UNIVERSITY OF FERRARA
ENGINEERING DEPARTMENT

DOCTOR OF PHILOSOPHY DEGREE IN SCIENCE OF ENGINEERING
CYCLE XXVIII
COORDINATOR PROF. STEFANO TRILLO

NOC-CENTRIC PARTITIONING AND RECONFIGURATION TECHNOLOGY FOR THE EFFICIENT SHARING OF GENERAL-PURPOSE PROGRAMMABLE MANY-CORE ACCELERATORS

SCIENTIFIC DISCIPLINARY SECTOR
ING-INF/01

CANDIDATE
DOTT. MARCO BALBONI

ADVISOR
PROF. DAVIDE BERTOZZI

ACADEMIC YEARS 2013-2015

UNIVERSITY OF FERRARA
ENGINEERING DEPARTMENT

DOCTOR OF PHILOSOPHY DEGREE IN SCIENCE OF ENGINEERING
CYCLE XXVIII
COORDINATOR PROF. STEFANO TRILLO

NOC-CENTRIC PARTITIONING AND RECONFIGURATION TECHNOLOGY FOR THE EFFICIENT SHARING OF GENERAL-PURPOSE PROGRAMMABLE MANY-CORE ACCELERATORS

SCIENTIFIC DISCIPLINARY SECTOR
ING-INF/01

CANDIDATE
DOTT. MARCO BALBONI

ADVISOR
PROF. DAVIDE BERTOZZI

ACADEMIC YEARS 2013-2015

Acknowledgements

I have waited for this moment for a long time, basically all my life of student. Now, I wonder what actually makes this moment unique. Today I feel like the first part of an exciting journey is overing and a new one is starting. Probably as a child spends his youth dreaming to become a man similarly I was dreaming to complete my studies to start my job career. However this moment is much more than this, more than the excitement of starting a new era of my life. This moment represents the chance to thank all the people that have helped and supported me on every step until here. This moment can repay a part, at least a small part, of the infinite faith put in me in all these years.

My PhD started when my advisor **Davide Bertozzi** gave me the opportunity to join the **MPSoC group at University of Ferrara**. Needless to say, Davide has been the key person during my PhD. He has been more than a simple advisor, he did not only show me the way for a prolific research but he also provided me pure lessons of life. In Ferrara, I had the chance to work on ambitious topics in a great research group. A research group built around the harmony and the enthusiasm of challenging always new research problems. Together with endless scientific stimuli, Davide surrounded me by an unconditioned trust and pushed me to go beyond my limits. In these years, I had opportunities that I could neither imagine to have. I traveled around the world taking part to prestigious conferences, I met some of the pioneers in my research field, I exchanged opinions and point of view with persons of every culture and professional background, I had the chance to work in many topics and many countries. I'm really thankful for all these unrepeatable experiences that made me grow and I will always bring with me.

A special thank you is due to prof. **Luca Benini** that had me under his wing as co-advisor for these years of research, believing in my potentiality and guiding me with his expertise, competence, knowledge and professionalism, and let me be part of the great **MultithermanLab Group at DEI at University of Bologna**. This latter group represented the best environment where I could ever imagine to develop my research and spend the most part of these years.

I want to thank all the persons of my research group in Ferrara. Alessandro S. and Daniele L. that have helped me immensely in the first period of my PhD. They represented my reference point for every problem, every doubt and every question.

Thanks for all your patience, I have learned a lot from you guys. I'm thankful to Alberto, Mahdi, Marta, Lorenzo, Gabriele and Hervé, too. A special thank to Luca to show me the right way to be part of this group and to support me in each moment of this PhD. It was fantastic to work side by side with all of you. I have been impressed by your fairness, your sensitivity and your intelligence. I enjoyed all the moments that we had together. I even enjoyed the most difficult of them as the times when we had to work hard until the early morning hours. Together we have been able to meet the more challenging of the deadlines. Thanks to all of you my friends. I wish you the greatest satisfaction from the life. I also owe all the people part of my other research group in Bologna. It was a honor to be part of the "Lab Famiglia" at the basement of that old building. I will bring all of you in my mind as the best friends and colleagues of my life. Probably we are destined to work in great companies or research groups in the future, but for me nothing will be comparable to our group. So again big thanks to Giuseppe, Francesco B., Francesco C., Francesco P., Daniele "Casarini", Daniele B., Andrea B., Chrstian P., Erfan, Igor, Davide, Thomas and especially to Andrea M. and Alessandro. You are all more than friends for me.

I could not forget the guys and girls at **ARM Ltd in Cambridge (UK)**, in particular Rekai, Radhika, René, Andreas S., Stefano, Sasha, Roxana and the king Omar. I had many enjoyable moments during my internship. A big thank also to the head of the group **Andreas Hansson** to be a great leader and to give me the opportunity to be part of ARM, actually one of the best world's company. Finally, I owe my parents and my whole family (mom, dad and Sabri) and in particular my cousins Elli and Luk to have sustained me in all my choices and to have patiently followed me through all my sad and happy moments of this long way. Their invisible, but still strong and caring presence have encourage me several times.

Marco

Ferrara, Italy, April 2016

Abstract

During the last few decades an unprecedented technological growth has been at the center of embedded systems design, with Moore's Law being the leading factor of this trend. Today, in fact, an ever increasing number of cores can be integrated on the same die, marking the transition from state-of-the-art multi-core processors to the new many-core design paradigm. The aim of such many-core devices is twofold: providing high computing performance and increasing the energy efficiency of the hardware in terms of OPS/Watt. Despite the extraordinarily high computing power, the complexity of many-core systems opens up several challenges to be tackled by designers, concerning the runtime management of the computing fabric. The challenge tackled by this thesis is twofold. One one hand, software parallelism does not scale to the same extent of hardware parallelism, therefore the problem arises about how to share the computation resources among a set of concurrent applications. On the other hand, management tasks of the many-core system become fundamental runtime operations, which need to be transparently executed while avoiding to suspend system computation.

This thesis provides a whole set of design methods to master the runtime complexity of feature-rich industry-ready many-core accelerators, relying on hardware extensions of the on-chip interconnection network (Network-on-chip, NoC). *The key idea is to exploit a Space-Division Multiplexing strategy to schedule the execution of concurrent applications that require to be accelerated onto the same array fabric of homogeneous processing tiles at the same time, thus enabling the efficient exploitation of the underlying hardware resources.* The most advanced application of this idea consists of embedded system virtualization on top of heterogeneous computing architectures, where multiple virtual machines running on a host processor may want to offload computation to a many-core programmable accelerator. In this context, virtualization implies flexible partitioning of computation and memory resources, isolation for protection, and reconfiguration for workload adaptivity at runtime. While resource management should be on burden of a "control tower" in software (hypervisor), partitioning, isolation and reconfiguration need to be assited in hardware, especially in the platform integration framework, consisting of the communication architecture.

The first contribution of this thesis consists of validating the novel SDM-based resource sharing paradigm. Therefore, it compares an SDM approach with a traditional one based on time-division multiplexing. To evaluate the different strategies, the thesis makes use of parallelized Image Processing benchmarks, whose execution is managed by an optimized version of the OpenMP Runtime Environment, needed to enable their parallel execution. The benchmarks are executed on different simulation environments (VirtualSoC and gem5), which required the customization and extension of such environments with new functionalities to simulate a General-Purpose Many-Core Programmable Accelerator. As a result, the thesis aims at capturing the impact on performance of parallelism, size and shape of partitions (numbers of computational clusters reserved, and their position in the manycore fabric), as well as memory configuration.

The second main contribution of the thesis consists of enabling a highly-dynamic resource management of manycore accelerators. In fact, the flexible sharing strategy of a networked many-core fabric depends ultimately on the runtime reconfiguration capability of the NoC routing function, therefore this thesis aims at a fast and scalable routing reconfiguration mechanism with minimum perturbation

of background traffic. The thesis provides at first a centralized solution to this problem, and finally a fully distributed one, and assesses the area and performance implications by means of an advanced FPGA prototype. This contribution paves the way for future fine-grained workload-adaptive system configurations, as well as for transparent online testing strategies of selective components.

Finally, the thesis aims at the deployment of the developed spatial partitioning strategies into more futuristic systems, characterized by the integration of manycore fabrics with emerging interconnect technologies. This thesis focuses on the photonic integration case study, and co-designs the partitioning and reconfiguration features of a programmable accelerators with the basic requirement of minimizing the static power overhead of optical NoCs. This is achieved through a re-use technique of laser sources across computation partitions.

Finally, the thesis re-architects the complete hierarchical communication infrastructure in a template heterogeneous parallel computing architecture with photonic integration, and comes up with a hybrid interconnect fabric that paves the way for future research.

Contents

List of Figures

List of Tables

List of Acronyms and Symbols

ASIC	Application-Specific Integrated Circuit
BISD	Built-In Self-Diagnosis
BIST	Built-In Self-Test
CMP	Chip MultiProcessor
CPU	Central Processing Unit
CSU	Central Synchronizer Unit
CVP	Clock Variability Power
DDR	Double Data Rate
DEMUX	Demultiplexer
DMA	Direct Memory Access
DUT	Device Under Test
DVFS	Dynamic Voltage and Frequency Scaling
FF	Flip-Flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GALS	Globally Asynchronous Locally Synchronous
GPPA	General-Purpose Programmable Accelerator
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HWPE	Hardware Processing Element
HWS	Hardware Synchronizer
IP	Intellectual Property
ISA	Instructions Set Architecture
ITRS	International Technology Roadmap for Semiconductors
LBDR	Logic-Based Distributed Routing
LFSR	Linear Feedback Shift Register
LIC	Local Interconnect
LUT	Look-Up Table
MIMD	Multiple Instructions Multiple Data
MISR	Multiple Input Signature Register
MoT	Mesh-of-Trees

MMU	Memory Management Unit
MPPA	Multi-Purpose Processor Array
MPSoC	Multi Processor System on Chip
MUX	Multiplexer
NI	Network Interface
NoC	Network-on-Chip
NPM	Native Programming Model
OCP	Open Core Protocol
P&R	Place and Route
PGAS	Partitioned Global Address Space
RR	Round Robin
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SDM	Space-Division Multiplexing
SoC	System on Chip
TCDM	Tightly Coupled Data Memory
TDM	Time-Division Multiplexing
TLM	Transaction-Level Modeling
TPG	Test Pattern Generator
VC	Virtual Channel
VLSI	Very Large Scale Integration
VPU	Visual Processing Unit

Introduction

Today, embedded systems are rapidly moving from small homogeneous systems with few powerful computing units, towards the integration of thousands of potentially heterogeneous cores on a single chip, leading to high-end heterogeneous Multi-Processor Systems on Chip (MPSoC). Indeed, driven by flexibility, performance and cost constraints of demanding modern applications, heterogeneous MPSoCs are now the dominant design paradigm in the embedded computing domain [95], and they are even shaping the future of high-performance computing systems, which are in desperate need of energy efficiency. Heterogeneous SoC architectures clearly provide a wider spectrum for the power/performance trade-off, combining host CPUs along with massively parallel accelerator fabrics, holding the potential of bridging the gap between the energy efficiency (GOPS/W) of hardwired hardware accelerators and the computational power delivered by throughput computing. As a potential source of hardware acceleration for many different algorithms [92], today, tile-based many-core programmable accelerators are gaining momentum. Unfortunately, apart from specific application domains (such as 3D graphics), the software parallelism does not keep up with hardware parallelism. Therefore, this latter runs the risk of going underutilized in real-life systems. Fortunately, another concurrent trend seems to be able to offset the above limitation. In fact, modern high-end embedded systems need to consolidate complex functionalities onto the same heterogeneous parallel hardware platform, requiring the concurrent execution and acceleration of several applications, possibly with heterogeneous and time-varying performance/reliability/power requirements.

Therefore, the efficient exploitation of the abundant hardware resources will progressively go through the sharing of such resources among a large number of concurrently executing applications. This new scenario, with *the advent of heterogeneous parallel computing architectures, has profoundly changed the panorama of both hardware and software design, bringing the runtime resource management concern to the forefront.*

The focus of this work is therefore on how to manage the parallel computation resources in SoC/HPC-converging heterogeneous parallel architectures, relying on two fundamental pillars: spatial-division multiplexing (SDM) and runtime system reconfiguration. In contrast to Time-Division Multiplexing, the SDM of manycore programmable accelerators in het-

erogeneous architectures enables fine-grain resource allocation, isolation of concurrently-executing software entities, and workload adaptivity through flexible partitioning. At the same time, partitioning of a manycore fabric turns out to be efficient only if "computation and memory quotas" can be dynamically allocated and re-negotiated, and if the fragmentation concern can be limited. This requires runtime modification of the system configuration, which is the second pillar of this thesis. While fostering the above concepts, the specific focus of this thesis is on (i) the validation of the SDM concept, (ii) the development of architecture design methods to support the SDM concept, (iii) and the support for a highly-dynamic SDM concept through the runtime reconfiguration of enabling features in the on-chip communication architecture.

The proposed management approach of many-core programmable accelerators is an enabling technology for the effective deployment of embedded system virtualization. This scenario strengthens the need for an optimized usage of parallel hardware resources to cope with this increased level of resource contention and dynamic application behavior. Partitioning of array fabrics of homogeneous processor cores (that by construction undergo a differentiation of operating conditions and suffer from the regularity-breaking effect of manufacturing faults) and isolation of derived partitions hold promise of pursuing the integration of functionality from separate users/devices onto NoC-based many-core processors, while meeting their potentially heterogeneous requirements. Following this trend, this thesis envisions the extensions of traditional time and space partitioning concepts to parallel hardware platforms to overcome the challenge of using shared (yet modular) resources in applications that are executed concurrently. However, a static partitioning scheme cannot keep up with the increased levels of adaptivity of modern embedded systems, therefore flexible partitioning should be the target. In practice, partitions should be set up or tore down with few or no restrictions, and their size and shape potentially changed at runtime [66]. Whether *such a usage paradigm will be feasible or not depends to a large extent on the capability of reconfiguring at runtime the routing function of the on-chip interconnection network (NoC), serving as the global communication fabric as well as the system integration framework.* The reason becomes apparent when we consider the lower-level implications of partitioning and isolation: allocating a computation partition means to identify a subset of the manycore fabric and to virtualize it as though the computation system were limited to it. Therefore, communication should be limited to the computation tiles belonging to the partition, and there should be potentially no interference between the communication flows of nearby partitions. Similarly, the temporary expansion or restriction of the partition size should be accompanied by the concurrent reconfiguration of the routing paths. Last but not least, the allocation of a new partition implies that the identified subset of the system has a valid routing function (e.g., deadlock-free). *Overall, a resource manager for SDM in manycore computing fabrics*

needs to have hardware assistance in the routing mechanism of the underlying on-chip interconnection network, and in the capability of its safe runtime modification.

In the on-chip domain, runtime reconfiguration of the routing function can be achieved either by non-reconfigurable fault-tolerant routing strategies, which tolerate a limited number of faults [34, 58, 60, 67], or reconfigurable routing mechanisms that allow unlimited changes to the network. Focusing on schemes of the second category, in literature, both static reconfiguration methods and dynamic ones are presented. The former ones consist of draining the network from ongoing packets, modifying routing tables to configure the new routing paths, and finally resuming traffic injection but at the cost of large performance penalties. On the contrary, dynamic reconfiguration techniques succeed in updating routing tables without stopping user traffic, but typically result int unacceptable implementation overhead for an on-chip setting [89, 24, 90, 108, 41, 8, 2]. Although runtime performance is more likely to be preserved, such approaches end up materializing architectures with lower operating speeds (or higher latencies) by construction. Furthermore, current approaches to runtime network configuration suffer from large hardware/software overhead and/or lack of scalability. In general, centralized approaches have the disadvantage that some reconfiguration tasks (e.g., the computation of the new routing function) are performed in software. In contrast, distributed reconfiguration mechanisms, like [84], suffer from sub-optimality of emergency routing solutions, and overly high implementation cost and complexity. This suboptimal scenario borrowed from literature is the starting point of the work in this thesis.

An intensive research effort is currently underway in an attempt to find a suitable design point for chip implementations, including Vicis [48], Immunet [112], Ariadne [4] and other reconfigurable routing frameworks [50, 46, 143]. With respect to these works, the recent adaptation of the Overlapped Static Reconfiguration (OSR) [89] methodology to the tight resource budgets of embedded systems (OSRLite [128]) provided an appealing trade-off between reconfiguration performance and implementation complexity. OSR relies on the principle that if packets with the old routing function are prevented from following packets using the new one, deadlock cannot occur. Enforcing this ordering mechanism is possible even without draining the network from ongoing packets, by propagating a separation token between old and new packets throughout the network. Notwithstanding that, several performance inefficiencies still affect the OSR mechanism, which can be fundamentally identified as the temporary suspension of traffic injection during the reconfiguration transient, the packet blocking behind the self-propagating epoch separation boundary, as well as the network-wide nature of each reconfiguration event. Furthermore, OSR mechanism is centralized (so the manager is on the critical path of the reconfiguration process, needing also a separate control network or virtual channel for communications), and can be triggered from just the root node of the network and not from each node. **Overcoming all the limitations**

and issues of OSRLite, materializing its potentials, thus designing an interconnection fabric supporting features suitable for a virtualized and dynamic environment is one of the goals of the work in this thesis.

In order to tackle the inefficiencies of the mechanism impacting the performance on ongoing communication flows while a reconfiguration takes place, on one side, I propose a set of performance optimizations spanning from simple to more aggressive ones trading performance speedups (approaching latency-insensitive reconfiguration) with a higher implementation cost, and at the same time aiming at speeding up the reconfiguration transient itself. The final outcome, considering the best optimizations, is to have both a limited region involved in the reconfiguration process, and a fully transparent reconfiguration from the performance point of view, thus enabling more flexible scenarios, i.e., partition scheduling and reshaping (restriction and expansion of an existing partition, merging of and splitting into two partitions), avoiding faulty links or switches, powering-off unused or overheated regions, setting up or tearing down reserved paths (hard QoS), this way *creating the support for a highly dynamic many-core environment*. On the other hand, another contribution of my work consists of tackling the overhead of a centralized mechanism, but moving from the following perspective: the synergistic exploitation of routing resources that are already there in many NoC implementations. In a sense, there is an overhead which is increasingly accepted in NoC design, and which is justified by other design goals, which consists of the use of multiple physical networks instead of logic ones. Although this seems to run contrary to much previous work [57, 142] it is actually motivated by how the relative costs of network design change for implementation on a single die [27]. Given this, the solution I proposed is to exploit the existing multiple physical networks to spatially separate resource allocations that may close dependency cycles: whenever a switch port processing old traffic has a routing dependency with a port already migrated to the new epoch, *an escape path is set up into another network plane and taken by the packets*, thus avoiding deadlock. So I developed a reconfiguration methodology around the above basic idea: through an engineered protocol for inter-network tunnel opening and closing, *the system performs a distributed and fast(low-latency) reconfiguration, guaranteeing both scalability and minimum perturbation of background traffic in all NoCs*. Applicability of this innovative mechanisms consists of the transparent execution of system management tasks in the background. For instance, in the presence of large many-core fabrics, small sections of the fabric can be selectively disconnected without suspending system operation, and tested in the background. From an implementation viewpoint, this can be achieved by a distributed test manager having an hardware assistance in the distributed capability of the interconnection network to selectively disconnect its links while re-routing affected traffic elsewhere. The results of this thesis pave the way for this ambitious scenario.

The burden of extracting peak performance from many-core accelerators is not just

on the hardware support, but also on the software layer. Efficient programming abstractions (programming models, compilers, runtime systems) are paramount to achieving efficient exploitation of manycores. In particular, modern embedded applications are increasing in complexity and often expose high degree of parallelism which is irregular in nature and/or dynamically generated. The tasking execution model represents a powerful abstraction to exploit this kind of parallelism, as it enables asynchronous, dynamic creation of units of work in a simple and straightforward manner. Notable examples of this programming paradigm include Cilk [73], Apple Grand Central Dispatch [6], Intel Carbon [79] or the current OpenMP specification [109]. The tasking abstraction provides a powerful conceptual framework to exploit irregular parallelism in target applications, but its practical implementation requires sophisticated runtime system support, which typically implies important space and time overheads. The applicability of the approach is thus often limited to applications exhibiting units of work which are coarse-grained enough to amortize these overheads. While this is often the case for general-purpose systems and associated workloads, things are different when considering embedded many-core accelerators. Minimizing runtime overheads is thus a primary challenge to enjoy the benefits of tasking on these systems. So, in my work, **I start from an optimized runtime environment supporting the OpenMP tasking model for an embedded shared-memory cluster [23], that minimizes the effect of major bottlenecks implied by the execution model, and I customize it to support a dynamic environment, which enables me to test several benchmarks and platform configurations (memory configurations and settings, different shapes of concurrent partitions) in order to validate the resource sharing approach based on Space-Division Multiplexing**.

Moreover, some of the ideas of this thesis were prototyped on FPGA. In fact, **this thesis reports on the prototyping of a 16-core homogeneous programmable multi-core accelerator with runtime-reconfigurable and dynamically virtualizable on-chip network**. The prototyped system validates the NoC capability for boot-time configuration and runtime reconfiguration of the routing function.

Finally, I consider emerging interconnect technologies and their role in re-architecting many-core programmable accelerators as well as heterogeneous parallel architectures as a whole. Indeed, in order to support scaling to future device generations, electronic NoCs will struggle to deliver the required communication performance within tight power budgets. In this respect, evolutionary as well as revolutionary interconnect technologies are currently being considered. On one hand, clockless handshaking materializes GALS systems [25, 122] that completely remove the system clock while reducing idle power to only the leakage power. On the other hand, the technology platform could be changed, by replacing electrical wires with optical links and networks. Although there is today consensus on the fact that optical interconnects can relieve bandwidth density concerns at integrated circuit boundaries, however, when it comes

to the extension of this emerging interconnect technology to on-chip communication as well, such consensus seems to fall apart. The main reason consists of a fundamental lack of compelling cases proving the superior performance and/or energy properties yielded by devices of practical interest, when re-architected around a photonically-integrated communication fabric. So, taking its steps from the consideration that many-core computing platforms are gaining momentum in the high-end embedded computing domain in the form of general-purpose programmable accelerators, in this thesis **I propose the first assessment of optical interconnect technology in the context of these devices, taking care of Optical NoC Interfaces and evaluating the performance and energy implications by means of an accurate benchmarking framework against an aggressively optimized electrical counterpart**. Overall, the above quality metrics paint a promising picture for augmenting GPPAs with optical devices, while clearly pointing to the most important candidate for optimization: static energy reduction through technology evolution as well as gating techniques. In order to minimize and mitigate the most significant contribution to static power dissipation in optical NoCs, **this work relies on the innovative principle of partitioning optical networks-on-chip, and of reusing laser sources across partitions.**

Highlights of key thesis contributions

Overall, the thesis is a comprehensive contribution to the advance in the field of many-core NoC-based system design. Here I report the key thesis innovations:

- extensions of cycle-accurate simulators with modeling capability of heterogeneous parallel computing architectures
- extension of transaction-level simulators with modeling capability of heterogeneous parallel computing architectures
- validation of SDM performance versus TDM one
- evaluation of the role of the partition shape over application performance
- runtime reconfiguration capability of the NoC routing function with transparency property with respect to background traffic
- distributed reconfiguration mechanims enabling selective link disconnection
- FPGA prototyping of an SDM- and NoC-enabled multi-core accelerator with runtime-reconfigurable routing function (through a centralized as well as distributed mechanism)

- augmenting Manycore Programmable Accelerators with Photonic Interconnect Technology
- SDM on top of photonically-enabled many-core accelerators

Thesis Organization

The contributions of this thesis are organized in Introduction, eight technical chapters, in addition to conclusions and future work.

Before presenting my novel contributions, **Chapter 1** first provides the necessary background to understand the work of this thesis. It surveys heterogeneous parallel computer architectures, mentioning various types of accelerators (in particular fostering GPPAs, General-Purpose Programmable Accelerators), then on-chip interconnections used for this kind of systems and, finally, it presents a specific routing mechanism, named Logic-Based Distributed Routing, on top of which the partitioning, isolation and reconfiguration mechanisms discussed in next chapters will be constructed.

Chapter 2 introduces the simulation environments used to simulate the platform considered in this thesis at different abstraction layers, presenting the modifications needed to support the new paradigm of heterogeneous parallel computing architectures, and pointing out benefits and disadvantages using one simulators versus the other.

Chapter 3 makes the point for Space-Division Multiplexing on GPPAs, presenting the evaluation of the approach considering the execution of a batch of real applications and comparing it with the Time-Division Multiplexing case study, thus proving the benefits in a dynamic scenario. Finally, this chapter also proposes an exploration of different partition shapes to underline how they can be crucial for the performance.

Chapter 4 presents both hardware and software support to enable the SDM approach, fostering runtime reconfiguration of the NoC routing function as basic hardware assistance requirement.

Chapter 5, first of all, presents real use case where a runtime reconfiguration of the NoC routing function is needed in the presence of background traffic, then it provides an exploration of reconfiguration approaches, starting from static strategies, and finally comes to the Overlapped-Static reconfiguration idea, which is the baseline mechanism used as the starting point of my work. The chapter continues by presenting the optimizations I implemented to overcome the performance penalties of the original mechanism. The implementation at micro-architecture level of the logic enabling the optimized mechanism is showed.

Chapter 6 presents the implementation of the final optimization of the OSR mechanism, called Tunneled-OSR, which makes it a fully distributed reconfiguration mechanism. The idea is to rely on the synergistic use of multiple physical networks already existing in the chip, materializing a fully distributed, fast and scalable reconfiguration mechanism. The chapter describes the complete mechanism at work, underlying the

advantages compared to the baseline OSR and also to the other mechanisms in literature. Finally, the T-OSR mechanism is evaluated in terms of area overhead, coverage, reconfiguration latency and impact on the ongoing traffic.

Chapter 7 extends the work presented in the previous chapters by reporting on the prototyping of the proposed design methods on a Xilinx Virtex-7 FPGA. The FPGA prototype comprises a large number of components that enable observability, controllability and debugability of the system under test. All these components are described in detail. Last, the chapter illustrates a use-case of the developed mechanism, consisting of the feasibility to perform selective runtime Built-In Self-Testing on system links while an application is running in the background.

Chapter 8 focuses on emerging technologies, in particular on optical interconnections. Here I present the first work on augmenting GPPAs with optical devices and, being the static power consumption an open issue, I try to propose a strategy for partitioning, reusing and selectively tuning the laser sources. Finally, I envision a whole new heterogeneous parallel computing system that relies on photonic interconnection networks to connect the host, main memory, the accelerator and other devices of the system.

In conclusion, I provide the final remarks on the work presented, summarizing the thesis outcomes and envisioning possible future developments of the discussed topics.

Chapter 1

Background

This chapter starts by surveying the evolution of computer architectures in the embedded domain with the advent of heterogeneous Multi-Processors Systems-on-Chip (MPSoCs) and many-core architectures in them, and presents relevant examples of such architectures, fostering in particular General-Purpose Programmable Accelerators (GPPAs). Furthermore, it introduces the proper background concerning the on-chip interconnection infrastructure represented by Networks-on-Chip (NoCs) and, finally, it presents the routing mechanism adopted by the NoC of this work, the Logic-Based Distributed Routing (LBDR), that stays at the core of the reconfiguration mechanism that will be addressed in Chapter 5. Overall, the chapter presents the scenario in which this thesis is inserted, highlighting and pointing out the issues and challenges for the designers of many-core programmable accelerator architectures that will be tackled in this thesis.

1.1 Heterogeneous parallel computer architectures

1.1.1 The end of Dennard scaling and the switch to multicores

Managing the power dissipation of current computer systems is a *Grand Challenge* problem. Power affects computer systems at all scales: from the computational ca-

Device or Circuit Parameter	**Scaling Factor**
Device dimension T_{ox}, L, W	1/k
Doping concentration Na	k
Voltage V	1/k
Current I	1/k
Capacitance eA/t	1/k
Delay time per circuit VC/I	1/k
Power dissipation per circuit VI	$1/k^2$
Power density	1

Table 1.1: Dennar scaling rules.

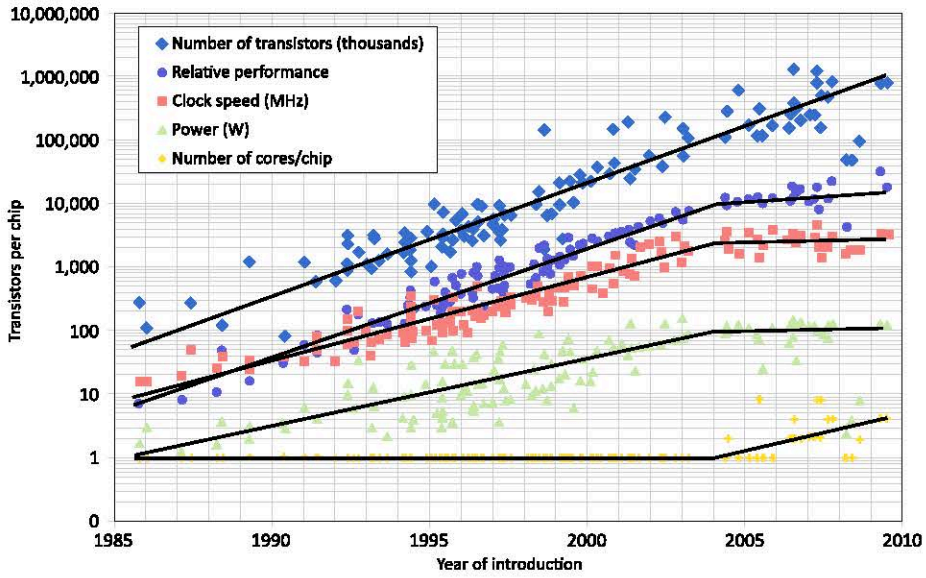


Figure 1.1: Moore's Law and corollaries. Data shows scaling trends, with clear shifts in trend lines at roughly 2004.

pacity of our large-scale data centers, to the processing performance of our high-end servers [21], and the battery life and performance of our mobile devices [125]. To today's computer architects, the emergence of power as a grand challenge [72] may seem like a relatively recent issue, but the reality is that the very earliest computer systems faced vexing power challenges. Over the decades that have followed, computer systems benefited from technology refinements that improved circuit performance, cost, and power. Gordon Moore's predictions of technology scaling linked integration levels (transistors per chip) to production cost [97]. For many years, these cost-driven integration improvements also translated quite naturally into performance improvements. Nearly concurrently, Dennard articulated a scaling principle that would lower supply voltages as transistors became smaller [39]. It is Dennard scaling that enables the transistor increases predicted by Moore's Law to be parlayed into performance improvements and power savings. Despite the benefits of Dennard Scaling, the power dissipation of integrated systems has spiked before.

So far, the power problem as we have faced it over the past decade is largely due to two effects. First, it is primarily a consequence of the end of the Dennard scaling rules (Table 1.1) that parlayed Moore's Law into performance and power benefits for more than three decades. Dennard scaling rests on several key shifts that can be made when transitioning to a smaller feature size. For example, smaller transistors can switch quickly at lower supply voltages, resulting in more power efficient circuits and keeping the power density constant. But supply voltages cannot drop forever. A breakdown of Dennard scaling occurred when voltages dropped low enough to make static power consumption a major issue. Second, even if Dennard scaling had continued on-track, our propensity for faster clock rates and larger die sizes meant that each generation's power dissipation was scaling up faster than Dennard effects were able to hold it in

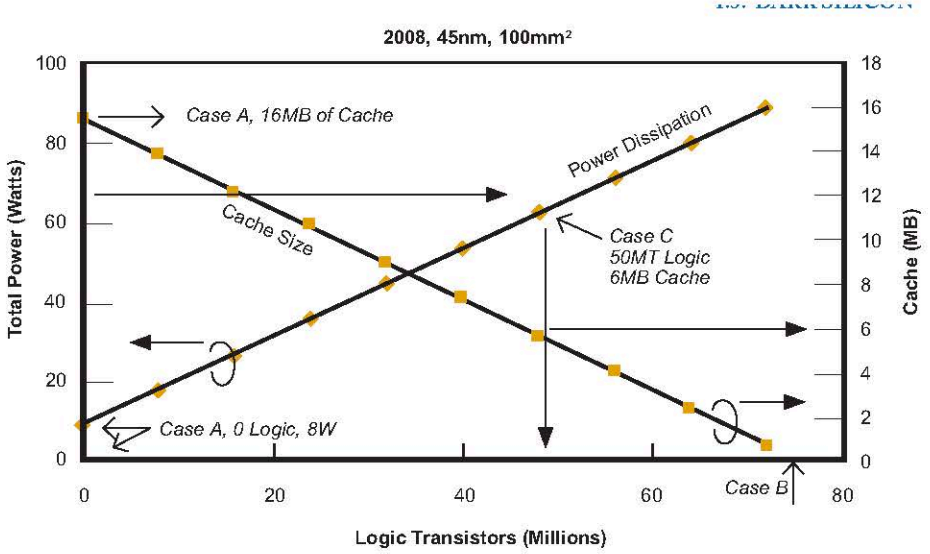
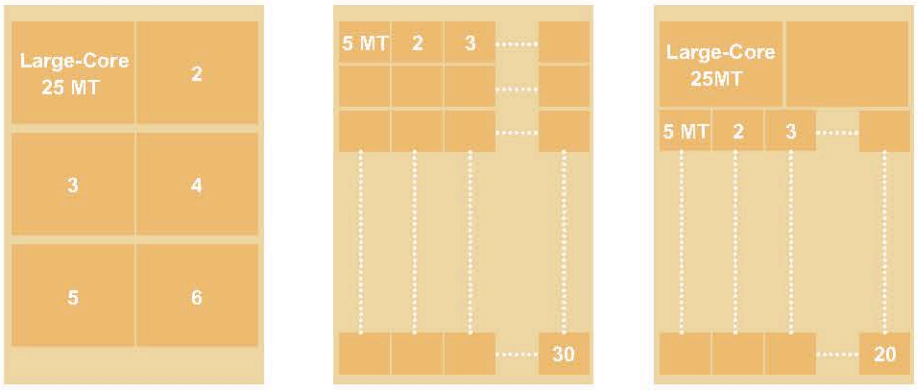


Figure 1.2: A range of implementation options trading off processor area devoted to cache, and resulting power tradeoffs.

check.

In particular, a key advantage of CMOS technology for many years was its lack of static power dissipation. That is, the complementary p- and n-networks in CMOS gates (theoretically) do not allow any path from supply voltage to ground, consuming power only when switching (dynamic power and some glitch power). Static power consumption was therefore safely ignored at the architectural level. However, when technology scaling broke the 100 nm barrier, transistors showed their analog nature: they are never truly off, and this allows sub-threshold leakage currents to flow. Worse, sub-threshold leakage currents are exponential to threshold voltage reductions. In Dennard scaling, the major mechanism to improve power efficiency is the reduction of the supply voltage, which assumes a reduction of the threshold voltage (since the difference of the two voltages dictates transistor switching speed). The rise of static power brought a complete stop to the power benefits architects took for granted for many technology generations. One-time reductions of static power consumption are possible but the trends remain the same with scaling. For example, current technologies employ multi-gate transistors also known as FinFETs. In these transistors, a fin between the drain and source is "wrapped" by silicon in a non-planar fashion to enable the gate to better encompass the channel, which reduces leakage. As one particular example, Intel switched to 3D or tri-gate transistors in their 22 nm technology [18]. While this change provided a step reduction in leakage going from 32 nm to 22 nm, further reductions will be limited in subsequent scalings.

In the 1980s and early 1990s (the heyday of Moores Law scaling), architects primarily improved performance by exploiting instruction-level parallelism (ILP)parallelism found in the dynamic instruction stream during execution of a program. To discover and exploit this parallelism, significant hardware resources were thrown at the problem. Sophisticated techniques such as out-of-order execution, branch prediction and



Large-Core Homogeneous	
Large-core throughput	1
Small-core throughput	
Total throughput	6

(a)

Small-Core Homogeneous	
Large-core throughput	
Small-core throughput	Pollack's Rule $(5/25)^{0.5}$=0.45
Total throughput	13

(b)

Small-Core Heterogeneous	
Large-core throughput	1
Small-core throughput	Pollack's Rule $(5/25)^{0.5}$=0.45
Total throughput	11

(c)

Figure 1.3: Enhancing throughput while maintaining power envelope.

speculative execution, register renaming, memory dependence prediction, among others, were developed for this purpose. These approaches can be highly complex and do not scale well. This results in diminishing performance returns (number of instructions executed in parallel) for increasing hardware investments. Dynamic power scales even worse, deteriorating the power efficiency of such approaches. In fact, as illustrated in Figure 1.1, power dissipation scales as performance raised to the 1.73 power for the typical ILP core: a Pentium 4 is about six times the performance of an i486 at 23 times the power [62]!

The **shift to multicore architectures started in 2004 as a reaction to this looming problem of increased power consumption and power density**. Effectively, we abandoned frequency scaling (which resulted in significant increases in both dynamic and static power consumption) in favor of laying down more cores on the same chip. This dramatic shift to chip multiprocessors (CMPs) in the past decade is a response to the power wall and the end of Dennard scaling. In particular, Borkar et al. [19] walks through an example for 45 nm technology that is still instructive. For a 45 nm chip with 150M transistors, Figure 1.2 shows a range of possible options for implementing the processor. To abide by the total limit of 150M transistors, one can use more logic transistors (x-axis) in opposition with fewer cache transistors. The resulting power dissipation is shown on the left y-axis, and the resulting cache capacity is shown on the right y-axis. As one increases the number of transistors devoted to logic, the power dissipation increases (because caches are "cool" from a power standpoint). Pollack's rule [111] argues that microprocessor performance scales roughly as the square root of its complexity, where the logic transistor count is often used as a proxy to

quantify complexity. From these rules of thumb, multiple parallel cores essentially always beat monolithic single cores on power-normalized performance. For example, Figure 1.3 shows three approaches that use parallel cores to enhance throughput while maintaining the same power envelope. Case (a) (far left of Figure 1.2) harnesses 6-way parallelism at a fairly coarse-grain, and is out-performed by Case (b) (far right), which is more aggressively parallel, when enough thread/task level parallelism exist in the workload. Case (c) represents heterogeneous parallelism, in which two large cores are mixed with several small ones, to good effect. An even more heterogeneous approach would be to include some specialized accelerators, which use very few transistors or chip area, but have large performance and power benefits when applicable. Overall, these examples and rules of thumb begin to explain the direction that industry has taken: a quick and aggressive adoption of medium-scale, on-chip parallelism. Parallelism helps with the impending power wall, by offering a path to high performance that does not rely on high clock rates and high supply voltages. Parallelism-particularly heterogeneous parallelism also helps with the so-called *utilization wall* [138] and the *Dark Silicon* problem [44].

1.1.2 Dark Silicon, the utilization wall and the rise of the heterogeneous parallelism

Our inability to scale a single core to further exploit ILP in a power efficient manner turned computer architecture toward exploring alternative kinds of parallelism (task/thread parallelism, data parallelism). Multi-core and many-core architectures are designed for explicit parallelism, and recalling Figures 1.2 and 1.3, they offer greater performance-per-watt than large monolithic approaches. Unfortunately, even homogeneous CMPs will not be sufficient to solve the power problem for more than a few more generations [96]. This road is also faced with the same problems as with the single core architecture: **we are unable to efficiently extract sufficient speedup from parallel programs** (Amdahls Law [5]). Furthermore, some postulate a near future in which the number of dynamically active transistors on a die may be greatly constrained, forming the "utilization wall" [44]. The concept of the utilization wall is that power envelopes may lead to scenarios in which few (perhaps 20% or less) of a chip's transistors can be "on" at a time. The argument for this possible future is exemplified in Figure 1.4. If transistor density increases in line with Moore's Law, a 45 nm chip will shrink to one-quarter the size at 22 nm in 2014, and one-sixteenth at 11 nm in 2020. Using the ITRS roadmap for scaling, the smaller chips would be more efficient, drawing the same power at 22 nm even though the peak frequency increases by a factor of 1.6, and 40% less at 11 nm with 2.4 peak clock speed. But, if we maintain the same chip area, we can pack four times the number of transistors at 22 nm and 16 times at 11 nm. For the same initial power budget this means that only 25% of the transistors can be powered-up in 22 nm, and 10% in 11 nm.

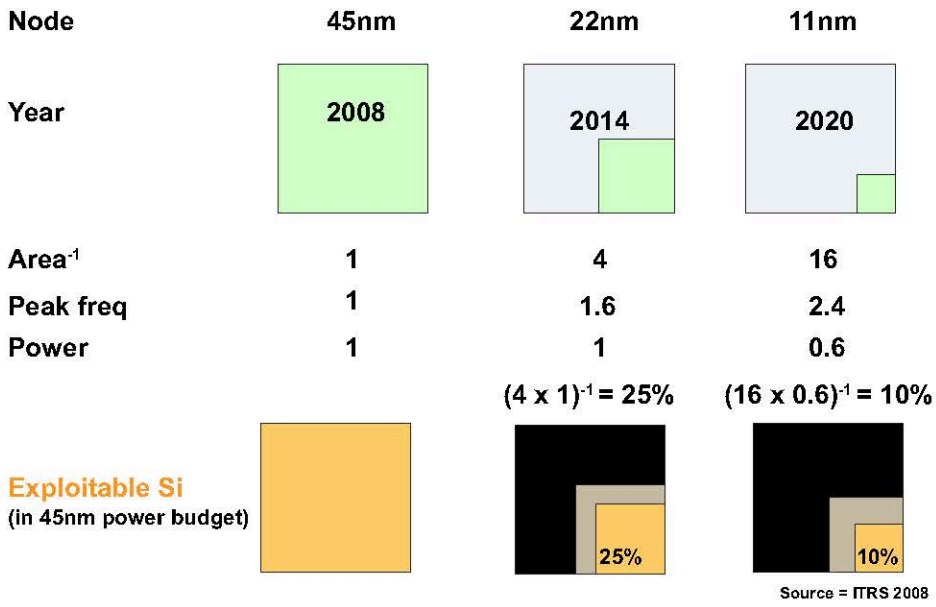


Figure 1.4: A depiction of Dark Silicon trends as seen by ARM.

The answer to the challenge of the utilization wall is **the rise of heterogeneous architectures** where some general-purpose cores are augmented by other cores of different microarchitectures or even specialized accelerators that offer outstanding performance-per-watt by being very lean hardware designs for a particular computational purpose. The approach of heterogeneous parallelism with specialized accelerators is well-suited for "Dark Silicon" scenarios. A large number of accelerators can be built on the same chip to be woken up only when needed. **These heterogeneous architectures are fast becoming the dominant paradigm after multicore**.

In fact, we do not need to speculate about future heterogeneity, as heterogeneous parallel computing is here today. If we examine the product lines from the major chip manufacturers we see that they now have separate multicore x86 architectures targeted at high performance (2-12 cores, 100W, 100 GFLOPs) and low power (1-2 cores, 10W, 10 GFLOPs), and are integrating data-parallel graphics cores onto their CPU devices with distinct programming and memory models [22]. In the embedded world, there are a range of cores at different performance/efficiency points (1-8 cores, 2W, 10 GMIPS) with a range of programmable graphics cores [7]. NVIDIA, Samsung, and Qualcomm all sell heterogeneous ARM/GPU processors with many fixed-function accelerator blocks for the smart phone market [27, 100, 113], and there are multiple start-ups with 64-100 core devices [31] for networking and telecom. This present-day processor heterogeneity forces system and software designers to address the difficult optimization challenge of choosing the right processor (both at design time and runtime) for their products power and performance requirements. Beyond simply considering heterogeneity in the types of instruction-programmable cores on-chip, the field is also increasingly considering approaches involving specialized accelerators that may not be instruction-programmable, and that are tuned to particular application kernels of in-

terest. Specialized accelerators are a particularly natural response to the Dark Silicon scenario in which we may have many more transistors than what we can power up at once. With these "dormant" transistors we could build a plethora of specialized accelerators that cost little either in terms of "active" area or power when not in use. The expectations of generality -all transistors must be useful to all applications- shift considerably in a Dark Silicon world, and what once might have been viewed as "niche" accelerators become a viable method for achieving performance goals under dramatic power constraints.

1.1.3 Other issues

Overall, computer systems have reached an intriguing inflection point. For architects, power has been a fundamental design constraint for well over a decade now, with the initial reaction being fairly localized, per-module efforts to improve power efficiency. These efforts have been the equivalent of turning lights off in unused rooms of one's house, i.e. very sensible, but insufficient in leverage to dramatically change the overall power-performance design landscape. The second wave in power-aware computing has been the recent and seismic shift toward on-chip parallelism.

Software and Programmability Issues: in many ways, the hardware industry's shift toward parallelism has occurred much faster than the abilities of the software and systems designers to react. We know how to build CMPs, and we must build them to keep Moore's Law rolling along. But we do not yet know how to program them efficiently both in terms of software development time and in terms of getting the best power-performance outcomes from them. Furthermore, the shift toward on-chip accelerators offers even greater programmability challenges. Finally, there are a host of programmability concerns that emanate from the basic goal of elevating power to a first-class design constraint alongside performance. For example, from a power perspective, information on the relative criticality of different communication or computation operations may be very useful, but current programming models offer few abstractions or constructs to help programmers manage this.

Reliability Tradeoffs: Until now, power-performance tradeoffs have been viewed by architects as a two-dimensional optimization landscape. There is emerging research, however, on the possibilities of three-dimensional optimization scenarios in which power, performance, and reliability are traded off against each other. Such tradeoffs are already frequently considered at the device and circuit level, but in ways that enable the architecture and software levels to be shielded from their effects; abstraction layers give the impression of perfect reliability even when device or circuit tricks are being employed.

Intuitively, there seem to be rich opportunities for raising the abstraction layer at which reliability, energy, and performance are traded off, in order to enable architects to exploit them as well. For example, operating with smaller supply voltage noise

margins (by lowering supply voltage) may offer high leverage on power savings, at the expense of possible calculation or storage errors. Likewise, reducing or eliminating parity/checksum protection on memory or interconnect also seems to offer some intuitive power/reliability tradeoff possibility. The key research questions in this space, however, focus on whether the power/performance benefits achievable through some approaches are large enough to be appealing given the serious impact of relaxing reliability guarantees to software.

Beyond the Processor Core: Much of the "first wave" of power optimizations focused on the CPU itself, because the most serious thermal and power density concerns were experienced there. And even more specifically, most optimizations were focused on the CPU's processor cores and cache memories. As we look, however, to future power issues and ideas, there is a growing need to look beyond the processor core. Data communications and on-chip interconnect will play an increasingly important role in power dissipation, especially since the adoption of parallelism has led to much higher levels of data motion and inter-processor communication in many cases. One also needs to consider the energy issues related to the memory hierarchy as well. Chapter 4 covers these topics in this book, but considerable future work in this area is likely to be forthcoming.

1.1.4 Many-core architectures

Modern homogeneous and heterogeneous multicore and many-core architectures are now part of the high-end and mainstream computing scene and can offer impressive performance for many applications. This architecture trend has been driven by the need to reduce power consumption, increase processor utilization, and deal with the memory-processor speed gap. However, the complexity of these new architectures has created several programming challenges, and achieving performance on these systems is often a difficult task. Several variants of many-core architectures have been designed and are in use for years now. As a matter of fact, since the mid 2000s we observed the integration of an increasing number of cores onto a single integrated circuit die, known as a Chip Multi-Processor (CMP) or Multi-Processor System-on-Chip (MPSoC), or onto multiple dies in a single chip package. Manufacturers still leverage Moore's Law (doubling of the number of transistors on chip every 18 months), but business as usual is not an option anymore: scaling performance by increasing clock frequency and instruction throughput of single cores, the trend for electronic systems in the last 30 years, has proved to be not viable anymore [3, 52, 19]. As a consequence, computing systems moved to multi-core1 designs and subsequently, thanks to the integration density, to the many-core era where energy-efficient performance scaling is achieved by exploiting large-scale parallelism, rather than speeding up the single processing units [53, 19, 3, 81]. Such trend can be found in a wide spectrum of platforms, ranging from general purpose computing, high-performance to the embedded world.

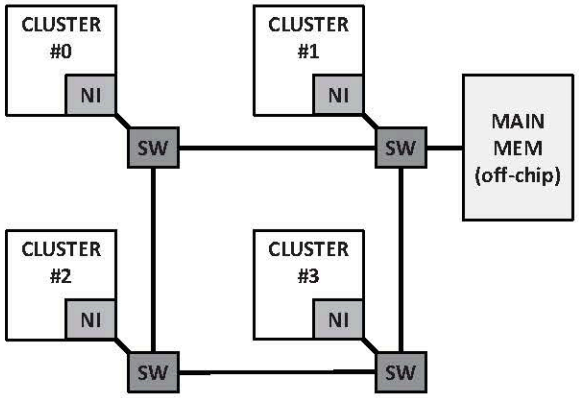


Figure 1.5: Clustered many-core architecture organized in a 2x2 mesh and off-chip main-memory.

In the general purpose domain we observed the first multi-core processors al- most a decade ago. Intel core duo [59] and Sony-Toshiba-IBM (STI) Cell Broad- band Engine [74] are notable examples of this paradigm shift. The trend did not stop and nowadays we have in this segment many-core examples such as the TILE-Gx8072 processor, comprising seventy-two cores operating at frequencies up to 1.2 GHz [32]. Instead, when performance is the primary requisite of the application domain, we can cite several notable architectures such as Larrabee [121] for visual computing, the research microprocessors Intels SCC [68] and Tera- scale project [135] and, more recently, Intels Xeon Phi [64]. In the embedded world, we are observing today a proliferation of many-core heterogeneous plat- forms. The so-called asymmetric of heterogeneous design features many small, energy-efficient cores integrated with a full-blown processor. Its is emerging as the main trend in the embedded domain, since it represents the most flexible and efficient design paradigm. Notable examples of such architectures are the AMD Accelerated Processing Units [22], Nvidia TEGRA family [100], STMicroelectronics P2012/STHORM [95] or Kalray's many-core processors [75].

The work presented in this thesis is focused on the embedded domain where, more than in other areas, modern high-end applications are asking for increasingly stringent and irreconcilable requirements. An outstanding example consist of the mobile market. As highlighted in [134], the digital workload of a smartphone (all control, data and signal processing) amounts to nearly 100 Giga Operations Per Second (GOPS) with a power-budget of 1 Watt. Moreover, workload requirements increase at a steady rate, roughly by an order of magnitude every 5 years.

From the architectural point of view, with the evolution from tens of cores to the current integration capabilities in the order of hundreds, the most promising architectural choice for many-core embedded systems is clustering. In a clustered platform, processing cores are grouped into small- medium-sized clusters (i.e. few tens), which are highly optimized for performance and throughput. Clusters are the basic building blocks of the architecture, and scaling to many-core is obtained by the replication

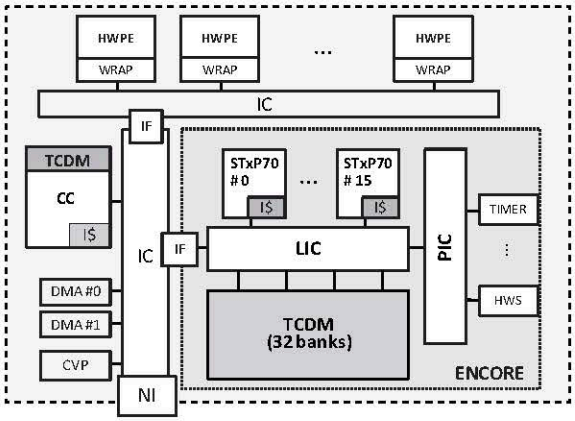


Figure 1.6: Overview (simplified) of P2012/STHORM cluster architecture.

and global interconnection through a scalable medium such as a **Network-on-Chip (NoC)**, which will be discussed later in this chapter. Figure 1.5 shows a reference clustered many-core architecture, organized in 4 clusters with a 4x4 mesh-like NoC for global interconnection. Next section reports some representative examples of recent architectures with a focus at the cluster level.

1.1.5 Cluster architectures: relevant examples

The cluster architecture considered in this work is representative of a consolidated trend of embedded many-core design. Few notable examples are described, highlighting the most relevant characteristics of such architectures.

ST Microelectronics P2012/STHORM

Platform 2012 (P2012), also known as STHORM [95], is a low-power programmable many-core accelerator for the embedded domain designed by ST Microelectronics [1]. The P2012 project targets next-generation data-intensive embedded applications such as multi-modal sensor fusion, image understanding, mobile augmented reality [15]. The computing fabric is highly modular being structured in clusters of cores, connected through a Globally Asynchronous Network-on-Chip (GANoC) and featuring a shared memory space among all the cores. Each cluster is internally synchronous (one frequency domain) while at the global level the system follows the GALS (Globally Asynchronous Locally Synchronous) paradigm. In Figure 1.6 is shown a simplified block scheme of the internal structure of a single cluster. Each cluster is composed of a Cluster Controller (CC) and a multi-core computing engine, named ENCore, made of 16 processing elements. Each core is a proprietary 32-bit RISC core (STxP70-V4) featuring a floating point unit, a private instruction cache and no data cache.

Processors are interconnected through a low-latency high-bandwidth logarithmic interconnect and communicate through a fast multi-banked, multi-ported tightly-coupled data memory (TCDM). The number of memory ports in the TCDM is equal to the

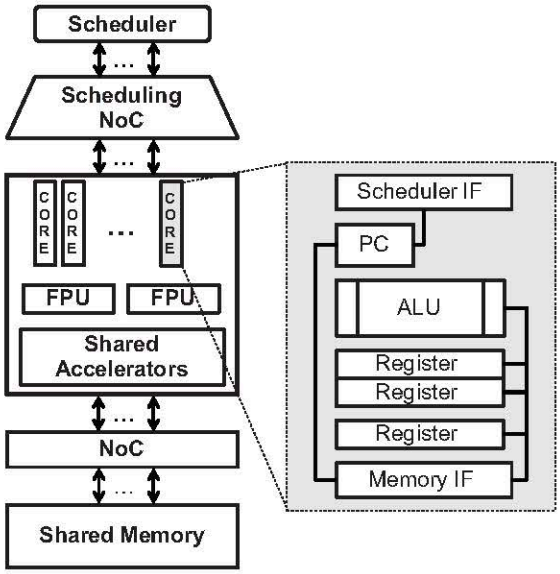


Figure 1.7: Plurality HAL architecture overview.

number of banks to allow concurrent accesses to different banks. Conflict-free TCDM accesses are performed with a two-cycles latency. The logarithmic interconnect consists of fully combinatorial Mesh-of-Trees (MoT) interconnection network. Data routing is based on address decoding: a first-stage checks if the requested address falls within the TCDM address range or has to be directed off-cluster. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. If no bank conflicts arise, data routing is done in parallel for each core. In case of conflicting requests, a round-robin based scheduler coordinates accesses to memory banks in a fair manner. Banking conflicts result in higher latency, depending on the number of concurrent conflicting accesses. Each cluster is equipped with a Hardware Synchronizer (HWS) which provides low-level services such as semaphores, barriers, and event propagation support, two DMA engines, and a Clock Variability and Power (CVP) module. The cluster template can be enhanced with application specific hardware processing elements (HWPEs), to accelerate key functionalities in hardware. They are interconnected to the ENCore with an asynchronous local interconnect (LIC). The first release of P2012 (STHORM) features 4 homogeneous clusters for a total of 69 cores and a software stack based on two programming models, namely a component-based Native Programming Model (NPM) and OpenCL-based [127] (named CLAM - CL Above Many-Cores) while OpenMP [33] support is under development.

Plurality HAL - Hypercore Architecture Line

Plurality Hypercore [110] is an energy efficient general-purpose machine made of several RISC processors. The number of processors can range from 16 up to 256 according to the processor model. Figure 1.7 shows the overall architecture and the single pro-

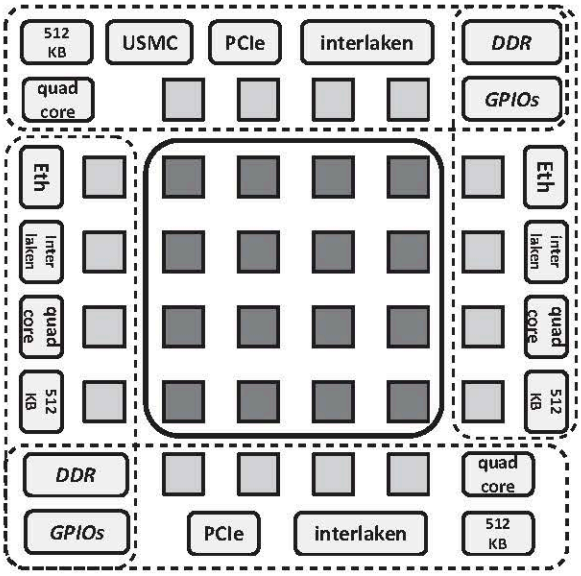


Figure 1.8: Overview (simplified) of Kalray MPPA architecture.

cessor structure, which is designed with the goal of simplicity and efficiency in mind (no I/D caches nor private memory, no branch speculation) to save power and area. The memory system (i.e., I/D caches, off-chip main memory) is shared and processors access it through a high-performance logarithmic interconnect, equivalent to the interconnection described for STHORM. Processors share one or more Floating Point Units, and one or more shared hardware accelerators can be embedded in the design. This platform can be programmed with a task-oriented programming model, where the so-called "agents" are specified with a proprietary language. Tasks are efficiently dispatched by a scheduler/synchronizer called Central Synchronizer Unit (CSU), which also ensures workload balancing.

Kalray MPPA manycore

Kalray Multi Purpose Processor Array (MPPA) [75] is a family of low-power many-core programmable processors for high-performance embedded systems. The first product of the family, MPPA-256, deploys 256 general-purpose cores grouped into 16 tightly-coupled clusters using a 28nm manufacturing process technology. The MPPA manycore chip family scales from 256 to 1024 cores with a performance of 500 Giga operations per second to more than 2 Tera operations per second with typical 5W power consumption. Global communication among the clusters is based on a Network-on-Chip. A simplified version of the architecture is shown in Figure 1.8.

Each core is a proprietary 32-bit ISA processor with private instruction and data caches. Each cluster has a 2MB shared data memory for local processors communication and a full-crossbar. Clusters are arranged in a 4x4 mesh and four I/O clusters provide off-chip connectivity through PCI (North and South) or Ethernet (West and East). Every I/O cluster has a four-cores processing unit, and N/S clusters deploy

each a DDR controller to a 4GB external memory. The platform acts as an accelerator for an x86-based host, connected via PCI to the North I/O cluster. Accelerator clusters run a lightweight operative system named NodeOS [98], while I/O clusters run an instance of RTEMS [30].

1.1.6 Many-core accelerators

Energy efficiency in terms of OPS/Watt is the most influencing factor for an embedded system design, with the future target to provide 100 GOPS within the power envelope of 1W. Heterogeneity is used as a key tool to increase the energy efficiency of a MPSoC and sustain the disruptive computing power delivered by such systems, by staying within an always shrinking market-driven power budget. Various design schemes are available today: systems composed by a combination of powerful and energy efficient cores [87], and also designs exploiting various types of specialized or general purpose parallel accelerators [98, 71]. The combination of different types of computing units allows the system to adapt to different workloads, providing computing power when running complex tasks or running on the more energy efficient cores when the performance is not required. And finally offloading computation to an accelerator, when high parallel computing capabilities are required.

A state-of-the-art heterogeneous MPSoC is shown in Figure 1.9, which is the NVidia Tegra-K1. It is immediately visible in the bottom of the image that a multi-core processor (Host processor), composed by four powerful cores and one smaller and more energy efficient, is flanked by a many-core embedded GPU acting as a parallel co-processor (Accelerator). The GPU is placed exactly above the host processor. However, even if MPSoCs are designed to deliver high computing performance with a low power consumption, achieving this goals is not a trivial task.

Such new design paradigm opens the door to several challenges. In this thesis two of the many possible are addressed: Hardware design space exploration complexity

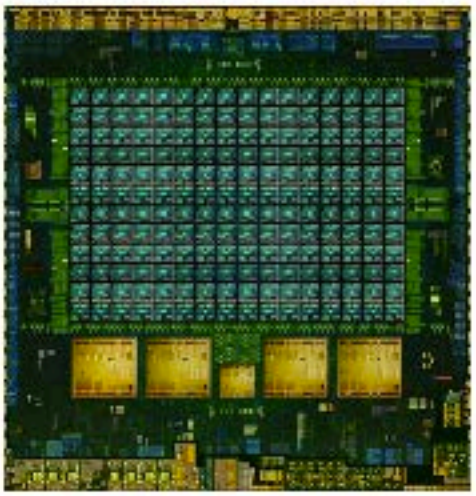


Figure 1.9: NVidia Tegra K1 floorplan.

and **Performance scalability**.

Hardware design space exploration complexity

Hardware designers have been relying for years on virtual platforms as a tool to reduce the time to market of a chip design, forecast performance and power consumption and also to enable early software development before the actual hard- ware is available. However, the complexity of modern systems forces hardware designers to cope with a huge design space to be explored to find the best trade-off among energy consumption, area and performance delivered. Several simulation frameworks are available today off-the-shelf but almost all of them suffer of three main problems, which make them not suitable to model a complex MPSoC:

- *Lack of models for deep microarchitectural components*: hardware designs with more than hundreds of computing units use various architectural components, to allow efficient and scalable communication between cores (e.g. Networks-On-Chip) and complex memory hierarchies. Such components have to be modeled at the micro-architectural level to enable accurate power estimations and performance measurements.

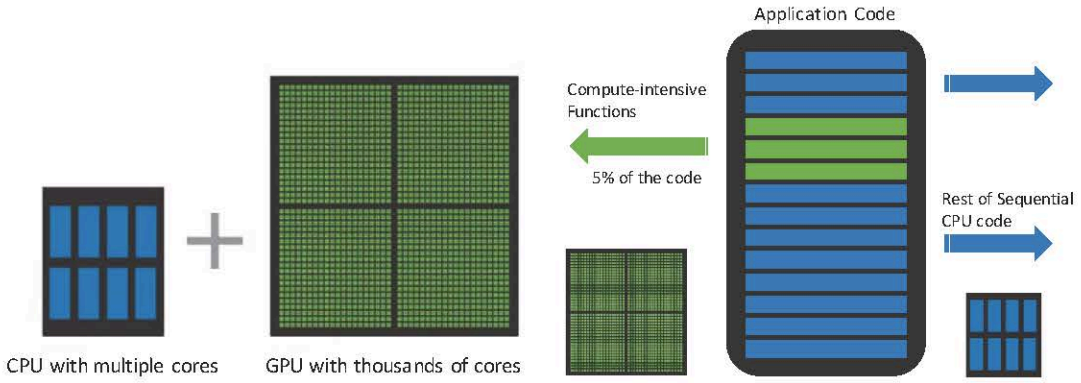
- *Lack of support for Full System simulation*: modern MPSoCs are composed by a Host processor and one or more accelerators. The host processor is usually in charge of executing an operating system (e.g. Linux), while the accelerators are used as a co-processors to speedup the execution of computationally heavy tasks. In this scenario the interaction between host processor and accelerators, being it a memory transfer or a synchronization, may have a significant effect on applications performance. Virtual platforms have to accurately model such interactions to enable precise application profiling.

- *Sequential simulation*: most of the available modeling tools are relying on a sequential execution model, in which all components of the design are simulated in sequence by a single application thread. In the near future MPSoCs will feature thousand of computing units, and such a modeling technique will make the simulation time of a reasonable application to be to slow for practical use.

Performance scalability

Even if Pollack's rule states that the increase of performance is proportional to the square root of the increase in complexity of a system, achieving such performance is not a trivial task. Programmers seeking for applications performance are thus obliged to know architectural specific details, and apply complex programming patterns to adapt their applications to the specific target hardware.

One of the most performance affective problems is the memory wall, which is due to a huge gap in the technological advance between CPU and memory speed. An efficient utilization of the memory hierarchy is thus critical for performance, especially in a system with thousand of cores where the required memory bandwidth can be extremely high. However due to some design choices taken for the sake of area and power consumption reduction, the hardware is not always able to automatically fill the gap of memory latency. One example is the choice to substitute data caches with scratchpad memories, because the latter with the same size in bytes occupies 30% less area than a cache. Programmers can not rely anymore on data caches to hide the external memory access latency, and try to overlap as much as possible computation with communication. One common programming pattern is DMA double buffering, in which computation is divided in chunks and while the actual is computed the next one is read from external memory. Such type of design choice forces application programmers to know deep hardware related features to boost the performance of their code, leading often to complex and error-prone programming. A software runtime is presented in this thesis which automatically handles external-memory-to-scratchpad memory transfers, without any intervention of the programmer. Another design related challenge is memory sharing between host processor and many-core accelerator. A general purpose processor, when running an operating system, uses a virtual memory abstraction to handle the whole physical memory available on a platform. This is possible thanks to a Memory Management Unit (MMU), which is in charge to translate any virtual address to its equivalent in physical memory. State-of-the-art many-core accelerators are often not equipped with an MMU, meaning that only physical memory addresses can be used from within the accelerator. In a typical application the Host processor acting as a master is in charge of handling the main application flow, and input/output data buffers shared with the accelerator are created under the virtual memory abstraction. Since most many-core accelerators are only able to directly access physical memory, input/output buffers have to be copied into a memory region which is not handled under virtual memory, before being accessible from the accelerator. Those memory copies affect the overall performance of an application, limiting also the usability of the accelerator itself for real applications. An example is system virtualization, which has recently been enabled on embedded systems thanks to the advent of hardware support for virtualization in ARM cores. In a virtualized system several instances of an operating system (Guest) run at the same time on the same hardware, and all peripherals need to have a virtual counterpart to be visible by all guests. In this context several memory virtualization layers are involved, and a many-core accelerator without an MMU can not be easily virtualized and used by all the guests running on a system.



(a) Number of cores. (b) How GPU's acceleration works.

Figure 1.10: CPU vs. GPU.

1.1.7 Accelerator types: introducing General-Purpose Programmable Accelerators

This section seeks to explore many-core systems examples, as well as stand-alone systems with large numbers of cores like *Graphic-Processing Units* (GPUs) and various types of accelerators in the context of general-purpose parallel computing; this can also include hybrid and heterogeneous systems with different types of multicore processors.

Architectural heterogeneity and many-cores are the design paradigm for SoCs in the embedded computing domain, driven by flexibility, performance and cost constraints of demanding modern applications. SoC architecture and heterogeneity clearly provide a wider power/performance scaling, combining host CPUs along with massively parallel co-processors or accelerators. The first type of accelerators are the GPUs, also occasionally called *Visual Processing Units* (VPUs), a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. Accelerated computing offers unprecedented application performance by *offloading* compute-intensive portions of the application to the accelerator, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster. A simple way to understand the difference between a CPU and for example a GPU is to compare how they process tasks and furthermore a CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed to handle multiple tasks simultaneously and to process parallel workloads efficiently, as shown in Figures 1.10(a) and 1.10(b).

As shown in Figure 1.11 there are a lot of heterogeneous SoCs:

- *Shared-memory Multi-Processors* (SMP): it provides coarse-grain parallelism, strong memory consistency models, huge flexibility and is suited for few large independent threads.

- *Throughput computing*: it present leading core count, it can provide *Single Pro-*

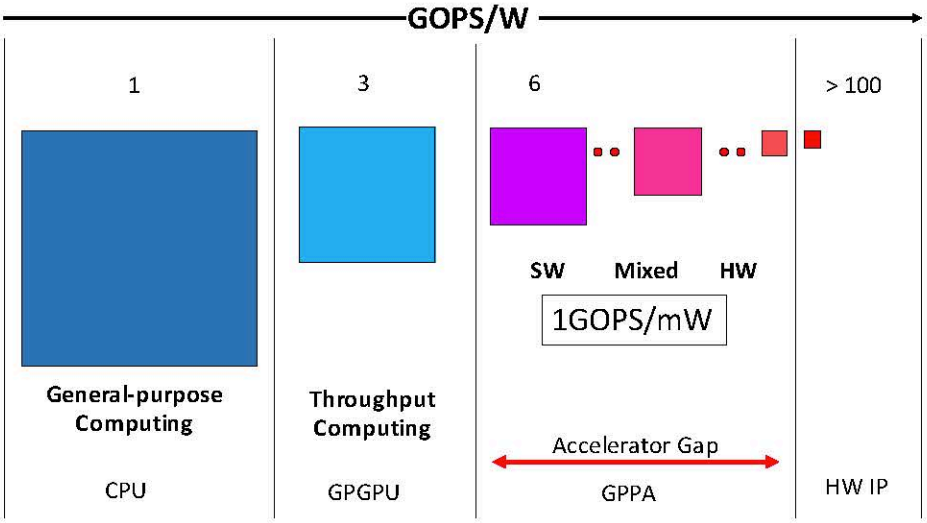


Figure 1.11: Heterogeneous SoCs and types of parallel computing.

gram- or *Single Instruction- Multiple Data* (SPMD or SIMD) parallel processing and is betfitting with thousands of small threads (better if almost identical) to expose massive hardware multithreading. This is the type of computing used in *General-Purpose Graphics Processing Units* (GP-GPUs), that relate the use of GPUs, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the Central Processing Unit (CPU).

- *Harware IPs*: they present the highest GOPS/W but have not flexibility and lower yield.

- *General-Purpose Programmable Accelerators*(GPPAs): They are cluster-based many-core and customizable accelerators that can support many, truly independent parallel computations based also on Multiple Instruction Multiple Data (MIMD), with parallel threads that are heavily dependent on local data content and present branch divergence.

This thesis will focus on these latters, that are holding the potential of bridging the gap between the energy efficiency (GOPS/W) of hardwired hardware accelerators and the computational power delivered by throughput computing. In contrast to graphics processing units, GPPAs present a more balanced trade-off between latency and throughput requirements, and a different usage model of the many-core device (that makes them suitable for applicability of optical interconnect technology, as we will see in Chapter 8). In the latest heterogeneous Systems-on-Chip (SoC), and even more in future ones, the quest for processing specialization to deliver ultra-high performance acceleration at reduced energy cost does not necessarily imply hundreds of dedicated hardware accelerators [88]. There are at least a couple of reasons against that approach. On one hand, the performance of a

specialized processing engine may in many cases be equally achieved by the parallel computation of programmable processing units [45]. Execution efficiency can thus be achieved without sacrificing programmability. On the other hand, the trend towards simplifying the microarchitecture design of system building blocks is becoming increasingly strong. Only a replication-driven approach ultimately pays off in terms of design productivity. There are two main architecture families that might in principle suit the need for many-core programmable accelerators: the former one consists of GP-GPUs [99] and is optimized for the single instruction multiple data/thread execution model (SIMD/SIMT), while the latter one relies on the multiple instruction multiple data (MIMD) model (although not limited to it).

MIMD programmable accelerators do not implement GPU-like data-parallel cores, with common fetch/decode phases which imply performance loss when parallel cores execute out of lock-step mode. They are rather independent RISC cores, well suited to execute both SIMD and MIMD types of parallelism. When coupled with a hierarchical organization into clusters like [110, 95, 61], such accelerators lend themselves to powerful programming abstractions such as nested parallelism [93]. In practice, a first level of parallelism can be used to distribute coarse-grained tasks to clusters, and one or more inner levels of fine-grained (e.g., loop-level) parallelism can be distributed to processors within a cluster.

Furthermore, one reason for the growing interest in many-core accelerators in the embedded computing domain is that there is a rapidly growing demand for a new type of interactions between the user and the device, based on understanding of the environment sensed in multiple manner (image, motion, sound, etc.) striving to create more friendly user interfaces (augmented reality, virtual reality, haptics, etc.). Despite the good degree of data parallelism, parallel threads in this class of applications usually expose a behavior which is heavily dependent on the local data content, resulting into many truly independent parallel computations. In such a situation, GP-GPUs lose efficiency due to large divergence between threads and the above motivations are **at the core of the decision to investigate in this work the potentials of flexible MIMD/SIMD General-Purpose Programmable Accelerators for the high-end embedded computing domain**.

1.2 Networks-on-Chip (NoCs)

Today's chip-level multiprocessors (CMPs) feature up to hundred discrete cores, and with increasing levels of integration, CMPs with hundreds of cores, cache memories, and specialized accelerators are likely to appear in the near future. As semiconductor transistor dimensions shrink and increasing amounts of IP block functions are added to a chip, the physical infrastructure that carries data on the chip and guarantees quality of service begins to crumble. Many of today's systems-on-chip are too complex to utilize

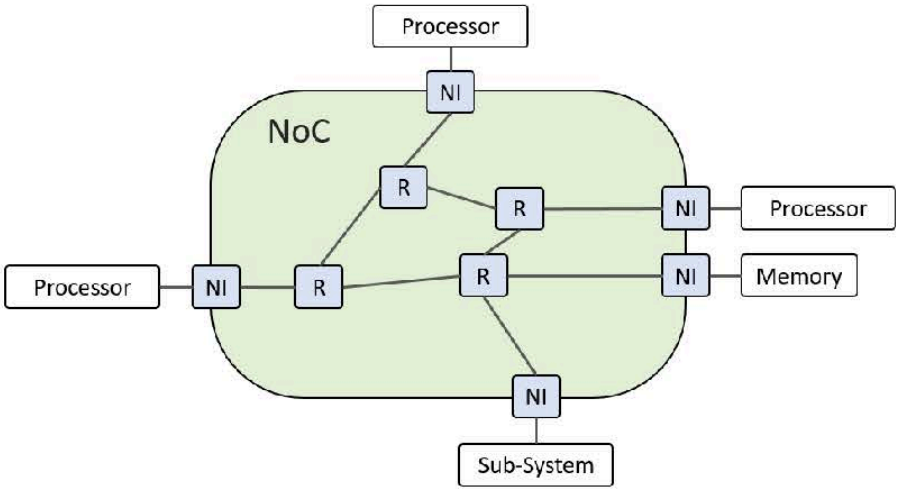


Figure 1.12: Network-on-Chip system.

a traditional hierarchal bus or crossbar interconnect approach. Yesterday's village traffic has turned into today's congested freeways. In fact, the classic interconnection solutions based on shared buses or direct connections between the modules of the chip are becoming obsolete as they struggle to sustain the increasing tight bandwidth and latency constraints that these systems demand. So conventional buses and ad-hoc wiring solutions are not able to manage too many cores with too many signals. In this scenario, among all feasible solutions that have been proposed to cope with the on-chip communication infrastructure, the most promising solutions for the future chip interconnects are the *Networks-on-Chip* (NoCs) [14, 35, 37]. NoCs are the most viable solution that lead to meet the performance and design productivity requirements of a complex on-chip communication infrastructure, providing an infrastructure for better modularity, scalability, fault- tolerance, and higher bandwidth compared to traditional infrastructures.

Furthermore, here are two reasons why today's SoC's need a NoC IP interconnect fabric:

- *Reduce wire routing congestion*: network on chip interconnect fabric technology significantly reduces the number of wires required to route data in a SoC, reducing routing congestion at the backend of the design process. Backend wire routing congestion has become one of the most significant factors causing late designs as the number of IP blocks on a SoC has increased.

- *Higher operating frequencies*: NoC technology simplifies the hardware required for switching and routing functions, allowing SoCs with NoC interconnect fabrics to reach higher operating frequencies. Furthermore, for long or speed-sensitive paths, the architect can easily place pipeline registers along any connection, allowing for higher frequencies. Precise placement of pipeline registers (also called "register slices") allows the interconnect to exactly accommodate the SoC's timing budget and meet its target frequency, with less pipeline register latency and no

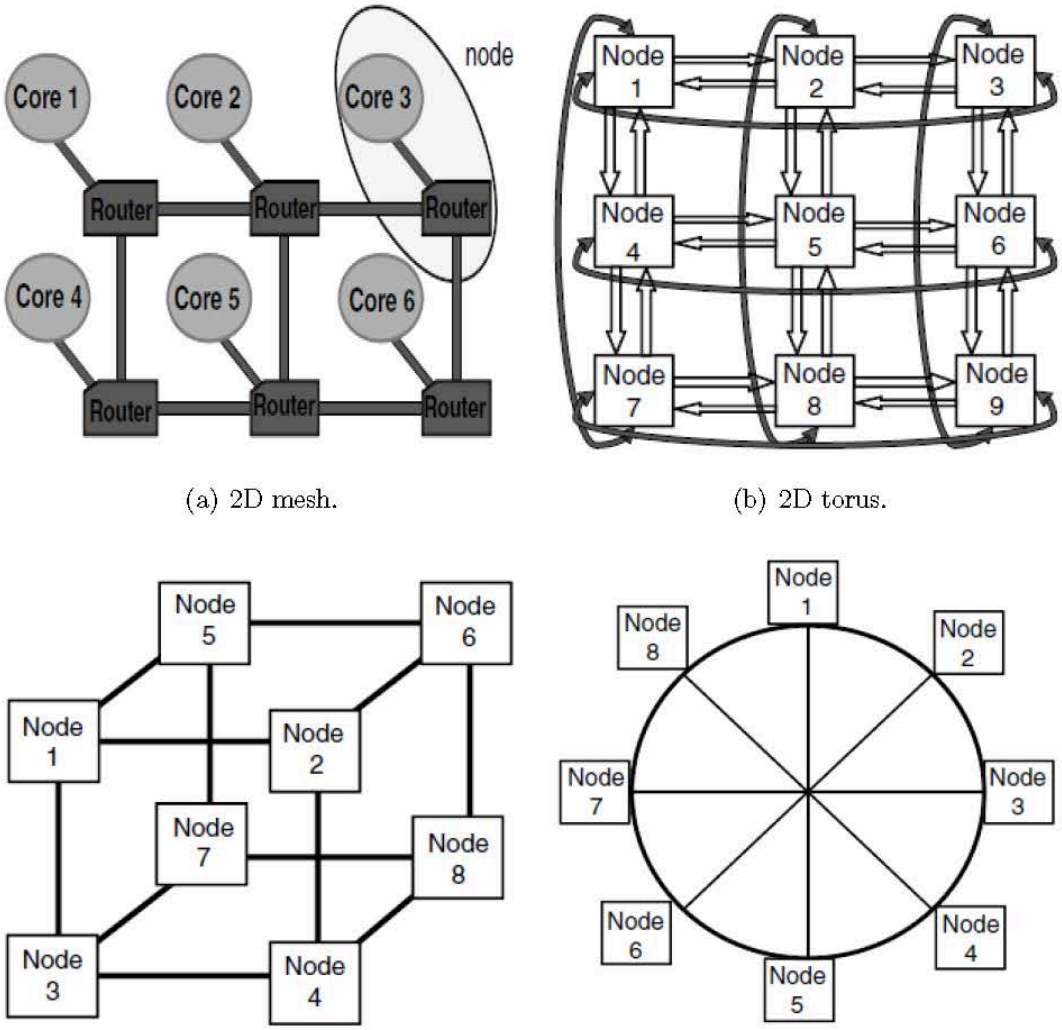
effect on neighboring IP block timing. In addition to pipelining, distributed globally asynchronous locally synchronous (GALS) technology allows synchronous modules with locally generated clocks, with asynchronous connections between them. In short, NoC technology's simpler switching and routing hardware, fine granularity pipelining capability, and GALS allow NoC interconnects to achieve higher operating frequencies than inferior multi-layered bus or crossbar interconnects.

Basically, NoCs are networks composed by routers and channels used to interconnect the different components on the chip. The NoC paradigm addresses these issues by shifting shared buses from on-chip networks. In NoC architectures, processor modules exchange data by dedicated point-to-point links, interconnected by routers. Figure 1.12 shows the overall view of NoC model and its three basic elements that are the *Physical Link* (PL), the *Router* (R) or *Switch* (SW) and the *Network Interface* (NI). The first connects the nodes and actually implements the communication. The second instead, implements the communication protocol: the router basically receives packets from the links and, according to the address in each packet, it forwards the packet to the core attached to it or to another shared link. Finally, the last block is the network adapter, and this block makes the logic connection between the IP cores and the network, since each IP may have a different interface protocol compared to the network.

1.2.1 NoC topologies

A NoC can be characterized by the structure of the routers connection. This structure or organization is called *topology*. The network topology is a key factor for the performance and cost of any NoC design.

The performance of a NoC can be evaluated by three parameters: bandwidth, throughput and latency. The bandwidth refers to the maximum rate of data propagation once a message is in the network. Throughput is defined as the maximum traffic accepted by the network [42], that is, the maximum amount of information delivered per time unit. Finally, latency is the time elapsed between the beginning of the transmission of a message (or packet) and its complete reception at the target node. Depending on the way the router is connected to the end node we can differentiate between direct topologies and indirect topologies. In the first case, direct topology, each router is associated to a processor (or more) and this pair can be seen as a single element in the system, called node of the network. In this type of topology, each node is directly connected to a fixed number of neighbours and a message between two nodes goes through one or more intermediate nodes (Figure 1.13). The most common NoC implementations are based on orthogonal arrangements of the routers. In these arrangements, nodes are distributed in a n-dimension space and the packet moves in one dimension at a time. These topologies are the ones that present the best trade-off between cost and performance, and also present good scalability. The most common



(a) 2D mesh.

(b) 2D torus.

(c) 3D hypercube.

(d) Octagon.

Figure 1.13: NoC with direct regular topologies.

solution is the 2-D mesh direct topology, it is also the most used for the latest commercial products and industrial prototypes, like the Tilera multicore processor family [32], the Intel 80-core TeraFlops Polaris chip [29] and the Kalray MPPA-256 cores [75].

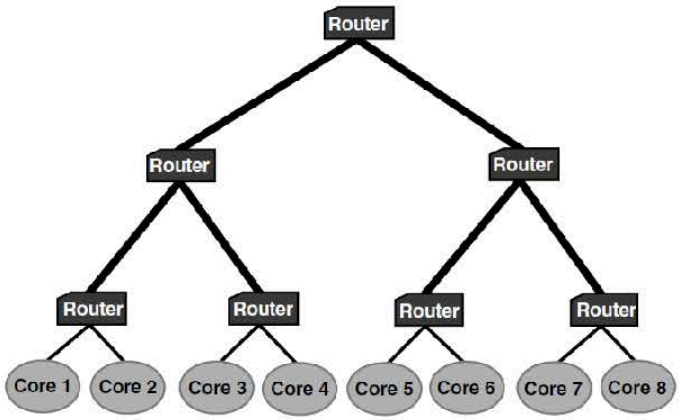


Figure 1.14: NoC with indirect topology (Fat-tree).

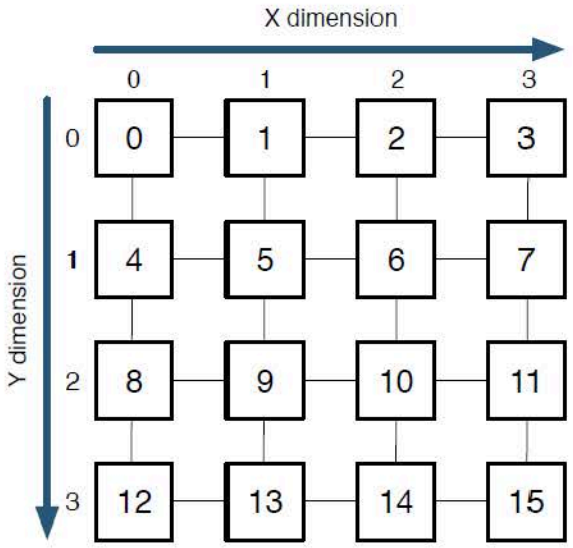


Figure 1.15: 4-ary 2D-mesh NoC topology.

In an indirect topology (Figure 1.14) not every router is connected to processing units as in the direct model. Instead, some routers are only used to propagate the message through the network, while other routers are connected to the logic and only those can be source and/or target of a message. Another possible classification for NoC topologies is related to the regularity of the connection between routers. In regular networks (Figure 1.13), all the routers are identical in terms of number of ports connecting to other routers or elements in the network. In particular, in regular topologies there is a regular predefined pattern that defines how the nodes are connected with each other. Other properties of the network can be:

- Symmetry: a topology is symmetric when the network is the same from every router's point of view.
- Switch degree: it is defined as the number of input/output ports of a router.
- Homogeneity: a topology is homogeneous if all its routers have the same degree, that is, the same number of ports.
- Bisection bandwidth: it is defined as the smallest aggregated bandwidth of all the pairs obtained by dividing the topology into two equal-size halves.
- Hop count: it is defined as the maximum number of routers that must be traversed in the topology in order to travel in the network from an end node to another through a minimal path

In this work the 2D mesh topology is assumed in a way that every router is identified within the network by its coordinates on a n-dimensional space. A router in a n-dimensional graph will be numbered by two coordinates, (x, y), one of each dimension. Figure 1.15 shows a 4-ary 2-mesh (4 x 4 2-D mesh) and moving from router 6, with

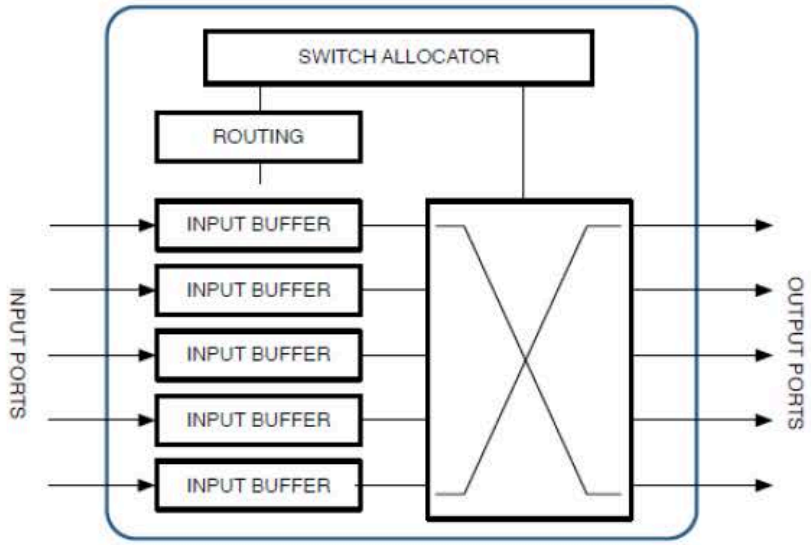


Figure 1.16: Main module in a VC-less router.

coordinates (2, 1), in Y-direction results in router 2 with coordinates (2, 0). Routers are usually numbered by a single ID, computed as a function of the coordinates and the number of the routers per dimension.

1.2.2 The router

The routers, or switches, are the basic building blocks of the interconnection network. Their design critically affects the performance of the whole network both in terms of throughput and latency. Routers are connected through links, to other routers or to terminal nodes. Their function is to make routing decisions and to forward packets which arrive through the incoming links to the proper outgoing links. Typically, the routers include the following four main modules (Figure 1.16):

- Buffers: the task of a buffer is to store temporarily units of information (typically called flit, message or packet). These buffers typically follow a FIFO (First In First Out) access policy; if associated with the input ports, its called input buffering, if associated with the output port, its called output buffering. Their dimension and size greatly affects the cost of each router and the performance of the system.

- Crossbar: the crossbar is the non-blocking network element that allows the connection of all input buffers of the router to all its outputs (buffer or port if the buffering at the output is not implemented). Crossbars are classified by their radix, i.e. the maximum number of connections they can make.

- Routing Unit: this unit is in charge of decoding the unit of information provided by the incoming message, and based on the routing function and destination of the message, computes the most suitable output ports to transmit the message.

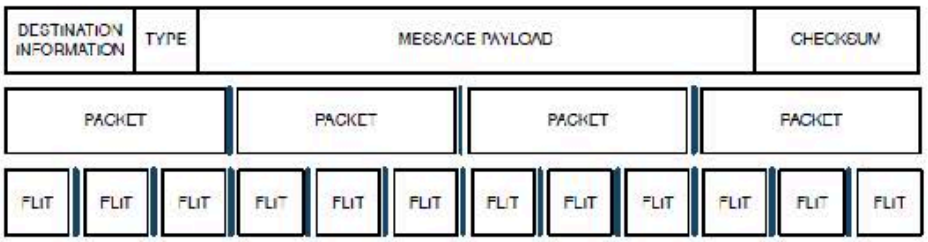


Figure 1.17: Data units.

- Arbiter Unit (Switch Allocator in the figure): the arbiter is in charge of resolving conflicting requests for the same resource and granting only one of them. These units usually guarantee that the grant is received by the request with the highest priority.

Large network packets (Figure 1.17) are broken into small pieces called flits (flow control digits). The first flit, called header flit, holds information about the packets route (ID or destination address) and sets up the routing path for all subsequent flits associated to the packet. The head flit is followed by zero or more body flits which contain the actual payload message of data. Finally, the final flit, called tail flit, performs the closing connection when it passes through a router. The message enters into the network from a source node and is switched or routed towards its destination through a series of intermediate routers. Four many types of switching techniques may usually be used for this purpose: circuit switching, packet switching, virtual cut-through switching, and wormhole switching.

In *circuit switching*, a dedicated path is established between the source and the destination before data transfer initiates. Once the data transfer is initiated, the message will never be buffered nor blocked. This is performed by injecting in the network a head flit, which contains the destination of the transmission and acts as some kind of routing probe that progresses towards the destination node reserving the channels that it gets.

In the packet switching (*store and forward*), a message is divided into packets that are independently routed towards their destination. The destination address is encoded in the header flit of each packet. The entire packet is stored at each intermediate node and then forwarded to the next node along its path. The main advantage of packet switching is that the channel resource is occupied only when a packet is actually being transferred. The major drawback of packet switching is that, since the packet is stored entirely at each intermediate node before departure, the time to transmit a packet from source to destination is multiplicative with the number of hops in the path. Furthermore, at each intermediate node, we need buffer space to hold at least one packet.

In order to reduce the time to store the packets at each router, virtual cut-through switching is used [76]; where, a message leaves the router before it is completely re-

ceived, unless the next channel required is occupied by another packet. Despite this, the buffer requirements are the same as *store and forward* and *packet switching*. This switching technique is used commonly in off-chip high-performance interconnects [36].

Wormhole switching is a variant of virtual cut-through that avoids the need of large buffer space. With this switching technique, a packet is transmitted between routers in units of flit. In wormhole switching, packets are forwarded immediately before they are entirely received, but as opposed to virtual cut-through, there is no need of further space for the rest of the message in case the message blocks. Instead, the message is kept along its path possibly using buffers from different routers. The main disadvantage of this technique comes from the fact that only the header flit has the routing information. If the header flit cannot advance in the network due to resource contention, all the trailing flits are also blocked along the path and these blocked messages can block other messages.

The transmission of a flit between the input and output ports in a router is a task performed by the switching technique. The flow control is in charge of administrating the advance of the packets between the routers. Buffers are temporary resources where to store flits, but they are finite blocks. Instead, the message is kept along its path possibly using buffers from different routers. There are three flow control mechanisms that are commonly used: *ack/nack*, *stop&go* and *credit-based*.

The first flow control protocol (ack/nack) is used for failure detection and retransmission purposes. When the flits are sent to a link, a copy is kept locally at the sender. When the flits are received, either an ack or a nack signal is sent back. Upon receipt of the ack, the sender deletes the copy of the flit, whereas upon receipt of the nack, the sender stops transmitting from its queue and retransmits its local copy starting from the corrupted one.

The stop&go flow control is a very simple realization of an on/off flow control protocol. This mechanism reduces the signaling between the sender and the receiver. When there is an empty buffer space, a go signal is activated. Upon the unavailability of buffer space, a stop signal is activated. stop&go threshold need to be carefully defined not to overflow the queue or to make an inherent resume of traffic flow.

In the credit-based flow control, there is a counter for each channel that tracks the empty buffer space available at the destination buffer. This counter is initialized with the remote buffer size. When flits are sent from the source, the counter is decremented. When flits are consumed by the destination on the other hand, credits are produced in the remote node to indicate that more empty space is available.

The network topology defines the physical organization of the network composed by the nodes, and thus the available path between all the nodes. The routing algorithm is responsible for deciding which path the message has to follow to be effectively routed from its source to its destination. The choice of the routing algorithm becomes of outmost importance in the network performance. But, even in the presence of available

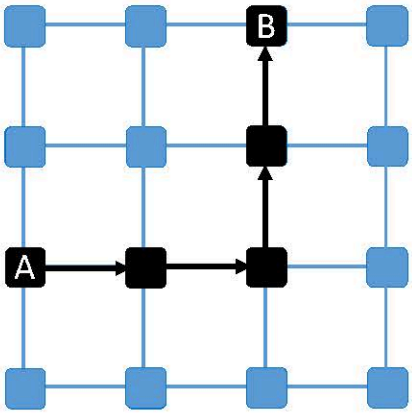


Figure 1.18: XY routing from router A to router B.

physical paths, there are several situations that prevent message delivery:

- Deadlock: the routing is in deadlock situation when two or more packets are waiting for ever for each other to be routed forward. Both these packets reserve some resources and both are waiting for each other to release the resource. The routers do not release the resources before they get the new resources and so the advance is impossible. The two common ways to deal with deadlock events are deadlock avoidance, achieved by employing a deadlock-free routing algorithm, and deadlock detection/recovery.

- Livelock: it occurs when a packet keeps spinning around its destination without ever reaching it. This problem exists in non-minimal routing algorithms. Livelock should be avoided to guarantee packets throughput

- Starvation: Using different priorities can cause a situation where some packets with lower priorities never reach their destinations. This occurs when packets with higher priorities reserve the resources all the time. Starvation can be avoided by using a fair sheduling policy or reserving some bandwidth only for low-priority packets [37]

The *Dimension Order Routing* (DOR) is a typical turn algorithm. The algorithm determines in which direction packets are routed during every stage of routing [36]. XY routing algorithm (Figure 1.18) is a dimension order routing which routes packets first in x-direction to the correct column and then in y-direction to the receiver. XY routing suits well on a network using mesh or torus topology. The coordinates of the routers (x, y) are their addresses. XY routing never runs into deadlock nor livelock [38].

1.3 Logic-Based Distributed Routing

Routing can be implemented as source routing or distributed routing. In source routing, the source node computes the path and stores it in the packet header. Since the

header itself must be transmitted through the network, it consumes significant network bandwidth. In distributed routing, however, each switch computes the next link that will be used while the packet travels across the network. The packet header contains only the destination ID. Distributed routing can be implemented in different ways. The approach followed in regular topologies is the so called algorithmic routing, which relies on a combinational logic circuit that computes the output port to be used as a function of the current and destination nodes and the status of the output ports. The implementation is very efficient in terms of both area and speed, but the algorithm is specific to the topology and to the routing strategy used on that topology. To deal with non-regular topologies, switches based on forwarding tables were proposed. In this case, there is a table at each switch that stores, for each destination end node, the output port that must be used. This scheme can be easily extended to support adaptive routing by storing several outputs in each table entry. The main advantage of table-based routing is that any topology and any routing algorithm can be used, including fault-tolerant routing algorithms. However, memories do not scale in terms of latency, power consumption, and area, thus being impractical for NoCs.

In this thesis, considering the routing algorithm of my NoC's routers, I rely on a very simple mechanism that removes the routing tables at every switch, thus enabling the implementation of almost any routing algorithm on irregular topologies. The mechanism, referred to as *Logic-Based Distributed Routing* (LBDR) [49], relies on three bits per output port at every switch and a small logic block containing several gates. It allows for implementing most of the existing distributed routing algorithms in suitable topologies for NoCs. Only two routing bits and one connectivity bit are required along with a small logic block per output port. LBDR is applicable to any routing algorithm that enforces minimal paths for every source-destination pair.

It is worth recalling that LBDR is a routing mechanism that supports the most widely used routing algorithms for irregular topologies, including segment- based SR [50] routing. As proved, whenever a faulty 2D mesh topology can be handled by LBDR (including its de-route capability), it is always possible to find a suitable SR instance that can be used in combination with LBDR to route that topology. Simply, a fault detection is equivalent to a change in the topology, and the routing, connectivity and deroute bits of all the switches have to be programmed from scratch or incrementally updated with respect to the original fault-free scenario. This is on burden of a configuration algorithm, which needs the list of failed links to recompute the configuration bits for correct routing with the available communication resources. Failure of a switch input or output port (and associated internal logic) can be viewed as the failure of the connected link. This is the case of XY, SR, and up-down(UD) [54] routing.

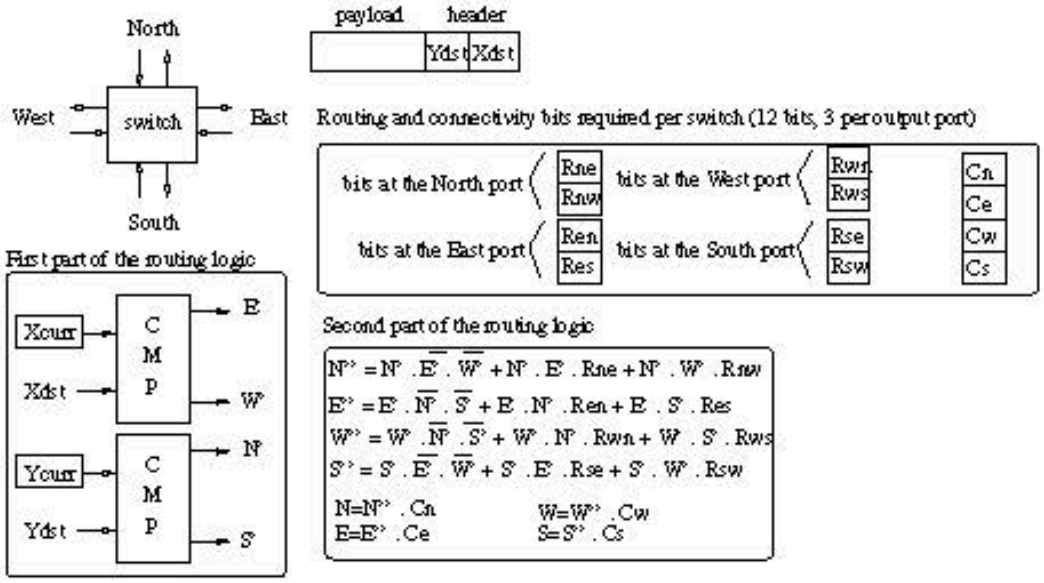


Figure 1.19: LBDR method.

1.3.1 LBDR description

Figure 1.19 shows the details of LBDR. The mechanism relies on the use of only three bits per switch output port. Therefore, 12 bits are needed per switch. The value of these bits depends on the topology and the routing algorithm being implemented, and are computed and uploaded to the switches before normal operation.

Bits are grouped in two sets: routing bits and connectivity bits. Routing bits indicate which routing options can be taken, whereas connectivity bits indicate whether a switch is connected with its neighbours. Regarding the routing bits, the bits for the E output port are labeled R_{EN} and R_{ES}. They indicate whether packets routed through the E output port may later at the next switch take the N port or S port, respectively. In other words, these bits indicate whether packets are allowed to change direction at the next switch. Similarly, for output port N the bits are accordingly labeled R_{NE} and R_{NW}, for output port W R_{WN} and R_{WS}, and for output port S R_{SE} and R_{SW}.

Regarding the connectivity bits, each output port has a bit, referred to as C_X indicating whether a switch is connected through the X port. Thus, connectivity bits are C_N, C_E, C_W, and C_S.

The routing logic is divided in two parts (see Figure 1.19). The first part computes the relative position of the packet's destination. For this, two comparators are used and X_{curr} and Y_{curr} are compared with X_{dst} and Y_{dst}. From that logic one or two signals may be active (if the packet is in the NW quadrant then N and W signals will be active). Note also that packets forwarded to the local port are excluded from the routing logic.

Once the Np, Wp, Ep and Sp signals are computed, the second part of the logic comes into play. It is made of four logic units, one for each output port. Each one can be built with only two inverters, four AND gates and one OR gate. As all of them are similar we will describe only one, the logic associated with the N output port. The

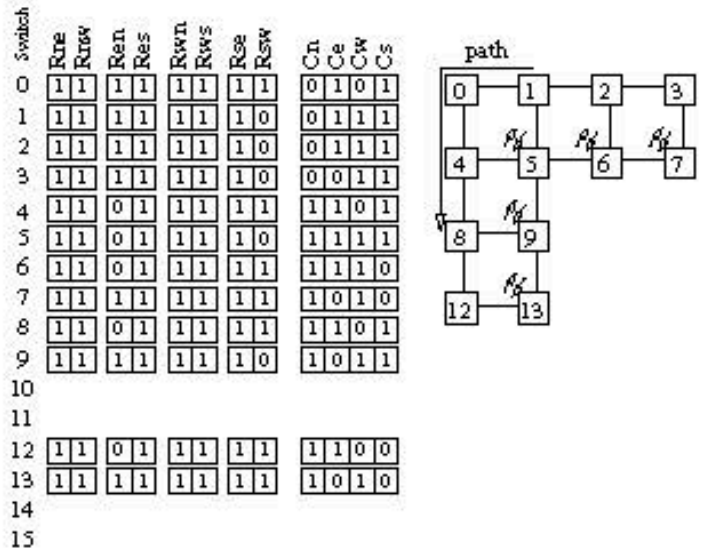


Figure 1.20: Example of LBDR for an irregular (p) topology with UD routing.

N output port is considered for routing the incoming packet when either one of the following three conditions is met:

- The packet's destination is on the same column ($N' \times \overline{E'x}\overline{W'}$).
- The packet's destination is on the NE quadrant and the packet can take the E port at the next switch through the N port ($N' \times E' \times R_{NE}$).
- The packets destination is on the NW quadrant and the packet can take the W port at the next switch through the N port ($N' \times W' \times R_{NW}$).

If none of the above conditions is met, then the N port can not be taken for routing the packet. Additionally, the connectivity bit C_N is inspected in order to filter the N port.

LBDR will mimic performance of most of the routing algorithms. This is the case for the XY and UD routing algorithms. In these algorithms, the routing restrictions are located in the same relative position through all the rows and columns. As an example, Figure 1.20 shows all the bits for UD in a p topology. Notice that for the path 1-8 output port S is not taken at switch 1 because there is a NW routing restriction at switch 5 (bit R_{GW} is zero at switch 1). This decision does not impact on performance as the S output port cannot be taken to forward properly the packet (packet would never be able to turn to W in the column).

However, there are situations (e.g. more advanced routing algorithms like SR) where LBDR induces some inefficiencies. Therefore, LBDR reduces adaptiveness.

It is important to note that only the routing bits referring to a routing restriction are set to zero and the remaining ones are set to one, even those that refer to switches not existing in the topology (for instance bit R_{NW} at switch 0). However, they must be set to one in order the mechanism to work properly. This can be better seen through

an example. Imagine the path at Figure 1.20 from switch 13 to switch 7. Signals N and E are active at switch 13. N is activate as R_{NE} although it does not make sense for routing purposes. This allows the packet to being forwarded north until it reaches switch 5, where it will take east direction. Notice that output port E will never be taken at switches 13 and 9 due to the connectivity bit C_E.

Chapter 2

Virtual platforms for heterogeneous parallel computer architectures

In this chapter, I present the two simulation environments I use and augment in this thesis, trying to cope with the requirements of heterogeneous parallel computing architectures, and of many-core programmable accelerators in them. In particular, I provide a description of the VirtualSoC virtual platform and of the modifications I applied to allow modeling and simulation of a multi-cluster accelerator system with clock cycle accuracy. Then, I describe the gem5 environment, targeting a transactional accuracy. In particular, I report on how I customized it to model heterogeneous systems and to run parallel benchmarks in order to evaluate novel resource sharing strategies. Finally I compare the features of the two simulators, highlighting their complementarity.

Key novelty: development of virtual platforms to simulate high-performance heterogeneous parallel computer architectures at different abstraction layers.

2.1 Introduction

As reported in Chapter 1, several simulation frameworks are available today off-the-shelf [24, 29, 75, 77, 84, 87, 126, 136], but almost all of them suffer from three main problems, which make them not suitable to model a complex MPSoC:

- Lack of models for deep micro-architectural components: hardware designs with more than hundreds of computing units use various architectural components, to allow efficient and scalable communication between cores (e.g. Networks-On-Chip) and complex memory hierarchies. Such components have to be modeled at the micro-architectural level to enable accurate power estimations and performance measurements.

- Lack of support for Full System simulation: modern MPSoCs are composed by a Host processor and one or more accelerators. The host processor is usually in charge of executing an operating system (e.g. Linux), while the accelerators are used as a co-processors to speedup the execution of computationally heavy tasks. In this scenario the interaction between host processor and accelerators, being it a memory transfer or a synchronization, may have a significant effect on applications performance. Virtual platforms have to accurately model such interactions to enable precise application profiling.

- Sequential simulation: most of the available modeling tools are relying on a sequential execution model, in which all components of the design are simulated in sequence by a single application thread. In the near future MPSoCs will feature thousands of computing units, and such a modeling technique will make the simulation time of a reasonable application to be too slow for practical use.

In this chapter I provide two examples of simulators that can cope with the requirements of MPSoCs systems, focusing on heterogeneous architectures composed by a host processor and a programmable accelerator.

2.2 SystemC VirtualSoC development

2.2.1 Overview

Performance modeling plays a critical role in the design, evaluation, and development of computing architecture of any segment, ranging from embedded to high performance processors. Simulation has historically been the primary vehicle to carry out performance modeling, since it allows for easily creating and testing new designs several months before a physical prototype exists. Performance modeling and analysis are now integral to the design flow of modern computing systems, as it provides many significant advantages: i) accelerates time-to-market, by allowing the development of software before the actual hardware exists; ii) reduces development costs and risks, by allowing for testing new technology earlier in the design process; iii) allows for exhaustive design space exploration, by evaluating hundreds of simultaneous simulations in parallel.

High-end embedded processor vendors have definitely embraced the heterogeneous architecture template for their designs as it represents the most flexible and efficient design paradigm in the embedded computing domain. Parallel architecture and heterogeneity clearly provide a wider power/performance scaling, combining high performance and power efficient general-purpose cores along with massively parallel many-core-based accelerators. Examples and results of this evolution are AMD Fusion, NVidia Tegra and Qualcomm Snapdragon. Besides the complex hardware, generally these kinds of platforms host also an advanced software eco-system, composed by an

operating system, several communication protocol stacks, and various computational demanding user applications.

Unfortunately, as processor architectures get more heterogeneous and complex, it becomes more and more difficult to develop simulators that are both fast and accurate. Cycle-accurate simulation tools can reach an accuracy error below 1-2%, but they typically run at a few millions of instructions per hour. The necessity to efficiently cope with the huge HW/SW design space provided by this target architecture makes clearly full-system simulator one of the most important design tools. Clearly, the use of slow simulation techniques is challenging especially in the context of full-system simulation. In order to perform an affordable processor design space exploration or software development for the tar- get platform, trade-off accuracy for speed is thus necessary by implementing new virtual platforms that allow for faster simulation speed at the expense of model- ing fewer micro-architecture details of not-critical hardware components (like the host processor domain), while keeping high-level of accuracy for the most critical hardware components (like the many-core accelerator domain). We present in this chapter VirtualSoC [20], a new virtual platform prototyping framework targeting the full-system simulation of massively parallel heterogeneous system-on-chip composed by a general purpose processor (i.e. intended as platform coordinator and in charge of running an operating system) and a many-core hardware accelerator (i.e. used to speed-up the execution of computing intensive applications or parts of them). VirtualSoC exploits the speed and flexibility of QEMU, allowing the execution of a full-fledged Linux operating system, and the accuracy of a SystemC model for many-core-based accelerators.

The specific features of VirtualSoC are:

- Since it exploits QEMU for the host processor emulation, unmodified operating systems can be booted on VirtualSoC and the execution of unmodified ARM binaries of applications and existing libraries can be simulated on VirtualSoC.
- VirtualSoC enables accurate manycore-based accelerator simulation: there is a full software stack allowing the programmer to exploit the hardware accelerator model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.
- The host processor (emulated by QEMU) and the SystemC accelerator model can run in an asynchronous way, where a non-blocking communi- cation interface has been implemented enabling parallel execution between QEMU and SystemC environments.
- Beside the interface between QEMU and the SystemC model, it is implemented a synchronization protocol able to provide a good approximation of the global system time.

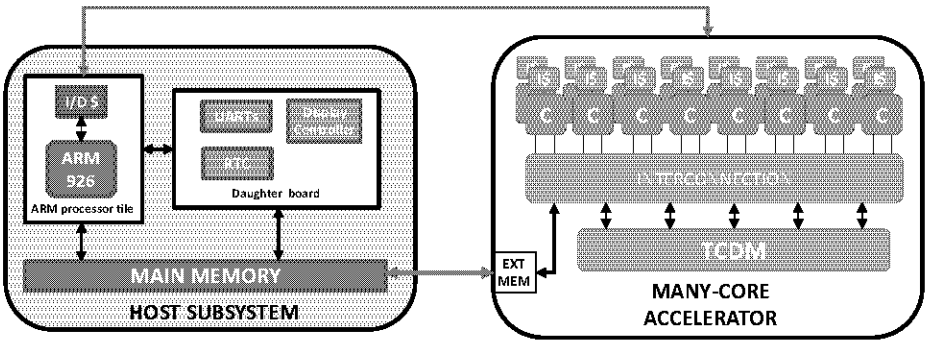


Figure 2.1: Target simulated architecture.

- VirtualSoC can be also used in stand-alone mode, where only the hardware accelerator is simulated, thus enabling accurate design space explorations.

To the best of my knowledge, there are no existing public domain, open source simulators that rival the characteristics of VirtualSoC. This section focuses on the implementation details of VirtualSoC and evaluates the performance of various benchmarks, and finally presents some example case studies using VirtualSoC.

One of the main advantages of VirtualSoC is that it is a SystemC simulator, enabling a cycle-accurate simulation but also allowing the users to create a new emulation framework based on QEMU and SystemC which overcomes the issues of other simulators. QEMU is chosen amongst all simulators because it is fast, open-source and also very flexible enabling its extension with a moderate effort. Our approach is based on thread parallelization and memory sharing to obtain a complete heterogeneous SoC emulation platform. In our implementation the target processor and the SystemC model can run in an asynchronous way, where non-blocking communication is implemented through the use of shared memory between threads. Beside the interface between QEMU and a SystemC model, there is a lightweight implementation of a synchronization protocol able to provide a good approximation of a global system time. Moreover, it was designed a full SW stack allowing the programmer to exploit the HW model implemented in SystemC, from within a user-space application running on top of QEMU. This software stack comprise a Linux device driver and a user-level programming API.

2.2.2 Baseline architecture

Modern embedded SoCs are moving toward systems composed by a general purpose multi-core processor accompanied by a more energy efficient and powerful many-core accelerator (e.g. GPU). In these kinds of systems the general purpose processor is intended as a coordinator and is in charge of running an operating system, while the many-core accelerator is used to speed up the execution of computing intensive applications or parts of them. Despite their great computing power, accelerators are not

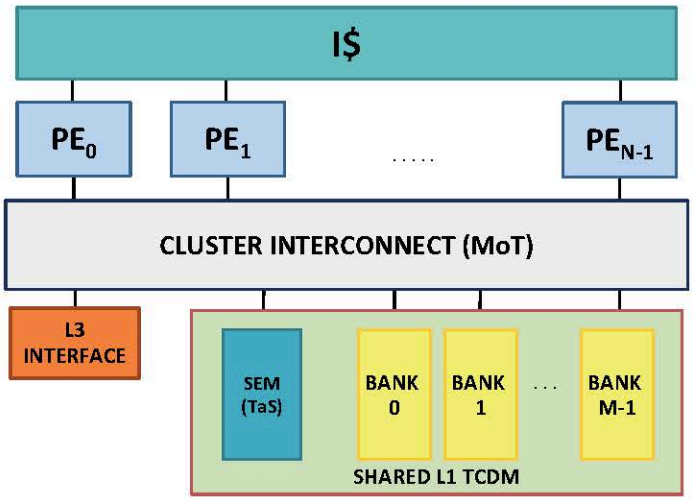


Figure 2.2: Single cluster of a programmable many-core accelerator.

able to run an operating system due to the lack of all needed surrounding devices and to the simplicity of their micro-architectural design. The architecture of VirtualSoC (shown in Figure 2.1 is representative of the above mentioned platforms and composed by a many-core accelerator and an ARM-based processor.

The ARM processor is emulated by QEMU which models an ARM926 processor, featuring an ARMv5 ISA, and interfaced with a group of peripherals needed to run a full-fledged operating system (ARM Versatile Express baseboard). The many-core accelerator is a SystemC cycle-accurate MPSoC simulator. The ARM processor and the accelerator share the main memory, used as communication medium between the two. The accelerator target architecture features a configurable number of simple RISC cores, with private or shared I-cache architecture, all sharing a Tightly Coupled Data Memory (TCDM) accessible via a local inter- connection. The state-of-the-art programming model for this kind of systems is very similar to the one proposed by OpenCL or by OpenMP: **a master application is running on the host processor which, when encounters a data or task parallel section, offloads the computation to the accelerator. The master processor is in charge also of transferring input and output data**.

2.2.3 Many-core single cluster accelerator

The proposed target many-core accelerator template can be seen as a cluster of cores connected via a local and fast interconnect to the memory subsystem. The following sub-sections describe the building blocks of such cluster, shown in Figure 2.2.

Processing elements

The accelerator consists of a configurable number of 32-bit RISC processor. In the specific platform instance that I consider in this chapter I use ARMv6 processor models,

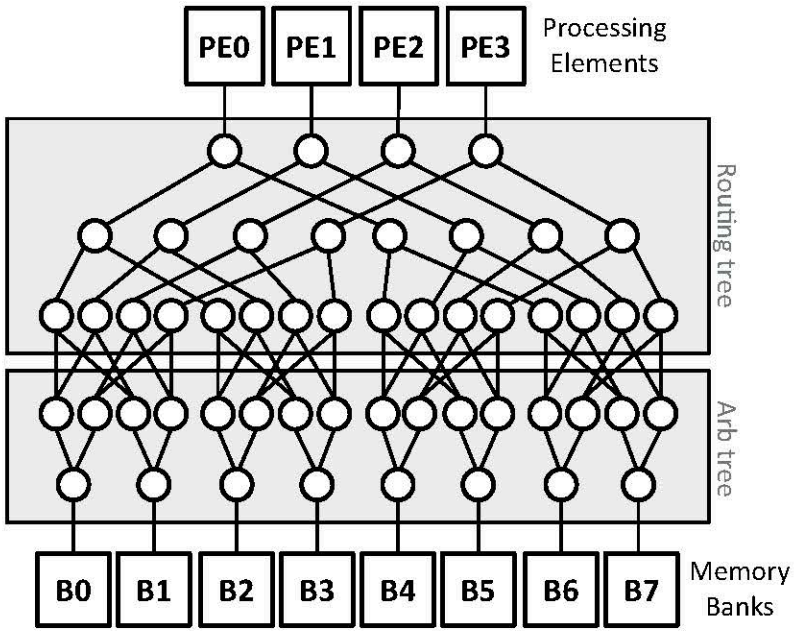


Figure 2.3: Mesh of trees 4x8 (banking factor of 2).

specifically the ISS. To obtain timing accuracy its internal behavior is modified to model a Harvard architecture wrapping the ISS in a SystemC module.

Local interconnect

The local interconnection has been modeled, from a behavioral point of view, as a parametric Mesh-of-Trees (MoT) interconnection network (logarithmic interconnect) to support high-performance communication between processors shown in Figure 2.3. The module is intended to connect processing elements to a multi-banked memory on both data and instruction side. Data routing is based on address decoding: a first-stage checks if the requested address falls within the local memory address range or has to be directed to the main memory. To increase module flexibility this stage is optional, enabling explicit L3 data access on the data side while, on the instruction side, can be bypassed letting the cache controller take care of L3 memory accesses for lines refill. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The crossing latency consists of one clock cycle. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates access and a higher number of cycles is needed depending on the number of conflicting requests, with no latency in between. In case of no banking conflicts data routing is done in parallel for each core, thus enabling a sustainable full bandwidth for processors-memories communication. To reduce memory access time and increase shared memory throughput, read broadcast has been implemented and no extra cycles are needed when broadcast occurs.

TCDM

On the data side, a L1 multi-ported, multi-banked, Tightly Coupled Data Memory (TCDM) is directly connected to the logarithmic interconnect. The number of memory ports is equal to the number of banks to have concurrent access to different memory locations. Once a read or write request is brought to the memory interface, the data is available on the negative edge of the same clock cycle, leading to two clock cycles latency for conflict-free TCDM access. As already mentioned above, if conflicts occur there is no extra latency between pending requests, once a given bank is active, it responds with no wait cycles.

Synchronization

To coordinate and synchronize cores execution the architecture exploits *HW semaphores* mapped in a small subset of the TCDM address range. They consist of a series of registers, accessible through the data logarithmic interconnect as a generic slave, associating a single register to a shared data structure in TCDM. By using a mechanism such as a hardware "test and set" it is possible to coordinate access: if reading returns 0 the resource is free and the semaphore automatically locks it, if it returns a different value, typically 1, access is not granted. This module enables both single and two-phases synchronization barriers, easily written at the software level.

Instruction Cache Architecture

The L1 Instruction Cache basic block has a core-side interface for instruction fetches and an external memory interface for refill. The inner structure consists of the actual memory and the cache controller logic managing the requests. The module is configurable in its total size, associativity, line size and replacement policy (FIFO, LRU, random). The basic block can be used to build different Instruction Cache architectures:

- Private Instruction Cache: every processing element has its private I-cache, each one with a separate cache line refill path to main memory leading to high contention on external L3 memory.

- Shared Instruction Cache: there is no difference between the private architecture in the data side except for the reduced contention L3 memory (line refill path is unique in this architecture). Shared cache inner structure is made of a configurable number of banks, a centralized logic to manage requests and a slightly modified version of the logarithmic interconnect de- scribed above: it connects processors to the shared memory banks operating line interleaving (1 line consists of 4 words). A round robin scheduling guarantees fair access to the banks. In case of two or more processors requesting the same instruction, they are served

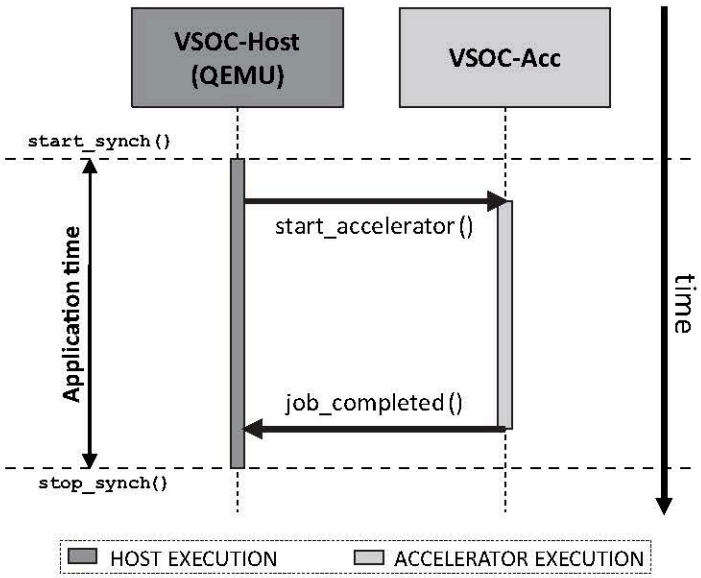


Figure 2.4: Execution model.

in broadcast not affecting hit latency. In case of concurrent instruction miss from two or more banks, a simple bus handles line refills in round robin towards the L3 bus.

2.2.4 Host-Accelerator Interface

In this section a description of QEMU-based host side of VirtualSoC (VSoC- Host) is provided, as well as the many-core accelerator side (VSoC-Acc).

Parallel Execution

In a real heterogeneous SoC host processor and accelerator can execute in an asynchronous parallel fashion, and exchange data using non-blocking communication primitives. Usually the host processor, while running an application, offloads asynchronously a parallel job to the accelerator and goes ahead with its execution (Figure 2.4). Only when needed the host processor synchronizes with the execution of the accelerator, to check the results of the computation.

In our virtual platform the host processor system and the accelerator can run in parallel, with VSoC-Host and VSoC-Acc running on different threads: when the thread of VSoC-Acc starts its execution triggers the SystemC simulation. It is important to highlight that the VSoC-Acc SystemC simulation starts immediately during VSoC-Host startup, and the accelerator starts executing the binary of a firmware (until the shutdown) in which all cores are waiting for a job to execute.

Time Synchronization Mechanism

VSoC-Host and VSoC-Acc run independently in parallel with a different notion of time. The lack of a common time measure leads to only functional simulation, without the possibility of profiling applications performance even in a qualitative way. Application developers often need to understand how much time, over the total application time, is spent on the host processor or on the accelerator. Also, without a global simulation time it is not possible to appreciate execution time speedups due to the exploitation of the many-core accelerator.To manage the time synchronization between the two environments, it is necessary that both VSoC-Host and VSoC-Acc have a time measurement system. VSoC-Host does not natively provide this kind of mechanisms, so it was instrumented to implement a clock cycle count, based on instructions executed and memory accesses performed. On the contrary for VSoC-Acc there is no need for modifications be- cause it is possible to exploit the SystemC time. The synchronization mechanism used in our platform is based on a threshold protocol acting on simulated time: at fixed synchronization points the simulated time of VSoC-Host and VSoC-Acc is compared. If the difference is greater than the threshold, the entity with the greater simulated time is stopped until the gap is filled.

2.2.5 Simulation software support

In this section, I provide a description of the software stack provided with the simulator to allow the programmer to fully exploit the accelerator from within the host Linux system, and to write parallel code to be accelerated.

Linux driver

In order to build a full system simulation environment VSoC-Acc is mapped as a device in the device file system of the guest Linux environment running on top of VSoC-Host. A device node /dev/vsoc has been created, and as all Linux devices it is interfaced to the operating system using a Linux driver. The driver is in charge of mapping the shared memory region into the kernel I/O space. This region is not managed under virtual memory because the accelerator can deal only with physical addresses, as a consequence all buffers must be allocated contiguously (done by the Linux driver). The driver provides all basic functions to interact with the device.

Hot side User-Space library

To simplify the job of the programmer I have designed a user level library, which provides a set of APIs that rely on the Linux driver functions. Through this library the programmer is able to fully control the accelerator from the host Linux system. It is possible for example to offload a binary, or to check the status of the current executing job (e.g. checking if it has finished).

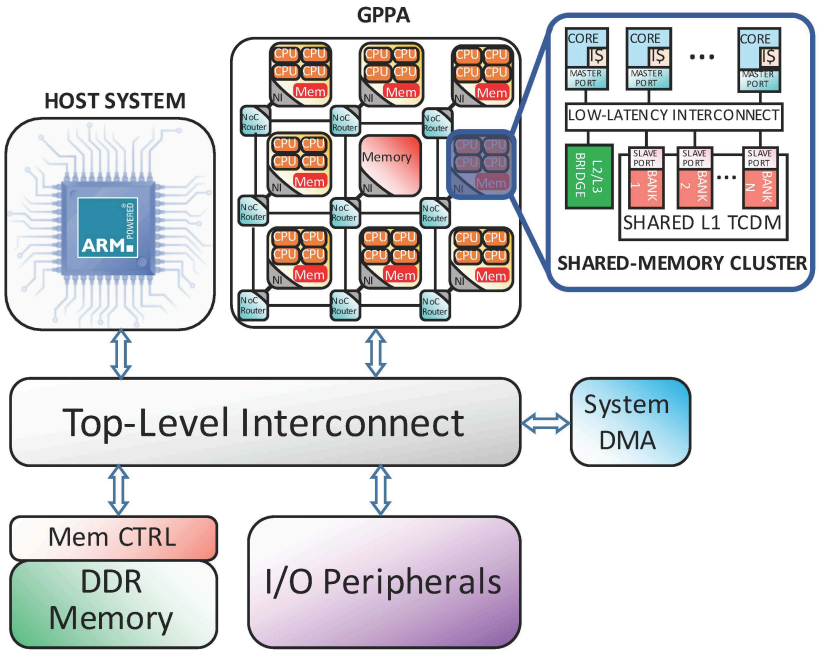


Figure 2.5: Heterogeneous (many-core accelerator-based) MPSoC architecture.

Accelerator Side Software Support

The basic manner I provide to write applications for the accelerator is to directly call from the program a set of low-level functions implemented as a user library, called **appsupport**. appsupport provides basic services for memory management, core ID resolution, synchronization. To further simplify programming and raise the level of abstraction it was also supported a fully-compliant OpenMP v3.0 programming model, with associated compiler and runtime library.

Summary

VirtualSoC leverages QEMU to model a ARMv6 host processor, capable of running a full-fledged Linux operating system. The many-core accelerator is modeled with higher accuracy using SystemC. Extended this combined simulation technology with a mechanism a gathering timing information is allowed that is kept consistent over the two computational sub-blocks. A set of experiments over a number of representative benchmarks demonstrate the functionality, flexibility and efficiency of the proposed approach. Despite its flexibility, VirtualSoC is still based on sequential simulation whose speed decreases when increasing the complexity of the modeled platform. In the next chapter this problem is tackled by exploiting off-the-shelf GPGPUs to speedup the simulation process.

2.2.6 Modifying VirtualSoC: the target multi-clusters version

In this section I present the modifications of baseline VistualSoC, to enable multi-clusters simulations, that is the target architecture to cope with the GPPAs require-

ments.

In Figure 2.5 is reported a typical MPSoC architecture. As described in the previously in this chapter, VirtualSoC enables the simulation of a many-core system considering up to 16 cores in the same cluster, in addition to a host processor that is simulated by QEMU. This is not enough to cope with the modern many-core architectures, that in fact present more clusters, each one with multi-core.

Figure 2.6 depicts exactly what I need to simulate: a multi-clustered accelerator (that will be the GPPA) that can contain up to 16 cores for each cluster. It consists of a configurable number of computing clusters (up to 9 in our basic setup), interconnected by a 2-D mesh network-on-chip. The topology of the NoC is a simple *nxn* mesh (extended up to 5x5 in some versions) . Each of the first 9 nodes includes a computing cluster and a L2 bank. Another node hosts the *Fabric Controller*, a special cluster instance with a single processor acting as a main controller for the whole many-core platform. This node interacts directly with the host system, and is in charge of the boot sequence of other clusters and their operation control. It has the fundamental role of managing NoC routing reconfiguration, setting up partitions and starting applications. Among the remaining three nodes, one switch is reserved to communications with an I/O interface (GPPA reading and writing ports), while the other two are temporarily left unused, and are available for future extension of the computation power. Every full-cluster block is linked to a switch of the on-chip network with two network interfaces (NIs), a master and a slave one, supporting OCP (Open Core Protocol). The master

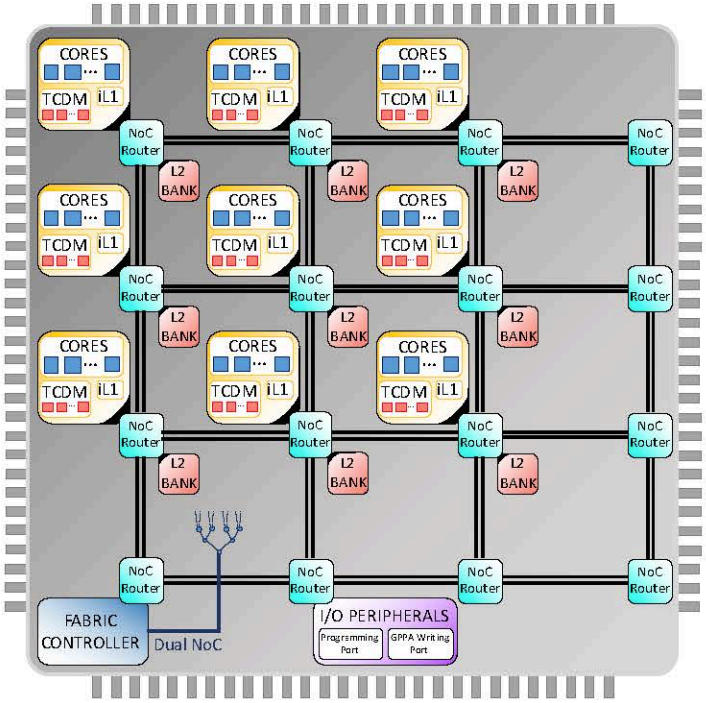


Figure 2.6: GPPA architecture example: 12 switches, 9 with computing clusters and L2 distributed blocks; 2 additional switches are reserved to the Fabric Controller (hypervisor) and to the I\O interface.

NI is dedicated to the core transactions, while the slave NI is used for accessing the internal cluster memory. Accesses to the L2 banks are feasible thanks to dedicated slave NIs. Every router has a block that manages LBDR routing. A Round-Robin arbiter (or allocator) is implemented inside each switch, enriched with up to 16 levels of priority to guarantee QoS. There is also the possibility, if a global hypervisor or Fabric Controller requires it, to set circuits between different sources and destinations, reserving those paths for critical packets (hard QoS).

Each cluster has an internal memory organized as private, per-core L1 instruction caches plus local L1 scratchpad data memory shared among all cores. I also re-architect the L2 memory as a multi-bank distributed shared memory, where each NoC router hosts a L2 bank. This requires to modify the switch arbitration logic, having an additional bidirectional port per switch toward the memory to be controlled. Finally, also the Network Interfaces need to be modified, this way introducing address interleaving for the L2 memory, being this a well-known strategy to speedup and optimize memory accesses. Overall, the memory system is organized as a partitioned global address space (PGAS). Each processor in the system can explicitly address every memory segment: local TCDM, remote TCDMs, L2, and L3 memory. Clearly, transactions that traverse the boundaries of a cluster are subject to NUMA effects: higher latency and lower bandwidth. When the GPPA has to perform a new computation, the code binary is copied via global direct memory access (DMA) into the L2. Data is stored in the L3 (main) memory, where it is originally allocated by host programs. Permanently hosting entire data structures in the L1 TCDMs is not feasible, due to a limited size of 256 KB. The software must thus explicitly orchestrate data transfers from L3 to L1 or L2, to ensure that the most frequently referenced data are kept close to the processors. To enable performance and energy-efficient transfers, each cluster is equipped with a local DMA engine.

Furthermore, there is the possibility to have up to two physical networks (one global and one local) and up to two Virtual Channels for each of them, to support different types of communications, separating the ones that can have conflicts and that can generate deadlock. Of course VC's arbiters and (de-)multiplexing logic are required to manage and control of this huge hardware.

To make the routers easier to be reached by the hypervisor, I implement a Dual NoC, providing an efficient propagation phase very useful to deliver (re-)configuration LBDR bits. Once the Fabric Controller got the new reconfiguring bits, I have to consider the way they are provided to each switch. If the focus is on functional validation I can consider a synchronous way, i.e. at the same time for all the switches of the NoC, but in this case I relax that assumption and consider a more realistic setting, where control bits of the reconfiguration process are brought by a dual network. Such a dual bus may be another 2D mesh superimposed to the main one, however this would be too much of an overhead. I more realistically envision a path topology connecting all

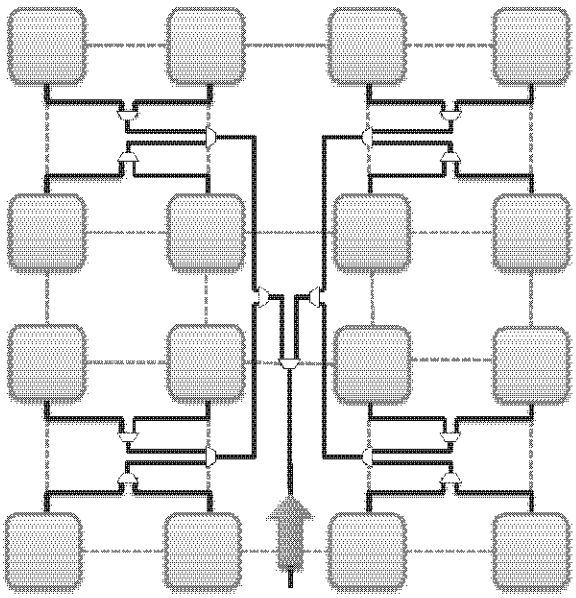


Figure 2.7: A tree topology Dual NoC to deliver configuring bits from the Fabric Controller to the routers of the NoC.

the switches of the main NoC, based on a binary tree of four logic levels, that allows to reach all the switches of the whole NoC and so to create or reconfigure a partition without being constrained to the path imposed by the dual bus. The tree topology, shown in Figure 2.7 improves and optimizes the distribution of the new configuration. This unconstrained control network guarantees to reach each switch in 3 cycles because I am considering information about reconfiguration bits modeled as a 3 flits packet, concerning the possible information width of the bus in worst case. Also thanks to this dual network, the cost of this phase (i.e. delivering new LBDR bits) is very low considering the whole scenario.

The final outcome of this part of the work is a working multi-clustered many-core system simulator (in Figure 2.8 is reported a 2x2 mesh of the new architectural template) with a multi-clustered accelerator. This is the baseline template that has been furthermore modified and extended in several ways to enable better modeling resources sharing on the GPPA architecture. The extensions include in particular modifications to the baseline NoC switches to enable the creation of dynamically reconfigurable par-

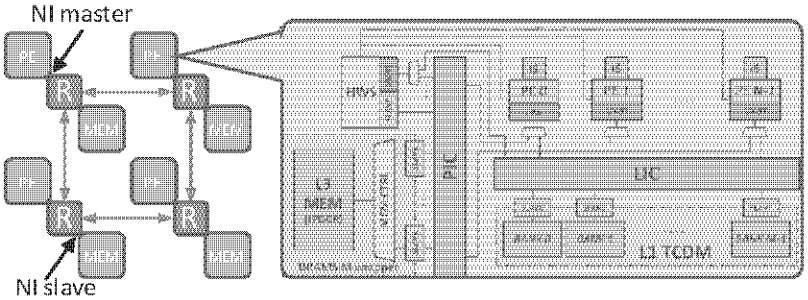


Figure 2.8: Simplified new VirtualSoC target architecture: 2x2 mesh.

titions and to fully support the GPPA NoC architecture, and are described in Chapter 5 and 6.

The main issue with this huge hardware is the simulation time: VirtualSoC in the multi-clustered variant presents a very poor instructions/sec ratio, so running real benchmarks the simulations' duration is sometimes unaffordable.

2.3 gem5 development toward heterogeneous parallel computer architectures

To overcome the long simulation duration of VirtualSoC, needing to run real Image Processing benchmarks that usually require a lot of simulation cycles, and furthermore to enable simulations 64bit architecture processors, I leverage the gem5 simulator [17].

gem5 is a modular, discrete, event-driven computer systems simulator platform. Its main features are:

- gem5 components can be rearranged, parameterized, extended or replaced easily to suit your needs.
- It simulates the passing of time as a series of discrete events.
- Its intended use is to simulate one or more computer systems in various ways.
- It is more than just a simulator; it is a simulator platform that lets you use as many of its premade components as you want to build up your own simulation system.

gem5 is written primarily in C++ (to model the behaviour of hardware components and the transactions between them) and python to configure the blocks in the system to be simulated. Most real components models are provided under a BSD style license. It can simulate a complete system with devices and an operating system in *full system mode* (FS mode), or user space only programs where system services are provided directly by the simulator in *syscall emulation mode* (SE mode). There are varying levels of support for executing Alpha, ARM, MIPS, Power, SPARC, and 64 bit x86 binaries on CPU models including two simple single CPI models, an out of order model, and an in order pipelined model. A memory system can be flexibly built out of caches and crossbars. Recently the Ruby simulator has been integrated with gem5 to provide even more flexible memory system modeling.

More in detail, the gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. It can simulate multiple interchangeable CPU models. gem5 provides four interchangeable CPU models: a simple one-CPI CPU; a detailed model of an in-order CPU, and a detailed model of an out-of-order CPU. The CPU models use a

common high-level ISA description. In addition, gem5 features a KVM-based CPU that uses virtualization to accelerate simulation.

There is a NoMali GPU model: gem5 comes with an integrated NoMali GPU model that is compatible with the Linux and Android GPU driver stack, and thus removes the need for software rendering. The NoMali GPU does not produce any output, but ensures that CPU-centric experiments produce representative results.

It is, as already said, an event-driven memory system. gem5 features a detailed, event-driven memory system including caches, crossbars, snoop filters, and a fast and accurate DRAM controller model, for capturing the impact of current and emerging memories, e.g. LPDDR3/4, DDR3/4, HBM1/2, HMC, WideIO1/2. The components can be arranged flexibly, e.g., to model complex multi-level non-uniform cache hierarchies with heterogeneous memories.

There is a trace-based CPU model that plays back elastic traces, which are dependency and timing annotated traces generated by a probe attached to the out-of-order CPU model. The focus of the Trace CPU model is to achieve memory-system (cache-hierarchy, interconnects and main memory) performance exploration in a fast and reasonably accurate way instead of using the detailed CPU model.

You can instantiate homogeneous and heterogeneous multi-core. The CPU models and caches can be combined in arbitrary configurations, creating homogeneous, and heterogeneous multi-core systems. A MOESI snooping cache coherence protocol keeps the caches coherent. Multiple ISA support is enabled: gem5 decouples ISA semantics from its timing CPU models, enabling effective support of multiple ISAs. Currently gem5 supports the Alpha, ARM, SPARC, MIPS, POWER and x86 ISAs.

Finally a co-simulation with SystemC is supported, in fact gem5 can be included in a SystemC simulation, effectively running as a thread inside the SystemC event kernel, and keeping the events and timelines synchronized between the two worlds. This functionality enables the gem5 components to interoperate with a wide range of System on Chip (SoC) component models, such as interconnects, devices and accelerators. A wrapper for SystemC Transaction Level Modeling (TLM) is provided with the code.

2.3.1 A customized gem5-based GPPA

Relying on the potentials of gem5 simulator, I implement the system reported in Figure 2.9. The host sub-system consists of a ARMv8 processor (64-bit Cortex A53 or A57). The cluster presents 1 or 2 cores with both instructions and data L1 cache memory. It is connected through a coherent crossbar to second-level cache memory (L2) and through another crossbar to a DRAM memory (DDR3).

The latter crossbar leverages on a special bridge I created to enable the bidirectional communication between two crossbar (unsupported in the baseline gem5), let the host sub-system communicate with the accelerator sub-system.

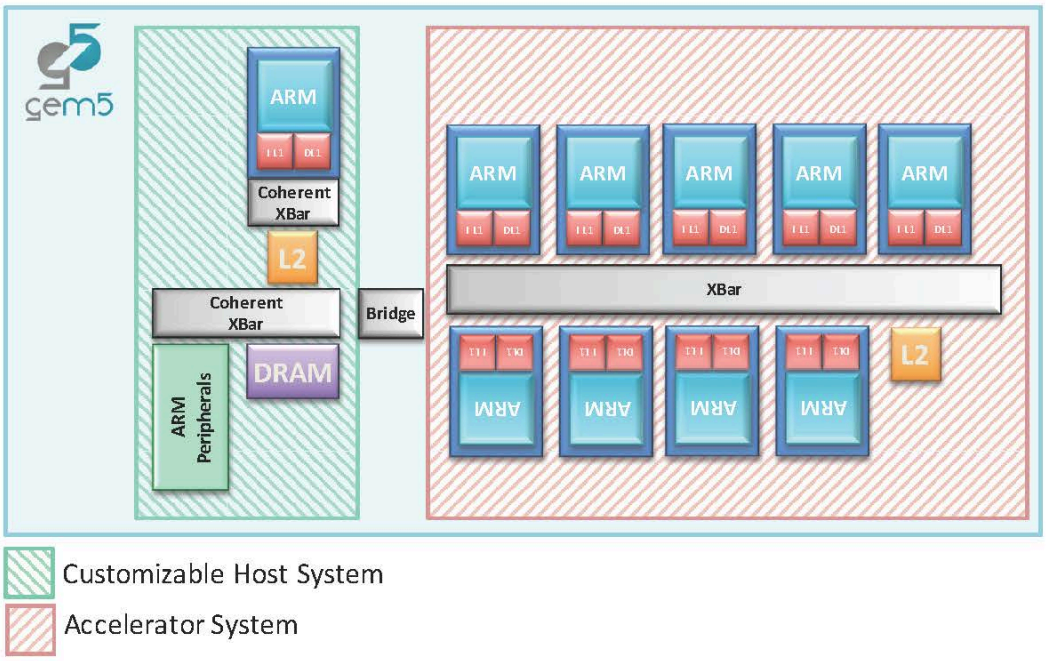


Figure 2.9: Blocks diagram of the whole system (host processor and many-core accelerator) implemented with gem5 simulation environment.

The accelerator is composed by 8 (extended to 9 in some version) ARMv8 clusters and through an internal non coherent crossbar the communication inter cluster is guaranteed. In this case, the memories inside a cluster are I-L1 and D-L1 cache(both private to the core) but I add (not seen in figure) also a TCDM memory per cluster, accessible by all the cores of the accelerator. In figure it is shown just one L2 block, but together with the modifications done on the VirtualSoC platform, I also extended the L2 memory to a multi-banks one. It is important to point out that this memory is shared between all the cores of the accelerator and also reachable by the host system (e.g. it is used to offload code from the host towards the accelerator).

The bridge block has also the important function to let the accelerator directly access (through the XBar-Bridge-CoherentXBar path), to the DRAM memory, but it blocks the attempts to access to the L2 of the host. This latter instead can access everywhere.

All the memory in this gem5 environment is physically mapped, so to access to one address you must explicitly refer to the physical address of the memory block you want to reach. This makes simpler to manage the porting of the applications on this simulator, in particular the offload phase by the host on the accelerator.

To simulate at least the first-level effects of a NoC, although having a simple crossbar (the model of the NoC in gem5 is not well-developed and presents several bugs), I map on the crossbar all the paths and introduce a delay matching the number of hops of the NoC a packet should cross to reach the destination: this let the simulation takes into account the first-level effects as the differences between paths and also the congestion at memory side. Congestion due to arbitration inside a switch is instead neglected.

2.4 Simulation platforms comparison

There are two main differences between the two simulation environments presented in this chapter. On one hand I have a SystemC cycle-accurate RTL equivalent simulator that is VirtualSoC. On the other hand I have an event-driven simulator mainly based on C++ and TLM modeling style. This means that if the experiments I need to run do not need to be accurate but just validating the functionality or the proper execution of an application I can rely on the higher instructions/sec ratio delivered by gem5.

If instead I want to be more accurate, for example to evaluate new hardware blocks or envisioning a next prototyping on FPGA, it is better to rely on VirtualSoC.

This is exactly what I do in the next chapters: to validate a new reconfiguration mechanism (requiring hardware blocks and new kinds of signaling) I use VirtualSoC. Instead, to run several benchmarks not taking care about accuracy I use gem5 because it lets me run real benchmarks with a 70% of speedup compared to VirtualSoC with all the features unblocked. The main features that slow down VirtualSoC simulations are the number of cores per cluster and the NoC routers.

2.5 Summary

In this chapter, I presented the two simulation environments used in this thesis, which are customized and augmented variants of VirtualSoC and gem5. Then, I highlight the benefits and disadvantages of each virtual platform. The key take-away is that there is not a best choice: it depends on research stage, on the research question, and on the system scale.

Chapter 3

Making the point for Space-Division Multiplexing for GPPA

In this chapter, first of all I present the two main approaches to schedule the execution of several applications that are offloaded on the accelerator by the host processor, asking to speedup their execution. Then I want to prove that Space-Division Multiplexing is more suitable for a many-core environment with respect to Time-Division Multiplexing because it enables resource sharing of the manycore fabric through partitioning. Finally, I explore the impact of the size and the shape of spatial computation partitions on a target benchmark.

Key novelty: experimental validation of the SDM approach for hardware resource sharing in many-core programmable accelerators.

3.1 Time-Division Multiplexing vs. Space-Division Multiplexing

When the host system wants to accelerate the execution of applications, an offload procedure begins, enabling the code to be accelerated to be copied or moved to the accelerator memory. When this phase is terminated the Fabric Controller (or hypervisor) inside the accelerator, or again the host system, triggers the execution of the application.

Moving now to a congested scenario, where a lot of applications are requesting to be accelerated, the common and most used way to overcome this situation is to use a *Time-Division Multiplexing* strategy (TDM), that consists of giving access to the

resources of the accelerator to just one application at a time, thus processing in a sequential way all the requests. In this case, the lucky application that wins or that is scheduled first can run but all the others are blocked, waiting for a free time slot to use the resources.

Another approach is to share the resources between the different applications, thus virtualizing the resources and enabling the *Space- (or Spatial-) Division Multiplexing* (SDM). With this approach, applications can be run concurrently but of course they cannot have all the platform available (in this case with the term "resources" I am referring to the computational tiles of a many-core accelerator).

If the speedups due to increasing the number of computational resource were ideal, TDM should be the most competitive approach for the many-core platform domain. But, due to Amdahl's Law and also due to the congestion on the network, from the literature I can deduce that having a large amount of resources is not always reflected into great execution speedups.

So this work moves from the fact that parallelism of application does not ideally scale with the available resources, and probably it is better to have the amount of resources that enabeles the speedup to be closer to ideal, thus not reserving the whole accelerator. This can be exploited using an SDM approach, which means to reserve different partitions to different applications and to enable a concurrent execution to avoid the waste of resources because of inefficient exploitation of the hardware parallelism. To prove this, I show several experiments with real Image Processing benchmarks.

3.2 Experimental evaluation

In this section I first describe all of the experimental setup; then I present the obtained results, first in terms of area and critical path, focusing on the critical path degradation and area overhead, characterizing the cost of the offloading and partitioning of the resources (managed by the Fabric Controller). Furthermore, I provide a comparison for different L2 configurations, then considering several image-processing benchmark runs when varying the number of computational resources dedicated to their execution (thus comparing the speedups given by a different parallelism). Finally, I consider an overall experiment to have a final comparison between SDM and TDM approach.

3.2.1 Experimental Setup

To collect the results shown in the subsections below, as already discussed in Chapter 2, I rely on both VirtualSoC [20], a SystemC-based cycle-accurate virtual platform simulator, and on a virtual platform based on the gem5 simulation environment [17]. This latter option enables to simulate also the ARM64 processors and to speed up the simulations of real benchmarks. The target architecture is the one already discussed in this thesis, in this case configured with 8-9 computational clusters and with various

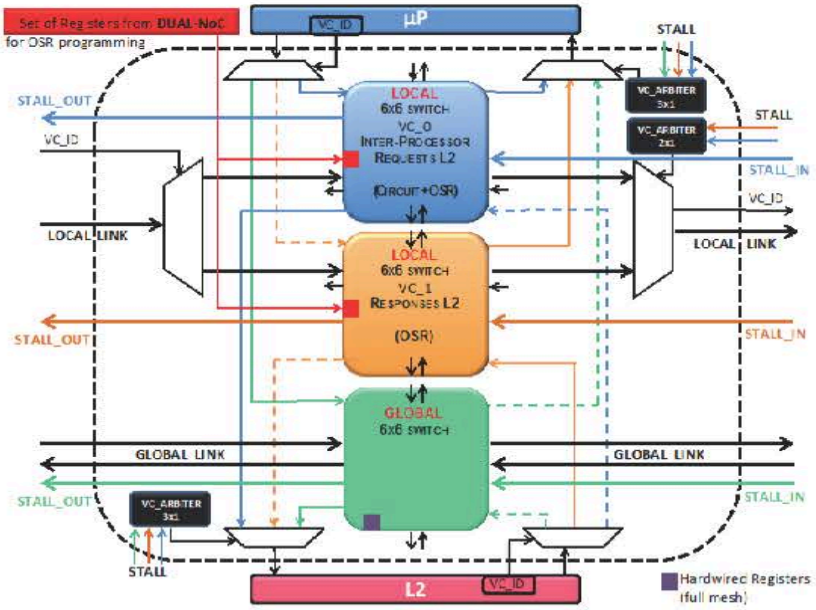


Figure 3.1: Compound switch of the GPPA.

configurations of L2 banks. To let the gem5 simulator match with the requirements for these experiments, I also make the L2 memory of the accelerator sub-system multi-bank and distributed, relying always on the non-coherent crossbar for the communication management. Furthermore I leverage on the implementation in Verilog code of the same platform developed in SystemC to get results about area and critical path, relying on a logic synthesis in a 40nm technology library performed with Synopsys tool.

3.2.2 Area and Critical Path

In this subsection I point out the comparison of effective HW usage as well as performance metrics (e.g., parallelization scaling, performance loss due to NoC sharing), considering the area as well as the critical-path of custom-tailored architectures for the support of Time Sharing vs Resources Sharing. The logic synthesis performed in this work has been carried out by means of a low-power standard Vth-40nm technology library. The target architecture considered for the SDM approach needs a physical global network to let the packets overcome the barriers of the partition, if necessary. In this case we are assuming global network has no VC but in other design we must rely on two physical networks, one local and one global, both with 2 VCs to manage different communication flows. The compound switch considered in this part of the experimental section is reported in Figure 3.1.

In the setup for this experiments, the switch supports two different networks:

- a Local Network, composed of two virtual channels (using the full switch replication approach), is used to serve local traffic within GPPA partitions:
  - Virtual Channel 0 is used for inter-cluster communications and for read-/write requests to L2 memory banks;

  - Virtual Channel 1 is used by L2 memory banks to provide responses to memory access requests;

- a Global Network is used to serve global network-wide communication traffic while avoiding interference with intra-partition local traffic. This network is used to program the L2 banks with the execution code for each cluster, and to feed processing data to clusters. Since we currently envision all communication flows on this network to be made up of write transactions (i.e., offload code is written into the GPPA, while data is fetched by clusters by programming DMA transfers through DMA devices which are external to the GPPA, and which will in turn write into the GPPA), there are no request- response dependencies, hence virtual channels are not required on this network to avoid message-dependent deadlock.

The top switch in Figure 3.1, associated to virtual channel 0, is provided with a round-robin arbiter and supports an optimized version of OSRLite(Overlapped Static Reconfiguration), to enable runtime reconfiguration of the routing function, and circuit switching (circuit is established at runtime and enables a full bandwidth reservation). The middle switch, associated to virtual channel 1, has the same functionality of the VC0 one except for the circuit switching support, not needed. We in fact currently envision guaranteed throughput communication only for cluster-to-cluster communications (in the form of write transactions). The bottom switch is a baseline switch used for global communication: it doesn't need circuit switching, nor reconfiguration support because its routing function is hardwired. It has actually separate links with respect to the former two switches, thus building up the 2 network architecture.

Finally, routing reconfiguration bits are carried to the first network (for intra-cluster communication) through a dual interconnect fabric. Note that the VC's arbiter blocks are characterized by enabling flit-level arbitration as opposed to internal switch arbiters that only perform packet-level arbitration. This is a requirement of the multi-switch approach to virtual channel implementation.

In order to evaluate the area complexity of each architecture (Time Sharing vs. Resources Sharing), I accomplished a 4x4 mesh synthesis at 500 MHz for both solutions. Results are reported in Figure 3.2 and are normalized with respect to the time sharing architecture. The complexity gap between time sharing and resources sharing is around 58%. The most area overhead contribution comes from the additional virtual channel (~34%). A non-negligible contribution (~15%) results from OSR_{Lite} and circuit switching logic. The dual-network results lightweight since it introduces an area overhead of 6%. The remaining overhead (3%) comes from the wrappers (mux and demux) and virtual channels arbiters, that are a little bit larger than the ones of the time sharing architecture (mux 3x1 vs. mux 2x1 and arbiter 3x1 vs. arbiter 2x1).

In order to analyze the delay complexity, I performed a 4x4 mesh logic synthesis at maximum performance for both the solutions. As a result, both platforms achieved a similar maximum operating speed around 750 MHz. Indeed, the $\mathrm{OSR}_{\mathrm{Lite}}$ reconfigura-

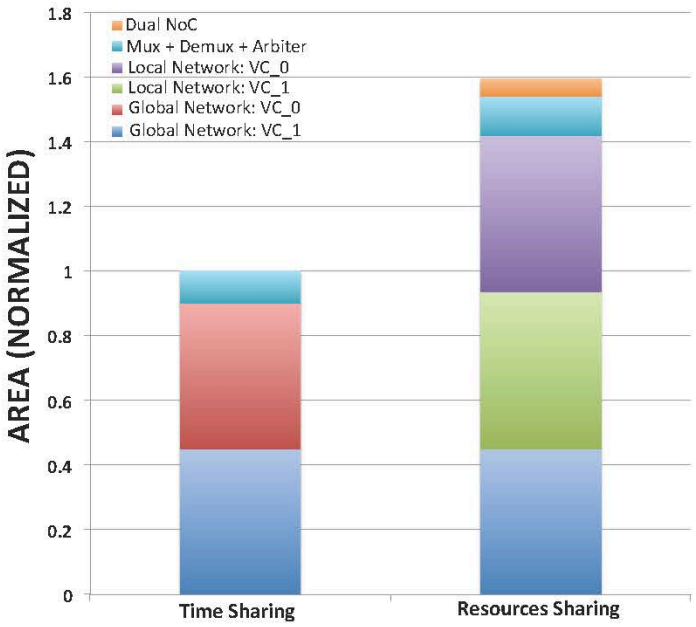


Figure 3.2: Normalized area: comparison between Time and Resource Sharing architectures. SDM approach needs a physical global network to let the packets overcome the barriers of the partition, if necessary. In this case we are assuming global network has no VC

tion scheme was designed to avoid long critical-paths and preserve the baseline switch performance timing.

3.2.3 Offload and Partitioning Cost Characterization

Here I describe the characterization of the offload procedure in terms of relative clock cycle inflation w.r.t. the basic offload scheme where there is no bridge at all, but just a locked driver that refuses multiple offload requests at the same time.

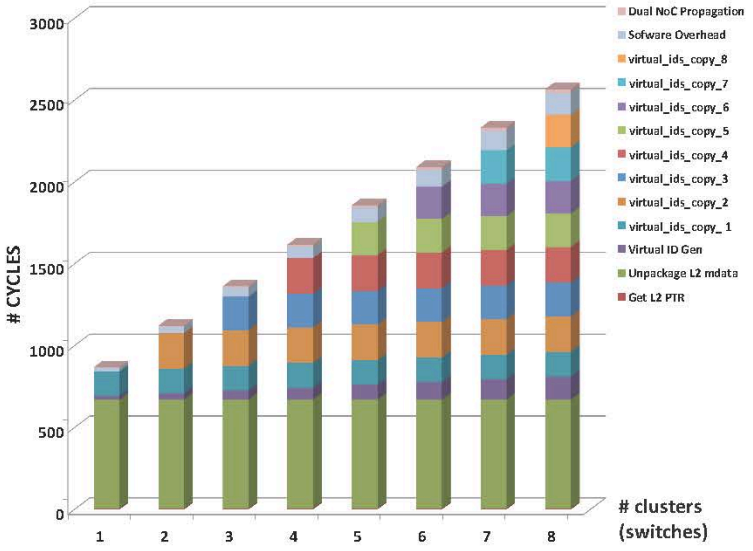


Figure 3.3: Offload costs chracterization increasing the number of cluster involved.

The Figure 3.3 presents the costs regarding the full offload procedure. I perform an evaluation of these costs increasing the number of clusters used by a partition. Each bar corresponds to a single offload and it shows all the different components need to properly setup and configure the platforms. The first step consists of the *mdata* unpackaging in order to prepare it for the kernel execution. In fact the kernel expects as arguments a list of shared variables pointers while the offload task descriptor presents a list of mdata. Thus, the FabricController (FC), or hypervisor, creates a list of this pointers reading all the mdata structures retrieving the corresponding pointers to the shared variables. This phase cost is independent by the partition size and it takes around 700 cycles (for only two shared variables!!! Bad!).

At this point the Fabric Controller (or local hypervisor) creates the proper virtual accelerator setup and it generates the virtual IDs for all the cluster involved.This phase is linear depended by the partition size and it grows up to 200 cycles for 8 clusters. As soon as the accelerator information are generated the FC pushes this configuration to each cluster TCDMs. This phase gets around 250 cycles for each transfer. The plot presents this cost separately for each copy in each cluster needed. Finally the FC setups the NoC and the plot presents this cost as two components: the software cost for NoC reconfiguration API call and the Dual NoC Propagation. The latter one presents a very low impact considering the whole scenario.

3.2.4 Application Benchmarking

Effects of Parallelism on Speedups

Here I am going to show for each application the level of supported parallelism dependending just on how the code is written, thus evaluating if in the benchmark it is predominant the portion of the code that can benefit from the improvement of the resources of the system, or if it is the sequential part being the most preponderant, thus nullifying the increased number of resources available for the execution. According to the Amdahl's law, indeed, the speedup a workload can gain increasing the number of dedicated resources, thus parallelizing its execution, is as farther from the ideal as the percentage of the parallelizable code is lower. To let the simulations show this, I need a particular setup of the platform configured both to minimize the time to access to the memories and to neglect multi-hop paths in the on-chip network, overlooking the arbitration due the congestion on a link, too. Thus, I can monitor the ideal speedup of the application, apart the contention at memory ports, entrusting just on an ideal crossbar the communication between computational clusters and memory banks and shortening the latencies to access to all the memory levels to only 1 ideal cycle. Figure 3.4 exhibits the results of the simulations for the applications considered, compared to the ideal speedup represented by the dotted black line. As depicted in the graph, there is one benchmark (ROD) that presents a speedup very close to the ideal case

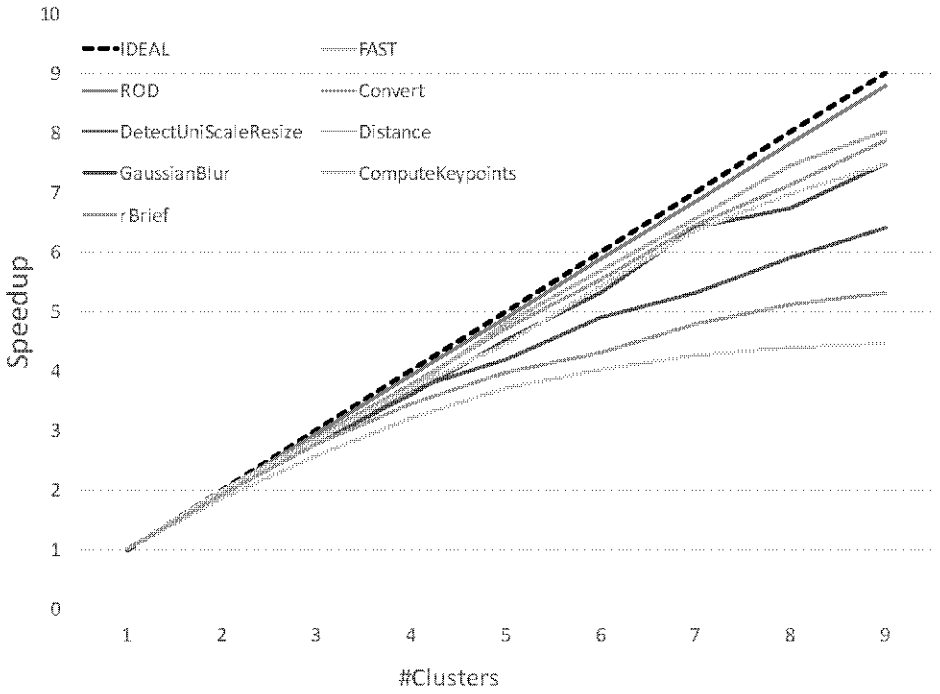


Figure 3.4: Speedups showed at increasing the number of computational clusters reserved for the application under test, considering, as platform setup, an ideal crossbar and all the memories access latency equal to 1 cycle.

and a set of four benchmarks with an almost ideal speedup especially for a low numbers of clusters and an acceptable degradation considering more aggressive parallelism. Finally a batch of three benchmarks, FAST-Rosten all over each others, emphasizes an evident degradation compared to the ideal case. Strengthened by these results, it is evident that for most part of the evaluated benchmarks the approach pursued by Time Division Multiplexing strategy, dedicating all the computational resources of the accelerator to the execution of a single application with the most aggressive level of parallelism, is not the best choice, because the speedup curve is targeted in a point the farthest from the ideal speedup. On the contrary, targeting a configuration of the partitions with less computational cluster allocated let the target move to a point in the curve closer to the ideal case, furthermore enabling the parallel execution of other applications at the same time because availability of part of the platform and thus optimizing the platform's resources usage.

Speedups at different L2 configuration

In this subsection I analyze several possible configuration of the L2 memory of the many-core accelerator, evaluating what is the best case concerning the resources sharing approach usage of the platform. In particular I compare the following configurations:

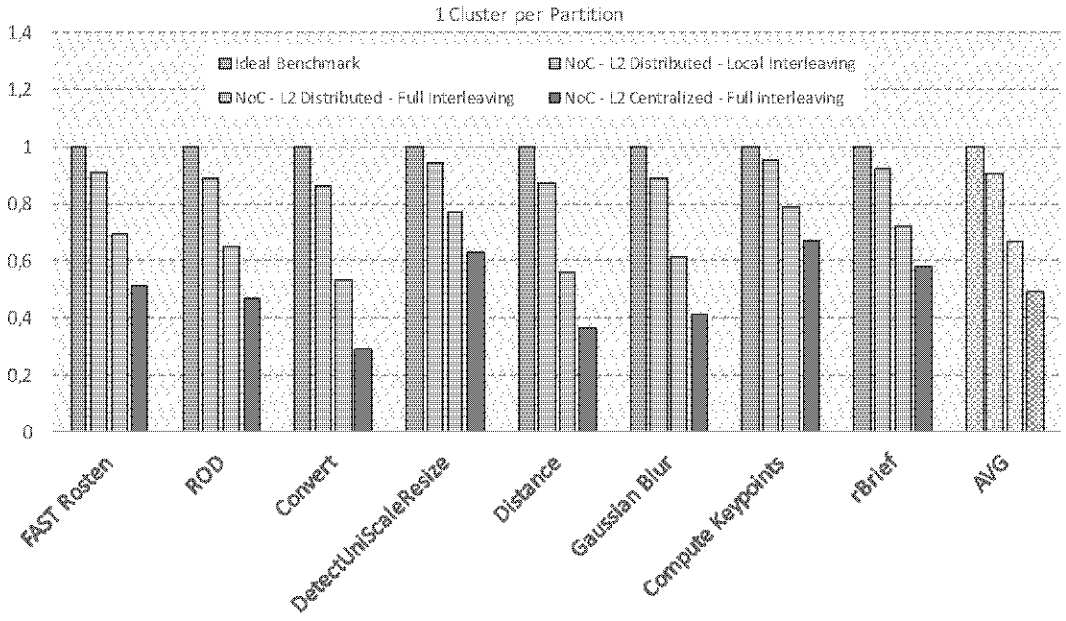
- Centralized L2: this configuration refers to the case of a three L2 banks (one per row of the mesh) relying on full interleaving between them. In case of fine partitioning it is unfeasible reserving one bank per partition because after reserv-

ing three partitions and having limited L2 banks means running out of available banks, making part of the accelerator unusable. Where possible, i.e. in a special case of the partitions mapping when I consider three partitions of three clusters, I also take into account a configuration that envisions one bank per partition, being this not a limit because I am using all the resources.

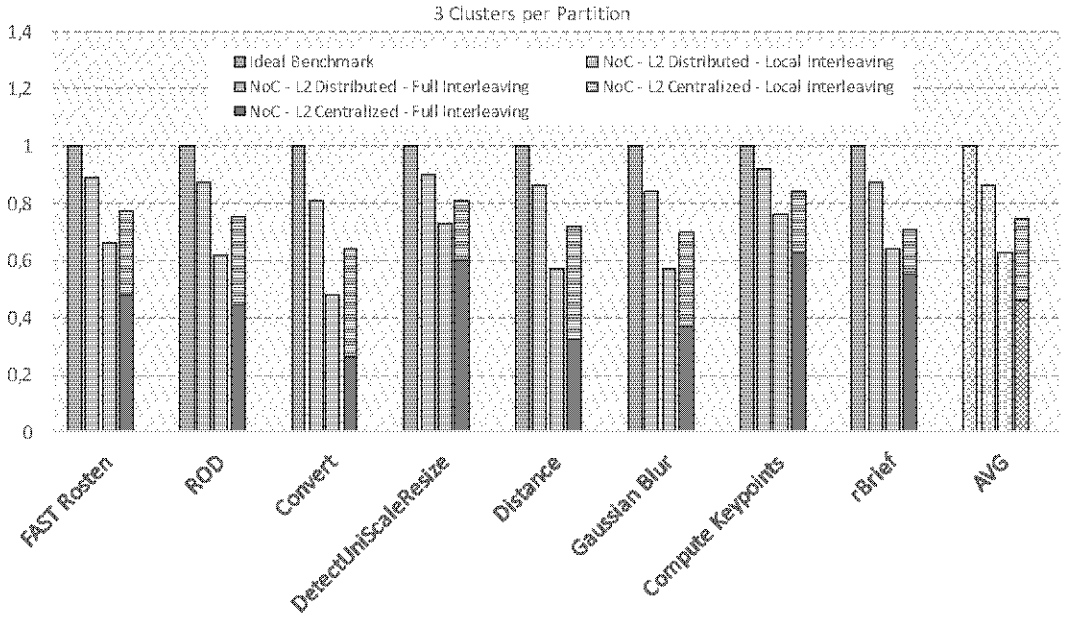
- Distributed L2 with full interleaving: in this case I am referring to a distributed multi-bank L2, spread in all the accelerator one bank per each NoC node and with addresses interleaving (in particular cache-line interleaving). During the execution of the benchmarks there is only a partitioning of computational resources (clusters) and the L2 memory is reachable from all the clusters through a global network.

- Partitioned distributed L2 with interleaving: I am considering the same configuration described in the previous point but in this case I add a partitioning of the memory, too; during the execution of an application all the code needed is locally reachable because it is loaded in banks inside the partition. Furthermore the addresses interleaving is limited to the banks inside the partition boundaries, thus guaranteeing total isolation of traffic running in a partition, apart the communications with DRAM.

Figure 3.5 shows the results of the different configurations considering the execution of each benchmark for partitions of one cluster (a) and three clusters (b). Finally, I also provide an average taking into account all the benchmarks speedups. We compared the different configuration described above normalizing the results to the ideal benchmark execution. The configuration that presents a multi-bank L2, distributed on all the accelerator nodes with the possibility of partitioning it in the same way of the computational resources also the memories, thus guaranteeing isolation of the on-going traffic inside the partition during the execution, is the best choice. This configuration is possible because of our hardware support allows the isolation of the traffic due to the partitioning of both clusters and memory banks, improving the results of baseline configurations and thus getting close to the ideal speedups of the benchmark. The distributed L2 with full interleaving configuration presents worse results because it is affected by the perturbation of the traffic from other partitions, thus nullifying the benefits of interleaving approach, and because of NUMA effects to reach the memory banks distributed over all the accelerator. The worst configuration is represented by the centralized L2: in this case, indeed, the performance is still decreased because of NUMA effects, depending on the latency to access different banks, and also because the perturbation due to the traffic from other partitions plays an inauspicious effect. Furthermore there is more congestion to access the memories because they are centralized. In figure 3.5(b), I consider also the improvement provided by isolating the L2 banks inside the partition, presenting this approach as local interleaving. As shown, there



(a)



(b)

Figure 3.5: Applications speedups normalized to ideal benchmark execution to evaluate different L2 configurations: (a) 1 cluster per partition, (b) 3 clusters per partition.

is a significative improvement of performance because it is clear that the perturbation coming from other applications is the most significant for the lowering it. Focusing on the average results, considering 1 cluster per partition, our experiments show that centralized approach causes a decreasing speedups of 51% compared with the ideal case. Introducing a full interleaving approach and a distributed L2 on all the accelerator nodes provides a soft improvement making the slowdowns reaching the 41%, because again the perturbation on the NoC link and the congestion to access memory banks is high. Finally, distributed approach with local interleaving, enabled by our hardware support, let the slowdowns decrease to 4%, and the performance get close to the ideal case, making using isolated partitions the winning strategy. The trends are confirmed

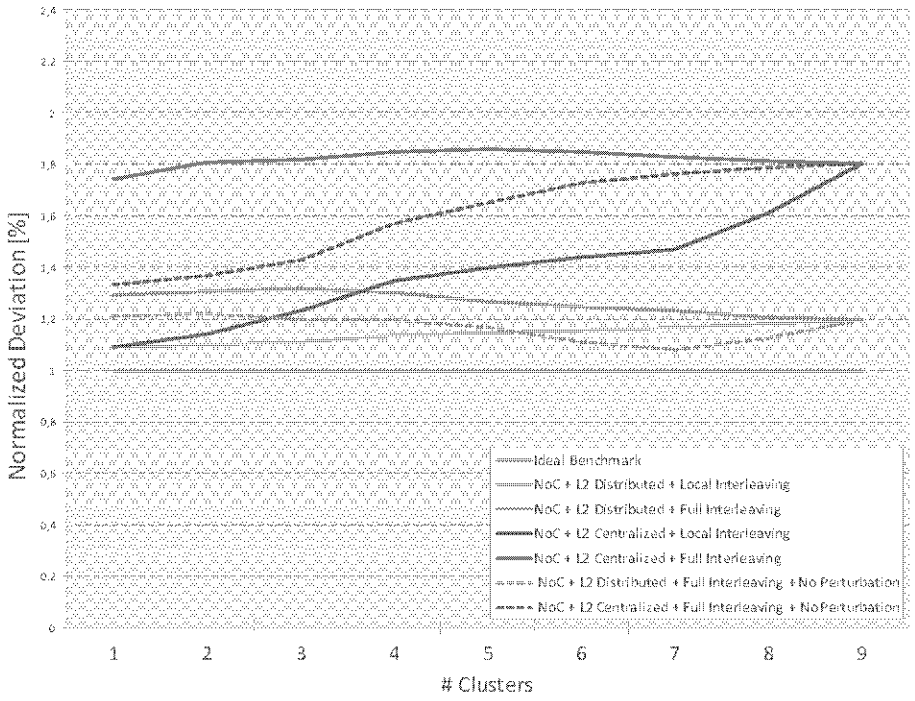


Figure 3.6: Deviation normalized to the ideal Fast-Rosten performance for several L2 configurations.

also considering the experiments with 3 clusters per partition. In this case, I add in purple also the configuration of centralized L2 but isolated inside the partition. Again, isolating the memory banks from the perturbation of other partition let the speedups improve up to being better than the distributed approach with full interleaving.

It is interesting to analize deeply the comparison between a distributed L2 with full interleaving vs. centralized L2 with isolation. Figure 3.6 depicts the comparison of all the possible L2 configurations just considering Fast-Rosten (the trends are the same also for the other benchmarks). The percentage of the deviation from the ideal benchmark is lower considering the centralized and isolated approach for a fine-grained partitioning, being for the case of 1 cluster per partition the same of the best approach that is proven to be the distributed with local interleaving one. Increasing the number of clusters the graph shows a breakeven point that is from 3 and 4 clusters, meaning that over 3 clusters the benefits of having a distributed L2 memory with full interleaving gain the upper hand.

3.2.5 Overall scenario: does SDM make sense?

Finally I consider a batch of benchmarks, in particular considering nine instances of the eight benchmarks (overall having a batch of 72 applications).

Figure 3.7 shows the execution time of all the batch considering a SDM approach with different configurations of the L2 memory , normalized to the TDM approach. While in this latter I am reserving all the accelerator resource to the applications, running them sequentially, in the former approach I am considering an aggressive

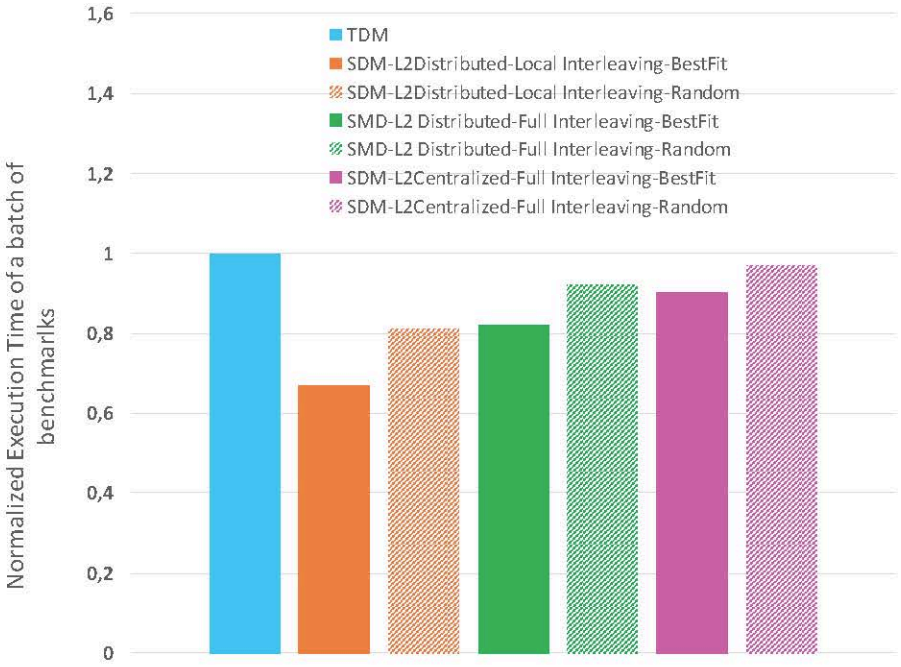


Figure 3.7: TDM vs SDM-Random vs SDM-BestFit with different L2 configurations.

virtualization relying on fine-grained partitioning that means having one cluster per partition.

The graph points out that with the best optimization of the L2 memory (distributed with local interleaving) I can reach up to 35% of speedup with SDM approach considering the best-fit scheduling of the application execution and an average of 19% considering a random scheduling.

Figure 3.8 represents the gaussian distribution of the results obtained with 100 simulations of the random scheduling, highlighting that the great part of the samples are on the left respect to the average value shown in the graph and so that the average shown is a plausible value, strengthening the assumption proposed in this chapter that is that SDM performs better than TDM in many-core environments.

3.3 Towards a resource virtualization environment: partition shapes make the difference

Here I study how the size and the shape of the partitions can affect the speedups of the benchmarks, relying on VirtualSoC simulation environment and considering just a parallel vesrion of Fast-Rosten [118] as monitored benchmark, configuring and modifying the OpenMP Runtime to enable the parallelism of the benchmark and mapping in the proper way the cores that are part of the team involved in the computation, being inside the considered partition.

Furthermore, to automate the LBDR bits generation, needed to setup new partitions and the routing restrictions, I create a script that can generate the list of all the

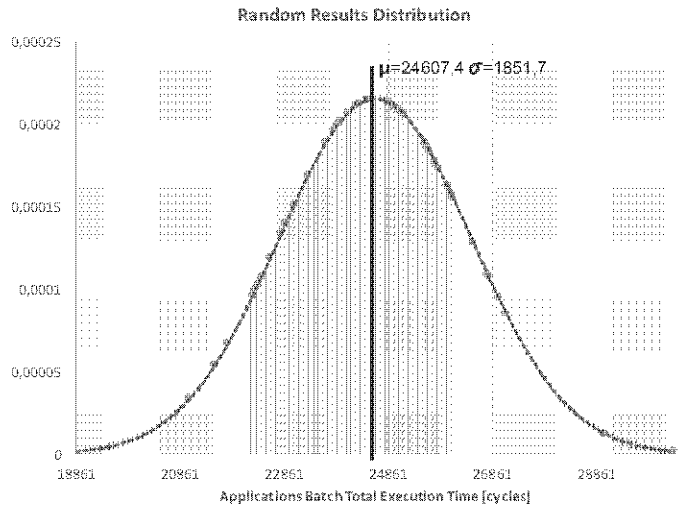


Figure 3.8: Gaussian Distribution of the random results.

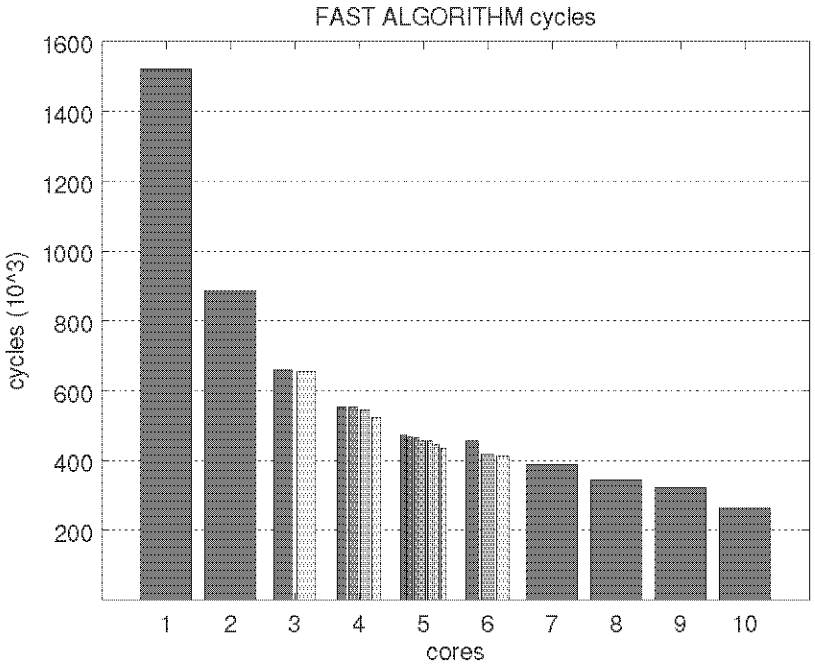


Figure 3.9: Execution times of FAST benchmark considering an increasing parallelism and dedicated computational resources.

needed bits just configuring the partition I want to create.

Figure 3.9 shows the execution time of the benchmark increasing the number of computational resources dedicated. The behaviour is the one expected: the execution time decreases increasing the parallelism.

However, as deeply studied in the second section of this chapter and shown in Figure 3.10, the speedups are not ideal and become worse increasing the number of clusters involved due to Amdahl's Law and to the overhead of communication (data and instruction are potentially in parts of the memory that are accessible by all the other cores and so congestion can occur).

Figure 3.11 represents instead a zoom on partitions of size 4, 5 and 6, reporting

several shapes named as the alphabetical letter that match with their footprint. It is surprising, as shown, that partitions with less cores but with a more suited shape present better results in terms of execution cycles to run to the end the benchmark. This means that the **shape of the partition makes the difference and it is not**

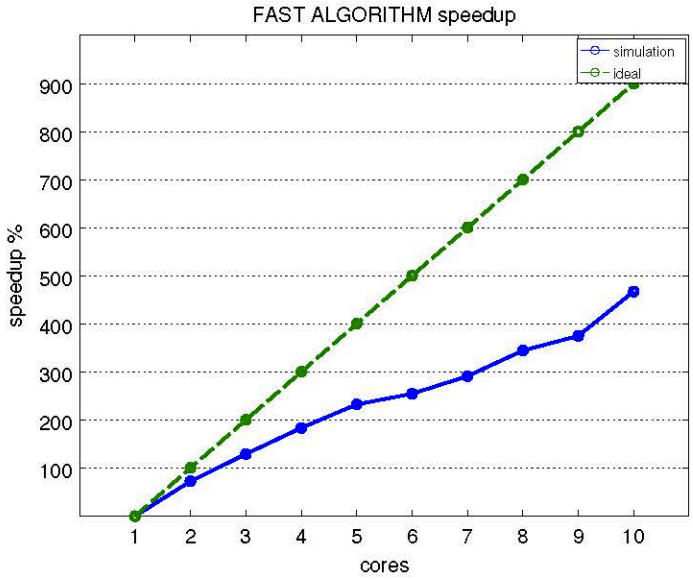


Figure 3.10: Speedups of FAST benchmark considering an increasing parallelism and dedicated computational resources: the values reported are the average of results for partition that can support more than one shape.

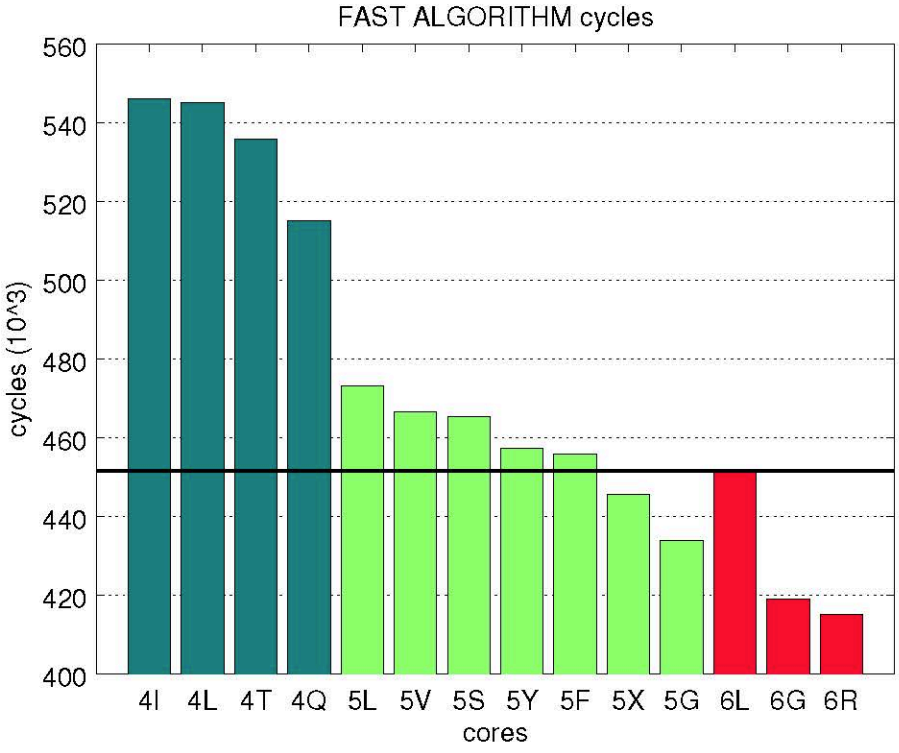


Figure 3.11: Execution times of FAST benchmark considering an increasing parallelism and dedicated computational resources: zoom on partitions of 4-5-6 clusters.

a trivial choice for the hypervisor that is responsible for the virtualization of the many-core system, because it determines how the communications flows are located in the network with the possibility to have more congestion and thus slowing-down the execution of the benchmarks.

3.4 Summary

In this chapter I evaluate the SDM approach to schedule several applications that require to be accelerated on the GPPA, proving that resource sharing through SDM approach, enabling the concurrent execution of several applications at the same time, has more benefits than reserving all the platform in time slots, as the traditional TDM strategies do. This is due to the fact that the parallelism of the application does not ideally scale with the increasing number of reserved resources, therefore TDM is inefficient because you keep computational blocks busy without getting the expected improvements in speedups. Finally, I prove also that size and shape of partitions to be set up are not trivial choices and can have a significant impact over the execution speedups of the benchmarks.

Chapter 4

How to Support SDM

In this chapter, first of all, I introduce software support for the many-core GPPA device, mentioning also the OpenMP Runtime that is used to parallelize the execution of the applications in the manycore fabric (thus running the experiments in Chapter 3), configuring the runtime environment to different teams of cores, this way matching also the size and shapes of the partitions created on the accelerator. Then I focus on hardware issues, showing several scenarios enabled by the SDM and the related possible issues that can be solved by augmenting the hardware with supporting features. Finally, I pinpoint a strategy to overcome these problems..

Key novelty: contribution of new design methods for SDM-enabled GPPA architectures.

4.1 Software support for SDM in a virtualized environment

Here I describe some of the software features to be enabled to exploit the potential of partitioned many-core accelerators, and used in this thesis.

4.1.1 Operating System and Hypervisor

In this section, the Operating System support that allows the sharing of a many-core accelerator is described. This infrastructure has been developed for KVM based virtualization systems, but can be easily extended for other hypervisors (e.g. XEN). In KVM I/O is completely emulated and not handled by the hypervisor itself, but is rather demanded to QEMU [13] which is the standard KVM machine emulator. Based on the last assumption, the introduction of a new virtualized devices does not involve any modification to KVM. Our main idea is to share the accelerator between different virtual machines in a space-wise fashion, with each virtual machine having an

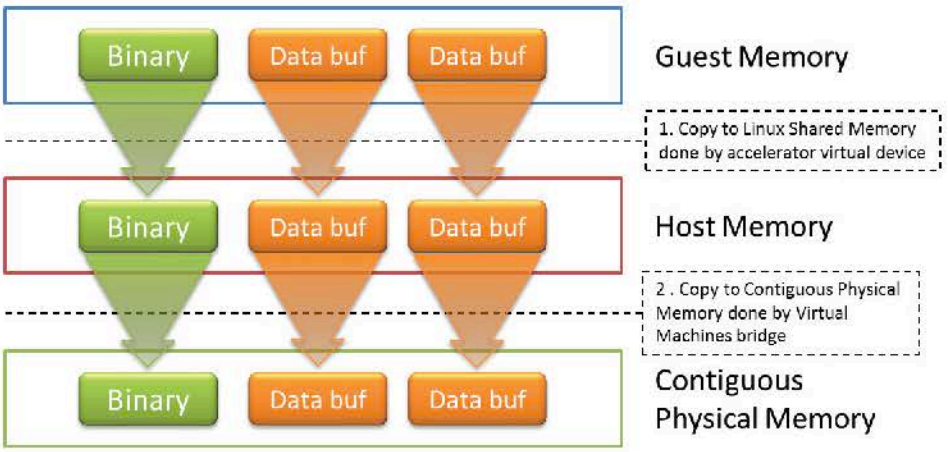


Figure 4.1: Memory copies to allow data sharing

isolated view of the accelerator with respect to the others. This is to allow offloaded kernels coming from different virtual machines to run concurrently onto the accelerator. Each Virtual Machine (executed by a separate QEMU process) will have the idea of a dedicated accelerator, thanks to a virtual device($vACC$) emulating the behavior of the accelerator. The virtual device is an extension of the architecture modeled by QEMU and is accessible by the guest operating system as a standard character device ($/dev/v\_acc$). Guests are also equipped with a Linux device driver ($acc\_vdriver$) communicating with the virtual Device. From the host system point of view the accelerator is a standard Linux device, appearing in the system as a character device ($/dev/acc$). The host has a complete view of the accelerator and is aware of the availability of all its resources(e.g. number of free clusters, memory). To handle concurrent access requests coming from different virtual machines, I implemented a bridge process (ACC *Bridge*) application which, running in the user-space of the Host system, is in charge of collecting all requests and forwarding them to the real Accelerator device through a Linux device driver ($acdriver$).

The main issue tackled in this framework is data sharing between a virtual machine and the accelerator. When offloading a kernel, the binary to be executed and any possible shared data buffer have to be visible to the accelerator. One assumption is that the accelerator is not able to deal with virtual memory references, thus only physical contiguous memory can be used to share data. During the boot of the host system a subset of the system memory (256 MB) is reserved, which will be addressed physically from both host and accelerator. Each time data sharing is needed, data is copied first from the Guest Virtual Memory space to the Host virtual memory space (Figure 4.1) and finally into the contiguous shared memory segment.

The whole framework is composed by the following layers (the description follows Figure 4.2):

1. *Acceleretor's virtual device driver*: The Accelerator Virtual driver is located at the very top of the virtualization stack presented in this work. It is used to

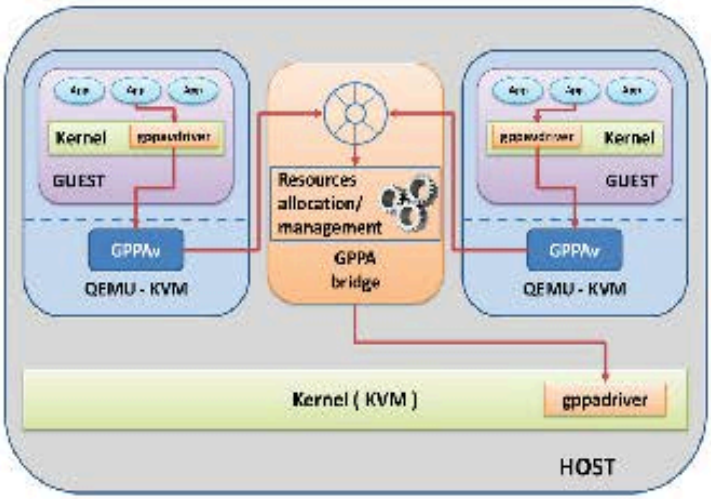


Figure 4.2: Many-Core Accelerator sharing infrastructure

give each guest Operating System the illusion of a dedicated Accelerator device. Applications communicate with the driver using the Linux ioctl system call. The interface implemented via ioctl defines the following services: *Task Offload*, *Wait Task Completion*. The first is used to offload a task to the accelerator, and takes as parameter the pointer to a task offload descriptor. The second service is used by applications to wait for the completion of a specific task.

2. *Accelerator emulation device (vACC)*: the emulation device is a software module which is developed as an extension of QEMU. Once designed each virtual device is attached to the bus of the platform modeled by QEMU and mapped at a user-defined address range. Any i/oread\i/owrite call made by applications running on a guest operating system, and falling within the address ranges where the custom devices are mapped, is caught by QEMU and redirected towards the virtual device. Each vACC device is interfaced with the ACC Bridge using POSIX queues, which are an Inter Process Communication mechanism provided by Linux-based systems. In particular, I define a single POSIX queue for messages going from guests to the ACC bridge, while a POSIX queue per virtual machine for messages coming back from the ACC Bridge. When the device is initialized, it first creates its private message queue, whose unique reference in the system is composed by the string "\queue." concatenated with the PID of the QEMU process. The virtual device is then attached to the shared POSIX queue. Whenever a request from a guest arrives to the vACC, it is immediately forwarded to the ACC Bridge using the shared POSIX queue. Before actually forwarding the request, the first copy takes place (Figure 4.1). For convenience data is copied from the Guest virtual memory to a Linux Shared Memory. I use

3. *ACC Bridge*: The ACC Bridge is the heart of the proposed virtualization infrastructure. In this module all decisions regarding the sharing of the Accelerator are taken. This module is a server process composed by two POSIX threads,

in charge of forwarding requests to the real Accelerator and providing responses to the various Guests, respectively. At startup this process creates the shared POSIX message queue used by all guests to push offload requests to the GPPA. This queue has a unique name inside the system which is known to all guests. The bridge accepts three possible commands from Virtual Machines: REGISTER_VM, OFFLOAD_TASK, TASK_END. The first one is used to register a new virtual machine running on the system. The second is used to actually request the offload of a task. The last command is sent by virtual machine waiting for the completion of a specific offloaded task.

4. *Host accelerator driver (acc_driver)*: This is the lowest level of our framework. The accelerator host driver is in charge of communicating with the real accelerator, receiving offload requests from the ACC Bridge. The request comes into the form of a pointer into the contiguous shared memory to the pointer of a task offload descriptor. The pointer is pushed into the tasks queue of the fabric controller.

4.1.2 OpenMP Runtime for cluster virtualization and parallel execution

Cluster-based architectures are increasingly being adopted to design embedded many-cores. These platforms can deliver very high peak performance within a contained power envelope, provided that programmers can make effective use the available parallel cores. This is becoming an extremely difficult task, as embedded applications are growing in complexity and exhibit irregular and dynamic parallelism. The **OpenMP** tasking extensions represent a powerful abstraction to capture this form of parallelism.

In this thesis I rely on a optimized runtime environment [23] supporting the OpenMP tasking model on an embedded shared-memory cluster.

OpenMP 3.0 introduces a task-centric model of execution. The new "task" construct can be used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When a thread encounters the task construct, it prepares a task descriptor consisting of the code to be executed, plus a data environment inherited from the enclosing structured block.

"shared" data items point to the variables with the same name in the enclosing region. New storage is created for "private" and "firstprivate" data items, and the latter are initialized with the value of the original variables at the moment of task creation. The execution of the task can be immediate or deferred until later by inserting the descriptor in a work queue from which any thread in the team can extract it. This decision can be taken at runtime depending on resource availability and/or on the scheduling policy implemented (e.g., breadth-first, work-first). However, a programmer can enforce a particular task to be immediately executed by using the if clause. When

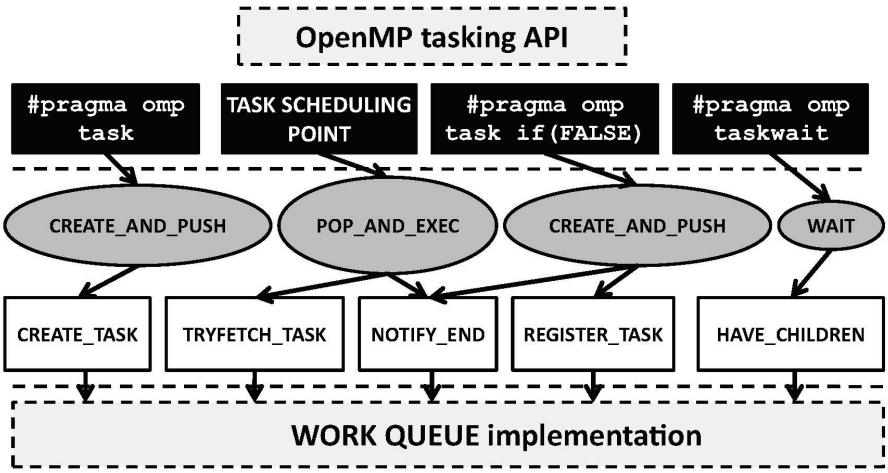


Figure 4.3: Design of tasking support.

the conditional expression evaluates to false the encountering thread suspends the current task region and switches to the new task.

On termination it resumes the previous task. Specifications also enable work-unit based synchronization. The "taskwait" directive forces the current thread to wait for the completion of every tasks generated from the current task region. Task scheduling points (TSP) specify places in a program where the encountering thread may suspend execution of the current task and start execution of a new task or resume a previously suspended task.

Figure 4.3 shows the layered approach to designing the primitives for the tasking constructs. These constructs are depicted in the top layer blocks (in black). To manage OpenMP tasks, the optimizations proposed rely on a main work queue where units of work can be pushed to and popped from (bottom layer block). The gap between OpenMP directives and the work queue is bridged by an intermediate runtime layer (gray blocks), which operates on the queue through a set of basic primitives (white blocks) to implement the semantics of the tasking constructs.

This design relies on a *centralized queue* with breadth-first, LIFO scheduling. Tasks are tracked through descriptors which identify their associated task regions and which are stored in the work queue. The two basic operations on the queue are task insertion and extraction. Inserting a task has two effects: i) creating a new descriptor for it, and ii) registering it as a "child" of the executing task (its "parent").

Let me consider the simple example of the task construct in the code snippet of Figure 4.4. The parallel directive creates a team of worker threads, then only one of them executes the single block. This thread acts as a work producer, since it is the only one encountering the task construct. The control flow for the rest of the threads falls through the parallel region to the implied barrier at its end.

The most important part of the implementation of the tasking execution model is Task Scheduling Points (TSP). Parallel threads are allowed to switch from one task to

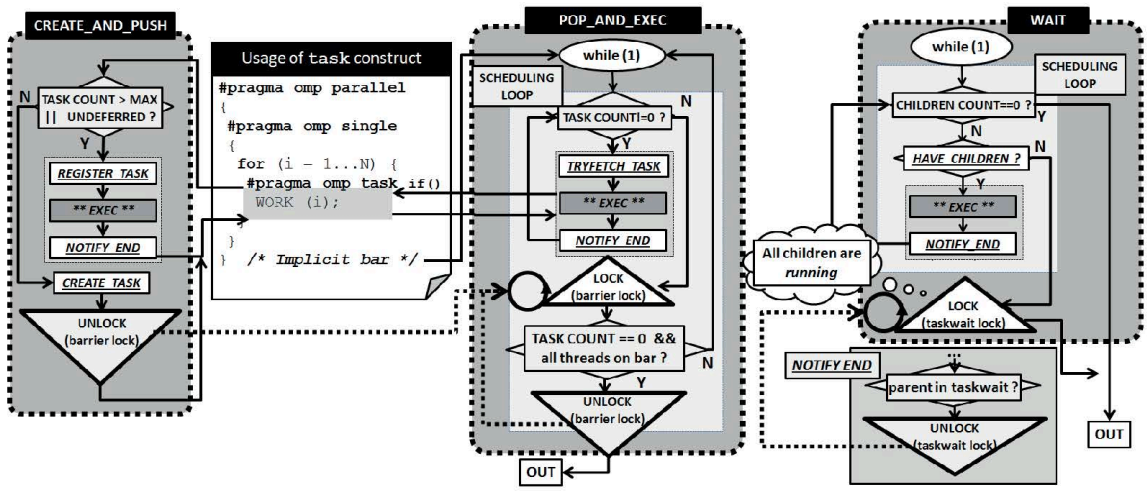


Figure 4.4: Design of task scheduling loop.

another:

- at task constructs;
- at implicit and explicit barriers;
- at the end of the current task;
- at taskwait construct.

In this thesis, I use the OpenMP runtime environment just briefly described, and setting the composition of the teams let me manage to run several applications with different levels of parallelism.

4.2 Hardware support for SDM in a virtualized environment

Space-Division Multiplexing is a scheduling approach for the execution of the applications, that need to be accelerated, that enables virtualization of the resources and resources sharing. The usage requirements, that means having a lot of applications that to the limit can run concurrently on the same platform, makes the environment highly dynamic.

As Figure 4.5 shows, enabling effective virtualization implies:

- some form of partitioning;
- isolation for protection;
- partition scheduling;
- partition reshaping;

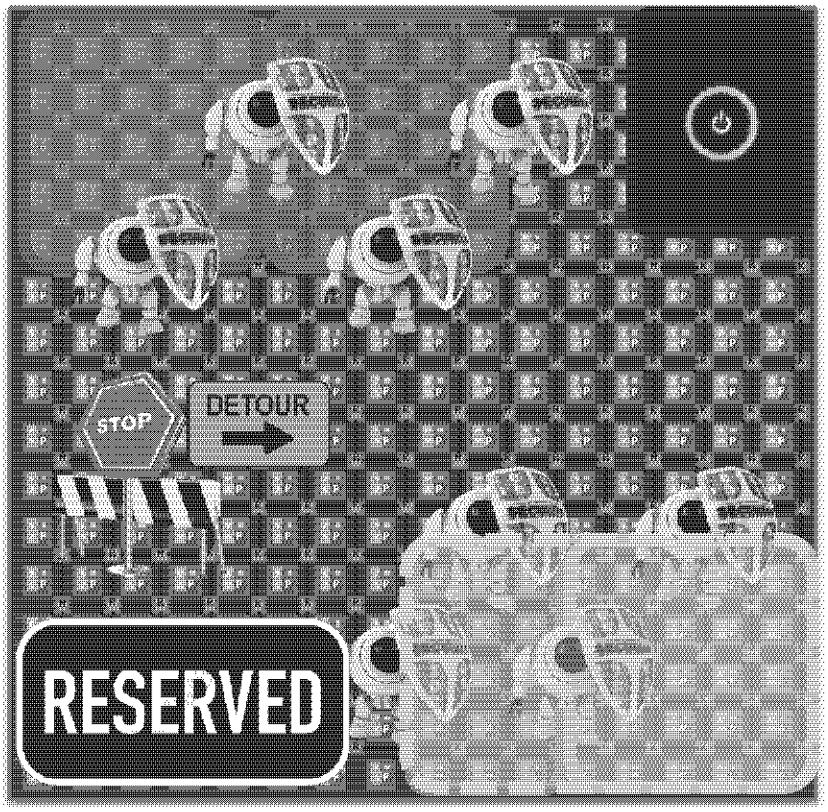


Figure 4.5: Highly dynamic environment.

- avoiding faulty links or switches
- powering-off unused or overheated regions
- setting up or tearing down of reserved paths (hard QoS)

In this scenario, *the isolation property* is a fundamental requirement: the traffic generated by different applications can collide in the NoC of the GPPA (as shown in Figure 4.6) as the NoC paths are shared between nodes assigned to different applications (a) but even for smart allocation schemes (b). Smart allocation is not enough: there is the need of true partitioning with partition isolation.

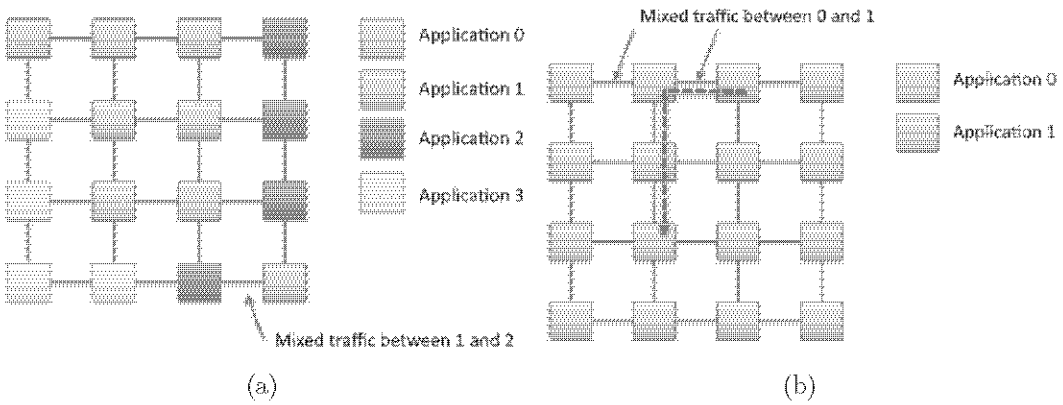


Figure 4.6: Violations of the isolation property.

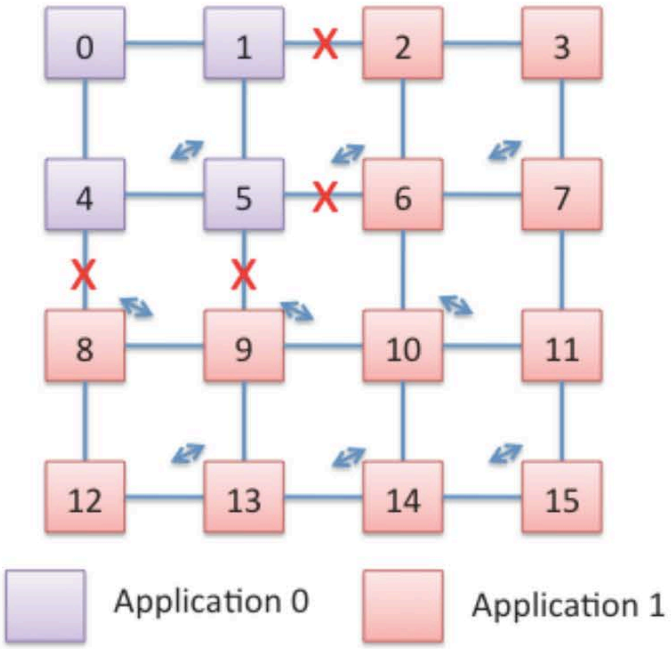


Figure 4.7: Partitioning support through connectivity bits.

A basic partitioning support is setting connectivity bits to zero at partition boundaries prevents messages from escaping from their partition (Figure 4.7): this solution has side benefits like having a complexity scaling with switch radix and not with network size, no modifications of the routing algorithm and no additional provisioning to guarantee deadlock freedom are required (as long as routing bits are not touched, the routing algorithm is not affected!), and finally no virtual channel are needed (yet).

The problem with this basic approach is that not all the partition shapes are feasible. This is due to the fact that there is a mismatch between partition shapes and the underlying routing algorithm and in fact another routing algorithm works for the same partition shapes! There are two possible solutions: i) setting up only those partition shapes that are legal for the chosen routing algorithm; ii) adapting the routing algorithms to the partition shapes.

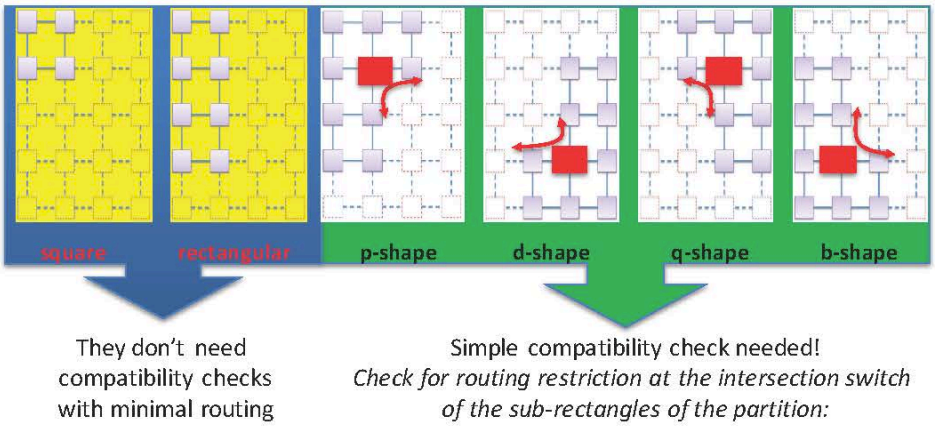


Figure 4.8: Mapping Restrictions to avoid unsafe partitions.

The first solution relies on the fact that the hypervisor will avoid incompatible mappings with a specified global routing algorithm, like defining predefined partition shapes (as depicts in Figure 4.8).

Another disadvantage arises with this solution: in fact this approach may reject allocation requests in heavily loaded scenarios, although enough free resources are available.

Considering the second solution, it relies on the fact that the hypervisor will avoid the rejection of an incompatible mapping request by adapting the underlying routing algorithm. (Figure 4.9). Special care should be devoted to this scenario in particular that previously compatible running partitions may become incompatible, deadlock may arise as an effect of the temporary coexistence of two routing functions and furthermore runtime reconfiguration procedure may induce performance perturbations in running partitions.

So finally the solutions to this problem are mainly two:

- having **Global routing without VCs**, that means tha tthe global traffic and the intra-partition traffic must follow the same routing algorithm that does not change, avoiding deadlock
- relying on **Per-partition routing with VCs**, where Different algorithms are implemented in each partition, locally deadlock-free but globally not: per-partition routing algorithms are unrelated. However global traffic support is not straightforward any more

In the latter solution two virtual channels are needed to separate local and global traffic to avoid deadlock and each virtual channel has its own routing algorithm(s), that means from the LBDR viewpoint that routing bits need to be duplicated as well, thus leading to a rough 2x increase in complexity of the routing algorithm!

As already said, a dynamic runtime environment means that runtime modifications of the routing algorithm may be needed in order to enable/maximize/adapt/prolong

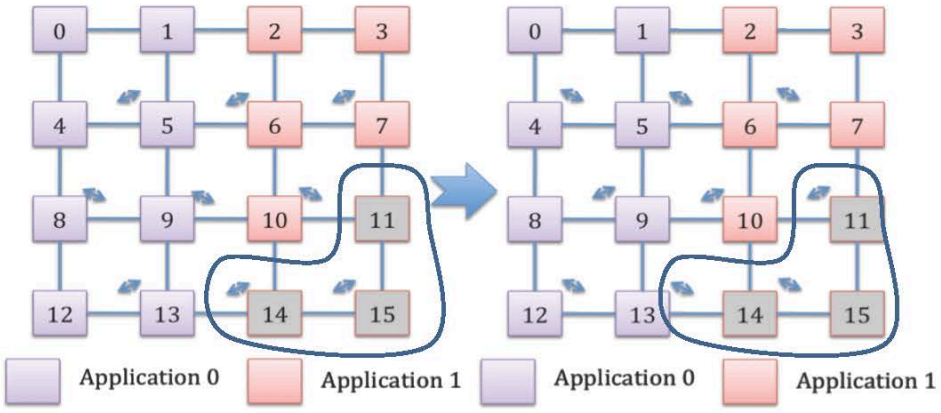


Figure 4.9: Routing algorithm adaptation: now the d-shape partition in grey can be allocated to Application 2!.

utilization of GPPA resources and so I need to change routing function also with background traffic running., thus enabling a dynamic QoS , lifetime testing and fine-grained platform control.

This flexible partitioning and resource management requires a control tower in software (hypervisor) but also a command execution support in hardware.

The proper course of action following the hypervisor commands is taken by the on-chip network. So the NoC becomes the system integration and control framework, thus he feasibility of flexible partitioning/resources management concept depends mainly on the runtime reconfiguration capability of the NoC routing function!

All the challenges about this latter will be tackled in the next chapters of this thesis.

4.3 Summary

In this chapter I introduced the sw and hw support to SDM approach. In particular, focusing on the hw, the virtualization of the resources, basis of the approach, requires to NoC the runtime reconfiguration capability, allowing to efficiently face several open issues of dynamic environments, as providing QoS, avoiding fragmentation among portions with different routing functions and safely enabling the reshap. The runtime reconfiguration capability of the NoC is the challenge that will be addressed in next chapters (Chapter 5 and 6) and stays at the core of this thesis.

Chapter 5

Runtime reconfiguration of the NoC routing function

In order to cope with an increased level of resource contention and dynamic application behaviour, the runtime reconfiguration of the routing function of an on-chip interconnection network is a desirable feature for multi-core hardware platforms in the embedded computing domain. The most intuitive approach consists of draining the network from ongoing packets before reconfiguring its routing tables, thus preventing the occurrence of deadlock from the ground up. The impact on application performance is however unacceptable. On the other hand, truly dynamic approaches are too much of an overhead for an on-chip setting. Recently, the overlapped static reconfiguration (OSR) method was proven to be capable of routing reconfiguration in the presence of background traffic with only a mild impact on the resource budget.

This chapter finds that this method is still far from materializing its potentials in terms of reconfiguration performance (both impact on background traffic, which is still there to some extent, and duration of the reconfiguration transient). Therefore, it proposes a set of optimization methods for OSR spanning the trade-off between performance improvements and implementation cost. To the limit, fully transparent reconfiguration is delivered.

Key novelty: design methods and associated circuit technologies for runtime reconfiguration of the NoC routing function, optimizing the baseline OSR_{Lite} mechanism.

5.1 Motivation and related works

Today, multi- and many-core architectures are gaining momentum as a potential source of hardware acceleration for many different algorithms[92], especially in the embedded computing domain[95]. At the same time, modern embedded systems integrate more

and more complex functionalities, requiring the concurrent execution of several applications onto the same hardware platform, possibly with heterogeneous and time-varying performance/reliability/power requirements. The recent trend for embedded system virtualization is finally strengthening the need for an optimized usage of parallel hardware resources in a contention-sensitive scenario.

Partitioning of array fabrics of homogeneous processor cores and isolation of derived partitions are gaining momentum as means of pursuing the integration of functionality from separate users/devices onto NoC-based many-core processors, while meeting their potentially heterogeneous requirements. Following this trend, the traditional time and space partitioning concept is being extended to parallel hardware platforms to overcome the challenge of using shared (yet modular) resources in applications that are executed concurrently. However, a static partitioning scheme cannot keep up with the increased levels of adaptivity of modern embedded systems, therefore flexible partitioning should be the target. In practice, partitions should be set up or tore down with few or no restrictions, and their size and shape potentially changed at runtime[66]. Whether such a usage paradigm will be feasible or not depends to a large extent on the capability of reconfiguring at runtime the routing function of the on-chip network (NoC), serving as the global communication fabric as well as the system integration framework.

Techniques for runtime reconfiguration of the routing function have been investigated in the off-chip networking domain, however their application to an on-chip setting is still in the early stage. These approaches are either non-reconfigurable fault-tolerant routing strategies, which tolerate a limited number of faults [34, 58, 60, 67], or reconfigurable routing mechanisms that allow unlimited changes to the network. I focus on schemes of the second category. Static reconfiguration methods simplify the problem at the cost of large performance penalties. They in fact consist of draining the network from ongoing packets, modifying routing tables to configure the new routing paths, and finally resuming traffic injection [16, 120]. This way, deadlock cannot occur during the reconfiguration transient, when packets of the old and of the new routing function could otherwise co-exist. On the contrary, dynamic reconfiguration techniques succeed in updating routing tables without stopping user traffic, but typically result into unacceptable implementation overheads for an on-chip setting [24, 90, 108, 41, 91, 8, 2, 89]. Although runtime performance is more likely to be preserved, such approaches end up materializing architectures with lower operating speeds (or higher latencies) by construction.

An intensive research effort is currently underway in an attempt to find a suitable design point for chip implementations, including Vicis[48], Immunet[112], Ariadne[4] and other reconfigurable routing frameworks[46, 143, 50]. With respect to these works, the recent adaptation of the **Overlapped Static Reconfiguration** (OSR) [89] methodology to the tight resource budgets of embedded systems provided an appealing trade-off between reconfiguration performance and implementation complexity[128]. OSR

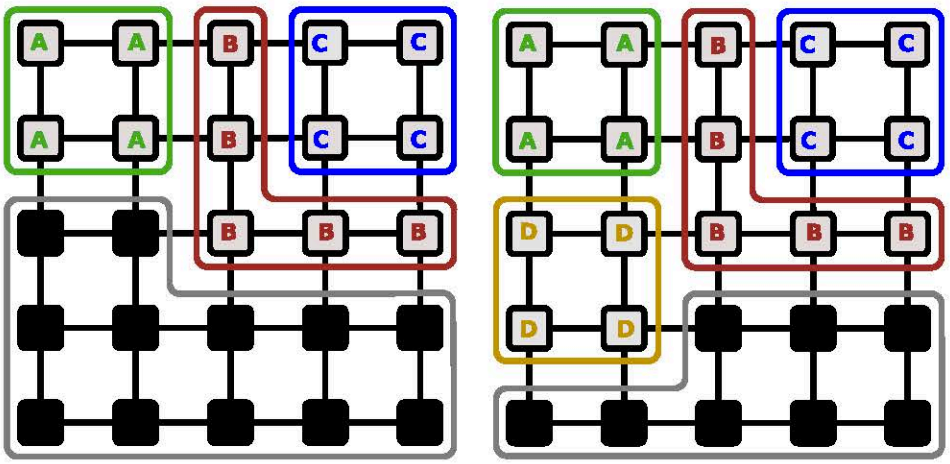
relies on the principle that if packets with the old routing function are prevented from following packets using the new one, deadlock cannot occur. Enforcing this ordering mechanism is possible even without draining the network from ongoing packets, by propagating a separation token between old and new packets throughout the network. Notwithstanding that, several performance inefficiencies still affect the OSR mechanism, which can be fundamentally identified as the temporary suspension of traffic injection during the reconfiguration transient, the packet blocking behind the self-propagating epoch separation boundary, as well as the network-wide nature of each reconfiguration event.

This work moves from the consideration that although the work in [128] has largely cut down on the implementation cost of OSR, thus bringing it within reach of embedded systems, the mechanism is still far from materializing its potentials in terms of reconfiguration performance. In practice, I aim at minimizing the performance penalties of ongoing communication flows while a reconfiguration event takes place. With this respect, **we propose techniques spanning a trade-off between the performance optimization they achieve and their implementation cost. To the limit, I prove the feasibility of a totally transparent reconfiguration process from the network performance viewpoint**. At the same time, I aim at speeding up the reconfiguration transient itself by making it local to the partition concerned with the modification of its routing function, while leaving the remaining part of the network unaffected. While in principle simple, this optimization requires careful engineering of switching hardware to avoid critical races and/or inconsistent states.

5.2 Inspiration

The new mobile usage models that are coming about require the execution of multiple use cases on the same device while optimizing resource consumption for each of them. On the other hand, the hardware and software design convergence in todays complex embedded systems call for an upgrade of architecture building blocks in the direction of runtime reconfigurability and adaptivity. As a result, applications should be able to frequently reconfigure the underlying hardware platform on-the-fly and in a cost-effective way, thus selecting the most convenient operating point that suits their needs and allows an efficient use of system resources.

Modern multi-core integrated systems achieve scalable computation horse-power and power efficiency by integrating a large number of processing cores on the same silicon die. This trend is unmistakable since current products already include tens and even hundreds of processing cores, such as the Tilera multicore processor [32]. In this context, on-chip interconnection networks (networks-on-chip, NoCs) are typically used to provide communication parallelism and the reference integration infrastructure for the whole system.



(a) Configuration A. (b) Configuration B.

Figure 5.1: Two NoC configurations where the routing algorithm needs to be adapted.

To address the new functionalities, the textbfNoC must be enriched with an efficient reconfiguration process which enables the smooth and transparent transition between system configurations. For instance, Figure 5.1 shows two different configurations of a multicore system over time. In the first one (configuration A) different applications are mapped to the NoC nodes and execute concurrently, while other resources are powered down. Later, the resource manager may trigger a chip reconfiguration to power on unused resources and thus activate a new application (configuration B).

The transition between configurations needs a careful design of the NoC routing algorithm, which establishes the paths for every packet in the network. At each configuration a different routing algorithm is needed. In both cases, the algorithm must be deadlock-free (should not introduce cycles in its channel dependency graph). However, in the transition between configurations, both algorithms can induce extra dependencies that lead to deadlock.

Therefore, in order to migrate from one configuration to the other, one possible approach is to drain the network, then changing the routing algorithm to the new one and finally resuming traffic injection with the new algorithm. This is the case of the so called traditional *static reconfiguration* (TSR). In this case system performance is likely to be heavily impacted by the reconfiguration process due to the temporarily low resource utilization. Alternatively, the network can be dynamically reconfigured, in the sense that traffic is not stopped during the reconfiguration process, but an effort is needed to avoid deadlock situations. This is typically achieved by devoting extra resources to the network. I refer to this case as the dynamic reconfiguration.

In this chapter I advance state-of-the-art in reconfiguration frameworks for NoC-based systems. However, instead of designing a brand new reconfigura-tion mechanism, I recognize the large amount of bibliography and proposals made for reconfiguration mechanisms in high-performance off-chip networks. In this sense, I pick the approach that better suits the NoC domain and the tight resource budgets of the on-chip envi-

ronment.

The *Overlapping Static Reconfiguration* process (OSR) in [89] enables a transparent system reconfiguration process. Furthermore, to cope with the on-chip resource budget I present also OSR_{Lite} [128] by which it is possible to reconfigure a whole 64-node network in a few hundreds of cycles, enabling the entire and transparent transition between any pair of independent and unrelated configurations. Moreover, this is achieved with no impact on network latency and with no impact on switch delay. The reconfiguration performance of OSR_{Lite} makes it the enabling tool for planned reconfigurations in multicore systems.

In this thesis I find that OSR_{Lite} is still far from materializing its potentials in terms of reconfiguration performance, so I propose a set of optimizations for the mechanism. The following specific scenarios are part of the ones that can be therefore materialized by the outcome of this work:

- Virtualization of the system. Our method enables the runtime division of the entire network into sets of virtual regions for assignment to different applications running concurrently.

- Power management. The reconfiguration mechanism can be exploited for powering down unused resources; such functionality becomes com- pulsory to keep power consumption levels to reasonable bounds.

- Reliability. When a NoC is augmented with transient fault tolerance, then this kind of faults can be tolerated without any loss of information. However, intermittent faults are likely to be an indicator of the gradual onset of a permanent fault (typically, a wear-out fault). In this case, OSR_{Lite} can be used to reconfigure the network so to exclude the affected link/switch component, before the permanent fault shows up and causes packet loss.

5.3 OSR: baseline mechanism

5.3.1 Native OSR technique

Typically, a routing algorithm is deadlock-free when its *channel dependency graph* (CDG) is acyclic (we do not consider fully adaptive routing algorithms). The CDG is set by representing the resources of the network by vertices (mainly the buffers associated with the ports of each switch) and the dependencies between two resources by arcs. There is a dependency between two resource r_1 and r_2 if a message can use r_1 and request r_2.

Two routing algorithms R_1 and R_2 are deadlock-free when they have an acyclic channel dependency graph. However, when using both algorithms at the same time new extra dependencies are induced potentially leading to deadlock. This can be seen

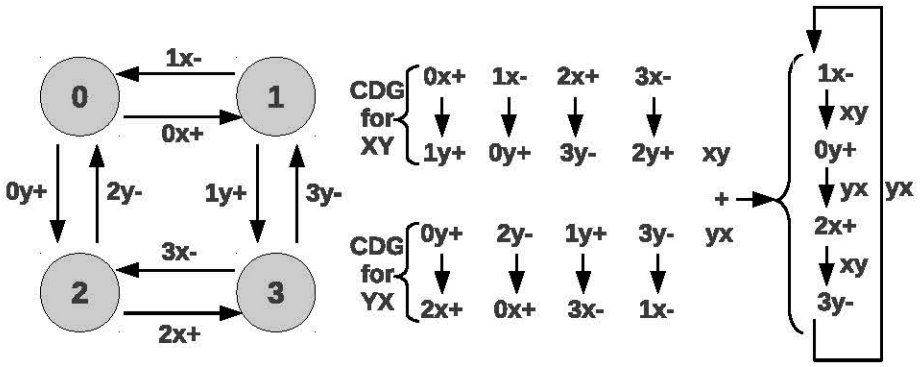


Figure 5.2: Channel dependency graph for two routing algorithms and the combination of both.

in Figure 5.2 where a cycle is formed when using two routing algorithms (XY and YX) at the same time. During a reconfiguration process I refer to R_{old} as the old routing function and R_{new} as the new routing function. Similarly, packets routed with R_{old} will be referred to as old packets and packets routed with R_{new} will be referred to as new packets.

The native OSR method is based on the fact that those cycles are created only when old packets using R_{old} are routed after new messages using R_{new}. If old packets are guaranteed to never go behind new packets the extra dependencies do not occur in practice and then no deadlock can be formed. Indeed, in a static reconfiguration process the entire network is drained thus guaranteeing old packets will never go behind new ones.

OSR is a static reconfiguration process but localized at link/router level, and not

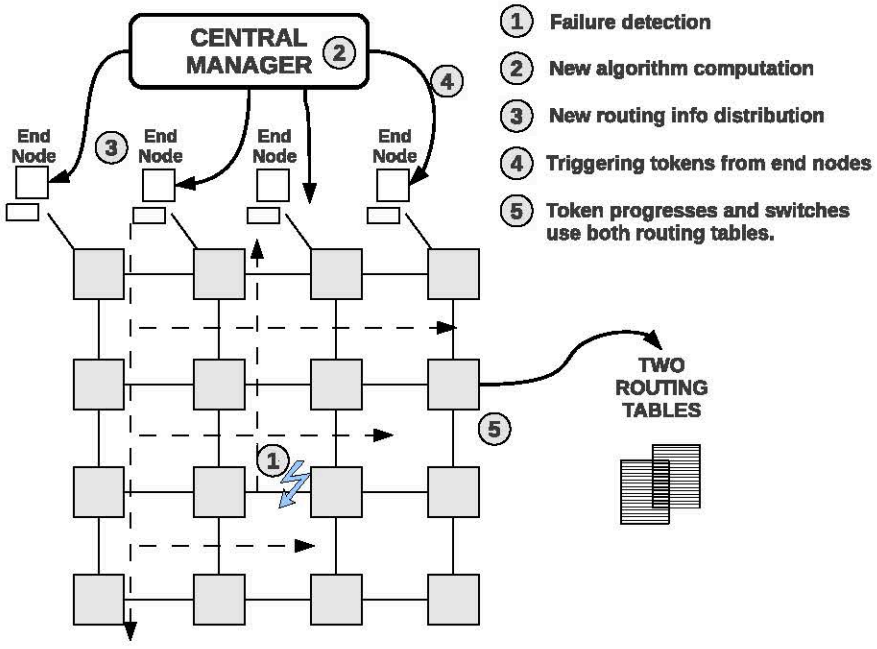


Figure 5.3: Reconfiguration steps performed in an OSR environment.

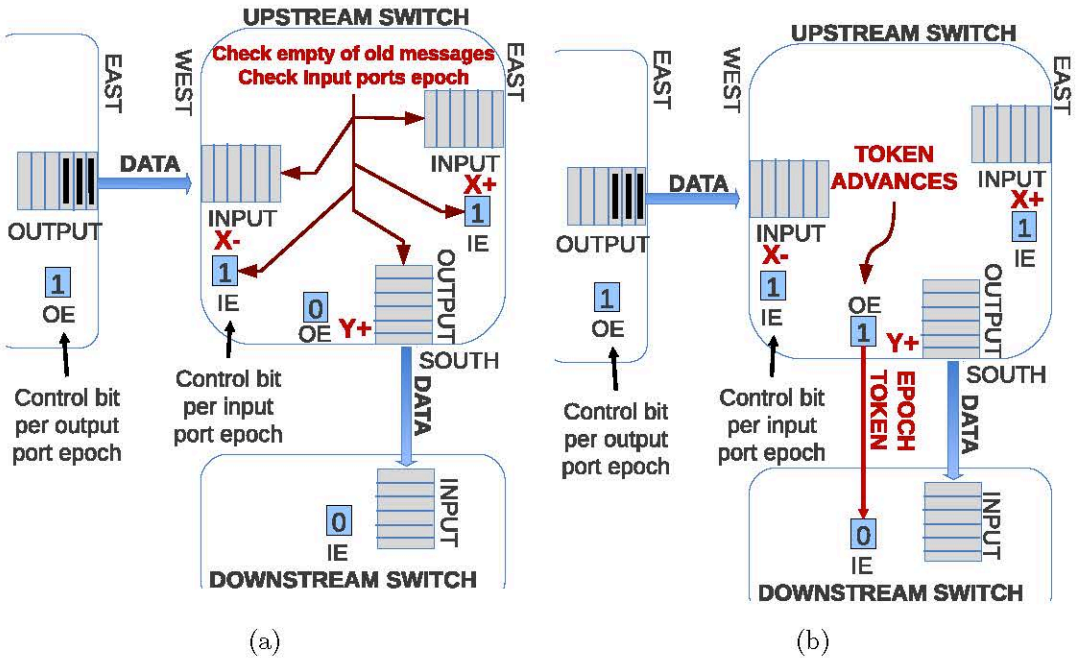


Figure 5.4: Token advance in a network: (a) check for absence of old messages and input ports epoch, (b) token signal propagation. The token separates old traffic from new traffic.

at network level. Indeed, it guarantees that new packets are only forwarded via links that have been drained from old packets. This is achieved by triggering a token that separates old packets from new packets. The token is triggered by all the end nodes and tokens advance through the network hop by hop. Indeed, tokens follow the CDG of the old routing function, draining the network from old packets. However, in contrast with static reconfiguration, the new packets can enter the network at routers where the token already passed. Figure 5.3 shows the complete native OSR mechanism, involving a central manager. In a first step, a reconfiguration action is triggered, either by the detection of a malfunctioning component or by a higher level manager in the system stack requiring a reconfiguration, e.g. a new application is admitted. In any case, when needed the central manager may receive event notifications through the network (step 1). Then, in step 2, the new algorithm for the new configuration is computed by the central manager. The resulting information is disseminated to all the switches in step 3. In step 4 the end nodes trigger the token and the OSR reconfiguration spreads throughout the network (step 5).

Figure 5.4 shows how tokens advance in a network. At a given output port, a token is triggered to the next downstream router indicating the output port has been drained from old packets. This is guaranteed when the token has been received through all the input ports of the switch that have old (Rold) output dependencies with the output port. These port dependencies can be extracted from the R_{old} routing algorithm. Notice that the token divides two epochs in the network, the old epoch (when packets are routed with the R_{old} routing function) and the new epoch (when packets are routed with the R_{new} routing function).

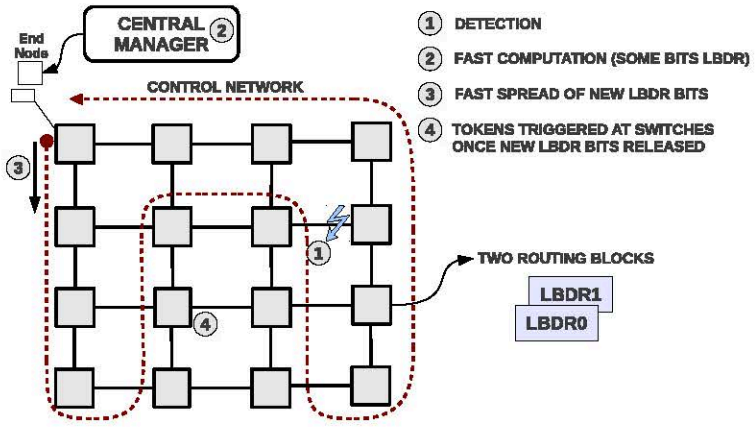


Figure 5.5: Reconfiguration steps performed in an OSR_{Lite} environment.

5.3.2 OSR_{Lite}

The OSR mechanism needs to be modified in order to better suit the NoC environment so to become an efficient and plausible mechanism for planned reconfigurations. Indeed, with OSR_{Lite} the main issues solved are the following:

- Codification of the routing information. During the reconfiguration pro- cess both routing algorithms coexist at the same time at routers. This means resources need to be sized for both algorithms. In OSR, routing tables were used to store the routing info. In NoCs, however, routing tables are an expensive resource in terms of access time, area, and power consumption. Therefore, hosting two routing tables per switch input port does not appear to be a cost-effective solution for OSR_{Lite}.

- Control virtual channel (VC) used in OSR. Different actions (sending routing information to routers, triggering the reconfiguration process) are performed during the OSR reconfiguration which imply the ex- change of information between a central manager and the routers or theendnodes. In [48] this was implemented by means of a control VC. Unfortunately, using VCs only for that purpose has a large impact on router implementation (will be seen later) and is not fully justified in an on-chip.

- Involvement of end nodes in the reconfiguration process. In OSR the end nodes were notified to trigger the reconfiguration. This is done by end nodes injecting the token directly as a new packet. In NoCs, reach- ing the end nodes via dedicated packets from the central manager would be a time-consuming course of action. In order to cut down on the re- configuration latency, involving only switches and not end-nodes in the reconfiguration would be an appealing property in a NoC setting.

OSR_{Lite} approach addresses all these issues. Figure 5.5 shows all the steps and the main modifications performed. In particular, it exploits a control network through

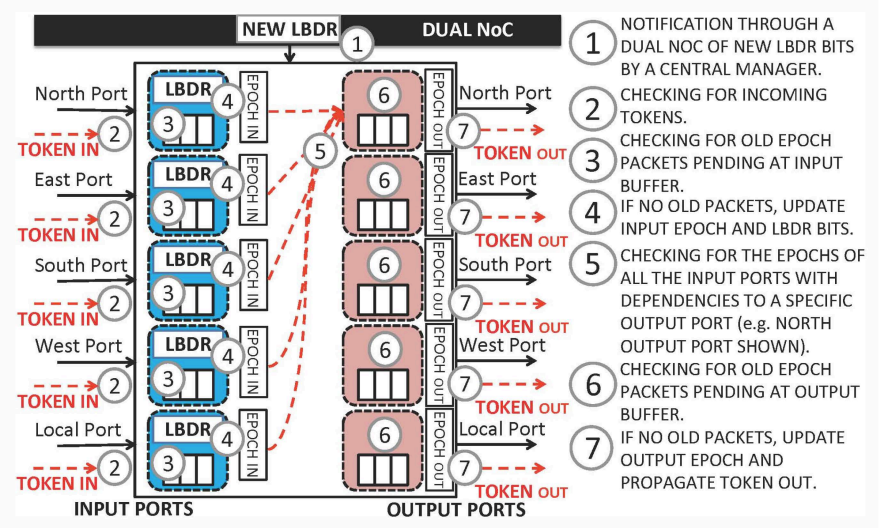


Figure 5.6: Reconfiguration steps performed in an OSR_{Lite} at switch-level.

which routers can inform about expected topology changes (e.g., an output link is having frequent transient failures and is going to fail soon, or a region of the NoC is overheated and needs to be powered down). The control network collects all the notification events and sends them to a central manager (step 1). If the reconfiguration is instead initiated by a resource manager in the context of power management or virtualization strategies, step 1 can be skipped. The central manager then computes the new configuration (step 2) and disseminates the new routing information to the switches (step 3). Then, every switch starts the OSR_{Lite} reconfiguration process in step 4. Notice that end nodes are not involved in the reconfiguration process.

The control network can be used also in step 3 for routing bit dissemination to the switches, through a dual network [56] for switch-to-global manager bidirectional signaling, thus offloading critical control tasks from the main data network. In that work, the dual network was used to notify diagnosis information to the manager following the main NoC testing phase, and to notify configuration bits of the routing mechanism to the switches. The same network could be reused for other purposes, such as congestion management, deadlock recovery and software debugging. In it is showed to be a cost-effective solution for control signaling, which can be easily and effectively made reliable through a combination of fault-tolerant and online testing strategies. For this reason, this work relies on such a fault-tolerant dual network to convey control information of the reconfiguration process.

Furthermore, [56] also reports an efficient computation algorithm that comes up with the routing configuration bits of a new network partitioning or topology shape. This is the algorithm the controller runs in step 2. Given that the control network and the computation algorithm are covered by previous work, now on I focus on the core reconfiguration process of the network and on the micro-architectural support for that. The reader should keep in mind that all these mechanisms will work together in the complete reconfiguration framework. In the next section I describe the router

implementation in more detail.

OSR_{Lite} implementation

Without lack of generality, the xpipesLite switch architecture [126] proves viability of OSR_{Lite} mechanism. The switch implements both input and output buffering, relies on wormhole switching and on a stall/go flow control protocol.

The switch architecture is extremely modular and implements logic-based distributed routing (LBDR): instead of relying on routing tables, each switch has simple combinational logic that computes target output ports from packet destinations. The support for different routing algorithms and topology shapes is achieved by means of 16 configuration bits for the routing mechanism of the switch (hereafter denoted as LBDR bits). LBDR bits carry the routing algorithm information (expressed in terms of routing restrictions), the connectivity information of switch output ports and special detour bits. *Such bits make LBDR a flexible routing mechanism while at the same time significantly cutting down on the memory requirements of routing tables.*

LBDR bits are computed by a central NoC manager and disseminated to the switch input ports through the dual control network. Indeed, two sets of LBDR bits are allocated at each router for OSR_{Lite}. Upon receiving the new routing bits, a router triggers the reconfiguration process by auto-generating initial tokens at its local input port (port connected to an end node) and processing the tokens accordingly.

The logic enabling the OSR_{Lite} mechanism is integrated into the above mentioned baseline switch taking care to preserve its modularity together with its performance. Thus, the OSR_{Lite} logic is designed in new modules plugged into the switch without affecting the existing blocks. Moreover, the new modules are instantiated for each switch port following the modularity of the baseline blocks (the OSR_{Lite} mechanism can be extended for switches of every arity by means of simple logic replication).

OSR_{Lite} at Input Ports

As a first step, the baseline switch was enhanced with a second routing logic unit ($LBDR_1$) collecting the new routing info coming from the central manager. This unit is connected to the input buffer as the baseline $LBDR_0$ block (see Figure 5.7) although is used exclusively for routing packets in the new epoch (new packets). The switch arbiters need to select the routing info from the appropriate routing logic block (either $LBDR_0$ or $LBDR_1$). This is obtained from a multiplexer configured by the current epoch of the input port (in a flip-flop).

In order to reduce the reconfiguration latency, the input port evolves to the new epoch as soon as there are no stored header flits at the input port with the epoch bit set to zero (Epoch 0 headers signal) and the token has been received from the upstream switch (upstream epoch signal). Notice that in the case of the ports connected to end node (local port; local port flag), the token is assumed to arrive with the arrival of

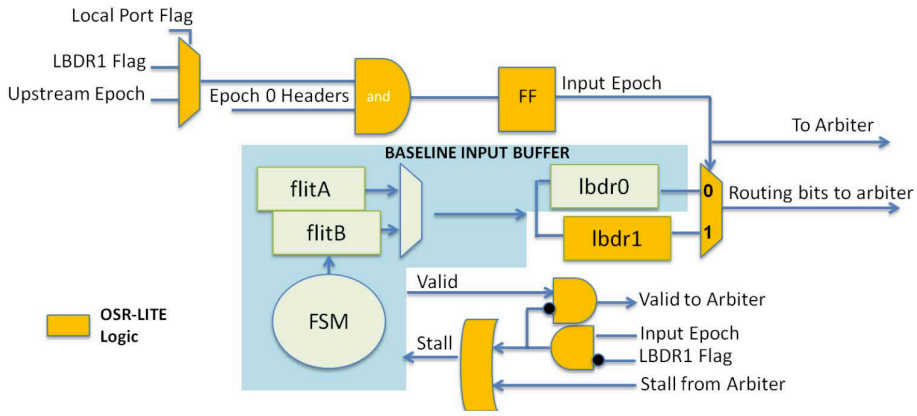


Figure 5.7: Switch input buffer enhanced with the OSR_{Lite} logic and a new set of routing mechanism.

the new configuration bits ($LBDR_1$ flag). In this case, the header flits located in the buffers are considered of the new epoch when the new configuration bits have arrived and the routing mechanism ($LBDR_1$) is set. Notice that local ports do not introduce dependencies between channels that may lead to deadlocks, therefore is safe to assume all the injected flits as belonging to the new routing function. To notice that the token propagation will always start from local ports at switches, not involving end nodes.

The number of flit headers to be routed by $LBDR_0$ and stored in the buffer is detected by a 2 bits counter monitoring the incoming and outgoing headers of the input buffer module. The counter increases its value when a header is accepted and the incoming token is low and decreases its value when a header is sent. In order to preserve the max performance of the baseline switch, sequential logic stages were exploited to avoid impacting the critical path in the OSR_{Lite} mechanism. Notice that the implementation prevents possible race conditions from occur- ring. For instance, a token may be received from the upstream switch be- fore the new routing bits are received. In that case, the header flits in the input buffers are stalled and declared not valid to the internal switch logic until $LBDR_1$ is set.

OSR_{Lite} at the Arbiters

OSR_{Lite} requires a lightweight new module plugged around the baseline arbiters. The logic is reported in Figure 5.8. Basically, a set of AND/OR logic blocks together with a set of EXOR blocks allow the arbiter to process an incoming header exclusively when the epoch of the switch input port is the same as the one of the destination output port.

On the contrary, a packet residing in an input port with the new epoch is stalled until the output port evolves to the new epoch (guaranteeing old packets go first and then new packets).

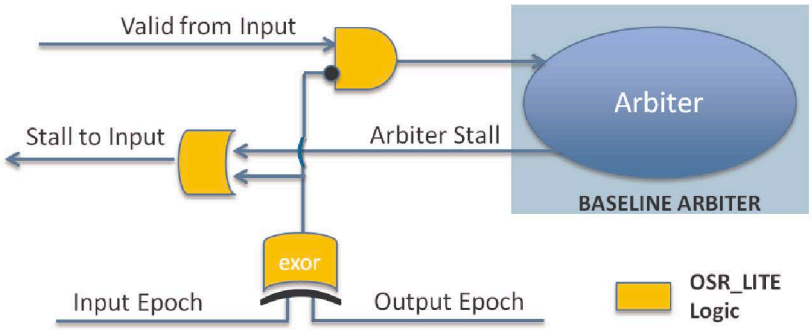


Figure 5.8: Switch arbiter enhanced with the OSR_{Lite} logic.

OSR_{Lite} at the Output Ports

Concerning the output port, an output port evolves to the new epoch when all the input ports with output dependencies to this output port have evolved to the new epoch. In order to efficiently deal with the dependencies, OSR_{Lite} takes profit of the routing bits used in LBDR. Routing bits indicate the routing restrictions that exist at neighboring switches. Therefore, they can be seen also as channel dependencies.

If the R_{xy} bit is set it means that there is a link dependency between the output port x and the output port y at the next switch. On the contrary, if the bit is reset it means there is no dependency and in that case I can safely assume no packets will come through the port x requesting output port y. Therefore, the output port needs to receive both the epochs of the input ports and the routing restrictions located at the neighboring switches. The mechanism is enabled by a set of OR blocks (each of them belonging to a different input port) followed by an AND block, as represented in Figure 5.9.

In contrast with the baseline OSR technique (where the routing restriction information was saved in the routing table), the OSR_{Lite} mechanism needs to obtain

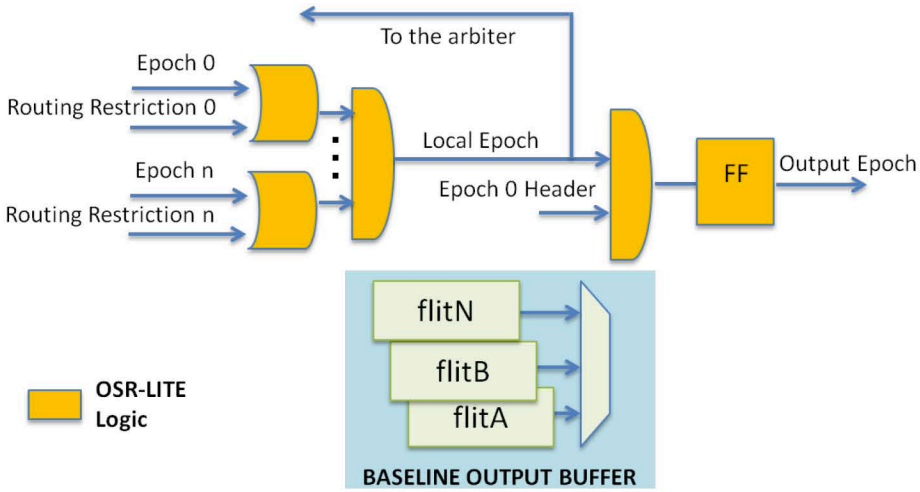


Figure 5.9: Switch output buffer enhanced with the OSR_{Lite} logic.

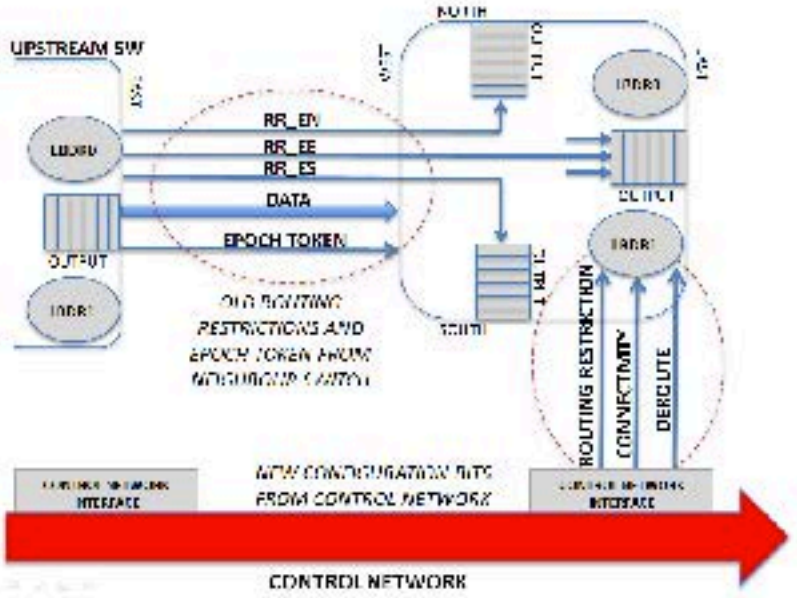


Figure 5.10: Configuration information from neighbor switches and control network.

channel dependencies from the routing logic located at neighbor switches. As a result, three additional routing bits are sent by the $LBDR_0$ logic of the upstream switch together with the token bit. To note that $LBDR_0$ received its routing bits information through the control network in an earlier configuration stage. In addition, the input port needs to send the incoming routing restriction signals to the appropriate output ports. Thus every link is extended by 4 additional wires (i.e. 1 token wire + 3 routing restriction wires). See Figure 5.10.

Finally, the token is sent by the output port to the downstream switch when all the input ports with dependencies with the output port have evolved to the new epoch, meaning all these input ports have drained all the old packets from their buffers (see the LocalEpoch signal in Figure 5.9). Once the network has completely migrated to Epoch 1, the central manager can safely fill $LBDR_1$ bits with a copy of $LBDR_1$ bits, and instruct all the switches to safely swap to Epoch0 again. This allows for the system to be ready in few cycles for a new reconfiguration process.

System-Level evaluation: propagation

In this section, I show how the OSR_{Lite} propagates over the network, simulating the reconfiguration process in an event-driven cycle-accurate network simulator. A 8×8 mesh is used with wormhole switching (although the proposed method also works for virtual cut-through switching). Figure 5.11 shows how OSR_{Lite} tokens propagate over a mesh when there is no traffic traveling through the network. The diagonal arrows represent the bidirectional restrictions imposed by the routing algorithm (Segment-Based routing in this case). In this figure, the numbers inside the switches represent the cycle when the token signal is propagated to its neighbors. Moreover, the arrows among switches depict the direction of the token signal propagations. As I can see, the

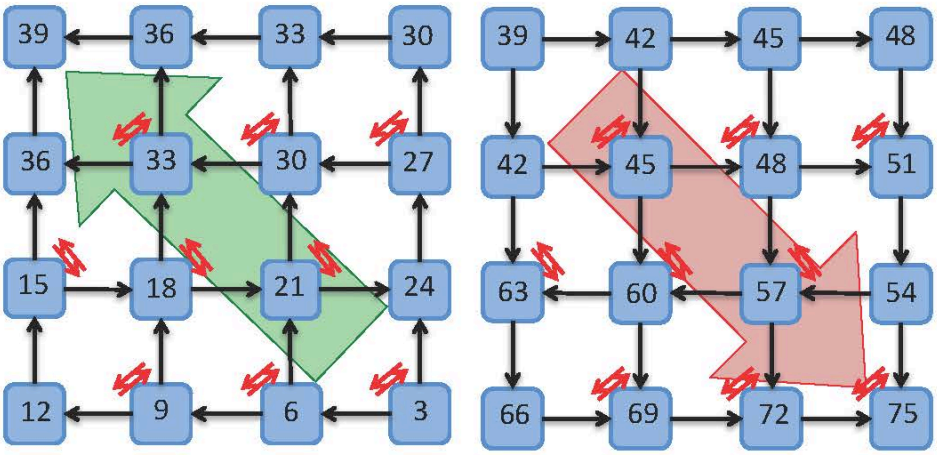


Figure 5.11: OSR-Lite propagation over a 4×4 mesh topology: (a) scrolling up, and (b) scrolling down.

token signals propagate among switches throughout the network in the order of the routing channel dependency graph, where Figure 5.11(a) follows a scrolling up zig-zag direction, and Figure 5.11(b) follows a scrolling down zig-zag direction.

When no messages are traveling through the network and a regular 2D mesh is considered then the number of clock cycles required for the OSR_{Lite} reconfiguration process is modeled by the following formula:

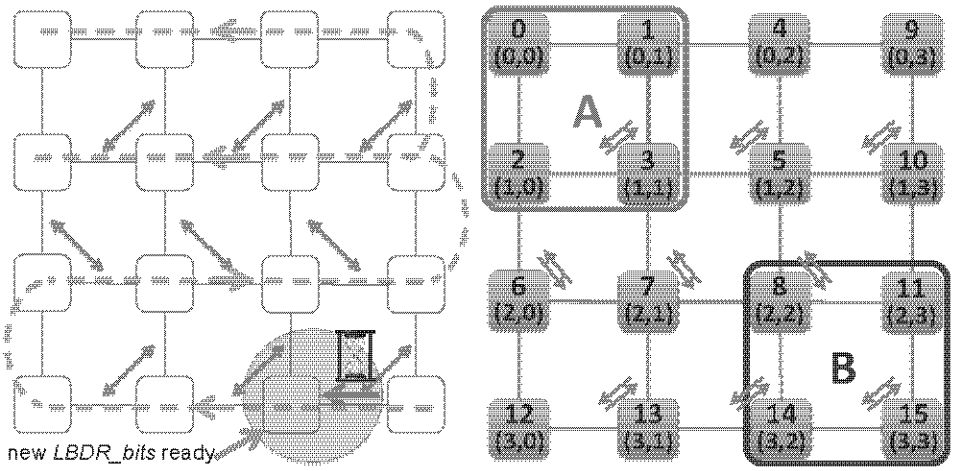
$$PropagationTime = (4 \times D \times (D - 1)) - 1$$

where D represents the mesh dimension. As I can see, it is a very fast process as the protocol uses only 223 cycles when a 8x8 mesh is considered.

The high speed of the OSR_{Lite} reconfiguration process allows to perform frequent planned reconfigurations without affecting the integrity of the system operations. However, when there are messages traveling through the network the switches must drain the input queues of old messages before propagating the token signal. This fact delays the OSR_{Lite} propagation depending on the network load. In the following, I analyze all the sources of inefficiency, as starting point for the optimizations proposed in this thesis.

5.4 Sources of inefficiency

The scroll-up phase showed in Figure 5.11(a) is then completed by a similar scroll-down phase since inter-switch links are bidirectional. The total reconfiguration time is given by the sum of the two phases (which amounts to 75 cycles) with some sensitivity to ongoing traffic conditions. Intuitively, token propagation is triggered by buffers emptied by old packets, hence traffic congestion can slow down token propagation.

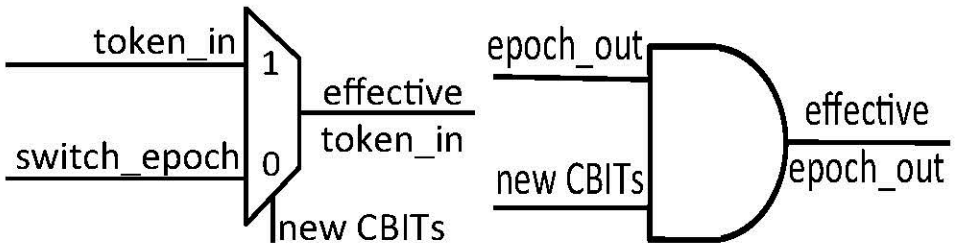


(a) (b)

Figure 5.12: OSR-Lite propagation over a 4×4 mesh topology: (a) scrolling up, and (b) scrolling down.

Finally, it is important to point out that new packets from input ports already in the new epoch cannot be forwarded to output ports that are still in the old epoch, to enforce the strict ordering principle of OSR. They have to wait for such outputs to evolve to the new epoch as well. See [128] for details. Despite a lower impact on background traffic performance than static reconfiguration methods, I identified two main sources of inefficiency in OSR_{Lite}. One one hand, *tokens have to be propagated throughout the entire network*. This requirement does not match with the most recent usage models for large integrated networks, which consist of network partitioning and isolation. In this case, OSR_{Lite} ends up making a local partition reconfiguration a global event, which not only causes performance inefficiencies, but also violates the isolation principle. As an example, when the routing function of partition B in Figure 5.12(b) needs to be reconfigured at runtime, token propagation is not limited to B, but spans the entire network, including the switches in A, which should however not modify their routing function. As a result, performance of traffic inside A is temporarily perturbed by the reconfiguration process without proper justification.

On the other hand, as shown in Figure 5.12(a), to understand what such perturbation actually consists of, I should recall that *the possible misalignment between token and routing bit propagation across the network causes blocking of traffic injection*, causing a temporary suspension of traffic injection during the reconfiguration transient and the packet blocking behind the self-propagating epoch separation boundary, due to the network-wide nature of each reconfiguration event. In fact, whenever a switch receives new routing bits, its local port instantaneously evolves to the new epoch. However, traffic injection is only resumed when the target output port of the head-of-the-line packet evolves to the new epoch as well. This requires that the token has been received at specific switch input ports. If such tokens are delayed, new traffic cannot meanwhile be injected into the switch by its local port. Such misalignment is highly



(a) (b)

Figure 5.13: Local Reconfiguration: processing at input ports (a) and at output ports (b).

likely, since tokens and routing bits are propagated through different transport layers and protocols. Tokens are in fact carried through dedicated wires that increase the width of inter-switch links. In contrast, routing bits are carried through the dual control network.

5.5 Optimization of reconfiguration mechanism

In this section I illustrate one of the contribution of this thesis, i.e. a set of performance optimizations that OSR_{Lite} could benefit from, spanning from simple to more aggressive ones trading performance speedups (approaching latency insensitive reconfiguration) with a higher implementation cost.

5.5.1 Local Reconfiguration

The main goal of this optimization is to restrict the reconfiguration procedure to only the affected partition, for instance partition B in Figure 5.12(b). In a global reconfiguration, the state of the switches in the network is aligned to the same epoch, but if I aim for local reconfiguration, such alignment is not guaranteed any more, and this could cause malfunctions in the reconfiguration process itself: resources may appear to be as already reconfigured or may block the token propagation.

To avoid this, I first of all changed the way tokens are coded during their inter-switch propagation. In the original implementation, tokens were coded by a change of the binary value of token signals. With our optimization, tokens are associated to 1 cycle pulses (*epochout-in*), which cause the digital value of associated state signals ($token_{in\text{-}out}$) to change. Also the arrival of a new set of *LBDR bits* for switch reconfiguration is denoted by a pulse on the $LBDR_{en}$ signal, which causes bit flipping on the associated *switch epoch* state signal.

With these assumptions, I can safely feed new *LBDR bits* only to the switches in the partition to be reconfigured, and devise a control logic to limit token propagation to only the partition area. For this purpose, the $epoch_{out}$ pulse in Figure 5.13(b) is filtered by an AND gate with *CBITS* signals that are part of the set of new LBDR

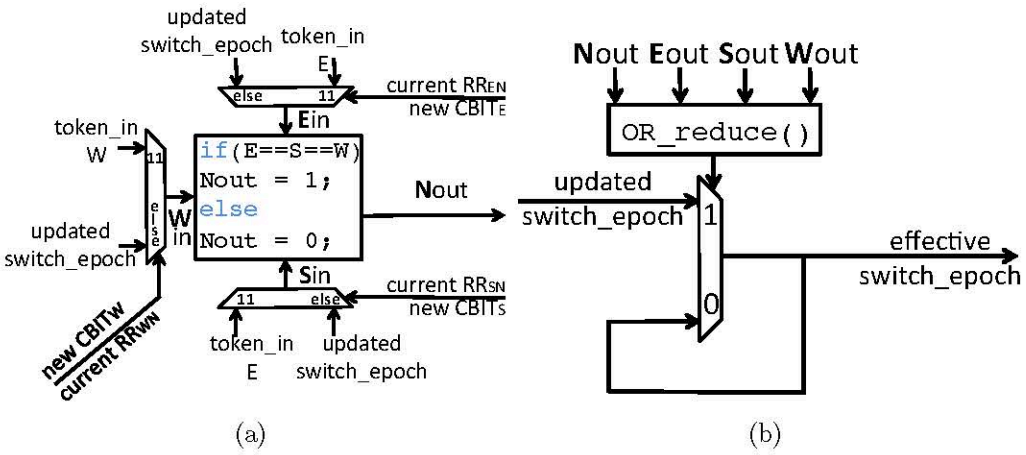


Figure 5.14: Logic behind synchronization between new *LBDRbits* and token propagation.

routing bits. Originally, *CBITS* indicated that some switch ports are not instantiated since belonging to the network boundary. I extend their meaning to treat similarly those switches that are on a partition boundary. The OSR mechanism inherently guarantees that *CBITS* switch in advance with respect to the $epoch_{out}$ pulse (i.e., no epoch transition needed if no new *LBDR bits* received). In Figure 5.13(a) I consider the logic that controls epoch transition at the input of the switches on a partition boundary. The key idea is to mask the *tokein* state signal whenever it is generated by an *epochin* pulse coming from a switch located in a different partition. As usual, masking is performed based on the new connectivity bits *CBITS*. Overall, if a token arrives and the input port is linked to another switch of the same partition (*CBITS* = 1), the token takes its effect. If the switch belongs to another partition, the token is in practice filtered off. In this latter case, the token is automatically triggered upon receipt of the new routing bits (which is denoted by a change of *switchepoch*). It should be observed that *CBITS* are not affected by a modification of the routing algorithm of the partition, which only changes routing restriction bits. Our optimizations enable more flexible scenarios, i.e. restriction and expansion of an existing partition, and also merging of and splitting into two partitions. In each case the logic is safe because *CBITS* always anticipate $epoch_{out}$ pulses or at least are set synchronously with $token_{in}$ and *switch epoch*. The mechanism may generate critical races only in case multiple outstanding reconfigurations are ongoing in the network. For the time being, I restrict our analysis to a single reconfiguration at a time, and leave the more concurrent scenario for future work.

5.5.2 Synchronized reconfiguration

To prevent the blocking of traffic injection at switch local ports because of the unsynchronized arrival of new LBDR bits and of tokens at switch input ports, I implemented the following optimization: the new routing bits are not notified to the switches right

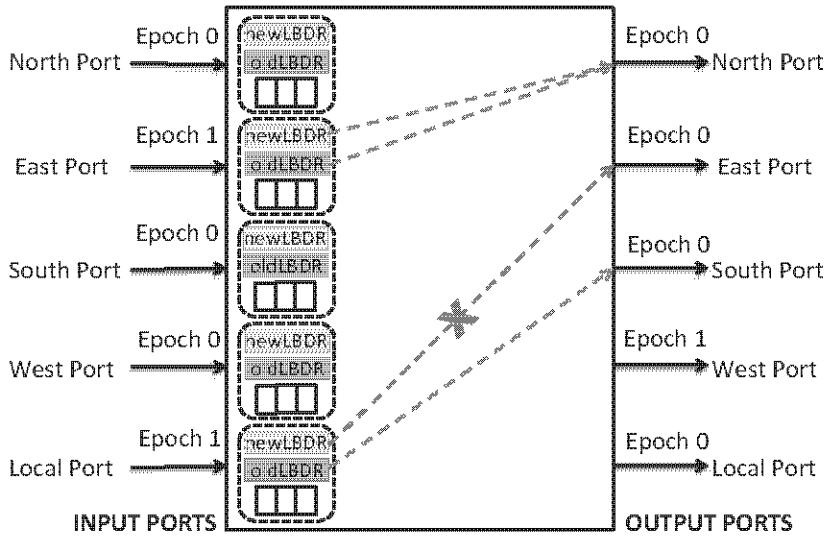


Figure 5.15: Optimized switch with two routing alternatives (old and new) at each input port.

away, but their notification is postponed till at least one output port meets all the requirements for a transition to the new epoch, regardless of traffic within the switch. This typically requires to wait for the arrival of tokens at those input ports that have routing dependencies with the target output port. The corresponding logic is illustrated in Figure 5.14. Let us focus on the north output port (Figure 5.14(a)). If switch input ports either have a routing restriction (i.e., no traffic from that port can cross the target output port) or belong to a different partition, they should be filtered off by the logic. In particular, as soon as new routing bits are notified to the switch, a token is automatically self-injected through these ports (see the *switchepoch* signal). In contrast, the remaining ports should be monitored for token arrival. When this occurs, the north output port is ready to migrate to the new epoch (Figure 5.14(b)). Only at this time, the new routing bits are actually notified to the switch and the epoch evolution can take place. During the monitoring time, the local port is still in the old epoch, and can safely keep injecting traffic based on the old routing function.

5.5.3 Epoch-conversion: towards a fully transparent reconfiguration

So far, the proposed optimizations can still live with a single set of LBDR registers per input port, as in standard non-reconfigurable LBDR switches. Only the control logic has been made more complex.

However, whenever a duplication of such registers is affordable, more aggressive optimizations can be devised. Indeed, I may rely on the fact that during reconfiguration the old routing paths are still available and can be used by the packets. Thanks to this, new LBDR bits can be notified to the switches right away, and new packets from switch local ports requesting output ports that have not evolved to the new epoch yet

RULES	OUT	CASE
R1: E_{in}=0 & $E_{out}[O_{old}]$=0	O_{old}	Old msg $\Longrightarrow$ old path
R2: E_{in}=0 & $E_{out}[O_{old}]$=2	-	Impossible to happen
R3: E_{in}=1 & $E_{out}[O_{new}]$=1	O_{new}	New msg $\Longrightarrow$ I new path
R4: E_{in}=1 & $E_{out}[O_{new}]$=0	-	Blocked msg
R4a: E_{in}=1 & $E_{out}[O_{new}]$=0 & $E_{out}[O_{old}]$=0	O_{old}	New msg converted to old
R4a: E_{in}=1 & $E_{out}[O_{new}]$=0 & $E_{out}[O_{old}]$=1	-	Blocked msg

Table 5.1: Routing rules for the baseline OSR method and when epoch conversion is enabled.

might be simply converted into old epoch packets. They would be using the old routing function instead of the new one, hence not breaking the OSR assumption, while at the same time crossing the token propagation barrier. This optimization is not restricted to switch local ports, but can be in principle applied to all input ports, and provides a latency-insensitive reconfiguration, as experimental results will prove.

With this optimization strategy, in each switch two sets of *LBDRbits* per input port are needed, one with the old routing bits (for the old routing function) and one with the new routing bits (for the new routing function). This is mitigated by the fact that I am actually duplicating few configuration registers and not entire routing tables like in table-based routing. Figure 5.15 shows the potential output ports that can be requested by the two versions of LBDR blocks in each input port. The example only focuses on two input ports promoted to epoch 1. A brand new packet stored into the local port, belonging to the new epoch, would request the south output port with the new routing function. This port is however still in the old epoch, hence the packet should be blocked. This packet can nonetheless take the output port computed by the old routing function safely and will be handled in the network as an old packet. Indeed, no tokens have been forwarded through the output port east yet, since the port is still draining old packets. The same rule can be used across all input ports, such as for packets from the east input port in Figure 5.15. In this case, both routing functions return the same target output port, which is still in the old epoch.

The applied optimizations are deadlock free by construction and can be summarized changing the basic rules of packet routing provided by baseline OSR_{Lite} mechanism (rules *R1* to *R4*), in particular replacing *R4* with two new rules *R4a* and *R4b*, as shown in Table 5.1. Let us assume E_{in} is the epoch of an input port and E_{out} is the epoch of an output port. Also, when a packet is routed at a given input port, O*old* is the output port computed when using the old routing function (*LBDRold*) and *Onew* is the output port computed when using the new routing function (*LBDRnew*). Applying these routing rules at each switch port, referred to as $OSR_{Lite\text{-}opt1}$, could introduce out-of-order delivery. Indeed, a new and an old packet injected by the same switch normally follow different paths. The old one can be blocked due to network congestion that might not involve the path of the new packet. Thus the latter, converted into an old packet

on the fly, can potentially reach the end node before the first packet, since now both packets are treated as old ones. Notice that out-of-order can be avoided if packets are not allowed to convert to old packets during their routing path over the network, i.e., applying the optimization only at local input ports. This will be the second version of the optimization and will be referred to as $OSR_{Lite\text{-}opt2}$. Both optimizations can be inferred in tandem with local reconfigurations, while they are mutually exclusive with respect to the synchronized reconfigurations, which however have a milder impact on the resource budget.

5.6 Experimental results

5.6.1 Assessing local reconfiguration

We consider the scenario shown in Figure 5.12(b) where there is a running partition A, in which a generic master *M0* is injecting a stream of packets to the generic slave *S3*, and I want to reconfigure partition B, launching a new OSR_{Lite} reconfiguration process. As result of an RTL-equivalent SystemC simulation of the network shown in Figure 5.16, our optimized approach supporting local reconfiguration gives many benefits. Indeed, a global reconfiguration significantly affects partition A, impacting on the arrival time of its running packets (about 46nsec for the monitored *M0* to *S3* stream, considering 1nsec = 1 clock cycle), because switches involved in the reconfiguration process have to wait for the token. Instead local reconfiguration does not impact at all partition A, since token propagation is limited to switches of partition B. A further benefit of our optimization emerges considering a new scenario, i.e., when the routing function of partition A itself needs to be reconfigured. If I set a global reconfiguration process, traffic blocking is correlated with the reconfiguration of the whole network, while a local partition implies token propagation only within the partition under reconfiguration. In particular, traffic at switch 0 is stalled for about 12nsec during local reconfiguration, as

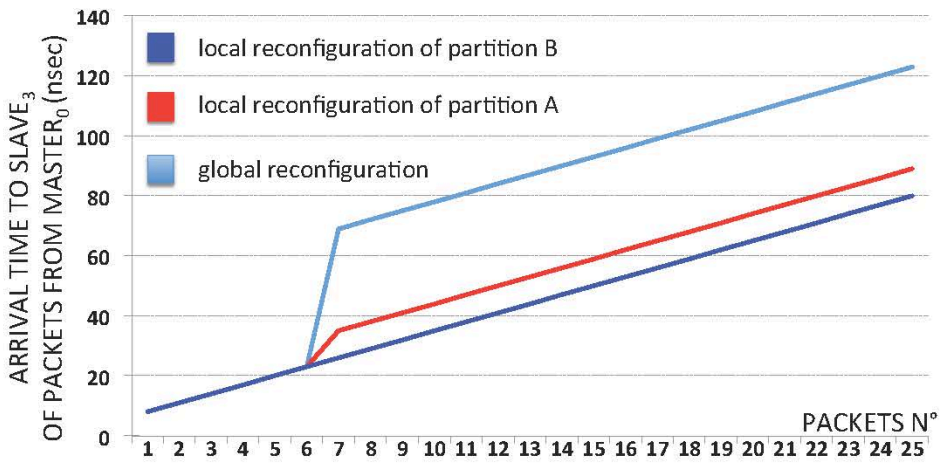


Figure 5.16: Experimental results and benefits of local reconfiguration.

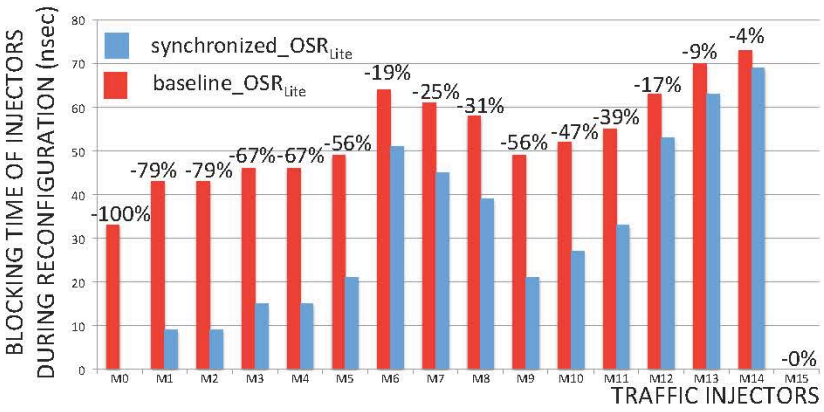


Figure 5.17: Blocking time of all traffic injectors: baseline OSR_{Lite} (red) vs optimized version (blue).

opposed to 46nsec during global reconfiguration, pointing out a significant improvement of about 72%.

5.6.2 Assessing synchronized reconfiguration

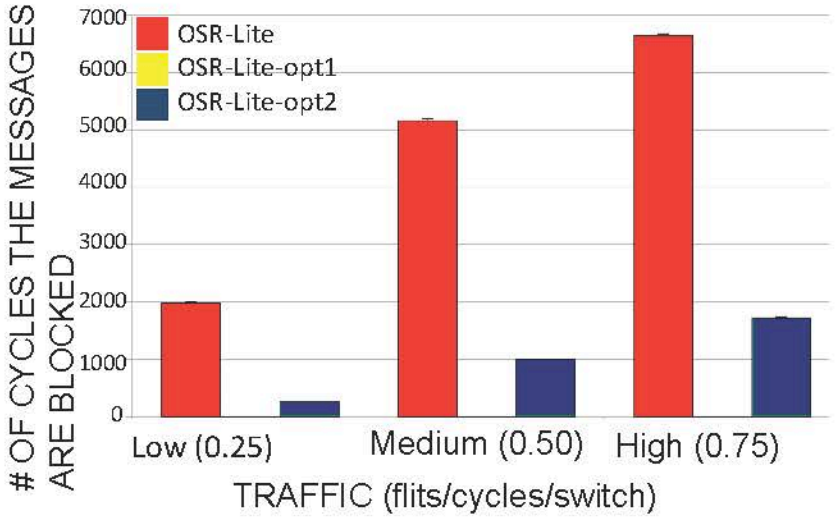
We consider a global reconfiguration involving the whole 4x4 NoC of Figure 5.12(b) , and also traffic generators connected with every input local ports of each switch, injecting traffic with a uniform probability distribution. Assuming 1 clock cycle equal to 1nsec, I compute the cycles during which traffic injectors are stalled by the reconfiguration process (not by the congestion of traffic in the network). We assume that all switches can be reached by the new LBDR bits almost simultaneously, an assumption that I experimentally verified by implementing the dual NoC as an H-tree-like topology (not reported for lack of space). A sub-optimal topology like a ring would add up a further source of reconfiguration time penalty. Hence, I am optimizing an already aggressive design point.

Our optimization significantly reduces blocking time (Figure 5.17). It totally eliminates it if traffic is injected towards an output port that matches the scroll-up token propagation direction, or reduces it if traffic is injected towards the output ports that will transition to the new epoch during the scroll-down phase. Figure 5.17 shows position-dependent speedups, since injectors that are furthest away from the token spreading point (hence receiving tokens later than others) benefit the most from this technique. In contrast, injector *M15* clearly has no improvement (0%) because the token spreads from it and its output ports evolve immediately to the new epoch.

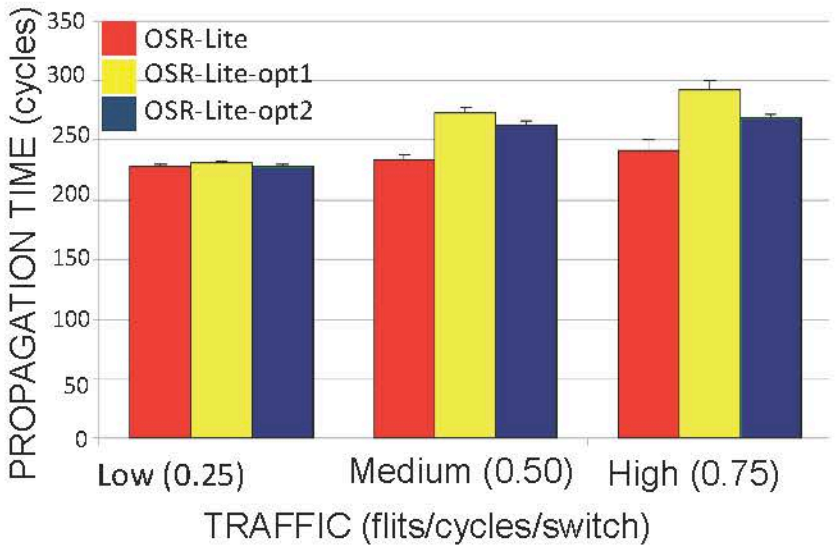
5.6.3 Assessing epoch-conversion reconfiguration

We compare three different schemes of the OSR_{Lite} mechanism: the baseline version and two optimized ones, $OSR_{Lite\text{-}opt1}$ and $OSR_{Lite\text{-}opt2}$, tuned to exploit the availability of two sets of LBDR registers. By synthesizing a 5x5 xpipesLite switch [126] on a

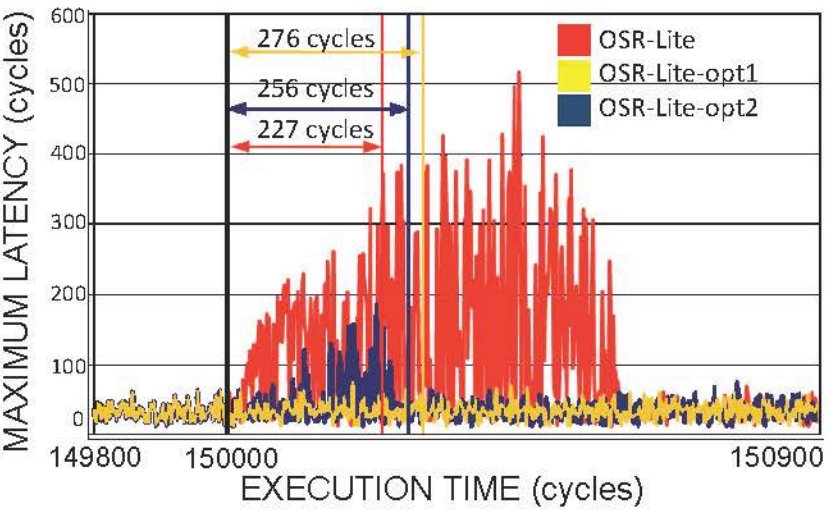
45nm industrial technology library, I derived an area overhead of roughly 8% when implementing two sets of registers per port instead of one, with no impact on the critical path. This overhead is quite reasonable, however in case triple modular redundancy were applied for the sake of fault-tolerance, the impact would be significantly larger. An 8x8 2D-mesh is used with wormhole switching (although the proposed methods also work for virtual cut-through switching) to evaluate the performance of the mechanisms under test. Flit size is set to 4 bytes and messages are 5-flit long. I have performed different simulations varying the injection rate, assuming constant packet generation rate for all end nodes. Figure 5.18(a) shows the number of cycles involved in the token propagation of the different OSR_{Lite} schemes, and taking into account the different injection rates. Each bar depicts the mean of 30 simulations varying the seed and shows the error bars, which represent the 95% confidence interval. Propagation times for the optimizations (both $OSR_{Lite\text{-}opt1}$ and $OSR_{Lite\text{-}opt2}$) are 17% to 12% more than OSR_{Lite} when medium injection rate is considered, since new messages now are not blocked, but rather converted to old ones, ultimately causing more network congestion which slows down token propagation. Figure 5.18(b) shows the number of cycles that the messages are blocked for across all reconfiguration schemes. I observe that $OSR_{Lite\text{-}opt1}$ is able to reduce the blocked messages up to almost 0 and, therefore, the reconfiguration process is totally transparent from the network performance point of view. On the other hand, the $OSR_{Lite\text{-}opt2}$ scheme still blocks some messages but despite this, it has the potential to diminish the blocked cycles by 5 times approximately with respect to the baseline scheme. Figure 5.18(c) represents the maximum network latency for the different reconfiguration schemes under uniform traffic with medium injection rate, where the reconfiguration process is invoked after 150K cycles. For all the schemes, the range for each reconfiguration period is shown, and the start and end times for reconfiguration are indicated with vertical lines. Notice that the x-axis just shows the execution time interval (from 149800 to 150900 cycles) where the three reconfiguration processes are carried out. The most important observation based on the figure is that the optimizations considerably reduce the need to block new messages at the switches during the reconfiguration transient. In addition, for the original OSR_{Lite} scheme and once the reconfiguration is finished, the messages continue experiencing high latencies until the network is stabilized. In contrast, the $OSR_{Lite\text{-}opt2}$ is capable of reducing as much as possible this negative effect while ensuring in-order-delivery of messages. Finally, a better performance is achieved when the $OSR_{Lite\text{-}opt1}$ scheme is enabled, obtaining a transparent reconfiguration process with no impact on the network performance. However, in this case the designer should come up with provisions (e.g., reorder buffers) to counter the risk of out-of-order delivery.



(a) Token Propagation Time



(b) Message blocking



(c) Maximum message latency with medium injection rate

Figure 5.18: Taking advantage of 2 sets of LBDR registers per port.

5.7 Summary

In this chapter I have optimized the OSR_{Lite} reconfiguration mechanism to make it suitable for highly dynamic and shared execution environments, based on the principle of flexible network partitioning. Reconfigurations do not require to drain the network from ongoing traffic, and are local to affected partitions. We have proposed different optimization strategies for network injectors to match increasing resource budgets. To the limit, I prove that fully transparent network reconfiguration is feasible. The work in this chapter paves the way for the frequent and fast partition reconfigurations that future applications will require to handle workload adaptivity, fault-tolerance and quality-of-service. While as a future work, I am investigating fast setup and tear-down of QoS circuits or evaluating the impacts of reshaping partitions with ongoing traffic via the fast reconfiguration mechanisms delivered by this work, as issue this chapter leaves still open the problem of signaling required to communicate with the global manager, in order to deliver the new (re-)configuration bits: the optimized OSR proposed is still a centralized mechanism. This represents an inefficient feature especially in scenarios where testing and, in case of faults, the prompt triggering of the reconfiguration of the resources is needed: in this cases the presence of the central manager is much than an overhead.

Chapter 6

Ultra-low latency, scalable and distributed reconfiguration

The work presented in this chapter exploits the existence of multiple physical networks in industry-relevant many-core processors in a synergistic way, for the sake of fast and scalable distributed reconfiguration of the routing function at runtime, thus enabling runtime testing.

Indeed, extending the principle of partially good die allowance to many-core processors, and testing them over time to detect the onset of permanent faults, are only feasible through proper support in the on-chip interconnection network. This implies the ability to reconfigure the routing algorithm at runtime to reflect changes in network topologies as fast as possible. Current literature cannot avoid a large hardware and/or software overhead when tackling this challenge, so here we want to address this problem deeply optimizing the mechanism of reconfiguration proposed in Chapter 5.

Key novelty: new design point for runtime, fast, scalable and distributed reconfiguration of the routing function in NoCs with a minimal impact on the background traffic.

6.1 Motivation and related works

While most network-on-chip (NoC) research contributions have focused on the architecture design principles or on the physical design flow so far, concerns associated with the low reliability of the silicon substrate at upcoming technology nodes are calling for new design methods for runtime management of the system interconnect. On one hand, sustaining manufacturing yield and device lifetime imply that a failure of a NoC component cannot cause the entire chip to be considered as defective. This goes beyond fault-tolerant routing algorithms [47] or flexible routing mechanisms as developed in

Chapter 5, since the above techniques fail to capture the transition from one network configuration to the next one, which is potentially deadlock-prone and penalizing for instantaneous performance of network traffic. On the other hand, testing complex many-core chips cannot be only a post-manufacturing course of action, but needs to make inroads into the lifetime of the device. One clear trend is toward fault detection and reconfiguration frameworks [84, 56], where network resources are tested aggressively to detect early signs of an upcoming fault through a built-in self-testing infrastructure. The key novelty of the testing challenge lies in the fact that NoC links should be taken offline during runtime testing, while at the same time guaranteeing uninterrupted availability of the NoC. In order to maintain maximum flexibility in link deactivation, the routing algorithm must be able to change dynamically in reply to changes in system state, while preserving deadlock freedom. Current approaches to runtime network configuration suffer from large hardware/software overhead and/or lack of scalability. In general, centralized approaches have the disadvantage that some reconfiguration tasks (e.g., the computation of the new routing function) are performed in software. In contrast, distributed reconfiguration suffers from sub-optimality of emergency routing solutions and overly high implementation cost and complexity. This work moves from a different perspective: the synergistic exploitation of routing resources that are already there in many NoC implementations. In a sense, there is an overhead which is increasingly accepted in NoC design, and which is justified by other design goals, which consists of the use of multiple physical networks instead of logic ones. Although this seems to run contrary to much previous work [142, 57], it is actually motivated by how the relative costs of network design change for implementation on a single die [140]. First, wiring resources are abundantly available on chip, for realistic tile sizes. Second, logic networks do not cut down on the amount of used buffering resources. Third, building multiple physical networks via replication simplifies the design and provides more inter-tile communication bandwidth. This is the reason why up to 5 physical networks can be found in industrial designs. Each one can even be customized for the needs of the specific traffic class it accommodates. Given this, this work proposes to exploit the existing multiple physical networks to spatially separate resource allocations that may close dependency cycles. The most straightforward way of accomplishing deadlock-free spatial separation is to double the number of resources used by a routing algorithm to escape from deadlock and to allow dependencies from new-epoch traffic to old traffic, but not vice versa. Whenever a switch port processing old traffic has a routing dependency with a port already migrated to the new epoch, an escape path is set up into another network plane. This way, deadlock cannot take place. The only requirement the escape network should fulfill consists of its compatibility with the network under reconfiguration from the message-dependent deadlock viewpoint, unless a specific course of action is set up to tackle this concern differently. This work develops a reconfiguration methodology around the above basic ideas and demonstrates a

substantial improvement over state-of-the-art in terms of reconfiguration latency, area overhead, impact over the performance of running traffic, and scalability to large networks.
An overview of existing fault-tolerant routing techniques has been reported by [139]. On one hand, routing tables and logic can be updated upon each fault occurrence [4, 40, 116, 139, 112]. On the other hand, bypass rules can be exploited to reroute around faults using local connectivity information [43, 143].

Runtime reconfiguration of the routing function has been first investigated in high-performance local area networks, spurred by the need to deliver incremental expansion capabilities. Static reconfiguration (SREC) has long been the dominant solution. With SREC, no packets can be routed according to the new routing function while there are still packets in the network routed according to the old one [132]. Dynamic reconfiguration (DREC) techniques overcome this limitation [24, 8, 2]. However, they are applicable only to a limited set of routing functions, or rely on dropping packets to avoid deadlocks, or appear to be more complex than the straightforward static approach, or have requirements on the minimal set of hardware resources implemented. For instance, the double scheme proposed in [107] proposes the spatial and/or temporal separation of the routing resources used by each routing function into two sets, and allows dependencies to exist from one set of resources to the other but not from both at any given time. Unfortunately, it requires the network to implement two sets of data virtual channels. Other approaches strike a trade-off between SREC and DREC. For instance, the work in [89] describes how the various phases of SREC can be overlapped in order to increase parallelism.

When it comes to on-chip networks, a few main approaches stand out. ARIADNE [4] is fully distributed, however it undergoes subtle effects: its latency badly scales with network size, and it does not guarantee a transparent transition between configurations. MD [43] routes packets adaptively through the shortest paths in the presence of a faulty link, as long as a path exists. The local visibility of this mechanism causes the network to become rapidly disconnected as the number of faults increases. OSR_{Lite} [128] has been proposed as an embodiment of native OSR into an on-chip environment. [133] improves [128] with an algorithm that extends coverage of fault patterns. The main issue with OSR_{Lite} is its centralized nature, which causes overhead for manager notifications, and puts software computation of the global manager on the critical path for the reconfiguration process. This latter disadvantage is also shared by the work in [139]. Recently, BLINC has significantly raised the bar for fast, deadlock-free, distributed and localized routing reconfiguration [84]. BLINC uses precomputed routing metadata to quickly evaluate localized detours upon each fault manifestation. Unfortunately, the complexity of this scheme is significant (more than 500 bits per router in an 8x8 mesh, with poor scalability as the network size increases). However, since [84] has demonstrated superior reconfiguration latency and fault tolerance with

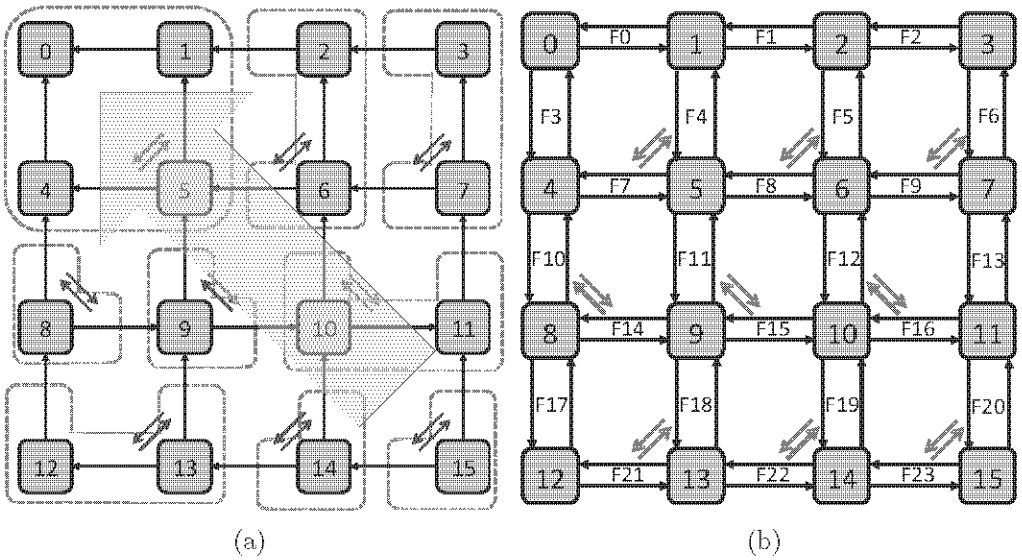


Figure 6.1: 4×4 2D mesh: (a)Segments and scroll-up token propagation, (b) faults ids.

respect to the main competing schemes in literature, I consider BLINC as the reference solution for comparison.

This work aims at a distributed routing reconfiguration method at runtime for NoCs. It borrows the same deadlock-avoidance principle from OSR/OSR_{Lite}, that is, separation of old and new packets with a token. However, the scheme is then augmented to become fully distributed, which was possible due to the same spatial separation concept for deadlock freedom proposed by the double scheme [107]. However, the difference with the original schemes is significant. Differently than OSR, I do not have a centralized control function, since our reconfiguration process is fully distributed. Differently than the double scheme, I do not envision a dedicated physical network only for reconfiguration, but I exploit existing ones, therefore I need to meet the additional constraint of minimum performance perturbation of the background traffic in the escape network through a smart escape strategy. **The outcome is a new design point for runtime and distributed reconfiguration of the routing function in NoCs.**

6.2 Main issues with OSR

The work relies on segment-based routing (SR). SR is topology-agnostic in nature, and works by partitioning a topology into segments (an example is in Figure 6.1). This allows to place bidirectional turn restrictions locally and independently within a segment, thus making the network deadlock-free.

Without lack of generality, I assume the uLBDR (Universal Logic-Based Distributed Routing) routing mechanism as proposed in Rodrigo et al. paper [116]. It has several routing configuration bits at each switch (26) that enable to take the proper routing decision based on the destination coordinates of the packet at hand, and on the routing

restrictions posed by SR. uLBDR supports non-minimal paths through the use of de-routes.

This work moves from the OSR_{Lite} runtime routing reconfiguration function, and ultimately augments it to overcome its global and centralized nature. Before delving into the proposed method, some OSR_{Lite} basics are recalled. In OSR, a global controller is in charge of initiating reconfiguration, either because of a planned decision (e.g., power management) or of an unexpected event (e.g., the likely onset of a permanent fault). In the latter case, the event needs to be notified to the manager. The manager computes the configuration bits for the new routing function to activate, and updates the corresponding registers in NoC switches through a dual NoC. However, the transition from the old to new routing function occurs in a controlled way, so to avoid deadlock. In practice, a separation token crosses the network in the order of its channel dependency graph, starting from a root node. As the token is received at switch input ports, the new routing function is activated as those ports are emptied by old traffic. Similarly, output ports evolve to the new epoch as the input ports that have no routing restrictions toward them have moved into the new epoch. In practice, the network evolves to the new routing function progressively, by enabling concurrent local and static reconfigurations at its switch ports.

Figure 6.1 shows the direction of token propagation across the network. Only the scroll-up phase is shown. The scroll-down phase, which causes the remaining links to receive the token, is omitted for lack of space. Figure 6.2(a) also shows the scroll-up token propagation tree. A switch with a given ID can fire a token in its output ports only when all tokens from the input ports have been received, and have internally propagated to the output ports through the stated rules.

The main issues with OSRLite are:

- the global manager is on the critical path of the reconfiguration process.
- the token propagation starts from a root node.
- a separate control network or virtual channel is needed for communication with the global manager.

6.3 Key idea: synergistic use of multiple networks

We overcome the main limitations of OSR_{Lite} in the direction of a fully distributed and dynamic reconfiguration mechanism by relying on the following key intuitions:

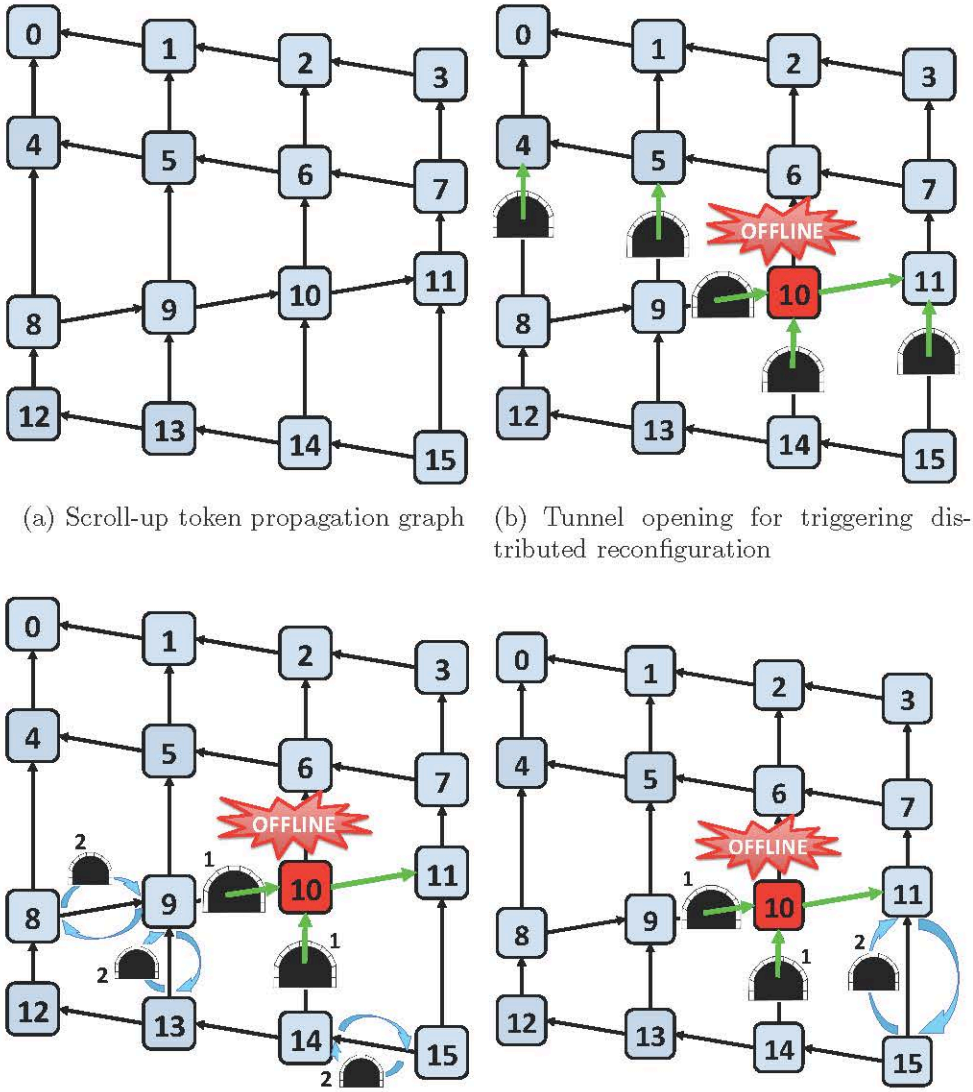
First, considering results of Trivino et al. [56], we can identify a region around a NoC fault that is affected by it. In practice, for each fault in the NoC, I don't have to update all the routing configuration bits of all switches in the NoC, but only of a limited subset of them. This means that it is possible to border the region where the routing function has to be changed.

Second, since each switch is involved only in a limited number of fault regions, the modifications of the uLBDR configuration registers for those cases could be encoded in a small table for each router. The size of the table would be 26 bits for each relevant fault. This way, the routing function should not be computed by a global controller, but would be encoded in distributed tables. The fault coverage of this approach will be addressed in section 6.7.4.

Third, making OSR_{Lite} a distributed mechanism implies that not only the root node, but also every node in the network can trigger token propagation, as an effect of a detected risk of malfunctioning in a link, or of a dedicated testing phase which is about to start in the link under test. Consider for instance Figure 6.2(b), where the indicated link needs immediate disconnection. Switch with ID 10 needs to avoid routing traffic through the critical link. To do that, OSR_{Lite} would require tokens in the two input ports from 9 and 14. Our scheme mimics the same behaviour by redirecting the links of those input ports into an escape network. When this happens, and switch 10 is drained by old traffic, no packets will cross the critical link any more, and a token will appear at the south port of switch with ID 6. For the same reason, a regular token will be concurrently triggered to the east. However, this way the token propagation would stop, because for instance switch 11 needs also a token from south to fire. Similarly for switches 4 and 5. Therefore, I need to open more tunnels. In the next section, this mechanism will be further optimized to reduce inter-network tunneling and speed-up the reconfiguration.

The proposed method has the key requirement of an escape network. For instance, networks carrying intra-partition or inter-core traffic in a many-core processor could be reconfigured on top of a global network (for I/O or memory controller communication). Alternatively, a network carrying one message type could be reconfigured on top of a network carrying a different message type, provided the two message types do not form a dependency chain rising the risk of message-dependent deadlock. For instance, memory requests messages cannot be tunneled into a response network, and vice versa. Nonetheless, another case falls within reach of this work, that is, multiple networks with multiple virtual channels each. For instance the memory request VC of physical network 0 could be tunneled into a memory request VC of physical network 1. This is message-dependent deadlock safe. Last but not least, the mechanism is complementary, that is, the role of the network under reconfiguration and the escape one can be flipped for the sake of exhaustive testing.

Finally, in order to enable the two coupled links during reconfiguration to have different routing functions, escape paths are assumed to go through the local ports of escape switches, hence ending up being multiplexed with the traffic from IP source cores.



(a) Scroll-up token propagation graph

(b) Tunnel opening for triggering distributed reconfiguration

(c) Tunnel propagation

(d) Eager token request

Figure 6.2: Token OSR.

6.4 Baseline mechanism

When putting together the three ideas from the previous section, I get the following reconfiguration methodology. A switch can enforce the fast disconnection of an attached link by triggering the token propagation process. It will handshake the opening of inter-network tunnels with nearby switches. Normal tokens are then fired by the target switch, which will trigger the scroll-up phase of the token propagation. Once completed, the scroll-up phase will trigger the scroll-down phase. Once the scroll-down phase reaches the OSR_{Lite} root node, then the missing scroll-up phase (since the token propagation started somewhere in the middle of the network) will be triggered, till the tokens reach the tunnels and these latter are closed. This completes the reconfiguration process.

Once reached by a token, the input port of a switch will behave like in vanilla OSR, with the difference that a fault identifier needs to travel with tokens. Fault IDs will

be searched in a local CAM memory, indicating whether that fault ID requires routing bit modifications at the switch under test or not. For this reason, the token can be a single-flit special packet carrying the fault ID in its body. There is a fault ID for every bidirectional link in the network (say M), however the number of CAM entries in a switch is limited, since not all faults affects its routing bits.

6.4.1 Identification of the region involved by a fault

The methodology does not need to affect the network as a whole. In fact, thanks to the notion of fault region around a faulty link, only switches in the fault region should be affected by the token propagation. Incoming links from boundary switches may be assumed to already exhibit a token, since incoming traffic will be eventually derouted inside the fault region.

6.5 Optimized mechanism

The three phases of the process (partial scroll-up, scroll-down, residual scroll-up) make it overly long in time. This motivates our next optimizations.

6.5.1 Tunnel propagation

The switch directly attached to the link to disconnect requests tunnel opening to upstream switches with channel dependencies with the link under test. These latter open such tunnels right after pending packets are completed. See for instance switches 9 and 14 in Figure 6.2(c). However, tunnels between 8-4, 9-5 and 15-11 are not opened, thus avoiding the need for dedicated multi-hop signaling. The novelty is that these switches then iterate the mechanism with their upstream switches in the scroll-up token propagation graph. That is, they pretend that tunnels are fired tokens, and try to fulfil the requirements to fire these tokens. For instance, switch 9 will handshake with switches 8 and 13 the opening of tunnels on the connecting links. Once this is completed, and switch 9 has internally processed all pending old traffic, tunnels between 9-10 and 14-10 will be closed, since all the traffic routed across those links will belong to the new routing function (that is, no traffic at all). Overall, tunnels are propagated backwards along the token propagation graph, till they reach the OSR_{Lite} root node. This mechanism overlaps the token partial and residual scroll-up phases.

6.5.2 Eager tunnel request

When switch 11 in Figure 6.2(d) receives the token from switch 10, it temporarily misses a token from south to fire. In order to avoid multi-hop dedicated signaling

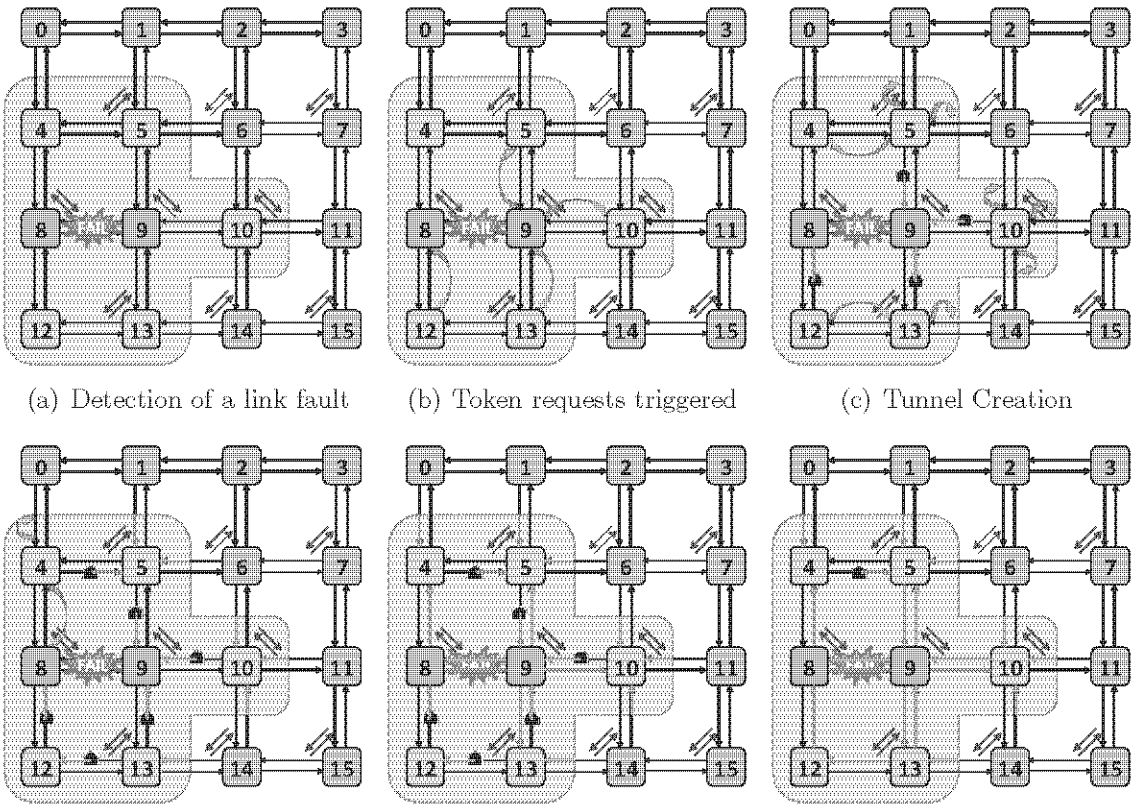
between 10 and 15 to open a tunnel in that location, switch 11 directly asks switch 15 for the missing token through an eager token injection handshaking. Switch 15 opens the tunnel, then in turn applies tunnel propagation with its upstream switches in the token propagation graph. This mechanism is applied by all switches in the network as they receive an incoming token, and speeds up the reconfiguration process significantly. In particular, it is applied also by switch 6 at the opposite side of the link to be put offline. In that case, the switch receives the token from the link under test, and handshakes tunnel opening on those input links that have routing dependencies with the target link. The relevant aspect here is that a few such input links will belong to the scroll-down phase. In turn, switches opening tunnels will propagate them further upstream, thus speeding up the partial scroll-up phase, and overlapping it with the scroll-down phase. Eager tunnel request raises a new condition for stopping tunnel propagation. This latter finishes not only when tunnels reach the OSR_{Lite} root node, but also when tunnel backward propagation is requested for a link which has (or is firing) a token. In that case, the tunnel is closed. An example is illustrated in Section 6.6.

The ultimate effect of our optimizations can be understood as though several spots were enlarging simultaneously on a white surface, thus contributing to cover the whole surface in the smallest possible time. Tunnels are the borders of such spots. As the tunnels propagate as a wave, the incident traffic is de-routed to the escape network, since it is old traffic that is trying to enter a new domain. In order to limit the overall mechanism to a fault region, I enforce that boundary switches of the fault-region do not issue token injection requests for switches outside the region. For this, 4 region connectivity bits per relevant fault ID are needed for each switch. I computed 12 table entries per switch for this purpose.

6.6 Mechanism at work

Let us consider a 4×4 mesh and the link between routers 8 and 9 becoming faulty (although still operational) or under test. Figure 6.3 illustrates the mechanism at work. The sequence of events is displayed under the assumption that token propagation inside switches takes 3 cycles as from the RTL characterization in [128]. In contrast, tunnel opening requests are processed in one cycle.

After detecting the fault event on the link (Figure 6.3(a)), the two switches, connected to the link, trigger the reconfiguration process by sending token/tunnel activation requests to neighbor switches to open tunnels at some of their output ports (those with dependencies with the link under test, see Figure 6.3(b). Once tunnels are opened (Figure 6.3(c)), and after an emptying transient of in-transit traffic of at least 3 cycles, switches 8 and 9 guarantee tokens are triggered through the faulty link (Figure 6.3(e)). Indeed, all input dependencies of the output port connected to the faulty link



(a) Detection of a link fault (b) Token requests triggered (c) Tunnel Creation

(d) Token request at boundaries (e) Token propagation on the link. (f) Tunnel propagation

Figure 6.3: Tunneling Mechanism at Work.

are either inexistent or there is a tunnel at the input port. Thus, it is guaranteed that no old traffic in any direction will arrive needing to cross the failed link.

Figure 6.3(c) shows the initial location of tunnels (at N output ports of routers 12 and 13, at S output port of router 5 and at W output port of router 10) and their equivalence with tokens at upstream switches of tunneled ports. Once tunnels are opened, they trigger new tunnels (Figure 6.3(c)). In particular, switch 12 and 13 request tunnel propagation to the east, since without a tunnel from there it is not possible to close their tunnels on the north ports (see token propagation requirements in Figure 6.1). However, switch 13 is on the boundary of the fault-region, therefore its request is dropped (equivalent to an incoming token in Figure 6.3(d)). Similarly, tunnel opening requests from switch 10 (affecting both scroll-up and scroll-down links) will be dropped. Finally, only one request from 5 is served (Figure 6.3(d)). Always in Figure 6.3(d), switch 4 is showed to be further propagating tunnels at its inputs.

In Figure 6.3(e) the normal OSR_{Lite} token propagation occurs. In particular, switches 8 and 9 have completed the processing latency of their input tokens and can fire this token on the output ports. Interestingly, I can see that the tunnel request from switch 4 is not served because a token is concurrently fired by switch 8 on the connecting link. After 3 cycles, the tunnel between 4 and 5 will be closed (it would be logically in Figure 6.3(g), not shown). It is exactly at the point in time illustrated by

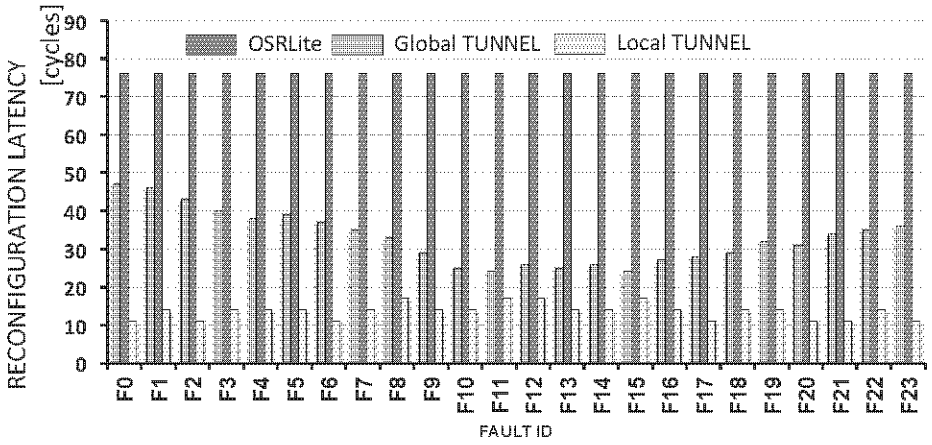


Figure 6.4: Reconfiguration Latency in a 4×4 mesh: baseline OSR_{Lite}(red), Global TOSR (Blue) and optimized Local TOSR (yellow).

Figure 6.3(e) that the link under test is actually put offline.

Finally, in Figure 6.3(f) tokens coming out from switch output ports have closed the associated tunnels. Here, token propagation becomes apparent, since the initial tunnels have disappeared by crossing the fault region boundary, and a derived tunnel is still open awaiting to disappear in the same way.

In order for the reconfiguration process to complete, a few scroll-down links are left (in black in figure), whose reconfiguration will be triggered by switch 5. This latter in fact has all conditions to fire a token to the west, which will in turn cause the token propagation across the remaining links.

6.7 Experimental evaluation

We evaluate our mechanism using an RTL-equivalent SystemC model of the xpipesLite NoC architecture [126] and considering 4×4 and 8×8 2D mesh topologies. I first analyze the reconfiguration latency, then the mechanism overhead, and finally the coverage and the performance impact.

6.7.1 Reconfiguration latency

We evaluate the reconfiguration latency in an unloaded 4×4 mesh network. I perform the analysis for every 1-link failure and for three different mechanisms. The first one represents the native OSR_{Lite} mechanism. The second and third ones represent our mechanism (Tunneled OSR, TOSR) with and without its limitation to the fault-region.

As Figure 6.4 shows, the baseline OSR_{Lite} mechanism (in red) provides a uniform reconfiguration delay, as OSR involves a global reconfiguration process, involving the whole network and starting from the root node. I did not consider possible signaling needs with the global controller, nor the new routing function computation of this latter, but only the pure reconfiguration latency. In blue, I show TOSR without the

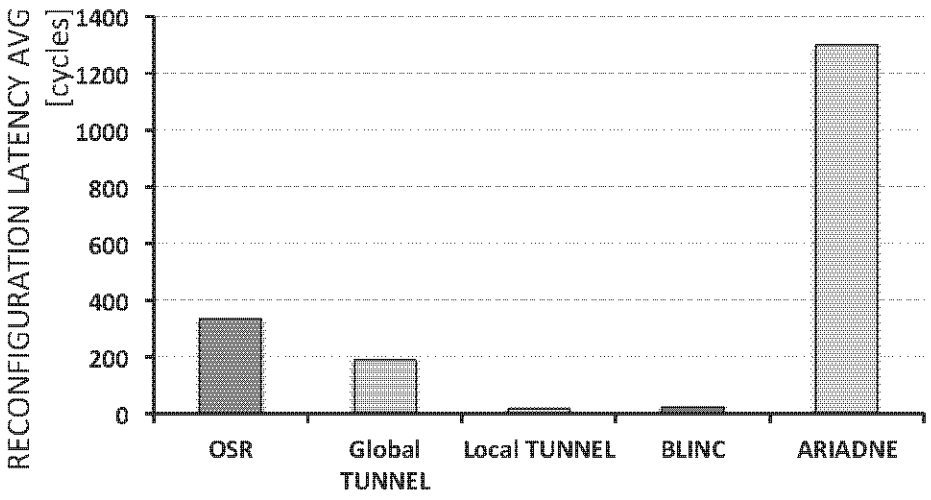


Figure 6.5: Average reconfiguration Latency in an 8×8 2D mesh.

fault-region optimization. As I see, reconfiguration latency is always better than the baseline solution with OSR_{Lite}. This reduction is achieved as the tunnel approach speeds up the reconfiguration process during both the scroll-up or scroll-down phase. As I can observe, TOSR triggered at failures in the center of the mesh achieve lower reconfiguration times as the scroll-up and scroll-down phases are balanced and take the same time. Contrary to this, at the corner of the mesh (link F0) TOSR highly depends on the scroll-down phase which is slower as it needs to wait for the token given by the scroll-up phase to be created with tunnel mechanism.

Finally in yellow, the figure shows the TOSR reconfiguration when the fault-region optimization is included. In our case, the defined region includes from 5 up to 8 routers. As I can see, Local TOSR achieves faster reconfiguration times as the reconfiguration domain is much smaller. The reconfiguration dynamics are also changed, since faults in the middle of the NoC cause the largest fault-regions to be reconfigured.

Figure 6.5 shows the average reconfiguration latency for an 8×8 mesh network. We add available results for BLINC and ARIADNE for this network size (extracted from their publications). Only for (Local) TOSR I consider the worst case latency. Clearly, TSOR achieves about 35% of speedup with respect to the closest competitor, that is BLINC, under the same operating conditions. While BLINC's latency grows weakly with the network size (15% from 8x8 to 10x10), TSOR worst case latency stays constant because the maximum fault bounding region (8 switches) has already showed up in an 8x8 network. Therefore, our gap with BLINC widens.

6.7.2 Area overhead

We express area overhead in terms of number of additional register bits that each mechanism under test requires with respect to the baseline architecture not capable of reconfiguration. Considering the worst case of TOSR, as shown in Table 6.1, it scales better then other solutions proposed because of the fixed maximum dimension the re-

MECHANISM	8x8 2D mesh	16x16 2D mesh
TOSR	464	464
BLINC	648	2368

Table 6.1: Area overhead in terms of register bits.

gion to be reconfigured can reach. We have to consider that, as shown in [56], in the worst case a router can be involved in 16 faults, so it needs 26 LBDR bits for each fault. This creates an overhead of 416 bits per switch. Additionally, to support the TOSR optimization with the token propagation localized in a region around the failure, I need 4 extra Connectivity bits to inform a router that it's on the boundary of a reconfiguration region, so it has to absorb the token request during the propagation. Also in this case I have to consider that a switch can be part of different region boundaries, depending on the position of the fault that is occurring. In the worst case, I calculate that a switch can be part of 12 boundaries, giving 48 additional bits. Already in an 8x8 mesh, this result outperforms BLINC's one, essentially due to its preference list to provide good-enough emergency paths.

As the table shows with a 16×16 mesh, while TOSR stays constant, BLINC's requirements skyrocket, due to the longer children sets and preference list, clearly denoting a lack of scalability of this latter scheme.

6.7.3 Impact on packets' latency

BLINC is too conservative and does not exploit the routing capabilities of SR since multiple valid paths are possible from any pair of source-destination nodes. This negative impact can be alleviated in BLINC by using the expensive preference list table. In contrast, with our mechanism, I manage to use minimal paths for every source-destination pair even when the failure is present. This is achieved since I rely on already computed entries for all the one-link failure cases. Non-minimal paths are taken only when bypassing the failed link, thus being minimal for the topology with the failure. As a result, our mechanism yields a steady state with better-performing routes. Figure 6 in [84] quantifies this penalizing gap for BLINC as an increase by 3% of the average hop count with respect to optimal routes, the same optimal routes that this work provides.

Next, I then explore the impact of tunnel opening on the two coupled networks during the reconfiguration transient. We consider two 4×4 mesh NoCs. The injectors are synthetic testbenches set to generate uniform random traffic.

The first experiment is set considering a medium injecting rate on the network under reconfiguration, with 0% injection rate on the escape network. A reconfiguration is triggered after 45 cycles of the simulation, when the link between switches 8 and 9 needs to be put offline. Figure 6.6 shows instantaneous maximum packet latency over time. Our approach gives improvements from 1% to 11% with respect to OSR_{Lite},

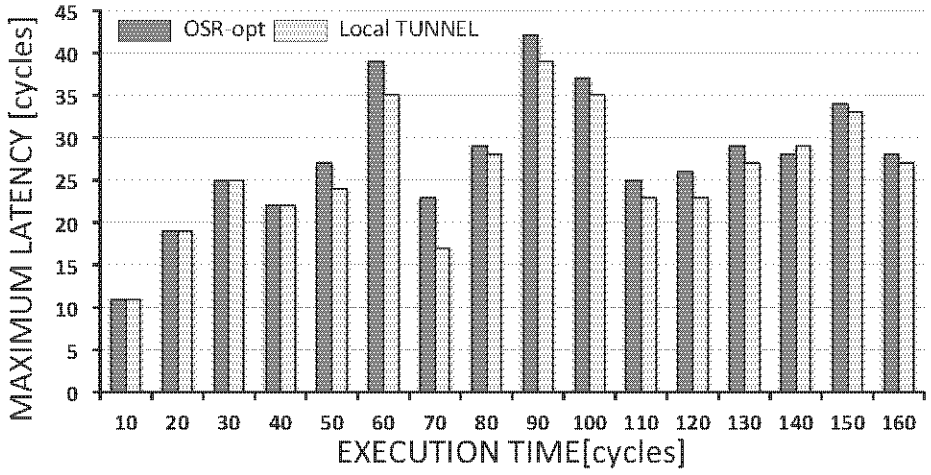


Figure 6.6: Impact on upper-network considering a medium injection rate.

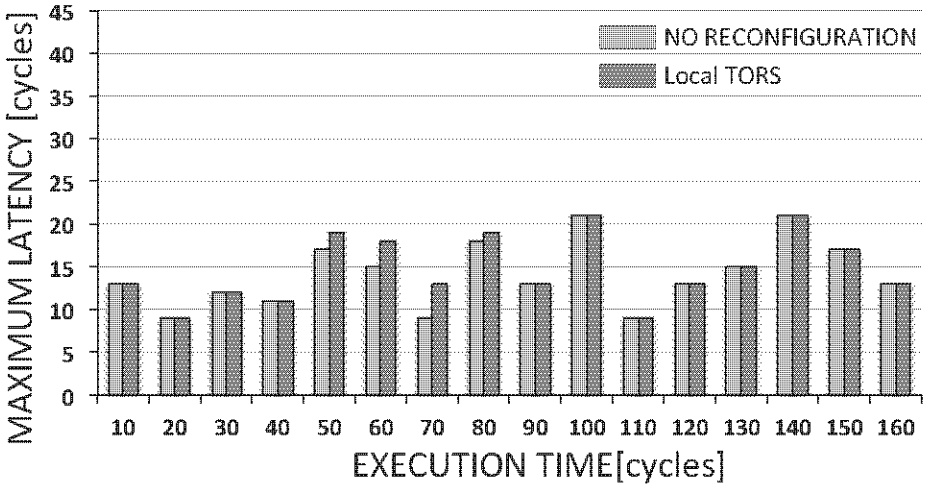


Figure 6.7: Impact on escape network traffic considering a 5% injection rate.

and an average improvement of 6%, which means the escape network is providing the expected improvements in performance predictability during reconfiguration. It is worth recalling that the OSR_{Lite} variant used here is the one which yields quasi-transparent reconfiguration from Chapter 5, hence it is not the native OSR_{Lite}, which would have been trivially outperformed by 40%.

The counterpart of opening tunnels is a perturbation on the traffic in the escape network. I test this effect considering two different setups: first considering 5% of the traffic from the masters injected into the escape network, then 40% . In the former case, the effect of having tunnels opened causes less than 9% worst-case increase of the maximum latency (Figure 6.7), while in the latter case, due to a bigger amount of traffic in the network, the latency is worsened by about 45% in the worst case (Figure 6.8). But being the reconfiguration mechanism very fast, this perturbation dies out quickly (roughly 30 cycles).

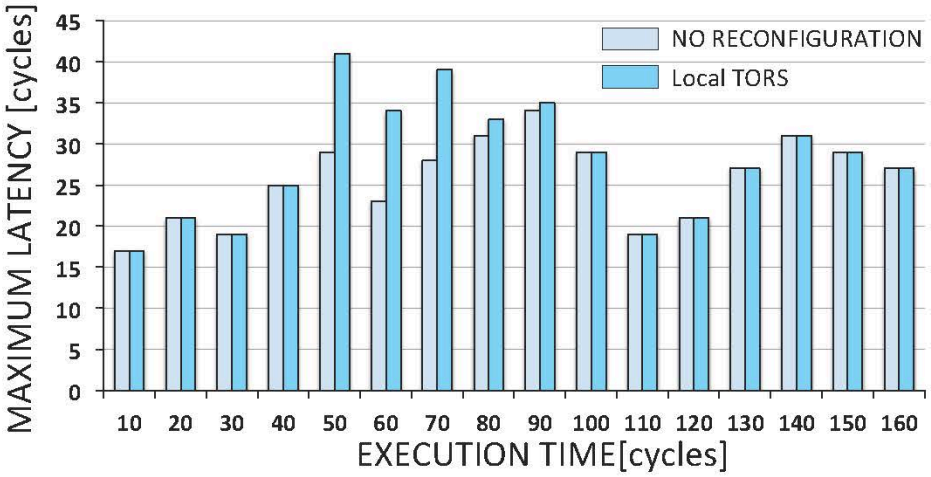


Figure 6.8: Impact on escape network traffic considering a 40% injection rate.

6.7.4 Coverage

Coverage for n-failures is defined as the percentage of n-link failure combinations in a 2D mesh that are supported by the BLINC states that is able to support any failure combination whenever every link failure is localized on a different segment. This is a property of the SR algorithm since it keeps connectivity and deadlock-free conditions by constructions. Indeed, a failed link represents a turn restriction located inside the segment. Therefore, BLINC achieves 100% coverage for 1-link failure cases. For link failure combinations with two failed links in the same segment BLINC relies on an external and conservative solution that will recompute the algorithm again. Therefore, BLINC does not achieve (by its own) 100% coverage for 2-link failures and beyond. Our mechanism achieves the same coverage of BLINC since it also relies on the segment-based approach. That is, I achieve 100% 1-link failure coverage. For 2-link failures the coverage is 98.8% for an 8x8 mesh, which grows to 99.3 for a 10x10 mesh. The proof is in [56], since it relies on a similar table of encoded 1-link faults, but implemented in software. As stated in that work, failure combinations are compatible in the sense that the correct actions to perform in the network is the addition of each individual action to handle each link failure. Coverage increases with network size since the number of segments increases as the network size increases. For the remaining cases, like BLINC I rely on an offline re-segmentation process. However, the application of the new routing function is facilitated by the fact that OSR_{Lite} is already in place.

6.8 Summary

In this chapter, I show that **the synergistic exploitation of multiple physical networks can lead to a fast, low-impact and scalable dynamic reconfiguration of the routing function at runtime**. We bound the area affected by a fault, and devise a mechanism for the fast yet controlled switching of the routing function to

the new epoch in it. I rely on concurrent token and tunnel propagation, thus quickly moving the boundaries between new-old traffic and old-new traffic respectively. We show minimum perturbation of the escape NoC, and only for an overly short amount of time with respect the reconfiguration latencies of competing approaches. The mechanism can finally scale to a large number of cores, since the bounding area of faults stays the same.

Chapter 7

FPGA Prototyping

Today the converging trend toward multifunction integrated architectures is slowed down by the lack of a proper runtime reconfiguration framework of the on-chip interconnect. A runtime reconfiguration is needed whenever the occurrence of events at runtime causes the need for a different resource allocation, such as in the cases for graceful degradation of system performance, power management, thermal control, etc. This thesis has been focused on developing design methods to introduce such a dynamism into the on-chip network, to cope with the highly dynamic environments modern systems need to face, enabling virtualization and space-division multiplexing of the resources . This chapter reports on the prototyping of the design methods developed in previous chapters on a Xilinx Virtex-7 FPGA using Xilinx Vivado IDE and IP Integrator to create the system. Boot-time configuration, runtime reconfiguration of the routing function and dynamic virtualization of the interconnect fabric are especially validated on the FPGA prototype, where a 4x4 multi-core system has been implemented and managed. The advanced form of platform control is achieved via hardware/software co-design and co-optimization.

Key novelty: implementation and functional validation of the design methods on a FPGA board, furthermore evaluating area overhead and critical path after synthesis and Place & Route. First real use case: online selective testing.

7.1 Introduction

NoC design principles have recently reached a stage where they start to stabilize, in correspondence to their industrial uptake. In fact, NoCs are an indisputable reality since they implement the communication backbone of virtually all large-scale system-on-chip (SoC) designs in 45nm and below.

On the other hand, the requirements on embedded system design are far from stabilizing and an unmistakable trend toward enhanced reconfigurability is clearly un-

derway. Reconfigurability of the HW/SW architecture would in fact enable several key advantages, including on-demand functionality, on-demand acceleration, shorter time-to-market, extended product life cycles and low design and maintenance costs. Supporting different degrees of reconfigurability in the parallel hardware platform cannot be however achieved with the incremental evolution of current design techniques, but requires a disruptive and holistic approach, and a major increase in complexity. At the same time, fault tolerance was previously an issue only for specific applications such as aerospatial. Today, *due to the increased variability of components and breadth of operating environments, reliability becomes relevant to mainstream applications.* Similarly, new reliability challenges cannot be solved by using traditional fault tolerance techniques alone: the reliability approach must be part of the overall reconfiguration methodology.

In the highly parallel landscape of modern embedded computing platforms, the system interconnect serves as the framework for platform integration and is therefore key to materializing the needed flexibility and reliability properties of the system as a whole. Therefore, time has come for a major revision of current NoC architectures in the direction of increased reconfigurability and reliability.

In addition, a key property that novel NoCs cannot miss is to guarantee a potentially fast path to industry, since NoC deployment is today a reality. An important requirement for this purpose is the efficient testability of candidate NoC architectures. This property is very challenging due to the distributed nature of NoCs and to the difficult controllability and observability of its internal components. When I also consider the pin count limitations of current chips, I derive that NoCs will be most probably tested in the future via built-in self-testing (BIST) strategies.

Finally, there is an increasing need in embedded systems for implementing multiple functionalities upon a single shared computing platform. The main motivation for this are the constraints set for systems size, power consumption and/or weight. This forces tasks of different criticality to share resources and interfere with each other. Integration of multiple software functions on a single multi- and many-core processor (multifunction integration) is the most efficient way of utilizing the available computing power. For a mixed-criticality multifunction integration, the NoC should be augmented to support partitioning and isolation, so that software functions can be protected from unintended interferences coming from other software functions executing on the same hardware platform. This feature is a key enabler for the virtualization of embedded systems, that is, an effective and clean way of isolating applications from hardware.

This chapter reports on the prototyping of a Network-on-Chip capable of supporting all of the advanced features described above. The presented prototype on the GP-NaNoC switch the innovative design methods proposed in this thesis, *raising the level of abstraction of the network as whole, envisioning it to work as hardware support for highly dynamic environments required by modern heterogeneous systems.* Then

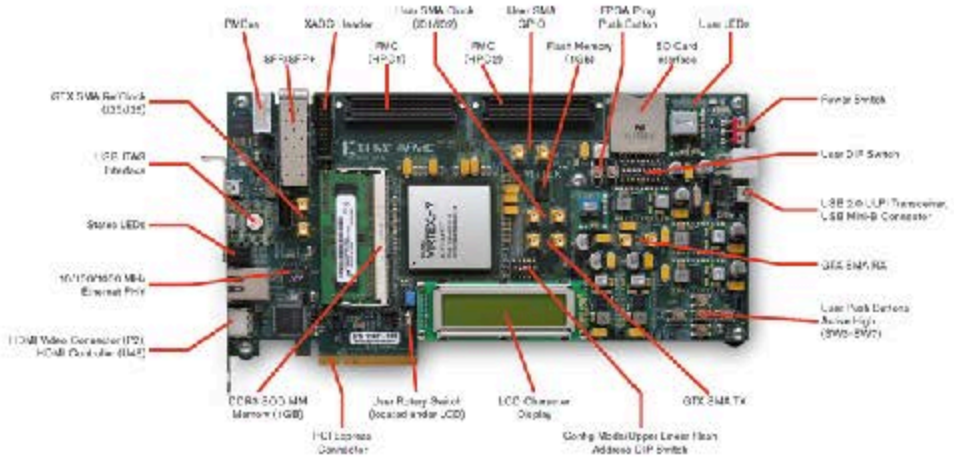


Figure 7.1: VC707 baseline prototyping board.

it validates the (re-) configuration capabilities that preserve safe network operation in the presence of wanted (e.g., virtualization) and unwanted (e.g., manufacturing defects, intermittent faults) effects, relying on the most optimized OSR mechanism developed in this work with the synergistic use of multiple networks. The prototyped system implemented inside the FPGA is a homogeneous multicore processor, which resembles programmable hardware accelerators of hierarchical, high-end embedded systems, or basic computation clusters of many-core processors.

The rest of the chapter is organized as follow. I described the prototyping platform, represented by the Xilinx Virtex-7 evaluation board named VC707, then the readers can find the high-level view of the platform, where I provide an overview of the design-flow for platform implementation and present the basic components adopted in the demonstrator that is composed of a NoC-based system and a supervision system; finally it describes the application and the reconfiguration algorithm running respectively on top of these latter systems. Next in this chapter I demonstrate runtime reconfiguration of the routing function upon transient failures, highlighting the dynamic virtualization of the interconnect fabric. Finally I present a possible application of this design that is runtime testing.

7.2 FPGA platform

The target system to prototype is overly complex, hence calling for high-end FPGAs and development boards, not to incur integration capacity limits.

The Virtex-7 FPGA VC707 Evaluation Kit was selected for our task. It is a full-featured, highly-flexible, high-speed serial base platform using the Virtex-7 XC7VX 485T 2FFG 1761C and includes basic components of hardware, design tools, IP, and pre-verified reference designs for system designs that demand high-performance, serial connectivity and advanced memory interfacing. The included pre-verified reference designs and industry-standard FPGA Mezzanine Connectors (FMC) allow scaling and

customization with daughter cards. The XC7VX485T FPGA features 485760 logic cells, 75900 CLB slices, 2800 DSP slices, 37080 kb of block RAM, 14 total I/O banks and 700 max. user I/O.

The key features of the evaluation board (see Figure 7.1) are as follows:

- **GA VC707 Evaluation Kit**: ROHS compliant VC707 kit including the XC7VX 485T 2FFG 1761 FPGA
- **Configuration**: Onboard JTAG configuration circuitry to enable configuration over USB, JTAG header provided for use with Xilinx download cables such as the Platform Cable USB II, 128MB (1024Mb) Linear BPI Flash for PCIe Configuration, 16MB (128Mb) Quad SPI Flash
- **Memory**: 1GB DDR3 SODIMM 800MHz / 1600Mbps, 128MB (1024Mb) Linear BPI Flash for PCIe Configuration, SD Card Slot, 8Kb
- **Communication and Networking**: GigE Ethernet RGMII/GMII, SGMII, SFP+ transceiver connector, GTX port (TX, RX) with four SMA connectors, UART To USB Bridge, PCI Express x8 gen2 Edge Connector (lay out for Gen3).
- **Display**: HDMI Video OUT, 2 x16 LCD display, 8X LEDs
- **Expansion Connectors**: FMC1 - HPC (8 XCVR, 160 single ended or 80 differential, user-defined pins), FMC2 - HPC (8 XCVR, 116 single ended or 58 differential user-defined pins), Vadj supports 1.8V, IIC.
- **Clocking**: Fixed Oscillator with differential 200MHz output used as the system clock for the FPGA, programmable oscillator with 156.250 MHz as the default output, default frequency targeted for Ethernet applications but oscillator is programmable for many end uses, differential SMA clock input, differential SMA GTX reference clock input, Jitter attenuated clock used to support CPRI/OBSAI applications that perform clock recovery from a user-supplied SFP/SFP+ module.
- **Control and I/O**: 5X Push Buttons, 8X DIP Switches, Rotary Encoder Switch (3 I/O), AMS FAN Header (2 I/O).
- **Power**12V wall adapter or ATX, Voltage and Current measurement capability.
- **Debug and Analog Input**: 8 GPIO Header, 9 pin removable LCD, Analog-Mixed Signal (AMS) Port.

7.3 Baseline System

An ambitious Virtex 7 FPGA-based platform was conceived for this research project. The high-level view of the design can be found in Figure 7.2. Here I present the baseline

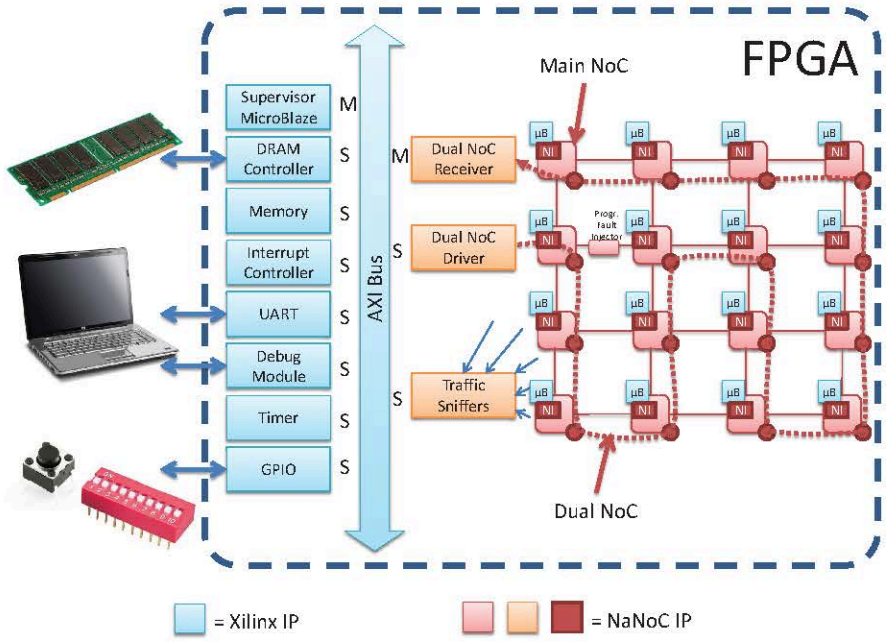


Figure 7.2: FPGA platform overview.

system that is the starting point of my work and that I enhanced with new features developed in this thesis.

The system comprises a large number of components within the FPGA. As can be seen on the left side of the diagram, a relatively standard Xilinx subsystem is instantiated first; this comprises an AXI interconnect linking together a MicroBlaze (to run the supervision software), a small memory and an external DRAM controller, and several peripheral controllers required to run software on the MicroBlaze and to communicate with a laptop.

The right side of the diagram depicts the components that have been modified in this thesis, including some that were specially developed for the FPGA prototype and will be presented in this chapter. This part of the system, after my modifications is the "Device Under Test" of the platform, whose functionality is to be verified. It comprises mainly:

- The main NoC, built as a 4x4 mesh of the GP-NaNoC switch architecture presented previously.
- The dual NoC, built as a chain that follows the topology of the main NoC. The dual NoC is in charge of configuring the main NoC and of collecting status information (e.g. fault detections) from the main NoC.
- At each node of the main NoC (see also Figure 7.4), a MicroBlaze and a memory (by Xilinx) are connected to the switch by means of Network Interfaces. The MicroBlaze NI has an AMBA AXI NI while the memory is given an AMBA AHB NI.

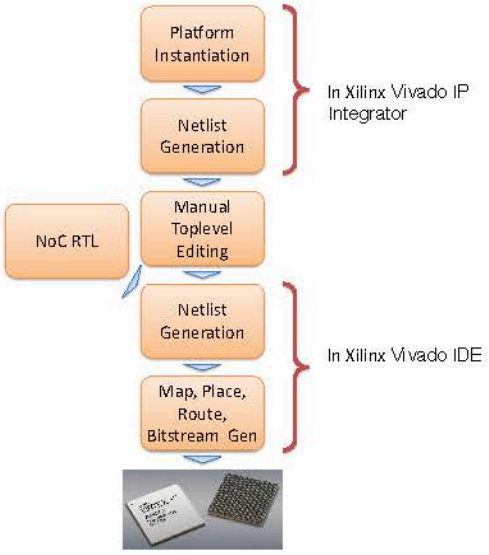


Figure 7.3: Design flow for platform implementation.

- Two special blocks have been designed in this thesis to connect the dual NoC to the supervision subsystem. These blocks allow the supervision MicroBlaze to receive notifications by the dual NoC, and to reprogram it.
- A sniffer module monitors traffic along all links of the main NoC mesh, computing link utilization. It is designed so that the supervision subsystem can probe it at regular intervals and transfer its contents towards a user's laptop.
- A fault injection module has been instantiated along a mesh link. This simple module, connected to a physical button on the FPGA board, provides a method to inject faults on that link to test the platform's fault-tolerance and the NoC reconfiguration capability..

To build this platform, I proceed in steps (Figure 7.3). First, I instantiate within Xilinx Vivado IP integrator a complete design comprising all the supervision subsystem, the 16 additional MicroBlazes, and the corresponding 16 memories. At this stage, no NoC is instantiated yet. Using Vivado for this task allows us to efficiently connect and configure all the Xilinx blocks, and facilitates the instantiation of the top-level HDL files. Additionally, this makes it possible to subsequently load the applications into all 17 MicroBlazes memories, and to debug those processor step-by-step, directly through the Xilinx toolchain, which is Eclipse-based. After the first pass of synthesis, however, we remove from the design the Xilinx AXI subsystem which is connecting the 16 additional MicroBlazes and memories, and swap in the NoC (main and dual) in its place. I then proceed to finish the implementation flow within Xilinx Vivado by performing mapping, placement and routing, and generating the final bitstream. We leverage some key features of the Virtex 7 board, apart from the FPGA chip. The on-board DRAM is used to provide sufficient space for the software running on the supervision MicroBlaze to work. Physical buttons and switches of the board are

connected to an on-chip GPIO controller to allow the user to interact with the platform. Finally, a laptop can be connected to the board by means of two cables to monitor the platform's operation; one cable carries serial port signals (piggybacked onto a USB port) and the other carries JTAG signals (also piggybacked onto a USB port). The former is used to read the board's outputs, while the latter allows for programming the board and interactively debugging the on-FPGA MicroBlazes. An Ethernet cable had initially been considered instead of the serial interface, in light of its higher throughput, but the Ethernet PHY of the board was found to be defective.

Custom-written software runs in three locations of the system: on the supervision MicroBlaze, on the 16 MicroBlazes connected to the main NoC, and on the external laptop.

- The software on the supervision MicroBlaze is tasked with oversight of the main NoC and data NoC, with regular polling of the Traffic Sniffers, and with interfacing with the external world through the serial interface.

- The 16 MicroBlazes connected to the mesh run micro-benchmarks developed in this thesis. These micro-benchmarks have the main role of generating traffic on the mesh, so that the various platform features can be tested. Real functional behaviour was implemented: the nodes perform pipelined matrix multiplications, exchanging data in producer-consumer fashion. More advanced applications could not be implemented due to the lack of I/O interfaces on these nodes and due to lack of memory to instantiate a full C library.

- The user's laptop is connected to the board through a JTAG-over-USB cable and a serial-over-USB cable. The former can be leveraged mainly by the Xilinx toolchain, allowing for board programming and debugging. The latter is monitored to display in real-time the platform status and link utilization

7.3.1 Basic components: the on-chip network

A 4x4 mesh with one core and one memory per switch has been chosen as target on-chip network of the FPGA platform. In particular, Figure 7.4 represents the basic components instantiated to realize the 4x4 mesh. A MicroBlaze and a memory are connected to each switch through two Network Interfaces. Finally, a sniffer is placed on each bidirectional network link to monitor the network traffic. The sniffers collect information about the traffic crossing the switch-to-switch and NI-to-switch links and deliver such information to the global manager (i.e., the supervision MicroBlaze). Both the NIs and the switches have been designed ad-hoc to support the target on-chip network where fault-tolerance, testing capability and reconfigurability features are guaranteed. Note that the MicroBlaze also includes a directly-connected BRAM of 128 kB (not shown in the figure) to store its application software; loading the binary image of the application

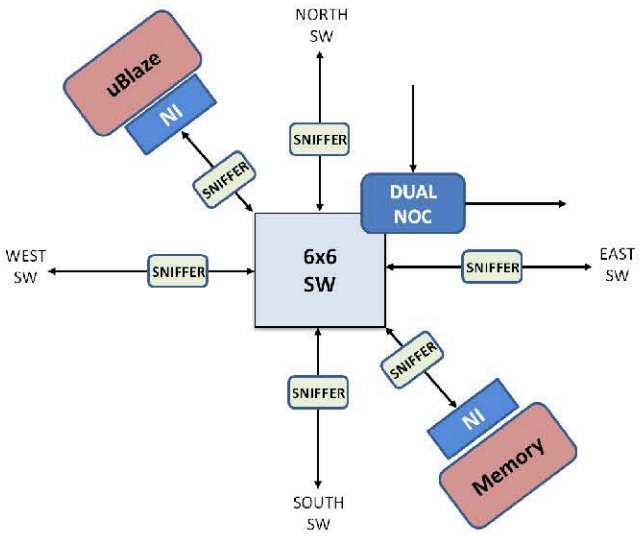


Figure 7.4: Basic components of the on-chip network.

into the AHB memory would be unnecessarily problematic from the toolchain viewpoint. However, I explicitly use the AHB memory as storage and for inter-processor communication in the application.

Network Interface

We instantiate two types of NIs: an AXI initiator NI to interface with the MicroBlaze, and an AHB target NI to interface with the memory. This choice was deliberate (e.g., both could have been AXI) to demonstrate interoperability among the two. NI used in this implementation is reported in Figure 7.5

Due to the relatively simple needs of the MicroBlaze core, which does not support multiple transaction IDs, I save area by instantiating a small AXI initiator NI with support for only one such ID. However, the NI is still supporting all AXI features. Both AXI and AHB NIs, and their interoperability, were extensively tested in RTL and on the FPGA.

The Switch

The GP-NaNoC switch architecture adopted in this work is an extension of the GP-NaNoC switch presented in the previous chapter. The switch implements logic-based distributed routing (LBDR), relies on wormhole switching and implements both input and output buffering. The crossing latency is thus 1 cycle in the link and 1 cycle inside the switch. This section briefly summarizes the key features of the switch together with the extensions introduced to meet the target FPGA platform requirements.

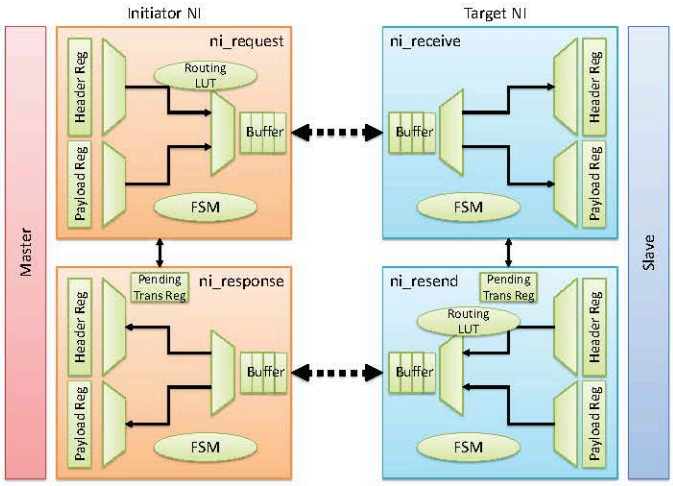


Figure 7.5: Network Inteface blocks diagram.

Routing Logic extensions

In principle, the LBDR selection logic computes the destination output port by reading the destination address information contained in the header flit of each packet. In particular, the LBDR logic performs a comparison between the destination address (Dest ID) and the local switch ID (Local ID). When the local switch ID matches the destination address, the packet is forwarded to the local port (i.e., to the core). However, the FPGA platform is enhanced with two nodes per switch thus each switch integrates two local ports. As a result, further information must be added to the incoming destination address of the header flit and the LBDR logic must be extended to determine whether the packet should be routed to the first or the second local port. The destination address information has been extended by 1 bit (Core Flag). The additional bit is exploited to determine the target core at the destination switch. If Dest ID matches Local ID, the Core Flag bit is used to distinguish between the two local ports.

7.4 System Under Test with Xilinx Vivado

The system described in previous section was realized before this thesis in Xilinx ISE and Xilinx Platform Studio. Being 7-series boards moved to Xilinx Vivado, a new tool provided by Xilinx, part of the work for this chapter consist of porting the original platform code to let it work in Vivado Design Suite. This step is fundamental because, as shown in Figure 7.6, the new tool guarantees a better optimization of the resources utilization of the FPGA, enabling to implement the escape network that is at basis of the optimization of the reconfiguration mechanism, and also cuts-off the synthesis, place & route and bitstream generation times.

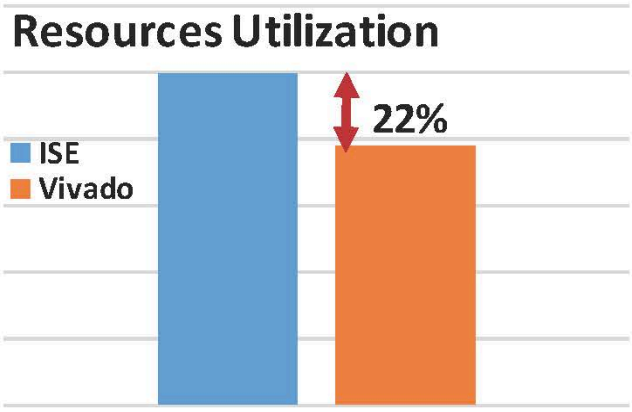


Figure 7.6: Different resources utilization in terms of Look-Up Table: ISE vs. Vivado

7.4.1 The physical platform implementation

After the porting steps, modifying the needed blocks, I finally implemented in Verilog and VHDL code the enhanced routing reconfiguration mechanism, enriching the routers of new features and implementing also the logic to support and control the tunnels-tokens propagation, as reported in Figure 7.7.

Again, it is necessary to point-out that, doubling the networks and adding new logic, finally having the opportunity to test the system is feasible thanks to Vivado that runs optimized synthesis (up to 250MHz for this system) and Place & Route algorithms: in ISE in fact the total utilization of resources required by the new system

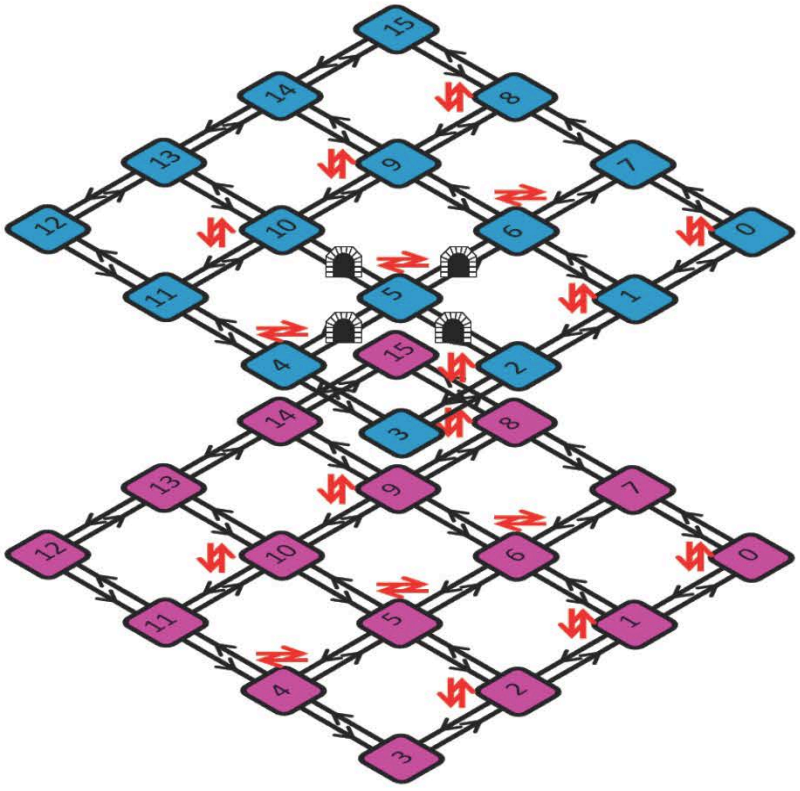


Figure 7.7: Escape network and tunnels mechanism implementation.

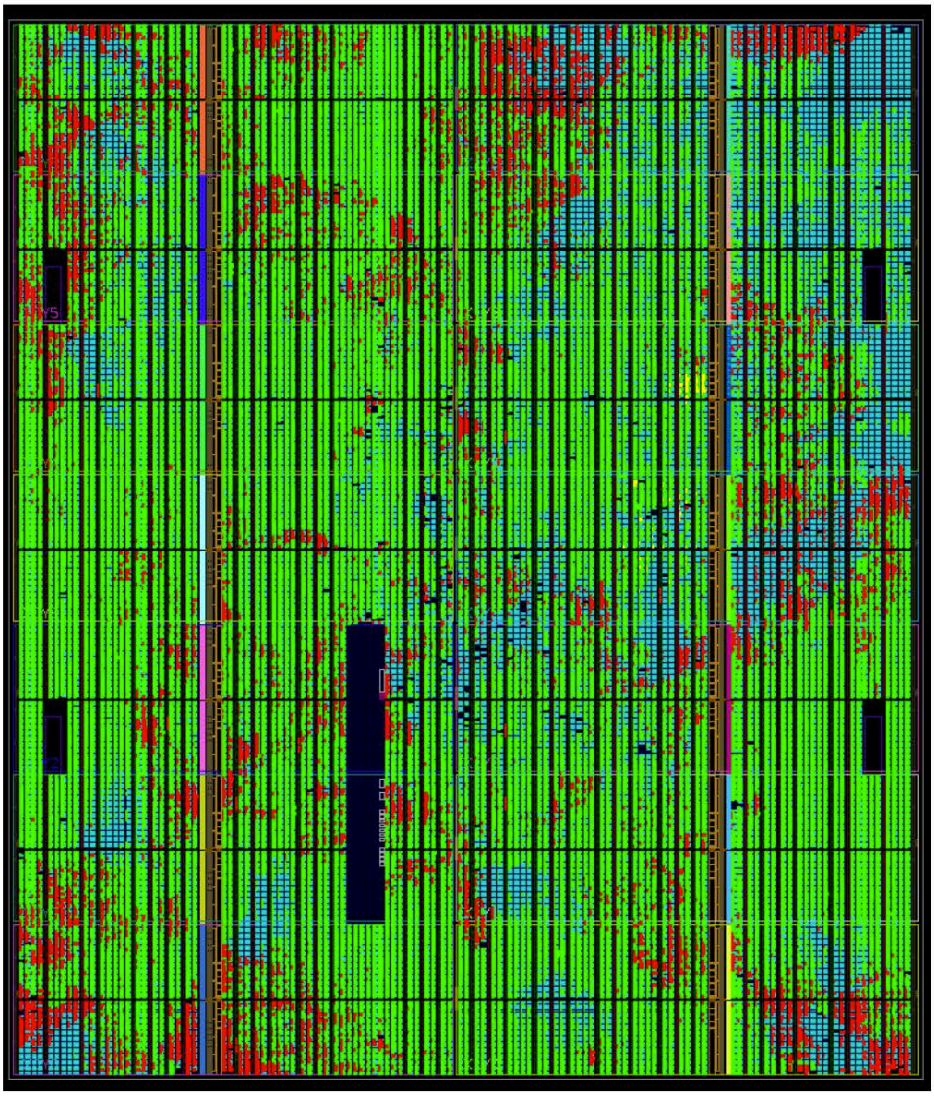


Figure 7.8: Layout of the full FPGA design.

was not supported because it exceeded from the total memory of the FPGA.

Some steps of the implementation flow described in Figure 7.3 can be parallelized; for example, the initial platform description involves several blocks which can be independently synthesized in parallel. Even after joining all the pieces together, the mapping stage can be run on two threads in the Xilinx toolchain, and the placement and routing in four. Despite this, I measure end-to-end flow runtimes of about 5 hours on a dual-chip Opteron 6378 (16 threads/core) server with 128 GB of RAM. I observe peak memory utilization close to 10 GB during implementation. The layout of the platform implementation can be seen in the screenshot of Figure 7.8. The occupation of the FPGA resources is around 98%.

The very high resource utilization features impose a significant timing overhead as routing necessarily becomes more convoluted and less efficient. I record a maximum operating frequency of 100 MHz.

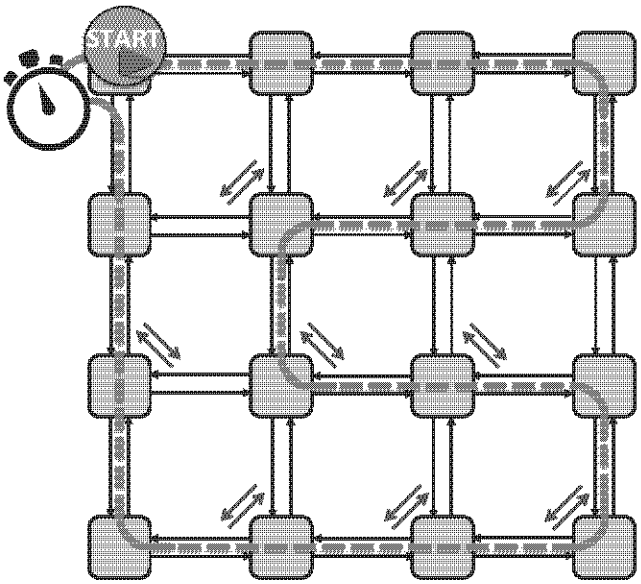


Figure 7.9: Path followed throughout the network by testing token, to sequentially test all the links.

7.5 New mechanism's application: Lifetime Testing

The baseline mechanism has a testing infrastructure that enables, at boot time, to test at the same time all the links of the network, freezing the application and let it start after the end of the test phase. This does not match with *runtime testing* requirements.

So to cope with the latter ones, in this section, I describe the modifications needed to enable runtime testing, relying on these main ideas:

- I implement a *testing token propagation* mechanism (based on different tokens, not the reconfiguration mechanism ones. This token, reaching the switch, enables the testing of the links of the router, that keeps the token until the end of the testing. The token propagation is through dedicated links and follows the path shown in Figure 7.9, moving between adjacent switches.

- When a switch receives the token, it checks for the links that are not yet tested (the testing procedure in fact involves the bidirectional links, so during the loop propagation of the token some of them could be already tested and it is not necessary to re-test them, and differently from the original testing phase, it tests sequentially the remaining links, properly triggering the reconfiguration procedure and keeping the possibility to route the traffic.

This way, at the end of the testing token propagation, each link of the network is tested without making unavailable all the network: in the regions that are not under test, indeed, the applications can run normally. As Figure 7.9 shows, I add a timer to the design to let this testing be repeated after a pre-selected amount of time, guaranteeing to find faults, if present in the network, with a good responsiveness.

7.5.1 Mechanism at work

Being the testing on bidirectional links another modification is needed: in fact the switch with the testing token needs to communicate to its adjacent switch that it must test the logic connected to the link under test. To make this feasible, I add communication signaling between the two switches, to let them organizing the bidirectional testing phase and control tokens and tunnels. When tokens from both the parts of the link I want to test are produced by the logic inside the routers the real test of the link can finally started, relying on BIST infrastructure, as seen before already part of the baseline router.

In Figure 7.10 are reported the main steps of the complete testing mechanism, better explained as follows:

- 1- Switch ID0 (SW0) is the first that receives the testing token (triggered by the reset, or boot, of the system once, then timed) and starts the test of its East link. To dismiss the link, and this way avoid traffic to be injected through it, SW0 send a tunnel request to switch ID4 (SW4) because South port is the only one having dependencies toward the east port of SW0, so this latter needs to have no traffic coming from that input port.

- 2-3- SW4 receives the request from SW0 and, after the tail of the current packet routed towards SW0 has been processed, it can finally open the tunnel. All the packets from SW4 to SW0 are now tunneled in the escape network. A signal, the same that triggers the opening of the tunnel, reaches also the input port of SW0, mimic a real OSR token coming from South, thus hacking the original OSR mechanism and let the local port to migrate its epoch to the new one, that takes into account the unavailability of the east link.

- 4- SW0 ha now the routing function reconfigured and so can send a token to East, informing switch ID1 (SW1) that SW0 wants to trigger the thes of the biderctional link. This way, an internal local manager allows SW1 to prepar to test its West link for the testing phase.

- 5- Following the same steps of SW0, SW1 sends a tunnel request towards SW5 North port and SW2 West port, thus triggering the opening of the tunnels.

- 6- As point 3, after waiting for the tail of the packet, SW2 and SW5 open the tunnels towards SW1, mimicing real tokens for OSR and triggering reconfiguration of the routing function of SW1

- 7- Having received all the tokens from input ports that have dependencies with the West port, thus ensuring that traffic cannot longer be routed on it until the end of the testing phase, SW1 sends a token-out signal on its West link.

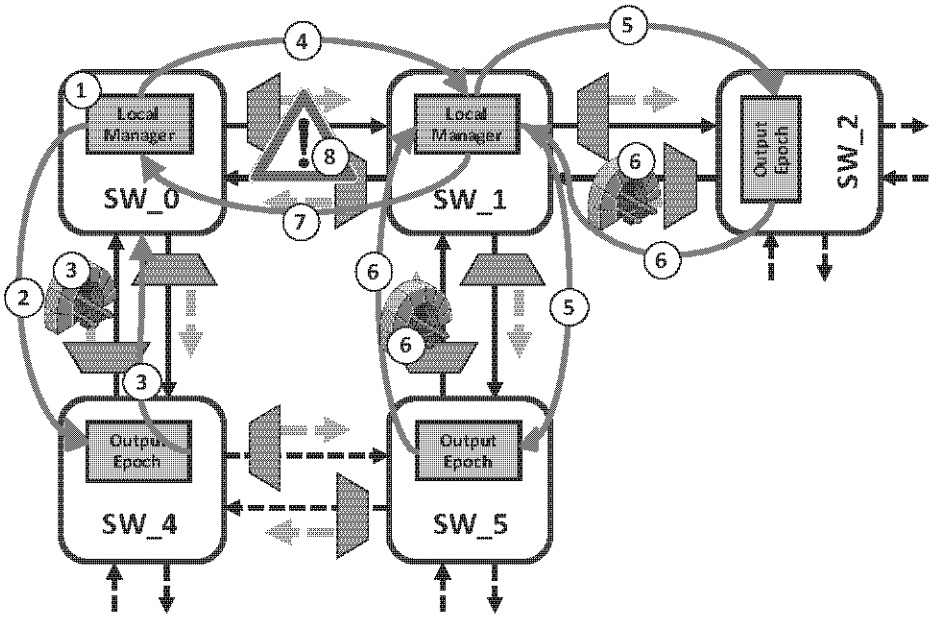


Figure 7.10: Main steps of the testing mechanism: lightblue arrows indicate the paths towards the escape network.

- 8- Now, the logic at the basis of the mechanism ensures that no trafic can be routed on the bidirectional link, so the effective testing, relying on BIST, can finally starts to check if faults occurred.

When the testing phase of the link ended, the testing token is passed to SW1 that will continue the selective and sequential test of its links.

7.6 Experimental results

7.6.1 Area overhead

Now I am presenting the results obtained after synthesis and Place & Route procedures by Xilinx Vivado tool. Figure 7.11 shows the overhead at switch-level at 30%, mainly due to look-up tables needed to encode a fault ID and the necessary routing bits that, being the mechanism not managed by a software hypervisor but distributed, are necessary to understand and select the actions to trigger when a fault occurs. This can be significant, but it represents a quite good trade-off considering the features that the router now has.

The situation becomes better considering the whole system. In fact, as depicted in Figure 7.12, the routers represent a relative part of it and this let the total area overhead decrease at 13%, that is acceptable. This result relies on the fact that the secondary network is already present on-chip, but the mechanism introduces additional logic to exploit it also for the reconfiguration procedure. The overhead considered in this case, apart the relative contribution of the routers, is due to multiplexers, demultiplexers, signals and internal logic to manage the events and create the tunnels towards the

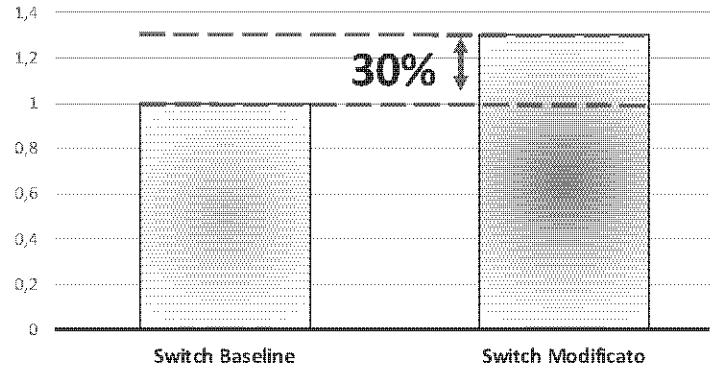


Figure 7.11: Area overhead: baseline switch vs. enriched switch enabling runtime testing.

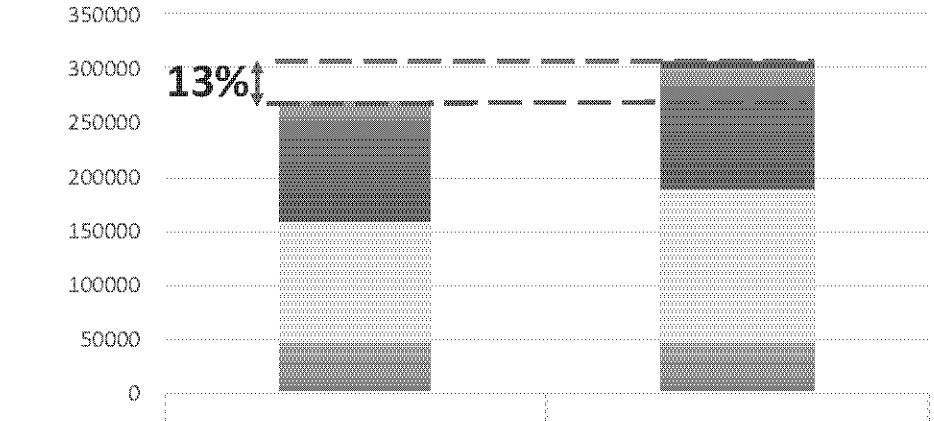
secondary network.

7.6.2 Critical path

There are no variations on critical path, because on both systems compared in this thesis, it is due to a path that is not modified. This results is very promising because it means that the additional logic does not burden the critical path of the system, thus avoiding issues for the new system to correctly work on the FPGA board.

7.6.3 Reconfiguration time

Now I move on taking into account the latency of the reconfiguration transient. To get the results I consider a 4×4 2D mesh without ongoing traffic and reported in Figure 7.13.



Additional Logic	0	7808
Interfaces	17875	17875
Escape NoC	92952	92952
Main NoC	109568	139476
Host	24860	24860
Dual NoC	3456	3456
Microblazes	20329	20329

Figure 7.12: Area overhead: baseline system vs. system enabling runtime testing.

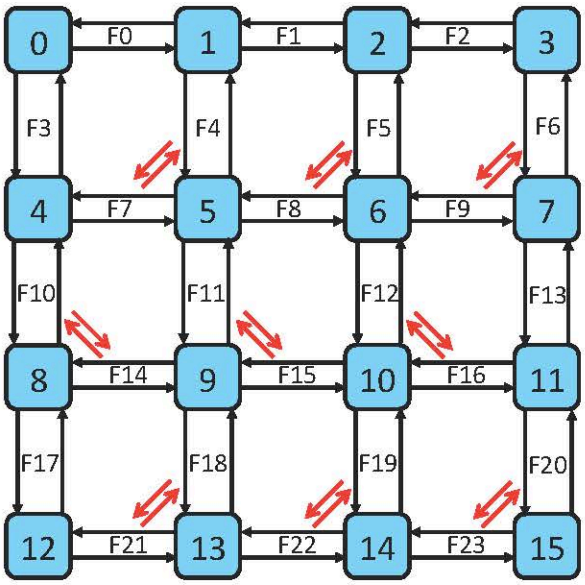


Figure 7.13: Fault IDs for a 4×4 2D mesh.

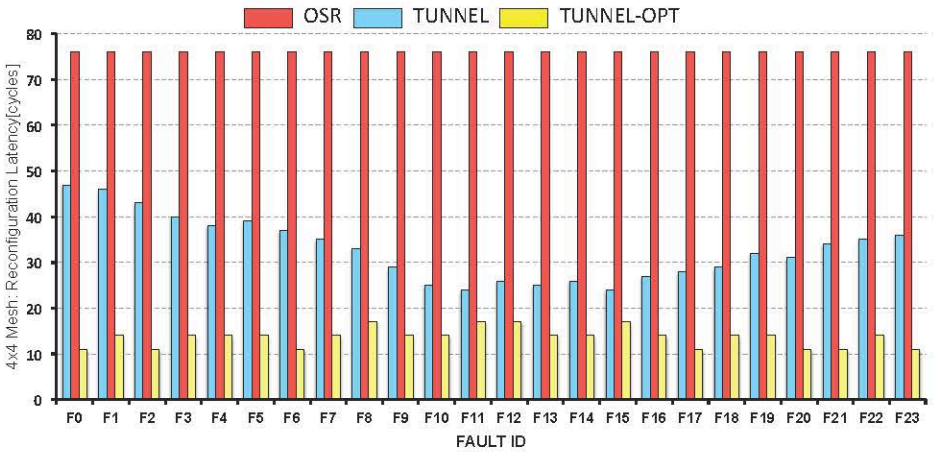


Figure 7.14: Reconfiguration transient latency of different mechanisms: in red the baseline OSR, in blue a first optimization of the tunneled version and finally, in yellow, the best optimization of the tunneled OSR.

As Figure 7.14 shows, considering different fault IDS, results are the same of Chapter 6, thus proving that the SystemC cycle-accurate RTL simulator used is very good to model real hardware, although the simulation times are very long.

It is clear, observing the figure, that the optimized T-ORS provides to the system enormous speedups, from the 40% of the worst case up to 65% in the best case. The graph also depicts the distributed nature of the mechanism: reconfiguration transients duration depends on the position on the network of the faults, while in the baseline OSR, that is a global mechanism, that presents the same value for each fault position.

7.6.4 Impact on main network traffic

Finally I complete the system inserting test-pattern generators, i.e. traffic injectors, to inject in the system pseudo-random and uniform traffic, to evaluate the impact on the main network of the implemented mechanism.

Figures 7.15, 7.16, 7.17 show the latency of the packets to reach the destination, considering testing windows of 500 cycles for each link, and specified for each link. On the x-axis there is also information about the switches testing order, given by the testing token propagation path shown before.

In Figure 7.15 I consider an injection rate of 40% on the secondary network (thus the 60% of the traffic in injected in the main network). I can notice that:

- The minimum latency is constant for both the 2 systems, because there is always a communication between a master and a slave on the same node.

- Variations of the maximum latency can be positive or negative (+ or - 10%) due respectevely to uLBDR routing, that in presence of some faults routes the packets on sub-optimal and longer paths, and to the de-congestion of some paths, that means having less traffic because it is tunneled on the escape network consequently speeding up the "journey" to the destination of some lucky packets.

- The average latency is more or less the same (+ or - 3%) for the system with or without runtime testing. This effects is due to the phenomenons depicted in the prevoius points: de-congestion of some paths and longer available paths usually mediate between each other.

- Trends are position-dependent due to LBDR algorithm that has an intrinsic priority of the output to be chosen when a packet needs to move to a particular quadrant.

To complete this type of study Figures 7.16 and 7.17 present results considering both 100% and 40% injection rate on the main network. The former presents always a worst latency of the packets because the network is congested and the brief benefits of opening tunnels are covered by having longer routing paths. The latter instead present a little improvement of latency because in the secondary network there is not a lot of traffic, so the de-congestion effect of the main network is less negligible.

If I consider the testing of the all links in the whole network (around 12000 cycles being the amount of links to be tested 24), instead, as shown in Figures 7.18 and 7.19, the effect that triumphs is de de-congestion of the network (relying on the fact that the secondary network is not congested): in this case the total speedups reach 33% of improvement, considering the best case and an injection rate of 100% on the main network.

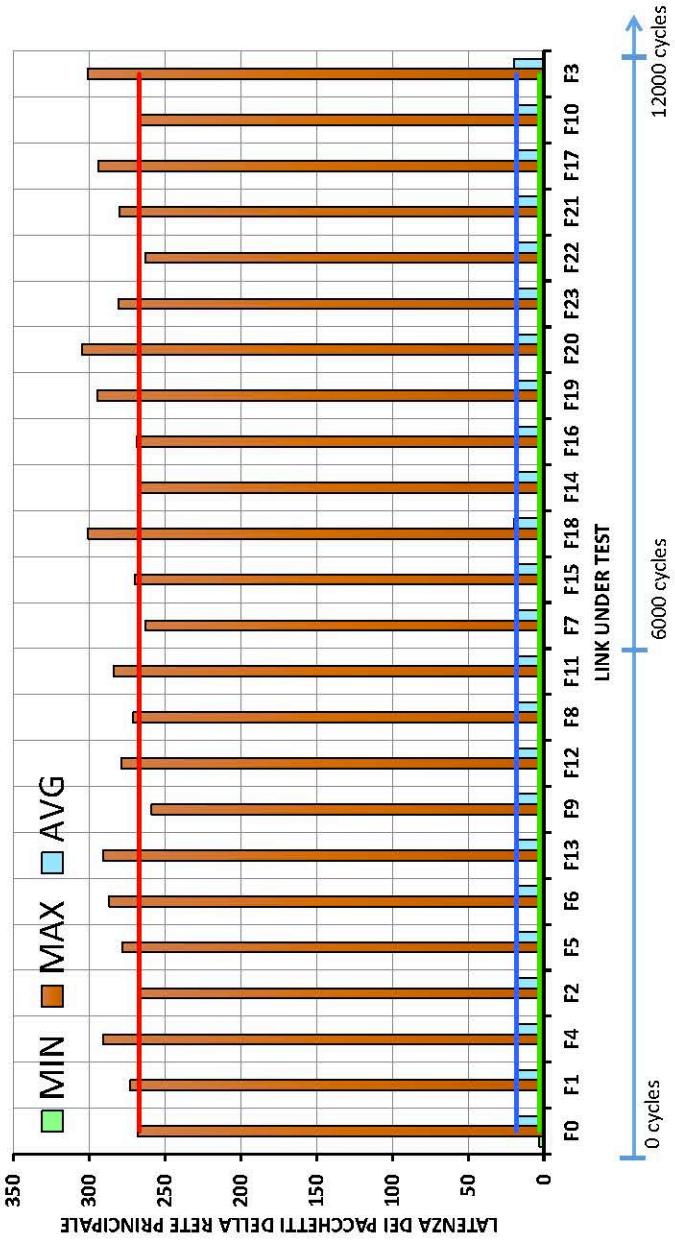


Figure 7.15: Minimum, average and maximum rrival latency of packets to destination considering an injection rate on the main network of 60% specified for links under test. Horizontal lines are the references (min is green, max is red and avg is blue) concerning the system without the runtime testing mechanism, without faulty or unconnected links. On x-axis the simulation time is reported.

7.7 Summary

In this chapter I focused on prototyping the mechanism developed in Chapter 6 on an FPGA board. First of all, I presented the baseline system on top of which I built the augmented one enhanced by new features that enable the optimized mechanism of distributed reconfiguration of the NoC routing function. I explained the fundamental steps to port the old system code in Vivado environment tool, that provides a better place & route and synthesis algorithm, and enables me to gain enough space on the FPGA to implement the whole system. I evaluated this latter in terms of area overhead

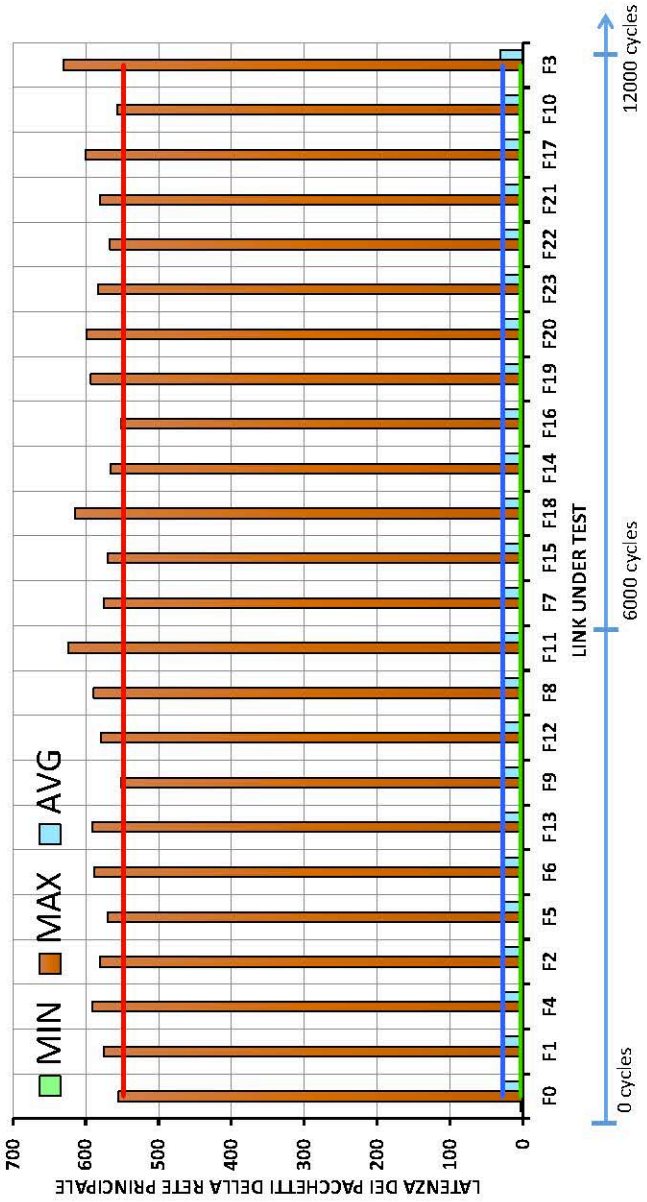


Figure 7.16: Minimum, average and maximum rrival latency of packets to destination considering an injection rate on the main network of 100% specified for links under test. Horizontal lines are the references (min is green, max is red and avg is blue) concerning the system without the runtime testing mechanism, without faulty or unconnected links. On x-axis the simulation time is reported.

(acceptable considering the whole system) and critical path (that does not change). Finally I provided a real use case for the technology developed, paving the way for real-time testing of many-core architectures. In conclusion, I asserted that everything works fine on the board, focusing in particular on the transient of the reconfiguration latency and on the impact over ongoing traffic in the main network, which is positive especially with less traffic on the secondary network, since the main network is de-congested.

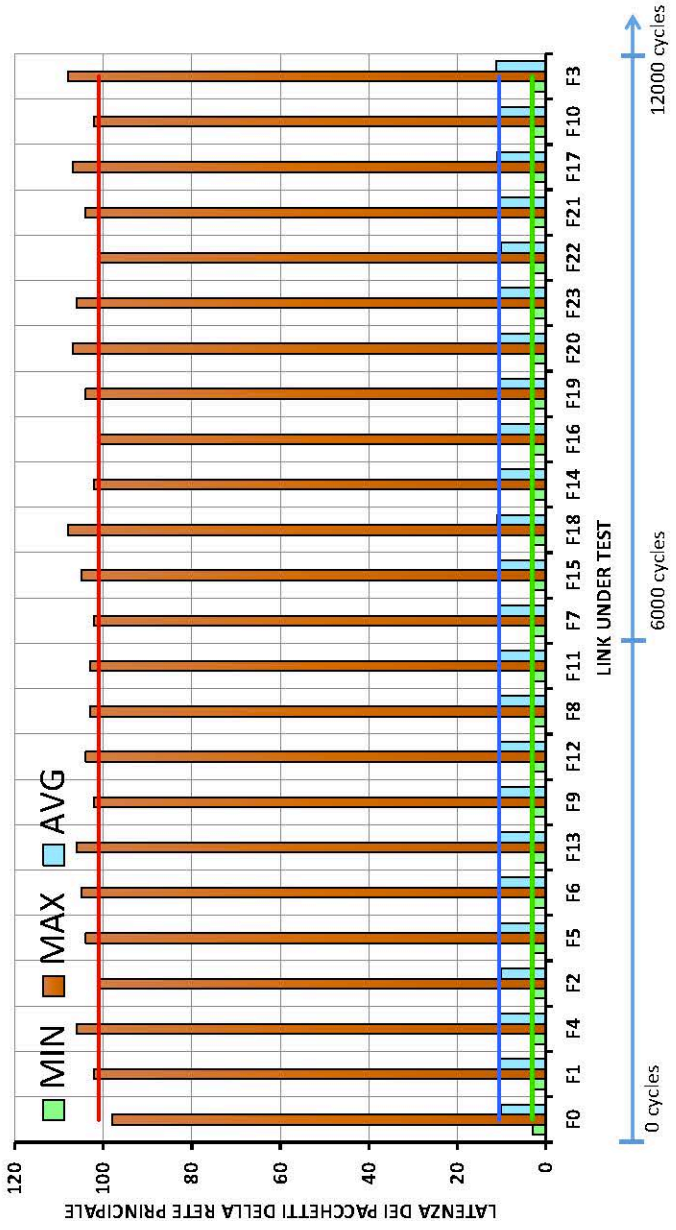


Figure 7.17: Minimum, average and maximum rrival latency of packets to destination considering an injection rate on the main network of 40% specified for links under test. Horizontal lines are the references (min is green, max is red and avg is blue) concerning the system without the runtime testing mechanism, without faulty or unconnected links. On x-axis the simulation time is reported.

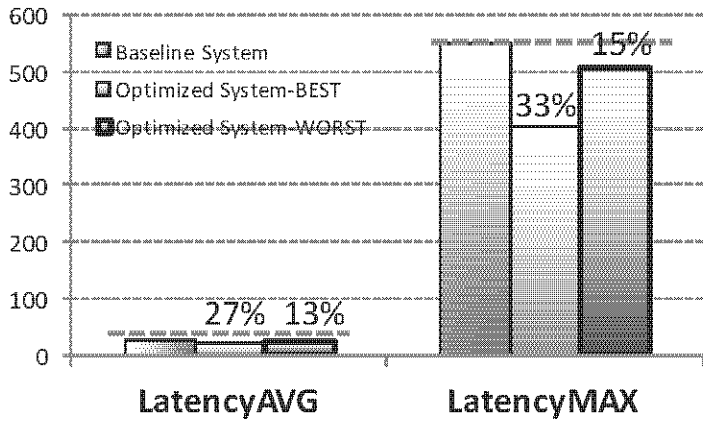


Figure 7.18: Focus on the avg and max latency of arrival of the packets in the main network considering an injection rate of 100% on it.

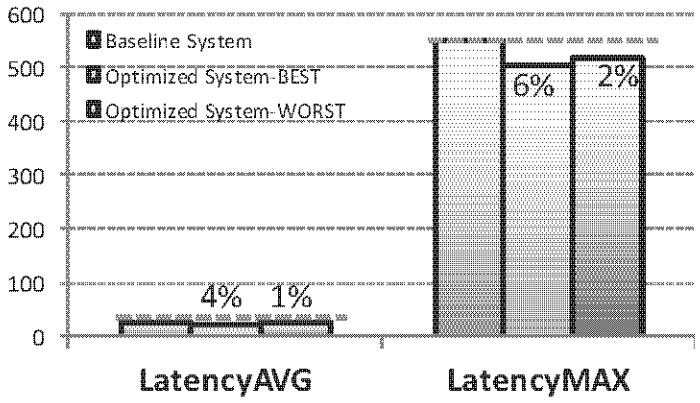


Figure 7.19: Focus on the avg and max latency of arrival of the packets in the main network considering an injection rate of 40% on it.

Chapter 8

Optically-Enabled GPPA

In this chapter I focused on emerging technologies for the on-chip network. First of all, I introduced a study that wants to prove that evolutionary technologies (e.g., asynchronous GALS systems) or revolutionary ones (optical links and networks) can replace the baseline synchronous electrical wires in the future, thus solving the issue about delivering the communication performance required by modern SoCs, within tight power budgets. Then, focusing on the silicon photonic technology, I present a validation of the potentials of Optical NoCs (ONoCs), by presenting the first hybrid GPPA augmented with photonic interconnects. Finally, I bring this new platform within reach of the SDM resource sharing paradigm (the core of this thesis) by proposing a strategy to mitigate the laser power overhead. THe basic idea consists of switching off unused laser sources, and to reuse activated sources as much as possible across computation partitions. Last but not least, as as proof of concept, I envision a whole heterogeneous parallel system, including the programmable manycore accelerator but also the host and main memory, interconnected by a photonic interconnection fabric.

Key novelty: first GPPA augmented with photonic interconnects, and validation of te abatement of NUMA effects. SDM on top of photonically-integrated many-core accelerators.

8.1 Optical NoCs: do they make sense?

Networks-on-chip (NoCs) are today at the core of multi- and many-core systems, acting as the system-level integration framework. In order to support scaling to future device generations, NoCs will struggle to deliver the required communication performance within tight power budgets. In this respect, evolutionary as well as revolutionary interconnect technologies are currently being considered. On one hand, clock-less handshaking materializes GALS systems that completely remove the system clock while reducing idle power to only the leakage power. On the other hand, the technology plat-

form could be changed, by replacing electrical wires with optical links and networks. This work provides a comprehensive power analysis of the two technologies under test on a path-by-path basis, by comparing them with each other and with a baseline synchronous NoC. The outcome of this work can support the selection of interconnect solutions for future many-core systems where power is the primary concern, as well as the runtime selection policy of routing paths in the context of hybrid interconnect fabrics.

8.1.1 Introduction

Multi- and many-core processing architectures typically comprise a number of identical tiles, each one having computational capabilities, private and shared/distributed cache resources [117, 95]. Tiles are tied up with an on-chip interconnection network (NoC) that provides the communication and synchronization backbone for parallel application threads [101, 86]. This implies a central role of NoCs as the system integration framework. In particular, as the system size scales up and communication bandwidth requirements force the system interconnect to operate in the multi-GHz domain, NoCs will need significant research investment to overcome their limits in providing the expected performance within a reasonable power envelope [72].

This concern is paired with another challenge: chip-wide distribution of a global clock is becoming a major design bottleneck, if at all feasible. While NoCs suffer from the same concern, they also have to deal with the connectivity issue of multiple voltage and frequency islands (VFIs)[103]. At this point the question arises: which interconnect technology will be able to cope with both the power and the synchronization challenges in future multi-core systems? Currently, two options can be envisioned as long-term solutions.

On one hand, clockless handshaking is at the core of asynchronous NoCs, which build up the communication infrastructure of globally asynchronous and locally synchronous (GALS)[25, 123] systems. This technology is on the evolution path of electronic interconnect fabrics, and has recently become more appealing as bundled-data asynchronous NoCs [124, 55] are proving capable of relieving the area and power overhead of traditional Quasi-Delay Insensitive (QDI) implementations, at the cost of exposing a few timing constraints to the synthesis flow. However, practical viability of asynchronous interconnect technology is still jeopardized by a long-lasting and still unsolved issue: the lack of mature CAD-tools supporting design automation. This is not only an issue of design productivity and flexibility, but also of suitable design optimization.

On the other hand, I can take a more revolutionary approach by changing the technology substrate, and opting for silicon nanophotonic links and networks [77]. On-chip photonics holds promise for low-latency, bitrate-transparent, weakly distance-sensitive and bandwidth-rich communication [137, 12]. This has a number of architectural impli-

cations as well, such as the abatement of NUMA effects. However, translating the raw features of optical networks into actual system-level performance and power benefits is non-trivial, since they revolve around heterogeneous technology integration. First, they are highly sensitive to static power (*i,e., due to laser sources and thermal tuning*). Second, the electronic-photonic interface is a main source of overhead not only at the circuit level for domain conversion, but also at the architecture level. Third, feasibility of this emerging interconnect technology depends not only on the maturity of optical components, but essentially on the possibility to amortize the large step in manufacturing cost throughout large volumes.

Some decision variables on the most suitable interconnect for future multi-core systems go beyond the visibility researchers can have right now, such as proper CAD tool availability for clockless electronic design, maturity of silicon photonic devices and cost per bit for optical links. Essentially, addressing most of these issues depends on whether the proper effort will be allocated to solve them, depending on a priori choice on which direction to pursue with highest priority. Therefore, research should help to drive the investment of effort and resources toward the most appealing solution, given the requirements of the connectivity problem at hand. This is the challenge this work takes on.

We aim to compare and provide an evaluation of three different on-chip interconnect technologies for future power-efficient multi/many-core systems:

- current (electronic synchronous NoCs),
- evolutionary (electronic asynchronous NoCs),
- revolutionary (optical NoCs).

To our knowledge, literature works contrast synchronous vs. optical NoCs [9] and synchronous vs. asynchronous interconnect fabrics [141], but a comprehensive overlook like the one targeted by this work is currently missing.

More specifically, this work investigates whether a generic core-level communication flow, currently mapped onto a synchronous link in an electronic 2D mesh topology, can be more efficiently mapped onto an asynchronous link of a clockless NoC or onto an optical path of a wavelength-routed optical NoC topology. As a result, I deliver break-even points as a function of static power, inter-core distance, and bandwidth requirement of the communication flow at hand.

8.1.2 Target architecture

Our target architecture relies on a multi-core programmable accelerator composed by 16 processing Tiles. Each of them operates as both initiator and target for communications over the system interconnect. Tiles are disposed over the die area following a grid structure, and are assumed to be 2mm$x$2mm.

We consider that at a given point in time the task graph of an application needs to be mapped onto the system, targeting the accelerator as a whole, or a spatial partition derived in it by the runtime manager. The graph is annotated with communication bandwidth requirements for each inter-task communication flow. The research question is: *which interconnect technology can accommodate a specific communication flow so to minimize communication total power?* The answer depends on parameters such as distance of the cores where end tasks are mapped, and communication bandwidth requirements, in addition to the different nature of the interconnect fabrics under test.

We thus assume three alternative variants of the system interconnect. The Electronic solutions consist of synchronous and asynchronous variants of the same NoC switch architecture. They are both laid out as 2D-mesh topologies, using an industrial low-power 40nm technology library. When it comes to the optical switching fabric, I assume a 16x16 λ-router [102, 119] (see Figure.8.1), which is a wavelength-routed topology delivering contention-free all-to-all connectivity. This optical network is laid out manually based on the guidelines from [82]. Special emphasis was given to modeling waveguide crossings arising when the interconnection network itself is overlapped with the **Power-Distribution Network** (PDN). The latter one is in fact needed to bring the multiplexed optical carriers from off-chip continuous-wave laser sources to the optical modulators contained in the optical network interfaces, which follow the baseline grid-based positioning. The optical NoC is assumed to be vertically stacked on top of the processing layer.

Our analysis takes a path-oriented approach, that is, I investigate power efficiency of routing paths allocated in the different networks under test to host a given communication flow. I rely on the assumption that all components of all interconnect fabrics are power-gated by default. Therefore, there is no consumed power without network activity. When a communication flow needs to be mapped, I assume the activation of only the components that are on the selected routing path. For the electronic NoCs, activating a routing path implies the activation of the network interfaces of the end cores, and of the switches along the routing path, assuming that power gating granularity is that of the NoC switch. Similarly, for the optical NoC I activate the electronic as well as the optical components that build up the transmission side, the wavelength-routed optical path, and the receiver side. Clearly, upon activation of a routing path, an infrastructure cost arises in all networks that can be amortized for other communication flows. That is, activated switches can be used to map other communication flows on unused ports, thus minimizing the number of activated switches to map new flows. Similarly, once a laser source is turn on to feed a specific routing path, the mapping engine should consider that other optical paths are available in the topology that are powered by the same laser source. Our experimental results will quantify the benefit of this resource sharing approach in all kinds of networks.

Last but not least, links in all interconnect technologies have matched bandwidths of

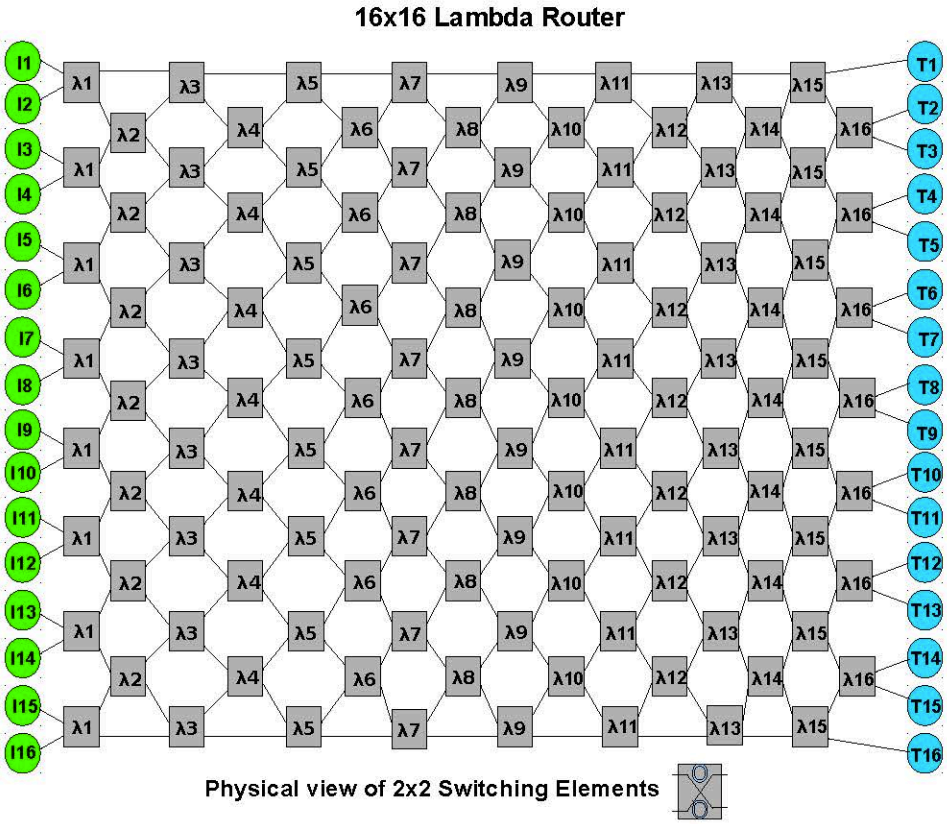


Figure 8.1: 16x16 Lambda router logic scheme.

roughly 30 Gbit/sec. As a side effect, the aggregated topology bandwidths are different, since contrarily to the electronic networks, the optical one uses wavelength-routing to deliver contention-free communication. This means that whenever the decision will be taken to map a communication flow to the optical medium, its performance will be higher by construction. Therefore, this work can focus on power efficiency of allocated communication paths in revolutionary interconnect solutions without expecting any subtle performance trade-off. Clearly, this choice is a worst case for the optical medium, since it incurs a higher infrastructure cost to support wavelength-routing.

8.1.3 Baseline synchronous design

The switch architecture is inspired by the xpipesLite architecture, which represents an ultra-low complexity design point in the space of electronic NoCs [126]. Our instance implements XY algorithmic routing, wormhole switching and a 32-bit flit width. The input buffer is set to the minimum dimension (*two slots for flow control requirements*), while the output buffer dimension is set to six slots. In this design, one clock cycle is taken to traverse the switch and one clock cycle to traverse the link connecting two switches. Design convergence was achieved at 950 MHz. In order to preserve the generality of the design and support cores with different operating frequencies that access a fixed-frequency NoC, dual-clock FIFOs are included at the network interfaces.

8.1.4 Asynchronous design

Among the existing asynchronous design styles, QDI solutions are robust and reliable, however show limited benefits in terms of area and power. On the other hand, asynchronous bundled-data designs offer better performance, area and power at the cost of additional engineering efforts: in fact, relative timing constraints must be enforced in order to avoid timing hazards and ensure a correct circuit operation. Since low-power communication is our primary concern, I revert to the most power-efficient design style by considering a 2-phase bundled-data switch design [55]. This latter is the exact asynchronous counterpart of the synchronous xpipesLite switch [126], that ensuring architectural homogeneity. Similarly to the synchronous architecture, the input buffer is minimal (*a single asynchronous pipeline stage*) while the output buffer dimension is set to *six slots* as in synchronous switch. XY-routing and wormhole switching are also considered. On average, payload flits are switched in and out at 950-to-1GHz equivalent operation speed, depending on the input-to-output port connection. The key difference with respect to the synch. switch consists of the interface components, which in this case consist of synchronous-to-2-phase bundled-data converters (inspired by [123]) and of their dual converters at the receiver side.

8.1.5 Optical design

The proposed micro-architecture of the optical link is shown in Fig. 8.2. As can be seen, our design does not only include Electro-Optical (E/O) and Opto/Electrical (O/E) conversion circuits, (*i.e., driver and ring modulator at the transmission side, optical receivers at the opposite side*) but also incorporates **serializers (SERs) and deserializers (DESERs)**, **source synchronizing circuits**, **flow control blocks**, as well as **dual-clock FIFOs**. For this reason, after flits are forwarded to the appropriate path depending on their destination, they need to be converted into a 10 Gbit/sec optical bitstream for modulation. The number of serializers is defined based on the optical bit parallelism. For the sake of link bandwidth matching with electronic interconnect solutions, 3 serializers of 11 bits (working in parallel) are necessary to serialize the 32 bits of a given fit, and to guarantee a matched communication bandwidth of roughly 30 Gbps. The reception side is symmetric.
Another key issue to be addressed in the micro-architecture is the **re-synchronization** of received optical pulses with the clock signal of the electronic receiver. This work assumes **source-synchronous communication**, which implies that each point-to-point communication requires a strobe signal to be transmitted along with the data on a different wavelength, and utilized to correctly sample data in the receiver domain. Such strobe signal is generated starting from the electrical clock of the transmitter, and removes the need for phase-locked loops (PLLs) or delay-locked loops (DLLs) at the receiver. We assume that clock gating is implemented on the source-synchronizing

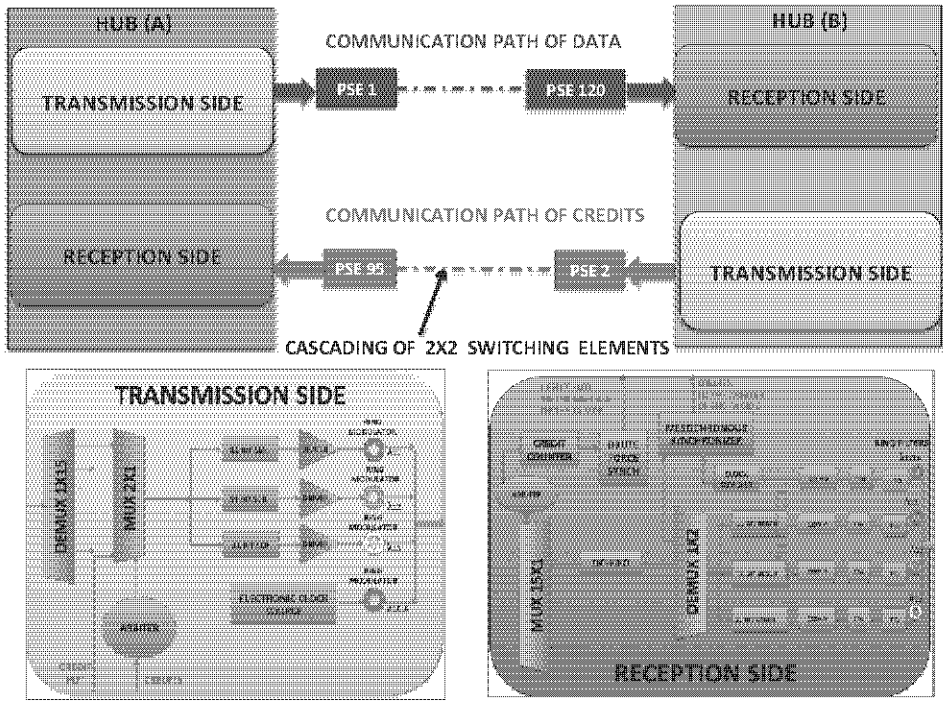


Figure 8.2: Proposed micro-architectural view of the optical link.

signal, therefore when no data is transmitted, the optical clock signal is gated.
Another typically overlooked issue consists of the **backpressure mechanism**. We consider **credit-based flow control**, because credit tokens can reuse the existing communication paths for data, and exploit the low dynamic power of optical networks. As depicted on top of Fig. 8.2, the communication link must be bidirectional. In fact, in addition to sending data to any other Hub, the receiver Hub must transmit credits to its associated transmitter.

8.1.6 Energy and power modeling

This section presents the energy and power modeling on which our work is based. For both synchronous and asynchronous NoCs, a small subsystem consisting of two 5-ported switches and their inter-switch link is synthesized, placed and routed by using a Low-Power 40nm industrial technology library. Given the regularity of a 2D mesh, *the energy to go across multi-hop routing paths* was extrapolated from the synthesized subsystem. Also, the contribution of network interface components at the communicating peers was included, except for the (de-)packetizers, which are common to all interconnect solutions. For the asynchronous NoC I used our own asynchronous synthesis flow: design is first manually constrained for max-delay, and then min-delays are enforced in case of possible timing violation using an iterative procedure (for more details please refer to [55]). Power metrics are calculated by back-annotating the switching activity of block internal nets, under the assumption of uniform continuous traffic, and in turn importing waveforms in the Synopsys PrimeTime Tool. In synchronous designs, clock gating is applied for the sake of realistic measurement of static power, whereas

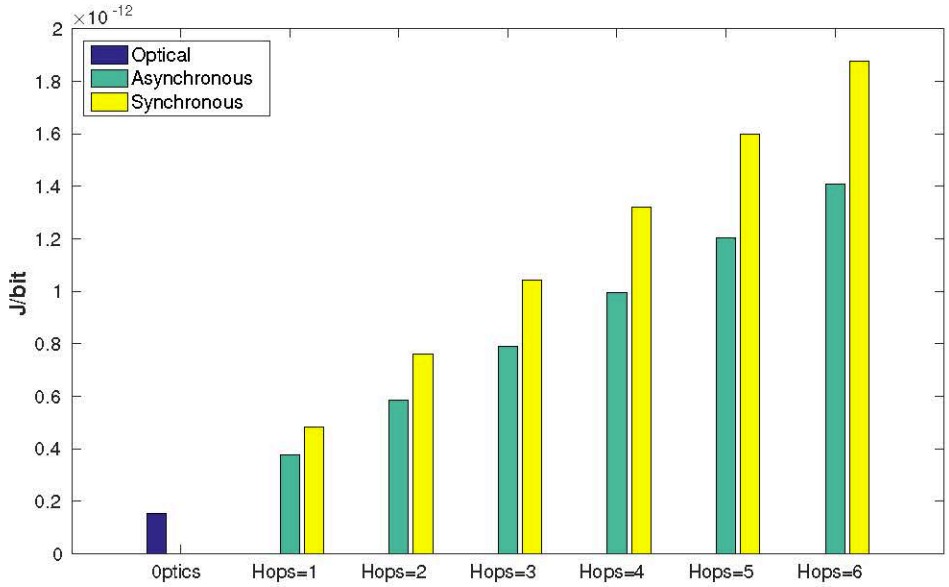


Figure 8.3: Contrasting Energy-per-bit: Optical vs. Synchronous vs. Asynchronous designs.

the energy-per-bit is obtained by removing the static power from the total power on a component-basis.

The power modeling of a *point-to-point optical connection* is given by the static power values consumed by: laser sources, thermal tuners, transmitters (i.e., the driver-ring modulator couple), receivers (i.e., photodetectors, trans-impedance amplifiers, and comparators), and the source-synchronous clock. The number of thermal tuners depends on the number of 2x2 switching elements that are involved along a specific optical path (*in the 16x16 λ-router I may have overall 32 Ring Resonators (MRRs): 30 in the interconnection network, one for modulation and another one as a filter to eject the optical signal before entering the photodetector.*) While the number of these components must be replicated as many times as the level of bit parallelism (**supposed to be 3 in our study**), the clock contribution is instead composed of one laser source, one transmitter, one receiver, and two thermal tuners as well. Besides, the power dissipation of Optical NIs is added. It consists of the power consumption of the sequence: Demultiplexer, Multiplexer, SERs, at the transmission side, followed by DESERs, DC-FIFO, Multiplexer as well as synchronizers and a credit counter at the receiver side. Static power and dynamic energy parameters, as well as their relative ratio, are consistently selected from the same literature source [12]. Finally, [104] report the energy/power values that I assumed for all basic blocks of our architecture.

8.1.7 Energy-per-bit analysis

Energy-per-bit is reported in Fig.8.3. Since it is rather insensitive to the propagation distance, the optical interconnect fabric turns out to be the most energy efficient solution when moving a bit across a complete optical path. The gap with respect to

the electronic solutions widens when these latter have to switch information across a multi-hop routing path. The total energy-per-bit of the optical point-to-point communication is given by 3 important contributions: one due to data propagation $Edata$, one due to flow control signaling Efc, and another one due to the clock domain synchronization $Eclk$. Data bits should cross multiplexers, serializers, optical transmitters and receivers, deserializers, demultiplexers, and a dual-clock FIFO for an end-to-end transmission. A similar path is taken by credit flits on the way back from the receiver to the sender, including a mesochronous synchronizer for clock consistency in the network interface. Clearly, a data flit is always associated with a corresponding credit flit propagating backwards. Finally, I consider that a transition of the source-synchronous clock is incurred every three data bits transmitted. As opposed to the optical communication, the electronic synchronous and asynchronous paths consume more energy due to the inherently increased number of hops in Mesh topologies (i.e., because there are many switches to traverse for reaching the desired destination). Synchronous solutions consume more energy compared with the asynchronous counterpart because they need an additional buffer stage at the switch input port, while the asynchronous switch uses a single MOUSETRAP pipeline stage when minimum buffering is required. With 1-hop communications in electronics, the optical link consumes **x2.5** less energy than the asynchronous counterpart. At 6 hops the asynchronous communication consumes **x9.3** more energy with respect to its optical counterpart. Synchronous communication consumes on average **30**% more energy with respect to asynchronous technology.

As opposed to the optical communication, the electronic synchronous and asynchronous paths consume more energy due to the inherently increased number of hops in Mesh topologies (i.e., because there are many switches to traverse for reaching the desired destination). Synchronous solutions consume more energy compared with the asynchronous counterpart because they need an additional buffer stage at the switch input port, while the asynchronous switch uses a single MOUSETRAP pipeline stage when minimum buffering is required. With 1-hop communications in electronics, the optical link consumes **x2.5** less energy than the asynchronous counterpart. At 6 hops the asynchronous communication consumes **x9.3** more energy with respect to its optical counterpart. Synchronous communication consumes on average **30**% more energy with respect to asynchronous technology.

8.1.8 Static power assessment

Fig.8.4 shows the static power (including its breakdown) incurred by each design to sustain a single communication flow. The flow is assumed to be mapped to 1-hop routing paths in electronic NoCs (2 network interfaces plus 2 switches plus 1 link). For the optical link, the distance does not really matter. What determines the static power is mainly the laser power used to feed a specific optical routing path. In fact, I measured the worst-case laser power requirements for each laser source in the topology, which in

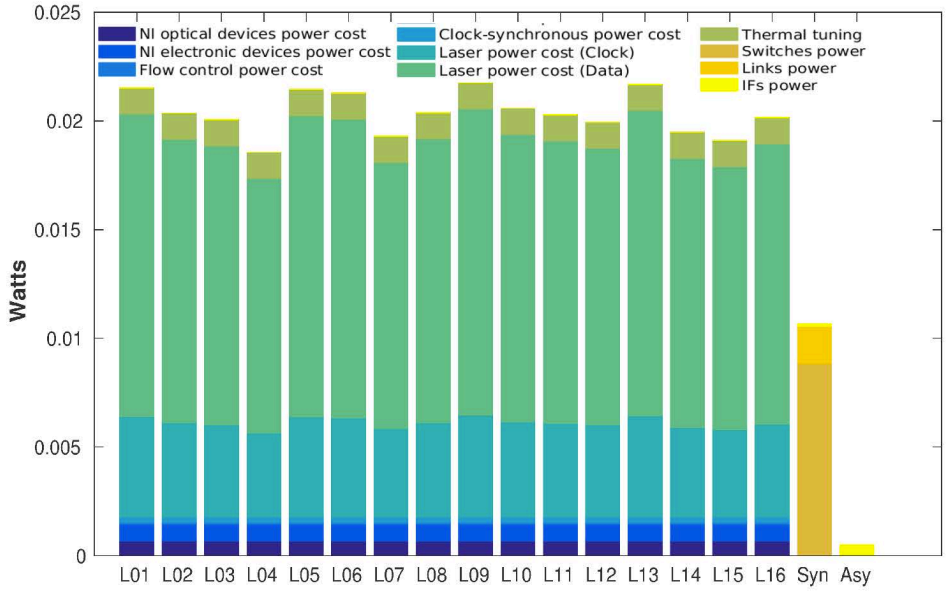


Figure 8.4: Static power breakdown to sustain single communication flows, mapped 1-hop away in electronics.

turn depends on the worst-case insertion loss across all optical paths (which is quite sensitive to the crossings waveguides caused by the overlapping of the interconnect with the PDN). Since laser power is tuned on the worst-case for each wavelength, what makes the difference is the strength of the laser source that powers the optical path on top of which a specific communication flow gets mapped. Fig.8.4 then illustrates all possible static power values, depending on the target wavelength associated with the communication flow under test. The Asynchronous design represents the most power efficient solution because there is no clock distribution network. Asynchronous switches practically exploit an ideal scenario where ideal clock gating is applied: the circuit consumes only leakage power in idle state and it is automatically woken up whenever a data must be computed.

The breakdown in Fig.8.4 shows that the asynchronous static power is dominated by the contribution of Sync-to-Async/ Async-to-Sync interfaces, because applying clock gating to such an hybrid interface is not trivial, therefore the interface power consumption is in some way tainted by the clock even in idle state.
In synchronous designs, interfaces consumes less power since clock gating is successfully applied. However also the switch internal circuitry must be gated, resulting in a much complex clock distribution network and a higher static power contribution. As opposed to the dynamic energy, the non-trivial fixed power overhead of nanophotonic devices cannot be as effectively amortized. As can be seen in Fig.8.4, the static power cost of the optical link is dominated by the laser power. Most of this contribution is tightly dependent on the support of the adopted bit parallelism and on the need for clock synchronization.

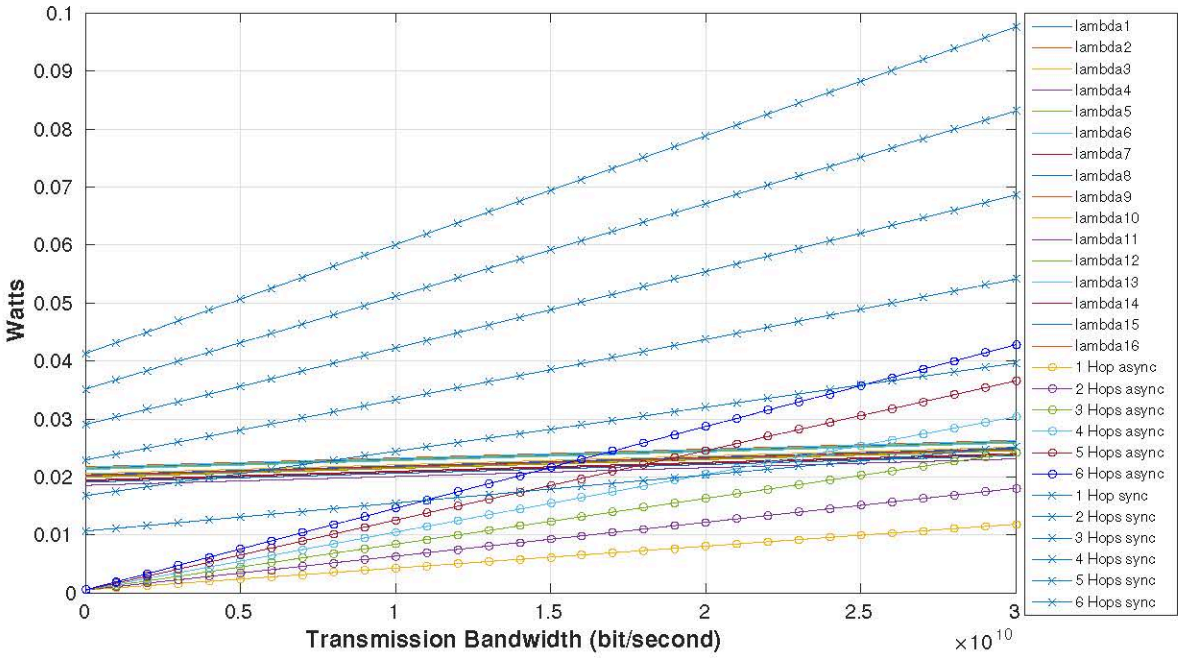


Figure 8.5: Break-even bandwidth for power efficiency with no laser source reuse.

Ultimately, the optical technology is strongly static-power dominated, consuming about **x2** more than synchronous paths, whereas **x42** more than the asynchronous counterpart. Clearly, this gap should be bridged by a significant reduction of dynamic power and by the tentative mapping of multiple communication flows to the same laser source.
In fact, this latter powers different source-to-destination connections in a wavelength-routed topology (16 in this case, out of the 256 possible connections the topology delivers).

8.1.9 Power vs. communication-bandwidth requirements

The above sections show that electronic networks have a lower static power (**Ps**) cost but a higher dynamic energy per bit (**Ed**) when compared to the optical interconnection. Giving these parameters the designer should choose the most appropriate communication architecture, considering the total power consumption:

$$\mathbf{Ptotal} = \mathbf{Ps} + \mathbf{Ed} \star \mathbf{Bandwidth}$$

It is well known from the literature that optical communications work better over long distance and higher transmission bandwidths, while asynchronous fabrics take advantage of idleness. The exact trade-off for on-chip networks is hereafter quantified. In general, it exists a bandwidth requirement **B** on an electronic path which generates a switching activity high enough to match the power consumption of the optical path, effectively counterbalancing its static power overhead: In the following plots, I will be searching for such break-even bandwidths for power efficiency.

Fig. 8.5 shows the total power contrasting: Optical vs. Asynchcronous vs. Synchronous communication paths with an increasing transmission bandwidth. All possible mapping options of a communication flow to the reference topologies are reflected by the parametric curves in the plot. In optics, communication may go across any of the 16 available paths for each wavelength channel. In electronics, total power is referred to the target number of hops in the 2D mesh. Since optical technology is static power dominated, its corresponding design points (i.e., the 16 wavelengths shown in the figure) show a smooth slope because of the lower energy-per-bit (dynamic energy is also constant among wavelength paths). The picture is completely different when considering synchronous and asynchronous technologies. Their total power in fact increases more rapidly with the hop count, hence resulting in higher slopes in the figure. The plot in Fig. 8.5 can drive the designer to choose where to map a communication flow: to an available n-way electronic path vs. to an available wavelength channel in the optical medium.

Interestingly, 1-hop communications in synchronous NoCs are almost always more power efficient than with any optical mapping. The contrary holds for 2-hop electronic paths. Beginning from 3 hops up, any optical path in the λ-router turns out to be more power efficient.

Asynchronous paths exploit their low initial y-axys coordinate, thus they are always more power efficient than their clocked counterparts. With respect to optics, I see interesting break-even points. With low transmission bandwidths (**lower than 17Gbps**) the aynchronous path turns out to be the most power efficient solution, whereas for higher transmission bandwidths the optical technology results competitive at least for communications that would be mapped more than 3 hops away in electronics. Finally, the clear indication is that moving away from synchronous NoCs is always desirable to meet tight power budgets.

A possible way to amortize the laser power cost of optical paths would be to reuse the same set of laser sources for multiple and parallel communication flows mapped to disjoint paths. This is a common scenario in manycore systems, since they are supposed to serve multiple communication flows at the same time. Next, I present a case study where 8 parallel communication flows are running at the same time. For the optical technology, this means that I can reuse the same laser source as much as possible at the price of an increased infrastructure cost for the multiple network interfaces that are active at the same time (i.e, more SERs, DSERs, etc ...). For the asynchronous network, this means that 8 parallel communications are happening concurrently between eight pair of cores (from now, synchronous design points are not considered anymore because they are suboptimal). Whether such communications reuse or not the same switches or every time activate new switches does not really matter, since the static power cost of the asynchronous network is in any case negligible.

Fig. 8.6 shows the total power in the above case study, contrasting Optical and

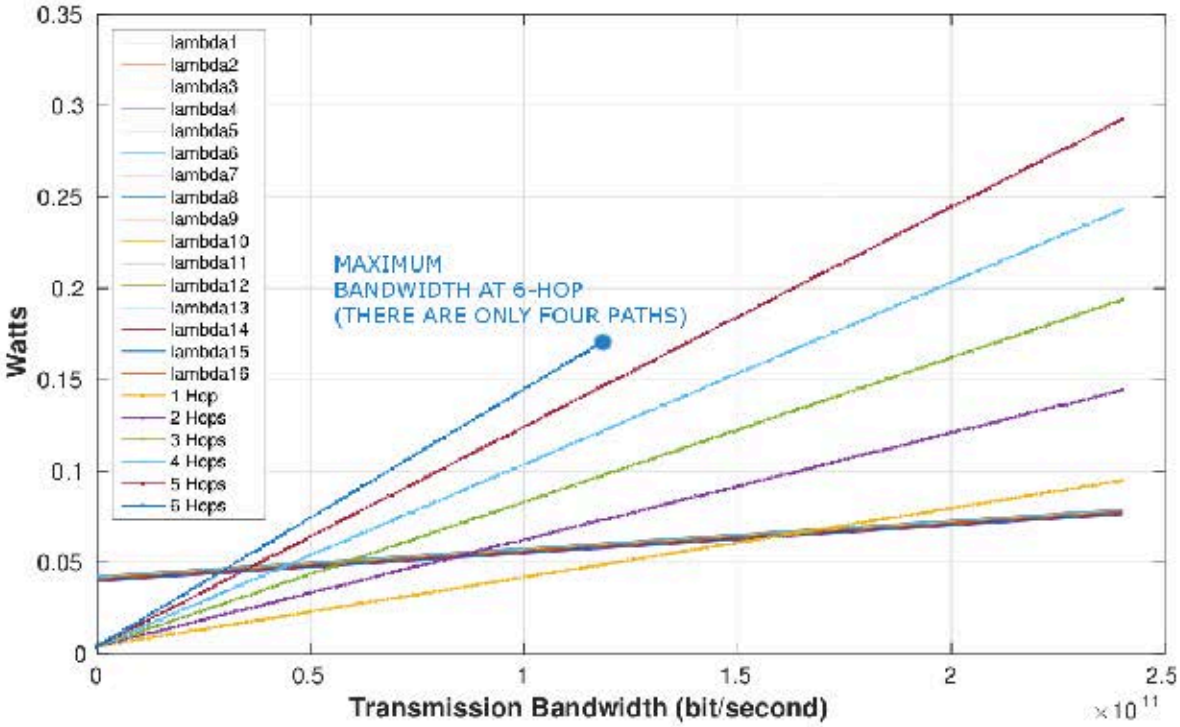


Figure 8.6: Break-even bandwidth for power efficiency with laser source reuse.

Asynchronous designs. We represented in abscissa the cumulative transmission bandwidth achieved by the eight parallel communication flows (i.e., sharing factor of 8). We derived curves also for intermediate sharing factors, which were not showed for lack of space.

Curves should be read as follows: *is it more power efficient to map 8 concurrent flows to n-way electronic hops (with n ranging from 1 to 6) or to 8 optical paths powered by the same laser source at a given wavelength?*

With respect to the previous case (i.e, sharing faction=1), I can observe that for cumulative transmission bandwidths larger than 63 Gbps even 8 3-hop paths could be more effectively realized optically. In contrast, for 8 1-hop flows, the break-even transmission bandwidth should be as large as 168 Gbps or more. Given the chosen size of the 2D mesh electronic topology and the chosen routing algorithm, only 4 paths are 6 hops long, therefore the maximum cumulative bandwidth peaks at 120 Gbps.

Summary

In this work I performed a wide-scope power-oriented analysis between evolutionary (synchronous-asynchronous) electronic interconnect technologies and revolutionary optical signaling. By reading both Fig. 8.5 and Fig. 8.6, an interesting consideration arises. Since the static power overhead of an asynchronous NoC is negligible, this sounds like an ideal condition to materialize an hybrid interconnect fabric, combining clock-less handshaking with optical switching. When the break-even bandwidth is exceeded, the optical path could be preferred, since the static power cost is offset. Clearly, the runtime selection process of the routing path on one technology vs. the

other should take into account that mapping a flow to the asynchronous fabric causes local communications, while in optics the selection of a wavelength channel does not necessarily lead to the selection to communicating peers that are spatially close to each other. However, these considerations on the mapping policy go beyond the scope of this work, which provides the quantitative numbers on top of which a mapping decision could be taken. Interestingly, it does not exist any transmission bandwidth which makes the synchronous path an acceptable solution with respect to both its evolutionary and revolutionary counterparts.

8.2 Validation of the potentials of the concept inside a system

8.2.1 Introduction

There is today consensus on the fact that optical interconnects can relieve bandwidth density concerns at integrated circuit boundaries. However, when it comes to the extension of this emerging interconnect technology to on-chip communication as well, such consensus seems to fall apart. The main reason consists of a fundamental lack of compelling cases proving the superior performance and/or energy properties yielded by devices of practical interest, when re-architected around a photonically-integrated communication fabric. This work takes its steps from the consideration that many-core computing platforms are gaining momentum in the high-end embedded computing domain in the form of general-purpose programmable accelerators. Hence, the performance and energy implications when augmenting these devices with optical interconnect technology are derived by means of an accurate benchmarking framework against an aggressively optimized electrical counterpart.

Optical interconnect technology has yielded a rich design space for on-chip communication architectures [78, 10, 106, 136, 28, 80], including the re-architecting of the DRAM memory sub-system [12], the revision of the processor-memory interface [130], or the development of new coherence protocols custom-tailored for the optical transport medium [80]. This significant amount of work has finally contributed to the foundation of cross-layer design methodologies for designing new optical networks [11].

Unfortunately, the above experimental evidence has not translated into a stabilization of roadmaps for industrial uptake of this on-chip communication technology yet. This consideration is further exacerbated by the high cost targets for introducing it, and by the far-from-consolidating maturity of basic optical components. Fundamentally, the main challenge to revert this trend consists of *showing a compelling advantage (if any) for on-chip nanophotonic interconnection networks (ONoCs) with accurate modeling assumptions, and while meeting the requirements and operating conditions of real-life user devices and workloads.*

A milestone contribution in this direction comes from [61], which aims at tackling the bandwidth and latency bottlenecks in on-chip interconnect and off-chip memory access in graphics processing units (GPUs) by means of optical links and 3D-stacked technology. *This work follows the same track, and aims at extending the feasibility analysis of optical interconnect technology when integrated into industry-relevant objects. In particular, the focus is on the high-end embedded computing domain, where photonic networks have already been proven to be promising for DRAM memory access [65]. In this work, a key component to sustain the performance-per-watt metric of embedded computing platforms is investigated as a candidate for photonic integration, namely a general-purpose manycore programmable accelerator.* In fact, driven by flexibility, performance and cost constraints of demanding modern applications, heterogeneous Systems-on-Chip (SoCs) are the dominant design paradigm in the embedded computing domain. SoC architecture and heterogeneity clearly provide a wider power/performance scaling, combining host CPUs along with massively parallel general purpose programmable accelerator (GPPA) fabrics. These latter hold potential of bridging the gap between the energy efficiency (GOPS/W) of hardwired hardware accelerators and the computational power delivered by throughput computing. In contrast to graphics processing units, applicability of optical interconnect technology to GPPAs is faced with a more balanced trade-off between latency and throughput requirements, and by a different usage model of the manycore device. The distinctive contributions of this work are as follows:

1) I re-architect the communication infrastructure and the processor-to-interconnect interface in a GPPA architecture inspired by real devices, driven by the requirements of the system at hand. To our knowledge, this is the first time insights and guidelines are given to exploit optical technology in emerging GPPAs.
2) Aware of the difficulty in making the case for purely optical interconnect fabrics, I conservatively and realistically come up with a hybrid architecture where specific kinds of transactions are selected for switching on the optical transport medium.
3) I carry out a performance characterization of system operations with the hybrid interconnect fabric, and benchmark it against a competitive electrical baseline. Our focus is not just on the performance of NoC read and write transactions, but rather on their aggregation into higher-order operations relevant for the system at hand (e.g., computation offload, instruction cache refills, explicit data memory management). As a side effect, performance of such operations is not just determined by the system interconnect, but rather by the cooperation of several components (e.g., the DRAM subsystem, DMA architecture, memory hierarchy). This work captures such interdependency.
4) The ONoC architecture is designed by following a cross-layer design methodology, where the quality metrics of the selected design point include awareness of the degra-

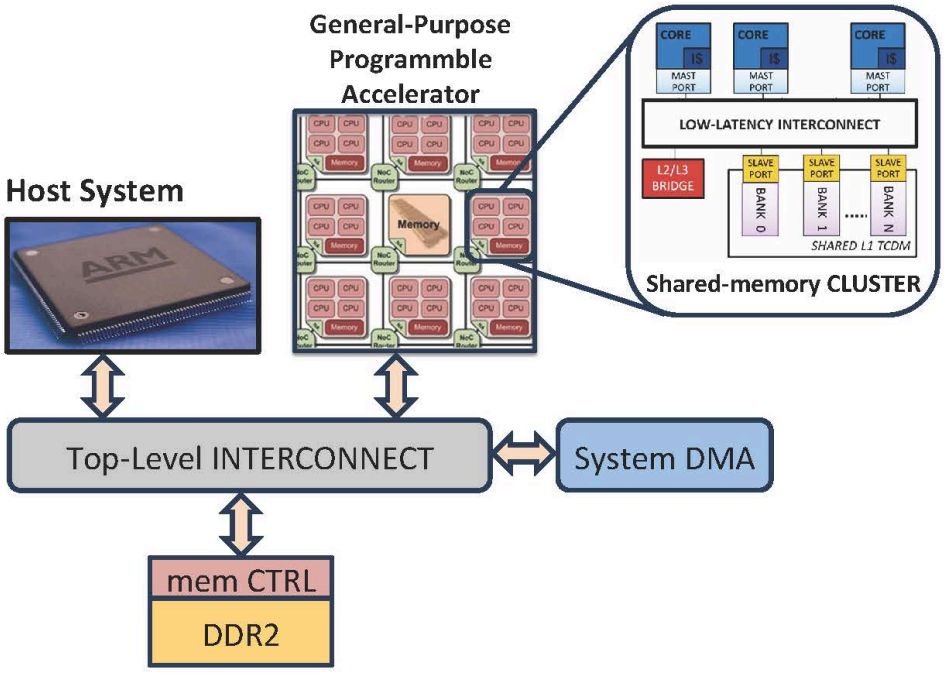


Figure 8.7: Heterogeneous (many-core accelerator-based) MPSoC architecture.

dation effect of place&route constraints over insertion loss, and the overhead of the upper layers of the optical network interface beyond the domain conversion circuits (e.g., buffering, flow control, virtual channels, synchronization).
5) Energy efficiency figures are provided by accounting for the execution time of real-life workloads, for a parametric set of quality metrics for the fast-evolving optical devices, and for the energy of electrical components on a 40nm low-power industrial technology library (which makes the electrical counterpart extremely competitive).

8.2.2 GPPA motivations

In the latest heterogeneous Systems-on-Chip (SoC), and even more in future ones, the quest for processing specialization to deliver ultra-high performance acceleration at reduced energy cost does not necessarily imply hundreds of dedicated hardware accelerators [88]. There are at least a couple of reasons against that approach. On one hand, the performance of a specialized processing engine may in many cases be equally achieved by the parallel computation of programmable processing units [45]. Execution efficiency can thus be achieved without sacrificing programmability. On the other hand, the trend towards simplifying the microarchitecture design of system building blocks is becoming increasingly strong. Only a replication-driven approach ultimately pays off in terms of design productivity.

There are two main architecture families that might in principle suit the need for many-core programmable accelerators: the former one consists of GP-GPUs [105] and is optimized for the single instruction multiple data/thread execution model (SIMD/SIMT), while the latter one relies on the multiple instruction multiple data (MIMD) model (al-

though not limited to it).

MIMD programmable accelerators do not implement GPU-like data-parallel cores, with common fetch/decode phases which imply performance loss when parallel cores execute out of lock-step mode. They are rather independent RISC cores, well suited to execute both SIMD and MIMD types of parallelism. When coupled with a hierarchical organization into clusters like [110, 95], such accelerators lend themselves to powerful programming abstractions such as nested parallelism [93].

One reason for the growing interest in many-core accelerators in the embedded computing domain is that there is a rapidly growing demand for a new type of interactions between the user and the device, based on understanding of the environment sensed in multiple manner (image, motion, sound, etc.) striving to create more friendly user interfaces (augmented reality, virtual reality, haptics, etc.). Despite the good degree of data parallelism, parallel threads in this class of applications usually expose a behavior which is heavily dependent on the local data content, resulting into many truly independent parallel computations [95]. In such a situation, GP-GPUs lose efficiency due to large divergence between threads.

The above motivations are at the core of this work's decision to investigate the potentials of optical interconnect technology in the context of flexible MIMD/SIMD General-Purpose Programmable Accelerators for the high-end embedded computing domain.

8.2.3 Target architecture

A common embodiment of architectural heterogeneity is a template where a powerful general-purpose processor (usually called the *host*), is coupled to a general-purpose programmable many-core accelerator (GPPA) composed of several tens of simple processors, where critical computation kernels of an application can be offloaded to improve overall performance/watt [95, 70, 69, 71]. Figure 8.7 shows a block diagram of such a system. The focus of this work is on GPPA many-core design, which I describe in details in the following subsections.

Cluster and memory architecture

The GPPA is a cluster-based many-core computing system. Clusters are the central building block of several recent many-cores [75] [110] [95]. These processors consider a hierarchical design, where simple processing units (PU) are grouped into small-medium sized subsystems (the *clusters*) sharing high-performance local interconnect and L1 data memory. Scaling to larger system sizes is enabled by replicating clusters and interconnecting them with a scalable medium like a network-on-chip (NoC). The simplified block diagram of the target cluster is shown in the rightmost part of Figure 8.7. It contains several simple RISC32 processor cores (typically up to 16), each featuring a private instruction cache. Processors communicate through a multi-banked,

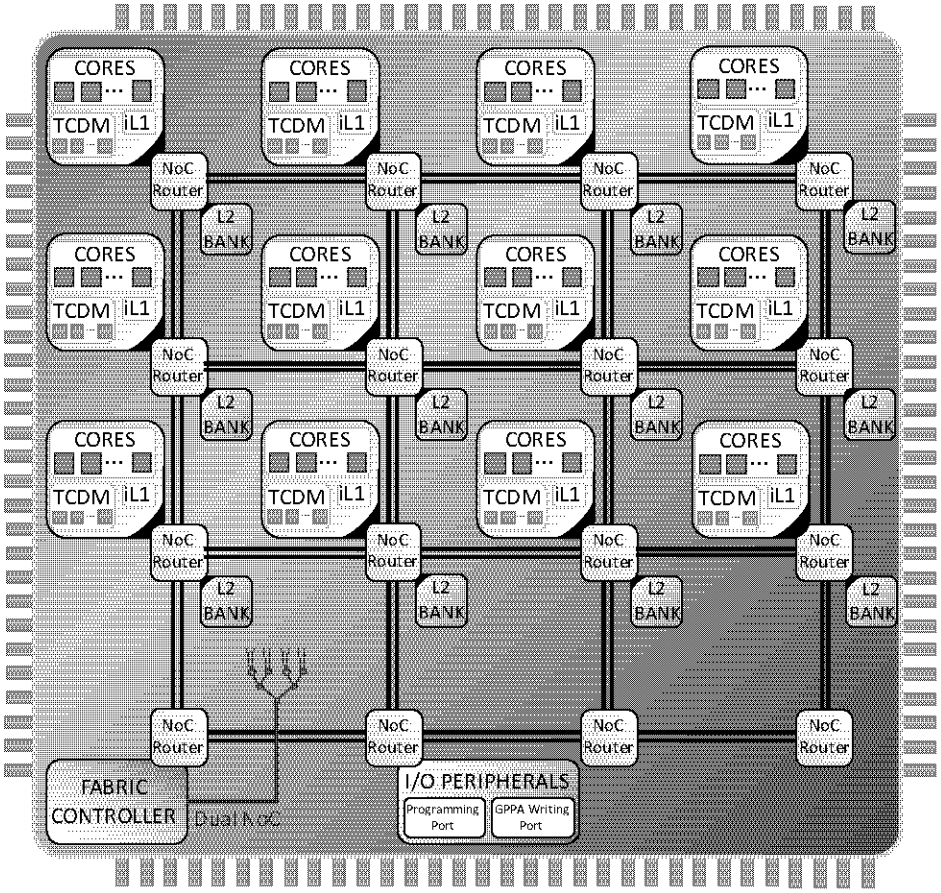


Figure 8.8: GPPA Architecture.

multi-ported Tightly-Coupled Data Memory (TCDM). This shared L1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth data interconnect. This is a very common design choice for constrained embedded many-cores, as the area and power overheads of hardware-managed caches (as compared to scratchpads) is very significant, and coherency protocols encounter severe scalability issues when interconnecting a large number of nodes.

Figure 8.8 depicts the global GPPA architecture. It consists of a configurable number of computing clusters (up to 12 in our setup), interconnected by a 2-D mesh network-on-chip. The topology of the NoC is a simple $n \times n$ mesh. Each of the first 12 nodes includes a computing cluster and a L2 bank. Another node hosts the "Fabric Controller", a special cluster instance with a single processor acting as a main controller for the whole many-core platform. This node interacts directly with the host system, and is in charge of the boot sequence of other clusters and their operation control. It has the fundamental role of managing NoC routing reconfiguration, setting up partitions and starting applications. Among the remaining three nodes, one switch is reserved to communications with an I\O interface (GPPA reading and writing ports), while the other two are temporarily left unused, and are available for future extension of the computation power.

Every full-cluster block is linked to a switch of the on-chip network with two network interfaces (NIs), a master and a slave one, supporting OCP (Open Core Protocol). The master NI is dedicated to the core transactions, while the slave NI is used for accessing the internal cluster memory. Accesses to the L2 banks are feasible thanks to dedicated slave NIs.

Each cluster has an internal memory organized as private, per-core L1 instruction caches plus local L1 scratchpad data memory shared among all cores. The L2 memory is architected as a distributed shared memory, where each NoC router hosts a L2 bank. To minimize the probability of conflicts on a single L2 bank, the NoC implements **address interleaving** among L2 banks at the granularity of an instruction cache line. Overall, the memory system is organized as a partitioned global address space (PGAS). Each processor in the system can explicitly address every memory segment: local TCDM, remote TCDMs, L2, and L3 memory. Clearly, transactions that traverse the boundaries of a cluster are subject to NUMA effects: higher latency and lower bandwidth.

When the GPPA has to perform a new computation, the code binary is copied via global direct memory access (DMA) into the L2. Data is stored in the L3 (main) memory, where it is originally allocated by host programs. Permanently hosting entire data structures in the L1 TCDMs is not feasible, due to a limited size of 256 KB. The software must thus explicitly orchestrate data transfers from L3 to L1 or L2, to ensure that the most frequently referenced data are kept close to the processors. To enable performance and energy-efficient transfers, each cluster is equipped with a local DMA engine.

Baseline ENoC architecture

In the baseline architecture, the **electronic on-chip network** (ENoC) is built on top of the xpipesLite NoC architecture [126], and customized to fit the GPPA system-level requirements.

Figure 8.9 shows the compound switch developed for such GPPA ENoC. It is a 7x7 switch with 4 bidirectional ports for the geographical destinations and 3 to support all the network interfaces (1 master and 2 slaves) to connect the cores, the internal cluster memory and the L2 memory bank. The compound switch can be broken down into two different physical networks:

- A **Local Network** serves local traffic within GPPA partitions and guarantees traffic isolations across partitions [116]. Without lack of generality, I adapt from [51] proper synchronization mechanisms between communicating peers, so that it becomes possible to perform inter-cluster communication only through write transactions. Should this not be the case, then 2 VCs would be needed in the local network. This network implements overlapped static reconfigurations (OSR) as a runtime reconfiguration mechanism for the routing function, thus enabling the

dynamic management of partitions (setup, teardown, shape redefinition) in the optimized version proposed in Chapter 5.

- A **Global Network** supports global network-wide and I/O communication traffic while avoiding interference with intra-partition local traffic. This network is physically disjoint from the local one in order to infer the highest degree of isolation. Communication flows on this network are made up of both write transactions (code offload to the GPPA, data transfer from L3 into the GPPA local memory) and request/reply transactions (on an L1 instruction cache miss). Therefore, the global NoC includes 2 virtual channels (VCs) in order to avoid message-dependent deadlock. Following the design philosophy in [57], they are implemented by replicating the single VC-less switch twice. The replicated switches do not need any reconfiguration support because their routing functions are hardwired.

ONoC and Network Interface

This section presents the optical NoC and the specialized network interface. The optical network replaces the *global* electronic network of the many-core accelerator, while the local electronic network stays the same (hybrid approach).

Topology and routing method selection

Wavelength-routed optical NoCs (WRONoCs) rely on the principle of wavelength-selective routing. This mechanism yields contention-free global connectivity, and saves the time spent in routing and arbitration by hardwiring these tasks to wavelength

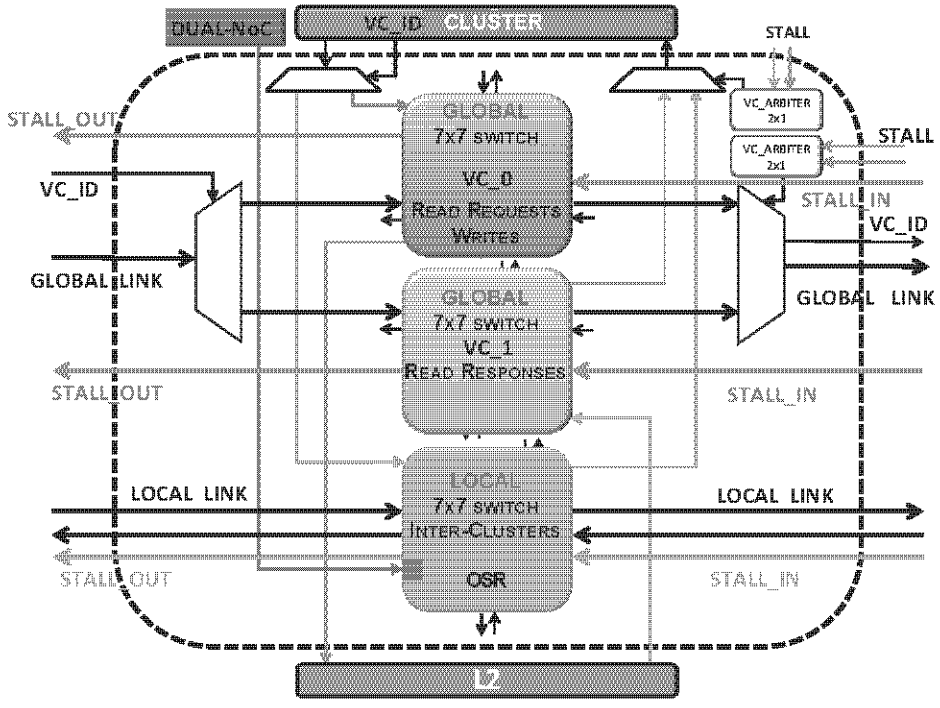


Figure 8.9: Compound switch of the electronic on-chip network.

selection [102, 131]. An alternative would be to use Space-Routed ONoC topologies (SPONoCs) [26]. SPONoCs use the wavelength-division multiplexing degree of freedom to enhance the bit parallelism of each communication flow, while WRONoCs use it to deliver contention freedom. Moreover, SPONoCs incur an unpredictable path setup latency. In contrast, WRONoCs heavily suffer from the serialization penalty on each communication flow, and from the poor scalability with network size, which rapidly leads to the proliferation of laser sources. I find the latency-throughput trade-off spanned by WRONoCs more suitable for the GPPA context, while SPONoCs are more suitable for scenarios featuring long-lasting connections. Among the possible WRONoC topologies, we selected an optical ring inspired by [83], which has been proven in [114] to minimize the design predictability gap between logic scheme and physical implementation.

Network Interface architecture

This section describes the network interface (NI) architecture for the optical network as depicted in Figure 8.10. The reader is referred to [104] for the basic NI design principles, while only the customizations for a GPPA setting are hereafter discussed. Clearly, WRONoCs move most of their control logic to the NIs, since the switching fabric is in itself a non-blocking crossbar. The upper layers of NI architectures, beyond basic domain conversion circuitry, should therefore not be oversimplified with overly abstract models.

To avoid message-dependent deadlock, every network interface needs separate buffering resources (virtual channels, VCs) for each message class. The requirement is instead met by construction in the optical switching fabric, since the lack of optical storage and the contention freedom automatically deliver the consumption assumption needed for deadlock freedom [63].

The final buffering architecture stems from considering another requirement of wavelength routing: each initiator needs an output port for each possible target, and each target needs an input port for each possible source. Overall, each target comes with 2 FIFOs (the 2 VCs) for each potential initiator.

At the transmission side, one optimization is feasible: the same 2 FIFOs are shared for all destinations and flits are sent to different optical paths afterwards. For each one of those paths, there is an arbiter that grants access to the ONoC and keeps a credit count of the empty slots at the reception buffers. Therefore, by only replicating buffers at the target while sharing those at the initiator it is still possible to associate flow control credits between every initiator-target pair.

All the FIFOs at both the transmission and the reception side must be dual-clock to move data between the processor frequency domain (we assume 700MHz) and the one used inside the NI. As hereafter explained, the latter depends on the optical bit parallelism. I used the dc-FIFO architecture presented by [129] with a size that guarantees

maximum throughput (5 slots at the transmission side). However, at the reception side, I must consider the round-trip latency in order to allow uninterrupted communications, ending up with 15-slot dc-FIFOs.
After flits are sent to the appropriate path depending on their destination, they need to be translated into a 10 GHz bit stream in order to be transmitted through the optical NoC. In fact, I assume 10 Gbit/sec modulation on each wavelength. This serialization process is parallelized to some extent to increase bandwidth and reduce latency. 3-bit parallelism means that 3 serializers of 11 bits each work in parallel to serialize the 32 bits of a flit, resulting on a bandwidth of 30 Gbps. The bit-parallelism determines the frequency inside the optical NI: 1.1 ns (0.1*number of bits) are needed to serialize a flit with 3-bit parallelism, but only 0.8 ns are needed with 4-bit parallelism. In turn, this also impacts the size of the reception dc-FIFOs based on round-trip latency, which increases from 15 to 17 slots when moving from 3 to 4-bit parallelism.
Another key issue to be considered in NIs is the resynchronization of received optical pulses with the clock signal of the electronic receiver. In this work I assume source-synchronous communication, which implies that each point-to-point communication requires a strobe signal to be transmitted along with the data on a separate wavelength. With current technology, this seems to be the most realistic solution, especially considering the promising research effort that is currently being devoted to transmitting clock signals across an optical medium [85]. The received source-synchronous clock at the reception side of the NI is then used to drive the de-serializers and, after a clock divider, the front-end of the dc-FIFOs. I assume that a form of clock gating is implemented, therefore when no data is transmitted, the optical clock signal is gated. Similarly, I assume clock gating for all electronic components, both in the baseline and in the hybrid interconnect solution.
When it comes to back-pressure management, I opt for credit-based flow control because credit tokens can reuse the existing communication paths.

The top-level NoC

In order to model the interconnection of the GPPA with host processor, L3 memory and system DMA at the top-level of the hierarchy, I use another xpipesLite switch connecting the above blocks together.

8.2.4 Usage model

This section describes all kinds of communication scenarios investigated in this work.

Offloading scenario

When a host application wants to offload some computational kernels to the GPPA, it needs to collect code (the kernel executable) and data (e.g., pointers to data in L3) into

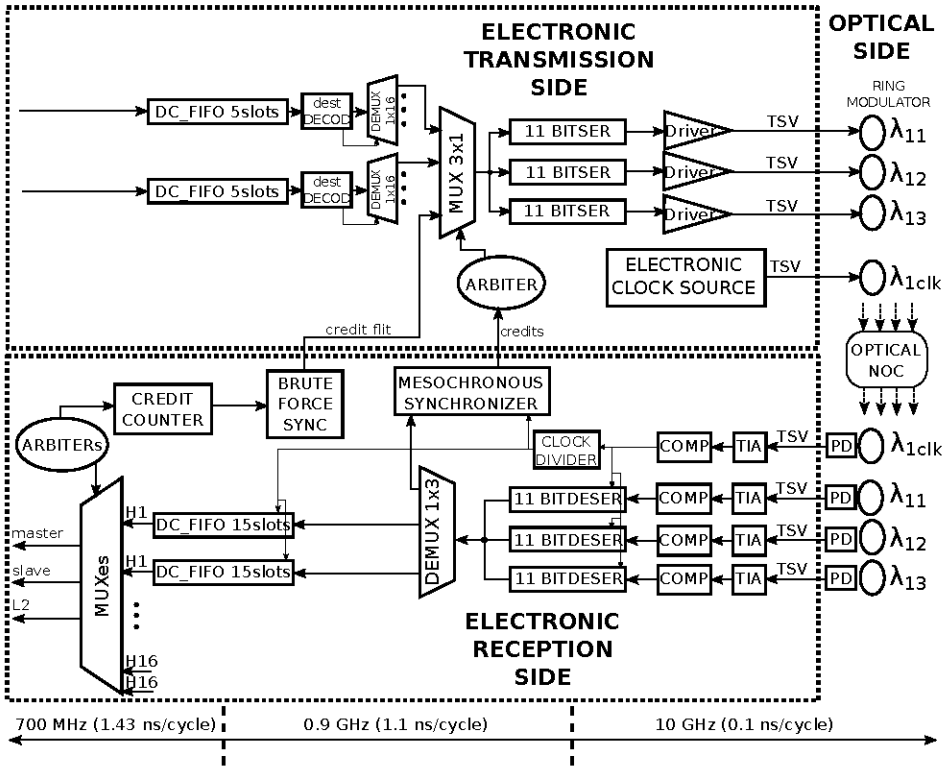


Figure 8.10: Optical Network Interface Architecture with 3-bit parallelism.

metadata structures that are forwarded to the fabric controller (FC). This is done on the host processors, which initiates a copy of the kernel executable through the system DMA into the GPPA L2 memory, from which the cores inside the cluster can fetch instructions. It has to be underlined that the cost of offloading computation to the GPPA should be kept as small as possible, otherwise it may completely hide all the benefits introduced by code acceleration. Since a relevant portion of the offload cost is in the executable copy, it is important that the sustainable bandwidth to accomplish this operation is high.

Partitioning scenario

The heterogeneous MPSoC system described in previous section features a powerful, virtualization-ready host processor. The host is capable of running multiple guest operating systems (or virtual machines, VM), each of which can potentially require the GPPA to accelerate parts of the applications it is executing. To maximize the usage of the many-core accelerator I consider a scenario where multiple VMs are allowed to concurrently offload computation to the GPPA by creating isolated cluster *partitions*. The NoC disallows communication between clusters belonging to different partitions.

Runtime scenario

Once the offload and reconfiguration (partitioning) sequences are complete, the kernel executable is launched on the selected clusters. Upon program start all the involved

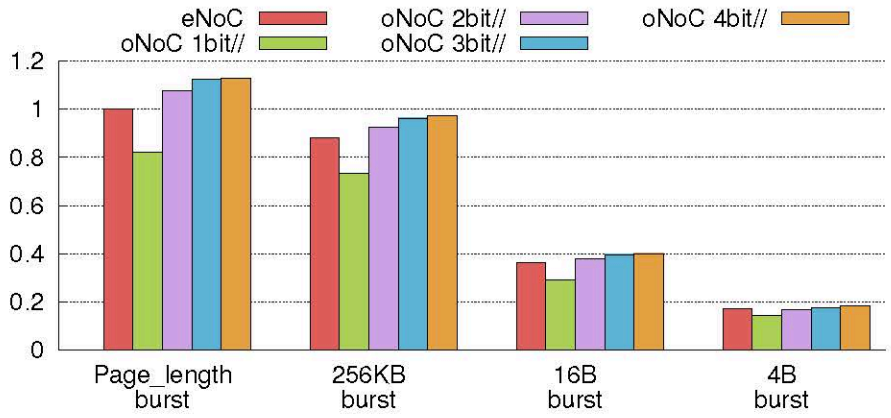


Figure 8.11: Normalized offload bandwidth as a function of DMA burst size.

cores experience cold instruction cache effects, which implies massive cache refill traffic. This is both a latency-sensitive and bandwidth-sensitive operation (the first word of a burst read is sensitive to latency, while the rest of the burst is sensitive to bandwidth). Regarding data traffic to/from the L2, the partitioning mechanism does not divide L2 memory in a partition topology-aware manner. In other words, all the cluster partitions have access to the whole L2 memory, with no affinity between clusters and their local L2 banks. While this is prone to NUMA effects, it allows better usage of L2 memory space, as no a-priori logic partitioning is done, which would lead to memory waste.

8.2.5 Experimental results

Next, basic system operations that enable the above usage model are characterized from a performance viewpoint, when running on top of the hybrid interconnect vs. the ENoC counterpart. The compound ENoC is overclocked (1 GHz) with respect to the clock speed of the processor cores (700 MHz). It thus requires the use of decoupling dual-clock FIFOs between clusters and switching fabric, which are placed after the network interfaces. Such FIFOs also serve as a key enabler for the implementation of dynamic voltage and frequency scaling. The hybrid interconnect uses an overclocked 1 GHz underlying single-layer ENoC for intra-partition traffic too. Up to 4-bit parallelism has been explored for the ONoC layer vertically stacked on top of it. The whole GPPA system with the NoC variants has been modeled and simulated with cycle accuracy in RTL-equivalent SystemC by augmenting the baseline VirtualSoC simulation environment [20].

Code offload

The system DMA reads from L3 the code to offload by means of burst transactions of parametric length (from 4 bytes to the DRAM page size), and then writes it into the GPPA L2 banks, where the code is interleaved by line address. This system operation stresses the bandwidth properties of interconnects under test.

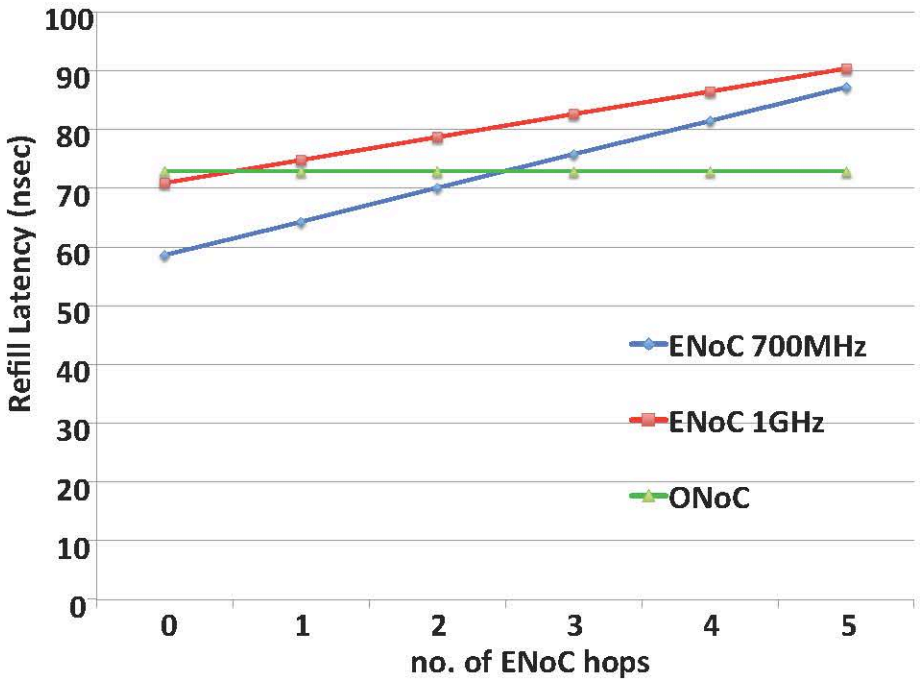


Figure 8.12: Instruction cache refill latency as a function of the number of hops to the target L2.

Code offload bandwidth is reported in Figure 8.11, normalized to that of the compound ENoC with max. burst size. For small 4-byte bursts, there is fundamentally no difference in performance between the NoCs under test. This is due to the fact that the transfer of code chunks is slowed down to an impressive extent by the latency to reach and access the off-chip L3 DRAM, which makes all other contributors negligible. As the burst size for L3 access is increased, this overhead is amortized over multiple code words, and NoC differentiation takes place. For the largest possible burst size, a 3-bit ONoC outperforms the baseline ENoC by roughly 13% in terms of improved offload bandwidth. However, with only 1-bit parallelism, 18% is the amount of performance degradation with respect to the ENoC. Finally, with 4-bit optical paths, ONoC performance turns out to be saturated.

Instruction cache refills

During program execution, each processor core incurs instruction cache misses that trigger refill operations from L2 banks. I considered a 10 cycle access latency for them. The effect of conflicting L2 accesses is here neglected due to the implementation of address interleaving, and is only considered in the final experiment in section 8.2.6. We measured the latency for the complete cache line refill operation, and reported it in Fig.8.12. Such latency depends on the distance (number of network hops) to the target L2 bank, which is plotted on the x-axys. Times are then referred to a processor core in the top-left cluster of the GPPA.

Two major results are apparent here. First, refill performance on top of the ONoC is

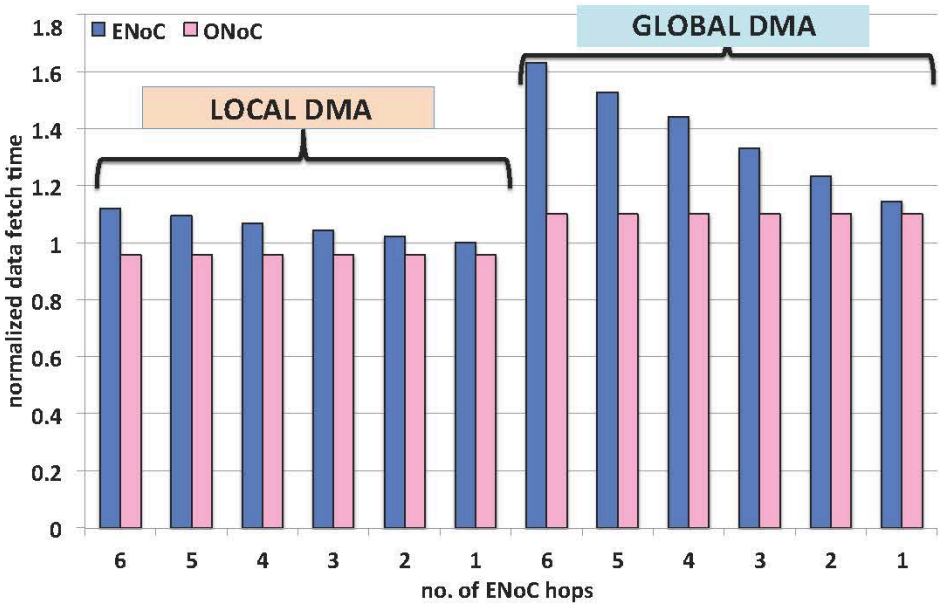


Figure 8.13: Fetching time for 16B data chunks.

position-independent, which ends up smoothing the NUMA effects in this architecture. At the same time, I find that only for accesses to the L2 bank of the same switch performance with the ENoC is slightly better. In all other cases, the multi-hop nature of the ENoC causes a widening performance gap with respect to the ONoC.

This result is a bit counterintuitive because ENoCs are believed to be far better on short distances. This is actually true for 1-bit parallelism, however the latency overhead of the ONoC is rapidly absorbed by increasing bit-parallelism. In fact, this enables a shorter length of the optical packet, and a higher ejection (injection) rate from (into) input (output) dc-FIFOs. From a latency viewpoint, with 3-bit parallelism the ONoC degrades FIFO-to-FIFO latency of flits by only 0,05% with respect to the ENoC. Such degradation grows to 2,8x with 1-bit parallelism. The power implications are addressed in section 8.2.5.

We then experimentally verified that only taking away the dual-clock FIFOs in the ENoC shifts the break-even point to a larger number of hops (2), as showed in the figure. However, this prevents application of dynamic voltage and frequency scaling policies that decouple processor speed from network speed.

Data Fetching from L3

There are two options for explicitily-managed data fetching at runtime. First, the within-cluster local DMA can be programmed to perform a read transaction from L3. Second, the system DMA can be instructed to do the same thing. However, data is then written (not read) into the GPPA (either in L2 or in L1).

Figure 8.13 reports normalized data fetch time for small data sets (16 bytes per fetch) when the local DMA is used as opposed to the global DMA, as a function of the number of hops to the GPPA I/O interface. DMA programming time is included in

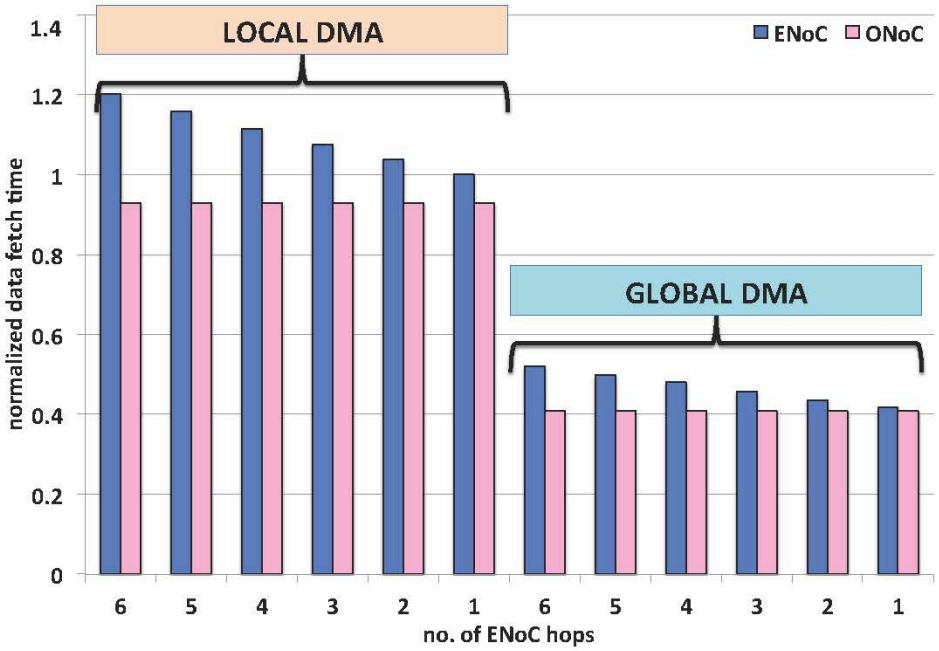


Figure 8.14: Fetching time for 5kB data chunks.

the reported results. Clearly, the system DMA is not effective for this case since the programming time of the DMA cannot be amortized over a large data transfer time, except for short range communications, where the performance difference is not significant. Interestingly, the ONoC can preserve this condition and make it independent of fetching core position in the network. As a result, the choice of the local vs. global DMA is almost irrelevant in the presence of an ONoC as global transport medium.

In contrast, when the fetched data set is significant (5k bytes, see Figure 8.14), the system DMA is clearly the right choice for a twofold reason. First, the programming time can be more easily amortized. Second, the DMA is closer to the L3, hence preventing read requests for L3 from going through the GPPA interconnect. Consequently, only read responses are forward by the system DMA to the GPPA in the form of write transactions to cluster L1 (or L2).

Power analysis

All of the electronic components (both in the ENoC and in the hybrid NoC) have been synthesized, placed and routed on a low-power 40nm industrial technology library in order to provide realistic power measurements. Clock gating was applied. Packetizers and depacketizers have not been considered since they are the same in both interconnects under test.

The static power of the hybrid NoC is derived from the composition of the power consumption of all its subblocks such as local ENoC plus ONoC NI components (frequency converters, muxes and demuxes, SERs and DESERs as well as all blocks required for flow control). The static power contribution of all optical devices is then given by: laser sources (assumed to be off-chip, yet included in the power budget),

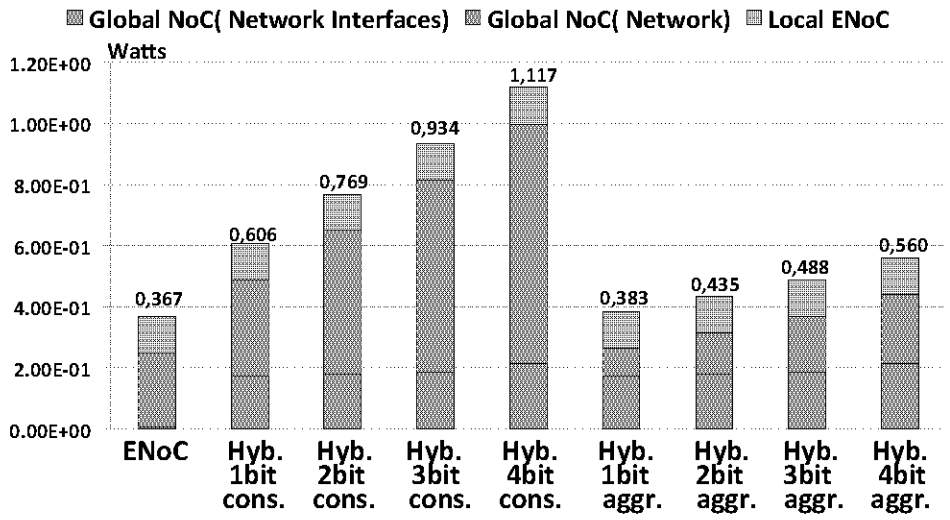


Figure 8.15: Static power for the compound ENoC vs. hybrid ONoC variants.

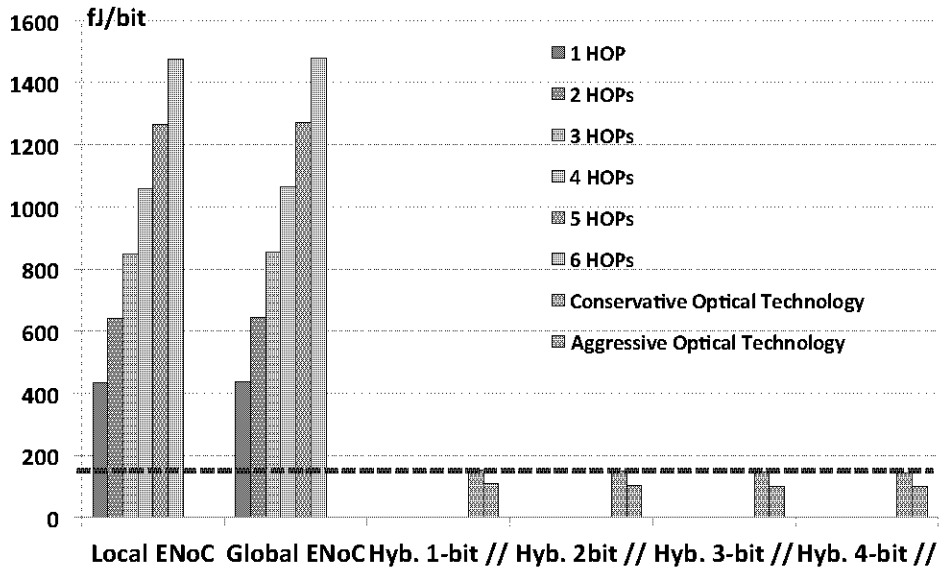


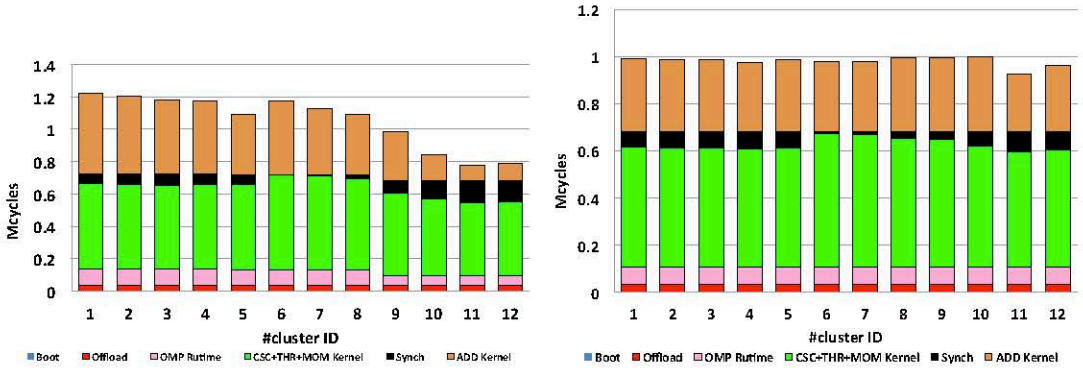
Figure 8.16: Dynamic power for the NoCs under test. The compound ENoC is broken down into its local and global networks, and so is the hybrid NoC.

thermal tuning, transmitters, receivers, and optical clock support.

In order to deliver contention-free full connectivity across 256 optical paths, 13 laser sources, 256 transmitters and receivers as well as 768 MRRs are needed. This hardware cost should be replicated for each bit of parallelism, except for the optical clock support, which is shared among the bit-parallel streams.

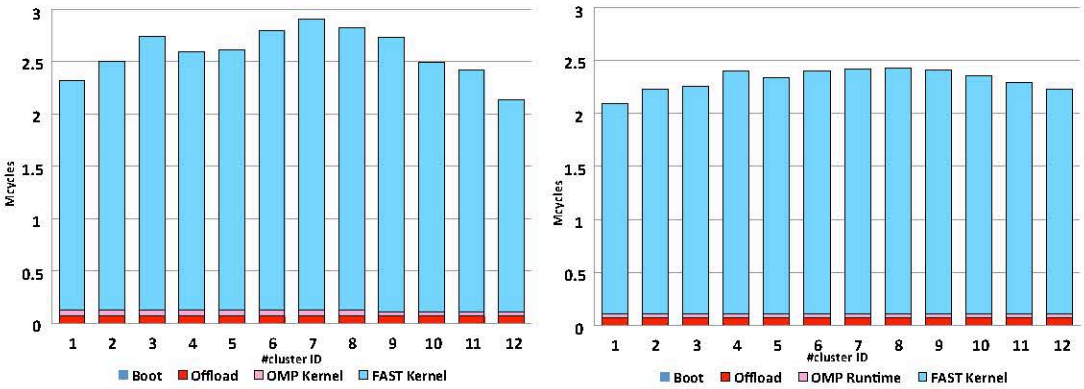
Figure 8.15 compares the total static power of ENoC vs. hybrid NoC assuming two distinct sets of parameter values for the basic optical components, namely conservative and aggressive technologies (we use the same physical parameter values reported in [115]). They reflect state-of-the-art silicon photonics as opposed to optimistic predictions for future device evolution, and are based on the projections in [12].

With a conservative optical technology, the ENoC is clearly more power efficient than ONoC regardless the bit parallelism. This is mainly due to the higher static power



(a) ENoC. (b) Hybrid NoC.

Figure 8.17: Color Tracking execution distribution among clusters.



(a) ENoC. (b) Hybrid NoC.

Figure 8.18: FAST execution distribution among clusters.

overhead consumed by all of optical devices in the network, especially by laser sources. In constrast, such an overhead is mitigated when an aggressive optical technology is considered. More precisely, the ONoC differs from only 4% up to 35% with respect to the ENoC counterpart depending on the bit parallelism.

We then computed energy-per-bit required for transmitting data over the alternative switching fabrics. For the ENoC, it amounts to 209 fJ/bit/switch. Figure 8.16 shows the energy-per-bit comparison between the ENoC and the hybrid NoC for each bit parallelism (1,2,3 and 4). As can be seen, the hybrid NoC has position-independent results and turns out to be more energy efficient than ENoC (up to an order of magnitude) regardless the specific optical technology, thus confirming that, at least in terms of enegy-per-bit, the ONoC is definitely out-of-reach.

8.2.6 Application benchmarking

We compare execution time between ENoC- vs. hybrid NoC-based GPPA platforms for real workloads. Our benchmarks are two common computer vision applications: color tracking, implemented from the open source computer vision library (OpenCV)

for single color tracking, and FAST [118], which is a corner detection for image features extraction.

Color tracking consists mainly of four kernels: color space conversion (CSC), threshold (THR), moments computation (MOM), and finally a pixel-wise addition (ADD) on the input image of the tracking segments.

Without lack of generality, both applications are mapped on the GPPA as a whole, thus emulating a 12-cluster partition that cooperatively process the computational task. 2 cores are active in each cluster. The same program is executed on each cluster, but fed by different image portions. The benchmarks should be monitored with respect to different features. In color tracking the execution is independent of the processing data, hence NUMA effects are in principle more visible. In FAST, although clusters process the same amount of pixels, the execution flow depends on the actual pixel content, hence potentially leading to divergence between clusters. The plots in Figure 8.17 compare the execution time on each cluster, expressed in Mcycles, to perform the color tracking on a single QVGA 24-bit input frame on the two platforms under test. The hybrid NoC improves the execution time by 18.1% with respect to the baseline ENoC. The hybrid NoC execution benefits also from an improvement of alignment due the abatement of NUMA effects. This is more evident on the ADD kernel, which is memory dominated. Please note that the OMP contribution consists of the overhead for the OpenMP runtime environment (OMP) [94], and that the offload time is considered as well. Figure 8.18 shows the same analysis on FAST. The application consists of a single kernel which works using a stencil pattern of accesses. Even in this case, the image is splitted into independent stripes, so that the computation is distributed among the clusters. Also the runtime (OMP) overhead is still there. The bottom chart shows the execution time in Mcycles on each cluster of the hybrid NoC platform on a QVGA 24-bit image. At the top, the same application is using the ENoC platform. In this case, the hybrid NoC improves the execution time by 16.5%.

Summary

The work proposes the first assessment of optical interconnect technology in the context of GPPA devices for the high-end embedded computing domain. The system is re-architected around an optical interconnect fabric, under a realistic hybrid integration strategy. When put at work with realistic workloads, the photonically-integrated GPPA turns out to be extremely effective in speeding up application execution by at least 15%. This translates into a static power overhead of 2.5x, which is however expected to go down to 1.3x with future silicon photonic technology. In contrast, the ONoC results more energy efficient than the ENoC counterpart (up to an order of magnitude) regardless the specific optical technology, thus confirming that in terms of energy-per-bit, the ONoC is definitely hard to beat. Overall, the above quality metrics paint a promising picture for augmenting GPPAs with optical devices, while clearly pointing

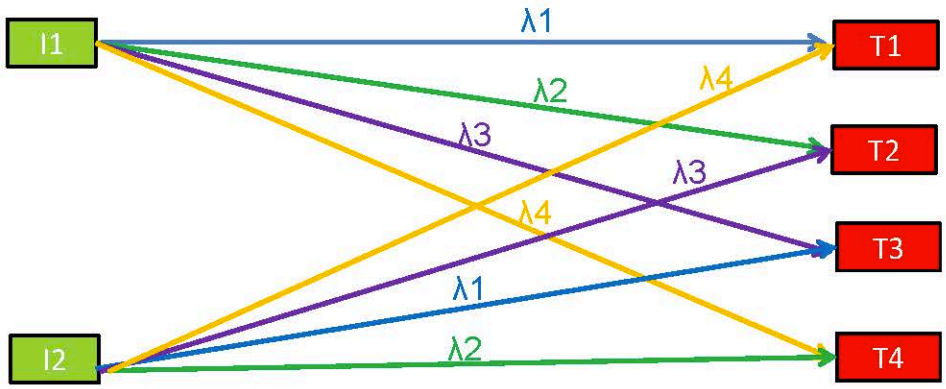


Figure 8.19: The wavelength routing concept.

to the most important candidate for optimization: static energy reduction through technology evolution as well as gating techniques.

8.3 SDM on top of a photonically-enabled GPPA

Many researchers are currently at work to assess the congruent multiples in performance and energy efficiency that should be expected by the photonic integration of multi- and many-core processors. However, such processors and their interconnection networks are typically viewed as monolithic resources, which fails to capture the most recent trends in the usage model of these computation-rich devices. In fact, partitioning of computation and communication resources is gaining momentum as a way of enabling application concurrency, and of consolidating software functions with heterogeneous requirements onto the same platform. Optical NoCs have never been embodied in this context. This work bridges this gap and proposes a partitioning technology for wavelength-routed ONoCs, including an algorithm for online allocation of wavelengths, that aims at their maximum reuse across partitions. This way, laser sources that are not in use at a given point in time can be powered-off, thus mitigating the most significant contribution to static power dissipation in optical NoCs.

8.3.1 Selection of photonic NoCs

The implementation of an electronic NoC capable of partitioning and isolation has already been addressed in previous literature. However, there is currently no such technology for optical NoCs, and the approach cannot be a simple transposition of previous solutions due to the different nature of the interconnection medium. With a photonic interconnect, the ultimate objective while partitioning should be to reuse wavelengths as much as possible. In this direction, I consider WRONoCs due to the potentially significant reduction that the proposed approach may yield on their static power dissipation. At the same time, I enjoy their benefits in terms of contention-freedom and performance-guaranteed communication.

In fact, wavelength-routed optical NoCs rely on the principle of wavelength-selective routing, which enables every initiator to communicate with every target at the same time by using different wavelengths. For example, initiator I1 can use $\lambda 1$, $\lambda 2$, $\lambda 3$, and $\lambda 4$ to reach targets T1, T2, T3 and T4 respectively. Similarly, other initiators will reach the same targets with non-conflicting wavelengths (e.g. wavelength $\lambda 2$ for T1), as illustrated in Figure 8.19. The topology connectivity pattern is chosen to ensure that wavelengths will never interfere with each other on the network optical paths. WRONoCs support contention-free all-to-all communication with a typical modulation speed of 10 Gbps/wavelength. The optical network is implemented on an optical layer vertically stacked on top of the baseline electronic layer, and with off-chip laser sources.

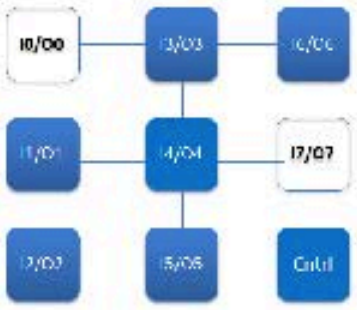
In the target architecture, that is the same of previous section in this chapter, I consider wavelength routing for the partition-capable network, although this requires some customization to work around the global connectivity it delivers. Instead, communication with the off-chip memory ports is possible from every node by using four separate and cost-effective photonic buses with a different communication protocol: two buses for the requests from nodes to the two memory controllers using the multiple-writer-singlereader protocol, and two buses for the replies using the single-reader-multiple-writer protocol. In this case, it is not cost-effective to use a laser-greedy WRONoC, as the alternative choice allows to implement all the required communication paths with only 2 wavelenghts. The arbitration of the wavelengths for off-chip DRAM access is justified by the fact that accesses are bursty and sporadic, since they are aimed at uploading or downloading processing data onto/from the GPPA. If more wavelengths are required in order to increase memory access bandwidth, this can be easily delivered by space-division-multiplexing (i.e., by increasing the number or waveguides) rather than by increasing the number of wavelengths. In any case, should I need more wavelengths for memory access, not overloading optical power waveguides with too many splitters. Therefore, it is reasonable to conceive dedicated laser sources for the off-chip memory network and dedicated sources for the partition capable network. The latter are the explicit target of our optimization.

In this work, I test several well-known WRONoC topologies for inter-node communications in the partition-capable photonic NoC: the λ-router, the GWOR and several ring variants. However, I take a radically different perspective to their comparative analysis: their suitability for laser source reuse in the context of a partition-enabled multicore architecture.

8.3.2 Dynamic partitioning

WRONoC topologies are designed with enough wavelengths to guarantee all-to-all communication. However, the GPPA must allocate isolated partitions to service several requests concurrently. This means that, at any given moment, many of the communication paths that are implemented in the chip will not be used. If I choose the nodes

	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
I_0	-	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7
I_1	λ_5	-	λ_1	λ_2	λ_3	λ_4	λ_7	λ_6
I_2	λ_3	λ_5	-	λ_1	λ_2	λ_7	λ_4	λ_5
I_3	λ_1	λ_5	λ_6	-	λ_7	λ_2	λ_3	λ_4
I_4	λ_6	λ_4	λ_5	λ_7	-	λ_1	λ_2	λ_3
I_5	λ_4	λ_3	λ_7	λ_5	λ_6	-	λ_1	λ_2
I_6	λ_2	λ_7	λ_3	λ_4	λ_5	λ_6	-	λ_1
I_7	λ_7	λ_2	λ_4	λ_6	λ_1	λ_3	λ_5	-



Background Partition = [I0,I7]
uses wavelength {lambda7}
no. of active laser sources = 1
Candidate New Partition = [I2,I5]
reuses wavelength {lambda7}
no. of active laser sources = 1
Candidate New Partition = [I1,I2]
uses wavelengths {lambda1, lambda6}
no. of active laser sources = 3

Figure 8.20: Truth table of the 8x8 gwor and basic example to set up partitions with and without wavelength reuse.

that compose each partition so that intra-partition communications reuse wavelengths as much as possible, I will have several unused wavelengths and will be able to power them off.

Our wavelength-reuse methodology is based on a distinctive property of optical NoCs, experimentally verified with real workloads: an optical transport medium is capable of smoothing out non-uniform memory access (NUMA) effects. That is, access latencies to distributed L2 memory banks are almost position-independent on a 2D mesh of processing elements. The practical implication is that, while with an electronic NoC partitions should group cores that are physically located close to each other, with ONoCs the notion of locality does not make sense. Hence, efficient partitions may be set up by grouping cores that are physically placed far apart from each other. This is a relevant degree of freedom that our methodology exploits to come with partition configurations that optimize the degree of wavelength-reuse.

Basic Idea

Let us consider Figure 8.20, where the truth table of an 8x8 gwor topology is illustrated. This topology does not deliver self-communication, which is then assumed to be implemented via an electronic shortcut. The topology delivers connectivity to a 3x3 array fabric of computation clusters, where one core serves as the fabric controller, hence cannot take part to any partition. Let us assume that a background partition is instantiated, including clusters $\lambda 0$ and $\lambda 7$, which use $\lambda 7$ as their communication wavelength (from $\lambda 0$ to $\lambda 7$ and vice versa). At this point in time, $\lambda 7$ is the only powered-on laser source. Let us then assume that a new partition has to be activated, consist-

ing of two computation clusters. Figure 8.20 illustrates two options. In the first one, clusters $\lambda 2$ and $\lambda 5$ are activated based on a smart selection policy where they keep making use of the same $\lambda 7$ for their inter-cluster communication. Instead, should the runtime manager select clusters 1 and 2, then laser sources $\lambda 1$ and 6 would need to be activated to deliver intra-partition communication. In this sub-optimal case, 3 laser sources would be on at the same time after the second partition is set up. Clearly, it is possible to come up with a partition allocation algorithm that can meet connectivity requirements while making a conscious use of laser sources.

Greedy algorithm

We propose a greedy algorithm to allocate partitions of any number of nodes in real time. Before executing the algorithm to service a new request, I have a set of already allocated nodes and wavelengths for the existing partitions. The set of already allocated wavelengths always includes the two that are used for communication with the memory controllers. The greedy algorithm follows several steps to generate the new partition:

- Randomly choose a free node to start the partition.
- Until I have the desired number of nodes in the new partition, I keep adding nodes following these steps:
  - Try to reuse the allocated wavelengths to add a new node to the partition.
    - First, I find the free nodes I can reach from the nodes in our partition using already allocated wavelengths.
    - Out of those free nodes, I select only the ones that require minimum number of extra wavelengths for full connectivity inside the partition.
    - If there are several free nodes that are equally good, I choose randomly among them.
  - If there are no nodes reachable from the partition with the allocated wavelengths and I still have just one node in the new partition, I try to find one that satisfies the *symmetric property*: the same wavelength is used to communicate the two nodes in the two directions.
  - If the previous points failed, I simply choose a free node randomly. After choosing the next node, I add to the allocated wavelength list all the wavelengths needed for full connectivity in the new partition.

The algorithm takes a locally optimal decision at each step, and never backtracks. The complexity of the algorithm is $O(n^2)$, which makes it perfectly within reach of online execution. The actual execution time depends on the chosen processor and its internal parallelism. I will only chose to apply the algorithm if its overhead is compensated by the execution time of the request. In our experimental setup, we

consider the time to run the algorithm negligible, and demonstrate that its application would be cost-effective up to a 45% overhead with respect to the request execution time.

Exhaustive algorithm

As a high performance alternative, I also introduce an exhaustive search algorithm that finds the best possible partition for every new request. This algorithm always finds a partition that minimizes the number of allocated wavelengths. Its use on a real system is infeasible due to the high complexity and execution time, but I include it as a comparison point. The algorithm checks all the possible combinations of free nodes to build the requested partition, and then chooses the one that results on a system with minimal number of wavelengths. If there are several options that are equally good, it randomly chooses one of them. So far, the two algorithms do their best to service the current request. However, the decision for the current partition may affect future partitions. In the exhaustive search, I include two optimizations to choose the best option among all the ones with equal number of wavelengths and improve long-term results:

- Maximize wavelength reuse. I prioritize wavelengths that are already being used in several partitions. This way, we will still have large wavelength-reuse values after we remove a partition.
- Minimize wasted wavelengths. I characterize a wasted wavelength as an allocated wavelength that is used to communicate nodes inside a partition with nodes outside the partition. These communication paths will never be used, reducing the opportunities to reuse this wavelength in new isolated partitions.

Note that these optimizations cannot be applied to our greedy algorithm, where nodes are added one by one. If I applied it when adding a node, I would reduce the reusing opportunities to add the next ones.

8.3.3 Static partitioning

In the previous section, I started from a fully connected optical network and dynamically set partitions on top of it. We now explore a different option: partitions statically built on the chip at fabrication time. This option lacks the flexibility of the dynamic partitioning to accommodate requests of any size, but gives us the opportunity to design very power efficient partitions that reuse a minimum number of wavelengths. In practice, it means having several smaller and independent ONoCs instead of a single big one. I must carefully decide the number of partitions to build and their size, because it will not be possible to modify them later on. I analyze the request trace and extract the most common partition size and the most useful partition mix. I decide

to test two different static configurations: one with homogeneous static partitioning (4 partitions of 4 nodes each, 4 being the weighted average partition size), and one with a mix of static partitions (4 partitions of 2, 4, 4, and 6 nodes, respectively, because this is the mix that would best fit all the partition mixes we observe over time). All the small networks are built with optical rings, and the same wavelengths are reused as much as possible across them. This radical design choice is fully compatible with modern programming models. In fact, the notion of cluster-based many-core accelerators is now central in two very representative examples: OpenCL and OpenMP. Both of them ensure portability among different accelerator targets by allowing the runtime system to map the user request to a smaller number of physical resources. With the latest version, OpenMP 4.0 is going further in the direction of integrating the notion of computation clusters in the programming interface, and guarantees that a smaller number of available clusters than those possibly requested at the application level does not constitute a problem.

8.3.4 Methodology

We set up a simulation platform that processes request traces and generates execution time and wavelength-usage results for all the configurations. The first step, common for the dynamic and static partitioning schemes, is to randomly generate several request traces (each one from a different host computer) that will all simultaneously target the GPPA. This traces store the number of nodes (randomly chosen between 2 and the total number of nodes divided 2) and the execution time required to process each request, as well as the computation time in the host until the next request. In the dynamic partitioning configurations, each request in the traces will be processed following several steps:

- If there are enough free nodes to accommodate the new partition, I run the algorithm to choose the nodes that minimize the number of allocated wavelengths.
- If there are not enough free nodes (but there are at least 2), I assign them all to service the request and extend the execution time. I distribute the aggregate execution time for all the nodes and apply a 10% penalty for each missing node
- If there are no free nodes (or there is just one free node), we deny the request. The computation will be run at the host computer, again extending the execution time and penalizing for the lack of parallelism. Extending the execution time (both in this and in the previous case) will delay the whole trace from that host computer.

In the static partitioning configurations, the trace processing will be slightly different:

- If there is a free static partition that fits the request, it is assigned.

- If there is not, I look for a bigger partition, in which some of the nodes will be left unused.

- If both options failed, I look for a smaller partition and extend the execution time applying the penalization.

- If there are no free partitions, I deny the request.

As I can see, the static partitioning configurations are less flexible and will result in longer execution times.

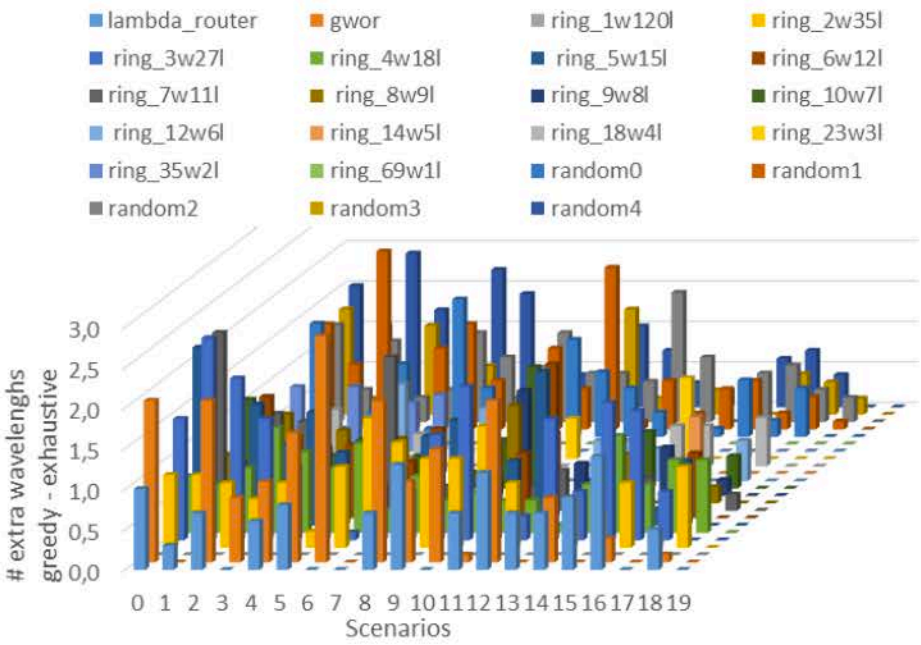
8.3.5 Results

This section presents the results for wavelength usage and laser power savings, pointing out the trade-offs between the dynamic and the static partitioning strategies.

Characterization of the algorithm

We first focus on the dynamic partitioning strategies, and determine how good our greedy algorithm is at reusing wavelengths in comparison with the exhaustive search algorithm, which is much more complex. I analyse the number of extra allocated wavelengths to create a single new partition in the λ-router, gwor, several ring designs with varying number of wavelengths and waveguides, and several random communication matrixes that do not correspond to real topologies (but are anyway useful to test the algorithm). To obtain meaningful results, I analyse the allocation of a new partition from 20 different initial scenarios. To create each of the initial scenarios, I set a small random trace and run it on every topology with the greedy algorithm.

That way, I get an equivalent starting point for every topology. Figure 8.21 shows the number of allocated wavelengths for our greedy algorithm over the exhaustive search algorithm, for new partitions of 4 and 8 nodes. I notice that when there are already many allocated nodes, and, therefore, many allocated wavelengths (towards the right-hand side of the graphs), it is easier for the greedy algorithm to find a partition that needs as few extra wavelengths as the exhaustive search. This is specially true when I create a bigger partition, because there are less degrees of freedom, so our algorithm is more likely to find an optimum set of nodes. To create the 4-node partitions, the greedy algorithm needs to add an average of 2.7 wavelengths across all scenarios, compared to 2.2 for the exhaustive search. For the 8-node partitions, the average number of added wavelengths is 7.9 and 7.6, for the greedy and exhaustive algorithms, respectively. Our greedy algorithm never needs to add more than 3 extra wavelengths over the exhaustive search algorithm, and rarely more than 2, which is an outstanding result for such a low complexity algorithm.



(a) New partition of 4 nodes

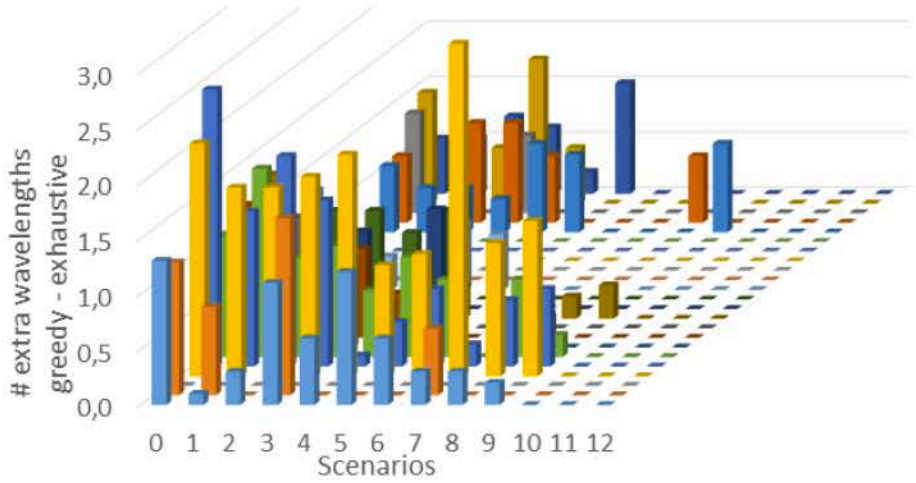


Figure 8.21: Number of allocated wavelengths for our greedy algorithm over the exhaustive search algorithm for all the considered topologies in 20 random initial scenarios. The scenarios are ordered from the highest number of free nodes (scenario 0, 16 free nodes) to the lowest (scenario 19, 4 free nodes).

Partitioning comparison of different topologies

Following the same initial-scenario methodology as in the previous section, I now perform pairwise comparisons to demonstrate that the greedy algorithm does not favour a topology over another, but rather it is the inherent characteristics of each topology that make it behave better or worse in each scenario. I compare two topologies with the greedy and the exhaustive search algorithms, and prove that, at each testing point, the same topology performs better than the other regardless of the algorithm.

Figure 8.22 shows the results of the comparison of the λ-router with the 15-wavelength ring. The positive values correspond to the cases where the λ-router needs more extra wavelenths, and a larger absolute value means a larger difference between the two

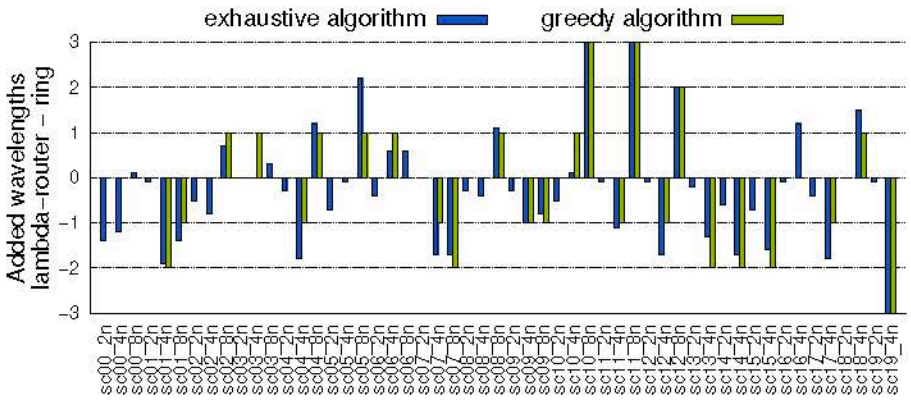


Figure 8.22: Comparison of the λ-router with the ring in 20 random initial scenarios and new partitions of 2, 4, and 6 nodes. The bars represent the number of allocated wavelengths in the λ-router over the ones allocated in the ring to set partitions of different sizes in the 20 scenarios, with the greedy and the exhaustive algorithms. Note that a larger value for the exhaustive algorithm does not mean that it allocates more wavelengths, it simply means there is a larger difference between the topologies. The absolute number of allocated wavelengths is always smaller for the exhaustive algorithm.

topologies. We notice that in every case, the greedy and exhaustive bars have the same polarity. We also the λ-router with the gwor and the randomly generated truth tables for all-to-all communication with 16 wavelengths, seeing that this was true in 99.7% of the scenarios. This points out that in each scenario one topology is more difficult to handle than the other due to its features, not to the algorithm.

Logical-level wavelength-on time

We now run the complete traces as explained in previous section and calculate the aggregated wavelength-on time, that is, the sum of the number of cycles each wavelength is used, considering that when a wavelength is not used in any partition, the corresponding laser source can be switched off. I must remember that two wavelengths are always kept on in order to guarantee communication with the memory controllers. For the dynamic partitioning configurations we have tried to match the number of wavelengths across all topologies: 16 for the λ-router and 15 for the gwor and ring. This way, I can fairly compare how each topology reacts to partitioning requests at a logical level. In this case, we introduce also the two statically partitioned configurations, as explained before. The inflexibility of the static configurations leads to a longer execution time compared with the dynamic partitioning ones, in particular, 19% extra cycles for the homogeneous partitioning and 14% extra for the mix.

Figure 8.23 depicts the aggregated wavelength-on time for the different topologies and partitioning strategies. For the dynamic partitioning I notice that our algorithm is able to cut the wavelength-on time almost in half from the always-on baseline, and is only slightly worse than the exhaustive search. Out of the three topologies,

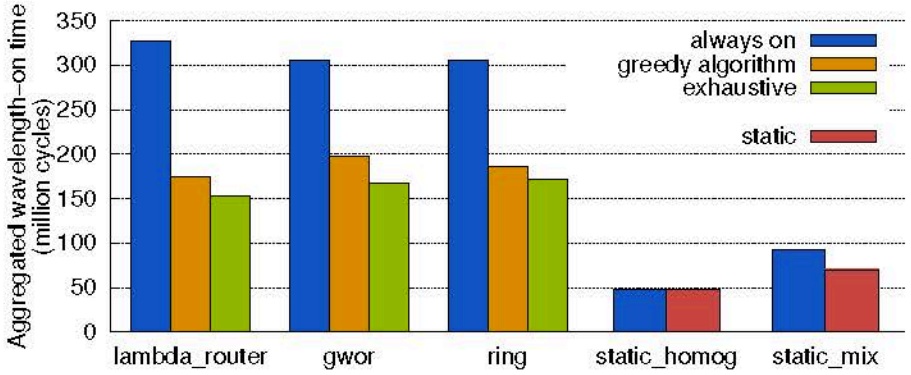


Figure 8.23: Aggregated wavelength-on time for different topologies and partitioning strategies.

the λ-router is the one that achieves the best results, even though it starts with one wavelength more than the others. The static partitioning configurations result on a much better wavelength-reuse, and the extended execution time does not reflect on longer usage time for the laser sources. In this case, switching off the unused lasers does not result in a large improvement, but the implementation of several small static partitions that reuse the same wavelengths is already an optimized starting point. Out of the two static configurations, the one with homogeneous partitions obtains the best results, as it only needs two wavelengths.

Energy analysis

To realistically calculate the laser power for the topologies, we take into account the place&route constraints of the 3D architecture. I assume an 8mmx8mm die size and consider the best optical parameters. I compute the maximum insertion loss of the optical network and calculate the minimum power required to reliably detect the optical message at the destination side. I set the GPPA frequency at 1GHz. The I physical design of the optical ring is manually generated, while the λ-router is automatically generated. The gwor is left out of the comparison due to the complexity of its physical design and the clear supremacy of the ring over filtered-based topologies. I assume that the laser stabilization time is included in the partition set-up time, along with the execution time of the greedy algorithm.

Figure 8.24 shows the energy spent by the laser sources to run the traces on the λ-router, the ring, and the static configurations. I clearly see that the λ-router cannot compete with the ring, even though it was the topology that achieved the best wavelength-reuse in the previous section. Again, the exhaustive search gives only slightly better results than the greedy algorithm. The static partitioning consumes much less laser power, and the best choice among all the configurations is the statically partitioned ONoC with homogeneous partitions. In that case, the 2 implemented wavelengths must be always on in order to support the communication with the mem-

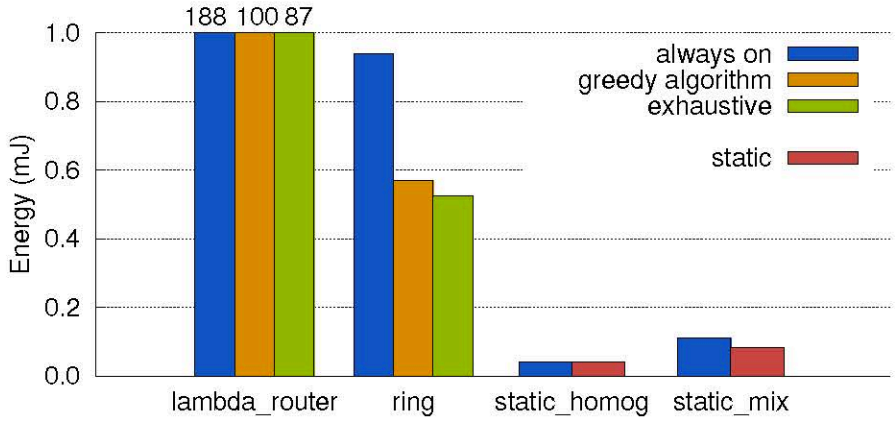


Figure 8.24: Laser source energy for different topologies and partitioning strategies.

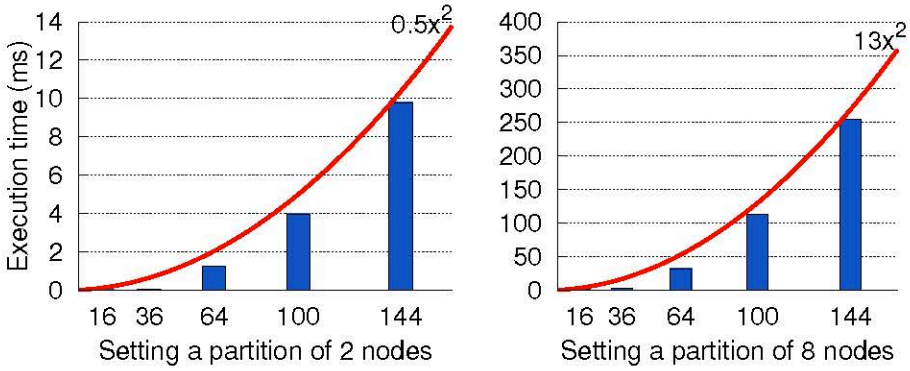


Figure 8.25: Execution time of the greedy algorithm to allocate a new partition of 2 and 8 nodes with increasing number of nodes in a ring topology. As a reference, I also plot a quadratic curve in each graph.

ory controllers. I must remember, however, that this benefit comes at the cost of a rigid chip architecture that involves worse performance. In our experiments, I consider that the time to set up the partitions is negligible. This allows us to compare the greedy and exhaustive search algorithms under the same trace, which would otherwise be impossible. The outstanding energy savings I achieve give us a large margin for the execution time of the algorithm before its use stops being cost-effective. For example, with the ring I can afford an overhead of 45% over the execution time of each request before I lose the energy savings. This can certainly accommodate the greedy algorithm, but not the inefficient exhaustive search.

Scalability of the algorithm

In this section I demonstrate that the execution time of the greedy algorithm scales quadratically with the number of nodes in the system, confirming the complexity of $O(n^2)$.

Figure 8.25 shows the execution time to build a partition of 2 and 8 nodes with an increasing number of nodes in a ring topology, on top of an ARMv7 processor simulated

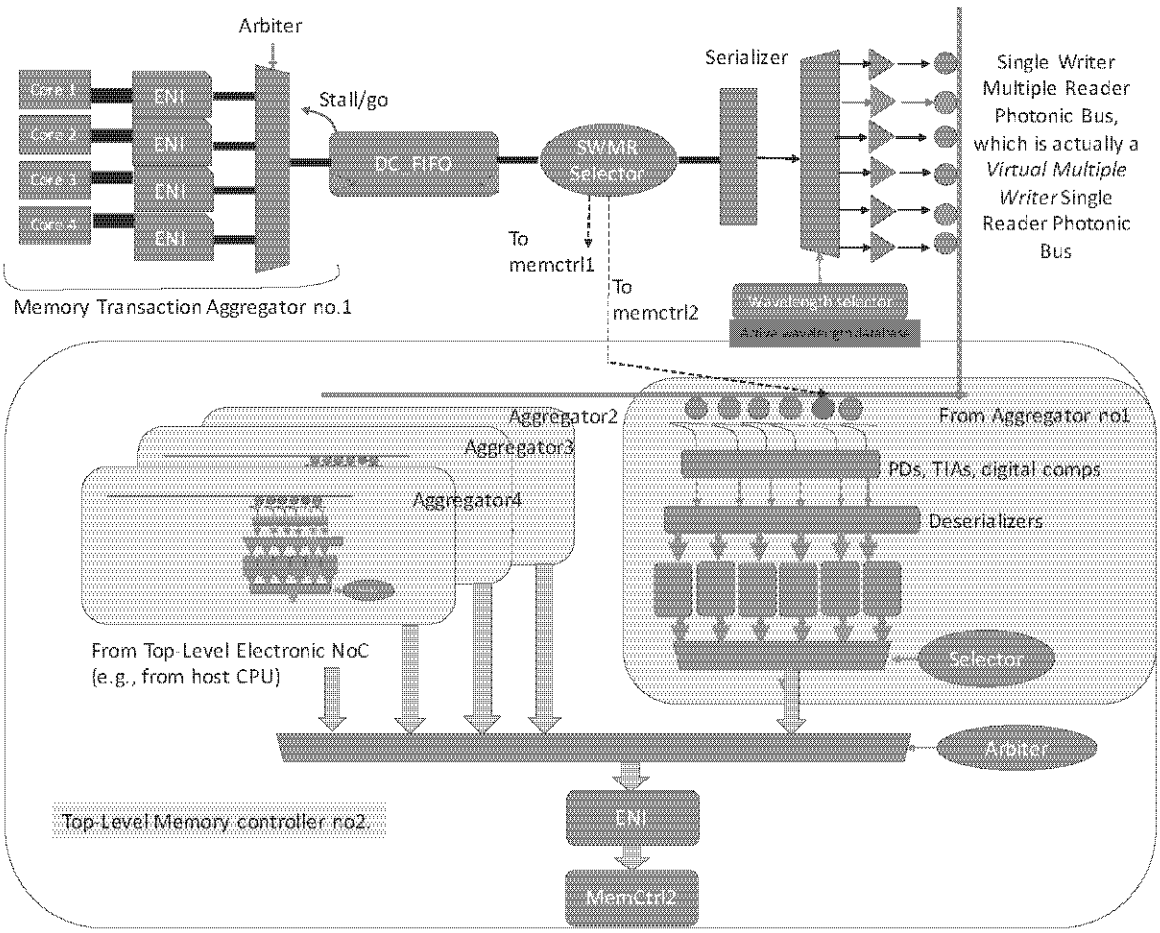


Figure 8.26: Envisioning a request network for the whole hybrid system.

on gem5. The observed trend corroborates the polynomial complexity and confirms the suitability of the algorithm for its integration on larger systems. The exhaustive search algorithm was also tested under the same scenarios, resulting in exorbitant execution times.

8.4 What's next?

Considering all the work in this chapter **we can envision for future works the Photonic Integration of the Heterogeneous Parallel Computer Architectures as a whole**, that means imaging a whole system, including the accelerator with its on-chip interconnect but also the host and main memory with the off-chip interconnection, interconnected by a photonical interconnection fabric (on-chip and one at top-level), with hybrid network interfaces based on asynchronous and optical technologies, thus paving the way for the emerging technologies affirmation, outclassing the synchronous technology.

Figure 8.26 depicts is a first attempt to provide an idea about the interconnections of the system, in particular the request network, for proof of concept. Here I am considering four memory aggregators inside the GPPA and two platform-level (not

GPPA-level) memory controllers/DRAM channels. For example, consider that at a given point in time, the active wavelength channels are: $\lambda 2$, $\lambda 5$, $\lambda 13$. In this case, each memory transaction aggregator randomly selects a single wavelength for transmission from that memory aggregator. Potentially, all memory aggregators could use the same wavelength channel and this is not a problem, since every aggregator modulates wavelengths on a different waveguide on which all memory controllers are listening. It is important to point out that packets stored in each aggregator may be potentially heading to any of the memory controllers (i.e., DRAM channels). At the same time we design also the response network and we are integrating this new features in our simulators.

This is just the starting point to come to design a system photonically-augmented as whole, but this challenge will be address in future works.

8.5 Summary

This section of Chapter 8 explored the implications of embodying the optical interconnect technology in an environment where computation and memory resources are partitioned and isolated. This reflects a usage model that is gaining momentum in the field of multi- and many-core processors, as a way to consolidate mixed-criticality applications onto the same platform, and to enable the concurrent execution of many programs at the same time. I demonstrate that this environment is of significant benefit for wavelength-routed ONoCs, which tend to the proliferation of laser sources as the core count increases. Partitioning actually yields laser power savings, and I present two approaches with different trade-off points to extract this benefits. On one hand, wavelengths are reused across partitions by using an online greedy algorithm for wavelength allocation and partition configuration. On the other hand, I present statically partitioned ONoCs and demonstrate their superior physical properties, which ultimately lead to hard-to-beat total energy figures. This approach is compatible with the flexibility of modern programming models, which can adapt to the parallelism the hardware platform exposes even if it is not the optimal one for the application at hand. However, if I am not ready to accept the drawback of building such a rigid chip architecture, I can opt for the first option and still achieve significant power savings. Finally, I envision a whole new heterogeneous parallel computing system that relies on photonic interconnection networks to connect the host, main memory, the accelerator and other devices of the system together.

Conclusions and Future Works

Complexities of scaling single-threaded performance have pushed processor designers in the direction of chip-level integration of multiple cores. Indeed, today microelectronic system design, as never before, is evolving under the effect of its two main drivers, the broadening complexity of applications and the opportunities along with the uncertainties of nanoscale technologies. On one hand, while technology is providing unprecedented levels of system integration, it is also bringing new severe concerns (overheating, tight power budgets, permanent and transient faults). On the other hand, the complexity of applications calls for large-scale SoCs and support for an increasing number of functionalities that must be supported by advanced interconnect fabrics providing high communication bandwidth together with an enhanced degree of dynamism and flexibility. NoCs, as mainstream industrial interconnect solutions, are generally believed to be the answer to such challenges. However relevant parameters such as supported topologies, switching technique, flit size, buffering styles, supported routing algorithms, etc. cannot longer represent the key differentiation between network-on-chip architectures. On the contrary, we are at the stage where the features of the on-chip network must match with the new complex requirements driven by application and technology scaling constraints that are out-of-reach for current NoC realizations. The constraints introduced by technology scaling require design methods able to provide fault-tolerance and testability to tackle the uncertainties of aggressive technology nodes and design methods able to support locally synchronous, globally asynchronous frequency domains to meet the power budget restrictions and the overheating concerns. Finally, in the era of multi- and many- core architectures as potential source of hardware acceleration in the embedded computing domain, NoCs must be envisioned to support combinations of applications that can run in several modes and that can be executed concurrently in a system changing over time, with heterogeneous and time-varying performance/reliability/power requirements. Such requirements call for design methods able to support system virtualization, partitioning and isolation capabilities of system resources. On the other hand, to exploit the real potential in terms of Gops/Watt of the hardware, there is the need also to support concurrent execution of different and parallelized applications at software layer, through efficient programming abstractions (programming models, compilers, runtime systems).

This work is a timely answer to the above concerns. The thesis proposes a Space-

Division Multiplexing approach as an innovative solution to share computing resources in a many-core programmable accelerator. First of all, the thesis has identified the basic design requirements needed to augment accelerator architectures with an enhanced degree of dynamism and flexibility, to enable effective virtualization of the resources through partitioning and guaranteeing isolation. Such requirements, having as key-enabler the capability of the NoC to support frequent and dynamic reconfiguration of the routing function, have been thoroughly investigated throughout the thesis leading to the novel design of reconfiguration techniques that have been integrated in a single NoC architecture. The thesis validates this novel mechanism while at the same time proves the co-existence and perfect support in the same switching fabric with other modern requirements, as built-in self-test and diagnosis frameworks to address post-production and lifetime permanent failures and reliable synchronization in a multi-frequency environment. In essence, the thesis contributes to the evolution of the NoC concept providing NoC-enabled architectures for the next-generation of many-core programmable accelerators where resource usage is optimized via an adaptive partitioning and isolation concept, and where selective disconnection of components from the system does not jeopardize system operation. Isolation and reconfiguration support in the NoC turn out to be a vital hardware assistance to materialize such advanced use cases.

More in detail, to address the new usage and management requirements, in my research activity I have optimized the OSRLite reconfiguration mechanism to make it suitable for highly dynamic and shared execution environments, based on the principle of flexible network partitioning. Reconfigurations do not require to drain the network from ongoing traffic, and are local to affected partitions. I have proposed different optimization strategies for network injectors to match increasing resource budgets. To the limit, I prove that fully transparent network reconfiguration is feasible. Secondly, I showed that the synergistic exploitation of multiple physical networks can lead to a fast, low-impact and scalable dynamic reconfiguration of the routing function at runtime. I bound the area affected by a reconfiguration and devised a mechanism for the fast yet controlled switching of the routing function to the new epoch in it. I rely on concurrent token and tunnel propagation mechanisms, and I proved minimum perturbation of the escape NoC, and only for an overly short amount of time with respect the reconfiguration latencies of competing approaches. The mechanism can finally scale to a large number of cores, thus coping with the scalability requirements of embedded systems. Furthermore the optimizations implemented in my research work pave the way for the frequent and fast partition reconfigurations that future applications will require to handle workload adaptivity, fault-tolerance and quality-of-service. All the experimental results were collected and the proposed optimizations were modeled and simulated with clock cycle accuracy in RTL-equivalent SystemC by augmenting the baseline VirtualSoC simulation environment. A further validation of the proposed enriched NoC has been performed by means of FPGA prototyping,

realizing a demonstrator of the optimized mechanism at work, using a leading-edge Xilinx Virtex7 FPGA.

To prove the advantages of scheduling the execution of concurrent applications on the accelerator following an SDM approach while overcoming the long simulation times provided by VirtualSoC, I augmented another simulation environment capable of simulating a complete system including the host processor and a GPPA, leveraging on the gem5 simulator hand-customized for this purpose. Here, relying on an optimized OpenMP runtime customized to support this dynamic scenario, I tested several Image Processing applications, analyzing different configurations of the platform, several memory settings and also different partition dimensions and shapes, finally compare TDM vs SDM. The final outcome is that resource sharing in a high contention and dynamic scenario is the best approach.

Finally I focused on emerging technologies, in particular on optical NoCs, augmenting the aforementioned many-core programmable accelerator with photonic interconnect technology. To the best of my knowledge, this was the first assessment of optical interconnect technology on this kind of devices, pointing out the benefits (i.e., abatement of NUMA effects, better energy-per-bit ratio) and the disadvantages, in particular concerning a complex network interface and a significant static power contribution. To project the photonic technology in an embedded virtualized environment, I explore partitioning strategies for wavelength-routed Optical NoCs, allowing to tear-down unused lasers and to keep them unused as much as possible, thus reaching the goal of mitigating the static power consumption and bringing it within reach of the embedded computing domain.

As a future work, I want to consistently develop the implications of such routing reconfiguration capability to the upper layers of the design hierarchy. At first, the first and foremost implication is on the concept of space partition, that is, on the grouping of neighboring computation units in an homogeneous parallel computing fabric to accommodate a single (parallel) application. Thanks to the reconfiguration property of the interconnect fabric, I will be able to introduce the concept of flexible space partition in shape and size, thus opening up unprecedented opportunities for resource utilization and power efficiency. In turn, this poses requirements on the runtime manager of the system, which should be able to support such flexibility by implementing some kind of application versioning. Thus I can contribute to evolve programmable accelerators towards unprecedented levels of runtime reconfiguration through a cross-layer approach to design, optimization and programming.

Bibliography

[1] Stmicroelectronics. http://www.st.com/, note = Accessed: 2016-02-08.

[2] Juan Ramón Acosta and Dimiter R Avresky. Intelligent dynamic network reconfiguration. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–9. IEEE, 2007.

[3] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures, volume 28. ACM, 2000.

[4] Konstantinos Aisopos, Andrew DeOrio, Li-Shiuan Peh, and Valeria Bertacco. Ariadne: Agnostic reconfiguration in a disconnected network environment. In Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, pages 298–309. IEEE, 2011.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, pages 483–485. ACM, 1967.

[6] Apple. Apple, Inc. Grand Central Dispatch. https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCDibdispatchRef/Reference/reference.html, note = Accessed: 2016-02-08.

[7] R. Merritt (ARM). Group describes specs for x86, arm socs. http://www.eetimes.com/document.asp?doc_id=1319306, note = Accessed: 2016-02-08.

[8] Dimiter Avresky and Natcho Natchev. Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. Computers, IEEE Transactions on, 54(5):603–615, 2005.

[9] Sandro Bartolini and Paolo Grani. Co-tuning of a hybrid electronic-optical network for reducing energy consumption in embedded cmps. In Proceedings of the First International Workshop on Many-core Embedded Systems, pages 9–16. ACM, 2013.

[10] Christopher Batten, Ajay Joshi, Jason Orcutt, Anatoly Khilo, Benjamin Moss, Charles Holzwarth, Milos Popović, Hanqing Li, Henry Smith, Judy Hoyt, et al.

Building manycore processor-to-dram networks with monolithic silicon photonics. In High Performance Interconnects, 2008. HOTI'08. 16th IEEE Symposium on, pages 21–30. IEEE, 2008.

[11] Christopher Batten, Ajay Joshi, Vladimir Stojanovć, and Krste Asanović. Designing chip-level nanophotonic interconnection networks. Springer, 2013.

[12] Scott Beamer, Chen Sun, Yong-Jin Kwon, Ajay Joshi, Christopher Batten, Vladimir Stojanović, and Krste Asanović. Re-architecting dram memory systems with monolithically integrated silicon photonics. ACM SIGARCH Computer Architecture News, 38(3):129–140, 2010.

[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track, pages 41–46, 2005.

[14] Luca Benini and Giovanni De Micheli. Networks on chips: a new soc paradigm. Computer, 35(1):70–78, 2002.

[15] Luca Benini et al. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In Proceedings of the Conference on Design, Automation and Test in Europe.

[16] Steven Manning Betker, Timothy R Vitters, and Renae M Weber. Method and system for dynamically assigning domain identification in a multi-module fibre channel switch, June 12 2007. US Patent 7,230,929.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 Simulator. ACM SIGARCH Computer Architecture News, 39(2):1–7, 2011.

[18] M. Bohr and K. Mistry. Intel's revolutionary 22 nm transistor technology. `http://download.intel.com/newsroom/kits/22nm/pdfs/22nm-detailspresentation.pdf`, note = Accessed: 2016-02-08.

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

[20] Daniele Bortolotti, Claudio Pinto, Andrea Marongiu, Matteo Ruggiero, and Luca Benini. Virtualsoc: A Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip. In Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, pages 2182–2187. IEEE, 2013.

[21] David M Brooks, Pradip Bose, Stanley E Schuster, Hans Jacobson, Prabhakar N Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish

Gupta, and Peter W Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. Micro, IEEE, 20(6):26–44, 2000.

[22] N. Brookwood. Amd fusion family of apus: Enabling a superior, immersive pc experience. http://sites.amd.com/kr/Documents/48423B/fusion/whitepaper/WEB.pdf, note = Accessed: 2016-02-08.

[23] Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. Enabling fine-grained openmp tasking on tightly-coupled shared memory clusters. In Design, Automation and Test in Europe Conference and Exhibition (DATE), 2013, pages 1504–1509. IEEE, 2013.

[24] Ruben Casado, Aurelio Bermúdez, Jose Duato, Francisco J Quiles, and José L Sánchez. A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks. Parallel and Distributed Systems, IEEE Transactions on, 12(2):115–132, 2001.

[25] Daniel M Chapiro. Globally-asynchronous locally-synchronous systems. Technical report, DTIC Document, 1984.

[26] Sai Vineel Reddy Chittamuru, Srinivas Desai, and Sudeep Pasricha. Reconfigurable silicon-photonic network with improved channel sharing for multicore architectures. In Proceedings of the 25th edition on Great Lakes Symposium on VLSI, pages 63–68. ACM, 2015.

[27] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. Heterogeneous multiprocessing solution of exynos 5 octa with arm® big. little? technology.

[28] Mark J Cianchetti, Joseph C Kerekes, and David H Albonesi. Phastlane: a rapid transit optical routing network. In ACM SIGARCH Computer Architecture News, volume 37, pages 441–450. ACM, 2009.

[29] Intel Corp. Intel teraflops project: 80-cores polaris chip. http://www.intel.com/pressroom/kits/teraflops, note = Accessed: 2016-02-08.

[30] OAR Corporation. Real-time executive for multiprocessor systems. http://www.rtems.org, note = Accessed: 2016-02-08.

[31] TILERA Corporation. Tile-gx8072 processor, product brief. http://www.tilera.com/sites/default/files/images/products/TILE-Gx8072_PB041-03_WEB.pdf, note = Accessed: 2016-02-08.

[32] TILERA Corporation. Tilera processors. http://www.tilera.com/products/processors, note = Accessed: 2016-02-08.

[33] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. Computational Science and Engineering, IEEE, 5(1):46–55, 1998.

[34] William J Dally, Larry R Dennison, David Harris, Kinhong Kan, and Thucydides Xanthopoulos. The reliable router: A reliable and high-performance communication substrate for parallel computers. pages 241–255, 1994.

[35] 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.

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

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

[38] Masood Dehyadgari, Mohsen Nickray, Ali Afzali-Kusha, and Zainalabein Navabi. Evaluation of pseudo adaptive xy routing using an object oriented model for noc. In Microelectronics, 2005. ICM 2005. The 17th International Conference on, pages 5–pp. IEEE, 2005.

[39] Robert H Dennard, VL Rideout, E Bassous, and AR LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. Solid-State Circuits, IEEE Journal of, 9(5):256–268, 1974.

[40] Andrew DeOrio, David Fick, Valeria Bertacco, Dennis Sylvester, David Blaauw, Jin Hu, and Gregory Chen. A reliable routing architecture and algorithm for nocs. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 31(5):726–739, 2012.

[41] Jose Duato, Olav Lysne, Ruoming Pang, and Timothy M Pinkston. A theory for deadlock-free dynamic network reconfiguration. part i. Parallel and Distributed Systems, IEEE Transactions on, 16(5):412–427, 2005.

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

[43] Mojtaba Ebrahimi, Masoud Daneshtalab, Juha Plosila, and Farhad Mehdipour. Md: minimal path-based fault-tolerant routing in on-chip networks. In Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific, pages 35–40. IEEE, 2013.

[44] 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.

[45] Kevin Fan, Manjunath Kudlur, Ganesh Dasika, and Scott Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on, pages 313–322. IEEE, 2009.

[46] Chaochao Feng, Zhonghai Lu, Axel Jantsch, Jinwen Li, and Minxuan Zhang. A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for network-on-chip. In Proceedings of the Third International Workshop on Network on Chip Architectures, pages 11–16. ACM, 2010.

[47] David Fick, Andrew DeOrio, Gregory Chen, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A highly resilient routing algorithm for fault-tolerant nocs. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 21–26. European Design and Automation Association, 2009.

[48] David Fick, Andrew DeOrio, Jin Hu, Valeria Bertacco, David Blaauw, and Dennis Sylvester. Vicis: a reliable network for unreliable silicon. In Proceedings of the 46th Annual Design Automation Conference, pages 812–817. ACM, 2009.

[49] Jose Flich and Jose Duato. Logic-Based Distributed Routing for NoCs. Computer Architecture Letters, 7(1):13–16, 2008.

[50] Jose Flich, Andres Mejia, Pedro Lopez, and Jose Duato. Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips. In Networks-on-Chip, 2007. NOCS 2007. First International Symposium on, pages 183–194. IEEE, 2007.

[51] Poletti Francesco, Poggiali Antonio, and Paul Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In Design, Automation and Test in Europe, 2005. Proceedings, pages 736–741. IEEE, 2005.

[52] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In Computational learning theory, pages 23–37. Springer, 1995.

[53] David Geer. Chip makers turn to multicore processors. Computer, 38(5):11–13, 2005.

[54] David Gelernter. A dag-based algorithm for prevention of store-and-forward deadlock in packet networks. Computers, IEEE Transactions on, 100(10):709–715, 1981.

[55] Alberto Ghiribaldi, Davide Bertozzi, and Steven M Nowick. A transition-signaling bundled data noc switch architecture for cost-effective gals multicore systems. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 332–337. EDA Consortium, 2013.

[56] Alberto Ghiribaldi, Daniele Ludovici, Francisco Triviño, Alessandro Strano, José Flich, José LUIS Sánchez, Francisco Alfaro, Michele Favalli, and Davide Bertozzi. A complete self-testing and self-configuring noc infrastructure for cost-effective mpsocs. ACM Transactions on Embedded Computing Systems (TECS), 12(4):106, 2013.

[57] F Gilabert, María Engracia Gómez, Simone Medardoni, and Davide Bertozzi. Improved Utilization of NoC Channel Bandwidth by Switch Replication for Cost-Effective Multi-Processor Systems-on-Chip. In Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip, pages 165–172. IEEE Computer Society, 2010.

[58] Christopher J Glass and Lionel M Ni. Fault-tolerant wormhole routing in meshes without virtual channels. IEEE transactions on parallel and distributed systems, 7(6):620–636, 1996.

[59] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to intel core duo processor architecture. Intel Technology Journal, 10(2), 2006.

[60] Maria E Gomez, Jose Duato, Jose Flich, Pedro Lopez, Antonio Robles, Nils Agne Nordbotten, Olav Lysne, and Tor Skeie. An efficient fault-tolerant routing methodology for meshes and tori. Computer Architecture Letters, 3(1):3–3, 2004.

[61] Nilanjan Goswami, Zhongqi Li, Ajit Verma, Ramkumar Shankar, and Tao Li. Integrating nanophotonics in gpu microarchitecture. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pages 425–426. ACM, 2012.

[62] Ed Grochowski and Murali Annavaram. Energy per instruction trends in intel microprocessors. Technology@ Intel Magazine, 4(3):1–8, 2006.

[63] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. Avoiding Message-Dependent Deadlock in Network-Based Systems-on-Chip. VLSI design, 2007, 2007.

[64] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, Grigorios Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor.

In Parallel and Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pages 126–137. IEEE, 2013.

[65] Gilbert Hendry, Eric Robinson, Vitaliy Gleyzer, Johnnie Chan, Luca P Carloni, Nadya Bliss, and Keren Bergman. Circuit-switched memory access in photonic interconnection networks for high-performance embedded computing. In High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, pages 1–12. IEEE, 2010.

[66] Robert Hilbrich and J Reinier Van Kampenhout. Partitioning and task transfer on noc-based many-core processors in the avionics domain. Journal Softwaretechnik-Trends, 30(3):6, 2011.

[67] Ching-Tien Ho and Larry Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. Computers, IEEE Transactions on, 53(4):427–438, 2004.

[68] John Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, Devon Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, pages 108–109. IEEE, 2010.

[69] Adapteva Inc. Parallela Reference Manual. `http://www.parallella.org/docs/parallella/manual.pdf`. Accessed: 2016-02-08.

[70] Texas Instruments Inc. Multicore DSP+ARM Keystone II System-on-Chip (SoC). `http://www.ti.com/lit/ds/symlink/66ak2h12.pdf`. Accessed: 2016-02-08.

[71] Xilinx Inc. Zynq-7000 All Programmable SoC Overview. `http://www.xilinx.com/support/documentation/datasheets/ds190-Zynq-7000-Overview.pdf`. Accessed: 2016-02-08.

[72] IRC. The international technology roadmap for semiconductors (ITRS). `http://www.itrs2.net/itrs-reports.html`, note = Accessed: 2016-02-08.

[73] Robert D Blumofe Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995.

[74] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, et al. Introduction to the cell multiprocessor. IBM journal of Research and Development, 49(4/5):589, 2005.

[75] Inc. Kalray. Kalray mppa manycore. `http://www.kalray.eu/products/mppa-manycore`, note = Accessed: 2016-02-08.

[76] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. Computer Networks (1976), 3(4):267–286, 1979.

[77] Nevin Kirman, Meyrem Kirman, Rajeev K Dokania, Jose F Martinez, Alyssa B Apsel, Matthew A Watkins, and David H Albonesi. Leveraging optical technology in future bus-based chip multiprocessors. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 492–503. IEEE Computer Society, 2006.

[78] Nevin Kirman, Meyrem Kirman, Rajeev K Dokania, Jose F Martinez, Alyssa B Apsel, Matthew A Watkins, and David H Albonesi. Leveraging optical technology in future bus-based chip multiprocessors. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 492–503. IEEE Computer Society, 2006.

[79] Sanjeev Kumar, Christopher J Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In ACM SIGARCH Computer Architecture News, volume 35, pages 162–173. ACM, 2007.

[80] George Kurian, Jason E Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C Kimerling, and Anant Agarwal. Atac: a 1000-core cache-coherent processor with on-chip optical network. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pages 477–488. ACM, 2010.

[81] James Larus. Spending moore's dividend. Communications of the ACM, 52(5):62–69, 2009.

[82] Sébastien Le Beux, Hui Li, Gabriela Nicolescu, Jelena Trajkovic, and Ian O'Connor. Optical crossbars on chip, a comparative study based on worst-case losses. Concurrency and Computation: Practice and Experience, 26(15):2492–2503, 2014.

[83] Sébastien Le Beux, Jelena Trajkovic, Ian O'Connor, Gabriela Nicolescu, Guy Bois, and Pierre Paulin. Optical ring network-on-chip (ornoc): Architecture and design methodology. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pages 1–6. IEEE, 2011.

[84] Doowon Lee, Ritesh Parikh, and Valeria Bertacco. Brisk and limited-impact noc routing reconfiguration. In Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, pages 1–6. IEEE, 2014.

[85] Jonathan Leu and Vladimir Stojanović. Injection-locked clock receiver for monolithic optical link in 45nm soi. In Solid State Circuits Conference (A-SSCC), 2011 IEEE Asian, pages 149–152. IEEE, 2011.

[86] Zheng Li, Jie Wu, Li Shang, Alan R Mickelson, Manish Vachharajani, Dejan Filipovic, Wounjhang Park, and Yihe Sun. A high-performance low-power nanophotonic on-chip network. In Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design, pages 291–294. ACM, 2009.

[87] ARM Ltd. big.little processing with arm cortex-a15 and cortex- a7. `http://www.arm.com/files/downloads/big/LITTLE/Final/Final.pdf`, note = Accessed: 2016-02-08.

[88] Michael J Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. ACM Transactions on Architecture and Code Optimization (TACO), 8(4):48, 2012.

[89] Olav Lysne, José Miguel Montan Ana, Jose Flich, Jose Duato, Timothy Mark Pinkston, and Tor Skeie. An efficient and deadlock-free network reconfiguration protocol. Computers, IEEE Transactions on, 57(6):762–779, 2008.

[90] Olav Lysne and José Duato. Fast dynamic reconfiguration in irregular networks. In Parallel Processing, 2000. Proceedings. 2000 International Conference on, pages 449–458. IEEE, 2000.

[91] Olav Lysne, Timothy Mark Pinkston, and Jose Duato. A methodology for developing deadlock-free dynamic network reconfiguration processes. part ii. Parallel and Distributed Systems, IEEE Transactions on, 16(5):428–443, 2005.

[92] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T Chakradhar, and Hans Peter Graf. A massively parallel, energy efficient programmable accelerator for learning and classification. ACM Transactions on Architecture and Code Optimization (TACO), 9(1):6, 2012.

[93] Andrea Marongiu, Paolo Burgio, and Luca Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pages 105–110. IEEE, 2012.

[94] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. Improving the programmability of sthorm-based heterogeneous systems with offload-enabled openmp. In Proceedings of the First International Workshop on Many-core Embedded Systems, pages 1–8. ACM, 2013.

[95] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a Many-core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications. In Proceedings of the 49th Annual Design Automation Conference, pages 1137–1142. ACM, 2012.

[96] C. Moore. Amd: frameworks for innovation. In Computer Architecture (ISCA), 2007 34th Annual International Symposium on. IEEE, 2007.

[97] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. IEEE Solid-State Circuits Newsletter, 3(20):33–35, 2006.

[98] NodeOS. Node operating system. `http://www.node-os.com`, note = Accessed: 2016-02-08.

[99] NVIDIA. Nvidia cuda programming guide. `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`, note = Accessed: 2016-02-08.

[100] NVIDIA. Nvidia tegra 4 family cpu architecture, white paper. `http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf`, note = Accessed: 2016-02-08.

[101] I. O'Connor and F. Gaffiot. On-chip optical interconnect for low-power. Springer, 2004.

[102] Ian O'Connor, Matthieu Briere, Emmanuel Drouard, Art Kazmierczak, Faress Tissafi-Drissi, David Navarro, Fabien Mieyeville, Joni Dambre, Dirk Stroobandt, Jean-Marc Fedeli, et al. Towards reconfigurable optical networks on chip. In ReCoSoC, pages 121–128. Citeseer, 2005.

[103] Umit Y Ogras, Radu Marculescu, Puru Choudhary, and Diana Marculescu. Voltage-frequency island partitioning for gals-based networks-on-chip. In Proceedings of the 44th annual Design Automation Conference, pages 110–115. ACM, 2007.

[104] Marta Ortín-Obón, Luca Ramini, Herve Tatenguem Fankem, Víctor Viñals, and Davide Bertozzi. A complete electronic network interface architecture for global contention-free communication over emerging optical networks-on-chip. In Proceedings of the 24th edition of the great lakes symposium on VLSI, pages 267–272. ACM, 2014.

[105] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose compu-

tation on graphics hardware. In Computer graphics forum, volume 26, pages 80–113. Wiley Online Library, 2007.

[106] Yan Pan, Prabhat Kumar, John Kim, Gokhan Memik, Yu Zhang, and Alok Choudhary. Firefly: illuminating future network-on-chip with nanophotonics. In ACM SIGARCH Computer Architecture News, volume 37, pages 429–440. ACM, 2009.

[107] Ruoming Pang, Timothy Mark Pinkston, and José Duato. The double scheme: Deadlock-free dynamic reconfiguration of cut-through networks. In Parallel Processing, 2000. Proceedings. 2000 International Conference on, pages 439–448. IEEE, 2000.

[108] Timothy Mark Pinkston, Ruoming Pang, and José Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. Parallel and Distributed Systems, IEEE Transactions on, 14(8):780–794, 2003.

[109] Plurality. OpenMP Application Program interface. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, note = Accessed: 2016-02-08.

[110] Plurality. Plurality: The HyperCore Processor. `http://www.plurality.com/hypercore.html`, note = Accessed: 2016-02-08.

[111] Fred J Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address). In Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, page 2. IEEE Computer Society, 1999.

[112] Valentin Puente, José A Gregorio, Fernando Vallejo, and Ramón Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In ACM SIGARCH Computer Architecture News, volume 32, page 198. IEEE Computer Society, 2004.

[113] Qualcomm. Qualcomm snapdragon 800 processors, product brief. `http://www.qualcomm.com/media/documents/files/qualcomm-snapdragon-800-product-brief.pdf`, note = Accessed: 2016-02-08.

[114] Luca Ramini and Davide Bertozzi. Power efficiency of wavelength-routed optical noc topologies for global connectivity of 3d multi-core processors. In Proceedings of the Fifth International Workshop on Network on Chip Architectures, pages 25–30. ACM, 2012.

[115] Luca Ramini, Paolo Grani, Hervé Tatenguem Fankem, Alberto Ghiribaldi, Sandro Bartolini, and Davide Bertozzi. Assessing the energy break-even point

between an optical noc architecture and an aggressive electronic baseline. In Proceedings of the conference on Design, Automation & Test in Europe, page 308. European Design and Automation Association, 2014.

[116] Samuel Rodrigo, Jose Flich, Antoni Roca, Simone Medardoni, Davide Bertozzi, J Camacho, Federico Silla, and Jose Duato. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip, pages 25–32. IEEE Computer Society, 2010.

[117] Alberto Ros, Manuel E Acacio, and José M García. Scalable directory organization for tiled cmp architectures. CDES, 8:112–118, 2008.

[118] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 32(1):105–119, 2010.

[119] Alberto Scandurra and Ian O'Connor. Scalable cmos-compatible photonic routing topologies for versatile networks on chip. Network on Chip Architecture, pages 121–128, 2008.

[120] Michael D Schroeder, Andrew D Birrell, Michael Burrows, Hal Murray, Roger M Needham, Thomas L Rodeheffer, Edwin H Satterthwaite, and Charles P Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. Selected Areas in Communications, IEEE Journal on, 9(8):1318–1335, 1991.

[121] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In ACM Transactions on Graphics (TOG), volume 27, page 18. ACM, 2008.

[122] Abbas Sheibanyrad and Alain Greiner. Two efficient synchronous asynchronous converters well-suited for networks-on-chip in gals architectures. Integration, the VLSI Journal, 41(1):17–26, 2008.

[123] Abbas Sheibanyrad and Alain Greiner. Two efficient synchronous asynchronous converters well-suited for networks-on-chip in gals architectures. Integration, the VLSI Journal, 41(1):17–26, 2008.

[124] Montek Singh and Steven M Nowick. Mousetrap: high-speed transition-signaling asynchronous pipelines. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 15(6):684–698, 2007.

[125] Martonosi Själander and Kaxiras. Power-effcient computer architectures: Recentadvances. In Synthesis Lectures on Computer Architecture, pages 1–96, 2014.

[126] Stergios Stergiou, Federico Angiolini, Salvatore Carta, Luigi Raffo, Davide Bertozzi, and Giovanni De Micheli. × pipes lite: A Synthesis Oriented Design Library for Networks on Chips. In Design, Automation and Test in Europe, 2005. Proceedings, pages 1188–1193. IEEE, 2005.

[127] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science and engineering, 12(1-3):66–73, 2010.

[128] Alessandro Strano, Davide Bertozzi, Francisco Trivino, José L Sánchez, Francisco J Alfaro, and José Flich. Osr-lite: Fast and deadlock-free noc reconfiguration framework. In Embedded Computer Systems (SAMOS), 2012 International Conference on, pages 86–95. IEEE, 2012.

[129] Alessandro Strano, Daniele Ludovici, and Davide Bertozzi. A library of dual-clock fifos for cost-effective and flexible mpsoc design. In Embedded Computer Systems (SAMOS), 2010 International Conference on, pages 20–27. IEEE, 2010.

[130] Chen Sun, Yu-Hsin Chen, and Vladimir Stojanović. Designing processor-memory interfaces with monolithically integrated silicon-photonics. In Conference on Lasers and Electro-Optics/Pacific Rim, page TuN4_3. Optical Society of America, 2013.

[131] Xianfang Tan, Mei Yang, Lei Zhang, Yingtao Jiang, and Jianyi Yang. On a scalable, non-blocking optical router for photonic networks-on-chip designs. In Photonics and Optoelectronics (SOPO), 2011 Symposium on, pages 1–4. IEEE, 2011.

[132] Dan Teodosiu, Joel Baxter, Kinshuk Govil, John Chapin, Mendel Rosenblum, and Mark Horowitz. Hardware fault containment in scalable shared-memory multiprocessors. ACM SIGARCH Computer Architecture News, 25(2):73–84, 1997.

[133] Francisco Triviño, Davide Bertozzi, and José Flich. A fast algorithm for runtime reconfiguration to maximize the lifetime of nanoscale nocs. In Proceedings of the 2013 Interconnection Network Architecture: On-Chip, Multi-Chip, pages 1–4. ACM, 2013.

[134] CH Van Berkel. Multi-core for mobile phones. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 1260–1265. European Design and Automation Association, 2009.

[135] 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. Solid-State Circuits, IEEE Journal of, 43(1):29–41, 2008.

[136] Dana Vantrease, Nathan Binkert, Robert Schreiber, and Mikko H Lipasti. Light speed arbitration and flow control for nanophotonic interconnects. In Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, pages 304–315. IEEE, 2009.

[137] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. In ACM SIGARCH Computer Architecture News, volume 36, pages 153–164. IEEE Computer Society, 2008.

[138] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In ACM SIGARCH Computer Architecture News, volume 38, pages 205–218. ACM, 2010.

[139] Eduardo Wachter, Augusto Erichsen, Alexandre Amory, and Fernando Moraes. Topology-agnostic fault-tolerant noc routing method. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 1595–1600. EDA Consortium, 2013.

[140] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. IEEE micro, (5):15–31, 2007.

[141] Pooria M Yaghini, Ashkan Eghbal, SA Asghari, and H Pedram. Power comparison of an asynchronous and synchronous network on chip router. In Computer Conference, pages 242–246, 2009.

[142] Young Jin Yoon, Nicola Concer, Michele Petracca, and Luca P Carloni. Virtual channels and multiple physical networks: two alternatives to improve noc performance. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 32(12):1906–1919, 2013.

[143] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for a fault-tolerant 2d-mesh network-on-chip. In Proceedings of the 45th annual Design Automation Conference, pages 441–446. ACM, 2008.

EU- or Italy-funded projects where I was invoved

- **Project:** *vIrtical*, SW/HW extensions for virtualized heterogeneous multicore platforms.
  Site: http://www.virtical.eu.
  Funding body: Collaborative Project: FP7-ICT-2011-7.
  Duration: 36 months (from 15 July 2011).
  Specification: Objective ICT-2011.3.4 Computing Systems.
  Partners: University of Bologna (Italy); University of Ferrara (Italy); Virtual Open Systems Sarl (France); STMicroelectronics Grenoble 2 Sas (France); Technological Educational Institute Of Crete (Greece); Sysgo Ag (Germany); Thales Communications Sas (France); Arm Limited (UK).

- **Project:** *PHIDIAS*, Ultra-Low-Power Holistic Design for Smart Bio-Signals Computing Platforms.
  Site: http://www.phidiasproject.eu
  Funding body: Collaborative Project: FP7
  Duration: 36 months (from 1 October 2012)
  Partners: University of Bologna (Italy); European Research Services GmbH (Germany); Ecole Polytechnique Federale de Lausanne (Switzerland); IMEC-NL (Netherlands).

- **Project:** *PHOTONICA*, Photonic Interconnect Technology for Chip-Multiprocessor Architectures.
  Site: https://sites.google.com/site/photonicaproject/home
  Funding body: FIRB 2008
  Duration: 36 months (from 1 October 2012)
  Specification: RBFR08LE6V
  Partners: University of Ferrara (Italy); University of Siena (Italy); University of Bari (Italy); University of Murcia (Spain).

Authors's Publications List

Conference Proceedings

[C1] M. Balboni, F. Triviño, J. Flich, D. Bertozzi, **"Optimizing the Overhead for Networks-on-Chip Routing Reconfiguration in Parallel Multi-Core Platforms"**, in Proceedings IEEE International Systems-on-Chip Symposium 2013, (SoCS13).

[C2] M. Balboni, M. Ortin Obon, A. Capotondi, L. Ramini, A. Marongiu, V. Viñals, D. Bertozzi, **"Augmenting Manycore Programmable Accelerators with Photonic Interconnect Technology for the High-End Embedded Computing Domain"**, in Proceedings IEEE International Networks-on-Chip Symposium 2014, (NoCS14).

[C3] M. Balboni, M. Ortin Obon, L. Ramini, L. Zuolo, M. Nonato, V. Viñals, D. Bertozzi, **"Partitioning Strategies of Wavelength-Routed Optical Networks-on-Chip for Laser Power Minimization"**, in Proceedings ACM II Workshop on Exploiting Silicon Photonics for Energy-Efficient Heterogeneous Parallel Architectures 2015, (SiPhotonics15).

[C4] M. Balboni, F. Triviño, J. Flich, D. Bertozzi, **"Optimizing the Overhead for Networks-on-Chip Routing Reconfiguration in Parallel Multi-Core Platforms"**, in IEEE Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems 2013, (ACACES13).

[C5] M. Balboni, J. Flich, D. Bertozzi, **"Synergistic Use of Multiple On-Chip Networks for Ultra-Low Latency and Scalable Distributed Routing Reconfiguration"**, in Proceedings IEEEACM Design Automation and Test in Europe 2015, (DATE15).

[C6] M. Balboni, J. Flich, D. Bertozzi, **"NoC-Centric Partitioning and Reconfiguration Technologies for the Efficient Sharing of General Purpose Programmable Accelerators"**, in Proceedings IEEEACM Design Automation and Test in Europe 2015, (DATE15).

[C7] M. Balboni, D. Bertozzi, **"NoC-Centric Partitioning and Reconfiguration Technologies for the Efficient Sharing of Multicore Programmable Accelerators"**, in Proceedings IEEEACM International Conference on High Performance Computing and Simulation 2015, (HPCS15).

[C8] G. Miorandi, M. Tala, M. Balboni, L. Ramini, D. Bertozzi, **"Evolutionary vs. Revolutionary Interconnect Technologies for Future Low- Power Multi-Core Systems"**, in Proceedings ACM 1st International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems 2016, (AISTECS16).

[C9] M. Tala, M. Balboni, D. Bertozzi, **"A Methodology to Populate the Design Space of Wavelength-Routed Optical Network-on-chip Topologies Leveraging the Add-Drop Primitive"**, in Proceedings IEEE International Networks-on-Chip Symposium 2016, (NoCS16).

Journal Papers

[J1] M. Balboni, A. Marongiu, D. Bertozzi, L. Benini, **"A Vertically Integrated Approach to Share Many-Core Accelerators between Virtualized Guest OSes in Heterogeneous MPSoCs**, submitted to IEEE Transaction on Computers, 2016.

Coauthored Project Deliverables

[D1] **"OpenMP programming model with multi-ISA compilation and QoS support"**

[D2] **"Hardware hooks for the programmable features of the system"**

[D3] **"vIrtical platform"**