposed system is a 2D-mesh NoC implemented with lightweight routers. Figure 5.2 shows the complete router microarchitecture. The communication on each of the four directions (i.e. north, south, east and west) is ensured by dedicated *ring stations* (RSs), which are essentially responsible for forwarding the traffic coming from the injection port (inj) to the output port, or to extract it through the ejection port (ejc). RS manage traffic travelling on the same direction form a physical ring. The original router design [LNLK13] uses a single *inter-ring switch* (IRS) to switch traffic travelling horizontally (i.e. from east-to-west, or from west-to-east) to one of the vertical directions (i.e. from north-to-south, or from south-to-north), but prevent the opposite traffic steering (i.e. from vertical to the horizontal direction). This choice was imposed by the implementation of the X-Y routeing algorithm on top of the ring-based architecture. Although this organisation ensure a chip-level communication within the physical mesh, it does not support the mapping of virtual topologies. To compensate, we provide a more flexible architecture: two symmetrical IRS, i.e., an X2Y switch and a Y2X switch allow the traffic to spill from horizontal directions to vertical ones, and vice-versa. To this end, the design uses the X-Y routeing protocol without constraints on how to steer traffic in vertical or horizontal direction. Thus packets can be routed alternatively on the X and Y

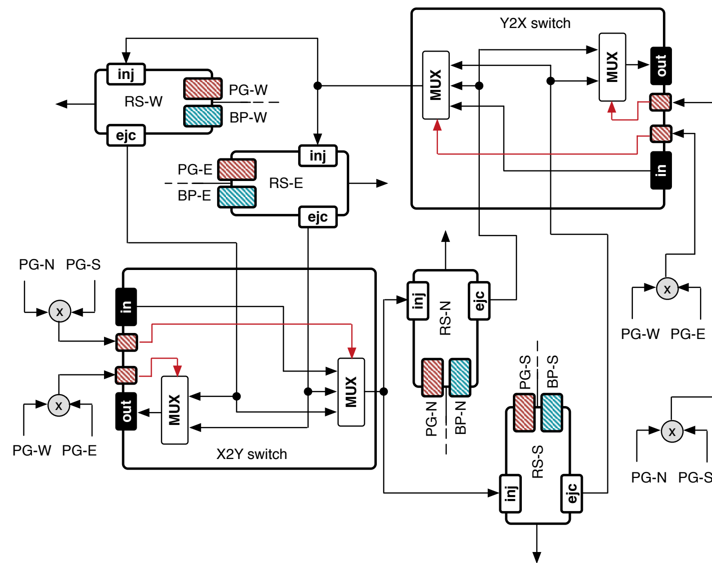


Figure 5.2: The lightweight router microarchitecture. Ring stations (RSs) have injection and ejection ports, and bypass (blue squares) and power-gating (red squares) bits. Inter-ring switches are power-gated depending on the state of the RS (grey circles are OR/AND gates).

dimension. Each of the IRS is composed of two multiplexers. One is responsible for selecting which of the inputs (e.g., traffic coming from the local PE, east-to-west direction, or west-to-east direction) has to be transferred to the injection ports of the RS in the opposite dimension (e.g. the traffic is injected in the north-to-south direction, or in the south-to-north direction). The second one selects, which traffic to present to the output port (i.e., this is the spill point, where traffic coming from the network is presented to the PE). For instance, the X2Y switch can present to the output port of the traffic coming from the ejection port of the RS-W and RS-E (see figure 5.2). Other two separated ports in each IRS (highlighted in red) control their state, thus allowing for selectively power-gating one of the two internal multiplexers or both. Considering the X2Y switch (similar is the case for the Y2X switch), the multiplexer connected to the output port can be disabled only when the RS-W and RS-E are in power-gated mode. The multiplexer responsible for forwarding the traffic in the opposite dimension is power-gated only when RS-N and RS-S are in power-gated mode. In a similar manner to IRS, each RS is provided with two additional ports controlling the state of the station itself. Power-gated port (highlighted in red) fully disables the RS, so that traffic arriving at the input port is dropped. Bypass port (highlighted in blue) partially disables the RS, allowing the traffic arriving at the port of entry to be directly injected on the output link. These two ports are controlled by two corresponding configuration bits: respectively PG and BP bits. While the configuration of the IRS depends on the state of the RS controlling them,

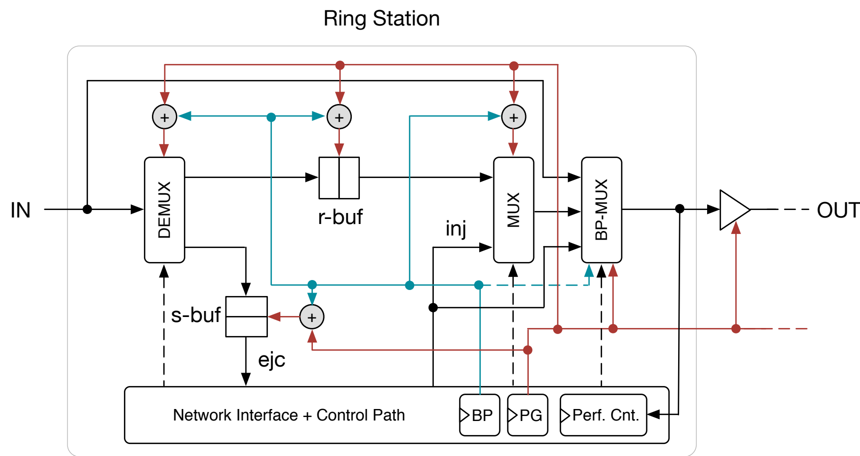


Figure 5.3: The internal structure of a RS with BP/PG bits and the link counter in the network interface (dashed lines represent selection signals for multiplexers/demultiplexers).

the state of RS can set independently from the others. Furthermore, the power-gated mode is dominant in the bypass mode, meaning that the configuration $\langle BP, PG \rangle = \langle 1, 1 \rangle$ leads the RS to be set with the power-gated mode. The internal organisation of an RS is depicted in Figure 5.3. Traffic arriving at the input port (IN) can continue to travel in the same direction or can be extracted. This decision is implemented within the demultiplexer as part of the routeing logic and requires only to analyse the destination field of incoming packets. Every time input traffic has to be ejected (i.e. extracted by the local PE or deflected in the opposite dimension), it temporarily stored in a local buffer (S-BUF). Similarly, the buffer R-BUF temporary stores packets that continue to flow in the same direction. It is worthy to note that both the buffers can be tiny compared to input buffers in conventional routers. The limited storage space required by the buffer is mainly due to the adoption of a traffic prioritisation policy, which allows the RS to privilege traffic that flows in the same direction, over the one coming from the injection port. This policy is implemented as a part of the selection mechanism of the output multiplexer (MUX). It is worthy to note that multiple R-BUF and S-BUF buffers can be used to support virtual channels. Another multiplexer (BP-MUX) selects which traffic to inject on the output link (OUT) when the RS is set in the bypass mode. In this mode, both the internal buffers, the input demultiplexer, and the output multiplexer are disconnected from the power source (power-gated), while traffic on the injection port is directly injected on the output link. In this way, the RS can save energy and limit the packet traversal delay. Interestingly, in this case, the drivers of the output link are still fully active. On the contrary, whenever the RS is set in the power-gated mode, all the internal components and the link drivers are switched-off. Four additional OR gates

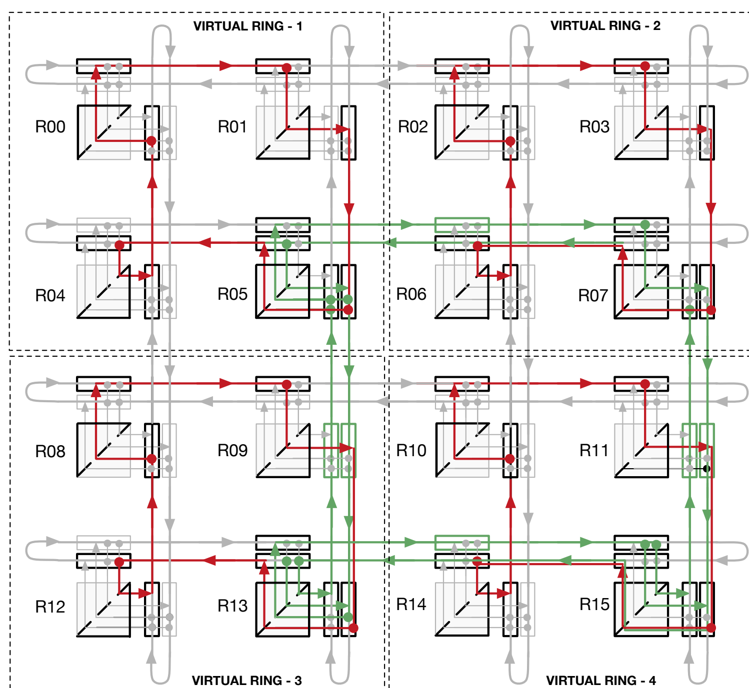


Figure 5.4: An example of virtual topology mapping: Grey structures represent components (i.e., interconnections, RSs or inter-ring switches) of the router that are power-gated. Red lines correspond to active links used to build local rings, while green lines show links of the mesh. Furthermore, green boxes represented components set in bypass mode and used to construct the mesh among the virtual rings correctly.

ensure that BP or PG bits control power-gate and bypass functionality for the internal components. As highlighted in the previous section, PG bit overrides the BP signal. Finally, a 16-bit counter is available for each output link. Every time a packet (flit) traverse the link, the counter automatically incremented. A single bit in the control router logic allows selecting the granularity at which traffic is monitored (i.e., if counters are updated when packets or single flits traverse the links). Figure 5.4 shows an example of mapping a hierarchical topology (i.e., local rings and a global mesh) upon a 4×4 physical mesh.

5.5 Evaluation

The evaluation has been done for the proposed architecture based on an in-house simulator to test scalability, considering synthetic random traffic and a matrix multiplication kernel written to exploit the data-driven capabilities of the underlying hardware. The simulated manycore design comprises up to 1024 cores implementing a simple 5-stage

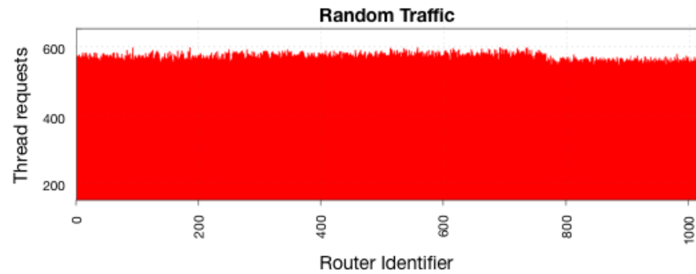


Figure 5.5: Distribution of random traffic over 1024-based CMP.

in-order execution pipeline (16 KiB I-cache + 16 KiB scratchpad memory). Each core supports a subset of the RISC-V ISA, as well as the execution of special instructions designed to manage threads and network configuration (see Section 5.3). Cores running at 2.0 GHz are directly attached to a 2-stage lightweight router clocked with a higher frequency (4.0 GHz). Power consumption and area are modelled by integrating Orion 3.0 [KLN15] and DSENT [SCK+12] models into our simulation tool. Uniform random traffic represents the worst case scenario regarding scalability since it uses software to take advantage of the network reconfiguration capability, and it maintains almost all the links active. To simulate this scenario, a pool of thread requests that are consumed by randomly selecting the cores needed to setup. Also, each core randomly selects the number of requests to consume (e.g., scheduling a new thread on a different core). The number of processed requests is directly proportional to the traffic generated by the routers on the network. Figure 5.5 shows the traffic distribution for a 1024-core CMP with a pool of 580K requests. Without loss of generality, a single virtual channel has been used. With this configuration, the overall power consumption of the interconnection is 47.13 W, which is less than 58.80 W of the conventional counterpart (a power saving of 19.9%). Traffic generated by data-driven applications are more deterministic, thus offering greater opportunity for power saving. A matrix multiplication algorithm following a data-driven paradigm by using dedicated instructions is implemented, as presented in Section 5.3. The application has five dynamically scheduled threads, organised as follows: threads in charge of computing the same output element ($C_{i,j} = A_i \times C_j$) exchange data on a local virtual ring, arranged as a 8×2 matrix. A global mesh enforces global communication. In this configuration (of 1024 cores), the overall power consumption is reduced at 16.20 W, while the execution time is in line with the execution on conventional 2D-mesh-based CMP. Also, by varying the number of available cores, an almost scalar speedup has been experienced, while the reconfiguration of the whole chip requires less than 3000 CPU cycles. When comparing the area occupation, the proposed solution is 39.4% less expensive than conventional routers, offering more opportunity for design scaling. These preliminary results show the benefits of using a

dynamically configurable and lightweight interconnection for manycore CMPs.

5.5.1 Additional Discussion

In this sub-section, we will discuss on the used simulation tool and also about the compatibility with the previous work.

5.5.1.1 Simulation Environemnt

We use an in-house simulator for performing experiments. The tool has been implemented in C/C++ for improving simulation speed. The simulator is based on a distributed approach where the simulation of each PE is bind to a specific core of the host simulation machine. In this way, the speed of simulation is improved against a single-core approach. The basic idea is to simulate the execution of a dataflow thread recurring to the implementation of a subset of the RISC-V ISA. We have added a small set of custom instructions to that ISA, to interact with our proposed SDNoC. In our implementation, we started from a simple in-order architecture, with a 5-stage pipeline. The in-house simulator is still under development state, and at the present state, it has not full capability of performing cycle accurate simulations. The core for the in-order PEs of the implementation has been inspired by other projects available online (such as RISC-V Rocket core). We extended the implementation by adding the interface with our NoC. So in each host core, a single tile is simulated. Since we are relying on dataflow execution model, the synchronisation of the activities among simulated tiles is achieved by setting a very restricted number of the mutex to protect shared resources (i.e., essentially the frame memory belonging to a specific PE, more than one thread tries to update it by writing a new value).

5.5.1.2 Compatibility

RADA (chapter 4) provides a software interface that dataflow applications can exploit to execute and gather information at runtime. This chapter proposes a microarchitecture that tries to merge the benefits of dynamic reconfiguration. Here both the work assume that interconnect subsystem is 2D-mesh based. In both the works, few similar customised instructions are added to the core ISA to allow the software layer to control and monitor the system. Apart from that, they both support data-driven PXM. So incorporating of two models are possible while the monitoring tools will use the extended data-driven APIs to monitor the thread execution, and SDNoC approach will help to reduce the data traffic via its supports to virtual topology.

5.6 Summary

The design of an Exascale machine requires improvements in all levels of the system, from application to chip microarchitecture. In this context, the interconnection subsystem plays a key role, since it is required to exchange demanded data among thousands of PEs efficiently. In this chapter, an integrating reconfiguration features into a lightweight router microarchitecture (in such a way that the software layer can directly control the network topology) has been proposed. Preliminary simulations confirm the advantages of such interconnection architecture. This work can be considered as the first brick of the framework, in next chapter a complete hybrid (ring and 2D-mesh topology based design) has been proposed and tested in FPGA which can be effectively used by the dataflow PXMs for further performance improvements.

5.7 Acknowledgement

This chapter is an edited version of the published paper entitled **Software defined Network-on-Chip for scalable CMPs** by Alberto Scionti, Somnath Mazumdar and Antoni Portero in IEEE International Conference on High Performance Computing & Simulation (HPCS), (pp. 112-115), 2016.

6 Customised NoC Architecture

Modern computer chips wrap a large number of homogeneous, small, low-power processing elements in a single die. Packing many cores together offer an excellent parallelism, but comes with multiple issues, mainly related to the overall energy efficiency of the chip. Recent times, power-aware computing is becoming popular for the increasing power demand for computation. Enhancing the performance per unit of energy has become the primary goal of the current generation of hardware designs. Interconnects play a fundamental role in information exchange of a large number of processing elements integrated on the same die. From this perspective, it becomes clear that inefficient interconnect subsystems quickly become the limiting factor for achieving high-performance in the chip. Efficient interconnects are required to support the information exchange of such a vast number of processing elements (PEs), still providing low latency, high network throughput and scalability with a better area and power costs. In this chapter, a hybrid, scalable and efficient Network-on-Chip (NoC) architecture has been proposed, which is designed to support future manycores chips. The proposed NoC design remains agnostic on the architecture of the PEs. It fuses rings and 2D-mesh topology to provide high-performance while processing local (rings) and global (mesh) traffic efficiently. Our experimental results show that our NoC architecture is scalable up to 1024 PEs with robust performance in multiple traffic scenarios compared to the well-known 2D-mesh topology. The proposed design can improve the average throughput and also reduce the average latency for a 1024 cores system. Similarly, the proposed design is also power efficient compared to well-known 2D-mesh topology based NoC design.

The chapter is organised as follows: Section 6.1 provides an overview of the importance about an efficient NoC for large core counts. Section 6.2 provides the overview of the target system, and Section 6.3 describes the actual implementation of the NoC design with details of modified router architecture, ring switch and the proposed packet for efficient processing, while Section 6.4 presents the details about the achieved latency,

throughput, area and power cost. The comparison with 2D-mesh topology has also been performed. Finally, Section 6.5 presents the possible applicability of the proposed design together with possible future works, while Section 6.6 summarised the chapter with the findings.

6.1 Introduction

Traditionally, HPC applications are either compute or communication-centric. However, there is no easy way to categorise the traffic generated by the applications in the interconnect subsystem of CMPs [BWFM09]. At runtime, threads communicate at different levels, and the flow of data they generate depends on several architectural factors (e.g., type and topology of the interconnection, the presence of distributed memory banks, the number of level in the cache hierarchy). With the growing adoption of massive multi-threaded applications, the amount of data exchange among the PEs started to push the limits of traditional interconnections. Furthermore, with the emerge of physical limitations in the way heat is removed from the chip, to achieve higher performance, data exchanged inside the interconnect needs to be optimised focusing on bandwidth, latency and energy cost. In fact, inefficient interconnects may lead to reduce the overall system performance and consume a significant portion of the area and power budget of the chip [HVS+07].

Moving from multicore to manycore designs equipped with hundreds to thousands of PEs [BSP+16a], the probability of resources contention greatly increases. Thus, the amount of conflict-free resource sharing inside the chip should be maximised to reduce power cost and also to improve the performance. In scalable interconnect architectures (e.g., Networks-on-Chip – NoCs) all PEs share the communication resources, hence increasing the possibility of contention and quickly defeating the advantage of substantial parallelism provided by the higher core count. To unleash the full capability of today's and future CMPs, NoC based interconnect must offer high bandwidth, low latency, possibly memory coherency support, and better I/O integration. In the past, NoC demonstrated to be a possible solution for implementing massively parallel CMPs, thanks to the advantages offered compared to other alternatives regarding wiring area and power cost [LCOM07, BCGK04]. For instance, bus topology was very popular but started to suffer from high energy consumption, low scalability, and low bandwidth for larger core counts. The main reason is ascribed to the physical capacitance of bus wires which grows with the number of connected modules, thus resulting in a growing wire delay. For a small number of PEs, ring topology [DT04b] demonstrated to be very effective, requiring a low-radix router. Interestingly, a ring topology can outperform mesh topology for moderate to high memory access locality based workloads [RS97]. Rings also have

been used in commercial systems (e.g., in the Intel Xeon Phi co-processors where a dual ring topology is used). The main advantage of using ring is its deterministic latency to reach the destination. However, similar to a bus, the ring interconnects also suffers from low bandwidth for large core counts and becomes difficult to scale to hundreds of cores due to its limited bisection bandwidth. Efforts, such as [AFY+16, HJ01, VBS+95], have been made to make local and global rings connecting via "bridge routers" to improve the scalability together with performance and better energy consumption. On the other side, 2D-mesh topology has become very popular to connect a large number of PEs (e.g., TILE [EWB+07], Polaris chip [HVS+07]) due to its scalability and the highest level of fault tolerance. However, 2D-mesh topology suffers from space and power trade-offs [HVS+07, VHR+08a, BD06] for very large number of connected cores which is also verified during our experiments. In fact, a 2D-mesh router using an internal crossbar switch can consume the largest amount of the power budget [HVS+07]. So, to overcome these issues, proposals combining 2D-mesh with rings have been presented. Where, [BZ07, AFY+16] are such few examples of hierarchical topologies.

In particular, power efficiency becomes a major concern for NoC design while connecting several hundreds of cores inside the chip. In fact, the on-chip network can consume a large fraction of the whole chip power budget. For instance, the NoC for the MIT Raw processor [WPM03] can consume up to 36% of total system power; while Vangal et al. [VHR+08a] showed that on the Intel TeraFLOPS chip, the NoC uses up to 28% of tile power. Other experiments [HPD12] have shown that for large core count (i.e., a 256-core based CMP) conventional 2D-mesh NoC consume up to 45% of the total energy. However, NoC performance and power consumption are very much influenced by routing mechanism and topology. Specifically, topology impacts network latency and power consumption.

Thus, an efficient interconnection plays a vital role in providing a scalable, high-performance communication medium for very large core counts. In this chapter, a hybrid on-chip interconnect is proposed for kilocore CMPs, where rings and 2D-mesh NoC are combined. This hybrid approach can exploit the applications traffic's localisation [BAM10, KSG+09] to confine the traffic of various applications inside the rings for a better traffic management. The proposed architecture provides a customised router architecture which processes the traffic and can also bypass it (if necessary). The ring topology has been chosen because of their simple design, which provides contention-free traffic without consuming unnecessary power. A 2D-mesh topology has a large bisection bandwidth but suffers from the large diameter. Hence, the proposed approach aim to use 2D-mesh interconnect for high-speed data transfer between far away PEs, while exploiting rings for local data exchange between PEs that are close to each other. A tiled-CMP with a very high number of PEs (up to 1024 in our experiments) has

been envisioned here, (similar approaches based on NoC already been proposed such as [ZGHC15, GHKM11]). The proposed architecture is completely agnostic with regards to the specific architecture of the PEs, thus supporting the implementation of customised accelerators on FPGA devices also. There is a growing attention to FPGA devices mainly for their reconfigurable capabilities. Modern devices offer enough hardware resources to implement computing systems equipped with hundreds of specialised cores. In this context, saving hardware resources used by interconnecting logic greatly contribute to reduce overall power and area cost. The high efficiency of the proposed NoC architecture allows the use of a minimal amount of hardware resources, which contribute to saving energy and also to reduce the area cost of the accelerator.

In this chapter, the contributions can be summarised as *i)* the modified hybrid NoC design is detailed here and the hybrid ring-mesh based NoC topology is evaluated by implementing it on an FPGA device. *ii)* It has been shown that the design can scale from few cores up to 1024 cores, and also it outperform the standard 2D Mesh topology with lower latency and higher throughput and providing a better power efficiency. *iii)* Finally, to conclude the paper how this approach can benefit applications which can dynamically vary the requirement of some allocated resources (mainly cores) (e.g., MapReduce and applications adhering to an explicit dataflow execution model) has also been discussed.

6.2 System overview

On-chip packet-switched micro-network of interconnects (i.e., NoCs) provides the physical substrate used by PEs to communicate each other and also to the memory. Hybrid topologies exploit the fact that most of the communication in a parallel application affects a group of resources (i.e., PEs and routers) that are close each other [DEM+09]. Hence, the optimisation of the local communication may lead to a large improvement regarding packet latency, throughput and energy efficiency. A two-level hierarchical interconnect is designed by combining the capabilities of small rings to provide high-performance and high-energy efficiency, along with a conventional 2D-mesh topology for delivering packets between PEs that are located far away from each other. By combining these two physical topologies, the scalable design can achieve high-throughput, lower latency, and better energy efficiency, with regards to conventional 2D-mesh flattened design.

At the basis of the proposed design, there is the observation of how the traffic moves in motorways and takes exits. Once the traffic exits the main motorway, it is injected into small roads where simple decisions need to be taken. Conversely, more complex decisions and management policies are needed at the level of the global interconnect. Proposed design provides two levels of communications: global traffic is managed by

a more complex crossbar switch based, modified 2D-mesh routers, while local traffic is injected or ejected by small rings. Traffic travelling in these two levels of the hierarchy is decoupled and processed differently to improve the system throughput and reduce the communication latency (similar to [AFY+16, UMB10, DEM+09]). Although mesh routers are more complex than switching stations (i.e., switch modules responsible for driving the traffic in the ring or injecting/ejecting it towards the attached PE or the mesh) of the rings. The majority of the traffic remains restricted to the rings, thus allowing the proposed architecture to achieve good levels of performance and efficiency by exploiting the localisation of the application traffic.

The proposed architecture is aimed at connecting up to 1024 PEs and uses the deterministic and deadlock free X-Y routing algorithm. Figure 6.1 depicts the main architecture of the reference CMP. A group of four PEs are locally connected through a small ring called *ringlet*. Within a ringlet, one of the PEs is designated as the master core. It is responsible for injecting/ejecting traffic towards the global traffic channels. For this, a link between the ring switch and a mesh router is enabled along with dedicated buffers. The advantage of this architecture is the absence of a dedicated bridge component to connect the mesh and the ringlets. The proposed NoC is divided into blocks, and a group of four ringlets forms a *block* unit. These four ringlets are directly linked to the mesh router, which is responsible for moving traffic outside the *block*. For instance, to support 256 cores, 16 modified mesh router and 64 ringlets are needed. More complex traffic management policies can be applied at the block unit level, and multiple blocks are globally connected through a 2D-mesh topology. To support traffic (in and out) in the mesh network, each router is equipped with a high performance 8×8 internal crossbar switch. The smart packet processing implemented in the mesh routers allow decoupling global and local traffic. Every time the destination of a packet is outside the local block, the packet is forwarded to another mesh router, thus bypassing PEs in the ringlets, and minimising the overall latency.

6.3 Proposed Network-on-Chip architecture

In this section, the main components of the proposed NoC microarchitecture are described. Specifically, the internal organisation of the mesh router and the ring switch for the master core.

6.3.1 Modified 2D-mesh router

Figure 6.2 depicts the internal organization of the mesh router, while Table 6.1 provides its main architectural characteristics. The router employs a 8×8 crossbar switch to

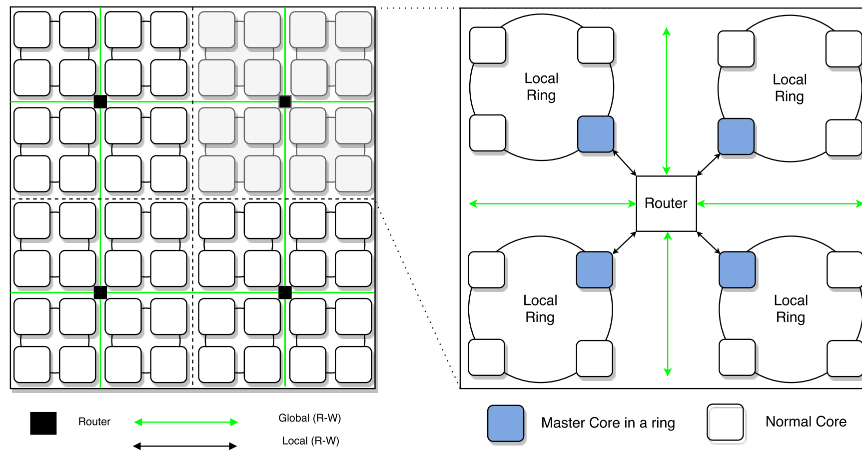


Figure 6.1: An instantiation of the proposed scalable NoC: 256 PEs organised into 4×4 block units, each connecting four ringlets.

support global traffic movement in both dimensions (i.e., north-south and east-west) and traffic exchange with local ringlets. Four channels are used for driving traffic within the 2D-mesh network (global traffic). The other four input channels are used to steer traffic to/from the ringlets (local traffic). Each ringlet is associated with a dedicated channel so that the traffic exchange with the master ring switch happens through this dedicated link. In general, routers can have a significant number of VCs to hold a large amount of incoming traffic. For each input link, two virtual channels (VCs) are introduced which allow for better support quality of service (QoS) and also prevent deadlocks. In Figure 6.2 the path taken by control information carried by the packet headers is highlighted in red, and with blue lines show the control signals activated by the internal router stages. Conversely, output channels do not use VCs, thus contributing to saving power. Large buffers requirements and QoS overheads reduce the ability to support a high number of cores with an efficient area and energy usage [GHKM11]. Also, a large number of VCs consume a huge chunk of energy, since more input buffers are needed to keep traffic separated. It is worth to mention that buffers are one of the largest leakage power sources in the router. Their power consumption can represent up to 64% of the total router's leakage power [CP03] (sometimes comprise up to 74% of the total NoC power budget [SCK$^+$12]), and also a significant amount of dynamic power [WPM03]).

Interestingly, single-flit packets represent the large segment of the network traffic for real applications [MJW12]. Following this, the router is optimised for managing single flit packet. In the design, a packet length of 42-bits is selected, where 32-bits are used to transport data, and the remaining 10-bits are devoted to carrying header information. The size of the packet has been chosen to take into account that increasing the packet size, leads to a quadratic increment of the internal crossbar switch overhead. Thus the packet size is maintained as small as possible [LNP$^+$13]. The internal router

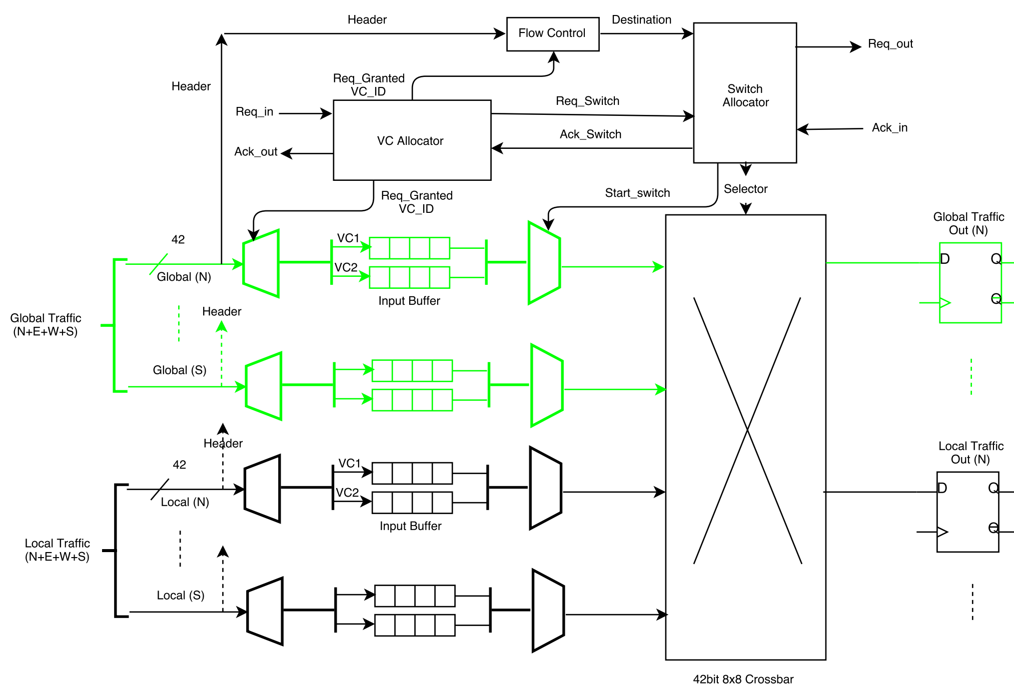


Figure 6.2: Modified 2D-mesh router microarchitecture: two groups of local/-global channels are used to manage traffic within the 2D-mesh and traffic exchange with local ringlets.

is organised into a four stages pipeline: routing stage, flow-control stage, VC allocation stage, and switch allocation stage. However, with the aim of reducing the latency of the packets to traverse the router, the proposed design has been optimised in such a way the operations performed by the four stages can happen in parallel, thus reducing the overall latency to 1 cycle. The entire packet transfer can be restricted to a single cycle thanks to the following design choices: (i) the adoption of the store-and-forward mode; (ii) the optimisation of the routing logic for processing the single-flit packet with a reduced overall size. These design choices lead to a router architecture with a latency of one cycle in most of the cases. It is worth to note that both wormhole and virtual-cut-through do not offer any significant comparative advantage. Since that, the entire packet can be processed in parallel by the routing logic. The employed routing mechanism is based on the X-Y dimension order routing (XY-DoR) algorithm since it provides a simple implementation with a deterministic routing latency. Decoupling the traffic between local ringlets and mesh, the probability of congestion in the 2D-mesh becomes negligible, so the need for an adaptive algorithm (e.g., hot potato routing [Bar64], also known as "deflection routing") disappears. In particular, the routing logic with the flow-control module is fused together. A speculative allocation technique for both the VC allocation stage (VCA) and the switch allocator stage (SA) has also been implemented.

Table 6.1: Mesh-router: main microarchitectural parameters.

Features	Parameters
No. of input and output ports	8 each (4 ringlets, 4 mesh)
Width of each port	42-bits (32-bits payload, 10-bits header)
No. of Virtual Channel	2 per input port
Packet switching	Store-and-Forward (SAF)
Switch allocator arbitration	Round-robin
Packet Routing	X-Y dimension order routing
Router pipeline stages	4 stages

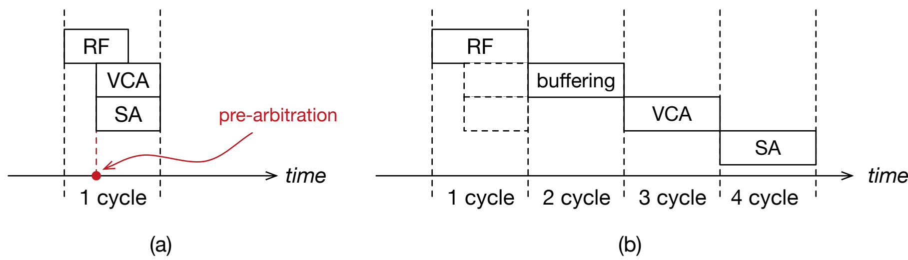


Figure 6.3: Timing: a) best-case (success) and b) worst-case (failure) of pre-arbitration.

In case the pre-arbitration fails, the packet is buffered while VCA and SA arbitration are performed sequentially. In that case, the latency increases up to four cycles. The timing of the proposed mesh router in the event of pre-arbitration success shown in Figure 6.3 (a), while the event of failure represented in Figure 6.3 (b). Every time there is an incoming packet, the following operations are performed by the router's modules:

- *Routing/Flow control module* (RF) extracts the packet header and processes the information to determine the destination router. In case the packet destination is within one of the four ringlets belonging to the block, the RF module selects the corresponding output channel, reducing the latency of the VCA and SA module. A control signal is thus used to drive the input multiplexer (MUX) at the input port. In this phase, speculative operations are performed to pre-allocate channels.

- *VC allocator module* (VCA) is responsible for allocating buffer resources for incoming packets by selecting one of the VCs. An allocation request signal (i.e., req_{in}) is set, and if the selected VC has space to buffer the incoming packet, an acknowledge signal (i.e., ack_{out}) is set too. In that case, the selected VC is also signalled both to the RF module and the SA module.

- *Switch allocator module* (SA) performs two steps of arbitration. First, multiple VCs in each input port are arbitrated to select one the available VC. Then, each one of the selected VCs is routed to the selected output port.

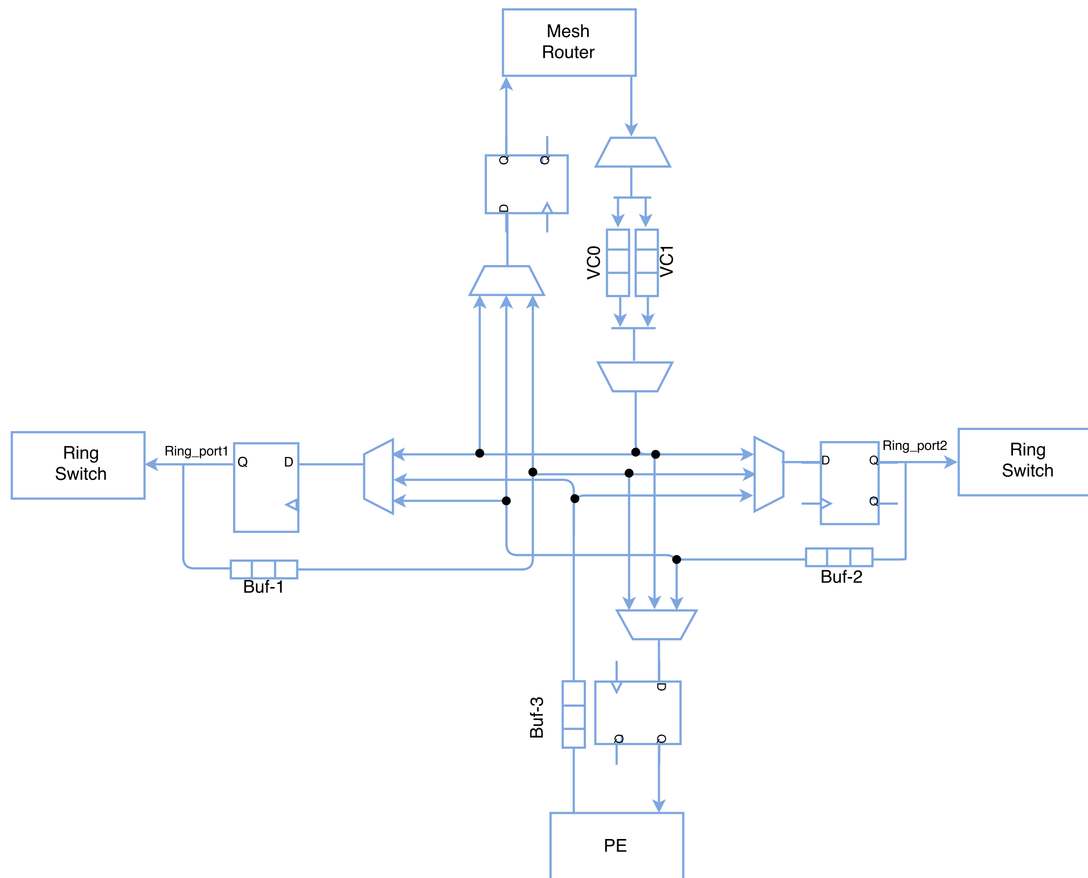


Figure 6.4: The microarchitecture of the RS of the ringlet's master: horizontal dimension is used to create the bidirectional ring connection, while vertical dimension connects the mesh router and local PE of the ringlet.

6.3.2 Ring switch

A bidirectional ring is implemented upon the structure of a *ring switch* (RS) to achieve a high-performance while keeping power consumption low. The RS is responsible for driving the traffic within the ring, and also to steer it towards the mesh router or the local PE. Figure 6.4 depicts the microarchitecture of the RS. Compared to conventional RSs, the microarchitecture is customised by incorporating buffers (similar to VCs) to allow the ring to steer the traffic to/from the mesh router. The RS is composed by two main multiplexers which manage the traffic within the ring.

To avoid a complex control logic, the RS uses prioritisation of the traffic travelling in the same dimension (i.e., traffic that remains within the ring and moves in the same direction). Prioritisation helps to reduce the size of internal buffers too (buffers Buf-1 and Buf-2, see Figure 6.4). In particular, one of the two directions is selected as with high priority, thus moving first in the RS. Prioritisation mechanism is implemented directly in the control logic of input-output multiplexers. The interface with the local

PE is implemented using a dedicated buffer (i.e., Buf-3) which is written by the PE and the RS reads it (i.e., the PE injects traffic in the ring). The buffer is accessible by the PE within its address space. Similarly, traffic that is ejected by the ring is collected temporarily in a local output buffer, from where the PE can extract the payload. The interface with the mesh router is implemented in a similar way: traffic injected in the mesh is stored temporarily in a small buffer, from where it is transferred to the input link of the mesh router. Traffic that is ejected from the mesh router is moved within a VC buffer. When the mesh router tries to access the RS, two VCs are implemented to support resource contention better. From this viewpoint, the RS implements a round-robin selection strategy between the two VCs to keep control logic simple. When packets move within the ring or between a ring and the mesh router, the following steps are performed by the RSs:

- The multiplexer of each input port determines the destination based on the packet's header information and also based on the arbitration.
- Packets from the ring ports (see Figure 6.4, horizontal dimension) have higher priority compared to packets coming from the processing core or the mesh router. Thus, such packets are moved first from the input port to the output port, with a minimal delay. This arbitration strategy also ensures that packets already in the main ring traffic flow are quickly routed to prevent the saturation of the network. Specifically, to enable the transfer, the RS sets the request signal of the next switch in the ring (by following the travelling direction of the packets), waiting for the acknowledge signal to be set by the peer switch.
- When the master RS receives a request from the mesh router to inject packets in the ring, the available two VCs buffers are used to temporary store the packets. If there is space in the selected VC buffer, the RS enables the corresponding acknowledge signal of the mesh router. Each buffer will take turns to send out the packets via round-robin arbiter to exhibit fairness.

It is worth to note that, to minimise the amount of resources used by routing structures, RS modules which are not connected to the mesh router have the same structure depicted in Figure 6.4, except for the mesh router interface. In that case, this interface has been removed to save area and power.

6.3.3 Packet header processing

One important aspect of designing a high-performance interconnect is represented by the organisation of the packets. To maximise the performance of the proposed NoC architecture, the mesh router and RS microarchitecture are optimised considering packets

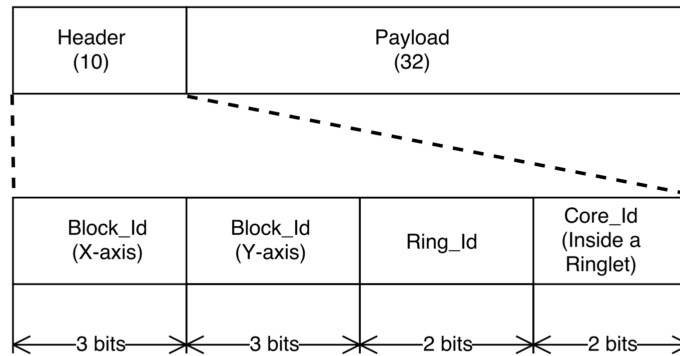


Figure 6.5: Packet header organisation.

composed of a single flit. The structure of such packets is represented in Figure 6.5. Packets have a fixed length of 42-bits: 10-bits of header information and 32-bits of payload. In particular, packet header information has been organised in such way its processing time is minimised. The payload can be used mainly to transport application data. The packet structure can be further augmented for fine grain control of the NoC (see subsection 6.5.1).

The header carries information on the destination of each packet, and similarly to standard TCP/IP network connections, each packet is treated independently from the others. In the case of packets belonging to the same flow, they will be processed in the same manner, so that a deterministic latency can be achieved. Specifically, the packet header is organised into four sub-fields. Two sub-fields are used to uniquely identifying the PE within the ringlet and the ringlet connected to the mesh router. This two sub-fields are used to move traffic among the ringlets belonging to the same block. Since there are four ringlets in a block, each containing four PEs, the length of these two sub-fields is set to 2-bits each. On the other hand, the remaining two sub-fields are used to identify the destination router within the 2D-mesh. To uniquely specify the block identifier 6-bits is needed since the target system is composed of up to 1024 cores that are further organised into a grid of 8×8 blocks. Out of six bits, 3-bits are used to identify packet destination on the X-axis and 3-bits for its position on the Y-axis. To further improve scalability and efficiency of the proposed architecture, bypass logic is also implemented to save energy. Bypassing the packets also reduce the latency [KPKJ07]. This feature is included in the mesh router. The proposed bypass logic uses *left-to-right* parsing (see Figure 6.5) for implementing header processing. Specifically, the following steps are executed to process the packet header information:

- *Traffic confined in the ring*: the block identifier sub-fields are reset, while ring and PE identifiers are set.
- *Traffic injected/ejected to/from the 2D-mesh*: all sub-fields are set; while the by-

pass logic is disabled.

- *Traffic confined in the 2D-mesh*: the bypass logic is enabled.

6.4 Evaluation methodology

The proposed NoC architecture is evaluated taking into account the following parameters: average network latency, average throughput, power consumption, and area cost. Since the proposed interconnect design well fit with an implementation on FPGA devices, the area cost has been evaluated regarding allocated hardware resources. The RTL for the entire VHDL design description of mesh routers and ringlets is synthesised. Next, corresponding FPGA bitstream is generated using Vivado Design Suite 2016.2. All the components of the proposed NoC design have been implemented and validated on a Xilinx Virtex-7 XC7VX690-3 FPGA device, which is the most potent variant of its class. All the tests were done setting the clock speed to 400 MHz. In the following, the results of four parameters are reported by comparing the proposed architecture with the reference design (based on a traditional flattened 2D-mesh interconnect).

6.4.1 Cost metrics

Cost metrics are represented by the power consumption and the resource utilisation when the proposed design is implemented on the FPGA device. In the following, the obtained results are analysed separately.

6.4.1.1 Resource utilisation

Two designs on a relative scale are compared and the values are reported as the percentage of the total used resources (see Table 6.3). To this end, LUTs, FFs and Block RAMs (each 36Kb in size) are counted. In the proposed design, for the implementation of a single block unit (i.e., four ringlets connected to a mesh router), the total number of used LUTs, FFs and BRAMs is 2434, 2768 and 48 respectively. Specifically, four ringlets consume a total of 1076 LUTs, 1800 FFs and 40 BRAMs, while resource consumption of the mesh router is reported in Table 6.2.

In Table 6.2, the resource utilisation and power consumption between a standard 2D-mesh router with the proposed design are compared. Unlike a standard router, the modified one can support sixteen cores via four ringlets with around 2× increment in the resource consumption compared to a traditional mesh router and with a less than 0.4W increment of power consumption. Regarding the power consumption, the values of static power (due to leakage currents and which depends on the manufacturing process)

Table 6.2: Area and power comparison between a standard router architecture and the proposed mesh router.

	Resources Utilization			Power Consumption (in Watts)	
Router	***LUTs***	***FFs***	***BRAMs***	***Static***	***Dynamic***
2D-mesh	699	572	5	0.323	0.047
Proposed	1358	968	8	0.324	0.075

Table 6.3: Relative resource utilisation in Vivado (values are in percentage).

	System Configuration (No. PEs)						
	16	**32**	**64**	**128**	**256**	**512**	**1024**
	Proposed router design						
LUTs	0.31	0.63	1.25	2.51	5.02	10.03	20.06
FFs	0.11	0.22	0.45	0.89	1.79	3.58	7.15
BRAMs	0.54	1.09	2.18	4.35	8.71	17.41	34.83
	Ring switch design						
LUTs	0.25	0.50	0.99	1.99	3.97	7.95	15.90
FFs	0.21	0.42	0.83	1.66	3.32	6.65	13.30
BRAMs	2.72	5.44	10.88	21.77	43.54	87.07	174.15
	Conventional 2D-Mesh router design						
LUTs	2.58	2.11	4.23	20.65	41.31	82.61	165.23
FFs	1.06	2.11	4.23	8.45	16.90	33.80	67.60
BRAMs	5.44	10.88	21.77	43.54	87.07	174.15	348.30

and dynamic power are presented separately. Results show that the proposed design consume 1.0 mW and 28.0 mW more respectively regarding static and dynamic power. This result is strictly correlated to a large number of memory blocks (BRAMs) used by the proposed design.

Resource utilisation of the block (unit topology of proposed design) with publicly available FPGA friendly NoC generator CONNECT [PH12] is also compared. It is worth to note that the CONNECT has been tested at 150 MHz due to its failure on higher clock speeds, while for the proposed design the clock frequency was set to 400 MHz. In this experiment, the design saves 74.65% of LUTs and 39.51% of FFs compared to CONNECT (official source code was not changed) to connect sixteen cores via an on-chip network. CONNECT also has used 1728 DRAM blocks (64-bits each) while the proposed design has used 48 BRAM blocks (36-kbits each).

Scaling the system up to 1024 PEs, 64 modified mesh router and 256 ringlets are needed. Such architectural blocks consume up to total 155776 LUTs, 177152 FFs and 3072 BRAM blocks. From the Table 6.3, it can be seen that the proposed model is very

resource efficient compared to the standard flattened 2D-mesh design. For connecting sixteen cores (one block unit), it can save 2% LUTs, 0.7% FFs and 2.2% of BRAM compared to the standard 2D-mesh topology. Although, it might seem small saving when the design is scaled to 1024 cores the resource saving increases up to 129.3% for LUTs, 47.2% for FFs and 139.3% for BRAMs. It is interesting to note that overall, as the NoC size increases by doubling the number of cores, the resources consumption of the proposed model grows linearly as $\approx 2\times$, which may imply that the design is modular and well scalable.

6.4.1.2 Power consumption

In Figure 6.6, the static and dynamic power distribution for all the NoC configurations has been presented. It can be identified that, initially, the static part dominates the power consumption while as the size of the network grows it started to diminish. Interestingly, it is also noticed that the static part does not increase as the network size increases but stays almost constant. For example, the total static power consumption for one topology block (16 cores) is 0.649 W while it grows up to 0.796 W for 1024 cores. It can also be seen from the Figure 6.6, where the amount of static power, is more related to the implementation of the mesh routers (if compared to ringlets). However, it is worth to note that ringlets in all cases consume more FFs and BRAMs, while router consumes more LUTs (specifically, in the range of 0.06% to 4.2% (see Table 6.3)).

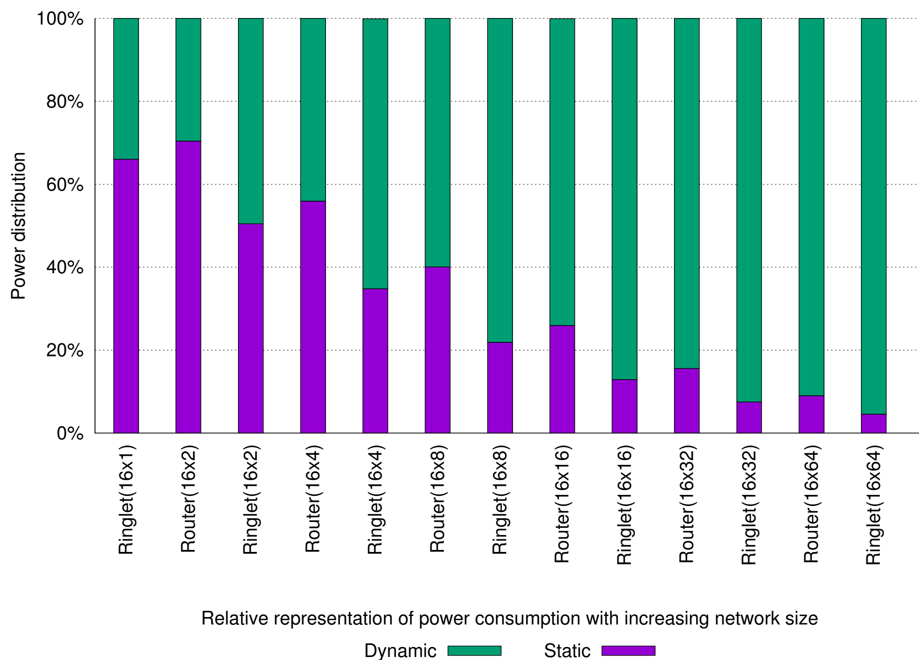


Figure 6.6: Static and dynamic power distribution

Next, the power consumption comparison between the proposed model and the standard 2D-mesh in Figure 6.7 is presented. Here, for the proposed architecture the energy consumption of both the mesh router and the ringlet is distinguished. For one topology block (16 cores), the power consumption is 0.399 W and 0.492 W respectively, for the mesh router and the ringlet. However, as the size of the network grows, the total power consumption of ringlets starts to dominate. For instance, for 16 topology blocks (i.e., 256 cores), the power consumption of routers is 1.276 W while the 64 ringlets consume 2.703 W, which is more than $2\times$ of total routers energy consumption. Following this trend, for 1024 core configuration, all the ringlets consume around $2.5\times$ of the total routers power consumption. Apart from that, for network size of 16 cores, both the proposed design and the flattened 2D-mesh consume almost the same amount of power. However, as the network grows, the 2D-mesh starts to consume more power. For instance, with a 16×8 cores configuration, the proposed model consume 2.4 W while the conventional design consumes 4.5 W. The situation becomes worse when it touches 32.8 W for connecting 1024 cores, which represents 141.26% relatively more power compared to the proposed design.

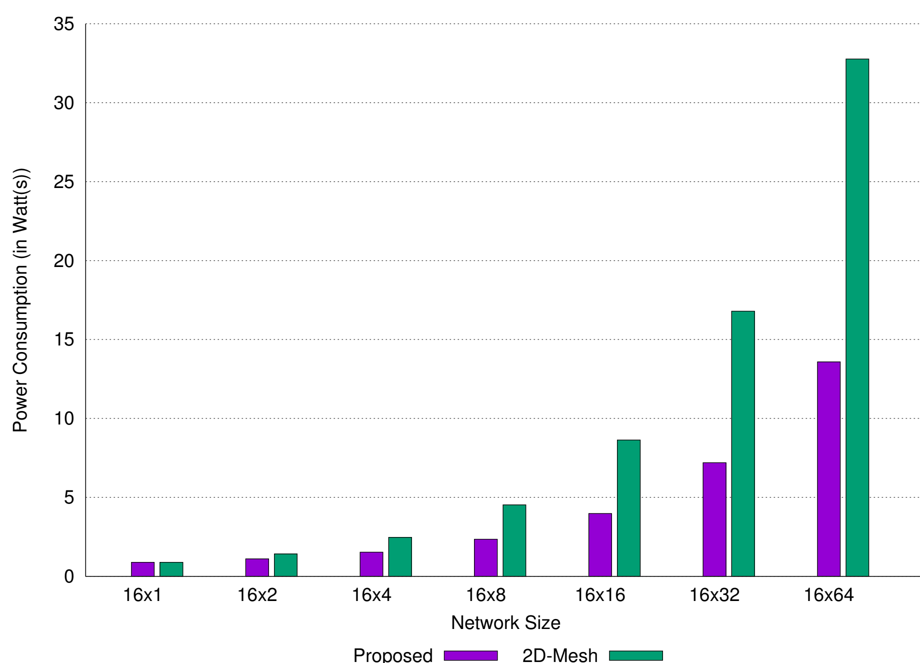


Figure 6.7: Total power consumption with increasing network size.

6.4.2 Performance metrics

The proposed routers and ringlets are tested under well-known three statistical traffic patterns, and the performance (throughput and average latency) of the hybrid topol-

ogy is evaluated. VHDL-based cycle-accurate models for the traffic pattern generation was developed. In modern computing applications, communication requirements are dynamic and unknown before the execution. Three well-known traffic patterns (such as uniform random, bit-reversal and transpose [DT04b]) are considered to represent the way the real-world applications generate their data traffic. Mainly the network latency (i.e., the time for a packet to move from source to destination, including the time for a packet to cross the channel) and the throughput is evaluated. Bit-reversal and transpose do not support smooth traffic operations. For the experiment, a large number of packets that needs to be independently routed to a dynamically determined destination are synthetically generated. Four packet injection rates (I_r, measured in packets/cycle): specifically $I_r = \{0.25, 0.50, 0.75, 1.00\}$ are used. For $I_r < 1.00$, nodes generating traffic are selected randomly, while with $I_r = 1.00$ (worst case) all the nodes inject packets at the same time. The stress-test was to ensure the working capability of the proposed NoC design under both bandwidth and worst/average latency scenarios.

6.4.2.1 Network latency

Figures 6.8, 6.9, 6.10 show the average packet latency as a function of the four injection rates, when the three different traffic patterns are used. Bars show that the network latency increases with the increased size of the injection rate, as well as the increase of the network size. However, the proposed system shows very much consistency with increasing network configuration. For low injection rates (i.e., $I_r = \{0.25, 0.50\}$), the latency for each traffic pattern remains very consistent with the others. When increasing the injection rate up to $I_r = 0.75$ packets/cycle and using the bit-reversal traffic pattern, the latency is minimised. The worst case for the packet latency is represented by the transpose traffic pattern with an injection rate of 1.00 packets/cycle.

When comparing the proposed architecture with conventional flattened 2D-mesh, it is found that for all the three traffic patterns, the proposed design outperforms traditional NoC design, by keeping the latency lower. Specifically, analysing the behaviour of the 2D-mesh NoC, it is found that 2D-mesh design is very consistent for latency increments for all the cases, while it also has its largest latency for the transpose traffic pattern with the injection rate of $I_r = 1.00$ packets/cycle (similar to the proposed). The latency is improved by 10% for all three traffic patterns for smallest network configurations (i.e., 16 cores). While, latency is improved by 120.13% for uniform-random traffic, by 114.6% for transpose traffic, and finally by 123.6% for bit-reversal for the largest case (i.e., 1024 cores).

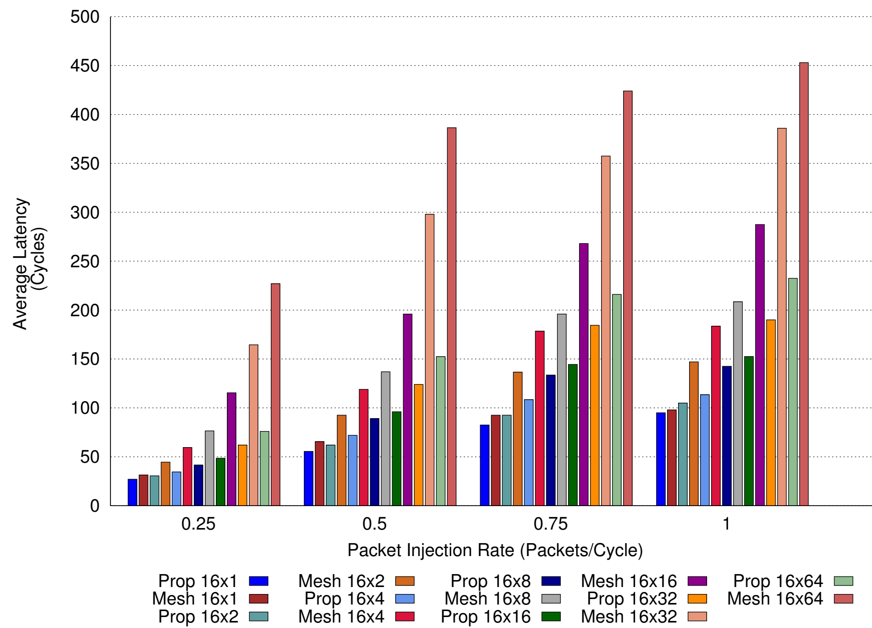


Figure 6.8: Average packet latency in uniform random traffic pattern.

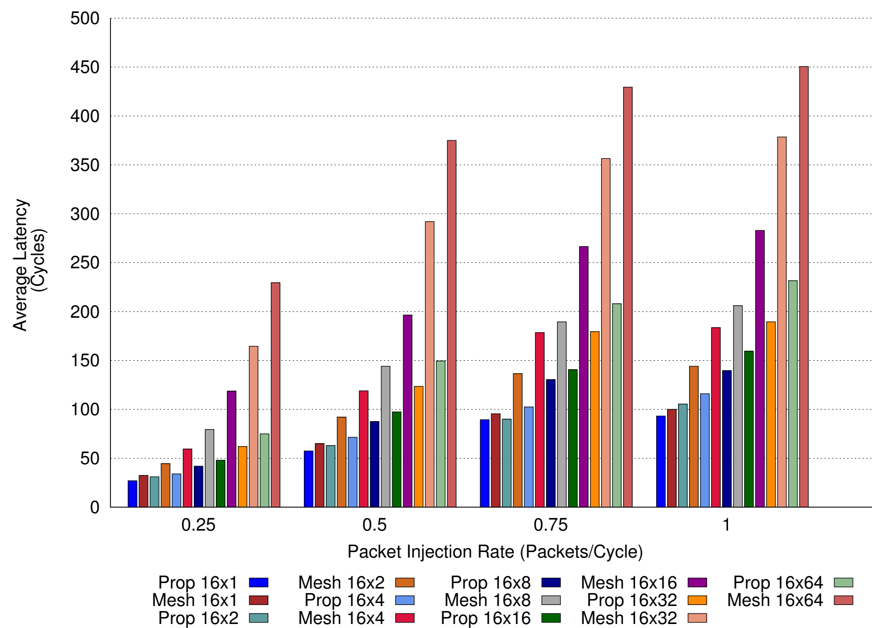


Figure 6.9: Average packet latency in bit-reversal traffic pattern.

6.4.2.2 Network throughput

Figures 6.11, 6.12, 6.13 show the achieved network throughput for all three traffic patterns. Similar to latency, the network throughput is also consistent with the offered packet injection rate. In fact, in the proposed design the average throughput increases as the number of PEs increases. From this viewpoint, by analysing the number of pack-

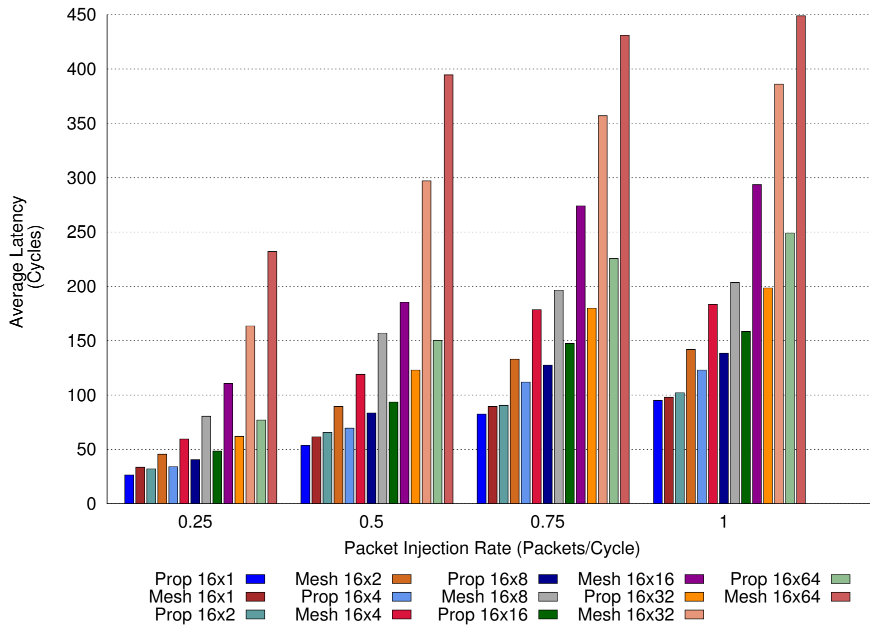


Figure 6.10: Average packet latency in transpose traffic pattern.

ets delivered per cycle, an increase of 2× factor with the increase of the number of PEs in the network has been observed. For instance, with an injection rate equals to $I_r = 1.00$ packets/cycle and a uniform random traffic pattern, the average throughput increases from 12 packets/cycle for a single block unit (i.e., 16 cores) to 22 packets/cycle for two block units (i.e., 32 cores). Similarly, for a configuration with 512 cores, the average throughput is 344.5 packets/cycle, while it increases up to 680 packets/cycle for a 1024 cores configuration. Again, it represents approximately an improvement of a 2× factor with the network size doubling. This trend is also followed by the proposed design when other traffic patterns are considered. It clearly shows that proposed NoC architecture is capable enough to offer higher performance and scalability compared to traditional flattened 2D-mesh. Interestingly, a similar trend in 2D-mesh throughput has also been observed. Conversely, when the injection rate is low (i.e., $I_r = \{0.25, 0.50\}$ packets/-cycle), proposed design has performed better for the transpose traffic pattern. For an injection rate equals to 0.75 packets/cycle, the proposed design performed well for all the traffic patterns, while for largest network configuration (i.e., 1024 cores), again the design shows the best throughput for transpose traffic pattern. However, considering the worst injection rate case, it is worth to note that best throughput is achieved with the uniform-random traffic, while traditional 2D-mesh topology did not demonstrate a similar consistency among the patterns.

These results clearly show that the proposed design can improve the performance of the NoC regarding higher throughput and lower average latency compared to the traditional 2D-mesh topology. The capability of this design to sustain such performance

also with high injection rates and random traffic patterns (which represent a critical pattern) can be mainly ascribed to the hierarchical organisation of the network. In fact, most of the traffic is kept inside ringlets or is exchanged by ringlets connected to the same mesh router. Such organisation (ringlet-oriented) is the main contributor to the scalability of the proposed design.

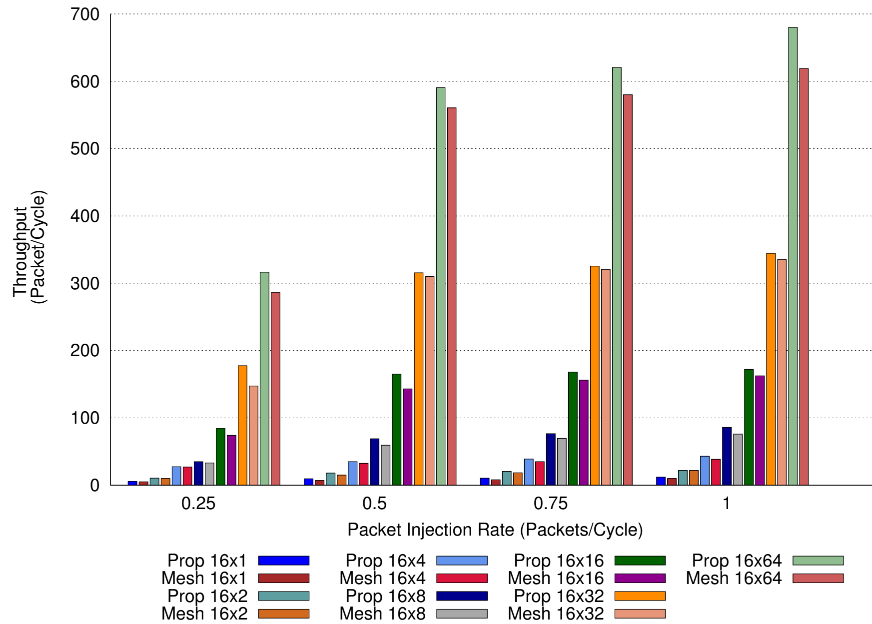


Figure 6.11: Average network throughput in uniform-random traffic pattern.

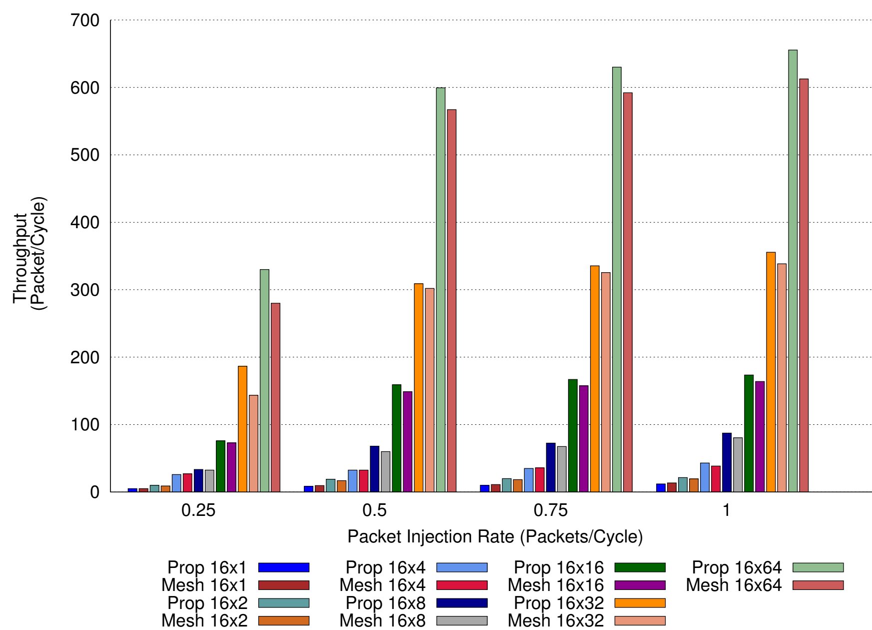


Figure 6.12: Average network throughput in bit-reversal traffic pattern.

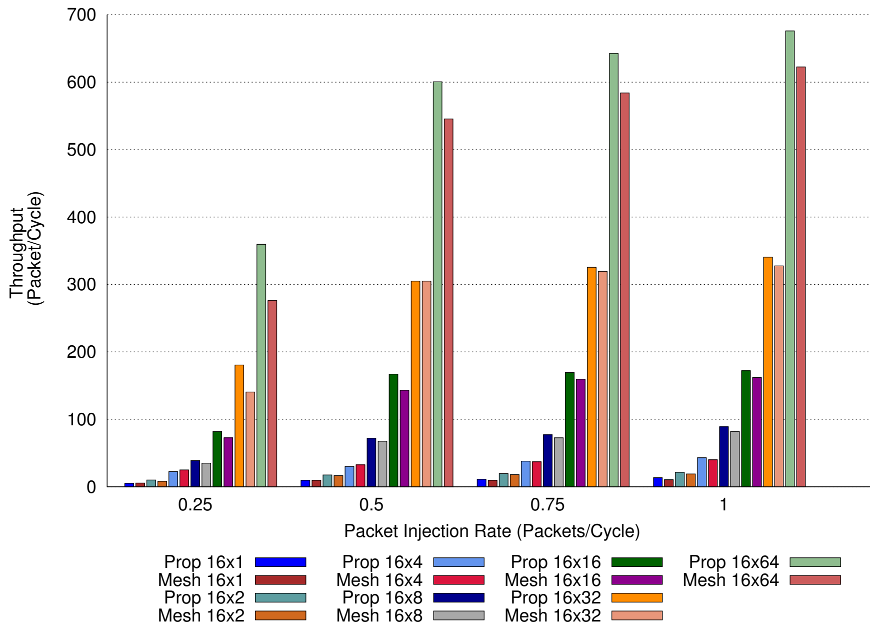


Figure 6.13: Average network throughput in transpose traffic pattern.

6.4.2.3 Network scalability

To better analyse the scalability of the proposed design, a set of experiments are further performed specifically aimed at evaluating the average packet latency and throughput with the increasing number of cores in the network. To this end, the four injection rates are averaged ($I_r = 0.625$ packets/cycle) for all the three traffic patterns. The results are shown in Figure 6.14. From the plot, it is evident that the proposed NoC architecture shows a significant reduction of the average latency up to 128 cores, compared to the 2D-mesh. Specifically, moving from a configuration with 16 PEs to the one with 128 ones, the latency increases from 65 to 100 cycles. Conversely, for the same two configurations, the 2D-mesh topology shows an increment of the latency from 72 to 156 cycles. This behaviour has been observed irrespective of the traffic pattern. Then, for the next two network configurations (i.e., 256 and 512 cores respectively) the latency increment exhibited by the proposed design is not linear (e.g., the increment from 128 to 256 PEs shows a lower slope of the curve), while in the 2D-mesh the latency increment still follows a linear trend. Finally, considering the largest configuration (i.e., 1024 cores) the trend is still not linear for the proposed design and the latency drops to 170 cycles (this trend is similar in all the three traffic patterns). The trend of average packet latency improvement is also analogous to the other traffic patterns. Conversely, 2D-mesh increases the latency up to 377 cycles. It is worth to note that, although the two architectures have similar behaviour when the number of the PEs increases, the average latency is always significantly lower attained by the proposed design. If this observation is combined with

the lower power consumption and resource utilisation, it can clearly be said that the proposed design scales more easily than conventional ones and it can be a good candidate for supporting next generation high-performance manycore accelerators. To further con-

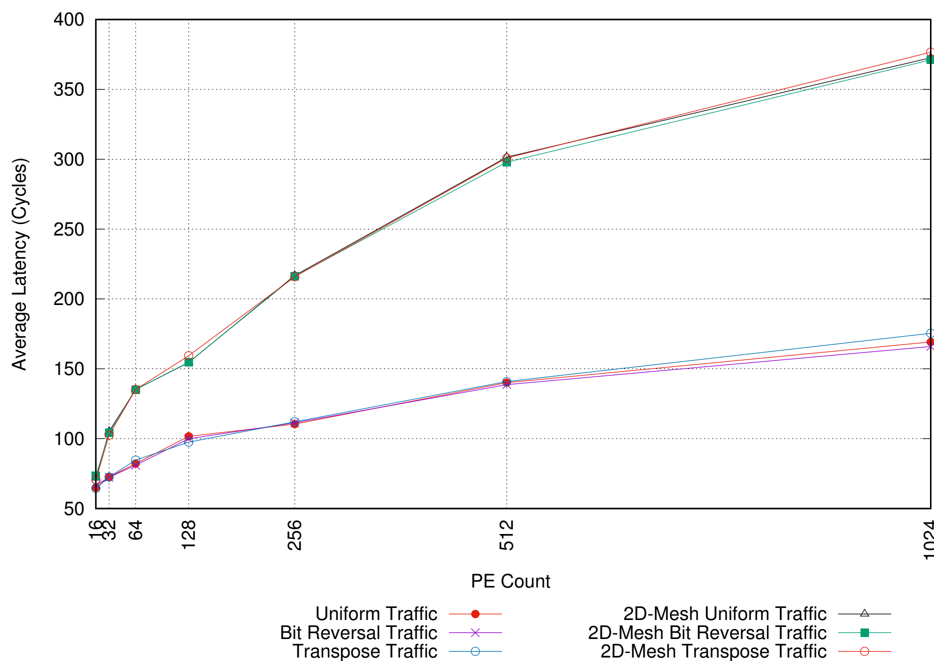


Figure 6.14: Average packet latency with increasing network size.

firm the scalability of the design, how the average network throughput improves with the growing network size (using the same packet injection rate for the three traffic patterns – $I_r = 0.625$ packets/cycle) is also analysed. The results of this experiment are shown in Figure 6.15. From the trend, it is evident that the average throughput tends to increase almost linearly by a factor $\approx 2\times$ when the number of PEs is doubled (considering all the three traffic patterns). For instance, considering the transpose traffic pattern, for a single block unit (i.e., 16 cores) the average throughput is 9.8 packets/-cycle while it increases to 17.13 packets/cycle for 32 PEs. Similar behavior is observed when moving from 128 cores (69.25 packets/cycle) to 256 cores (147.7 packets/cycle), as well as when moving towards the largest configuration, i.e., from 512 cores (288 packets/cycle) to 1024 cores (570 packets/cycle). Although the 2D-mesh topology shows similar behaviour for lower core counts, it is important to highlight that the average throughput is always lower than the proposed design, and quickly start to decrease when the core count increases (i.e., for more than 128 cores proposed ring-mesh combination outperform the 2D-mesh topology). Finally, how the network performs with increasing number of PEs is compared, by plotting the average throughput versus the number of processing cores and also the average throughput versus the average latency together. The result of this comparative analysis is reported in Figure 6.16. Here, all the reported values of

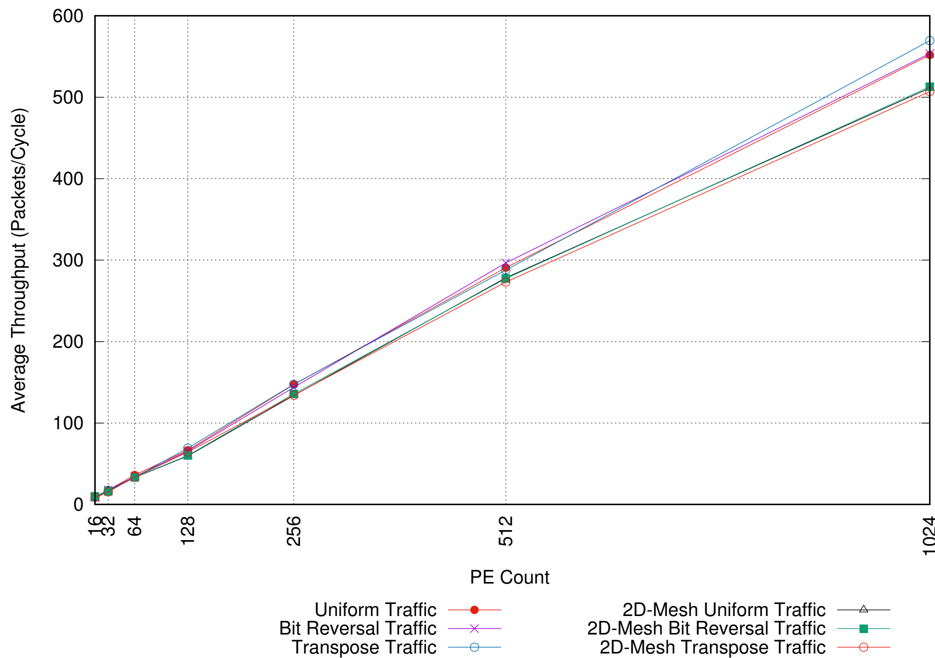


Figure 6.15: Average network throughput with increasing network size.

latency and throughput are the average of all the experiments. From the plot, it appears that the average latency grows by a factor of $1.25\times$ when moving from 256 cores to 512 cores while all the other cases a lower latency growth can be seen. The NoC design also registers the average throughput growth of a factor of $\approx 2\times$ for almost all cases. The proposed design has shown its robust performance by improving its throughput higher than latency, and even the latency growth is started to reduce with the increase of the network size.

6.5 Applicability and future improvements

In this section, the possible extensions of the basic working mechanism of the proposed network architecture are discussed which are aimed at better supporting multiple applications with dynamic resource requirements. For instance, application based on the MapReduce [DG08] programming model can vary the number of required PEs during their execution. In fact, in general, the number of PEs required by mapping functions is higher than of the number required by reducing functions. Similarly, applications build upon explicit dataflow programming models show the same behaviour: dataflow graphs representing the execution flow and the threads' dependencies can grow and shrink during the application lifetime. With the aim of supporting such applications, a set of features are provided which allow exploiting better the large number of PEs that the NoC design can support, as well as to provide resiliency of the system against failures.

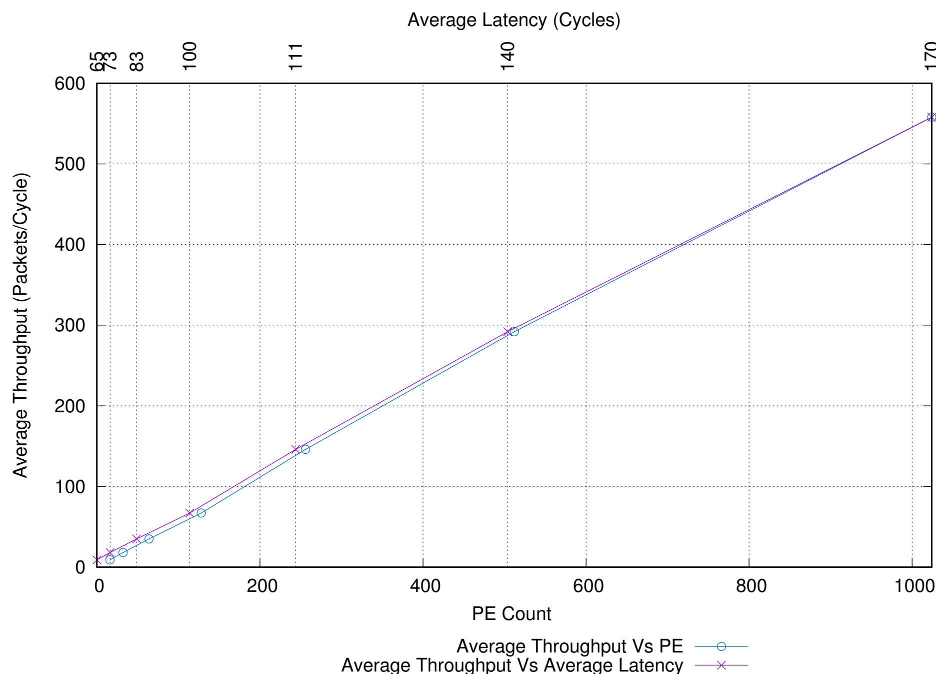


Figure 6.16: Comparing average network throughput and average packet latency with increasing network size.

Specifically, the active elements of the network hierarchy (i.e., mesh routers and RSs) is allowed to be selectively bypassed or completely switched off.

6.5.1 Morphing capability

Smart selection of specific topology to run applications can optimise cost and improve performance. The operating system (OS) can work in conjunction with the NoC support to exploit such feature to optimise the thread communication overheads. To this end, the routers and RSs can process a special configuration packet called *morph* packet, which specifies how to configure single links within the mesh router or the RSs. In HPC environments, hypervisor/OS is responsible for generating such control packets consulting the compiler (see for example [MDM04c]).

The morph packet is organised in such way the 32-bits of the payload are used to carry configuration information. Specifically, the payload is composed of four subfields: hierarchy level (HL=1-bit), execution region size (ERS=10-bits), link configuration (LC=16-bits), and PE type selector (PTS=5-bits). Figure 6.17 (left) shows the internal organisation of the morph packets.

- *Hierarchy level* (HL): Single bit allows distinguishing if the configuration must be applied to a RS (HL = 0) or to the mesh router (HL = 1).
- *Execution region size* (ERS): This field is formed by the next 10-bits in the payload.

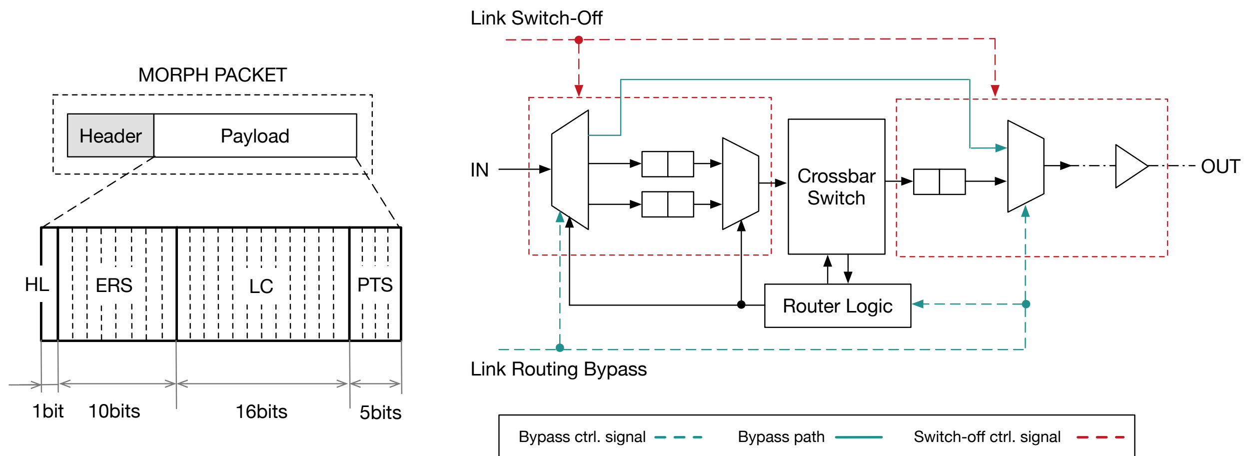


Figure 6.17: Internal organisation of the morph control packet (left), and the corresponding control structures in the mesh router (right).

With this length, it is possible for an application demanding for its execution a subset of the total core count, as well as the whole CMP computing resources.

- *Link configuration* (LC): This field can use up to 16-bits to specify how to configure single links. Each group of 2-bits in this field allows specifying the state of the corresponding link. In the case of the mesh router, links are eight in total (i.e., north, south, east and west for the 2D-mesh and four to connect to the ringlets). Conversely, a RS has four links at most. Thus only a reduced number of bits in this field are used. A group of 2-bits allows to set the link into three main states:
  - *Active*: Link is fully active, and the traffic normally flows inside the router/switch for routing decision.
  - *Bypass*: Link is configured in such way the incoming traffic is directly presented to the corresponding output port, moving in the same direction. For instance, bypassing the east channel of a mesh router allows to inject the traffic directly in west output channel.
  - *Switch-off*: Link is completely switched off, by disabling the logic governing it. The router/switch logic governing the other links is reconfigured accordingly.
- *PE type selector* (PTS): The remaining 5-bits can be used to target special resources in the chip (such as dedicated accelerating cores when a heterogeneous environment is used).

Morph packets can be generated at the OS/hypervisor level in such way they can be sent selectively to a subset of the routers or in a broadcast. However, the way control

information is organised in the payload, allows the routers and RS not involved in the configuration process to skip their processing, thus avoiding further throughput limitations. Starting from the requests of the application, routers can create *execution regions* for the each application (i.e., an execution region corresponds to the dynamic group of PEs required by the application). For instance, it is possible for a mesh router to dynamically restrict the execution of an application to two ringlets, and using the other two ringlets for the execution of another application (application awareness). The proposed morphing solution is flexible enough so that it can be exploited to tailor the NoC topology to the application requirements. Similarly to [SMP16], by selectively bypassing or disabling links, it is possible to allow the NoC to assume a special (virtualized) configuration that provides more performance for the specific application.

Figure 6.17 (right) shows the modification to the control signals to allow morphing capabilities. Blue dashed lines represents signals set to bypass crossbar switch and input/output channel buffers. Every time this signal is set, packets entering in the selected input channels are directly driven by the output links (blue line). Conversely, the switch off control signal (it is dominant over the bypass signal) completely disable input and output channel logic. Traffic entering in switched off channels is dropped. Similarly to mesh router, bypass and switch off logic directly operates on the multiplexers/demultiplexers governing the RSs (see Figure 6.4, section 6.3.2).

Morphing capabilities can also be used to improve further power saving (such as power gating technique based on a traffic activity threshold [PDB14]) and ensure system resiliency. Whenever any running application does not use a portion of the chip, they can be switched off. When an application requires more resources, switched off elements (e.g., a ringlet) can be enabled by resetting switch off control signals. The proposed extension can play an important role to ensure system resiliency to faults. By detecting faulty PEs, or a failure in the router/switch logic, the component can be easily bypassed.

Finally, the morphing capability is mainly aimed at improving the energy-cost by allowing multiple applications to share the spatial as well as computational resources of the NoC. The primary idea of generating the execution-region is to achieve better performance via traffic localisation by effectively controlling the routers or the ring-switches using the morph packets.

6.6 Summary

Following the current trend, there will be general purpose chips with hundreds of PEs. To efficiently use the massive built-in parallelism, the traffic inside the chip need to be managed efficiently, both from the power (reducing hotspots) and performance (lower latency and throughput) perspective. Thus, a simple yet scalable two-level hybrid hi-

erarchical interconnection is proposed where ring and a 2D-mesh topology are fused together without using any bridge router. The proposed NoC design is implemented on a high-end FPGA device. In experiments using multiple synthetic traffic patterns, it is shown that the design is scalable while keeping high performance regarding throughput and latency. This proposed topological design can also be customised using the special configuration packets to exploit chip resources better, depending on the specific application requirements. Finally, it is discussed that how the proposed design can be utilised for the applications whose requirements are dynamically changing over their execution lifetime. Experimental results also showed that the hierarchical organisation of the interconnect could easily outperform the capabilities of traditional 2D-mesh NoCs. After discussing the dataflow thread management and also the possible NoC platform, it is kind of interesting to verify the thread to core mapping quality by proposing an analytical model (chapter-7).

6.7 Acknowledgement

This chapter is an edited version of the submitted paper entitled **A High-Performance Interconnect for Future Scalable Manycore CMPs** by Somnath Mazumdar and Alberto Scionti at the Journal of Parallel and Distributed Computing, Elsevier.

7

Analytical Model

Modern computing chips are composed of multiple, simple, low-power processing cores. Increasing the number of processing cores in a single chip brings the opportunity to exploit the inherent massive level of thread parallelism and further improved performance. However, efficient allocation of applications (threads) to available cores is a complicated process. Failing to do so, the mapping can be the limiting factor for achieving better performance on a tiled chip-multiprocessor (CMP). In this chapter, we propose a mathematical formulation based on mixed integer linear program (MILP) to map application threads on cores at worst-case scenario by keeping into account the spatial topology of a two-dimensional mesh (2D-mesh) Networks-on-Chip (NoC). Our model allows evaluating in absolute term the performance of different mapping and routing algorithms. The proposed analytical model is general enough to consider a different optimising policy from energy to latency and a different number of memory controllers. In the experiments, we have shown that the proposed approach achieves a 40% reduction over the traditional zig-zag heuristic, therefore showing that there is a range for improving application threads mapping on cores.

The chapter is organised as follows: Section 7.2 formally introduces the core mapping problem while in the following section 7.3, we introduce the mathematical formulation based on MILP. In Section 7.4, we present and discuss the results of our thorough simulation campaign. Finally, Section 7.5 summarises the chapter.

7.1 Introduction

Recent computing platform paradigm has been shifted towards the communication-centric design which is optimised both for performance and power. Over the past few years, researchers from industry and academia started to develop chips containing 100+

cores. (e.g., PEZY-SC processor[1]) or 1000+ cores (e.g., Epiphany-V Chip[2]). Stacking many low powered processing cores together offers a great level of parallelism, but requires to solve multiple execution management issues. In fact for chips with large core count, there could be higher chances of resource contention due to a significant amount of data exchange between the applications.

On-chip packet-switched micro-network of interconnects (i.e., NoCs [DT04b]) provides the physical substrate used by processing cores to communicate each other (also to the memory). In general, NoC's architectural components (such as channel width, buffer size, thread-to-core mapping, and routing algorithms) are very critical for better support for data traffic [BM06b]. There is a need to optimise latency and energy cost for the data exchange inside the NoC to achieve higher performance. An increasing number of core count and communications between threads inside the NoC may yield to hotspot issues. Hence, on-chip communication may become a barrier for higher performance. The main design goals for NoC based interconnects are higher bandwidth, low energy cost, and low latency. However, NoC designs with several hundreds of processing cores inside the chip suffer from large energy consumption. In fact, the on-chip network can consume a significant fraction of the whole chip power budget. For instance, experiments [HPD12] have shown that for large core count (i.e., a 256-core based CMP) conventional 2D-mesh NoC consume up to 45% of the total energy; while the NoC for the MIT Raw processor [WPM03] can consume up to 36% of total system power. In another work, Vangal et al. [VHR+08b] showed that on the Intel TeraFLOPS chip of 80 cores, the NoC uses up to 28% of tile power.

Thread-to-core mapping problem (T2CMP) in NoC is a problem instance to assign given applications' threads on the available processing cores to optimise user-given performance metrics (such as energy cost, latency, throughput). In general, NoC based research works are mainly focused on the reduction of either overall energy consumption [HM05, SK04] or communication cost [SBG+08, PBB+03] by placing applications threads on cores using various complex methodologies. At execution time, application threads require data exchange with other running in other cores or with primary memory (needing to fetch or to write the data). Clearly, threads with a large amount of data packet exchange (or high memory requirements) should be placed as close as possible to improve the performance and to reduce energy consumption [CLK07]. The presence and positioning of multiple memory controllers (MCs) should be considered since they can manage different memory modules/banks to serve the requests from cores.

In this chapter, we propose a mixed-integer linear program (MILP) based solution

[1]http://pezy.co.jp/en/products/pezy-sc.html

[2]https://www.parallella.org/wp-content/uploads/2017/01/million_cores.pdf

for thread-to-core (T2C) mapping so that the overall energy consumption can be reduced (in worst case traffic scenario). Our analytical model can place the application threads on the available core to optimise *(i)* energy consumption or *(ii)* overall average latency of data packets. The main contributions of our work are:

- We propose an MILP model to map application threads on available cores for optimising the power cost and latency.
- Our model supports near data processing (NDP) which is unique to our model, and we also show how changing the number of memory controllers affect the latency and energy consumption inside the chip.
- We provide worst case QoS regarding energy cost, latency.
- Finally, we compare our proposed model with the existing approaches such as zigzag heuristic and show our model achieves 13% reduction in the average energy consumption and 27% reduction in the average packet latency.

7.2 Problem description and assumptions

The main idea of increasing core count together with robust NoC architecture is mainly to host multiple application on a single chip to increase the overall system utilisation. Usually, threads communicate with other threads or other resource components (such as local memory banks, last-level cache (LLC), DRAM controllers). A good criterion is to place communicating threads close to each other to minimise latency and energy consumption. However, after threads terminate their execution, the empty cores tends to be scattered around the NoC floor (see Figure 7.1). This situation can force to assign the application threads to a distant core which may cause a significant performance drop due to long communication distances (both increasing the power consumption and the risk of the traffic contentions [JP14, AAS13]). Hence, an efficient topological mapping of cores onto the NoC can save communication energy [HM05] and also bandwidth [MDM04a]. Mapping of application threads on processing cores in NoC is one of the most important issues in NoC design. In this section, we introduce the graph based models to represent the network topology, the applications and the snapshot of the current status in the proposed model. The 2D-mesh topology based NoC is composed of several processing cores or *tiles* arranged on a $X \times Y$ grid. Here, X and Y denote the total number of tiles on the x-axis and y-axis, respectively. Each tile contains a *router*, and it is connected to a local core and also to the adjacent NoC tiles via network links. The NoC routers are in charge of routing the data packets to their corresponding destination, while the round-robin arbitration allows modelling the separate queues as a single incoming queue for

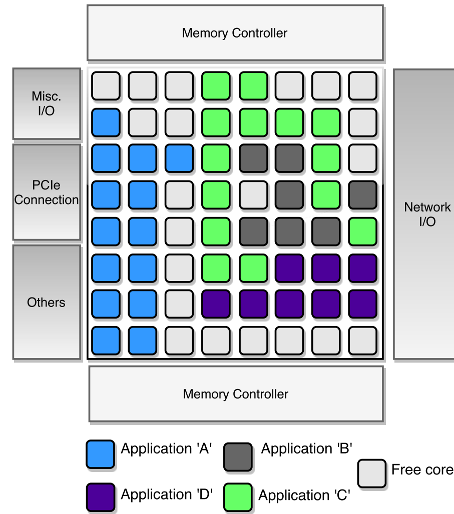


Figure 7.1: Representation of multiple application running on a NoC

data packets. As represented in Figure 7.2 (a) the router contains the input ports (IPort) with *input buffers*. Each input ports connects to a *crossbar switch* and then connect to any output port via the crossbar switch. The input buffers consist of multiple virtual channels (VCs) (for simplicity we have shown two for each port).

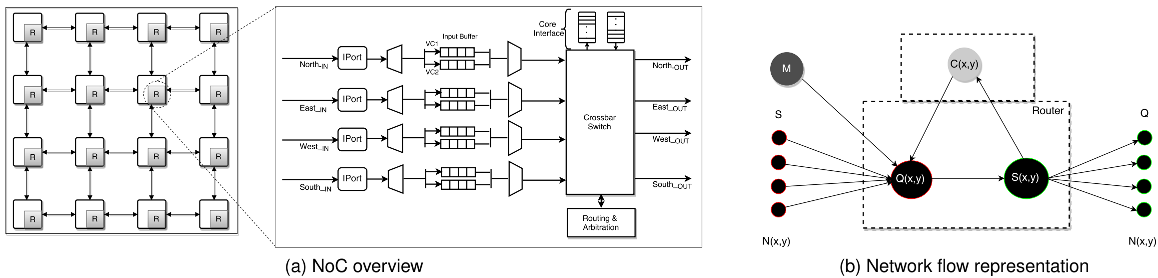


Figure 7.2: NoC model representation: via architectural model (left) and also via theoretical model (right)

Given a router $i = (x, y)$, let $\mathcal{N}(i)$ be the set of adjacent (directly linked) routers. In a 2D-mesh layout $\mathcal{N}(i)$ can be denoted as $\mathcal{N}((x, y)) = \{(x+1, y, (x-1, y), (x, y+1), (x, y-1)\}$. Further, the memory controllers (MCs) are connected to a subset of NoC routers ($\mathcal{N}(M)$). Now, we describe how these elements can be modelled using a graph $\mathcal{G}(N)$. Let $\mathcal{C}$, $\mathcal{Q}$, $\mathcal{S}$ be the sets of nodes in a graph representing cores, queues (input buffers) and crossbar switches, respectively. Figure 7.2 (b) is the resulting network flow representation.

- We assume each core can execute at most a single thread. The traffic data generated by the core goes through the buffer. While the incoming traffic directed

towards the core is injected from the crossbar switch. In $\mathcal{G}(N)$ the core $c = (x, y)$ is represented by a node $C(x, y) \in \mathcal{C}$.

- The input buffer is essentially a queue to hold the traffic coming from all directions for further transmission. The queue is modelled as a node $Q(x, y) \in \mathcal{Q}$. It temporarily stores local traffic coming from the core $C(x, y)$ and from the adjacent routers $\mathcal{N}((x, y))$. All the outward traffic traverses through the switch $S(x, y)$.
- Next, the switch is modelled as a node $S(x, y) \in \mathcal{S}$ and it is connected to the corresponding (local) core $C(x, y)$ and also to the buffers of the adjacent routers $\mathcal{N}((x, y))$ via the links.
- The MC is represented by a single node M in $\mathcal{G}(N)$ which is connected to the queue of the routers in $\mathcal{N}(M)$. In the case of multiple MCs, the node M is connected to all the routers in $\mathcal{N}(M)$.

The graph representation $\mathcal{G}^N$ of the overall NoC topology is obtained by the opportune replication of the above described graph. In particular, the nodes of graph $\mathcal{G}^N$ are $\{\mathcal{C} \cup \mathcal{Q} \cup \mathcal{S} \cup \{M\}\}$.

We represent an application as a weighted directed acyclic graph (DAG) $\mathcal{G}^A = (\mathcal{A}, \mathcal{T})$. Where each node $a \in \mathcal{A}$ is an application thread, and each weighted and directed arc $t \in \mathcal{T}$ is associated to a triple (s^t, d^t, p^t), representing the source thread s^t, the destination thread d^t and the weight of the arc p^t. In particular, the weight p represents the number of data packets that would be exchanged between the threads. In detail, $\mathcal{G}_i^A$ representing an application i that is generated as follows: Let a_i and t_i be the number of threads (nodes) and data exchanges (directed arcs) in the application i, t_i random arcs are added to $\mathcal{G}_i^A$ with weight (threads' data exchange) p obtained from a given distribution D_{CPU}. Moreover, each node a_i of the graph has an additional arc from the MC M to a_i with weight extracted from a given distribution D_{RAM}. An example of $\mathcal{G}^A$ with $a = 5$ and $t = 6$ is shown in Figure 7.3. It is worth to note that in our model the packet generation by a thread follows *worst case* situation, that is all the traffic to be sent between two threads sent at time zero.

We introduce an assignment graph $\mathcal{G}^S$ to represent the snapshot of the current state of the NoC. The assignment graph $\mathcal{G}^S$ is a bipartite graph $\mathcal{G}^S = (\{\mathcal{A} \cup \mathcal{C}\}, E)$ representing ready threads a to be assigned to available cores c. More specifically, two disjoint sets of nodes $\mathcal{A}$ and $\mathcal{C}$ are ready threads to be executed and the cores, respectively. Finally, the edge set $E = (a, c)$ represents all the possible threads to core assignments. Assigning threads to cores correspond to find a *perfect matching* $\mathcal{G}^M = (a, c)$ on $\mathcal{G}^S$. In other words, every node in $\mathcal{G}^S$ has to be assigned to exactly one edge in $\mathcal{G}^M$. However, over the time two events may happen due to the dynamic nature of workload:

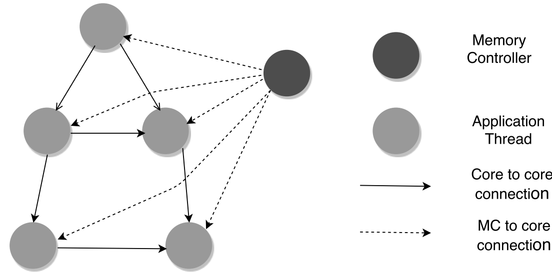


Figure 7.3: Basic system graph representation including application threads and associated single memory controller

- A new application is allocated, and a set of arcs representing the assignment is added to $\mathcal{G}^M$.
- A thread n terminates its computation releasing its core c and its corresponding edge (n, c) as well.

An instance of the T2CMP is therefore given by a (NoC) graph $\mathcal{G}^N$ representing the NoC, a current system status at time τ $\mathcal{G}^M_\tau$ and a new incoming application $\mathcal{G}^A$ (that needs to be allocated). Thus solving the T2CMP corresponds to find a new matching $\mathcal{G}^M_{\tau+1}$ in which $\mathcal{G}^A$ has been added to $\mathcal{G}^M_\tau$ minimising a given objective function (energy or latency). Our proposed model for solving this instance of T2CMP adopts the following assumptions:

- Given the current snapshot $\mathcal{G}^M_\tau$ of the NoC at time τ, we map an application $\mathcal{G}^A_i$ to it.
- Graph $\mathcal{G}^N$ represents the 2D-mesh NoC topology and 2D-mesh with *bi-directional links*.
- Messages are composed of a variable number of packets with equal size (granularity). We have considered single-flit packets since they represent a large segment of the network traffic for real applications [MJW12].
- We use store-and-forward flow control mechanism that allows transferring a single data packet in a single time unit (cycle), and the network link capacity is not a resource constraint.
- Routers have infinite input queue per VC while the sources (core) have infinite buffers, and sinks (core) immediately consume packets arriving at their respective destinations. Hence no packet can be lost during the transmission and also no packets would be blocked when some of the input buffers are full.

- We assume that *(i)* all the packets are generated/injected at time zero (to represent worst case scenario), *(ii)* we do not capture the cost of having a thread blocked and waiting for data to be transferred from another thread. We also do not consider the time explicitly.

7.3 Mathematical formulation

In this section, we propose a mathematical formulation for the T2CMP. The decision variables of the model are of two types: A family of *binary assignment variables* and several families of *continuous flow variables*. More specifically:

- $x_c^a \in \{0,1\} \;\; \forall c \in \mathcal{C}, a \in \mathcal{A}$, where x_i^a is 1 if thread a is assigned to core i.
- $CQ_c^t \;\; \forall c \in \mathcal{C}, t \in \mathcal{T}$ is the flow of traffic t from Core to Queue of the tile i.
- $MQ_c^t \;\; \forall c \in \mathcal{C}, t \in \mathcal{T}$ is the flow of traffic t from MC to Queue of the tile i.
- $RC_c^t \;\; \forall c \in \mathcal{C}, t \in \mathcal{T}$ is the flow of traffic t from Router to Core of the tile i.
- $QR_c^t \;\; \forall c \in \mathcal{C}, t \in \mathcal{T}$ is the flow of traffic t from Queue to Router of the tile i.
- $RQ_{sd}^t \;\; \forall s \in \mathcal{C}, d \in \mathcal{N}(s), t \in \mathcal{T}$ is the flow of traffic t from Router of the tile s to Queue of adjacent tile d.

Given these decision variables the resulting mathematical formulation is:

$$\min \quad \frac{\sum_{t \in \mathcal{T}} \sum_{q \in \mathcal{Q}} (E_R \cdot QR_q^t + E_L \cdot RQ_q^t)}{\sum_t p^t} \tag{7.1}$$

$$\sum_{i \in \mathcal{C}} x_c^a = 1 \quad \forall a \in \mathcal{A} \tag{7.2}$$

$$\sum_{a \in \mathcal{A}} x_c^a \leq 1 \quad \forall c \in \mathcal{C} \tag{7.3}$$

$$CQ_c^t = p^t x_c^{s^t} \quad \forall c \in \mathcal{C}, t \in \mathcal{T} \tag{7.4}$$

$$RC_c^t = p^t x_c^{d^t} \quad \forall c \in \mathcal{C}, t \in \mathcal{T} \tag{7.5}$$

$$\sum_{c \in \mathcal{N}(M)} MQ_c^t = p^t \quad t \in \mathcal{M} \tag{7.6}$$

$$\sum_{s \in \mathcal{N}(d)} RQ_{sd}^t + CQ_s^t = QR_s^t \quad s \in \mathcal{C} \setminus \mathcal{N}(M), t \in \mathcal{T} \tag{7.7}$$

$$\sum_{s \in \mathcal{N}(d)} RQ_{sd}^t + CQ_s^t + MQ_s^t = QR_s^t \quad s \in \mathcal{N}(M), t \in \mathcal{T} \tag{7.8}$$

$$\sum_{d \in \mathcal{N}(s)} RQ_d^t + RC_s^t = QR_s^t \quad s \in \mathcal{C}, t \in \mathcal{T} \tag{7.9}$$

$$x_c^a \in \{0,1\} \quad \forall a \in \mathcal{A}, c \in \mathcal{C} \tag{7.10}$$

$$CQ_c^t, QR_c^t, RC_c^t, MQ_c^t \geq 0 \quad \forall t \in \mathcal{T}, c \in \mathcal{C} \tag{7.11}$$

$$RQ_{sd}^t \geq 0 \quad \forall t \in \mathcal{T}, s, d \in \mathcal{C} \tag{7.12}$$

The objective function (Equation 7.1) aims at minimising the average energy consumption of the packets. The energy consumption is computed by summing the contribution of the amount of energy required to pass through a router E_R and a link E_L, respectively. In our evaluation, $E_R = 0.9776$ pJ/bit and the amount of energy required for a single bit to cross a 1mm link is $E_L = 0.51$ pJ/bit (as reported in [WSK+05]).

Constraints (7.2) states that all threads have to be assigned exactly to a single core. Whereas constraint (7.3) ensures that each core hosts at most one single application thread. Constraints (7.4–7.5) enforce the flow entering and exiting from the NoC graph's source and sink nodes. More specifically, flow is allowed to enter the network from core c only if thread a is a source of traffic (with $a = s^t$) and it is assigned to core c. The resulting amount of flow circulating is p^t. Similarly, flow is allowed to exit from the NoC graph, if thread a is a sink of traffic ($a = d^t$) and it is assigned to core i.

The flow balance on Queue and Router of each core i is regulated by constraints (7.7–7.8) and (7.9), respectively. Note that, the flow balance for the Queue nodes has to be formulated separately for nodes directly connected with MC and for nodes without direct communication. The flow balance for the MC is given by constraint (7.6) in which the amount of flow of traffic $t \in \mathcal{M}$ (p^t) has to exit from node M and flow only to the nodes connected with a MC. Finally, constraints (7.10–7.12) define the variables of the problem.

The same model can be easily adapted to consider the case in which some threads are preassigned to cores, and such decisions cannot be undone. Let introduce the set $\mathcal{P}$, containing the information on the preassigned cores, i.e., $(a, c) \in \mathcal{P}$ if thread a is preassigned to core c. Such *forcing constraints* are easily modelled by adding the following constraint which ensures that the thread to core assignment is not changed:

$$x_c^a = 1 \quad \forall (a, c) \in \mathcal{P} \tag{7.13}$$

7.3.1 Alternative objective function

An alternative objective function aiming at minimising the average latency of each packet has also been developed and tested.

$$\min \frac{\sum_{q \in \mathcal{Q}} (D_q(\sum_{t \in \mathcal{T}} QR_q^t))}{\sum_t p^t} \tag{7.14}$$

The latency objective function (Equation 7.14) substitutes Equation 7.1 and represents the average waiting time for each packet. Where D_q is a *stepwise linear monotonically increasing function* with breakpoints at each unit used to represent the total waiting time (number of cycles needed to route all the packets) of the queue q. We assume that *(i)* all the data packets are available to be sent at time τ_0, and *(ii)* a switch is able to route a packet every cycle (recall Section 7.2). Thus the nth packet in a queue

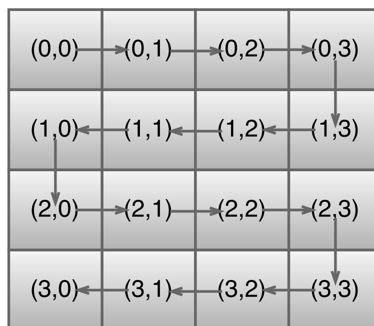


Figure 7.4: T2C mapping process in zig-zag heuristic algorithm for NoC size 4×4

has to wait n cycles, therefore the total waiting time for all the b packets in the queue is $D_q(b) = \frac{b(b-1)}{2}$. Observe that, due to the presence of the stepwise objective function, this alternative formulation results in a structurally more complex problem to solve.

7.3.2 Heuristic algorithms

In our comparison, we have used *zig-zag* (ZZ) (see figure 7.4) way of T2C mapping. A zig-zag is a symmetrical pattern and also provides regularity during the mapping process. We also have used *random* assignment (RND) of T2C mapping for experimental comparison. RND simply assign a thread to a randomly selected empty core. Both ZZ and RND use XY routing algorithm. The XY routing algorithm selects the links that a packet must follow to reach its destination from the source. For routing the packets, we have used the deterministic XY dimension order routing (XY-DoR) algorithm. Here, the same set of paths is always selected for the same set of source and destination core. In XY-DoR routing scheme, all the packets always traverse first in the X direction (i.e., east or west) and then turns towards Y direction (i.e., north or south).

7.4 Simulation results

In this section, we first describe the test instance generation procedure, and we also describe the proposed algorithms. A detailed report on our results and a discussion on the practical applicability of the proposed approach follow in the next subsections.

7.4.1 Simulation environment

Recall that, a T2CMP instance is defined by a graph $\mathcal{G}^N$ representing the NoC, a current system status at time τ $\mathcal{G}^M_\tau$ and an application $\mathcal{G}^A$ be allocated. The computational results reported in this chapter are based on different sets of carefully generated instances

Table 7.1: Experiment configuration

Parameters	Values
Number of rows	4,8
Number of columns	4,8,16
Max queue length	Unbounded
Memory controller	1,2 sides
Packet inter-arrival time	Worst-case

considering various system related factors. We consider four network sizes, each NoC instance having number $S \in \{4 \times 4, 4 \times 8, 8 \times 8, 8 \times 16\}$ of processing cores. For each size, we considered both the cases of a single MC placed on a side and two MCs placed on two opposite sides (see Figure 7.1). Packet injection process has been considered as the worst case scenario (bursty traffic, e.g., (100 packets/1 cycle) while stable flow is (100 packets/100 cycles) similar to [MBDM05]). In Table 7.1, we report the configurations used during the simulation.

We resorted to a complex application generation strategy to generate a NoC snapshot $\mathcal{G}_\tau^M$ able to replicate time varying characteristics of T2CMP. First, starting from an empty NoC, we generate new applications to map up to a given threshold α_1. Next, these applications are allocated using the ZZ algorithm. Finally, to simulate the changes over time, each running thread has a given probability (α_2) to terminate its computation (i.e., to be removed) and thus emptying the corresponding core. Once the NoC snapshot $\mathcal{G}_\tau^M$ has been generated, additional applications are generated up to a given threshold (α_3). These new applications are the one that is to be assigned to solve the T2CMP.

We consider application i having a_i threads (nodes in the graph), with a_i randomly extracted from $[2, 8]$ (for the 4×4 NoC instances) up to $[24, 144]$ (for the 8×16 instances). While the number of arcs ranges from $[1.5 \cdot a_i, 3.0 \cdot a_i]$. The flow between threads ranges in $[40, 100]$ packets, whereas the flow from RAM is in $[10, 40]$.

We used four different algorithms during the simulation. They are:

- MILP$_0$: We use the proposed formulation to simultaneously map all the threads in the snapshot and the new incoming application. The optimal solution of this model gives the theoretical best map value that could be achieved. However, since thread migration is not allowed these results may be unattainable in practice.

- MILP: The snapshot is assigned with the ZZ heuristic, whereas the new incoming applications are mapped using the MILP formulation. The optimal solutions provided by this algorithm represents the best possible mapping for the new incoming application when thread migrations and consolidations are not allowed.

- ZZ: First the snapshot is assigned with the ZZ heuristic and next the new incom-

ing application is mapped using the ZZ again. This algorithm provides a quite standard mapping solution.

- RND: The snapshot is assigned with the ZZ heuristic, whereas the new incoming threads are randomly mapped to empty cores. This algorithm is included for the experimental comparison only.

To evaluate our proposed model, we generated ten instances for each of four NoC sizes and each of two MC types for a total of 80 instances. The four proposed algorithms have solved each instance for the two considered objective functions (minimisation of energy cost and minimization of latency). In total, we carried out 640 runs. All tests have been conducted on a Linux machine with Intel Xeon E5-2440 (@1.90GHz clock speed), eight cores and 16 GB of RAM. For executing our MILP models, we have used Gurobi (version 7.0.0) [Gur16] with standard settings and with a three-hour time limit. All the models and algorithms are implemented in Python (version 2.7). All the results reported in each row of the following tables are averaged over ten runs.

7.4.2 Results

We now report the results of the computational campaign. We first discuss the results of the more theoretical model ($MILP_0$), next we compare the behaviour of the proposed algorithms, and then we discuss the influence of the different parameters used in the generation of the instances (network size, memory controllers and objective functions). Finally, we discuss with some additional tests the effects of the NoC layout.

In Table 7.2, we report the dimensions of the MILP models used to formulate the instances. In the first column, we report the NoC size, and then we report the size of the model regarding constraints and number of variables. Each row of the table shows the average results over ten instances with the same NoC size. Note that the problem size it is only minimally affected by changes in the objective functions and MCs. The resulting formulations are very large, especially for the 8×16 cores, with more than 200000 variables and 100000 constraints.

Table 7.2: Problem size for the $MILP_0$ model

Network Size	**Constraints**	**Variables**
4x4	1515.6	2718.6
4x8	8170	15029
8x8	30738.2	58461
8x16	106859	206633

7.4.2.1 MILP solution quality

In this sub-subsection, we present the results of the proposed MILP. Here, in Tables 7.3 and 7.4, we first show the absolute performance of $MILP_0$. Next, we report the results of the MILP algorithm and a comparison between the two formulations. The first column shows the NoC size and results of the algorithm are reported in the remaining three columns. First, we report the percentage Gap from optimality, then the number of failures and finally the total execution time for the runs (expressed in seconds, the execution time was fixed to three hours). From Table 7.3, we can see that $MILP_0$ can solve at optimality all the smallest scenarios (4×4). When switching from one to two MCs, we do not observe great changes either regarding solution quality or regarding execution times. However, for bigger sizes, it appears that the presence of an additional MC causes slightly wider gaps. In all cases, the solver is always able to produce a feasible solution (no Fails are reported) although the distance from the lower bound can be as high as 80%.

Table 7.3: Performance of $MILP_0$ model while the objective function is minimisation of energy cost

Network Size	**Gap(%)**	**#Fails**	**Exe. Time (in secs.)**
Memory Controller=1			
4x4	0.00	0	122.04
4x8	14.82	0	9913.32
8x8	61.26	0	10800.00
8x16	76.53	0	10800.00
Memory Controller=2			
4x4	0.00	0	91.75
4x8	16.65	0	10026.73
8x8	64.47	0	10800.00
8x16	79.15	0	10800.00

While considering the latency objective function (Table 7.4), we observe a similar behaviour, and it is evident that these models are comparatively harder to solve. In fact, we both observe an increase in running times for small instances (from one minute to about ten minutes), and for the bigger instances, we observe increased optimality gaps (and even some failures). In particular, for the 8×16 CPU instances in some cases, the solver does not manage to produce a feasible solution within the three-hour time limit.

Table 7.4: Performance of $MILP_0$ model while the objective function is minimisation of latency

Network Size	Gap(%)	#Fails	Exe. Time (in secs.)
Memory Controller=1			
4x4	0.00	0	640.64
4x8	74.10	0	10800.00
8x8	90.55	0	10800.00
8x16	96.55	7/10	10800.00
Memory Controller=2			
4x4	0.00	0	626.11
4x8	72.19	0	10800.00
8x8	94.73	0	10800.00
8x16	97.51	2	10800.00

We now discuss the results of the MILP algorithm. Recall that the pure $MILP_0$ model is a relaxation of the T2CMP since it is allowed to assign even previously assigned threads freely. In other words, it solves a larger and less constrained problem in which no thread is preassigned, and they all can be freely moved around. Hence, the results produced by the $MILP_0$ may not be allowed in practice. On the other hand, the MILP model assigns only the new incoming threads without changes on the running threads preassigned using the ZZ heuristic.

Table 7.5: Performance of MILP model while the objective function is minimisation of energy

Network Size	Gap (%)	Absolute Gap (%)	#Fails	Exe. Time (in secs.)
Memory Controller=1				
4x4	0	0	0	3.21
4x8	0	0	0	3453.32
8x8	24.61	65.47	0	10800
8x16	42.16	142.11	0	10800
Memory Controller=2				
4x4	0	0	0	3.78
4x8	0.24	0.71	0	3532.46
8x8	26.67	73.17	0	10800
8x16	46.71	170.2	0	10800

In Table 7.5 and 7.6, we presents the results of the MILP. Observe that, the additional column "Absolute Gap" computes the gap between the lower bound (LB) provided

Table 7.6: Performance of MILP model while the objective function is minimisation of latency

Network Size	Gap (%)	Absolute Gap (%)	#Fails	Exe. Time (in secs.)
Memory Controller=1				
4x4	0.00	0.00	0	43.85
4x8	25.63	54.74	0	8899.07
8x8	59.44	208.18	0	10800.00
8x16	72.78	189.35	03(10)	10800.00
Memory Controller=2				
4x4	0.00	0.00	0	44.68
4x8	28.72	65.12	0	8966.85
8x8	66.46	277.03	0	10800.00
8x16	79.43	355.60	1(10)	10800.00

by the $MILP_0$ and the UB found by the MILP. In Table 7.5 we show the performance of the MILP algorithm. It can be seen that instances up to 4×8 can be solved at optimality. We found that the presence of two MCs consistently results in harder instances to solve. When considering the absolute gap, we see how, for larger instances, the distance from the LB computed by the $MILP_0$ can be far. When comparing MILP against $MILP_0$ (Tables 7.3-7.4), we observe how the MILP results in a problem easier to solve (both the Gap values and the computation times are smaller). When considering the minimisation of the latency (Table 7.6) we again observe how such models are harder to solve than their energy counterparts.

7.4.2.2 Comparisons of the proposed algorithms

We now discuss the performance of the MILP algorithm when compared to the other heuristic algorithms (ZZ and RND) as reported in Tables 7.7 and 7.8. Since the $MILP_0$ solver provides absolute LB values on the objective function, we can assess the comparison in absolute terms. As expected, the heuristic algorithms (ZZ and RND) provides worse solutions when compared to the MILP. However, the computation time for these algorithms is negligible. Note that, the gap between the algorithms reduces as the instance sizes increases. This behaviour highlights that larger instances are indeed harder to solve for the MILP.

Table 7.8: Gap from the LB(MILP) which is the best attainable for the three algorithms (MILP, ZZ, RND) while minimising the latency

Network Size	**MILP**	**ZZ**	**RND**
Memory Controller = 1			
4x4	0.00	77.37	86.36
4x8	46.13	237.14	229.54
8x8	198.08	384.85	331.57
8x16	342.52	385.07	376.32
Memory Controller = 2			
4x4	0.00	57.02	66.94
4x8	54.11	222.84	205.56
8x8	275.96	474.57	421.75
8x16	474.85	486.27	411.07

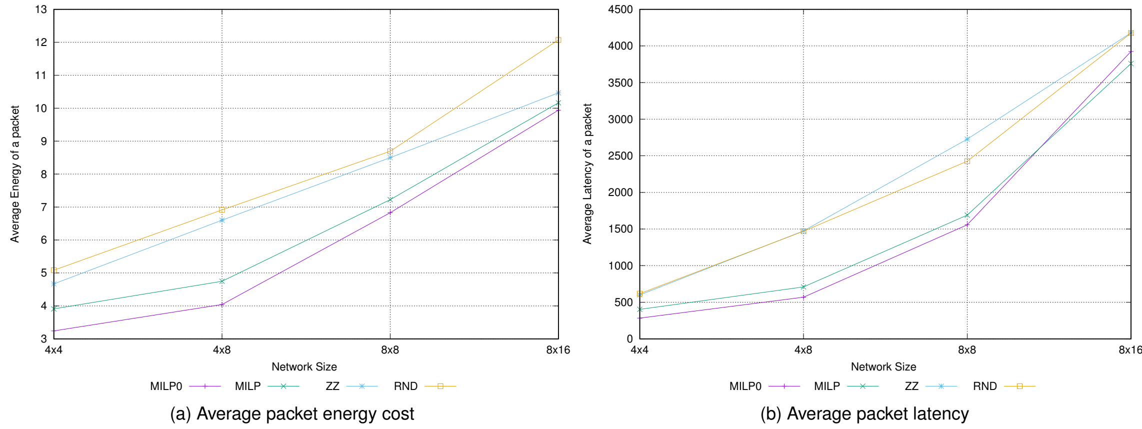
Table 7.7: Gap from the LB(MILP) which is the best attainable for the three algorithms (MILP, ZZ, RND) while minimising the energy cost

Network Size	**MILP**	**ZZ**	**RND**
Memory Controller = 1			
4x4	0.00	22.39	34.75
4x8	0.00	45.08	58.45
8x8	39.16	76.09	83.23
8x16	87.06	104.97	141.88
Memory Controller = 2			
4x4	0.00	21.44	34.18
4x8	0.25	42.95	46.15
8x8	46.15	85.52	90.95
8x16	106.60	117.79	152.87

From the results in Tables 7.7 and 7.8 it can be observed that overall the MILP approach can attain a reduction of 40% over the gap of provided by the ZZ heuristic.

7.4.2.3 Influence of network sizes, memory controllers, objective functions

We now highlight the influences of the different factors used in the instance generation procedures. In Figure 7.5, we report the energy consumption and packet latency for different network sizes, respectively. We can observe how, for small sizes (4×4 and 4×8) the and for both energy and latency, the performance of the exact algorithms (MILP_0 and MILP) exhibits a smaller growth compared to the heuristics. This behaviour shows



(a) Average packet energy cost

(b) Average packet latency

Figure 7.5: Relative average traffic performance

the benefits of using advanced optimisation and also that there is a substantial margin for improvement over the simple heuristics. However, given the NP-hardness of the problem, as the size of the instances grow more (8×8 and 8×16) the gap between the formulations and the heuristics gets smaller showing that the exact approaches require even higher computing times to be able to deliver good results. This behaviour is even more evident for the latency formulation which is harder to solve.

In Figure 7.6 we show the influence of the MCs representing the percentage of objective function reductions for the different algorithms. As expected the use of two MC contributes to reducing both the energy and latency. More specifically, the energy improvements range from less than 4% to more than 10%, with smaller improvements for the MILP formulations. Whereas, for the latency objective the reduction is over 20% for all four algorithms. It is expected since MILP algorithms produce optimised thread assignments and this means that the best possible solution can improve up to a 5%. On the other hand, ZZ and RND since they produce less optimised solutions have greater benefits from the presence of a second MC. In Figure 7.7, we show how these algorithms compare to each other when considering both the objective functions at the same time. On the two axes of the plots, we report the attained objective function for both values, and the performance of each algorithm are reported as a point on the plane. Observe that an algorithm is *dominated* regarding performance if there exists another algorithm with lower energy consumption and latency (i.e., placed down left in the plot). For each algorithm, we report two data points, one obtained when minimising the average energy consumption objective function and the second obtained when minimising the average latency. We expect these two data points not to dominate each other. In other words, given an algorithm, we expect the latency attained while minimising the latency to be smaller than the latency attained while minimising the energy consumption. Similarly, we expect a greater energy consumption when minimising latency than when minimis-

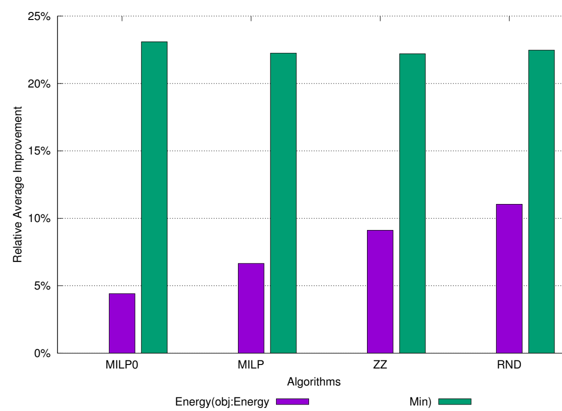
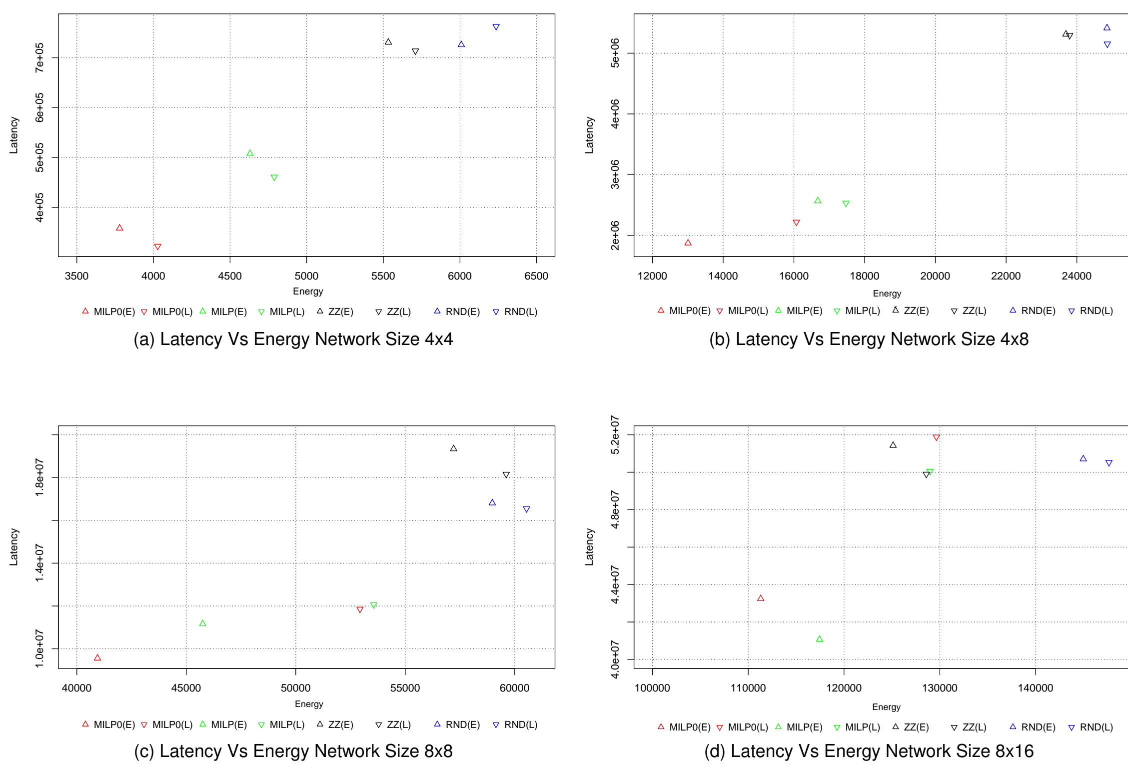


Figure 7.6: MC influence on four algorithms while objective function is energy cost and latency

ing energy consumption itself.



(a) Latency Vs Energy Network Size 4x4

(b) Latency Vs Energy Network Size 4x8

(c) Latency Vs Energy Network Size 8x8

(d) Latency Vs Energy Network Size 8x16

Figure 7.7: Comparing latency with energy cost with different network sizes

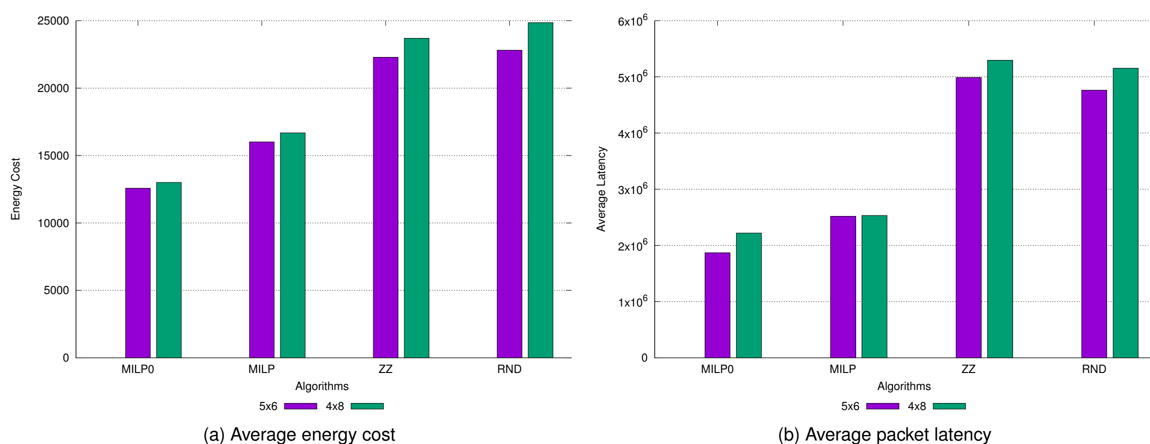
For small size problems (i.e., 4×4 NoCs) this is what actually happens (see Figure 7.7 (a)). The two data points of each algorithm are not dominated each other. Moreover, when considering the different algorithms we observe (as expected) the MILP_0 to dominate MILP which in turn dominates both the ZZ and the RND. Only in RND, which is the worst performing algorithm, the RND(E) dominates RND(L). The same behaviour

is also observed for Figure 7.7 (b) but we observe as the $MILP_0$ when minimising the latency is dominated by $MILP_0$(E), and as the distance between $MILP_0$(L) and MILP(L) is much reduced. This shows how already for 4×8 sizes the latency formulation is much harder to solve. When considering Figure 7.7 (c) (i.e., the 8×8 instances) we observe as $MILP_0$(L) and MILP(L) are producing comparable solutions. While, their energy minimisation counterparts, are still behaving as for the smaller problems. While ZZ and RND are still almost comparable (RND produces better latency results while obtaining a comparable energy consumption). Finally, in Figure 7.7 (d), when the largest instances are considered (8×16) we notice that the latency formulations ($MILP_0$(L) and MILP(L)) produce results that are comparable to the ZZ performance, while the RND results in poor energy performance. Whereas, the two energy based formulations produce similar results showing how, the gap that was present on the smaller instances, is lost because of the size of the problems. These plots show clearly *(i)* how the latency MILP models are harder to solve than the models based on the energy minimization objective function, *(ii)* that the two heuristics (ZZ and RND) are not dominated each other even if ZZ on average produces better results and *(iii)* how their quantitative performance are dominated by the two MILP based algorithms.

7.4.2.4 Layout effect

Here, we show how the proposed mathematical formulations can also be used to evaluate the impact of layout, and also how the performance are influenced by the shape of the NoC. More precisely, we consider an additional NoC configuration (5×6), and we compare it with a 4×8 NoC. To make a fair comparison, the same applications used in the 4×8 test cases are given as input to the 5×6 layout. Observe that, since the 5×6 has only 30 cores it may happen that an instance to be unfeasible since 4×8 contains more than 30 cores. However, in our test cases, this never happen. Results are reported in Figure 7.8, where the energy consumption and the latency are shown for the two layouts under comparison and four considered algorithms. It can be observed that in almost all the cases, due to its more regular shape, the 5×6 layout allows accommodating the same threads requiring more energy and latency. Comparing between 5×6 and 4×8 network size, it is interesting to note that 4×8 size consumes up to 3.4% more for $MILP_0$ and 4.2% more for MILP the energy cost. The higher cost can also be seen for latency, and it is 18.9% more for $MILP_0$ and only 0.6% more for MILP. Similar changes can also be observed for other two algorithms.

In Figure 7.9 and 7.10, we show the queue length obtained on a sample instance for the two layouts. In details in Figure 7.9, we report on the left the queue lengths for a NoC with a single MC whereas on the right the same threads are allocated on a NoC with two MCs. We can clearly observe that the presence of the second MC can alleviate



(a) Average energy cost (b) Average packet latency

Figure 7.8: Performance comparison between 5×6 and 4×8

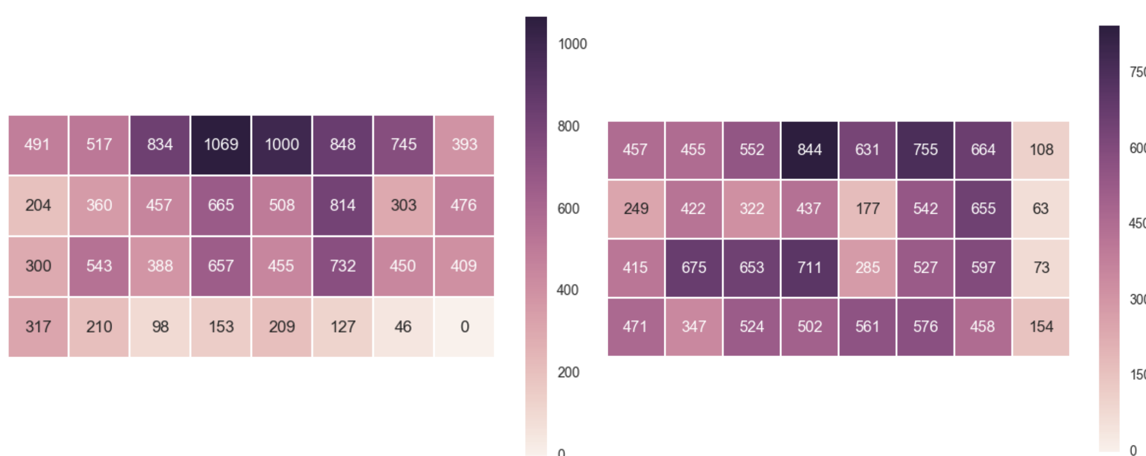


Figure 7.9: Comparing the queue length for network size 4×8: (left) MC=1, (right) MC=2

hot spots (which should be) and also the congestions since threads can access the RAM from both up and down sides. The hot spot reduction can be observed since in the left case the longest queue contains over 1000 packets when MC count is two it reduced to less than 850 packets. Similar reductions can also be observed when considering the 5×6 layout (see Figure 7.10).

7.4.3 Applicability of the model and practical findings

Mapping application threads on free cores while optimising the energy cost or optimising the average data packet latency is a complex task and it gets even more challenging as the NoC size increases. We are aware that the approach proposed in this chapter may not be used in real-time, but the aim of the work is to introduce a technique to evalu-

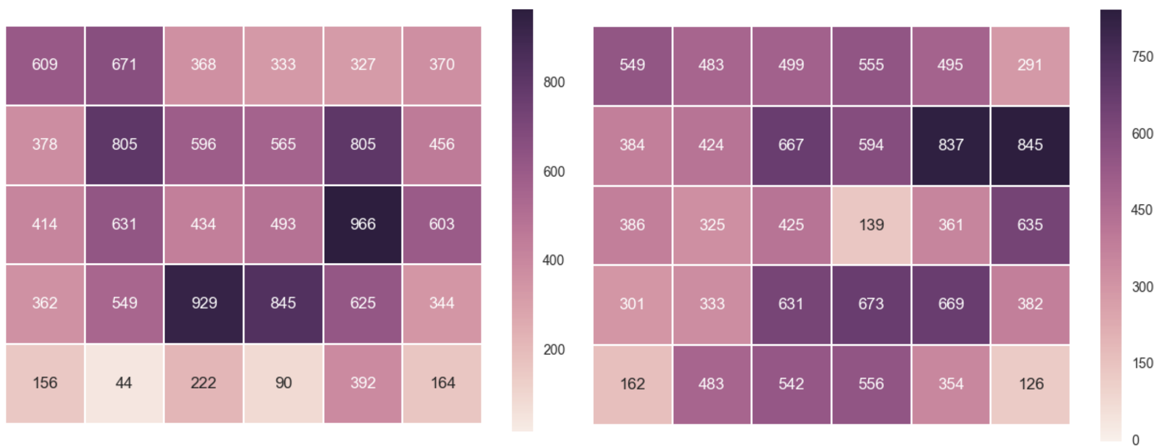


Figure 7.10: Comparing the queue length for network size 5×6: (left) MC=1, (right) MC=2

ate the best performance that could be attainable in practice. In fact, it is also shown that non-minimal routing policy allows, better hot spot reduction and higher fault tolerance [GN92], but adaptive routing needs complex control logic for routing which in turn may increase data packet delay. Our finding confirms that there is a margin for up to 40% improvement over simple rules (ZZ, RND). Moreover, if thread migration is allowed it may improve the results further. We also have shown the increased support (from one MC to two MC) for NDP improves both objective functions (energy by a 5%, latency more than 20%). Overall, we believe this model can be used as a yardstick for validating various T2C mapping policies and layouts.

7.5 Summary

Recently, power-aware computing is becoming popular for the increasing power demand for computation. There are chips with large low-power cores offering a huge potential for higher potentials. Efficient interconnects are required to support the information exchange of such a vast number of processing cores, but still need to provide low latency, high network throughput and scalability with a better area and power costs. In this chapter, we have proposed an MILP based formulation to optimise the power cost and latency also. The alternative objective function provides a needed flexibility because user's optimisation requirement can change over time. Our model achieves optimality in some cases and also provides the opportunity to increase the solution quality based on the growing solution time. It also supports NDP by changing the total number of MCs to optimise the energy as well as latency further. Future research directions include both the application of the developed model to different NoC topologies (such as folded-torus,

flattened tree and with the hybrid topologies such as mesh with ring) and the extension of the formulation to include the application migration support to make the model more robust.

7.6 Acknowledgement

This chapter is an edited version of the submitted paper entitled **An Analytical Model for Thread-Core Mapping for Tiled CMPs** by Marco Pranzo and Somnath Mazumdar at IEEE Transactions on Computers. Status: Under review

8

Conclusion and Future Work

This chapter describes the conclusions in brief obtained during this doctoral work and also lay out the contexts for future research works.

8.1 Contribution

In this doctoral thesis, a framework (mainly the essential components of the framework) has been proposed to improve the runtime adaptability of dataflow program execution models (PXMs) that are built around a data-driven approach. The framework for the efficient dataflow thread execution is based upon multiple components. They are: *Distribution*, *Monitoring*, *Execution*, and *Analysis*. It is also worth to note that to bind the features of these components, multiple (similar) software interfaces are also provided. The thesis mainly aims to connect the services provided by the components logically at runtime. It mainly provides a mechanism for management and monitoring the huge number of concurrent dataflow threads at NoC level at runtime enabling the high performance with low overheads. Improvements of this work can be made to make all the steps in an automatic way, but providing an effective autonomic approach is a very complex task.

In general, applications are composed of multiple threads and the threads may interact with each other during execution. To reach the ultimate goal, there are multiple run time execution related goals which must be satisfied. This hypothesis acknowledges the difficulty of achieving those goals with low overheads. The proposed framework subdivided the main goal into multiple smaller goals at various components.

- By efficiently distributing the dataflow threads at the hardware level, the performance would be improved. In Chapter 3 a hash based thread distribution mechanism for a data-driven PXM (Codelet) with very low overheads has been proposed. The proposed hash mechanism also shows that it can efficiently support larger

core counts with stable performance. Even though a more complex thread distribution policy could be employed but there is a no guarantee that the performance would be better, but certainly, the area and power cost would be increased.

- While distributing the threads, it is always recommended to monitor the activity of the threads. The proposed monitoring tool (RADA) can be used not only to provide its hardware abstraction but also allow the user to embed their thread scheduling mechanism so that the increased overhead of the proposed scheduling mechanism can be identified before hand. Apart from that in this tool, appropriate metrics could also be added but computing the metrics depends on the features available by the supporting AMM.
- By moving the thread control from a high level to low level, the underlying infrastructure must be capable of supporting the threads. In Chapter 5, a software defined NoC model has been shown which can reconfigure itself to provide better support to data-driven PXMs. This step is an important phase of the hypothesis, as it proposes a link between the dataflow threads and its physical substrate. The embedded logic in the SDNoC not only provides an efficient data-driven PXM support but also at a lower cost.
- To further extend the framework, Chapter 6 provides a hybrid NoC design to support both the control-driven and data-driven PXMs. The blocks in the design are used to exploit the data traffic localisation feature for better traffic management. The blocks are also ideal to implement the main components (TPs, VNs) of the employed Codelet PXM in this doctoral work.
- Verifying if the mapping of application threads on free cores is leading to an optimal or non-optimal solution, to achieve this, Chapter 7 has shown an analytical (MILP) model to verify the thread-to-core mapping quality. In this problem, the threads are represented via DAG. The main aim of the model is to reduce the energy cost and latency. The model also supports multiple applications to be mapped on the NoC with variable memory controllers.

The implementation of the framework has been in multiple steps using different simulation tools to show its feasibility. The reconfiguration capability of the SD-NoC design is also crucial to improve the run time thread performance. At the same time, multiple components based framework development approach also provides an easier way to improve the features of each steps independently. The main aim of this doctoral thesis is to show the utility of having a NoC based framework to improve the run time adaptability of data-driven threads. The framework improvements can be considered as the future work.

8.2 Future work

Multithreaded applications do not scale easily due to many issues (such as complex thread management, synchronisation, load imbalance, memory hierarchy). As it is already mentioned that this hypothesis must be considered as the first *skeleton* of NoC level autonomic dataflow thread management based framework. By this proposed framework the possibility of collaborating multiple phases (following the top-down approach) has also been shown (in previous chapters). In this section, the promising future research directions are briefly highlighted.

- The hash-based distribution mechanism is not new and has been successfully used in the network domain. However, the shown results in the concerned chapter is based on the synthetic applications so the future work would be to port real application benchmarks and perform the stress testing of the hash mechanism. It is always important to see is there any performance bottleneck or not and also to find out any hotspot in the system.

- The monitoring tool is now capable of embedding the thread scheduling unit and also shows the generated traffic due to the scheduler, but the fine-grain traffic overhead must be explored. For better thread support the feedback unit can be improved with other robust neural network model (such as recurrent neural networks (RNNs)). The current functional model should be replaced by a timing model for monitoring better thread performance.

- The SDNoC is also an interesting area to be explored. We need more accurate simulation (may be FPGA-based) to find the limitation of the model. In general, the more hardware resources can be added to support very intensive traffic, but it will cost power and area. It is also worth to detect the contention and also the reliability (or failure). These critical features are worth to explore, but the modification will certainly increase the router complexity with increased area and power cost.

- The features of proposed hybrid NoC can also be further enhanced by adding morphing capabilities to have more subtle power management (see section 6.5 for more detail).

- The presented analytical model can be used to check the thread-to-core mapping is optimal or not. If the mapping is not optimal, it is also very useful to know how much the optimality gap is. Although the proposed model has not been developed to be implemented in real-time but proposing new algorithm might improve the solution time. Finally, the performance of the analytical model can be tested on

other topologies such as folded torus, flattened butterfly or other hybrid topology (such as ring-2D mesh based).

Future chips will be shared among multiple applications which have different resource requirements. Hence multiple performances related metrics, and also the service level objectives or the QoS must be maintained. It is worth to mention that used test cases are designed keeping in mind their applicability of their concurrent execution at real time, without moving the system in an unstable state. The possibility of having a framework that supports autonomous concurrent, multiple data-driven thread execution over a non-empty set of computing resources, can bring the issues of non-deterministic behaviour which is another research domain worth to be further investigated.

Bibliography

[AAS13] Michael Opoku Agyeman, Ali Ahmadinia, and Alireza Shahrabi. Efficient routing techniques in heterogeneous 3d networks-on-chip. *Parallel Computing*, 39(9):389–407, 2013.

[ACP04] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. Multi-objective mapping for mesh-based noc architectures. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 182–187. ACM, 2004.

[Ada13] Inc. Adapteva. Epiphany architecture reference, 2013.

[AFF+09] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[AFY+16] Rachata Ausavarungnirun, Chris Fallin, Xiangyao Yu, Kevin Kai-Wei Chang, Greg Nazario, Reetuparna Das, Gabriel H Loh, and Onur Mutlu. A case for hierarchical rings with deflection routing: An energy-efficient on-chip communication substrate. *Parallel Computing*, 54:29–45, 2016.

[AN90] Arvind and Rishiyur S Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on*, 39:300–318, 1990.

[AR82] Romas Aleliunas and Arnold L Rosenberg. On embedding rectangular grids in square grids. *Computers, IEEE Transactions on*, 100(9):907–913, 1982.

[ARM13] ARM. big.little technology: The future of mobile, 2013.

[AW77] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20:519–526, 1977.

[BAM10] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[Bar64] Paul Baran. On distributed communications networks. *IEEE transactions on Communications Systems*, 12(1):1–9, 1964.

[BC11] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BCGK04] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. Cost considerations in network on chip. *INTEGRATION, the VLSI journal*, 38(1):19–42, 2004.

[BD06] James Balfour and William J Dally. Design tradeoffs for tiled cmp on-chip networks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198. ACM, 2006.

[BM06a] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[BM06b] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38:1, 2006.

[BSP+16a] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, Bin Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, June 2016.

[BSP+16b] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pages 1–2. IEEE, 2016.

[BWFM09] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97. IEEE, 2009.

[BZ07] Stephan Bourduas and Zeljko Zilic. A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 195–204. IEEE, 2007.

[CAB+13] Nicholas P Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A

Golliver, Rob Knauerhase, et al. Runnemede: An architecture for ubiquitous high-performance computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 198–209. IEEE, 2013.

[CCLL08] Ye-In Chang, Hue-Ling Chen, Sih-Ning Li, and Hung-Ze Liu. A dynamic hashing approach to supporting load balance in p2p systems. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 429–434. IEEE, 2008.

[CGMP10] M. Conti, S. Giordano, M. May, and A. Passarella. From opportunistic networks to opportunistic computing. *IEEE Communications Magazine*, 48(9):126–139, 2010.

[CGSVE95] David E Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten Von Eicken. Empirical study of a dataflow language on the cm-5. *Advanced Topics in Dataflow Computing and Multithreading*, pages 187–210, 1995.

[CHL+08] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.

[CLK07] Guangyu Chen, Feihui Li, and Mahmut Kandemir. Compiler-directed application mapping for noc based chip multiprocessors. In *ACM SIGPLAN Notices*, volume 42, pages 155–157. ACM, 2007.

[Cor94] Henk Corporaal. Design of transport triggered architectures. In *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV'94, Proceedings., Fourth Great Lakes Symposium on*, pages 130–135. IEEE, 1994.

[Cor97] Henk Corporaal. Microprocessor architectures: from vliw to tta. 1997.

[CP03] Xuning Chen and Li-Shiuan Peh. Leakage power modeling and optimization in interconnection networks. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 90–95. ACM, 2003.

[D+11] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[DAM+13] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 107–118. IEEE, 2013.

[dDdML+13] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[DEM+09] Reetuparna Das, Soumya Eachempati, Asit K Mishra, Vijaykrishnan Narayanan, and Chita R Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 175–186. IEEE, 2009.

[Den74] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[Den80] Jack Bonnell Dennis. Data flow supercomputers. *Computer*, (11):48–56, 1980.

[Den86] Jack B Dennis. Data flow computation. In *Control Flow and Data Flow: concepts of distributed programming*, pages 345–398. Springer, 1986.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGH+10] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, Aug 2010.

[DM75] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.

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

[DT04a] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[DT04b] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[EF15] Guy Even and Yaniv Fais. Algorithms for network-on-chip design with guaranteed qos. *arXiv preprint arXiv:1509.00249*, 2015.

[EG90] Paraskevas Evripidou and Jean-Luc Gaudiot. A decoupled graph/computation data-driven architecture with variable-resolution actors. Technical report, University of Southern California, Los Angeles, CA (United States). Dept. of Electrical Engineering, 1990.

[EJ03] Johan Eker and Jorn Janneck. Cal language report. Technical report, Tech. Rep. ERL Technical Memo UCB/ERL, 2003.

[EWB+07] Bruce Edwards, David Wentzlaff, Liewei Bao, Henry Hoffmann, Chyi-Chang Miao, Carl Ramey, Matthew Mattina, Patrick Griffin, Anant Agarwal, and John F. Brown III. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 2007.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[GAD+13] Puneet Gupta, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Rajesh K Gupta, Rakesh Kumar, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B Srivastava, et al. Underdesigned and opportunistic computing in presence of hardware variability. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 32(1):8–23, 2013.

[GBB+14] Roberto Giorgi, Rosa M Badia, François Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R Gao, Arne Garbade, Rahul Gayatri, et al. Teraflux: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8):976–990, 2014.

[GF14] Roberto Giorgi and Paolo Faraboschi. An introduction to df-threads and their execution model. In *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pages 60–65. IEEE, 2014.

[GHKM11] Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 401–412. ACM, 2011.

[Gio12] Roberto Giorgi. Teraflux: exploiting dataflow parallelism in teradevices. In *Proceedings of the 9th conference on Computing Frontiers*, pages 303–304. ACM, 2012.

[Gio15] Roberto Giorgi. Scalable embedded systems: Towards the convergence of high-performance and embedded computing. In *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on*, pages 148–153. IEEE, 2015.

[GJ79] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[GKW85] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28:34–52, 1985.

[GN92] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 278–287. ACM, 1992.

[GP+77] Kim P Gostelow, Wil Plouffe, et al. Indeterminacy, monitors, and dataflow. In *ACM SIGOPS Operating Systems Review*, volume 11, pages 159–169. ACM, 1977.

[GS15] Roberto Giorgi and Alberto Scionti. A scalable thread scheduling co-processor based on data-flow principles. *Future Generation Computer Systems*, 53:100–108, 2015.

[Gur16] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2016.

[GZM+17] Tongsheng Geng, Stéphane Zuckerman, José Monsalve, Alfredo Goldman, Sami Habib, Jean-Luc Gaudiot, and Guang R. Gao. *The Importance of Efficient Fine-Grain Synchronization for Many-Core Systems*, pages 203–217. Springer International Publishing, 2017.

[HAQT+04] Wei Hung, Charles Addo-Quaye, Theo Theocharides, Yuan Xie, N Vijakrishnan, and Mary Jane Irwin. Thermal-aware ip virtualization and placement for networks-on-chip architecture. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 430–437. IEEE, 2004.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79:1305–1320, 1991.

[HJ01] V Carl Hamacher and Hong Jiang. Hierarchical ring network configuration and performance modeling. *IEEE Transactions on Computers*, 50(1):1–12, 2001.

[HM05] Jingcao Hu and Radu Marculescu. Energy-and performance-aware mapping for regular noc architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4):551–562, 2005.

[HP86] Wen-reel Hwu and Yale N Patt. Hpsm, a high performance restricted data flow architecture having minimal functionality. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 297–306. IEEE Computer Society Press, 1986.

[HPD12] R Curtis Harting, Vishal Parikh, and William J Dally. Energy and performance benefits of active messages. *Concurrent VLSI Architectures Group, Stanford University, Tech. Rep*, 131, 2012.

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

[HZC+06] Yuanfang Hu, Yi Zhu, Hongyu Chen, Ronald Graham, and Chung-Kuan Cheng. Communication latency aware low power noc synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 574–579. ACM, 2006.

[I+88] Robert A Iannucci et al. *Two fundamental issues in multiprocessing*. Springer, 1988.

[Jac] Bruce Jacob. The case for vliw-cmp as a building block for exascale.

[JP14] Yuho Jin and Timothy Mark Pinkston. Pais: Parallelism-aware interconnect scheduling in multicores. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):108, 2014.

[KAH11] Somayyeh Koohi, Meisam Abdollahi, and Shaahin Hessabi. All-optical wavelength-routed noc based on a novel hierarchical topology. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pages 97–104. ACM, 2011.

[KBD07] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182. IEEE Computer Society, 2007.

[KDSA08] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 77–88. IEEE Computer Society, 2008.

[KET06] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-driven multithreading using conventional microprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 17(10):1176–1188, 2006.

[KFJ+03] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[KGA01] Krishna M Kavi, Roberto Giorgi, and Joseph Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *Computers, IEEE Transactions on*, 50:834–846, 2001.

[KK09] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proceedings of the 2nd international workshop on network on chip architectures*, pages 5–10. ACM, 2009.

[KKH+09] Dara Kusic, Jeffrey O Kephart, James E Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster computing*, 12(1):1–15, 2009.

[KKM+14] Hanjoon Kim, Gwangsun Kim, Seungryoul Maeng, Hwasoo Yeo, and John Kim. Transportation-network-inspired network-on-chip. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 332–343. IEEE, 2014.

[KLN15] Andrew B Kahng, Bill Lin, and Siddhartha Nath. Orion3. 0: a comprehensive noc router estimation tool. *IEEE Embedded Systems Letters*, 7(2):41–45, 2015.

[KPKJ07] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K Jha. Express virtual channels: towards the ideal interconnection fabric. *ACM SIGARCH Computer Architecture News*, 35(2):150–161, 2007.

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[KTJR05] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, (11):32–38, 2005.

[LCOM07] 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(3):23, 2007.

[Lee06] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[LK03] Tang Lei and Shashi Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 180–187. IEEE, 2003.

[LKGF+12] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 500–511. IEEE Computer Society, 2012.

[LNLK13] Junghee Lee, Chrysostomos Nicopoulos, Hyung Gyu Lee, and Jongman Kim. Tornadonoc: A lightweight and scalable on-chip network architec-

ture for the many-core era. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):56, 2013.

[LNP+13] Junghee Lee, Chrysostomos Nicopoulos, Sung Joo Park, Madhavan Swaminathan, and Jongman Kim. Do we need wide flits in networks-on-chip? In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 2–7. IEEE, 2013.

[MBDM05] Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 1, pages 27–32. IEEE, 2005.

[MCM+04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1):69–93, 2004.

[MDM04a] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, page 20896. IEEE Computer Society, 2004.

[MDM04b] 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*, pages 914–919. ACM, 2004.

[MDM04c] 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*, pages 914–919. ACM, 2004.

[MHH+15] Abderrahmen Mtibaa, Khaled A Harras, Karim Habak, Mostafa Ammar, and Ellen W Zegura. Towards mobile opportunistic computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 1111–1114. IEEE, 2015.

[MJW12] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[MOP+09] Radu Marculescu, Umit Y Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in noc design: system, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):3–21, 2009.

[MV15] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[N+90] RS Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[NGS15] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 298–310. ACM, 2015.

[NMN+14] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[OB04] Asuman E Ozdaglar and Dimitri P Bertsekas. Optimal solution of integer multicommodity flow problems with application in optical networks. In *Frontiers in global optimization*, pages 411–435. Springer, 2004.

[ODH+07] John D Owens, William J Dally, Ron Ho, DN Jayasimha, Stephen W Keckler, Li-Shiuan Peh, et al. Research challenges for on-chip interconnection networks. *IEEE micro*, 27(5):96, 2007.

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

[PBB+03] Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 63–72. ACM, 2003.

[PC90] Gregory M Papadopoulos and David E Culler. Monsoon: an explicit token-store architecture. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 82–91. ACM, 1990.

[PDB14] Ritesh Parikh, Reetuparna Das, and Valeria Bertacco. Power-aware nocs through routing and topology reconfiguration. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[PH12] Michael K Papamichael and James C Hoe. Connect: re-examining conventional wisdom for designing nocs in the context of fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 37–46. ACM, 2012.

[PL13] Mencer Oskar Tsoi Kuen Hung Pell, Oliver and Wayne Luk. *High-Performance Computing Using FPGAs*, chapter Maximum Performance Computing with Dataflow Engines, pages 747–774. Springer New York, New York, NY, 2013.

[RRB69] JE Rodrigues and Jorge E Rodriguez Bezos. A graph model for parallel computations. Technical report, Massachusetts Institute of Technology, 1969.

[RS97] Govindan Ravindran and Michael Stumm. A performance comparison of hierarchical ring-and mesh-connected multiprocessor networks. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 58–69. IEEE, 1997.

[SAPMVA+16] R Sandoval-Arechiga, R Parra-Michel, JL Vazquez-Avila, J Flores-Troncoso, and S Ibarra-Delgado. Software defined networks-on-chip for multi/many-core systems: A performance evaluation. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, pages 129–130. ACM, 2016.

[SBG+08] Sander Stuijk, Twan Basten, Marc Geilen, Amir Hossein Ghamarian, and Bart Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. *Journal of Systems Architecture*, 54:411–426, 2008.

[SBSK12] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 152–160. IEEE, 2012.

[SC05] Krishnan Srinivasan and Karam S Chatha. A technique for low energy mapping and routing in network-on-chip architectures. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 387–392. ACM, 2005.

[SC13] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1):60–76, 2013.

[SCK+12] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 201–210. IEEE, 2012.

[SIL+15] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, 2015.

[SK04] Dongkun Shin and Jihong Kim. Power-aware communication optimization for networks-on-chips with voltage scalable links. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 170–175. ACM, 2004.

[SMP16] A. Scionti, S. Mazumdar, and A. Portero. Software defined network-on-chip for scalable cmps. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 112–115. IEEE, 2016.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[SPS+07] M Aater Suleman, Yale N Patt, Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, and Doug Carmean. Asymmetric chip multiprocessors: Balancing hardware efficiency and programmer efficiency. *Univ. Texas, Austin, TR-HPS-2007-001*, 2007.

[SSPH14] Rasmus Bo Sørensen, Jens Sparsø, Mark Ruvald Pedersen, and Jaspur Højgaard. A metaheuristic scheduler for time division multiplexed

networks-on-chip. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 309–316. IEEE, 2014.

[STE06] Kyriakos Stavrou, Pedro Trancoso, and Paraskevas Evripidou. *Hardware Budget and Runtime System for Data-Driven Multithreaded Chip Multiprocessor*, pages 244–259. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[SZG13] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R Gao. An implementation of the codelet model. In *Euro-Par 2013 Parallel Processing*, pages 633–644. Springer, 2013.

[The99] Kevin Bryan Theobald. *EARTH: AN EFFICIENT ARCHITECTURE: FOR RUNNING THREADS*. PhD thesis, McGill University, Montréal Québec, Canada, 1999.

[Tos11] Suleyman Tosun. Cluster-based application mapping method for network-on-chip. *Advances in Engineering Software*, 42(10):868–874, 2011.

[TT11] Hung-Wei Tseng and Dean M Tullsen. Data-triggered threads: Eliminating redundant computation. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 181–192. IEEE, 2011.

[TT12] Hung-Wei Tseng and Dean Michael Tullsen. Software data-triggered threads. *ACM SIGPLAN Notices*, 47(10):703–716, 2012.

[TT14] Hung-Wei Tseng and Dean M Tullsen. Cdtt: Compiler-generated data-triggered threads. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 650–661. IEEE, 2014.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[UMB10] Aniruddha N Udipi, Naveen Muralimanohar, and Rajeev Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[VBS+95] Zvonko G Vranesic, Stephen Brown, Michael Stumm, Steven Caranci, Alex Grbic, Robin Grindley, Mitch Gusat, Orran Krieger, Guy Lemieux, Kevin Loveless, et al. *The NUMAchine multiprocessor*. Citeseer, 1995.

[VDBN98] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 140–151. ACM, 1998.

[VHR+08a] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[VHR+08b] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[WPM03] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 105. IEEE Computer Society, 2003.

[WSK+05] Pascal T Wolkotte, Gerard JM Smit, Nikolay Kavaldjiev, Jens E Becker, and Jürgen Becker. Energy model of networks-on-chip and a bus. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 82–85. IEEE, 2005.

[YAMJGE14] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. Hybrid dataflow/von-neumann architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1489–1509, 2014.

[ZGHC15] Naijun Zheng, Huaxi Gu, Xin Huang, and Xiaokang Chen. Csquare: A new kilo-core-oriented topology. *Microprocessors and Microsystems*, 39(4):313–320, 2015.