“Lucian Blaga” University of Sibiu
“Hermann Oberth” Engineering Faculty
Computer Engineering Department

Optimized Algorithms for Network-on-Chip Application Mapping

PhD Thesis

Author:
Ciprian RADU, MSc

PhD Supervisor:
Professor Lucian N. Vinţan, PhD

SIBIU, September 2011

Universitatea “Lucian Blaga” din Sibiu
Facultatea de Inginerie “Hermann Oberth”
Catedra de Calculatoare şi Automatizări

Algoritmi optimizați pentru maparea aplicațiilor pe arhitecturi de tipul Network-on-Chip

Teză de doctorat

Autor:
Ing. Ciprian RADU

Conducător ştiinţific:
Prof. univ. dr. ing. Lucian N. Vinţan

SIBIU, Septembrie 2011

Dedicată fratelui meu...   Dedicated to my brother...

http://alexraduland.wordpress.com/

Mulţumiri

Munca prezentată în această teză de doctorat a fost efectuată în Centrul de Cercetare pentru Arhitecturi Avansate de Procesare a Informaţiei (Advanced Computer Architecture and Processing Systems – ACAPS – http://acaps.ulbsibiu.ro) al Universităţii "Lucian Blaga" din Sibiu, România, în perioada 2008 – 2011.

Mulţumesc coordonatorului meu ştiinţific, domnul profesor *Lucian Vinţan*, pentru încurajarea şi ghidarea mea către cariera doctorală. Coordonarea ştiinţifică, sfaturile, corectările riguroase, comentariile constructive şi suportul său necondiţionat au fost esenţiale pentru succesul meu, încă din perioada când eram doar student.

Mulţumiri, de asemenea, domnului profesor *Theo Ungerer* de la Universitatea din Augsburg din Germania pentru că mi-a permis să fac parte din echipa sa de cercetare timp de cinci luni, ca stagiu de pregătire în străinătate a doctoratului meu. Perioada din Augsburg a fost plină de inspiraţie. Am câştigat multă experienţă şi am obţinut sfaturi folositoare.

În timpul doctoratului am avut plăcerea să lucrez cu prietenul şi colegul meu, *Horia Calborean*. Aş dori să îi mulţumesc pentru buna colaborare şi pentru observaţiile sale valoroase.

Aş dori să mulţumesc, de asemenea, tuturor membrilor Catedrei de Calculatoare, în special domnului conferenţiar univ. dr. ing. *Remus Brad*, domnului conferenţiar univ. dr. ing. *Adrian Florea* şi domnului asistent univ. dr. ing. *Árpád Gellért*. A fost o plăcere să lucrăm împreună.

Le sunt profund recunoscător şi domnului profesor *Nicolae Ţăpuş* şi echipei de cercetare de la Universitatea Politehnica din Bucureşti, România. Aş dori să mulţumesc în special domnului conferenţiar univ. dr. ing. *Emil Slusanschi* şi domnului asistent univ. *Alexandru Herişanu* pentru ajutorul oferit în exploatarea sistemului nostru HPC şi pentru că mi-au permis şi m-au ajutat să obţin o parte din rezultatele experimentale pe supercalculatorul de la universitatea dumnealor.

În plus, mulţumesc celorlalţi co-autori şi colegi, *Shahriar Mahbub* (masterat) şi *Andreea Gancea* (licenţă), cu care de asemenea am lucrat.

Vreau să îmi exprim sincera şi profunda recunoştinţă familiei mele şi *Nicoletei*, pentru sprijinul şi înţelegerea oferite.

Această lucrare a fost susţinută de contractul financiar POSDRU 7706: *Creşterea rolului studiilor doctorale şi a competitivităţii doctoranzilor într-o Europă unită cofinanţat din Fondul Social European prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007 - 2013.*

Sibiu, Septembrie 2011
Ciprian Radu
http://webspace.ulbsibiu.ro/ciprian.radu/

Rezumat

În zilele noastre, tendinţele tehnologice au determinat arhitecturile de calculatoare să ajungă la aşa-numitul *power wall*. Datorită continuei micşorări a tranzistorilor, densitatea de putere pe centimetru pătrat a ajuns la limita superioară. Din această cauză, arhitecţii de calculatoare au hotărât să înceteze îmbunătăţirea performanţei *design*-urilor acestora prin intermediul scalării frecvenţei. În loc de aceasta, mai multe procesoare sunt plasate pe acelaşi *chip*. Sistemele *multicore* şi *manycore* oferă o performanţă crescută faţă de arhitecturile cu un singur *core* (nucleu de procesare), prin efectuarea de procesare paralelă. De asemenea, arhitecturile de calculator specifice pentru aplicaţii îmbunătăţesc performanţa prin utilizarea de procesoare eterogene în locul celor omogene. Evident, astfel de arhitecturi trebuie să fie interconectate pentru a comunica. Potrivit viziunii HiPEAC [1], în momentul de faţă comunicarea defineşte performanţa. Reţelele de interconectare au o foarte mare importanţă. Cele bazate pe magistrală de transmisie (bus) nu sunt potrivite pentru sistemele *multicore* şi *manycore* pentru că ele nu scalează [2].

După anul 2000, reţele interconectate pe *chip,* numite arhitecturi *Network-on-Chip* (NoC), au fost propuse drept o alternativă fezabilă pentru reţelele bus. Reţelele NoC au avantaje importante cum ar fi modularitatea şi scalabilitatea, dar sunt și extrem de limitate în resurse. Ca urmare, există multe probleme de cercetare în domeniul NoC [3].

Maparea aplicaţiilor pe arhitecturi de tipul *Network-on-Chip* este una dintre cele mai oneroase probleme (NP completă), în această zonă de cercetare. De vreme ce o abordare exhaustivă este nefezabilă, pentru această problemă sunt folosiţi algoritmi euristici. *Scopul acestei teze este să evalueze şi să optimizeze algoritmi (mono-obiectiv şi multi-obiectiv) pentru maparea aplicaţiilor pe arhitecturi de tipul Network-on-Chip*.

Primul obiectiv al acestei teze este să se prezinte stadiul actual al algoritmilor proiectați pentru problema mapării aplicaţiilor pe arhitecturi Network-on-Chip. Apoi, propunem de asemenea o taxonomie pentru aceşti algoritmi.

Zona de cercetare a arhitecturilor *Network-on-Chip* este relativ nouă. Ca atare, unelte puternice şi mature sunt încă aşteptate. Din câte ştim, la această dată nu există un cadru unitar *open source* (gratuit) pentru evaluarea şi optimizarea algoritmilor pentru maparea aplicaţiilor pe arhitecturi de tipul *Network-on-Chip*. Cel de al doilea obiectiv al nostru este să proiectăm un cadru comun pentru evaluarea şi optimizarea algoritmilor pentru diferite mapări pe arhitecturi multiple de tipul *Network-on-Chip*.

Al treilea obiectiv este să optimizăm şi să adaptăm un algoritm de tipul *Simulated Annealing* pentru maparea aplicaţiilor pe NoCuri, folosind cunoștințe de domeniu.

Al patrulea obiectiv constă în evaluarea şi optimizarea (folosind cunoștințe de domeniu) algoritmilor evolutivi pentru maparea multi-obiectiv a aplicaţiilor pe NoCuri.

În cele din urmă, ne propunem să efectuăm o explorare automată, ghidată de aplicație, a spațiului arhitectural pentru Sisteme *on Chip*. Aceasta implică sisteme specifice aplicaţiilor, cu procesoare eterogene, utilizând o rețea NoC parametrizabilă.

Această teză aduce contribuţii originale în optimizarea sistemelor de tipul *Network-on-Chip*. Contribuim cu unelte pentru simulare şi *benchmarking*. Optimizăm algoritmi pentru problema mapării aplicaţiilor pe arhitecturi NoC. De asemenea, propunem o metodă de explorare automată, ghidată de aplicație, a spațiului arhitectural pentru Sisteme *on Chip*.

“Lucian Blaga” University of Sibiu
“Hermann Oberth” Engineering Faculty
Computer Engineering Department

Optimized Algorithms for Network-on-Chip Application Mapping

PhD Thesis

Author:
Ciprian RADU, MSc

PhD Supervisor:
Professor Lucian N. Vinţan, PhD

SIBIU, September 2011

Acknowledgments

The work presented in this PhD Thesis has been carried out in the Advanced Computer Architecture and Processing Systems (ACAPS) research lab (http://acaps.ulbsibiu.ro) at “Lucian Blaga” University of Sibiu, Romania, during the years 2008 – 2011.

I thank my PhD supervisor, Professor *Lucian Vințan*, for encouraging and guiding me towards the Doctoral degree. His scientific coordination, his advices, his thorough reviews, his constructive comments and his generous support were essential for my success, starting from the period when I was just an undergraduate student.

Grateful acknowledgements go also to Professor *Theo Ungerer* from University of Augsburg, Germany, for kindly allowing me to be part of his research team for five months, as my PhD external research stage. My research stage in Augsburg was inspiring. I gained a lot of experience and obtained good pieces of advice.

During my PhD work, I had the pleasure to work with my friend and colleague, *Horia Calborean*. I would like to thank him for the good collaboration we had and for his valuable observations.

I would also like to thank to all the members from the Computer Engineering Department, especially to Associate Professor Dr. Ing. *Remus Brad*, Associate Professor Dr. Ing. *Adrian Florea* and to Assistant Professor Dr. Ing. *Árpád Gellért*. It has been my pleasure working with them.

My deep gratitude goes as well to Professor *Nicolae Țăpuș* and to his research staff from Politehnica University of Bucharest, Romania. I would like to thank especially to Associate Professor Dr. Ing. *Emil Slusanschi* and to Assistant Professor *Alexandru Herișanu* for helping me with our HPC system and for allowing and helping me do part of my experimental results on their university supercomputer.

In addition, I thank to all the other co-authors and colleagues, *Shahriar Mahbub*, MSc and *Andreea Gancea*, BSc, with whom I have also worked.

I want to express my sincere and deep gratitude to my family and to *Nicoleta*, for their support and understanding.

This work was supported by POSDRU financing contract POSDRU 7706.

Sibiu, September 2011
Ciprian Radu
http://webspace.ulbsibiu.ro/ciprian.radu/

Author's Papers

Ciprian Radu, Lucian Vinţan, *Domain-Knowledge Optimized Simulated Annealing for Network-on-Chip Application Mapping*, Submitted to an Elsevier journal, September 2011.

Ciprian Radu, Shahriar Mahbub, Lucian Vinţan, *Developing Domain-Knowledge Evolutionary Algorithms for Network-on-Chip Application Mapping*, in review (since July 25th, 2011) at Journal of Systems Architecture (**Impact Factor: 0.667; 5-Year Impact Factor: 0.768**), July 2011.

Ciprian Radu, Lucian Vinţan, *UNIMAP: UNIFIED FRAMEWORK FOR NETWORK-ON-CHIP APPLICATION MAPPING RESEARCH*, Acta Universitatis Cibiniensis – Technical Series, "Lucian Blaga" University of Sibiu, Romania, ISSN 1583-7149, May 2011, Sibiu, Romania.

Ciprian Radu, Lucian Vinţan, *Optimized Simulated Annealing for Network-on-Chip Application Mapping*, Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS-18), Politehnica Press, pp. 452-459, ISSN 2066-4451, 24 - 27 May 2011, Bucharest, Romania.

Ciprian Radu, Lucian Vinţan, *Towards a Unified Framework for the Evaluation and Optimization of NoC Application Mapping Algorithms*, Sixth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES), Academic Press, Ghent, Belgium, pp. 163-166, ISBN 978-90-382-1631-7, July 14, 2010, Terrassa (Barcelona), Spain.

Ciprian Radu and Lucian Vinţan, *Optimizing application mapping algorithms for NoCs through a unified framework*, In Proceedings of the 9-th IEEE RoEduNet International Conference, Sibiu, Romania, pp. 259 - 264, ISBN 978-1-4244-7335-9, 24-26 June 2010. IEEE Computer Society. Indexed IEEE, ISI , Scopus

Ciprian Radu, Horia Calborean, Adrian Florea, Arpad Gellert, Lucian Vinţan, *Exploring some multicore research opportunities. A first attempt*, Fifth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES), Academic Press, Ghent, Belgium, pp. 151-154, ISBN 978-90-382-1467-2, July 2009, Terrassa (Barcelona), Spain.

Adrian Florea, **Ciprian Radu**, Horia Calborean, Adrian Crapciu, Arpad Gellert, Lucian Vinţan, *Understanding and Predicting Unbiased Branches in General-Purpose Applications*, Buletinul Institutului Politehnic Iasi, Tome LIII (LVII), fasc. 1-4, Section IV, Automation Control and Computer Science Section, pp. 97-112, ISSN 1220-2169, "Gh. Asachi" Technical University, 2007, Iaşi, Romania. Indexed Zentralblatt MATH

Adrian Florea, **Ciprian Radu**, Horia Calborean, Adrian Crapciu, Arpad Gellert, Lucian Vinţan, *Designing an Advanced Simulator for Unbiased Branches' Prediction*, Proceedings of 9th International Symposium on Automatic Control and Computer Science, ISSN 1843-665X, November 2007, Iaşi, Romania.

Ciprian Radu, Horia Calborean, Adrian Crapciu, Arpad Gellert, Adrian Florea, *An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture*, The 6th EUROSIM Congress on Modelling and Simulation, (EUROSIM 2007), ISBN 978-3-901608-32-2, 9-13 September 2007, Ljubljana, Slovenia (special session: *Education in Simulation / Simulation in Education I*).

Contents

"You see things; and you say, 'Why?'
But I dream things that never were; and I say, 'Why not?'"

George Bernard Shaw

1 Introduction

In the current days, the technology trends determined computer architectures to reach the so called power wall. Due to continuously shrinking transistors, the power density per square centimeter reached the upper limit. Because of this, computer architects decided to stop improving the performance of their designs by means of frequency scaling. Rather than this, more processors are placed on the same chip. Multicore and manycore systems provide better performance than single core architectures, by performing parallel processing. Also, application specific computer architectures yield increased performance by employing heterogeneous processors instead of homogenous processors. Obviously, such architectures must be interconnected in order to communicate. According to HiPEAC's vision [1], nowadays communication defines performance. Interconnection networks are of high importance. Traditional bus-based networks are not suitable for multicores and manycores, because they do not scale [2].

After year 2000 on chip interconnection networks, called Network-on-Chip (NoC) architectures have been proposed as a feasible alternative to bus networks. NoCs have important advantages like modularity and scalability but, they are also extremely resource limited. As such, there are many outstanding research problems in the NoC field [3].

Network-on-Chip application mapping is one of the most onerous, NP-hard, problems in this area of research. Since an exhaustive approach is infeasible, heuristic algorithms are used to address this problem. *The scope of this thesis is to evaluate and optimize Network-on-Chip application mapping algorithms (using single-objective and multi-objective approaches)*.

The first objective of this thesis is to realize a state of the art regarding the algorithms designed for the Network-on-Chip application mapping problem. Then we also propose a taxonomy for these algorithms.

The Network-on-Chip research field is relatively new. Therefore powerful and mature tools are still expected. To the best of our knowledge, there is not currently an open source unified framework for the evaluation and optimization of Network-on-Chip application mapping algorithms. Therefore, our second objective is to design a framework that uses a common frame for evaluating and optimizing different state of the art mapping algorithms on multiple NoC architectures.

The third objective of this work is to adapt and optimize a general Simulated Annealing technique, for NoC application mapping, using domain-knowledge.

Our forth objective is to evaluate and optimize (using domain-knowledge) evolutionary algorithms, for Network-on-Chip application mapping, through a multi-objective approach.

Finally, we aim to perform an application driven automatic design space exploration of System-on-Chip designs. This involves entire application specific systems, with heterogeneous processors, using a NoC as interconnection.

This thesis brings original contributions in the Network-on-Chip research field. We contribute with tools for simulating and benchmarking NoC designs. We optimize algorithms for the NoC application mapping problem. We also propose an application driven automatic design space exploration method for System-on-Chip architectures.

"Well I am certainly wiser than this man. It is only too likely that neither of us has any knowledge to boast of; but he thinks that he knows something which he does not know, whereas I am quite conscious of my ignorance. At any rate it seems that I am wiser than he is to this small extent, that I do not think that I know what I do not know."

Socrates

2 Network-on-Chip Architectures

In this chapter we do a brief presentation of Network-on-Chip (NoC) architectures. We start be defining what a Network-on-Chip is and what are its origins. We then continue with the most important characteristics of Networks-on-Chip. Next we detail the fundamental NoC components and we introduce the most important terms used in this research field.

We conclude our introduction into the field of NoC architectures by presenting what are the main Network-on-Chip research problems.

2.1 Definition and Origins

Since the invention of the integrated circuit in 1958, Moore's law [4] describes a trend in Computer Engineering that is still nowadays. For more than half a century, the number of transistors that can be placed onto a single chip doubles approximately every two years (initially it was one year, than Moore readapted its law) [5]. In the early beginnings, a computer system occupied an entire room. As technology evolved, in the 70s the Large Scale Integration (LSI) era began and the computers were rack-level systems. In the 80s, Very Large Scale Integration (VLSI) era began. This meant a system can be placed on a single board. Ten years later, in the 90s, we went to chip-level systems (ULSI – Ultra Large Scale Integration). Nowadays, billion transistors can be integrated on a single die. A chip is an entire system and so, the term System-on-Chip (SoC) was coined. Systems-on-Chip make use of parallel processing at all levels: Instruction Level Parallelism (ILP), Memory Level Parallelism (MLP) and Thread Level Parallelism (TLP) [6], [7], [8], [9]. We researched these levels of parallelism previously by focusing on branch prediction [10], [11], [12] and multicore architectures [13], [14]. SoCs are feasible for a wide range of applications. However, they determine the architects to focus on the complex aspects of the communication architecture.

The continuously growing number of transistors per chip leads to a bigger and bigger gap between logic gate delays and wire delays [15]. As compared with the gate delays, the global interconnection wires used by a typical bus interconnection network determine significantly higher delays.

Systems-on-Chip also incur problems related to complexity, design flexibility and productivity and system synchronization. Achieving global synchronization is getting harder and harder as technology advances and chip speed increases.

Currently, computer architects face with the difficult problem called Power Wall. As it may be seen in the following figure, the power density grows exponentially with the clock frequency.

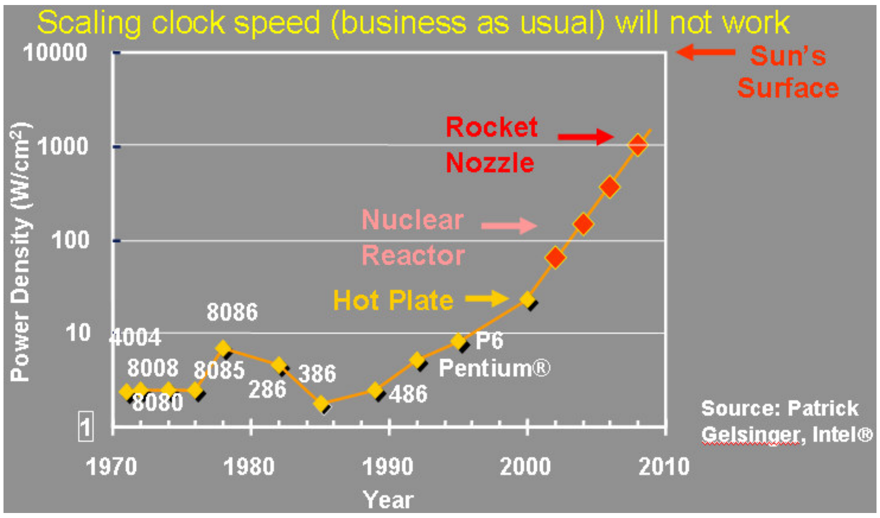

Fig. 1 Power Wall (image adapted from: CS 194 Parallel Programming Why Program for Parallelism?, Katherine Yelick, Berkeley)

The Power Wall is what mainly determined the appearance of multicore and manycore architectures [16]. Parallel programming is needed to exploit multicores. Obviously, such architectures require scalable interconnection networks. It is well-known that the bus is not a scalable interconnection network [2].

The gap between on-chip and off-chip communication is increasing. On-chip, we have greater bandwidth and shorter latencies but, the power budget is smaller. Besides *scalability*, on-chip communication also means *flexibility*, *simplicity* and *efficiency*. Flexibility is achieved by no longer using application-specific wiring (like buses do). Simplicity refers to modular, structured and regular design. Efficiency means the interconnection's ability to share global wires between different communication flows. Communication is a performance bottleneck. Because of this, the design shifts from a processing-centric to a communication-centric approach.

Simply stated, a **Network-on-Chip (NoC)** is a communication network that is used on a single chip. A Network-on-Chip consists of a number of interconnected heterogeneous devices (e.g. general or special purpose processors, embedded memories, application specific components, mixed-signal I/O cores) where communication is achieved by sending packets over a scalable interconnection network. No global wiring is used by a NoC. Wiring resources are shared by the communicating devices. The idea appeared in the 90s but it started to be researched only from year 2000. Some of the first papers introducing the NoC concept are [17], [18], [19], [20], [21], [22] and [23].

The limitations of bus based interconnects are presented in [17] and an on-chip, packet-switched, interconnection network is proposed. The authors of [18] propose a NoC design methodology and introduce the honeycomb structure. In this design, each IP core is placed on a hexagonal node connected to three switches. The switches are directly connected to their nearest neighbors.

The Network-on-Chip architecture was introduced in [19] as a better alternative to global wiring structures, used to interconnect different Intellectual Property (IP) blocks[1]. The NoC in [19] has a *regular* tile-based architecture that offers several advantages over

[1] An IP can be: a CPU core, a DSP core, a video processor etc.

traditional interconnection networks. The structured network wiring allows a better control of the electrical parameters of the network's wires. This provides the opportunity to obtain reduced power consumption. Another advantage of NoCs is given by modularity and standard network interfaces, which provide re-usability and interoperability of the modules. Wiring resources are shared by the communicating IPs: when one module is not communicating, other modules can still use the wiring resources used by the (now) idle module. No global wiring is used by a Network-on-Chip. The IPs communicate by sending packets to one another.

MicroNetwork [20] is a specific NoC architecture that was integrated into SoCs. The MicroNetwork designers argue that NoCs are the fundamental communication architectures for complex System-on-Chip designs.

A packet-switched router architecture for NoCs is presented in [21]. A two-phase design methodology for a MxN 2D mesh NoC is presented in [22]. The first phase derives a concrete architecture from a general NoC template. The second phase maps the application onto the concrete architecture to form a concrete product.

NoC is presented as a new paradigm in [23]. The Network-on-Chip characteristics are outlined. The authors introduce a protocol stack, made of three *network layers*: *physical* (wiring), *architecture and control* (composed of: data link (flits), network (routing of packets), transport (messages into packets and vice-versa)) and *software* (application and system).

The Network-on-Chip research field is relatively new and of high importance. In HiPEAC's vision [1], nowadays *communication defines performance*. Communication is essential at three levels: (1) between a processor and its memory, (2) between a multicore's different processors and (3) between processing systems and input/output devices. At the processor – memory level, the impact of communication on performance is basically controlled through cache hierarchies. At the other two levels, it is the role of the interconnection network to deal with the communication cost so that performance is less affected. More precisely, more and more processors are integrated on the same chip. Since *power defines performance*, multicores are now the solution for achieving higher performance. In this context, traditional buses, which allow the processors to communicate, no longer suffice. Networks-on-Chip provide the scalability that buses lack. Therefore, NoCs will have an increasing importance in the following years. The growing interest in this area of research is stressed out in HiPEAC's vision [1]: interconnects is one of the clusters on which HiPEAC's roadmap is built.

A component-based hardware design methodology is envisioned in the future [1]. This means that systems will be built from standard reusable components like memories, cores and interconnection networks. This design technique applies however at multiple levels. The level of abstraction increases progressively. Basic blocks (gates, registers, ALUs etc.) make components (processors, NoCs etc.). Components are then used to create different kinds of chips (CPUs, GPUs and so on), which in turn are used to obtain systems that also are interconnected, leading to systems of systems.

Obviously, the importance of interconnection networks increases as the number of communicating components raises. For intra-chip communication, the NoC is the solution and this is due to at least one factor: scalability. As the number of cores increases, the impact of memory bandwidth and memory latency becomes more and more stringent. Networks-on-Chip help at controlling the problems of memory bandwidth and latency. However, *NoCs have a lot of issues that need solving*. For example, they still require a lot of power and occupy large areas of the chip.

More precisely, research in the field of interconnection networks is required by all of HiPEAC's current research objectives: Design Space Exploration (DSE), concurrent programming models and auto-parallelization, design of optimized components, self-adaptive systems and virtualization.

Performing *Design Space Exploration (DSE)* for entire systems is currently a challenge. Unified DSE frameworks, that include the interconnection networks, are estimated to be available only between years 2016 and 2020 [1]. HiPEAC Consortium also estimates that the design space of interconnects will be feasible for exploration only around the year 2015. Only then, network traffic models, benchmarks and realistic performance/power models will be available for on-chip interconnection networks.

Developing *concurrent programming models* requires network interface mechanisms which efficiently support the cache coherence protocols and the communication between processors.

Electronic Design Automation (EDA) refers to a set of methods and tools that help at improving the system's design efficiency. EDA includes (among others) hardware/software modeling and *partitioning and mapping applications* to Multi-Processor System-on-Chip (MPSoC) architectures (related to this is the mapping problem for Network-on-Chip architectures). EDA has several challenges related to interconnection networks:

- full system simulation, including the interconnections;
- designing application-specific networks;
- designing reusable interconnection modules through interface standards.

Creating interconnection network architectures which reduce power, latency and integration area is a challenge of *designing optimized components*. The interconnection network may also be optimized by using dynamic power management techniques. Another goal is to design on-chip memory hierarchies.

A challenge of *self-adapting systems* is to design *fault tolerant network* architectures and protocols. The network traffic may also be monitored and controlled. Such data may be used by the run time system for self-adaptation.

Network interconnection is important for *virtualization* as well, from the point of view of system security and quality of service. The network may be physically or logically partitioned. A research challenge is to identify how network topologies and routing algorithms can help at system partitioning and isolation.

2.2 Characteristics

We enumerate next the most important Network-on-Chip characteristics: structured wiring, modularity, scalability, reliability, data abstraction, network modeling and productivity.

The NoC allows its communication links to be reused. No application-specific wiring is used. This approach increases the network's performance by reducing its delays and power consumption. The NoC is modular because it is made of several building blocks. As compared to a bus interconnection, the NoC has a scalable topology. The data communicated is divided into messages. Messages are composed of packets with a well-defined structure. Then, packets are further divided into flits and, flits into phits, according to the layered network modeling used. The data transferred through a NoC is more reliable because the packets are verified for errors. All the above imply a higher NoC design productivity.

NoCs are very similar to general purpose networks (for traditional parallel and distributed computers). However, they are different in at least two ways: system cores granularity and homogeneity [15]. While the general networks are coarse grained and homogenous, NoCs are rather fine grained and heterogeneous. The NoC is placed on a single chip along with its IP cores. It is much more constrained by resources and it usually interconnects different processors.

2.3 Terminology and Fundamentals

We present in this section the main components of a Network-on-Chip architecture. We also briefly present and classify the network topologies, routing and switching algorithms. We focus only on the ones which are most common in the NoC research field. During this section we also define some of the most used terms in networks domain. We note that a glossary is available at the end of this thesis, in Chapter 10.

2.3.1 Main Components

The following figure presents a typical Network-on-Chip architecture. The NoC is placed on a single chip, which is divided into regular *tiles*. A tile is a part of the chip that contains an IP core and a network router. The tile is also called a Network-on-Chip *node*. As it may be seen below, the IP cores are usually heterogeneous in such design because, generally speaking, applications are heterogeneous. We can have general purpose processors (CPU 1, 2, 3, 4), application specific cores (Application Specific Integrated Circuits, Digital Signal Processors etc.), memory modules, input/output devices and so on. Obviously, the NoC tiles can also be irregular, i.e. of different sizes.

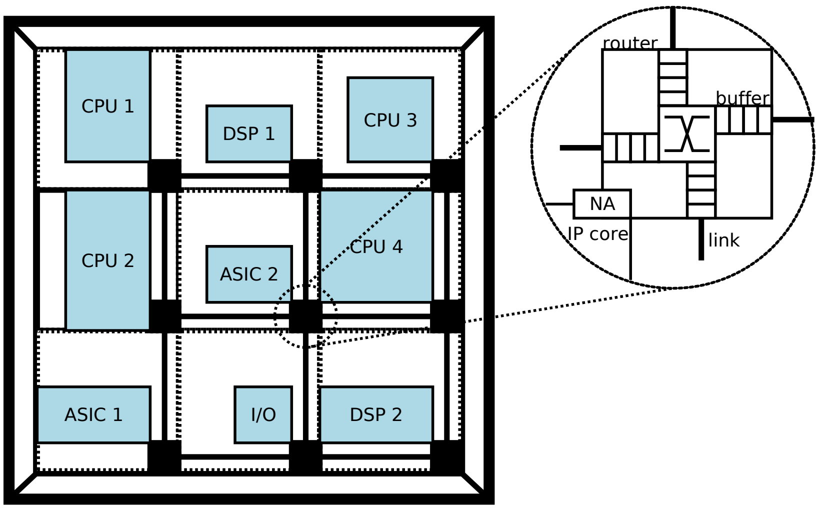

Fig. 2 NoC main components

In NoC terminology, an IP core is also called a *Processing Element (PE)*. The PE performs computation and communicates with other PE’s by *messages*, which are sent through the communication network. The IP core is connected with the network using a

Network Adapter (NA). NA's purpose is to provide an interface between the cores and the network. It specifies how the communication services are made available to any IP core type. As it may be seen in the following figure, the Network Adapter separates computation from communication.

The Network Adapter provides two interfaces: one for cores and another one for the network. It handles the messages generated by the cores by breaking them into smaller units called *packets*. A packet is the logical unit of information that is transmitted trough a network route, using *routers*. The packet is made of the following parts: a header, a data payload and a tail (or trailer). The *packet header* is the front of the packet and contains information about the source and destination NoC nodes. This helps the NoC to decide the path of the packet (its route). The data payload is the second packet component. It holds the data transmitted by the IP core across the NoC. The *packet tail* marks the end of the packet and it typically contains codes for error checking and correction (if possible).

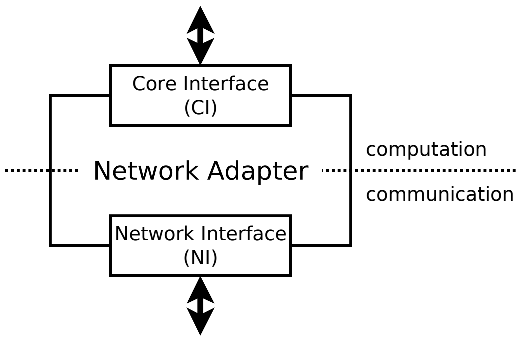

Fig. 3 The Network Adapter

A packet is made of flow control units (*flits*). A flit is the minimum unit of information that can be transferred across a link and either accepted or rejected [24]. Each flit is made of one or more physical units (*phits*). The phit is the minimum size datagram that can be transmitted in one link transaction [15]. Usually a flit is equivalent to a phit. This is also the case in this entire work. The *link* is a set of wires and fibers that carries an analog signal [24]. Transmitters and receivers are used to convert digital signals to and, respectively, from analog signals. A communication *channel* is composed of a transmitter, a link and, a receiver.

The *router* is the NoC component which drives the information though the network. It uses a *routing algorithm* to determine which of the possible paths, from source to destination, are used as routes and which route is taken by each particular packet. Buffers are used to store the flits until the router can handle them. The router contains a *switch* that provides the means to route the information. A *switching mechanism* determines how and when the data traverses its route[2]. In NoC architectures *packet switching* is used. That means messages are broken into a sequence of packets and that packets are individually routed. This is opposed to *circuit switching* where the entire network path is reserved until the whole message is transmitted.

Networks-on-Chip use a four level OSI layering. The top level is the *Application/Presentation level*. Here, messages are injected by IP cores into the NoC. At the *Session/Transport level*, the Network Adapter splits messages into packets. Packet flits are then routed from their source node to their destination at the *Network level*. Flits are divided into phits, which are communicated though links at the *Link/Data level*.

We present next some of the most common network topologies, a typical NoC router architecture and some of the most known routing and switching algorithms.

[2] According to [24], when data traverses its route is associated to a flow control mechanism. We have however adopted the terminology from [3], where the switching notion encapsulates the flow control mechanism.

2.3.2 Topologies

The network's topology specifies how its nodes are interconnected by links (communication channels). Some of the most important properties of a topology are the node *degree*, the topology *diameter* and the *bisection bandwidth*. The node degree is the number of channels entering and leaving each node. The network diameter is the length of the maximum shortest path between any two nodes of the network. The bisection bandwidth is the sum of the bandwidths of the minimum set of channels that, if removed, partition the network into two equal unconnected set of nodes. From a topological point of view, a network may be direct or indirect. In a *direct network* each node has two functions. It produces or consumes packets and it also acts as a switch. In an *indirect network*, the node has only one of the two functions.

Next, based on [24] [25] [26], we briefly show the most common network topologies. This is not by far an exhaustive coverage of the network topologies space. We start with topologies for direct networks and then we move to indirect networks.

2.3.2.1 Some of the Most Common Network Topologies

The most popular direct networks are called *k-ary d-cube* networks. Such a network has *k* nodes in each of its *d* dimensions. Each two (horizontally or vertically) adjacent nodes are interconnected. The edge nodes may have wrap-around links. In this category of networks we encounter: the linear array, the ring, the mesh, the torus and the cube.

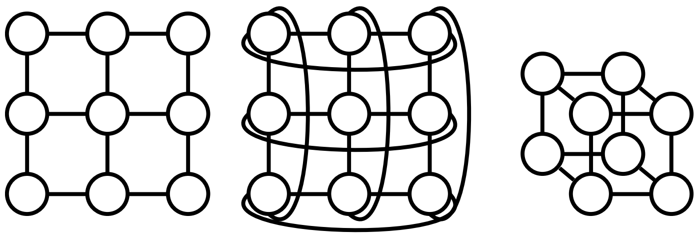
Fig. 4 From left to right: 2D mesh, 2D torus, cube

Mesh (array) topologies are also called *d-dimensional k-ary mesh*, and torus topologies are also called *d-dimensional k-ary torus* but, they are also collectively called *meshes*. The node degree is between *d* and *2d*. The network diameter is *d(k - 1)*. For an even *k*, the bisection is k^{d-1}. For an odd *k*, the bisection is a little bigger.

A lot of different topologies may be obtained from k-ary d-cube networks. Such an example is the *hypercube*.

In geometry, the hypercube is the n-dimensional analogue of a square (n = 2) or a cube (n = 3). It is a closed and convex geometrical figure that has the topological graph made of groups of opposite parallel line segments, equal in length, aligned and perpendicular to each other [27]. The hypercube is also called an *n-cube*. The 4-cube is called a *tesseract*. In networks domain, the hypercube is a 2-ary (binary) d-cube network.

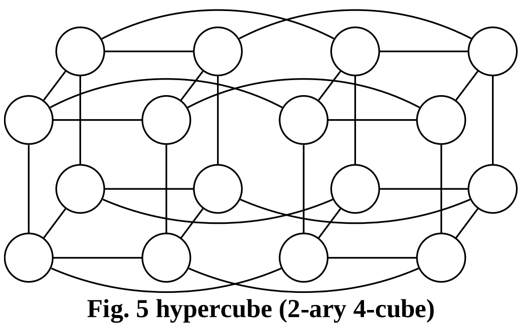
Fig. 5 hypercube (2-ary 4-cube)

In k-ary d-cube networks the diameter increases with $\sqrt[d]{N}$, where *N* is the number of nodes. However, for *tree topologies* the diameter increases only logarithmically ($2\log_2 N$ for a tree with N leaves). The bisection is one. A binary tree has degree three.
A tree topology has a root node. This node is connected to a number of (descendant) nodes, which may also be connected to a number of descendant nodes and so on until we reach the leaf nodes (i.e. nodes with no descendants). A tree in which every node (except the leaves) has a fixed number *k* of child nodes is called a *k-ary tree*. Trees may be *balanced* or *unbalanced*. A tree is called balanced when the distance from each leaf node to the root is the same.

A major disadvantage of tree topologies is that, for any two nodes, there is a single route between them. This means no fault tolerance is available. Also, removing a single link from a tree bisects the network. For these reasons, *fat-tree* [26] topologies have been proposed. In a fat-tree, a node's forward link (to the parent node) has twice the bandwidth of its backward link (to the child node). A fat-tree may also be an indirect network. In this situation, only the leaf nodes of the tree are network nodes which inject/receive packets. The rest of the nodes (including the root) perform only as switches.

The root node of a tree topology is clearly a bottleneck in the network. This problem may be solved by creating more roots. Such a network is called a *butterfly*. This is an indirect network. Its basic building block is a 2x2 switch that crosses one of each pair of edges. Similar to meshes and tori, the family of butterflies is called *k-ary d-fly* [25]. The butterfly has a diameter of $\log_2 N$ and a bisection of *N/2*.

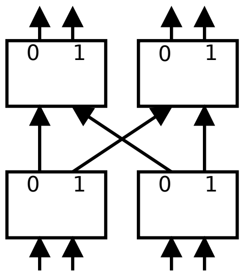

Fig. 6 The basic building block of a butterfly

One of the most important indirect networks is the *crossbar*. A N × M crossbar directly connects *N* inputs to *M* outputs with no intermediate network hops. This type of network is also called a *non-blocking network*.
The crossbar requires no buffering. There is a direct connection between any (input, output) pair of nodes. However, its cost increases rapidly as the number of inputs and/or outputs increases significantly. Ideally, it would be to have an entire network as a crossbar but, this is not possible because of its high cost. Otherwise, any router from a direct network contains a crossbar.

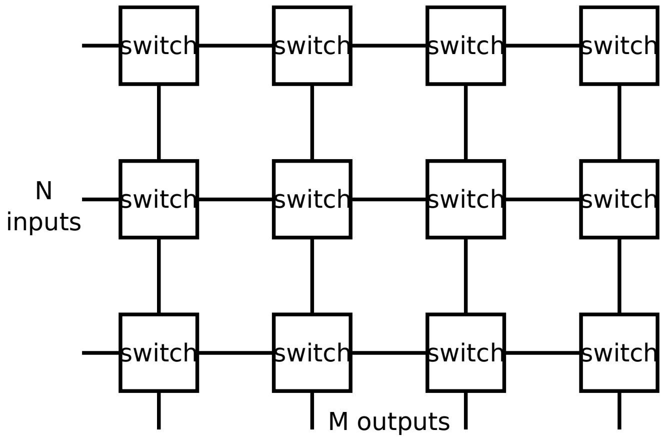

Fig. 7 N x M crossbar

Crossbars are also used at building other indirect networks. Such an example is the *Clos* [25] network. This is a three-stage network in which each stage is composed of a number of crossbars switches. A Clos is usually characterized by a triple *(m,n,r)*, where *m* is the number of middle-stage switches, *r* is the number of input and output switches and *n* is the number of inputs and outputs that each input and output switch has. The *r* input switches are *n x m* crossbars. The middle switches are *r x r* crossbars and the *r* output switches are *m x n* crossbars. The main advantage of the Clos network is given by its high routing path diversity: for any traffic, there are *m* possible routes available. A (2,2,2) Clos network is also called a *Benes* network.

Obviously, a lot of other network topologies exist in the literature. For example, the *concentrated mesh* [28] is a mesh-like directed network where a router services four nodes. The *flattened butterfly* [29] is a variant of the butterfly network. It halves the cost of a Clos network, with a similar performance. There are also hybrid topologies and irregular networks but, we confine to the most popular ones.

2.3.3 Router Architecture

We briefly present next a typical router architecture for Networks-on-Chip along with its mode of operation.

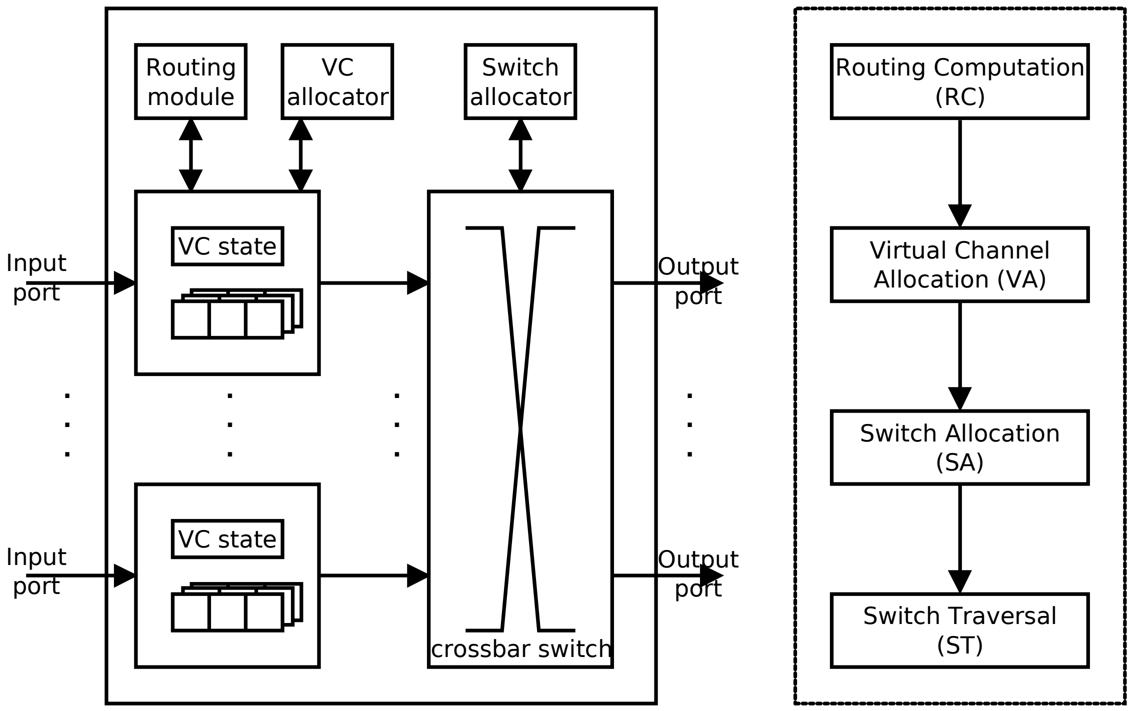

Fig. 8 Typical router architecture (left) and router operation (right)

As shown in the above figure, the router data path consists of buffers and a (crossbar) switch. The input buffers store flits while they are waiting to be forwarded to the next NoC node. When virtual channels are used, there are multiple input buffers for each physical channel so that flits can flow as if there are multiple virtual channels. When a flit is ready to move, the (crossbar) switch connects an input buffer to an appropriate output channel. Three modules are used for controlling the router data path: a routing module, a virtual channel (VC) allocator, and a switch allocator. These control modules determine the next hop (NoC node), the output virtual channel, and respectively when the switch is available for each flit.

The router operation mode has four phases: Routing Computation (RC), Virtual channel Allocation (VA), Switch Allocation (SA) and Switch Traversal (ST). These phases often represent one to four pipeline stages in modern virtual channel routers. When a head flit arrives at an input channel, the router stores it in the buffer for its corresponding virtual channel and determines the next hop for the packet (RC phase). Knowing the next NoC node, the router then allocates an output virtual channel (VA phase). Finally, the flit competes for the switch (SA phase) and moves to the output port (ST phase). Optionally, buffers for the output ports may be used. If so, the flits will stay in these buffers until the channel is ready to transmit them. If output buffering is not available, the flits will not be allowed to traverse the switch until the output channel is available.

2.3.4 Routing Algorithms

In this section, we briefly present (based on [24] [25] [26]) some of the most common routing algorithms. A routing algorithm has the purpose of establishing the path taken by a packet, from where it was injected into the network, the source node, up to its destination node. Before presenting different routing algorithms, we present the taxonomy for routing protocols.

2.3.4.1 Taxonomy for Routing Protocols

We adopt the routing protocols taxonomy from [26] with only a few minor observations. It can be seen that routing mechanisms may be classified based on many factors.

Based on how many destinations a packet has, the routing may be *unicast* (single destination) or *multicast* (multiple destinations).

The routing decisions can be taken by a single controller. In this case the routing is *centralized*. When the routing path is determined at the source node, before the packet is injected, we have *source routing*. The routing path may also be determined while the packet travels from node to node. In such cases, the routing is *distributed*. Hybrids of these three types of routing decisions are also possible and they are included in *multiphase routing*.

A routing algorithm may be implemented using a routing table or a finite state machine. Table lookup means searching for a routing path into the routing table. Finite state machine allows routing decisions to be taken at runtime.

The adaptivity criterion is the one that we are going to focus on in this section. *Deterministic (non-adaptive) routing* means that the same path is chosen every time a packet must be routed between the same (source, destination) node pair. *Adaptive routing* uses information about the network state and thus, it may provide different paths at different moments. *Oblivious routing* is another type of routing adaptivity. It is not presented in the cited taxonomy, but it is described in [25]. It does not use information about network’s state and it also does not provide the same routing path every time. Oblivious routing uses a stochastic mechanism to determine the path. For example, a random algorithm that uniformly distributes the traffic across all of the available paths implements an oblivious algorithm. *Oblivious routing* may be seen as a hybrid between deterministic and adaptive routing.

Progressiveness refers to how the channel is reserved in order to route a packet forward. With *progressive* routing a new channel is reserved at each new routing operation by moving the packet header forward. *Backtracking* routing allows the packet header to return, releasing the reserved channel. A backtracking protocol considers that it is better to search for alternative paths than to wait for a channel to become available. If the channel is faulty, it will remain unavailable until it is repaired. Potential paths are searched in a depth-first manner by probing the network with a header flit. When the header cannot go forward, it returns to the last acquired channel, releases it, and it continues its search from there.

Minimality is related to the length of the routing paths. A *minimal (profitable)* routing always generates routing paths of minimum length. *Nonminimal (misrouting)* means that the paths may be longer than the minimum path (for example, to avoid network congestion).

Routing a packet can take into consideration all of the available paths (*complete, fully adaptive routing*) or just a subset of them (*partially adaptive*).

2.3.4.2 Dimension-Order Routing

Dimension-Order Routing (DOR) [25] is a deterministic routing protocol used for k-ary d-cube networks (like meshes and tori). The digits of the destination address are seen as a radix-*k* *d*-digits number and they are used one at a time to direct the packet. Each digit is used to select a node in a given dimension. DOR is also a minimal routing algorithm. Thus, at each step the shortest direction is selected: a torus may be traversed either from left to right or from right to left and, respectively, from up to down or from down to up.

We find the preferred direction in each dimension by computing the relative address for each digit *i* of the source and destination addresses:

$$m_i = d_i - s_i \bmod k$$

$$\Delta_i = m_i - \begin{cases} 0, m_i \le \dfrac{k}{2} \\ k, othewise \end{cases}$$

The preferred direction for dimension *i* is then computed like:

$$D_i = \begin{cases} 0, |\Delta_i| = \dfrac{k}{2} \\ sign(\Delta_i), otherwise \end{cases}$$

After computing the preferred direction vector ($\vec{D} = \overrightarrow{D_0 D_1 ... D_d}$), the packet will be routed in one dimension at a time. It travels in a dimension until it reaches the same coordinate like the destination (in that dimension).

Dimension-Order Routing is a very simple algorithm. It is used very much with mesh and torus networks because it is easy to implement and because it is deadlock-free (except when d = 1 – one dimension). DOR is also a minimal-path routing algorithm.

This algorithm is used for hypercubes as well. In this case it is named *e-cube routing*. For bidimensional meshes or tori it has two variants: XY and YX. With XY routing the X (horizontal) direction is routed first. With YX routing, the packet travels first in vertical direction.

2.3.4.3 Destination-Tag Routing

Destination-Tag Routing is a deterministic algorithm, like Dimension-Order Routing but for k-ary n-fly butterflies. The destination address is viewed as an *n*-digits radix-*k* number. This is the only information used to route a packet. The source address is not used for routing. Each digit of the destination address is used to select the output port at each step of the route.

2.3.4.4 Valiant Routing

This algorithm may be applied to any topology that has at least one path between each pair of nodes (*connected topology*). The principle is to route packets through an intermediate node that is randomly chosen. Routing the packet from the source to the intermediate node and from the intermediate node to the destination can be done with any routing algorithm. The random selection of the intermediate node makes Valiant’s algorithm to be an oblivious routing mechanism. Because of the randomly selected node, this algorithm is able to balance the network load better than a deterministic algorithm. However, this algorithm is not minimal.

2.3.4.5 Minimal Oblivious Routing

Such an algorithm tries to achieve good load balance without sacrificing the locality. The intermediate node is still chosen randomly but, with the restriction that the resulted path is minimal.

A minimal version of Valiant’s algorithm for k-ary d-cube network can be implemented by restricting the intermediate node to reside in the minimal quadrant determined by the source and destination nodes [25]. The minimal quadrant is the smallest d-dimensional sub-network, cornered by the source and the destination nodes.

2.3.4.6 Turn Model Routing

Any routing algorithm faces two major problems: deadlock and livelock. *Deadlock* is a situation when a packet waits for an event that cannot occur, for example when no message can advance toward its destination because the queues of the message system are full and each is waiting for another to make resources available. *Livelock* means a case when routing a packet never leads to its destination (it can only occur with adaptive non-minimal routing).

Deadlock-free routing may be achieved by restricting the possible routes that may be followed by packets. The purpose is to eliminate cycles. Dimension-Order Routing for example, is deadlock-free because packets are not allowed to cycle. However, DOR significantly reduces the path diversity: in a 2D mesh, from eight possible directions, four of them are restricted. Freedom from livelock is also ensured because DOR is minimal.

The turn model [30] provides a way of achieving deadlock- and livelock-free routing algorithms that are partially adaptive. Like in the case of DOR, the turn model restricts the routing algorithm to take some particular directions, turns. As compared to DOR, only a quarter of the possible turns are prohibited in k-ary d-cube networks.
What the turn model does is to restrict one turn. This effectively eliminates the cycles and leads to three possible routing algorithms, which are deadlock- and livelock-free [30]: west-first, north-last and negative-first.

With *west-first routing*, a packet must always travel west first. This is because it is disallowed to turn back west. *North-last routing* does not allow a packet to change direction once it starts traveling up. Finally, *negative-first routing* is characterized by not allowing turns from a positive direction to a negative one.

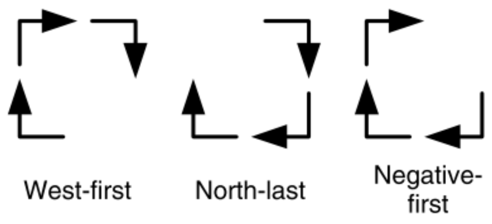

Fig. 9 Routing algorithms generated by the turn model

Obviously, these three algorithms may be implemented as minimal or non-minimal.

P-cube routing is another routing algorithm based on the turn model that applies to hypercubes. Let us consider that we want to route a packet from node *s* to node *d* in a binary n-cube. Thus, the source and destination nodes can be written as $s = s_{n-1}s_{n-2}...s_0$ and $d = d_{n-1}d_{n-2}...d_0$, where $s_i, d_i \in \{0,1\}, \forall i = \overline{0, n-1}$.The algorithm builds set E as $E = \{i \mid s_i \neq d_i, \forall i = \overline{0, n-1}\}$.The size of E is the Hamming distance between *s* and *d*. This set is then divided into the disjoint sets E$_0$ and E$_1$: $E_0 = \{i \mid i \in E, s_i = 0, d_i = 1\}$, $E_1 = \{i \mid i \in E, s_i = 1, d_i = 0\}$. P-cube routing has two phases: first, the packet is routed in the dimensions of E$_0$, in any order, second the packet travels in E$_1$'s dimensions (in any order). This algorithm is deadlock- and livelock- free [30].

2.3.4.7 Odd-even routing

Another partially adaptive routing protocol that is deadlock-free is the odd-even [31] turn model. Compared to the previous turn model, its main advantage is that it provides a more even routing adaptiveness for non-uniform traffic patterns.

The odd-even routing restricts the east to north and east to south turns at any nodes located in an even column. For the odd columns, the north to west and south to west turn are prohibited.

2.3.4.8 Torus routing

As we have mentioned earlier, deadlock freedom can be ensured by restricting the routing algorithm to use some paths, under certain conditions. Another way to prevent deadlocks from occurring is to use *virtual channels*. A virtual channel is a group of multiple buffers associated to the same physical channel. Using virtual channels, cycles can be broken [24].

The torus routing protocol uses virtual channels to avoid deadlocks. It is used with tori networks and employs Dimension-Order Routing with dateline classes to each dimension. A dateline is a conceptual line across a channel of a ring network (or within a single dimension of a torus). Each time a packet crosses a specified dateline for each dimension it is routed through another virtual channel. Dateline classes basically transform a torus into a mesh.

2.3.4.9 Other Routing Algorithms

Obviously, there are a lot more routing algorithms available in literature. For example, Planar adaptive routing [32] applies to *n* dimensional meshes, tori and hypercubes. The main idea is to provide adaptivity in only two dimensions at a time. Thus, from the point of view of this routing algorithm, the topology is seen as a series of bidimensional planes. Another example is Duato's protocol [26], which starting from a deterministic or partially adaptive routing mechanism generates a fully adaptive routing protocol.

2.3.5 Switching Mechanisms

There are three layers which must be distinguished between in the operation of interconnection networks [26]: the physical layer, the switching layer and the routing layer. The physical layer contains the link-level protocols (e.g.: X.25, HDLC) used for transmitting packets and for managing the channels between adjacent nodes. The switching layer offers mechanisms for forwarding packets across the network, by making use of the protocols from the physical layer. Finally, the routing layer has the responsibility of establishing a path through the network for each message.

The switching technique determines when and how switches are set to connect router inputs to outputs and also the moment when a message may be transferred across the network. There are basically two switching techniques: circuit switching and packet switching.

In **circuit switching**, a physical path from the source to the destination is reserved prior the message is transferred through the network. This path will be kept reserved until all the message reaches the destination.

With **packet switching**, a message is divided into smaller parts called packets. Each packet is routed separately from source to destination. No path is reserved until the whole message is routed. The first few bytes of a packet form the logical header of the packet. This typically contains routing and control information.

When all the packet's flits are buffered at each intermediate node before they are sent forward, this switching technique is called **Store-And-Forward (SAF) switching**. Note that packet switching and store-and-forward switching denote the same switching technique [26].

However, it is not necessarily to wait for the whole packet to arrive before start sending the already arrived flits forward through the network. Flits can be forwarded as soon as the routing was performed and an output buffer is free. The data does not even need to be buffered at the output and can cut through to the input of the next router, before the whole packet was received at the current router. This is another basic switching technique, called **Virtual Cut-Through (VCT) switching**. This technique determines messages to be pipelined in the interconnection network, and it works with flits, as compared to store-and-forward switching, which works with packages. Compared to SAF switching, VCT switching has the advantage that it can send messages faster. However, it has the disadvantage that it can block entire routing paths in the network when a message gets blocked somewhere in an intermediate node. In such a case, VCT switching falls back to SAF switching: the whole data will get buffered at the node where the head flit currently is. The performance of VCT reduces to the performance of SAF at high network loads. Big buffers are still required for VCT switching (since it can transform into SAF under high loads). Another approach would be to buffer the flits in the nodes where they currently are (and not to buffer all of them at the node where the head flit is). This switching technique is called **wormhole switching**. Wormhole switching has thus, the advantage that it allows the network to use small buffers. This makes it the preferred switching technique for Networks-on-Chip.

2.4 Common Topologies and Routing Protocols

After a brief presentation of some of the most popular network topologies and routing protocols, we focus onto the Network-on-Chip field. We show how NoCs are usually topologically organized and which are the most common routing protocols used.

According to [33], meshes and tori are the topologies which are used in more than 60% of the papers currently published in the Network-on-Chip research area. The rest of the percentage is taken by fat-trees and crossbars, in equal proportion, followed by rings. This study also concludes that there is no single fixed topology that clearly outperforms all the others. This is because the NoC's performance strongly depends on the application which runs on it. Therefore, it is useful to research the relation between the application mapping problem and network topologies.

It is also shown in [33] that deterministic routing is used in 70% of the cases. Hybrid routing techniques, like DyAD [34] or deflective routing were also proposed. DyAD automatically switches between Dimension-Order Routing and odd-even routing [35]. Deflective routing avoids network hotspots by adaptively misrouting packets.

The NoC survey from [36] concludes that XY routing is mostly used in Network-on-Chip research. It also confirms that mesh and torus topologies are the most researched ones, followed by ring, butterfly, octagon and even irregular networks.

An evaluation of the mesh and torus topologies, using different routing protocols is performed in [37]. Because the torus has long wrap around links, the torus is considered to be folded[3]. The Network-on-Chip is evaluated in terms of latency, bandwidth, power consumption and power/throughput ratio. The following routing protocols are used: XY, odd-even [35], negative-first [30] and Duato [26]. It is shown that the XY routing protocol is the best routing technique in a mesh. This is not the case with the torus: XY routing yields the highest power consumption and power/throughput ratio. The more adaptive the routing protocol is, the lower the power consumption of the torus is. This is because an adaptive algorithm makes better use of the wrap around links. The authors conclude that a torus topology is better than a mesh when network latency is the objective. But, in terms of power consumption, the mesh topology gives better results than the torus. However, this study is performed only with two stochastic traffic patterns: uniform and hotspot. No traffic patterns from real applications are used. Another drawback is given by the fact that NoC scalability is not addressed: the mesh and torus have a fixed 4 x 4 size. It would also be interesting to find out how these topologies, with different routing algorithms, behave from a multi-objective point of view. For example, rather than evaluating the ratio between power and throughput, it would be useful to determine a tradeoff among two or more objectives.

Topology scaling is taken into account in [38]. In this paper, multiple topologies are evaluated in terms of power consumption and technology scaling. The two objectives were chosen because the topology has a severe impact on the power consumption and

[3] A folded torus has all its links equal in length

because a topology that is optimal at a current integration technology may lose its optimality after some technology generations. This research starts with 2D mesh and torus topologies. Then, high-dimensional meshes and tori are also studied. Other topologies included in this research are hierarchical meshes and tori and express cubes. A hierarchical mesh/torus, *H-v*, has the property that channels connect not just adjacent nodes but also v-nodes away neighbors in each dimension. These channels are naturally longer and they are called express channels. Similarly, the express cube is a hierarchical network, noted as *E-v*. The metric used is the average energy required by a flit to reach its destination. This is obtained with an analytical model which is general enough so that it may be applied to all the topologies. The model allows an energy efficiency comparison of the network topologies, across different technologies. A uniform random traffic pattern is assumed to be applied to the networks. This is helpful for the analytical model because the average hop count can be easily computed. Based on this model, the authors find that hierarchical and express cubes save more energy than high-dimensional tori. The authors also try to evaluate the topologies under different traffic patterns. To this end, they resort to network simulation for obtaining the average hop count. Then, the same analytical model is applied to find out the energy consumption. Simulation is performed with RSIM, on a 10 x 10 network. Several SPLASH benchmarks are run. Their traffic patterns are collected and the average hop count is computed. Simulation has the advantage of allowing applying the analytical model on real applications. However, scalability is still an issue since only 10 x 10 networks (2D torus, H-2, H-3, H-4, E-2, E-3, E-4) are analyzed.

The design space of Network-on –Chip topologies is explored in conjunction with the application mapping problem in [39]. A recursive bi-partitioning heuristic algorithm is used to map the IP cores onto NoC tiles. The IP cores execute a real application (MPEG4 decoder) and other several synthetic traffic patterns, all of them being described through Communication Task Graphs. The topologies used in this research are: 2D mesh, ring, Spidergon and crossbar. The Spidergon is a topology similar to a bidirectional ring but, it also has links that connect the opposite nodes. Only deterministic routing is used with all topologies (Dimension Order Routing). It is shown that the ring performs generally worse than the other topologies because it has fewer channels. At the other extreme is the crossbar, which performs better than all the considered topologies. Spidergon is more scalable than a mesh. An interesting result is that for the MPEG4 application, the crossbar and mesh topologies behave worse than Spidergon and even than the ring. The metric considered in this paper is network throughput.

2.5 Research Problems

Marculescu et al. [3] identified the following major research problems in the field of Networks-on-Chip:

1. traffic modeling and benchmarking;
2. application scheduling;
3. application mapping;
4. routing;
5. switching;
6. Quality of Service (QoS) and congestion control;
7. power and thermal management;

8. reliability and fault tolerance;
9. topology design;
10. router design;
11. network channel design;
12. floorplanning and layout design;
13. clocking and power distribution;
14. analysis and simulation and
15. prototyping, testing and verification.

These NoC research problems determine a very complex and difficult to explore design space, which has four dimensions: *application characterization*, *communication paradigm*, *communication infrastructure* and *analysis and solution evaluation*.

Application characterization means identifying the target applications and their associated traffic patterns, and performing application mapping and scheduling. Traffic modeling and benchmarking determines effective architectures, and application partitioning allows the optimization of the NoC performance and power consumption. The following three research problems belong to the application characterization category: traffic modeling and benchmarking, application mapping and application scheduling. This thesis focuses on the application mapping problem, which will be presented in Chapter 3. There we will also briefly describe the scheduling problem by showing how it interacts with the mapping problem.

The communication paradigm for Network-on-Chips can be improved by knowing the application mapping and traffic characteristics. The effectiveness of the NoC communication infrastructure depends on the routing and switching algorithms, Quality of Service and congestion control, power and thermal management, and techniques for increased reliability and fault tolerance.

The communication infrastructure is influenced by topology, router, channel and clocking strategies that can be used for the NoC design. Using a global clock becomes difficult due to smaller process geometries, larger wire delays and higher levels of integration of multiple cores on a chip. Also, the design of the floorplan and layout of the network architecture becomes mandatory for irregular networks (in terms of topology and tiles).

Finally, Network-on-Chip evaluation and validation are necessary steps for ensuring NoC's compliance with the initial specifications.

2.6 Summary

This chapter presented a theoretical background for Network-on-Chip architectures. They offer the scalability that multicores and manycores require but, they have the big disadvantage of being resource constrained. As we showed, there are many research problems concerning NoC designs. One of them is Network-on-Chip application mapping. This is the main topic of this thesis and we introduce it in the next chapter.

The theory presented in this chapter and in Chapter 3 is strictly subordinated to the scope and objectives of this PhD thesis. In other words, we did not intend a text book like, exhaustive presentation, and we also do not claim a general valid rigor.

“The greater the difficulty, the more the glory in surmounting it.”

Epicurus

3 Network-on-Chip Application Mapping

In the previous chapter we showed that the Network-on-Chip research field deals with fifteen major problems. We will focus next only on one of them, namely Network-on-Chip application mapping.

We begin by defining the Network-on-Chip application problem and by showing that it is an NP-hard problem. Then we show this problem is directly connected to other two NoC research problems: scheduling and routing. We present the similarities between scheduling and mapping and we introduce two state of the art NoC scheduling algorithms.

We then propose a taxonomy for Network-on-Chip application mapping algorithms and we describe some of the state of the art algorithms for NoC mapping.

3.1 The Network-on-Chip Application Mapping Problem

The design flow of a Network-on-Chip architecture for a specific application implies the following three major steps [40]:

1. dividing the application into a graph of concurrent tasks (threads);
2. assigning and scheduling the application tasks to the available IP cores;
3. mapping each IP to a NoC tile, so that the metrics of interest are optimized.

The Network-on-Chip *application mapping problem* was formulated in [40] as the *topological placement of the IPs onto the on-chip tiles*. It is an instance of the *quadratic assignment problem*, which is proven to be an NP-hard problem [41]. The search space increases factorially with the system size. For example, a NoC with 8x8 tiles theoretically allows 64! mappings. Theoretically, mapping *N* IP cores onto *M* network nodes ($N \le M$) implies $\frac{M!}{(M-N)!}$ possible core arrangements on the NoC nodes. When the number of IP cores is identical to the number of network nodes ($N = M$), the number of possible mappings becomes $M!$. This is therefore a permutation, combinatorial, problem. It directly affects NoC’s performance in terms of latency, throughput, power consumption, energy etc. This is because typical network metrics like latency and power are directly proportional to distance.

A typical mapping cost function [42] is:

$Cost(\pi \in P) = \sum_{l \in L} BW_l = \sum_{1 \le i, j \le N} [bw_{i->j} \cdot Dist(i, j)]$, where π is a particular mapping from *P*, the set of all possible mappings. *L* is the set of NoC links which are used by the application. BW_l is the bandwidth delivered over link *l*. *Dist (i , j)* is the distance between nodes *i* and *j* (hop count) and $bw_{i->j}$ is the bandwidth required by node *i* for communicating its data to node *j*.

Consider for example the following two mappings π_1 and π_2. They consist of six processing elements placed onto a 2D mesh NoC. PE2 communicates 30 bits/s to PE6 and PE4 100 bits/s to PE3. We are interested to evaluate the two mappings using the above cost function.

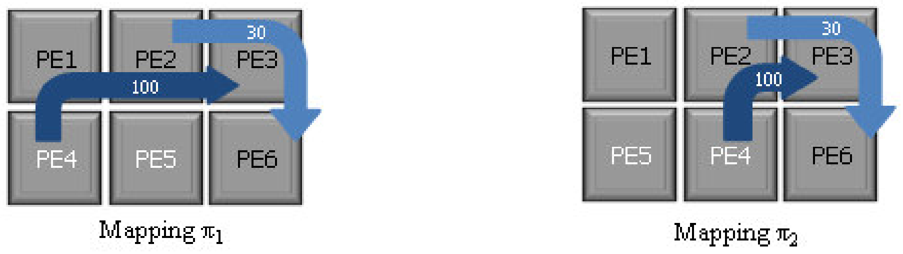

Fig. 10 Example of two mappings π1 and π2

For the first mapping, we have:

$$Cost(\pi_1) = bw(PE_2 \to PE_6) \cdot Dist(PE_2 \to PE_6) + bw(PE_4 \to PE_3) \cdot Dist(PE_4 \to PE_3)$$

$$Cost(\pi_1) = 30 \cdot Dist(PE_2 \to PE_6) + 100 \cdot Dist(PE_4 \to PE_3)$$

$$Cost(\pi_1) = 30 \cdot 2 + 100 \cdot 3 = 360$$

Similarly, for the second mapping we get: $Cost(\pi_2) = 30 \cdot 2 + 100 \cdot 2 = 260$. Notice that the only difference between the two mappings is the placement of PE4 and PE5. In the second mapping, PE4 is closer to PE3. Because of this, given the above conditions, mapping π_2 is better than mapping π_1.

In the field of embedded systems, an application is typically described through a **Communication Task Graph (CTG)**. A CTG is defined in [43] as a directed *acyclic* graph, $G' = G'(T, D)$, where each vertex, $t_i \in T$, is a an application task (a computational module in the application). A task typically has assigned to it information like: *execution time* on every type of Processing Element (PE) available for the NoC, *energy consumption* (when assigned to a certain PE), *task deadline* (the time until the task associated with the CTG node must complete its execution [44]), etc. A directed arc between t_i and t_j, is noted as $d_{i,j} \in D$ and has a value associated to it, which represents the communication volume ($v(d_{i,j})$, usually expressed in bits) exchanged between tasks t_i and t_j. Each arc shows both data and control dependencies. A *data dependency* marks that there is a communication between the two tasks (t_i and t_j) [45]. A *control dependency* indicates that a task cannot be executed before its predecessor tasks are not completely executed [45]. Thus, a data dependency is basically an undirected arc between two tasks. When such an arc is present between two tasks, it means that the two tasks are communicating. When the arc is directed, the arc's arrow shows a control dependency between the two tasks.

Note that a CTG is defined as an acyclic directed graph. However, in reality, the tasks of an application may exhibit a communication pattern which creates loops. Loops are not usually modeled with a CTG because of real-time considerations. For hard real-time applications, unbounded loops are avoided because they do not allow bounds on graph execution times. It is not possible to guarantee that the worst-case communication volume path can be executed under the specified deadline. It is preferred that deadlines can be assigned to tasks and a CTG typically has a period attached to it. The CTG can therefore be reiterated after a certain amount of time [46].

The Directed Acyclic Graph (DAG) model of a parallel program is used in [47] to address the scheduling problem. In our humble opinion, the Network-on-Chip research community adopted the DAG model, from the scheduling research area, with the name of Communication Task Graph.

A task is defined in [47] as a set of instructions that are executed sequentially, on the same processor, without preemption. The task is a node in the DAG. It may have a weight attached to it, which represents the computational cost. However, a CTG does not weight the nodes because it is only communication oriented.

The DAG arcs model the communication messages and the precedence constraints between tasks. The arcs are weighted with communication costs. If two communicating tasks are assigned to the same processor, their communication cost will be neglected. The precedence constraints are what make the graph to be directed. They show how communication flows among tasks. A node is not allowed to start its execution until it receives all the messages from its parent nodes.

Program loops cannot be explicitly represented using the DAG model. Conditional branches are not included as well. According to [47], including loops and branches in the DAG model is an implicitly difficult problem. Additionally, many numerical applications (e.g.: Fast Fourier Transform) contain loops with a number of iterations known at compile time. For such programs, techniques like loop unrolling [6] can be applied. This way, one or more loop iterations can form a task. Also, large classes of numerical applications and data-flow programs have very few conditional branches.

Scheduling a DAG with probabilistic branches and loops was addressed in [48]. Each graph arc has a probability that the child node will be executed immediately after the parent node. Scheduling DAGs with conditional branches is made in [49] by using, beside the precedence graph, a branch graph, too. Although DAG models that deal with loops and/or conditional branches have been proposed, the Network-on-Chip research community adopted the simple DAG model, without loops and conditional branches. Therefore, a CTG does not model program loops nor branches. It focuses on the communications among the tasks of data-flow programs.

The acyclic property of a Communication Task Graph is dropped at a coarser level, denoted by an **Application Characterization Graph (APCG)**. An APCG models an application at the level of Intellectual Property (IP) cores and it is defined in [43] as: a directed graph, $G = G(C, A)$, where each vertex $c_i \in C$ represents an IP core and each directed arc, $a_{i,j} \in A$, characterizes the communication between cores c_i and c_j. This may be application specific information like communication volume. It can also be design constraints, like communication bandwidth, area of IP cores, etc. As in the case of a CTG, a directed arc of an APCG shows data and control dependencies. But, compared to a CTG, an APCG allows cycles. For example, we can have a bidirectional communication between two cores. Note that loops are still not desired in APCGs because of real-time constraints. It is often preferred to transform a directed graph into a Directed Acyclic Graph (DAG) [50]. This allows worst-case execution time analysis, which makes the APCG usable in hard real-time systems as well.

An Application Characterization Graph is obtained from a Communication Task Graph by scheduling the tasks on available IP cores.

Having the definitions for a CTG and an APCG, we can now illustrate the application mapping problem for NoCs using the following figure.

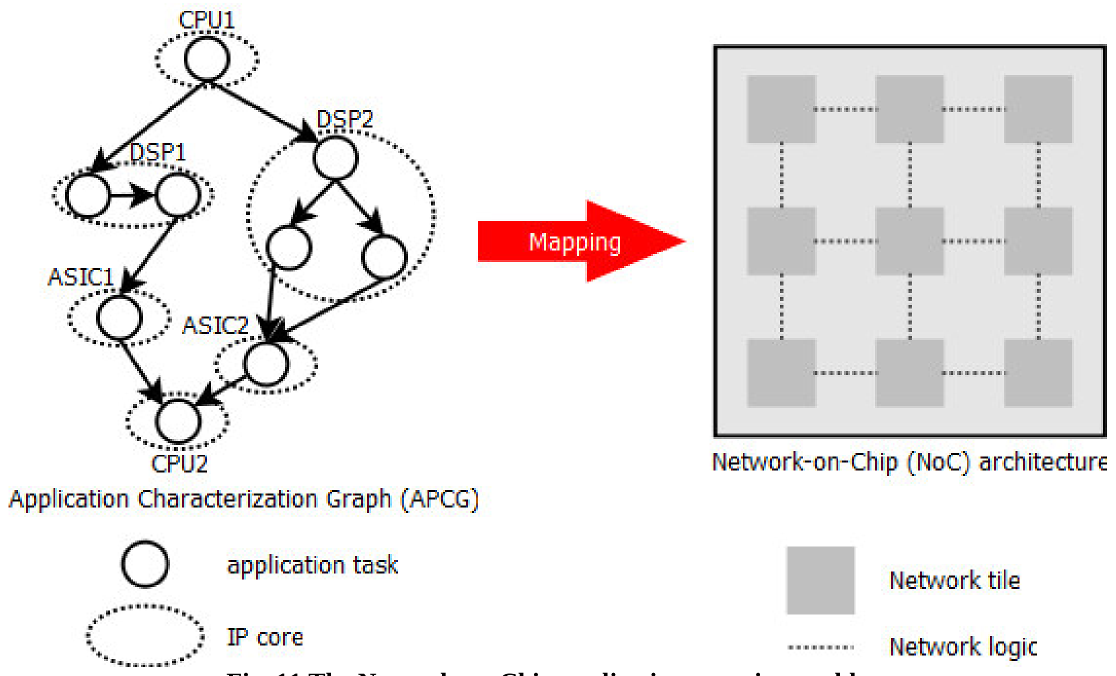

Fig. 11 The Network-on-Chip application mapping problem

Obviously, the NP-hard problem cannot be solved by means of exhaustive search. Heuristic algorithms [51] are employed with the purpose of finding the best topological placement of cores onto network nodes. The objective is to optimize network latency, its energy consumption, etc. Multiple objectives may be followed at the same time, too.

We show next that Network-on-Chip application mapping interacts directly with other two NoC research problems (see section 2.5): routing and application scheduling.

3.1.1 Application Mapping and Routing Problems

While a good mapping of cores onto network nodes can lead to energy savings, the routes used by the cores to communicate can have a great impact on the NoC's performance. The best topological placement of cores onto nodes is not enough to account for the performance of the network. The next figure shows an example where two minimal routes are available between the top-left and bottom-right tiles of a 2D mesh NoC. Choosing the proper route can increase the performance of the network.

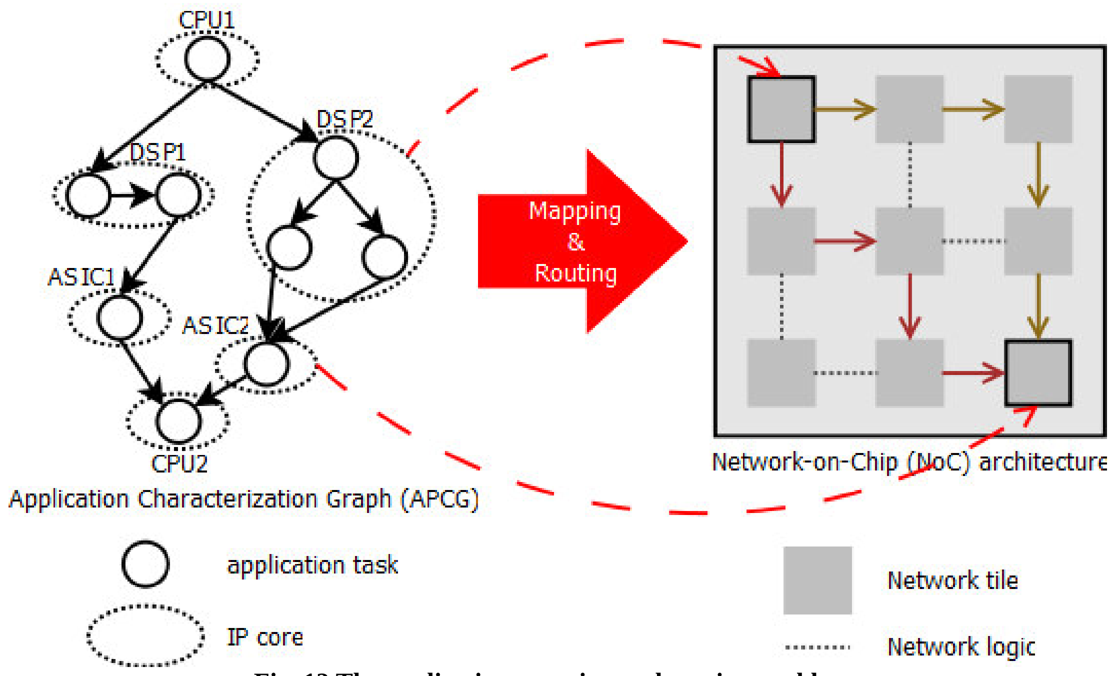

Fig. 12 The application mapping and routing problems

This shows that the **application mapping problem is tightly connected to the routing problem**. Usually it is not necessarily to generate routing paths when placing IP cores onto NoC tiles. A mapping algorithm may simply consider that the NoC architecture is using a particular routing protocol (like XY routing in [40]). However, routing information can help at obtaining a better mapping [52].

3.1.2 Application Mapping and Scheduling Problems

Before mapping the IP cores onto the Network-on-Chip tiles, the application's tasks and communication transactions must be assigned to the NoC resources. Additionally, the tasks' execution order must be established. This is called the **scheduling problem** for NoC architectures [53] and is an NP-hard problem as well. It has a considerable influence on the energy consumed by the IP cores when computing, due to their heterogeneity. For example, a DSP core may consume less energy than a general purpose processor when computing a Fast Fourier Transform. Also, the communication energy consumption of the NoC architecture is affected by the task assignment (because of the routing paths).

Therefore, the application mapping problem is connected to the scheduling and routing problems. The following figure illustrates this fact.

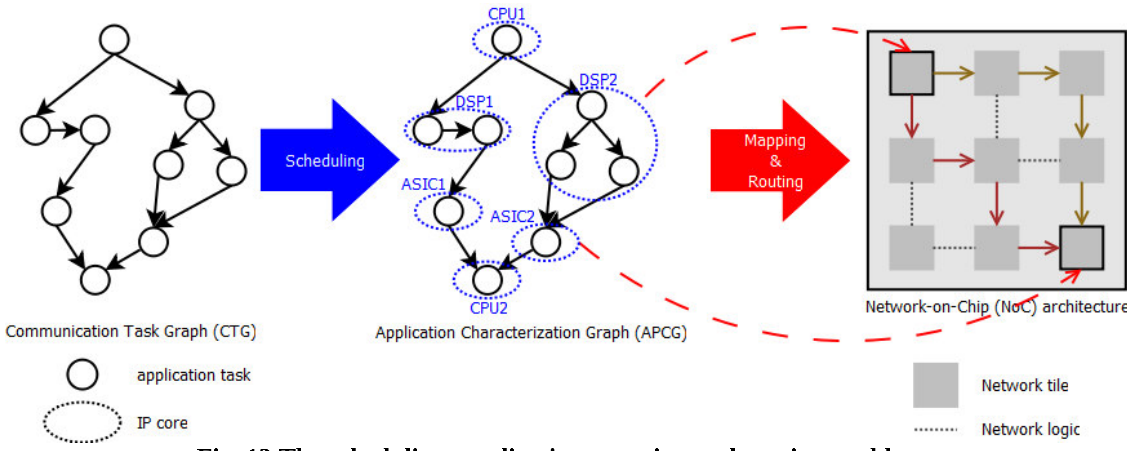
Communication Task Graph (CTG)
Scheduling
CPU1
DSP1
DSP2
ASIC1
ASIC2
CPU2
Application Characterization Graph (APCG)
Mapping & Routing
Network-on-Chip (NoC) architecture
application task
IP core
Network tile
Network logic

Fig. 13 The scheduling, application mapping and routing problems

An application is described through its Communication Task Graph. A scheduling algorithm is then used to assign application tasks (threads) to available IP cores and to specify their order of execution. After the scheduling step, the Application Characterization Graph is obtained. Then, using a mapping algorithm (which may generate the routing function as well), the IP cores are topologically placed onto the NoC tiles.

We observe that both scheduling and mapping algorithms for Networks-on-Chip have similar objectives. Increasing the performance and decreasing the energy consumption of a NoC, for a particular application, are two optimizations typically made by such algorithms.

Ideally, both scheduling and mapping problems should be treated together. In other words “scheduling” means mapping the application’s tasks onto the available IP cores, and “mapping” means mapping the IP cores onto the available NoC nodes. Therefore, both scheduling and mapping problems deal with application mapping onto a Network-on-Chip.

Nevertheless, because of the NP-hard complexity of the problem, mapping applications onto NoCs is divided in a two-step process: scheduling, followed by mapping.

We present next a two-step genetic algorithm which deals with the scheduling problem for Networks-on-Chip. An approach that deals with both scheduling and mapping problems is proposed in [53] where an energy-aware algorithm is presented. However, because they are both NP-hard problems, our main focus will be on the mapping problem. Therefore, we continue by proposing a classification of application mapping algorithms. Then, we present several application mapping algorithms, classified according to our taxonomy. We present here only an overview of the algorithms. We give detailed descriptions in [54].

3.1.3 Scheduling Algorithms for Network-on-Chip Architectures

3.1.3.1 Two-step, Single Objective Delay-Aware Genetic Algorithm

A two-step genetic algorithm is used in [55] for assigning the tasks of an application to the IP cores placed onto a NoC that uses a 2D mesh topology. It is assumed that IP cores are already associated to the network nodes. Hence, the mapping problem is not considered. Techniques for task scheduling are also employed. The two-step genetic algorithm has a single objective: map the tasks of an application onto network nodes so that the overall *execution time* is minimized. The main steps of this algorithm are presented next.

In the first step, the application tasks are assigned to classes of IP cores. In the second step, the tasks are assigned to the IP core (from the selected class) which provides the highest performance. The last step of the algorithm involves scheduling techniques (like ASAP – As Soon As Possible, or ALAP – As Late As Possible).

The execution time is expressed through a mathematical model developed for the estimation of the delays from a 2D mesh Network-on-Chip architecture. Three types of delays are identified and used by the algorithm: system delay, edge delay and average edge delay. The system delay of a task graph represents the delay given by the execution of its critical path. It has two components: the execution time and the communication time. The communication is modeled initially at a coarser level, through the average edge delay. Then, the edge delay is used so that a finer level modeling of the communication is used.

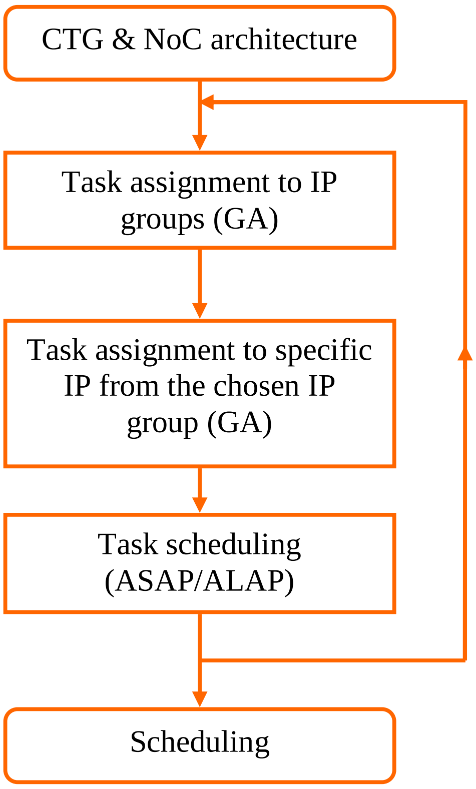
CTG & NoC architecture
Task assignment to IP groups (GA)
Task assignment to specific IP from the chosen IP group (GA)
Task scheduling (ASAP/ALAP)
Scheduling

Fig. 14 Overview of the two-step, single objective delay-aware genetic algorithm

Each of the two steps of this algorithm is a genetic algorithm. The first genetic algorithm computes the fitness as system delay, using the average edge delay. Its output is an assignment of tasks to classes of IP cores. It is used as input for the second genetic algorithm. This one computes the system delay using the edge delay metric. Its output is an assignment of tasks to a specific IP core (from the given IP core class).

Obviously, such a genetic algorithm could be made of only one step, by using from the beginning the exact edge delay, instead of the average edge delay. However, the authors argue that a single step genetic algorithm would be, either extremely slow, or it would not provide good results in a reasonable amount of time.

3.1.3.2 Energy-Aware Scheduling

The Energy-Aware Scheduling (EAS) algorithm [53] statically schedules the tasks and communication transactions of an application which is running on a Network-on-Chip with heterogeneous Processing Elements (PEs). The tasks are scheduled for execution under real-time constraints. The main idea is to give more slack (more time) to the tasks which have a higher impact on the Network-on-Chip energy consumption and performance. This algorithm is proposed for a NoC with an N *X* N 2D mesh topology, using XY routing. However, the authors claim that their algorithm can be adapted to any NoC architecture.

The following analytical model is used for calculating the average energy consumption for communicating a bit of data: $E_{bit}^{p_i,p_j} = n_{hops} E_{R_{bit}} + (n_{hops} - 1) E_{L_{bit}}$.

The Processing Element p_i sends one bit of data to PE p_j, which passes through n_{hops} network nodes in order to reach its destination. The energy consumed by a router and by a link is represented by $E_{R_{bit}}$ and $E_{L_{bit}}$ respectively.

The algorithm has two major steps. The first step allocates slack to all the tasks. A task scheduling priority is computed based on its execution time and the energy that a PE would require for executing it.

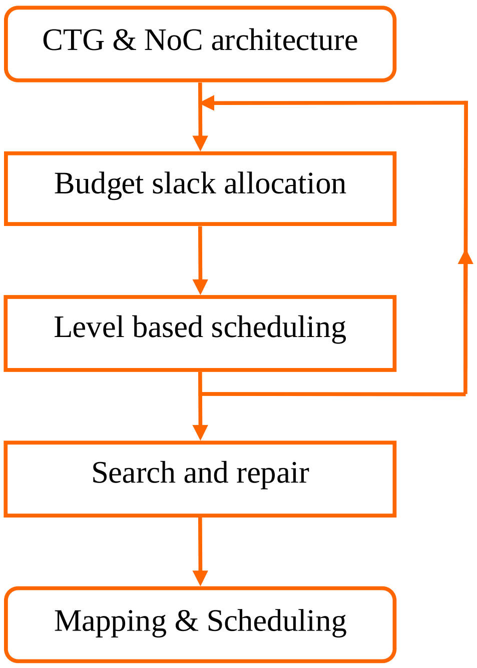

Fig. 15 Overview of the Energy-Aware Scheduling algorithm

The second step performs scheduling hierarchically. Starting from the root task of the CTG, each time the tasks that are considered for scheduling are those for which the tasks they depend on were previously scheduled. The purpose is that each task has a minimum earlier finish time less than its budgeted deadline. The EAS algorithm works only with those two steps. However, a third step is also employed so that the number of deadline misses may be further decreased.

Search and repair is a greedy approach with two working modes: Local Task Swapping (LTS) and Global Task Migration (GTM). The purpose of the LTS part is to reduce the deadline misses by scheduling the more critical tasks in front of the more uncritical tasks. This task reprioritization procedure is performed locally, at the level of each PE. Since the changes performed in the LTS mode are local, the communication and computation energy of the NoC are not affected.

The GTM part performs global changes. Such changes are required in the case of heavily loaded PEs. A critical task is migrated to another PE, which is selected in the order of its communication and computation energy consumption. Obviously, the GTM mode affects communication and computation energy of the NoC. It can lead to increasing the energy consumption but, the advantage is that deadline misses are reduced.

The performance of the EAS algorithm is compared to the one of an Earliest Deadline First (EDF) scheduler. The Communication Task Graphs (which represent the input of the algorithm) were obtained in three distinct ways: (1) randomly, using the TGFF tool [46], (2) from the E3S benchmark suite [56] and (3) from two real-world applications (an audio/video encoder and an audio/video decoder). Compared to EDF, EAS achieves significant energy savings for all applications.

3.2 Taxonomy for the Application Mapping Algorithms

An application mapping algorithm takes into consideration the characteristics of the application, and it has the purpose of finding the best placement of IP cores, onto the tiles of the Network-on-Chip architecture. Obviously, the application mapping algorithm must be aware of the NoC topology. The placement of the cores onto the network nodes can be made before the application starts to be executed and it cannot be changed afterwards. We call this **type of mapping** a *static* mapping. Obviously, the mapping process is iterative: multiple mappings are generated until the optimum mapping is found but, in case of static mapping, all the mappings are obtained before the application starts running. If the mapping of cores changes while the application runs, we have a *dynamic* mapping. This is typical for NoCs that are fault tolerant or application-adaptive. This kind of mapping could also lead to an increase of network performance and/or to a decrease in power consumption but, it is more difficult to implement (than static mapping is).

The factorial number of possible mappings can be decreased because it is very likely that not every mapping is feasible. This is because of the communication demands of the application and the hardware limitations of the underlying Network-on-Chip architecture. For example, consider that we have two communicating IP cores which require a bandwidth of *B* bytes/s. The NoC architecture may have some links that support such high bandwidth and other links that do not support it. In such a case, mapping the two IP cores so that they would require communicating over links that do not support the required bandwidth would generate an impractical mapping. The bandwidth requirement is an example of a **mapping constraint**. We define the mapping constraint (*MC*) as a restriction, derived from the requirements of the application and the characteristics of the Network-on-Chip architecture, imposed when associating IP cores to network nodes. Any mapping constraint may limit the size of the search space. An application mapping algorithm may or may not use one or more mapping constraints but, usually this should be an obvious thing to do because it would speed up the mapping algorithm. The difficulty of using a mapping constraint consists of having the means to evaluate if a mapping satisfies or not that constraint.

The application mapping algorithm explores the search tree of possible mappings and tries to find the best mapping (for a certain application and NoC architecture). In order to determine the best mapping, at least one **optimization goal** is required. Example of optimization goals can be: network performance, communication energy, power consumption, etc. Thus, a mapping algorithm may search for the best mappings by considering a *single objective* or even *multiple objectives*.

As we showed in Section 3.1.1, the mapping problem is also closely related to the routing problem. Any routing algorithm may be applied after the mapping has been done. However, if the mapping algorithm is not **routing aware**, it is possible that the best mapping does not actually provide the best network performance due to the fact that the routing paths were not considered when applying the optimization goals to the possible mappings. A mapping algorithm can thus, deal with identifying the routing paths for the mapped IP cores as well. The routing function can be *deterministic* or *adaptive*. Also, it should provide freedom from deadlock and livelock, and it may have other characteristics like being minimal.

To summarize, we have established that we have two types of application mapping algorithms: static and dynamic. Any mapping algorithm, whether static or dynamic has at least one optimization goal (single-objective or multi-objective). It may use (one or more) mapping constraints. Also, it may determine the routing function, during mapping. The routing can be deterministic or adaptive and it can have other properties like freedom form deadlock and others. We have thus four classification criteria:

mapping type	static	**optimization goals**	single objective
	dynamic		multiple objective
mapping constraints	with one or more mapping constraints	**routing awareness**	generates routes while mapping
	without any mapping constraint		does not generate routes

An application mapping algorithm can be static or dynamic. Either static or dynamic, the mapping algorithm can have a single objective (SO) or multiple objectives (MO) to optimize. Characteristics like using mapping constraints and being routing aware (RA) are optional and can be applied to any type of mapping algorithm (making it thus more specific).

Finally, we note that in [57], where the scheduling problem is also considered, the algorithms are classified as integrated or separated based on whether they treat NoC mapping and scheduling together or not. We consider this to be good classification criteria when including application scheduling, too. The algorithms presented in the above cited paper are for NoCs and for bus-based multiprocessor embedded systems. The NoC algorithms are classified only by whether they have routing awareness or not. The algorithms for bus-based systems are classified according to their optimization goal (energy minimization, handling soft real time constraints or memory awareness). Issues like mapping type and mapping constraints are also mentioned but they are not used as classification criteria. The single/multi objective (optimization goals) criterion is not included. Therefore, we consider our proposed taxonomy to be in accordance with the one from [57] but, more general and suitable.

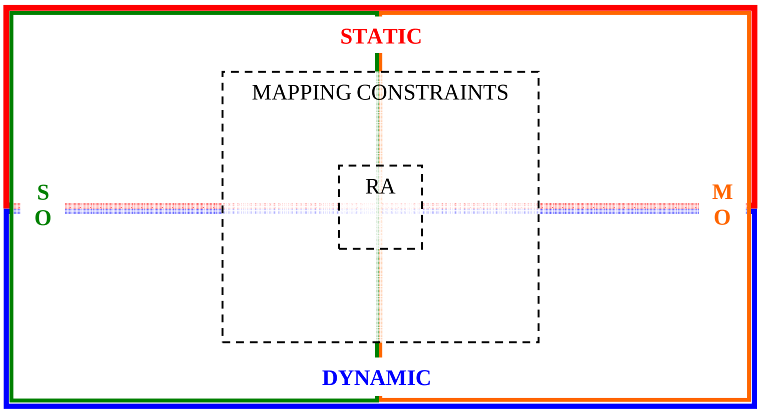

Fig. 16 Taxonomy for application mapping algorithms

3.3 Mapping Algorithms for Network-on-Chip Architectures

The following section briefly presents several of the most known Network-on-Chip application mapping algorithms available in literature. This is by far not an exhaustive list. A survey of algorithms for NoC application mapping and scheduling is available in [57]. It presents and compares some of the algorithms we present here and other algorithms, too. Another good overview paper about the NoC application problem is [42].

3.3.1 Simulated Annealing

Simulating Annealing (SA) [58] is a tree search technique that combines hill climbing [59] with a random walk in order to yield efficiency and also completeness. The hill climbing algorithm never makes downhill moves so it is guaranteed to be incomplete (because it can get stuck in a local maximum). On the other hand, moving to a successor tree node, chosen randomly from all the possible successors, is very inefficient but more complete. As we will see next, in the end, Simulated Annealing reduces to hill climbing.

The idea behind this algorithm comes from metallurgy, where annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low-energy crystalline state.

An analogy for simulated annealing could be made with dropping a ball on a non-uniform surface. We would like the ball to end up in the deepest place from our surface but, most likely, the ball will stop in a place which is deep but it is not the deepest one. If we shake the surface hard enough we will determine the ball to exit from the local minimum. However, we must not shake the surface very hard or it might leave the global minimum, too. The simulated annealing algorithm starts by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

Simulated annealing was proposed as an energy-aware mapping algorithm in [40], where it was compared to a branch and bound algorithm (see section 3.3.2). In [52], SA was extended so that it is also performance-aware. The analytical energy model presented in section 3.1.3.2 was used.

The advantages of Simulated Annealing are given by its ease of implementation, its applicability to many combinatorial optimization problems and the ability to give reasonably good solutions [60]. However, the parameters of the algorithm (like temperature) must be carefully chosen because SA can easily run for very long times until it gives a solution. It is shown in [40] that SA becomes practically infeasible as the problem size increases. However, the simulated annealing from [40] is a general implementation. Using domain- knowledge could potentially make SA faster and thus feasible.

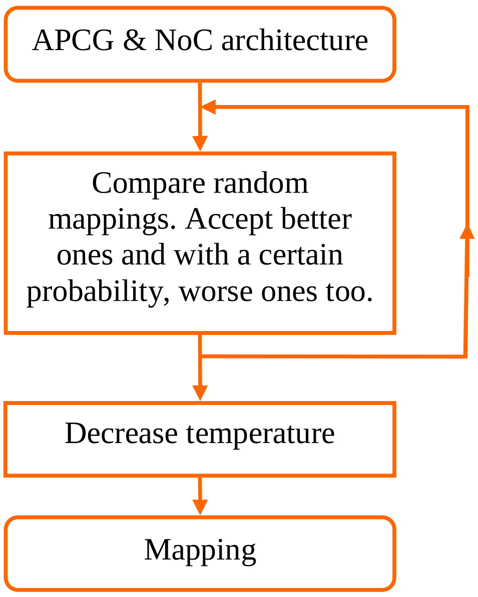
APCG & NoC architecture
Compare random mappings. Accept better ones and with a certain probability, worse ones too.
Decrease temperature
Mapping

Fig. 17 Overview of the Simulated Annealing algorithm

3.3.2 Branch and Bound

A Branch and Bound (BB) [51] algorithm is proposed in [40] for the mapping of IPs onto the NoC tiles. The mapping is energy-aware (the energy model from section 3.1.3.2 is used), the purpose being to optimize the total communication energy of the Network-on-Chip, while bandwidth constraints are satisfied. The algorithm is introduced for a 2D mesh network with *n x n* tiles, which uses the deterministic XY routing algorithm.

The algorithm builds a search tree where each node contains a (partial) mapping. Basically, the node from the search tree is an array. Each element from the array is an IP core. A value set for an element *i* from the array means that the NoC node identified by that value is assigned to the IP core with identifier *i*. Branch and Bound has two major steps which are iterated:

- **branch**: an unexpanded node is selected from the search tree; then, the next unmapped IP core (given by the selected node) is enumeratively assigned to the unoccupied tiles, and the child nodes are generated;
- **bound**: the new child node is checked if it is legal (i.e. it satisfies bandwidth constraints) and also if it can generate the best leaf nodes. If not, the node is not expanded further.

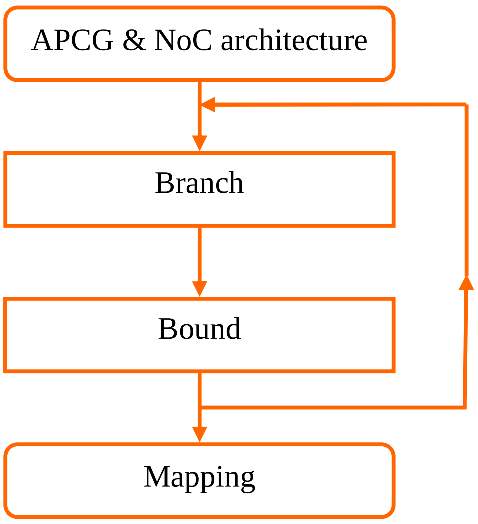

Fig. 18 Overview of the Branch and Bound algorithm

Each node from the search tree has a cost attached to it. This cost represents the energy necessary for the communication among the already mapped IPs. One can view this number as the cost of the mapping held by the node. The cost of a child node cannot be less than the cost of its parent node because the child contains one more IP core mapped to the NoC (and obviously this IP core produces extra network traffic). Also, a node is considered legal, if and only if the bandwidth requirements between the currently mapped IPs are met. When an internal node is found illegal, the sub-tree determined by it is obviously illegal and thus it does not need to be explored. Note that this approach does not lead to sub-optimality because bandwidth requirements are NoC architectural constraints: it is architecturally not allowed to violate them.

Besides checking if a node is legal, deciding whether or not to expand it requires the following two metrics, which are assigned to each node (from the search tree):

- **Upper Bound Cost** (UBC): the value of a node that is no less than the minimum cost of its legal descendant leaf nodes;
- **Lower Bound Cost** (LBC): the lowest cost of a node that its descendant leaf nodes can possibly achieve.

Thus, one may view the UBC as a metric that informs us which could be the **highest** cost of the best mapping that may result from the considered (internal) node. The LBC tells us which could be the **lowest** cost of the best mapping that may result from the considered (internal) node. Based on these observations, the following rule is inferred: a node is not expanded when its cost or its LBC is higher than the lowest UBC that was already found in the whole search tree.

The speed of the Branch and Bound algorithm depends on the way UBC and LBC are computed.

The UBC of a node is assigned the cost of the descendant legal leaf node with the smallest cost. A **greedy method**, which maps the remaining IPs to unoccupied tiles, is used to determine the leaf node with the smallest cost. At each step, the method takes the next unmapped IP with the highest communication demand, and places it to the topological location that is the closest to the ideal topological location (the placement depends on what tiles are still unoccupied). The ideal topological location is considered the one which best facilitates the communication of the IP to be mapped with the already mapped IPs (the communication volume between IPs is considered to be known). After the IP is placed onto the best unoccupied location, its minimum cost can be determined.

The performance of UBC computation is thus determined by the speed of the greedy mapping. The greedy method effectively tries to do a quick mapping so that it may find out what could be the cost of the best mapping that may be achieved starting from the considered internal node of the search tree. The accuracy of the UBC value strongly depends on the accuracy of the greedy mapping.

The LBC of a node is computed as the sum of three communication costs, which are generated by communications between: mapped IPs, unmapped IPs and mapped and unmapped IPs. LBC is essentially the cost of an ideal mapping. This is because each unmapped core is considered to be placed so that the minimum energy is consumed but, multiple cores could be placed onto the same node. So, LBC means: just place all the unmapped cores as best as possible; do not consider that a node gets occupied, i.e. that another core may no longer be placed there.

Having a way to compute the UBC and LBC of each node, the branch and bound algorithm is able to get closer and faster to the optimal solution (compared to Simulated Annealing). However, because the mapping problem is an NP-hard problem, more heuristics are used to significantly reduce the computational time and to provide a near-optimal solution.

To speedup the algorithm, the following techniques are used:

- *IP ordering*: IPs are mapped in the order given by their communication demand (higher communication demand IPs are mapped earlier);
- *Priority Queue (PQ)*: the nodes are branched in the order given by their cost (smallest cost nodes are branched first);
- *Symmetry exploitation*: the first core is placed only onto a part of the NoC nodes because the 2D mesh topology is symmetric (the rest of the nodes are just mirror nodes).

The IP ordering heuristic is useful for applications which do not have a uniform traffic pattern (obviously, in case of uniform traffic, no communication based ordering is possible). Placing the nodes with higher communication demand first reduces the number of nodes to be expanded.

The Priority Queue technique allows decreasing the minimum UBC. This also reduces the number of nodes to be expanded.

When the length of the Priority Queue becomes full, only certain child nodes will be further accepted. If the currently expanding node has the minimal UBC, then all its child nodes will be evaluated for insertion into the queue. Otherwise, only the child node with the lowest cost and the child generated by the greedy mapping (for UBC computation) will be evaluated for insertion in the queue.

It is shown that this branch and bound algorithm is tens of times faster than the general simulated annealing approach. The energy consumption determined by the mappings obtained with branch and bound is not much higher than that of general simulated annealing’s mappings.

Although a 2D mesh with XY routing was used, this algorithm can be adapted to any network topology, with any static routing mechanism.

3.3.2.1 Branch and Bound with Routing

The Branch and Bound algorithm (previously presented) was further extended in [52] so that the objective is not just to perform a mapping which minimizes the communication energy but, also to do a performance aware mapping by generating a deadlock-free deterministic routing function. The authors argue that the routing for NoCs should be deterministic, deadlock-free, minimal and wormhole-based. This is mainly because of two NoC architecture characteristics: resource limitation and stringent latency requirements.

As it can be seen in the Figure 19, the same two steps (branch and bound) are iteratively performed. However, in the branch step routing paths are also generated.

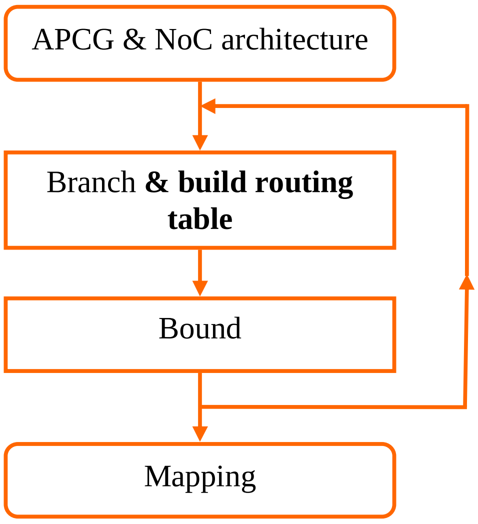

Fig. 19 Overview of the Branch and Bound algorithm with routing

This version of the Branch and Bound algorithm was also applied on a NoC with a 2D mesh topology. Anyway, the algorithm can be used with any network topology.

This routing-aware version of the Branch and Bound algorithm is better than the first one because it addresses the mapping problem from a more holistic perspective. However, it has the disadvantage of a higher computational time.

Compared to the original implementation of the Branch and Bound algorithm, the search tree nodes also contain a *path allocation table (PAT)*. This table stores the routing paths for the traffic among the occupied tiles of the network, and it is generated automatically (so that they are minimal, deadlock- and livelock-free).

Deadlock freedom is ensured by allowing only certain turns for each routing path. For this purpose, a Legal Turn Set (LTS) is used. The authors build their LTS by using the legal turns from the west-first [30] and odd-even [31] routing algorithms.

3.3.2.2 Conclusions to branch and bound

The ideal branch and bound algorithm has the advantage of being able to find the optimal solution. However, this might easily require a prohibitive time because the size of the search tree can increase exponentially. Not expanding the illegal nodes significantly reduces the search space. However, this is not enough to obtain a mapping algorithm that is fast enough. The two branch and bound algorithms employ several heuristics (prioritization of IP cores and nodes to be expanded, topology symmetry exploitation) in order to be able to reach a near optimal mapping fast enough.

The quality of the solutions given by the two mapping algorithms is strongly dependent on the energy model used and on how UBC and LBC are computed.

The energy model is a simple analytical model. At least one disadvantage of this model is that it is static: it may not be used with adaptive routing.

UBC computation relies on a greedy mapping technique that quickly finds a mapping by trying to place each core left unmapped on a NoC node that is still unoccupied so that the energy consumption is minimal. Making local optimal decisions does not necessarily lead to a global optimum. The greedy approach might not be the most suitable one for UBC calculation. However, it has the advantage of speed.

LBC is also not accurately computed. The approach is unrealistic because a core is put in the best place of the NoC without considering if a core was already mapped onto that NoC node. Once again, the approach is fast but it does not provide the tightest LBC.

Routing paths are also created fast. They are deadlock and livelock free but, it is hard to say they are optimal since network dynamics are not accounted for. Nevertheless, it is better to generate such routes rather than simply assuming a NoC with XY routing.

Compared to simulated annealing, the sole advantage of a branch and bound approach is speed. In terms of energy, the improvements given by branch and bound are almost negligible.

Branch and bound must make a tradeoff between huge memory consumption requirements and the amount of the pruned part of the search tree.

3.3.3 NMAP

NMAP [61] is another algorithm for mapping cores onto a NoC architecture with a 2D mesh topology. Similar to the branch and bound algorithm [40], NMAP performs the mapping by satisfying bandwidth constraints but, the algorithm is not energy-aware. However, it takes into consideration the traffic splitting among various network paths (*multi-path routing*). Compared to single-path deterministic routing, multi-path routing can further help at satisfying the bandwidth constraints by balancing the traffic across multiple network links.

The version of the NMAP algorithm which works with *single minimum-path routing* has three major phases.

In the initialization phase a first mapping is computed. This initial mapping is performed by mapping the cores in the order given by their communication demand. The core which has the maximum communication volume is mapped onto one of the nodes of the mesh that has a maximum number of neighbors. Then, the rest of the cores are mapped one by one, in the order given by their communication volume with the already mapped cores: each core is put onto an available node so that the communication cost with the already mapped cores is minimized.

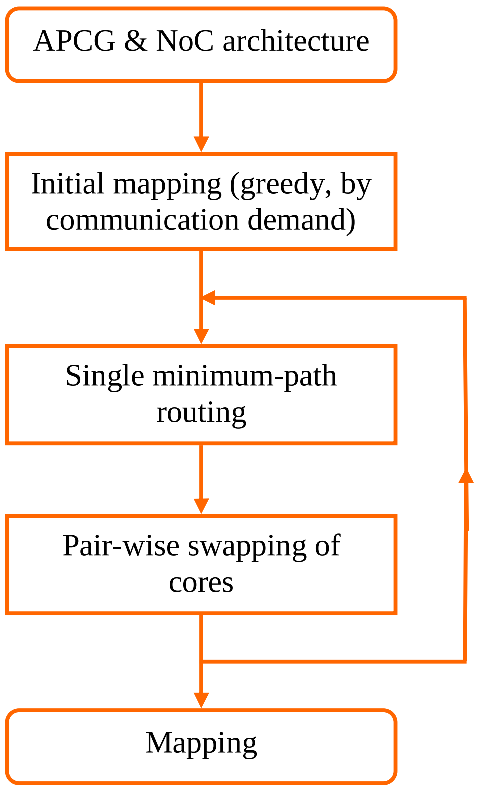
APCG & NoC architecture
Initial mapping (greedy, by communication demand)
Single minimum-path routing
Pair-wise swapping of cores
Mapping

Fig. 20 Overview of the NMAP algorithm with single minimum-path routing

The second phase of the algorithm involves minimum routing path computations. The pairs of communicating cores are sorted in decreasing order, by the value of their communication flows. For each pair of cores, a quadrant is formed. This marks the place from the topology where the shortest path between two cores exits. The minimum path is obtained by applying Dijkstra’s shortest path algorithm. Each path will have a weight assigned. This weight is given by the communication bandwidth sum of all the cores which use that path. After the routing between all core pairs is done, the communication cost is computed for each pair, where bandwidth constraints are satisfied.

The best mapping is found by pair-swapping core mappings and iteratively invoking step two of the algorithm until the best mapping is found.

The NMAP algorithm with *multi path-routing* allows the splitting of the traffic across multiple paths, for each pair of cores. The first phase of the algorithm (initialization) is the same like in the case of the single-path version. Basically, the difference consists on the fact the traffic can be split on multiple paths.

The authors of NMAP show that the average latency is higher when using minimum single-path routing (as compared to when multiple paths are used). This is because the network links are more congested when single paths are used for routing. The effect is emphasized when wormhole switching is used. In this case, entire paths can be blocked, which leads to a significant decrease of network bandwidth.

NMAP is essentially a greedy mapping algorithm. It is well-known that such techniques perform locally optimal choices, hoping that the global optimum will be found. Obviously, a greedy approach may easily fall in a local minimum[4], failing to reach the best solution (global minimum). Otherwise, NMAP emphasizes more the importance of coupling application mapping to routing. It can generate either single path or multi path routes.

3.3.4 Algorithm for Mesh Based On-Chip Interconnection Architectures

Both branch and bound [40] and NMAP [61] algorithms address the mapping problem for 2D mesh NoCs, by considering bandwidth constraints. The mapping algorithm presented in [62] considers, besides bandwidth constraints, latency constraints as well. The objective of the algorithm is to achieve a good trade-off between placing closer the cores which have high bandwidth traffic, so that the communication energy is minimized, and placing closer the cores with tight latencies in order to satisfy the performance constraints. Bandwidth and latency constraints can be viewed as mutually independent. For example, a cache miss does not require high bandwidth but, it needs a small latency.

The energy-latency trade-off is achieved by a two-stage algorithm for Mesh based On-

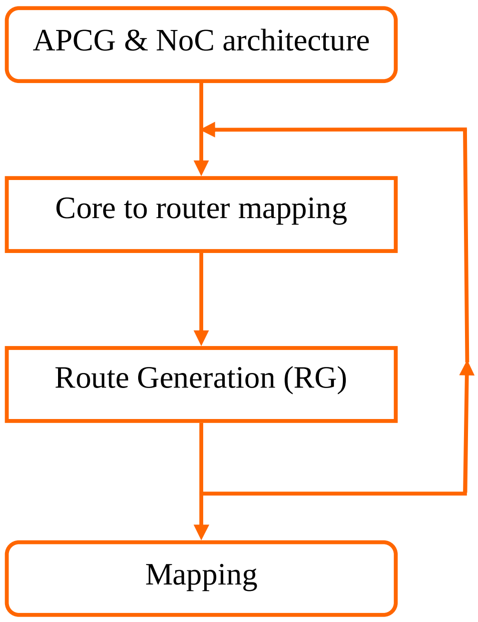

Fig. 21 Overview of the MOCA algorithm

[4] Local minimum for a minimization problem, local maximum for a maximization problem

Chip interconnection Architectures (MOCA).

In the first stage, a mapping of the cores on the mesh NoC is generated. In the second stage, the MOCA algorithm creates a custom route for each communication so that bandwidth and latency constraints are satisfied.

The placement of the cores is determined by recursively invoking a method (Fiduccia and Mattheyses [63]) that solves the graph partitioning problem for the APCG. The graph partitioning problem consists in dividing the graph into *k* disjoint parts with (approximately) equal size and having a minimum cumulative weight of the edges which cross partitions. The edges of the graph are weighted by considering bandwidth and latency constraints. The communications which require low bandwidth but, tight latency constrains are given priority over the communications with high bandwidth but, relaxed latency constraints. The reason for this approach is that bandwidth constraints can be fulfilled by finding alternative routing paths, whereas latency constraints cannot be satisfied in this way.

The graph partitioning process uses a slicing tree. The non-leaf nodes of this slicing tree are the directions of each cut (horizontal - H, or vertical - V) and its leaf nodes are the cores. Dummy nodes are used to track the communication traffic across partitions. The purpose of this kind of nodes is to determine the connected nodes with large weights to be placed close to each other. Each time the APCG is divided, the graph corresponding to the NoC topology is also cut. Through a divide and conquer approach, the APCG is split into approximately equally weighting parts. The NoC topology graph is split in the same way. Each APCG subgraph is assigned a NoC topology subgraph.

The second phase of the MOCA algorithm uses the slicing tree to generate a unique route for each communication. This phase has two sub-phases. The first one generates a minimal route for each communication flow by traversing the slicing tree. The second one searches for a minimal route for a communication flow *that was not successfully routed in the first phase (*because of bandwidth and latency constraints*)*. This hierarchical routing technique falls back to Dijkstra’s shortest path algorithm when it cannot find a proper route. Bandwidth constraints are however respected in this case as well, by restricting Dijkstra’s shortest path algorithm from using the links which would violate them.

The authors of the MOCA algorithm show that their algorithm has a lower complexity than that of the NMAP algorithm [61]. However, the solutions given by the MOCA algorithm could have deadlocks. To solve this problem, the authors suggest a post-processing step which introduces virtual channels at specific routers.

This algorithm is essentially an application of the Fiduccia and Mattheyses [63] algorithm for graph partitioning. Although this algorithm has a linear execution time, it cuts the APCG so that the resulted subgraphs are balanced in terms of the metrics of interest. Obviously, this balancing is approximate. This means that the gain obtained with a cut, as compared with the gain obtained with another cut, is inexact. More than this, a cut loses information because it eliminates graph edges. Thus, such approach works with local information and not with global information. This method is therefore fast but, it may easily find a suboptimal solution.

3.3.5 Multi-objective Genetic Algorithm

All the algorithms presented above deal with the mapping problem for NoC architectures in a *static* manner. They do not take into consideration the dynamic effects of the Network-on-Chip. Also, those algorithms are capable of providing mappings by optimizing a *single objective*, like energy, in [40], or performance, in [61].

In [64] it is presented a genetic algorithm which performs a *multi-objective* (power and performance) exploration of the mapping space of Networks-on-Chip. The NoC uses a 2D mesh topology with Dimension Order Routing and wormhole switching. The proposed algorithm provides not just a single solution (mapping), but rather a set of solutions (mappings). Each solution is a Pareto mapping in the sense that it gives a different tradeoff between the objectives which are to be optimized: power and performance. The authors motivate their multi-objective approach of the mapping problem by showing that it is possible that a certain mapping proves to be the best in terms of power but not in terms of performance as well (or vice versa).

The exploration of the mapping space is made in two steps. First, a NoC simulator is used to evaluate (in terms of performance and power) the available mappings. The evaluated mappings are used as inputs for the second phase of the algorithm, which generates the next mappings to be evaluated. For the second step of the exploration process, the SPEA2 [65] Genetic Algorithm is used.

This cited paper shows the potential of using well known and mature genetic algorithms like SPEA2 for a multi-objective Network-on-Chip application mapping. It would be interesting to evaluate other such algorithms, too. Not only genetic algorithms but, other evolutionary techniques might prove useful, as well. As it is shown in this paper, the success of such algorithms is given by suitable genetic operators (for the NoC application mapping problem).

3.3.6 Multi-objective Genetic Algorithm for Mapping and Routing

A multi-objective approach for the mapping problem is also proposed in [66]. Compared to the work from [64], this work addresses both mapping and routing Network-on-Chip problems, like it is done, for example, in [52]. Bandwidth constraints are also considered. The purpose is to obtain the Pareto set of mappings and routing functions so that the average *communication delay* is minimized and the *fault tolerance* capabilities of the network are maximized. Simulations performed on both synthetic and real traffic patterns show that a multi-objective optimization of the NoC gives better results than a sequential optimization.

3.4 Summary

This chapter introduced the NP-hard problem of Network-on-Chip application mapping. This is a permutation problem, which deals with the topological placement of IP cores onto Network-on-Chip tiles. We showed how it is directly connected to application scheduling and routing. Application scheduling involves assigning tasks to processors; it is also an NP-hard mapping problem. We presented two algorithms that perform application scheduling. One of them treats application scheduling and application mapping together. However, because both of them are NP-hard problems, they are usually treated separately by the research community [3].

In this PhD thesis we approach solely the Network-on-Chip application mapping problem. We introduced several state of the art algorithms that address our problem. We classified them according to a taxonomy proposed by us.

"Whatever you can do, or dream you can, begin it.
Boldness has genius, power, magic in it."

Johann Wolfgang von Goethe

4 Designing a Unified Framework for the Evaluation and Optimization of NoC Application Mapping Algorithms

The NoC application mapping problem is addressed by the research community through application mapping (heuristic) algorithms. As we have already shown in Chapter 3, these algorithms consider the characteristics of both the application and NoC architecture. However, currently, the existing application mapping algorithms are basically evaluated only on 2D mesh topologies. But, they can be extended, to work with other network topologies, too. These algorithms are evaluated only on some specific NoC designs and also, their performance cannot be directly compared because a common evaluation methodology is missing.

We propose a unified framework for the evaluation and optimization of Network-on-Chip application mapping algorithms, called **UniMap**. Such a framework will allow a better comparison of their performance. The framework will also be flexible so that many NoC designs (e.g.: different network topologies) can be used for testing the performance of the mapping algorithms. An overview of UniMap was published in [67], [68]. Our framework is an open source project available under GPL v3 license for the research community [69].

We have successfully used UniMap on our **High Performance Computing (HPC) System** [70] **from "Lucian Blaga" University of Sibiu, Romania**. Our HPC currently has 30 Intel Xeon E5405 homogenous quad cores (15 blades, 120 cores), operating at a frequency of 2 GHz. This means a total of **120 Intel cores**. This HPC system also includes 4 **IBM Cell** Broadband Engine (Cell BE) processors (2 blades, 36 cores). The IBM Cell is a heterogeneous multicore, consisting of a 64-bit dual thread PowerPC (master) core plus 8 SIMD processors. These (slave) vectorial processors, called SPU (Synergistic Processor Unit), are specialized for data intensive processing domains like cryptography, media and scientific applications. The HPC allocates 4.84 GB of DRAM memory for each two Intel quad cores and 7.85 GB of DRAM memory for each two IBM Cell cores. This means a total of **88.3 GB** of **DRAM** memory. The total storage capacity is approximately **1.2 TB**. We also performed simulations with UniMap on the HPC system from **Politehnica University of Bucharest, Romania**. UniMap is written in Java[5] which makes it highly portable and feasible to be further improved with concurrent programming characteristics.

We present in the following sections some related work which sustains our approach. Then, we show the design of our unified framework and propose some solutions for the major problems encountered.

4.1 Related Work

SUNMAP [71] is a tool for automatically selecting the best Network-on-Chip topology for a given application. It uses a generalized version of the NMAP algorithm (see section

[5] Except the NoC simulator, which is written in C++

3.3.3), which can be applied no just on a 2D mesh but, also on topologies like torus, hypercube, 3-stage clos and butterfly.

This framework uses multiple routing protocols (dimension ordered routing, minimum-path and traffic splitting). SUNMAP is a complex tool for automatically evaluating different topologies for Networks-on-Chip, in an application-aware context. The best topology is obtained by considering the minimization of the average packet latency, by satisfying bandwidth constraints, and the minimization of power consumption, by satisfying area constraints.

SUNMAP uses a single application mapping algorithm is. Considering other mapping algorithms too, will provide a more comprehensive view on the performance of different NoC architectures. This framework is actually focused on network topology selection and generation, rather than application mapping. Also, the mappings are evaluated with analytical models. The dynamic effects of the network may be captured only by using a Network-on-Chip simulator.

A framework for simulation and exploration of the mapping space was proposed by Ascia et al. [72]. They search for the best NoC mapping using a multi-objective approach. A Network-on-Chip simulator is used to evaluate the mappings by considering both power and performance metrics.

Three kinds of mapping algorithms are used (in a multi-objective version): genetic (see Section 3.3.5), branch and bound (see section 3.3.2) and NMAP (see section 3.3.3). Their NoC architecture uses a 2D mesh topology with Dimension Order Routing and wormhole switching.

While SUNMAP is flexible in terms of NoC architecture, the second approach presented here is flexible in terms of Network-on-Chip application mapping algorithms. UniMap takes the advantages of both approaches presented above. Multiple application mapping algorithms can be used to map real applications onto different Network-on-Chip designs. The mappings may be evaluated using either analytical models or a NoC simulator (developed by us). Also, multi-objective optimizations are possible.

4.1.1 NoC Designs with Topologies other than 2D Mesh

Obviously, a Network-on-Chip architecture with 2D mesh topology is not the best choice in every situation. The work from [73] sustains this affirmation. Long packet latencies are expected with 2D mesh topologies because there are no short paths between remotely situated nodes. More importantly, real-life applications have varying communication requirements. An alternative for 2D mesh topologies would be to use fully customized topologies. However, although such topologies provide better connectivity, they lose the property of structured wiring (which 2D mesh topologies have). A fully customized topology has wires with varying length, performance and power consumption. Also, it is very important to mention that specific routing algorithms are required for custom topologies. In [73] the network topology is a superposition of a few long-range links and a 2D mesh. By using long-range links, shortcuts between different regions of the network are practically created.

The previously cited paper proves that a customization of the 2D mesh topology significantly improves the performance of the Network-on-Chip. Therefore, one should not resume at evaluating an application mapping algorithm on 2D mesh topologies only. Researching how exactly application mapping algorithms behave on other network topologies could prove useful, especially if several application mapping algorithms are evaluated within a common NoC simulation framework.

4.1.2 Network Simulation for Application Mapping Algorithms' Evaluation

As it is shown in [72], a Network-on-Chip simulator is obviously required to capture the dynamics of a network. Analytical models may easily be insufficiently accurate.

As stated in HiPEAC's vision [1], the Network-on-Chip research field is too new, so that mature tools are still not available. They are expected only around year 2015. A recent survey of NoC tools [74] presents a comparison of some of the most known NoC tools. A common characteristic of the majority of these simulators is that there are not (yet) available for the research community. Besides the ns-3 framework [75] for Internet networks simulation, Noxim [76] is (according to this survey) the only network simulator currently freely available.

Noxim is a highly customizable simulator. It has parameters like: network size, buffer size, packet size, routing algorithm, packet injection rate, traffic pattern, etc. The simulator allows NoC evaluation in terms of throughput, delay and power consumption. However, it currently does not work with real applications and it uses just the 2D mesh topology. We found out it the mesh may only be as big as 19x19 nodes. This is a scalability issue since NoCs with 1000 nodes (Kilo-NoCs) are already researched [77]. Additionally, we cannot estimate NoC area using Noxim.

McPAT [78] is an integrated power, area and timing tool for multicore and manycore designs. It can model a 2D mesh NoC but, the interconnection part is not very parameterizable yet. McPAT is currently in a beta stage of development.

We have therefore decided to implement our own Network-on-Chip simulator. It is integrated into UniMap and it will be presented in section 4.3.

4.2 The Unified Framework Design

UniMap is composed of the following major modules:

- a model for representing real applications;
- a module for assigning the application tasks to IP cores (Scheduller);
- a module that contains application mapping algorithms (Mapper);
- a model for representing different Network-on-Chip architectures;
- a Network-on-Chip simulator.

This design reflects the interaction between the Network-on-Chip application mapping problem and the other two problems with which it interacts (routing and scheduling – see sections 3.1.1 and, respectively, 3.1.2). The modules are as decoupled as possible. This approach allows UniMap to be flexible, reusable (and modular).

We use eXtensible Markup Language (XML) schemas to describe real applications and Network-on-Chip architectures. The Scheduler, Mapper and NoC simulator modules do not interact directly. They communicate through XML models. This approach theoretically allows any NoC simulator to be used with UniMap. Similarly, any scheduling or mapping algorithm can be integrated as easy as possible.

The following figure illustrates these components and presents the design flow of the unified framework.

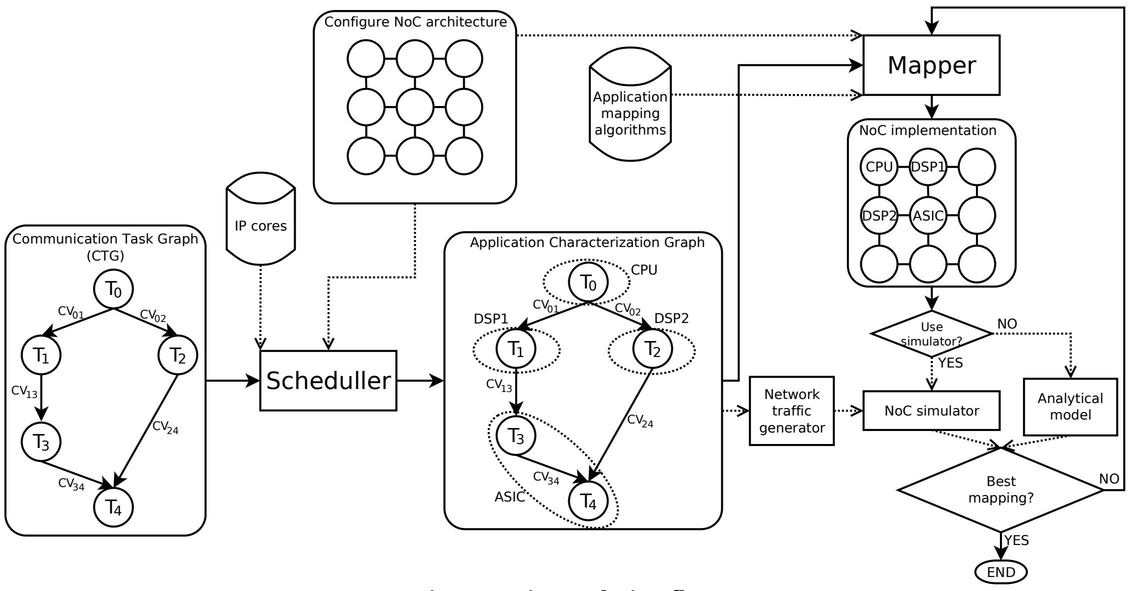

Fig. 22 UniMap design flow

An application running on a NoC architecture is described through its Communication Task Graph (CTG). The CTG presents the application partitioned into tasks (concurrent threads). It shows the communication pattern of the application: which tasks are communicating with which tasks and the communication volume of the data exchanged between tasks (e.g.: CV_{01} denotes the communication volume from task T_0 to task T_1).

We propose obtaining CTGs in three distinct ways:

1. randomly, by using the TGFF [46] tool;
2. from realistic embedded applications, using the E3S benchmarks suite [56];
3. from real-world multithreaded applications, using the CETA [50] tool.

The tasks must be first assigned to the IP cores. This can be done using a scheduling algorithm. For example the EAS algorithm [53] is able to perform scheduling under real-time restrictions, while trying to optimize the energy consumption of the NoC architecture.

The IP cores library from E3S was integrated in UniMap. For each IP core, information like task execution time and power consumption for a given task is known.

The output of the scheduling algorithm is the Application Characterization Graph (APCG). The APCG is the input for the mapping algorithm.

A main component of the framework will consist in a library containing (state of the art) application mapping algorithms' implementations. The performance of every mapping algorithm can be evaluated on multiple NoC designs, through our developed simulator.

The NoC simulator is another important part of the unified framework. An important aspect of the simulator consists in its flexibility. This will impact on the number of possible ways in which the simulated NoC can be configured. The simulator is also responsible with determining the network's performance represented through multiple objectives (performance, energy consumption, etc.). This allows a thorough comparison of the mapping algorithms, in a unified manner. For each selected network design (e.g.: the network topology can be varied), an application mapping algorithm will provide multiple mappings, until the best mapping is determined. The NoC simulator includes a network traffic generator which emulates the communicational behavior of the application (based on CTG and APCG graphs).

We justify in the following section why it is difficult to directly run real applications on the modeled NoC architectures. Afterwards, we present the three tools mentioned above in order to stress out how they model the communication pattern of an application through a CTG. Our network traffic generator is based on these CTG generating tools. We continue by describing how we model the execution of the application tasks by the IP cores. Finally, we present a Network-on-Chip simulator developed for this unified framework.

4.2.1 Model for Real Applications' Communication Patterns

4.2.1.1 The Problem of Running Real Applications on NoC Simulators

A major problem that network simulators have is given by how exactly real applications could be run. A network simulator is not capable of executing binary code. It only knows to route packets, it is communication oriented and not execution oriented. A complex simulator, which simulates IP cores that execute real parallel applications that communicate over an interconnection network, would be better suited. However, while currently there are scalable network simulators, multicore simulators do not provide the same scalability (they can usually run no more than tens of cores).

This is the main reason why traffic patterns are used for NoC research, instead of real applications. A traffic pattern which provides a statistic distribution of the communication is something simple to model and also fast to simulate. It can thus provide useful results in the early phases of NoC development. But, such a traffic pattern does not capture all the properties of a real IP core. The communication patterns of real applications can be bursty and reactive; they are not usually uniformly distributed.

It is shown in [79] why it is difficult to capture the communication patterns of real applications. The authors propose a network traffic generator that can emulate the communication of an IP core by grabbing the type and the timestamp of the communication events. Thus, a real application will run on a full system simulator and the communication patterns will be intercepted. The traffic generator will then be able to mimic the communication pattern of the IP core on different NoC architectures. In order to accomplish this, it is shown that the traffic generator needs to be *reactive*. This is because the network latency varies from one NoC architecture to another. If only communication timestamps would get collected, whenever a message is delayed (due to network congestion for example), this delay should propagate to subsequent messages as well.

Therefore, directly running real applications on modeled NoC architecture is not something easy to achieve. Also, our unified framework is intended to use traffic patterns from real application. Thus, like it is suggested in [45], we propose using CTGs to model the communication pattern of an application.

4.2.1.2 Using Communication Task Graphs to Model the Communication Patterns of Real Applications

4.2.1.2.1 Task Graphs For Free (TGFF)

A Communication Task Graph can be automatically generated using the TGFF [46]. This tool allows the user to generate task graphs of theoretically unlimited size, in a random fashion.

With TGFF, the nodes of the graph are tasks and the arcs of the graph represent the communication between the tasks (nodes). The node of the graph can also be seen as a processor (IP core), and the arc of the graph can be interpreted as a communication resource. An arc can also have a number associated to it. This represents the amount of communication. TGFF allows the user to specify the number of tasks a graph should have, the maximum in-degree and out-degree of the graph nodes, etc. An arbitrary number of attributes can be associated to processors and communication resources. Such attributes can be: execution time, power consumption, cost, etc. The values of the attributes are automatically (randomly) generated by TGFF, based on some parameters specified by the user (e.g.: the average value of the attribute and the interval within it can vary).

TGFF can create only acyclic task graphs. This is because TGFF takes into consideration (hard) real-time systems. Deadlines may be set on the tasks from a graph. Additionally, a period may be assigned to a task. This means that such a task must re-executed by a processor after a specified amount of time.

The advantage of using TGFF over simple (stochastic) traffic patterns (like bit reverse, bit complement etc.) is that it provides a standard method for generating random task graphs.

4.2.1.2.2 The E3S Benchmarks Suite

Another way of obtaining CTGs is the embedded systems synthesis benchmarks suite [56], which is based on the EEMBC [80] benchmarks suite. This benchmark suite contains task graphs for five embedded application types: automotive/industrial, consumer, networking, office automation and telecommunications. There is a version of each task graph for three kinds of systems: distributed (cords), wireless client-server (cowls) and system-on-chip (mocsyn). An application is described by a set of task graphs.

Like in TGFF, the task graphs from E3S are Directed Acyclic Graphs; they have a period and deadlines may be set on tasks. The *period* of a task graph is defined as the amount of time between the earliest start times of its consecutive executions [44]. A *deadline* is defined as the time by which the task associated with the node must complete its execution [44].

Each task graph is described using an ASCII file in the TGFF format. For example, task graph 2 from the *auto-indust-mosyn* benchmark is described as in Fig. 24.

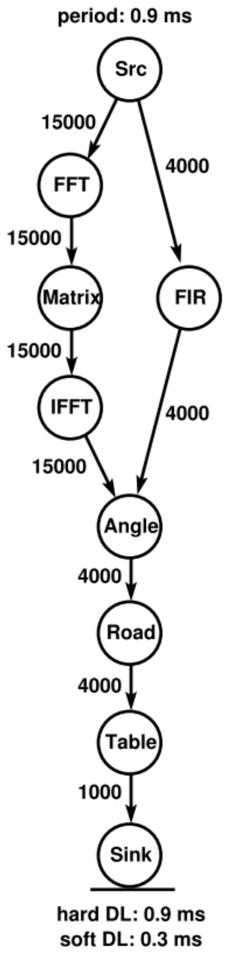

Fig. 23 The task graph 2 from the *auto-indust-mocsyn* benchmark

It can be observed that there are two main elements in the above task graph description file: TASK and ARC. Both of them are characterized by the TYPE attribute (which is a number). For the ARC element, the TASK attribute specifies the amount of communication required by the data exchange between the interconnected tasks. The values which correspond to each TYPE of ARC are specified in the same TGFF file (see Fig. 25).

```
@COMMUN_QUANT 0 {
0 4E3
1 8E3
2 15E3
3 1E3
}
```

Fig. 25 The communication volumes

For example, TYPE 0 means a communication volume of 4000 bits.

For the TASK element, the TYPE attribute specifies the type of the task. For example, TYPE 5 denotes a Fast Fourier Transform (Auto/Indust. Version). All these types are described in the *all-tasks* file (from E3S). For each type of task, performance indexes are available. They were obtained using the embedded processors from EEMBC. The next figure shows an example with such performance indexes, for the AMD ElanSC520 processor:

```
@TASK_GRAPH 2 {
PERIOD 0.0009

TASK src TYPE 45
TASK fft TYPE 5
TASK matrix TYPE 10
TASK ifft TYPE 9
TASK fir TYPE 6
TASK angle TYPE 0
TASK road TYPE 13
TASK table TYPE 14
TASK sink TYPE 45

ARC a2_0 FROM src TO fir TYPE 0
ARC a2_1 FROM fir TO angle TYPE 0

ARC a2_2 FROM src TO fft TYPE 2
ARC a2_3 FROM fft TO matrix TYPE 2
ARC a2_4 FROM matrix TO ifft TYPE 2
ARC a2_5 FROM ifft TO angle TYPE 2

ARC a2_6 FROM angle TO road TYPE 0
ARC a2_7 FROM road TO table TYPE 0
ARC a2_8 FROM table TO sink TYPE 3

HARD_DEADLINE d2_0 ON sink AT 0.0009
SOFT_DEADLINE d2_1 ON sink AT 0.0003
}
```

Fig. 24 Textual representation for task graph 2, from *auto-indust-mocsyn*

```
# AMD ElanSC520-133 MHz -- square
@CORE 0 {
# price buffered max_freq width    height    density   preempt_power commun_en_bit
io_en_bit idle_power
 33   1      1.33e+08 3.10e-03 3.10e-03 0.275    0          0          0       0.16
#---------------------------------------------------------------------------
# type version valid task_time preempt_time code_bits task_power
# Fast Fourier Transform (Auto/Indust. Version)
5      0     1    0.014     150E-6      7.1e+04   1.6
```

Fig. 26 Example of task execution times

It can be observed that for the FFT task, the above AMD processor requires an execution time of 0.014 seconds and the power consumption is 1.6 W.

With the E3S mocsyn benchmarks, a task is described through the following parameters:

- type = the type of the task (all task types are described in the *all_tasks* file);
- version = 0 (this parameter is always set to zero);

- valid = 0 or 1 (it is set to 1 when both *ips*[6] and *cbytes*[7] parameters are defined);
- task_time = $\frac{1}{ips}[s]$(the task execution time, in seconds);
- preempt_time = task preemption time (taken from the *pspecs* file);
- code_bits = 8cbytes (the code size of the task, in bits);
- task_power = the task power consumption, in Watts (taken from the *pspecs* file).

Each IP core has the following parameters:

- price = the core's commercial price (taken from the *pspecs* file);
- buffered = 1 (specifies if the communication of the core may be buffered; this parameter is momentary set to 1);
- max_freq = the processor's frequency, in Hz (taken from the *pspecs* file);
- width = the width of the core, in meters (computed from the *area* parameter, taken from the *pspecs* file);
- height = the height of the core, in meters (computed from the *area* parameter, taken from the *pspecs* file);
- density = 0.275 (this parameter always has this value);
- preempt_power = 0 (this parameter is momentary set to zero);
- commun_en_bit = 0 (this parameter is momentary set to zero);
- io_en_bit = 0 (this parameter is momentary set to zero);
- idle_power = $\frac{task\_power}{10}$ [W] (taken from the *pspecs* file).

4.2.1.2.3 Automatically Extracting Communication Patterns from Multithreaded Applications

CETA [50] is a tool freely available for academic research that allows an automatic run-time Communication Extraction from Threaded Applications (CETA). It is implemented as a Simics [81] module being dependent on the processor architecture and Operating System (OS). The current version of CETA runs on a virtual machine emulating a Pentium 4 processor and a RedHat 7.3 OS with version 2.4.18 of the Linux kernel (it includes SMP support).

While the (multithreaded) application runs, CETA traces the memory operations. In order to know the thread that performs the memory accesses, the context switches must be intercepted. Intercepting context switches is what makes this tool to be OS and CPU dependent.

CETA performs data flow analysis and (with some Python scripts) it generates the Communication Task Graph (CTG) that corresponds to application. Each node from this graph is a thread (or process). The directed edges of the graph show exactly how the communication takes place among the threads: which threads communicate with which threads and the amount of communicated data (expressed in bytes). Each thread is identified through its processes identifier (PID).

[6] Iterations per second (how many times the processor can execute the task in one second - a parameter taken from EEMBC)
[7] Code size in bytes (the code size of the task - a parameter taken from EEMBC)

Two important features of CETA are the Phase Partitioned CTGs and the Directed Acyclic CTGs.

CETA allows the partitioning of a CTG based on a user specified phase number. The phase is expressed in CPU clock cycles and what it basically does is to split a CTG into multiple CTGs. Each resulting CTG will contain only the communication that occurred within its corresponding phase. Such an approach can help at researching the network contention because the threads' communication can be more intensive only in certain phases of execution. Still, it must be noted that the user can specify a single phase number for splitting a CTG. The user cannot for example say that the first phase should be 1000 CPU cycles long, the second phase 2000 CPU cycles, etc. The phase number is constant.

CETA also has the ability of transforming a Phase Partitioned CTG into a Directed Acyclic CTG. This means eliminating the unbounded loops from a CTG, which is important for hard real-time applications. A Directed Acyclic CTG allows bounds on graph execution times.

Additionally, CETA permits the filtering of the data used for creating a CTG. The threads can be filtered by PID and communication volumes which are less than a specified number of bytes can be excluded. Also, the user can make use of the "magic" instructions [81] from Simics. All those elements contribute to a more focused communication tracing by obtaining the CTG for specific parts of the application.

CETA has the disadvantage of a significantly increased application simulation time (compared to the simulation time of Simics, without CETA). Also, the memory consumption can be very big. For example, the authors report ~2.87 GB of memory required for obtaining the data flow of the Tachyon [82] benchmark.

4.2.2 Model for the Processor Elements' Execution

In [45] synthetic benchmarks are considered the suitable kind of benchmarks for NoCs. The main reason is that synthetic benchmarks scale with the system size while still keeping the properties of some particular fixed size application benchmarks. A synthetic benchmark represents an abstraction for a task graph with known computation times and communication loads. Thus, such a benchmark does not actually contain application code but rather it tries to capture the communicational behavior of the application.

Synthetic benchmarks must capture the control and data dependencies between tasks. A simple traffic pattern like uniform random does not account for such dependencies. It simply injects packets into the network with constant probability. Such a stochastic behavior is unlikely to model a class of real applications. Obviously, a complete implementation of real applications captures all the dependencies. But, this approach is considered too complicated and time-consuming. It is considered in [45] that the model of the communication is enough for the evaluation of the communication architecture. The details of the computations performed are not necessary and thus, Finite State Machines that emulate the communication between the tasks of the real applications are proposed.

Communication Task Graphs (CTGs) are used for modeling the applications. TGFF [46] is proposed for generating such CTGs because it provides control and data dependencies.

Each Processing Element (PE) from the NoC is modeled as a Finite State Machine (FSM). The FSM is generated automatically from the CTG and mapping, and it contains the following information:

- **task list**: what tasks are mapped to the PE (this says to what PEs data can be send and from what PEs data can be received);
- **control information**: the data dependencies (e.g.: a communication with a certain PE will be initiated only after enough data was received from another PE);
- **processing time (P)**: how much time a PE requires to execute a task. When enough data was received, the PE will execute P operations (i.e. will wait for a certain amount of time) before generating a response;
- **transaction data amount (D)**: the size of the communication, which will be generated after processing.

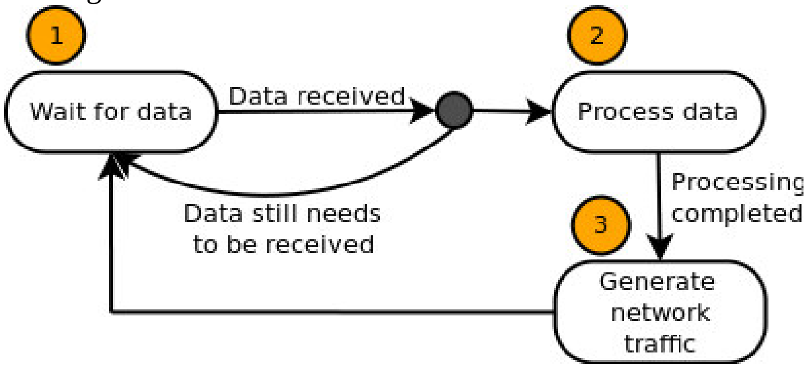

Fig. 27 The FSM associated to a PE

The PE that contains the root task of the CTG starts injecting packets into the network with a frequency specified by the CTG. While the PE with the root task is initially in the processing state (2) of the FSM, the rest of the PEs are in the waiting state (1). The PEs enter in the processing state after all required data are received. The duration of the processing is given by the task type and by the processing element type, too. After the processing is done, the PE enters in state 3. In this state, the generated data is sent to the required PEs. After all the generated data has been sent, the PE reenters in the waiting state.

4.2.3 An Interface for Representing the Inputs Used by the Unified Framework

This section proposes an XML based interface for describing the inputs (applications and NoC architectures) of our unified framework. Using XML schemas (XSDs), we represent the structures of:

- an application task;
- an Intellectual Property (IP) core;
- a Communication Task Graph (CTG), provided by any library of applications described through CTGs;
- an Application Characterization Graph (APCG), created by any scheduling algorithm;
- a mapping, generated by any application mapping algorithm;
- the NoC node;
- the Network-on-Chip link;
- the NoC topology.

We have chosen the eXtensible Markup Language (XML) for creating an interface to our unified framework's inputs because it is a very useful format for keeping and communicating data between decoupled systems, in a platform independent manner. XMLs are easy to create and manipulate by computers and are also human readable.

The main reason for creating an interface for the inputs of our framework is given by our purpose to design a unified way of researching the application mapping problem. Through this interface, the framework will expose what information it requires and uses, and also how this information must be organized.

Our XML interface uses XML binding tools to provide access to the XML documents from Object Oriented code. We use the Java Architecture for XML Binding (JAXB) [83] to map our interface to Java classes, and we do the same with C++, by making use of [84] (which is an XML data binding tool for C++).

The following subsections briefly present the XML Schema Definitions (XSDs) for each of the framework's inputs. More details are available in [85].

4.2.3.1 The Application Task

A task has the following components:

- ID: a unique task identifier;
- type: the category of tasks from which this task is part of;
- name: the name assigned to this task (optional parameter).

4.2.3.2 The IP Core

An IP core has the following characteristics:

- ID: a unique core identifier;
- name: the name assigned to this core (optional parameter);
- frequency: the clock frequency, in Hertz (optional parameter);
- width: the core's width, in meters (optional parameter);
- height: the core's height, in meters (optional parameter);
- idlePower: the power (in Watts) consumed by this core when it is idle (optional parameter).

Note that we may use width and height in order to compute the area of the core.

Any task that can be executed by a certain IP core is specified through an XML data type with two attributes. The *execTime* (optional) attribute tells us the execution time (in seconds) of the task, when it runs on the corresponding IP core. The *power* (optional) attribute specifies the power (in Watts) required by the task to be executed by the corresponding IP core.

4.2.3.3 The Communication Task Graph

The Communication Task Graph (CTG) describes an application that is divided into communicating tasks. The *ctgType* data type is used to represent a CTG.

The CTG has a unique identifier (the *id* attribute) and optionally, a *period*. The period of a CTG is a positive floating point number, expressed in seconds. When it is given, it tells us that the CTG must be periodically executed. Additionally, a CTG marks the communications between the tasks of the application. They are described using the *communicationType* data type.

The *communicationType* specifies the volume of data (in bits) sent from a task (source) to another task (destination). The source and destination tasks belong to the data type called *communicatingTaskType*. Each task is identified through the *id* attribute. Optionally, a communicating task may have deadlines attached to it.

A task deadline is a positive floating point number and it may be either *soft* (deadline) or *hard* (deadline).

4.2.3.4 The Application Characterization Graph

The *apcgType* has two attributes: a unique identifier (*id*) and a CTG identifier (*ctg*). The CTG identifier shows to which Communication Task Graph this APCG belongs to. An APCG contains a list with the available IP cores. Each core has at least one task assigned to it. The core – task association is described through the *coreType*.

Currently, the APCG *taskType* is identical to the *taskType* from the core XSD. In future versions, the APCG *taskType* may contain additional information, like the time when each task is scheduled to run.

4.2.3.5 The Mapping

The *mappingType* has two attributes: a mapping unique identifier (*id*) and the identifier of the APCG (*apcg*) for which this mapping was created. A mapping contains a list of NoC nodes. Each node must have at most on core assigned to it.

The *mapType* is used to represent the node – core association. Each core is represented through its identifier (see the ID element from the core XSD). This is the same for the network nodes. However, for the nodes, the identifiers are based on the NoC topology for which the mapping is created.

We use the XML schemas presented above for representing the applications which run on our unified framework and the IP cores map onto the NoC tiles. However, we also need to describe the Network-on-Chip architecture. Information regarding the NoC which will be simulated may also be needed by the scheduling and mapping phases. Thus, we similarly use the following XML schemas to describe the Network-on-Chip architecture.

4.2.3.6 The NoC Node

The network node is described through the *nodeType* XML data type. It has a unique identifier (*id*) and optionally, it may specify the identifier of the IP *core* which is mapped to it. The *cost* element is optional: it can be used to specify the node's cost, which can be any kind metric, like for example energy consumption. A mapping algorithm may use this element.

Additionally, each node must be connected to at least one network *link* (otherwise it would be isolated). A network link may be used by the node to inject (*type = "in"*) or to receive (*type="out"*) packets. The *value* attribute specifies the link's unique identifier.

Also, a routing table may be available for a node. For example, an application mapping might also provide the routing function. The routing table entry has three fields: the identifier of the *source* node, the identifier of the *destination* node and the identifier of the *link* which will be used to route the traffic coming from the source node and going to the destination node.

Parameters that are specific to a certain network topology may be described using the *topologyParameterType*. Each topology parameter is specified through the *type* attribute and it is connected to a *topology* and has a *value*.

4.2.3.7 The NoC Link

Each link is described using the *linkType*. It has a unique identifier and like in the case of the network node, a link may have a *cost* associated to it. For example, it can be the energy needed to transmit a flit. However, the main characteristic of a link is its *bandwidth*, expressed in bits per second.

The mandatory XML elements of a link are the source node (*sourceNode*) and the destination node (*destinationNode*): packets are sent from a network node (the source) to another network node (the destination).

Like in the case of network nodes, parameters specific to a certain topology may be specified using the *topologyParameterType*.

4.2.3.8 The NoC Topology

Nodes and links create the Network-on-Chip topology. Each topology is uniquely identified using the *id* attribute (we use the *name* attribute for a human readable topology naming).

Topology parameters may be used to set different properties of each kind of topology (a 2D mesh, for example, has two parameters: row and column).

4.2.4 The Scheduling Module

The scheduling module of our unified framework has the role to provide a set of algorithms that are responsible with mainly assigning the tasks of an application to available IP cores. We do not currently deal with planning the execution of the assigned tasks on the IP cores. Typically, a scheduling algorithm works with a Communication Task Graph as input and outputs an Application Characterization Graph. We propose a common interface to all of the scheduling algorithms.

Currently, the *Scheduler* interface simply defines the *schedule* operation, which involves reading a CTG, accessing the available IP core library (both defined using XMLs), assigning tasks to cores through a certain algorithm and outputting an XML file containing an APCG.

For now, we have implemented just three simple schedulers: random, direct and minimum execution time.

The *random scheduler* deals with the task-to-core assignment in a random fashion: for each CTG task, an IP core is randomly chosen from the library of IP core. Multiple tasks may be assigned to the same IP core and the same type of core may be used multiple times (e.g.: we may have 3 DSP cores with exactly the same configuration, each of them having at least one task scheduled for execution).

The *direct scheduler* just assigns each task to the first IP core that can execute it.

The *minimum execution time scheduler* assigns each task to the IP core which executes it the fastest time.

4.2.5 The Mapper Module

The mapping module holds the application mapping algorithms library of the unified framework. An application mapping algorithm has the role of topologically placing the IP cores onto the available Network-on-Chip nodes. It uses the Application Characterization Graph provided by the scheduling module: all the IP cores from this graph are assigned to NoC nodes. The APCG and the mapping produced by the algorithm are both described using the XML interface presented in Section 4.2.3.

We have developed an interface, called *Mapper*, for all the application algorithms implemented by our unified framework.

Currently, the *Mapper* interface simply specifies the *map* operation. This involves: reading the APCG data from an XML file, mapping the IP cores onto the nodes of the given NoC and outputting the obtained mapping into an XML file. As a basic precondition, this operation might throw a *TooFewNocNodesException* exception when the given NoC does not have enough nodes for all the cores that need to be mapped (each NoC node can hold only one IP core).

We have implemented the r*andom mapper* as a basic application mapping algorithm. This *Mapper* randomly places the IP cores onto the NoC tiles.

Another straightforward *Mapper* is the *exhaustive search mapper*. It performs mapping by generating all the possible mappings, which is obviously a factorial number. This mapper is clearly unfeasible but, we used it for generating the optimal solutions on a small 3x3 2D mesh NoC.

The mapping module contains all the application mapping algorithms developed in our unified framework. Currently, we have integrated the following algorithms: Simulated Annealing (see Section 3.3.13.3.2) and Branch and Bound (see Section 3.3.2). We have also developed an Optimized Simulated Annealing (it will be presented in Chapter 6). UniMap also integrates the jMetal library [86], which provides state of the art evolutionary algorithms for single and multi-objective optimizations. Exactly what evolutionary algorithms we used from jMetal will be detailed in Chapters 7 and 8.

We tried to implement the mapping algorithms as accurately as possible. It is well known that floating point numbers are represented with approximation by computers. For example, the number 0.1 cannot be represented in binary floating point with finite precision because: $0.1_{10} = 0.000110011..._2$. Since 0.1 would require an infinite number of bits in order to be represented in base 2, approximation is inevitable. Representing 0.1 on 24 bits leads to the value 0.100000001490116119384765625.

Floating point arithmetic in computers is therefore approximate. Knuth defines in [87] (page 218) the following relations for floating point comparison with approximation:

$$a \prec b \Leftrightarrow b - a > e \cdot \max(|a|,|b|) \quad \text{(}a\text{ definitely less than }b\text{)}$$
$$a \succ b \Leftrightarrow a - b > e \cdot \max(|a|,|b|) \quad \text{(}a\text{ definitely greater than }b\text{)}$$
$$a \approx b \Leftrightarrow |b - a| \le e \cdot \min(|a|,|b|) \quad \text{(}a\text{ is essentially equal to }b\text{)}$$
$$a \sim b \Leftrightarrow |b - a| \le e \cdot \max(|a|,|b|) \quad \text{(}a\text{ is approximately equal to }b\text{)}$$

Two relations are defined for testing if two floating point numbers are equal. According to Knuth, the “essentially” relation is somewhat stronger than the “approximately” relation.

All the relations above are based on a positive real number that specifies the degree of approximation considered. We set *e* to be the **machine epsilon**, which gives an upper bound on the relative error resulted due to rounding made in floating point arithmetic. This number is the smallest floating point number which may be represented by a computer. Thus, it is machine dependent but, it can be easily computed. Based on the algorithm from [88], our *MathUtils* class automatically computes this number and stores it in the MACHINE_EPSILON_FLOAT constant (so that this computation is performed only once per program run).

4.3 The Developed Network-on-Chip Simulator

In the following section we present ns-3 NoC, a Network-on-Chip simulator that the author of this Thesis started to develop during his five months of PhD external research stage at Augsburg University (Germany), Department of Systems and Networking, led by Professor Theo Ungerer. We decided to develop our own NoC simulator because the current tools for this (new) research field are still immature. According to HiPEAC’s vision [1], mature NoC simulators are expected only in 2015.

We describe next a modular, flexible and scalable NoC simulator. Our ns-3 NoC is an open source project, which we contribute to the Network-on-Chip research area.

4.3.1 The ns-3 Network Simulation Framework

The ns-3 simulator is a **discrete-event network simulator** targeted primarily for research and educational use. It is developed as a network simulator for Internet systems. An Internet stack is implemented (protocols like TCP, IPv4, IPv6, UDP, ARP etc. are available). Traditional routing protocols for Internet systems are also implemented. The simulator provides multiple models for interconnecting network nodes: point to point, CSMA, wireless etc. ns-3 is written in C++ and it also supports Python scripts. It is built after the successful ns-2 simulator, which was one of the most used simulators for network research. There are over 50% of ACM and IEEE network simulation papers from 2000 - 2004 that cite the use of ns-2. The ns-3 project started in 2006 and it is an open-source project.

ns-3 is focused on modularity, reusability and extensibility. It offers a clean design (with a high accent put on software design) and scalability. There is a set of fundamental components (objects) with which ns-3 works:

- **Node**: the representation of a network entity such as a personal computer, a router, etc. A Node can aggregate other components as protocol stacks and therefore, it has the capability to process packets;
- **Application**: a packet generator and consumer which can run on a Node and talk to a set of network stacks;
- **Topology**: represents the network’s topology and it is composed of two elements:
    - **NetDevice**: the Network Interface (the link between a Node and a Channel);
    - **Channel**: the medium used to communicate and interconnect NetDevice objects;

ns-3 supports virtualization by developing two modes of integration with real systems:

- ns-3 interconnects virtual machines (virtual machines run on top of ns-3 devices and channels);
- ns-3 stacks run in emulation mode and produce/consume packets over real devices.

To integrate with real network stacks and emit/consume packets, a real-time scheduler is used to lock the simulation clock with the hardware clock. The purpose of the real-time scheduler is to cause the progression of the simulation clock to occur synchronously with respect to some external time base. Without the presence of an external time base, simulation time jumps instantly from one simulated time to the next. With a non-real time scheduler, ns-3 freezes the simulation time during event execution and the simulator advances the simulation time to the next scheduled event. A real time scheduler has the same behavior but, it also attempts to keep the simulation clock aligned with the clock of real (not simulated) machines.

While most simulators generate text files as their output, ns-3 has a tracing system for decoupling the generation of trace events from their serialization to a trace file. Two serialization formats are currently supported: plain text and PCAP. PCAP is the file format used by known network protocol analyzers (network sniffers) like *Wireshark* (http://www.wireshark.org/) and *tcpdump*. ns-3 traces are what the simulator produces as output, and are typically used for tracking the communication that occurs in the simulated network.

ns-3 uses packet generators for creating traffic in the network. They are called applications. An application can perform the simple task of injecting or consuming network packets. The simulator currently does not support running real applications on the simulated network (allowing parallel applications, written in MPI, to run on an ns-3 simulator is a work in progress). Therefore, traffic patterns are typically used with ns-3 for evaluating the performance of a network.

The developers of ns-3 claim that this simulator is one of the fastest and the most memory efficient simulator currently available. However, parts of the simulator are still under development (for example, the visual part of ns-3 is still experimental). There is not yet a Graphic User Interface for building network topologies (the user has to programmatically interconnect the network nodes).

Although ns-3 is mainly a network simulator for Internet systems, the core of the simulator has been used for implementing a Network-on-Chip simulator. Essentially, the fundamental components of the ns-3 framework have been implemented for a NoC system.

The implementation of the ns-3 NoC simulator started from the work done in [89], where a Network-on-Chip simulator, called NoCSim, was developed using the Java programming language. The ns-3 NoC simulator is written in C++ and it is faster than NoCSim. Our ns-3 NoC simulator inherits the features of NoCSim:

- 2D mesh NoC topology;
- Irvine [90] NoC architecture;
- Dimension order routing (XY and YX) and other two simple routing algorithms based on network load: Static-Load-Bound and Self-Optimized;
- Wormhole switching;

- Deterministic (bit-complement, bit-reverse, matrix transpose) and stochastic (uniform random) traffic patterns;
- User specified packet size;
- Packet injection probability;
- Synchronous network packet injection;
- Input channel buffering;
- Measuring the average packet latency after simulating the NoC for a given number of cycles (the NoC may also be warmed up for a specified number of clock cycles).

The ns-3 NoC simulator is however a more flexible, scalable and faster simulator. It also adds more features. For example, it allows asynchronous network packet injection, more switching mechanisms and topologies and it measures power consumption and area. More details about ns-3 NoC are given in the next sections.

4.3.2 The ns-3 NoC Architecture

We start presenting our developed simulator by overviewing its architecture. The details about each ns-3 NoC component are detailed afterwards. The ns-3 NoC basic architecture is illustrated by the following figure, which presents the major components of a network node.

The **NocApplication** models an ns-3 application. It represents the Processing Element (PE) of a Network-on-Chip, being responsible with injecting packets into the network and with receiving packets from the network (obviously, it receives those packets which are sent to it by other Processing Elements).

Packets are injected into the network with a frequency specified by the parameter called *data rate*. The *data rate* is expressed in bits per second, and based on the user configurable size of packets, the packet injection frequency is determined. An ns-3 application runs for a given amount of time. Therefore, packets are injected into the network, until the running time of the application ends. Additionally, the user may specify a maximum amount of bits that an application can inject into the network.

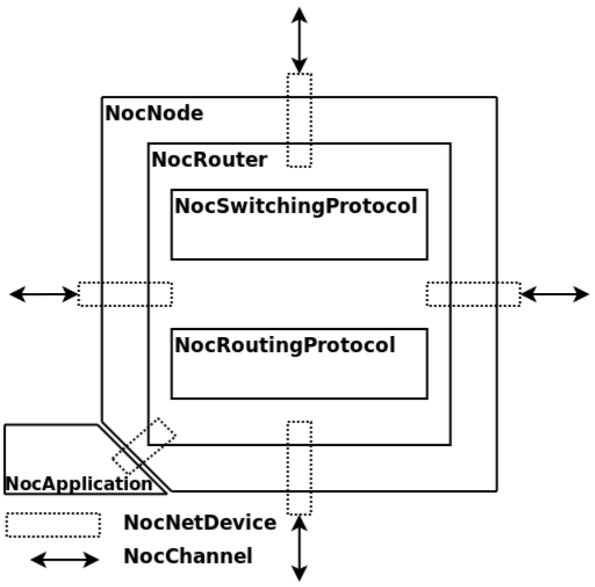

Fig. 28 The architecture of the ns-3 NoC simulator

This kind of ns-3 application allows for packets to be injected into the network asynchronously. However, the simulator also models an ns-3 application which mimics the behavior of synchronous network (like in [89]).

The synchronous ns-3 application allows packets to be injected into the network with a certain injection probability. This application can inject one packet per clock cycle. Each network router routes one packet per network cycle. Packets are injected for a certain number of cycles (specified by the user). Network-on-Chip clock frequency is an ns-3 NoC parameter.

The **NocNode** models the network node. It allows ns-3 applications to connect to it, and it also aggregates a router.

The **NocRouter** has the responsibility to route the packets through the network. It uses a switching mechanism for deciding when packets are sent forward, and a routing protocol which determines the next network node where the packet will go.

The **NocSwitchingProtocol** represents a common specification for all the switching techniques. Currently, the simulator implements store-and-forward, virtual cut-through and wormhole switching techniques.

NocRoutingProtocol is an interface for all of the routing protocols from the simulator. The routing protocols implemented are: Dimension Order Routing, Static-Load-Bound and Self-Optimized routing algorithms.

A **NocNetDevice** represents the Network Interface Controller (NIC). It connects network nodes to network channels, and it is responsible with sending and receiving packets from adjacent nodes. The router has direct access to the net devices. Based on what routing path is established, a certain net device will be chosen for sending a packet. The packet will travel through the channel associated to the selected net device and it will arrive at the neighboring node via the corresponding net device.

Each net device has one input queue and one output queue (which effectively allow for input and output channel buffering). The size of these queues can be specified by the user. Note that, since the size of the input buffers is limited, it is possible that there are not enough resources available to buffer all the packets which are injected into the network. This problem is solved by considering that there is a buffer with unlimited size located at each processing element. This approach was also adopted in [91] and it is based on the fact that each processing element can use its local memory to store the data it needs to send over the network, until the NoC is ready to transport it.

Each net device continuously monitors its input queue and as long as packets are available, it requires the router to provide a route for the packet from the head of the queue (the switching mechanism will decide when the packet may be routed). After the packet is ready to be sent, if the respective channel is available, the send will be made.

The **NocChannel** implements the communication channel used by the network. It is characterized by several parameters. The data rate is practically the bandwidth of the channel. It shows how many bits the channel can transfer per second. The delay parameter can be used to specify how much time the channel needs until it starts transmitting the available data. The channel allows for half-duplex or full-duplex communication. By default, we consider the NoC to have bidirectional links and thus we use full-duplex NocChannels. Finally, the physical length of the link may be specified.

4.3.3 The Packet Format

A packet that travels through the network is made of at least one flit. The first flit from a packet is called a **head flit**. A packet may contain additional flits, called data flits. Typically, the last data flit from any packet is called a tail flit. All types of flits contain a data payload but, the head

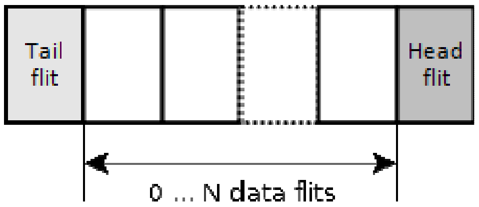

Fig. 29 Packet format

flit also contains a header. The packet header contains information used at routing it through the network. Basically, the source and the destination of a packet are codified in the header. The header is interpreted by the routers in the network. The data payload is not interpreted.

The head flit represents the header of the packet. Additionally to the header, a head packet can also contain a data payload, which is represented by one or more body flits. A data packet is made only of body flits.
The following table presents the parts which compose a header.

Header part	**Size [bits]**	**Description**
distance	8	The distance to the destination node, expressed relatively from the source node.
source	8	The address of the source node.
dataFlits	8	How many body flits the packet contains.
groupAddr	16	This field can codify a group of nodes to which a packet can be sent (useful for multicasting).
subdataId	8	Can optionally be used for package indexing. Ordering packets is important when they can reach the destination out of order (adaptive routing).
load	8	Propagates information about network load (useful for adaptive routing).

Fig. 30 shows the ns-3 NoC packet header.

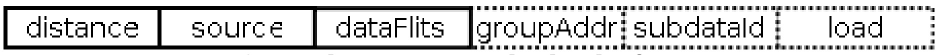

Fig. 30 The ns-3 NoC packet header format

The ns-3 NoC packet header is based on the Irvine NoC architecture [90]. Currently, our simulator uses only the first three header fields. The other three fields, useful for multicasting and adaptive routing are for further developments. We already used the *load* field while testing ns-3 NoC with two adaptive routing algorithms which will be presented in section 4.3.6.

Since ns-3 NoC works any kind of k-ary d-cube topology, the *distance* and *source* fields are technically implemented as arrays of *d* elements, 8 bits each (*d* is the topology number of dimensions). The header size increases automatically with the NoC topology dimensions.

For each dimension, we use the left most bit to mark whether the destination node is back or forward relative to the source node. For example, in a 2D mesh, this bit has the following meaning:

- 1 means West (horizontally) and North (vertically);
- 0 means East (horizontally) and South (vertically).

With the ns-3 NoC simulator, the size of the data payload is a parameter. The user can decide how much payload data a flit can hold. By default, the actual data content is implicitly set by ns-3. However, the simulator allows the user to specify the contents of the payload data.

In addition to the packet header (called **NocHeader**), the ns-3 NoC simulator also uses a so called packet tag: **NoCPacketTag**. This data structure conveniently stores additional information used for allowing a better software modeling of the dependence from head flits and data (body) flits:

- each data flit is tagged with the unique identifier of the head flit that it belongs to. This allows for a better control of the packets which make a packet;
- whenever a head flit remains blocked in the network (i.e. it cannot advance because the channel is busy), this information is kept because it can be used at notifying data flits that their corresponding head flit is blocked in the network[8].

The purpose of the packet tag is to keep information about the flits of packet, which is additional to the one kept in the header of a packet because it is used more for software modeling purposes. Another use of this is for keeping the time when the packet was injected into the network and the time when the packet reached its destination. This is used for collecting statistics.

4.3.4 Network Topologies

Initially, ns-3 NoC implemented just the 2D mesh topology and a variation of it. The topology used by the Irvine architecture is a 2D mesh that vertically interconnects any two nodes with two channels (not just one). As a (Bachelor) diploma project from “Lucian Blaga” University of Sibiu, Romania, coordinated by Professor Lucian Vințan, PhD and Ciprian Radu (author of this Thesis), student Andreea Gancea implemented a k-ary d-cube NoC topology [92]. This includes the 2D mesh, 2D torus, 3D mesh, 3D torus and hypercube network topologies. Additionally, the Dimension Order Routing protocol was generalized to be applied for any k-ary d-cube topology.

The following UML class diagram summarizes the kind of topologies that are currently supported by ns-3 NoC.

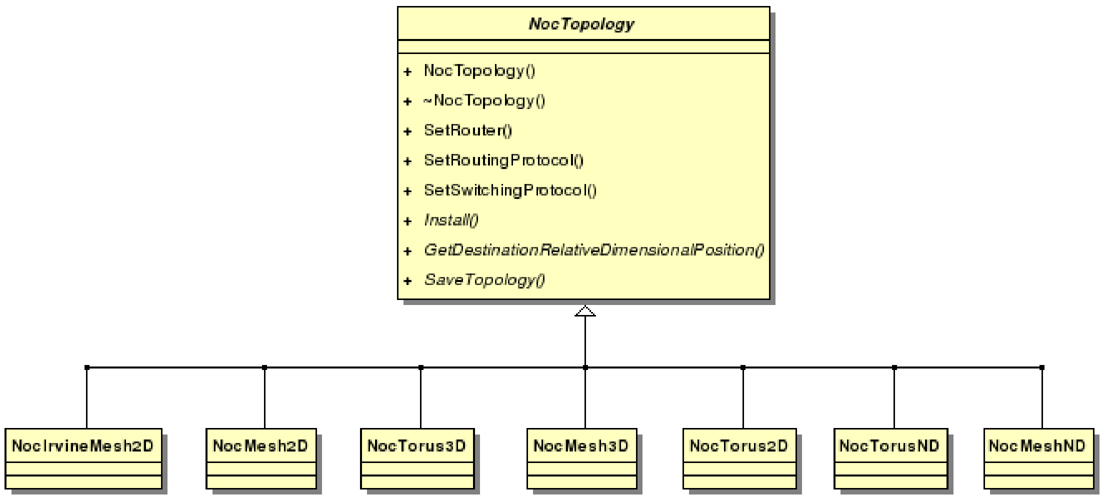

Fig. 31 The ns-3 NoC topologies

[8] The Virtual cut-through switching technique makes use of such information

All ns-3 NoC topologies inherit the *NocTopology* abstract class. As a consequence, they must implement three methods. The first, called *Install*, builds the topology by instantiating the network nodes and communication channels and then interconnecting them. The second method (*GetDestinationRelativeDimensionalPosition*) is invoked every time ns-3 NoC needs to know the relative offset of a destination node, from a source node, in any topology dimension. The third method is used to save the topology in UniMap's XML format. Using the setters of the *NocTopology* class, we can specify the size of the buffers and what kind of router, routing and switching protocols the NoC will use.

The class *NocMeshND* implements a k-ary d-cube topology. If we want to add links between the boundary nodes (so that we obtain tori), we use the class *NocTorusND*. Bi-dimensional and tri-dimensional meshes and tori can be instantiated with these two classes. However, for convenience, we also have the classes NocMesh2D, NocTorus2D, NocMesh3D and NocTorus3D. A k-ary d-cube (mesh, torus) is created by simply specifying an array with the number of network nodes (*k*) on each dimension (*d*).

4.3.5 Router Architectures

We present in this section the router architectures developed in ns-3 NoC. Essentially, we adopted the common router design available in the network literature [26], [25]. This router is for networks with a 2D mesh or 2D torus topology. We then adapted this router design for the general k-ary d-cube and the Irvine NoC topologies.

The following UML class diagram presents the ns-3 NoC router component.

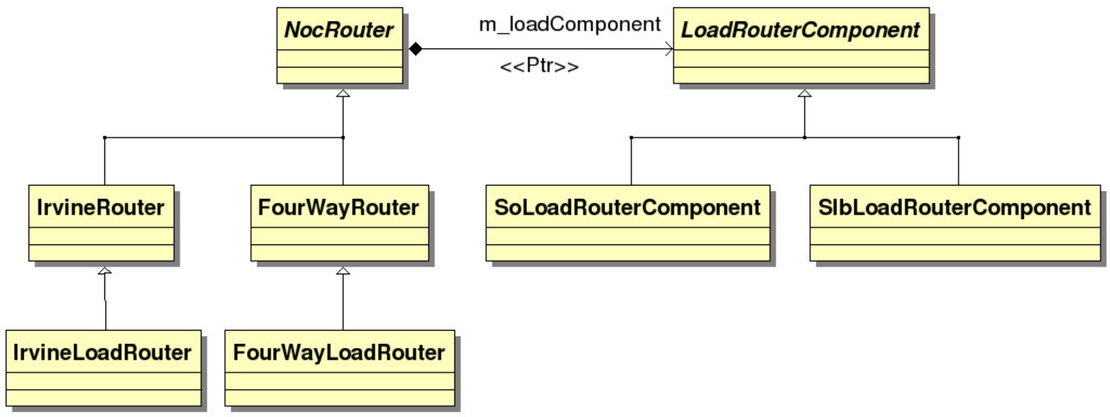

Fig. 32 The ns-3 NoC router component

Two router architectures are implemented in the ns-3 NoC simulator: a generic router for 2D mesh topologies (called **FourWayRouter**) and the Irvine router. Actually, the FourWayRouter is designed so that it can be used with any k-ary d-cube NoC topology (we gave it this name because it is inspired from a typical router for 2D mesh networks). An abstraction called **NocRouter** specifies a generic router for the simulator. This abstraction allows any concrete implementation of a router to make use of the load of the network. However, this is not mandatory: a router will be able to work with or without network load. The bridge design pattern [93] is used to decouple the NocRouter abstraction from the router component implementation that deals with network load information (**LoadRouterComponent**).

The LoadRouterComponent is an interface that requires its implementations to specify how a router is supposed to compute its local load, and also what load value must be propagated to the neighbor (from a certain direction). Note that the computation of the load values is the responsibility of the router, whereas a routing protocol aware of the network load (like SLB or SO) only uses the load values to perform route evaluations. The advantage of such decoupling is that any routing protocol can make use of load information, as it is computed by a certain load router component.
Currently, two load router components are implemented: one for the SLB algorithm and another for the SO algorithm.

4.3.5.1 The Four Way Router

The next figure shows the architecture of our developed four way router. This router is used with 2D mesh and 2D torus NoC topologies.

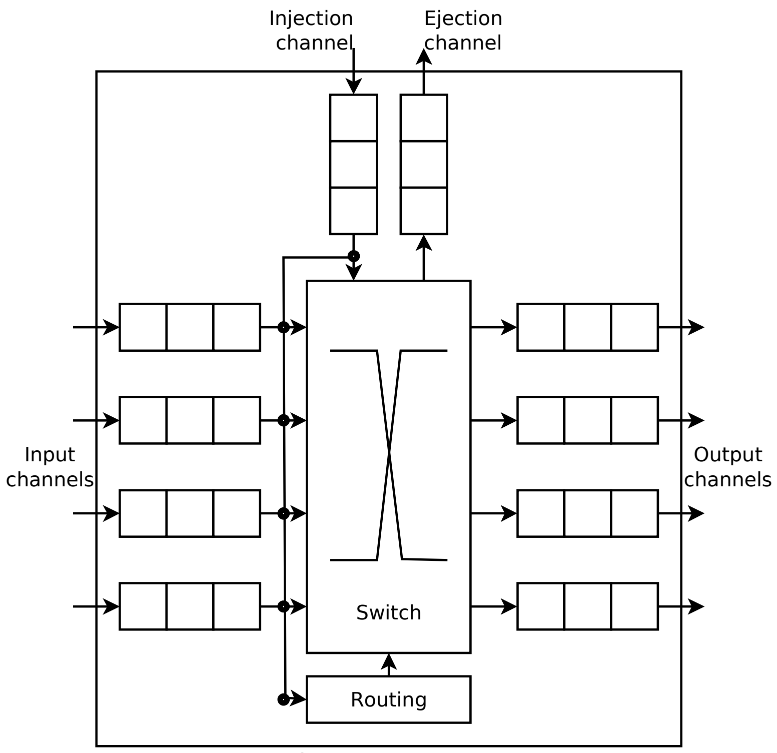

Fig. 33 The four way router architecture

It is composed from a routing module, a switch and ten channel buffers. The injection and ejection channels are used by the Processing Element to send and receive data to and from the NoC. The four pairs of input and output channels correspond to the four node neighbors a NoC node may have in a 2D mesh (or torus). Because of the grid structure of such topology, these channels are tagged with North, East, West and South identifiers. The size of each channel buffer is a simulator parameter. It is expressed in number of flits.

Each packet flit arrives at a router through its injection channel or through one of its input channels. Until it can be forwarded, it stays in the corresponding buffer channel.

In order to forward a packet, the router uses its Routing module to compute the output port[9] to which it will send the packet. This module has a routing protocol which determines the packet's route. Route computation is performed only for the head flit of a packet. The rest of the packet's flits will simply follow the head flit. This is possible because the router uses a small memory where is stores the route used for each packet. A packet route is kept until the tail flit of that packet is forwarded.

Once the packet route is known, the packet flit enters the switching phase. Here, a switching mechanism determines when the flits will leave the router and go to another one (through an output channel). In general, any flit cannot be forwarded until the assigned output channel buffer cannot hold that flit (because it is full). The switch works like a crossbar: at any time, any input port can be coupled to any uncoupled output port.

Once the flit reaches the output channel, it will be forwarded when that channel is idle, i.e. it can transmit another flit. If output channel buffering is not used, the switch will not allow a flit to pass while the channel is not idle.

Our router is capable of forwarding a flit in a single NoC clock cycle provided the needed output channel is able to transmit it in that cycle.

This basic and simple router architecture is not bounded by the number of input – output channel buffer pairs. By simply setting the router to have two such pairs for each NoC topology dimension, *this router architecture can be used with any k-ary d-cube NoC*. This is in fact done automatically when building the topology.

This router design still needs to be improved to make it model more realistically a practical router. As further work we intend to implement virtual channels, required for deadlock avoidance. A router pipeline will also allow us to model the router more accurately. Such pipeline typically has the following phases: route computation, virtual channel allocation, switch allocation and switch traversal. When using virtual channels, an allocator is needed to allocate virtual channels to packets and switch cycles to flits. An arbiter is required to resolve the situations when there are multiple requests for a single resource. For example, when two flits, one coming from North and another one from West, need to take the South path at the same time, which of them will be the one allowed first? An arbiter makes such decisions. It is the building block for allocators. An allocator performs the more complex task of matching a set of resources with a set of requesters. Any requester may need one or more resources. Our current router architecture relies on the internal event scheduler of ns-3 framework to perform the arbiter role. Each routing request is an event. Events are registered in an order and they are processed in a first in first out manner. This allows us to have a Round Robin arbitration for resources.

4.3.5.2 The Irvine Router

The Irvine router was proposed in [90] to route packets in a deadlock- and livelock-free manner. Freedom from deadlock is achieved by using separate routing paths for the vertical direction and unidirectional horizontal paths. Livelock freedom is obtained by using a minimal routing protocol.

[9] Or output ports, in case of multicasting. However, ns-3 NoC does not currently support multicasting.

As it may be seen in the following figure, the router consists of: an internal router, a right router and a left router.

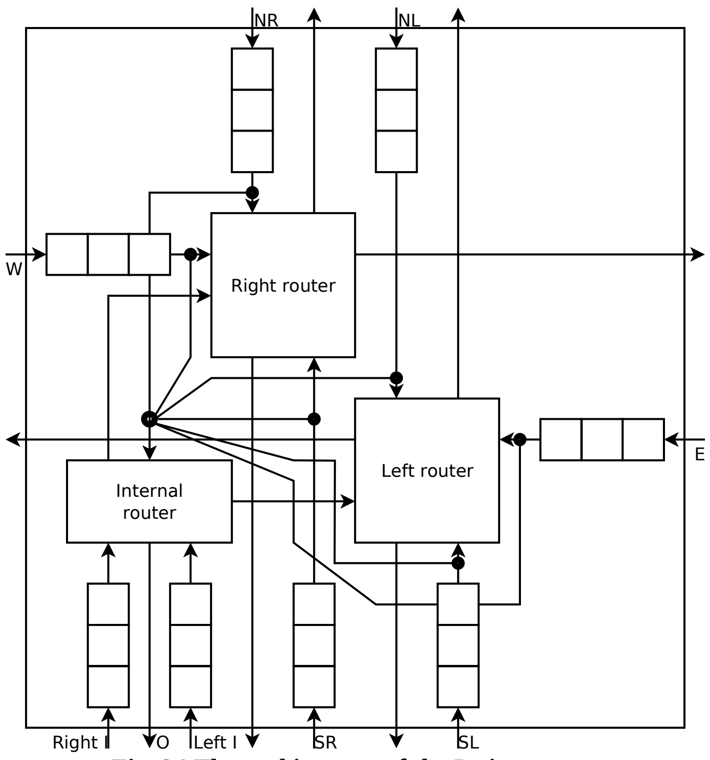

Fig. 34 The architecture of the Irvine router

Our simulator uses this router only for NoCs with an Irvine 2D mesh topology. The internal router is only used as an interface to the processing element. It receives data from the Processing Element through two injection channels (*Right I* and *Left I*). Everything that needs to go East (E) or West (W) is put in *Right I* buffer, or *Left I* buffer respectively. The internal router forwards everything from *Right I* to the right router and everything from *Left I* to the left router. The right router is capable of routing packets horizontally only from West to East, while the left router can route horizontally from East to West. For the North and South directions, both left and right routers have their own channels (NL, SL and NR, SR respectively).

4.3.6 Routing Algorithms

Both router architectures developed in ns-3 NoC work by default with a Dimension Order Routing (DOR) algorithm (presented in section 2.3.4.2). They do not account for the network load. From this point of view, the routing decision may be improved by avoiding the hot spots from the network. The hot spots can be identified by considering the load of each router, and by avoiding them, the network throughput can be improved. The following two algorithms perform routing by accounting for the network load.

4.3.6.1 Static-Load-Bound Routing

Two routing algorithms which use load information to generate network routes are proposed in [89]: the Static Load Bound (SLB) routing algorithm and the Self-Optimized routing algorithm [94].

Both algorithms work by performing the following generic steps (the differences between the two will be presented later).

First, a *routing function* creates the set of available output channels for a certain packet. Second, a *selection function* evaluates each possible route (determined by the previous step), and selects the best one for routing the packet.

Each router will have a *local load* value computed for it. This will be propagated to the router's neighbors. The load value is called "local" because it is not used by the router which owns it to perform its routing; it is only used to inform the neighbors about its load. For routing, a router only takes into account the load values of its neighbors.

The *propagated load value (pload)* does not include only the local load but also the load values of the neighboring routers. The reason for this approach is that, the closer a router is to a hot spot, the higher the propagated value will be. The propagated load value is computed as $pload = \frac{1}{3}\left(2 \cdot load + \frac{\sum pload(dir)}{|pload(dir)|}\right)$.

The *local load* value weights two thirds and one third of *pload* is given by the loads of the neighbors. Only the neighbors that represent possible routing directions are considered (for example, the router is not allowed to route a packet back to the direction it came from).

SLB evaluates all the possible routes with the relation: $quality = direction - busy\,channel - loaded\,router$.

The first parameter takes the value 2, when the evaluated direction leads the packet closer to the destination. If not, the direction parameter will be set to zero.

The second parameter is used to penalize a routing direction when its output channel is currently busy. In case of a busy output channel, the *busy channel* parameter will be set to 1. This means that a routing path with an occupied channel will have its quality value decreased by 1.

The third parameter, *loaded router*, has the value 4 when the neighbor router is too loaded. A router is considered too loaded when its load value exceeds a certain threshold. This threshold value defaults to 50, but it can be specified by the user.

The local load is computed as: $load = \frac{local}{8(6speed + data\,length)}100$. The parameter *local* represents a counter which is incremented each time a new flit is sent. This value is divided to a maximum load value, which is specified by the speedup used to send data flits (compared to head flits), *speed*, and by the length of the message, *data length*.

Since all the parameters used at evaluating a possible route are fixed, this algorithm is static in terms of how the network congestion is accounted for (all nodes which are too loaded are treated in the same way).

4.3.6.2 Self-Optimized Routing

The Self-Optimized algorithm (SO) uses the following formula to evaluate all the possible routes: $quality = direction - \%of\ remaining\ flits - 4pload$.

The first parameter, *direction*, is used for favoring the routing paths which are progressive, i.e. they lead to the destination. A value of 200 is considered for a progressive direction. A non-progressive direction receives no such bonus.

The parameter *%of remaining flits* marks the percentage of packet flits which still have to be sent on the selected output channel. If there is a route set for the packet, the number of remaining flits is divided to the length of the packet. This value penalizes a route more when it will be occupied longer.

The *pload* parameter is the propagated load, described above.

The local load is computed as $load = \frac{local}{\max load} 100$. The value *local* represents the sum of all flits which are in all of the input buffers. *maxload* represents the sum of the sizes of all of the input buffers.

Compared to the SLB algorithm, the SO algorithm is adaptive because it evaluates a route by considering the current network state.

4.3.7 Switching Techniques

The following switching techniques are implemented in the ns-3 NoC simulator: Store-and-Forward (SAF), Virtual Cut-Through (VCT) and Wormhole. Although wormhole switching is mainly used with NoCs[10], having SAF and VCT also implemented allows us to be able to measure the performance of the NoC, given by the switching technique.

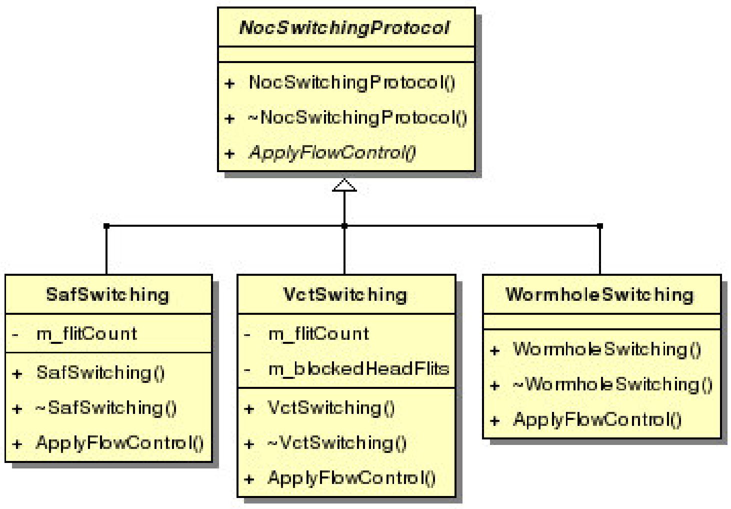

Fig. 35 The ns-3 NoC switching component

Fig. 35 presents the UML class diagram for the ns-e NoC switching component. The abstract class *NocSwitchingProtocol* specifies that any switching technique must implement the *ApplyFlowControl(…)* method, which determines when a given flit is

[10] This is actually the only switching technique implemented in [89]

allowed to be forwarded. Since SAF switching waits an entire packet before forwarding it, we use the *m_flitCount* hash table to store how many flits are still waited for every packet for which the head flit arrived at the switch. VCT uses the *m_blockedHeadFlits* member to know the head flits that remained blocked and cannot currently advance. All the packets with blocked head flits are accumulated in the buffers of the current router before they are allowed to be sent forward. Wormhole switching has the simplest flow control mechanism. A flit is allowed to pass every time the output channel is ready to transmit it.

4.3.8 Traffic Patterns

The traffic patterns currently used by the ns-3 NoC simulator are a set of communication patterns which consider the permutations that are usually performed in parallel numerical algorithms [26]. With these traffic patterns, the destination node for messages sent from a certain node is always the same. Therefore, those traffic patterns do not generate a uniform utilization of the network, but they offer high temporal locality.

Considering that each network node is identified by a unique number, which is binary codified on *n* bits, $(S_{n-1}, S_{n-2}, ..., S_1, S_0)$, we present next the traffic patterns used in this simulator.

4.3.8.1 Bit-complement Traffic

With this traffic pattern, the destination node is obtained by complementing each bit of the source node.

$$(S_{n-1}, S_{n-2}, ..., S_1, S_0) \rightarrow (\overline{S_{n-1}}, \overline{S_{n-2}}, ..., \overline{S_1}, \overline{S_0})$$

This traffic pattern determines packets to be routed diagonally. For example, in a 4x4 mesh, packets from node 1 (mesh coordinates (0, 1)) will be sent to node 14 (mesh coordinates (3, 2)). Note that the numbers have number of bits determined by the size of the mesh (in this case 2 bits).

4.3.8.2 Bit-reverse Traffic

This traffic pattern generates the destination address by reversing the bits of the source address. The first bit from the source will be the last bit of the destination, the second bit of the source becomes the last but one bit of the destination, and so on.

$$(S_{n-1}, S_{n-2}, ..., S_1, S_0) \rightarrow (S_0, S_1, ..., S_{n-2}, S_{n-1})$$

With this traffic pattern, some packets will be sent along one axis, while other packets will be sent diagonally. For example, in a 4x4 mesh, packets from node 5 (mesh coordinates (1, 1)) will be sent to node 10 (mesh coordinates (2, 2)), and packets from node 1 (mesh coordinates (0, 1)) will be sent to node 2 (mesh coordinates (0, 2)).

Note that it is possible for a traffic pattern to generate as a destination node exactly the source node. In such a trivial case, no traffic will actually be generated (e.g.: node 0 would have to send packets to node 0 with this traffic pattern).

4.3.8.3 Matrix-transpose Traffic

This traffic pattern generates the address of the destination node by concatenating the second half of the bits from the source, with the bits which make the first half of the source.

$$\left(S_{n-1}, S_{n-2}, ..., S_1, S_0\right) \rightarrow \left(S_{\frac{n}{2}-1}, S_{\frac{n}{2}-2}, ..., S_0, S_{n-1}, ..., S_{\frac{n}{2}}\right)$$

For example, in a 4x4 mesh, packets from node 6 (mesh coordinates (1, 2)) will be sent to node 9 (mesh coordinates (2, 1)).

4.3.8.4 Uniform random Traffic

Since all of the above traffic patterns do not determine a uniform utilization of the network, the simulator also uses a uniform traffic pattern. With this traffic pattern, the destination is generated without taking into account the source but rather using a uniform pseudorandom number generator.

4.3.9 Network Traffic Generator

Traffic patterns like the ones presented in the above section are (stochastic) micro-benchmarks [45] which typically show only certain network aspects. They also have the advantage of being scalable. They are useful for evaluating a particular design parameter of the communication architecture. For example different routing algorithms may be compared using a uniform traffic pattern because it heavily loads the NoC. However, they do not represent real applications. Benchmarks representing the communication patterns exhibited by real applications, offer better accuracy for measuring the overall performance of an application. Such benchmarks are mandatory when one needs to assess how good a given NoC architecture is for a certain application (domain).

This section describes how ns-3 NoC is able to generate network traffic from applications described through Communication Task Graphs (CTGs). To achieve this goal, we have developed an ns-3 application that implements the Finite State Machine (FSM) described in Section 4.2.2. By doing so, we have adopted the model proposed by the Open Core Protocol International Partnership (OCP-IP) for modeling real applications for NoC benchmarking [45]. OCP-IP currently works with some of the most prestigious NoC research groups from the world[11] to build a suitable benchmarking methodology for Network-on-Chip simulation [95].

The CTG describes the communications that occur between the tasks of an application. For each application task, the CTG tells us to what tasks it sends data, how much data is communicated to each task and from what tasks data must be received so that the communication may be started.

We also need to know to what cores the tasks of the application are assigned. This information resides in the Application Characterization Graph (APCG). Based on a CTG,

[11] Carnegie Mellon University, Massachusetts Institute of Technology, University of California at Berkeley and many others

the APCG is obtained using a scheduling algorithm. Having each task assigned to a core, we know the time required for the execution of every task.

All this information allows us to instantiate the Finite State Machine that describes how an IP core executes instructions. The FSM is implemented by an ns-3 application called **NocCtgApplication**.

Our simulator can use the mapping information (obtained with any application mapping algorithm) to assign a *NocCtgApplication* to each NoC node that has an IP core assigned to it.

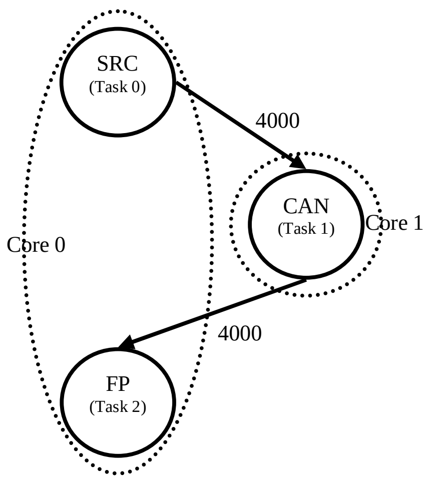

Fig. 36 Simple APCG with two tasks assigned to the same IP core

Each *NocCtgApplication* is programmed with information from the CTG and the APCG so that the FSM is able to mimic the execution of the IP core. Each CTG ns-3 application contains a list of tasks (called *task sender list* or *remote task list*) from which it must receive a certain volume of data before it can start its execution. Additionally, the CTG ns-3 application holds a list with all the tasks to which it will send data (called *task destination list* or *local task list*). After it receives all the data, the *NocCtgApplication* will wait for a period of time given by the task execution time, and then it will start sending data to all the tasks. The ns-3 application injects the data into the network in the form of packets that have a user customizable number of flits and payload data size.

The FSM proposed in [45] (see Section 4.2.2) contains control information regarding data dependencies. An example of such control information is: if an IP core received enough data, let it send a certain amount of data to another specified IP core. More control information is needed to program such Finite State Machine that models IP cores which execute multiple tasks. The problem is to know how the emulated IP core executes several tasks. The execution may be sequential, multithreaded or parallel. Even if tasks are executed sequentially, the FSM must have an execution schedule. Consider the Figure 36 APCG as example. In this case, tasks 0 and 2 are assigned to core 0. They have no dependency between them so, from the point of view of core 0, tasks 0 and 2 could be executed in any order. Still, task 1 is assigned to core 1 and the first task sends 4000 bits to task 1 which then sends 4000 tasks to task 2. Therefore, the FSM associated to IP core 0 must first execute task 0, send its data and then wait to receive 4000 bits from core 1 before starting to execute task 2. If the FSM associated with IP core 0 is simply programmed to know that it must receive data from IP core 1, trying to simulate the communication described in the APCG will result in a deadlock. This is because IP core 0 cannot send data until it receives 4000 bits from core 1. Core 1 also must wait for 4000 bits from core 0.

In order to avoid such kind of problems, our *NocCtgApplication*s automatically identifies which tasks are independent, i.e. they can execute and inject data into the NoC immediately since they do not wait for data from any other IP core. Therefore, our developed traffic generator allows each IP core to execute its independent tasks even if it has other dependent tasks. Regarding task scheduling, our traffic generator simply programs each IP core to execute its tasks in a (random) sequential order. Our experience gained with developing this traffic generator makes us to believe more rules are required for programming the Finite State Machines. Otherwise, the traffic generator might not model the real application correctly. We showed it may easily deadlock and we proposed a simple solution to avoid this problem. However, more work is required. In our opinion a scheduling policy is also needed for making the traffic generator accurately. OCP-IP recently published a work in progress traffic generator [95]. Its development started in parallel with ours, starting from the same FSM model. Their traffic generator works by defining a lot of rules for the FSMs, which are practically equivalent to a manual task scheduling. We argue the static, predetermined task scheduling might become cumbersome for complex benchmarks. It however has the advantage of keeping the traffic generator simple.

Like OCP-IP's traffic generator, our traffic generator considers that the root CTG tasks may have a period assigned to them. This period specifies with what frequency these tasks produce new data. By doing so, the Communication Task Graph can be reiterated. During application running, we can have multiple CTG iterations in flight.

Our ns-3 NoC traffic generator is mature enough to correctly model the benchmarks we used in this thesis. This is because these benchmarks do not have more than two tasks assigned per IP core. In this case solving the problem presented in the above example was enough to make our traffic generator work correctly. However OCP-IP's traffic generator allows for more flexibility and accuracy than ours. This is why, as further work, we plan to integrate it with ns-3 NoC.

The following UML diagram presents the main characteristics of the CTG ns-3 application.

NocCtgApplication is an ns-3 Application that knows to inject packets into the network based on the information provided by a CTG (and its associated APCG). The user specifies how many flits a packet contains (m_numberOfFlits) and also the payload size (m_flitSize) of each flit (in bytes). The field m_taskList contains a list with all the tasks assigned to the IP core represented by this ns-3 application. For each task, we know its execution time (m_execTime). While the helper class TaskData is used to keep the information about the tasks assigned to the IP core, the class DependentTaskData is used to keep the information about the tasks from the task sender and

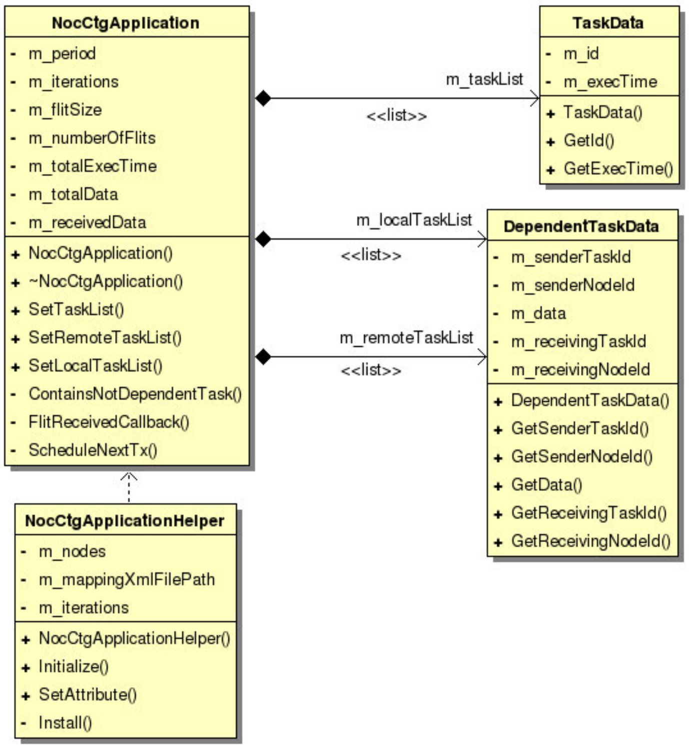

Fig. 37 UML class diagram for the ns-3 NoC network traffic generator

destination lists (m_ taskRemoteList and respectively m_taskLocalList). Each DependentTaskData object models a data dependency between two tasks.

A sender (remote) task is a task that sends data to the IP core (modeled by a *NocCtgApplication* object). It has a unique identifier (*m_senderTaskId*) and it is assigned to another IP core which is mapped to a certain NoC node (*m_senderNodeId*). The volume of data is kept in the field *m_data* (in bits). The task of the IP core that receives information from the sender task is identified through an ID (*m_receivingTaskId*) and through the associated NoC node (*m_receivingNodeId*).

A destination (local) task is a task to which data is sent by the IP core. The IP core's task which sends information to the destination task is marked through the fields *m_senderTaskId* and *m_senderNodeId*. The destination task is identified by *m_receivingTaskId* and *m_receivingNodeId*. Note that *m_receivingNodeId* is used by the *NocCtgApplication* to know to what NoC node it has to send packets.

The CTG ns-3 application starts in a waiting state. First, it waits to receive all the data from the tasks which send packets to the IP core. The field *m_receivedData* counts the bits received. The application is automatically notified when it receives packets through the callback *FlitReceivedCallback()*. When all data is received (*m_receivedData* = *m_totalData*), the application will simulate that it is executing. It will therefore wait for an amount of time equal to the execution time. Then, the application will start injecting packets into the network until it sends all its data. At each network clock cycle one flit is injected. The application schedules itself for another data transmission using the method *ScheduleNextTx()*. As mentioned earlier, non-dependent tasks are executed without waiting.

Each *NocCtgApplication* is executed for a specified number of iterations (*m_iterations*). A CTG iteration ends when all the data transmitted by all the IP cores has been received by the destination NoC nodes. New CTG iterations start after a specified CTG period (m_period). We may have multiple CTG iterations running at a moment in time.

NocCtgApplicationHelper is a helper class that allows the user to easily instantiate and install CTG ns-3 applications. It requires the user to specify the tasks assigned to the IP core, the task sender and destination lists, the CTG period and the number of CTG iterations.

A runnable instance of the ns-3 NoC simulator, called *NocMappingSimulator* uses the XML interface described in section 4.2.3 in order to get the network traffic pattern of an application described through a CTG. *NocMappingSimulator* uses [84] to read the required information from XMLs. It accesses the CTG, APCG and mapping data and then, based on all this information, it installs CTG ns-3 applications which simulate the traffic pattern of an application described through a Communication Task Graph. Obviously, the *NocMappingSimulator* also configures the available architectural parameters of the simulated Network-on-Chip.

4.3.10 Power Consumption and Area Estimation

This section describes how we integrated ORION 2.0 [96] into ns-3 NoC. ORION is a power and area simulator for Networks-on-Chip. It is developed at Princeton University for almost ten years. ORION offered one of the first power models for NoCs and it is now widely used by the research community for estimating NoC power consumption.

The initial ORION version offered a power model for NoC routers. Dynamic and leakage power models for the router's basic components (buffers, crossbars and arbiters) were proposed. ORION 2.0 is the latest version of this simulator. Compared to the initial one, it brings substantial improvements. Some of the most important are: link and clock power models and router and link area models. The technology models were also updated. Three operating types (high, normal and low power modes) are available at 90 nm and 65 nm technologies, scaling down up to 32 nm.

There are three major power components [97]: dynamic power, short circuit power and leakage power. Dynamic power is the power due to charging circuit nodes. Also called active power, this is used while the circuit is performing its functions. Short circuit power appears because of temporary short circuit current path while switching. It is usually small and thus ignored by architects. Leakage power is primarily due to an unwanted sub-threshold current in the transistor channel, when the transistor is turned off. Hence, leakage power in unintended as it does not contribute to the circuit's functions. As technology goes below 65 nm, leakage power becomes more important, as compared to dynamic power.

ORION formulates dynamic power as $P = E \cdot f$, where energy $E = \alpha \cdot C \cdot V_{dd}^2$. f is the clock frequency, α the activity factor, C the load capacitance and V_{dd} the supply voltage. The clock dynamic power model for clock, registers, buffers, allocators, arbiters and links are all obtained by defining how the load capacitance is computed. For example, the link load capacitance is $C_l = C_{in} + C_{gnd} + C_{cc}$.

Parameter C_{in} is the input capacitance of the next repeater, C_{gnd} and C_{cc} the ground and coupling capacitance of the wire. The values for different load capacitance components are obtained from different industry data sheets.

Leakage power is computed by measuring the total leakage current. ORION formulates this current as $I_{leak}(i,s) = W(i,s) \cdot (I'_{sub}(i,s) + I'_{gate}(i,s))$.

I'_{sub} and I'_{gate} are the sub-threshold and respectively gate leakage currents, per unit transistor width, for a specific technology. $W(i,s)$ is the effective transistor width of component *i* at input state *s*. For each *i* and *s*, I'_{sub} and I'_{gate} were obtained through simulations using models at 65 nm technology.

Router and link area is computed using a layout model for logic gates. Router area is estimated by summing the areas of all its building blocks (plus 10%, representing the spaces between the blocks). The area of a block is computed by decomposing it in its logic gates.

Buffer's area is computed by considering the word line and bit line lengths of the FIFO (First In, First Out) buffer: $Area_{fifo} = L_{word-line} \cdot L_{bit-line}$, where $L_{word-line} = F(w_{cell} + 2(P_r + P_w)d_w)$ and $L_{bit-line} = B(h_{cell} + (P_r + P_w)d_w)$.
B is the buffer size, in flits. *F* is the flit size, in bits. The memory cell width and height is given by w_{cell} and h_{cell} (d_w accounts for wire spacing). The number of read and respectively write ports are noted with P_r and P_w.

Link (wire) area is computed as follows: $Area_{link} = F(w_w + s_w) + s_w$, where w_w is the wire width and s_w is the spacing between wires.

We integrated the ORION 2.0 library in our developed ns-3 NoC simulator. ORION keeps all its parameters into a single configuration file (SIM_port.h). We kept the default values for all parameters, except a few, which are given by our simulator, at runtime. For example, we consider the designed NoC at 90 nm technology. Another example is the voltage supply, set to 1 Volt. The NoC clock frequency, flit and buffer size are set by ns-3 NoC. The other parameters set by ns-3 NoC are related to router architecture: number of router input and output ports and number of virtual channels. This allows us to model more accurately each ns-3 NoC router. For example, the Irvine router, as compared to the four way router, has a different number of input ports. The number of virtual channels is currently set to zero because we do not have yet virtual channels in ns-3 NoC.

The following UML diagram shows how ORION library interacts with ns-3 NoC.

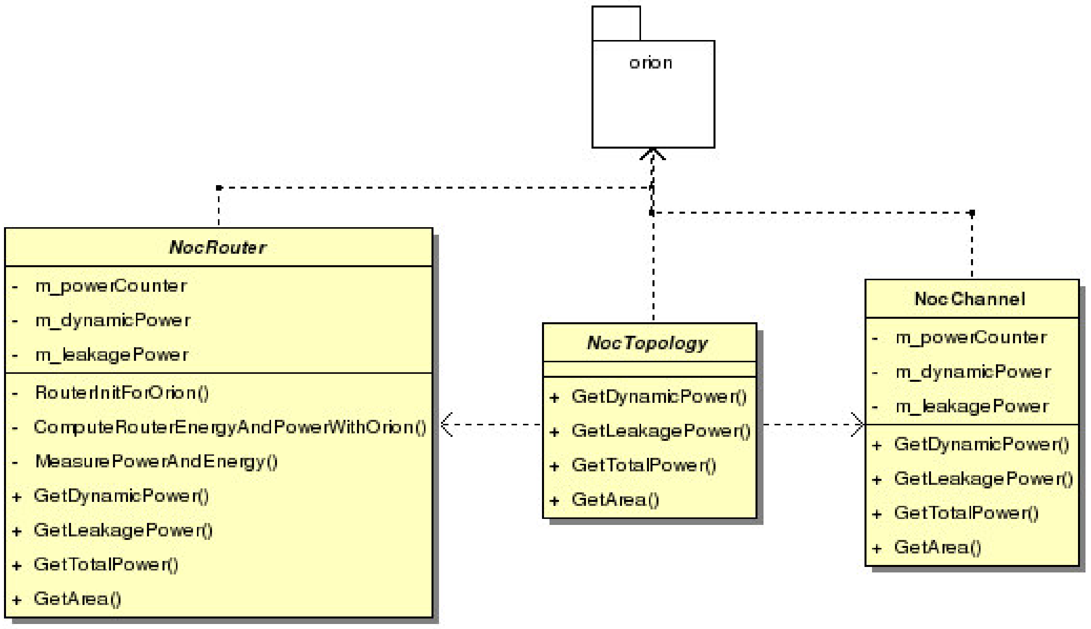

Fig. 38 ORION integration in ns-3 NoC

ORION is called by ns-3 NoC each time a flit is routed and each time a flit is transmitted through a link. For every flit, ns-3 NoC gets the dynamic and leakage power needed to route and transmit it. We use the *m_dynamicPower* and *m_leakagePower* members to sum all the measured power values. By counting the flits for which power was measured (*m_powerCounter*), ns-3 NoC can provide at any moment the average power consumed by all its routers and links. The power consumed by the entire NoC is computed with the methods from the *NocTopology* class. The total power consumed by the NoC (*GetTotalPower()*) is obtained by summing the dynamic and leakage powers. Similarly, the area occupied by routers and channels is computed. The *GetArea()* method from *NocTopology* class computes the area of the entire Network-on-Chip. This method accounts for all NoC routers and channels. If we also take into consideration the IP cores' area, we will know the area occupied by the entire System-on-Chip (SoC). The same rationing applies for power consumption. Obviously, since we also measure the application runtime, we can compute the energy consumption. Energy is measured by multiplying the total (average) power by application runtime.

4.3.11 Experimental Results

The ns-3 NoC simulator keeps statistics regarding the number of flits and packets injected into the network. Using ORION, our simulator can measure NoC power consumption and area. The latency of the packets is measured according to the definition from [25]: “the latency of a network is the time required for a packet to traverse the network, from the time the head of the packet arrives at the input port to the time the tail of the packet departs the output port”. The minimum and maximum packet latencies are also collected. All those statistics are automatically stored into a database.

We present next some preliminary simulation results published in [98], were we evaluated the potential of the NoC Irvine architecture and were we showed the impact of the buffers’ size on NoC’s performance. The following results express the network performance, through the average latency of the packets, as a function of packet injection probability. The synchronous version of the simulator was used. During the simulation, the first 1000 cycles were considered warm-up cycles [25]. Packets were injected into the network for 10000 cycles. Only the packets injected after the warm-up cycles were collected into the statistics. The Irvine architecture was used, with XY routing, wormhole switching, input channel buffers of 9 flits in size and packets of 8 flits in length. We set the flit size to a few bytes. The effects of speeding up the data flits, like it is done in [90], are shown it the following charts.

Fig. 39 shows how, on a 8x8 Irvine NoC, the average packet latency decreases as data flits are sent through the network using a clock frequency which is two or four times higher than the one used for advancing the head flits.

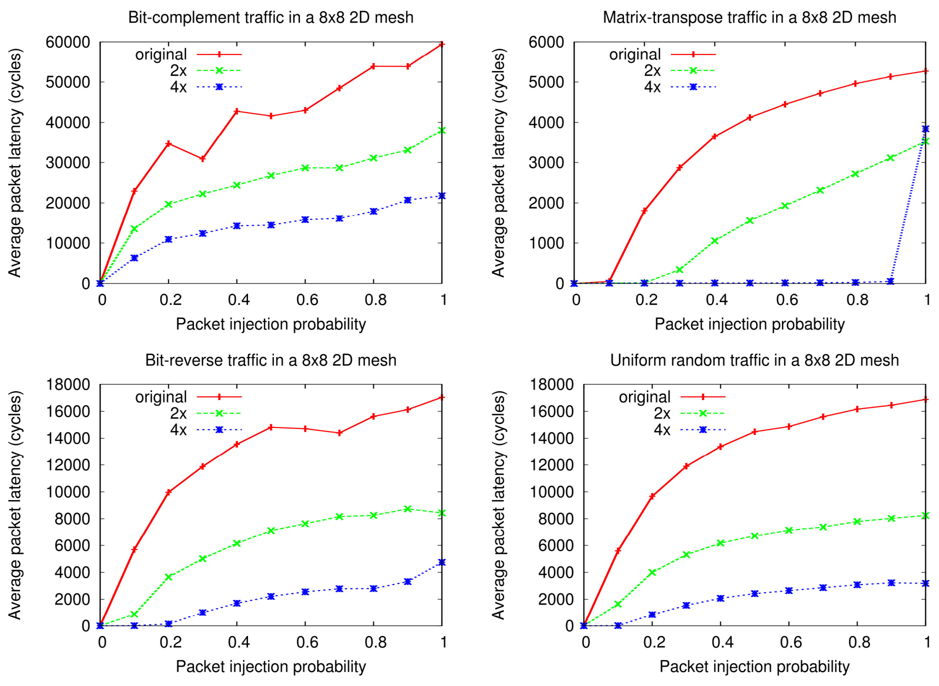

Fig. 39 The average packet latency on a 8x8 Irvine NoC architecture, while the speed with which data flits advance in the network varies for 4 different communication patterns

With the matrix-transpose traffic pattern and using a 4 times higher clock frequency for the data flits, the packet's average latency remains close to the zero-load latency, as long as the injection probability is lower or equal than 0.9. The Irvine architecture helps at decreasing the network congestion. This is also visible for the other three traffic patterns. The network is significantly less congested when data flits are transmitted faster than head flits. For the bit-complement traffic pattern, the average packet latency is fairly higher because each node injects packets. This is not true with the other traffic patterns because they can create traffic from a certain node to exactly the same node, which is not injected into the network. Therefore, we believe that this behavior might contribute to the bit-complement's higher packet latency.

We did similar simulations on a 4x4 Irvine NoC, too. The simulations on an 8x8 Irvine NoC took approximately 10 times more time than the simulations done on the 4x4 network. The longest simulation on a 4x4 network took around 2 minutes and a half (this is approximately 10 times faster than NoCSim [89]).

In order to further test the speed of the simulator, some simulations were also performed on a 32x32 2D mesh (1024 nodes). Because the simulation would take considerable much more time on such a big network, the simulation was run for only 10 cycles, with no warm-up period. The longest simulation took around six minutes and a half on a PC having the following characteristics: Intel quad-core at 2.66 GHz, 4 GB DRAM and a Linux Operating System (Ubuntu 9.10).

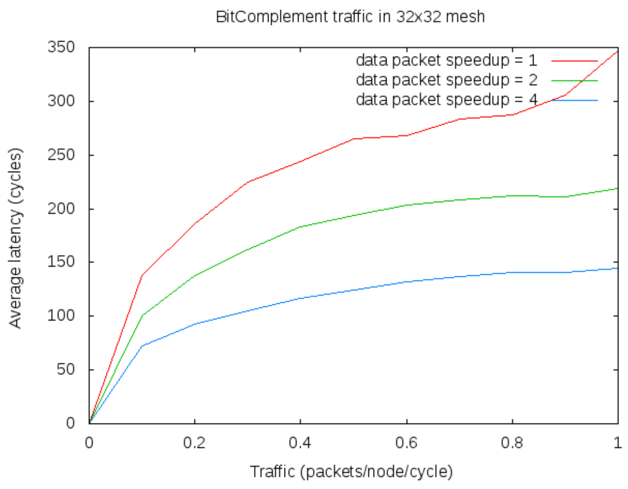

Fig. 40 Bit-Complement traffic, 32x32 Irvine NoC, wormhole switching, XY routing

The following simulation result shows how the ns-3 NoC simulator can be used at evaluating the impact of the available buffering resources on the NoC's performance.

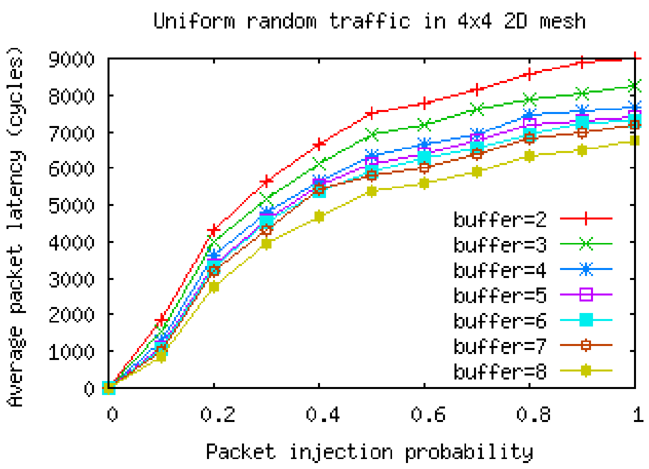

Fig. 41 The average packet latency on a 4x4 Irvine NoC architecture, while the size of the input buffers varies uniformly

As it can be observed, the more buffering resources are available, the better the performance of the NoC architecture gets. As expected, the size of the input channel buffers becomes more important as the number of packets injected into the network increases. The simulations were run using the uniform random traffic pattern, for 10000 clock cycles, with 1000 warm-up cycles. At each cycle, a flit can be injected in any node of the network, with a certain probability of injection. XY routing with wormhole switching has been used on a 4x4 Irvine NoC architecture. No speedup has been used for the data flits. Each packet has 9 flits, and the size of the input channels was varied uniformly, from 2 up to 8 flits.

In [92] we showed how the NoC performance varies on topologies like: 2D mesh, 2D torus, 3D mesh, 3D torus and hypercube. For example, the following figure shows the buffer size influence on the performance of a (2x2x2x2) hypercube. Unless specified otherwise, the simulator’s parameters have the same values as before.

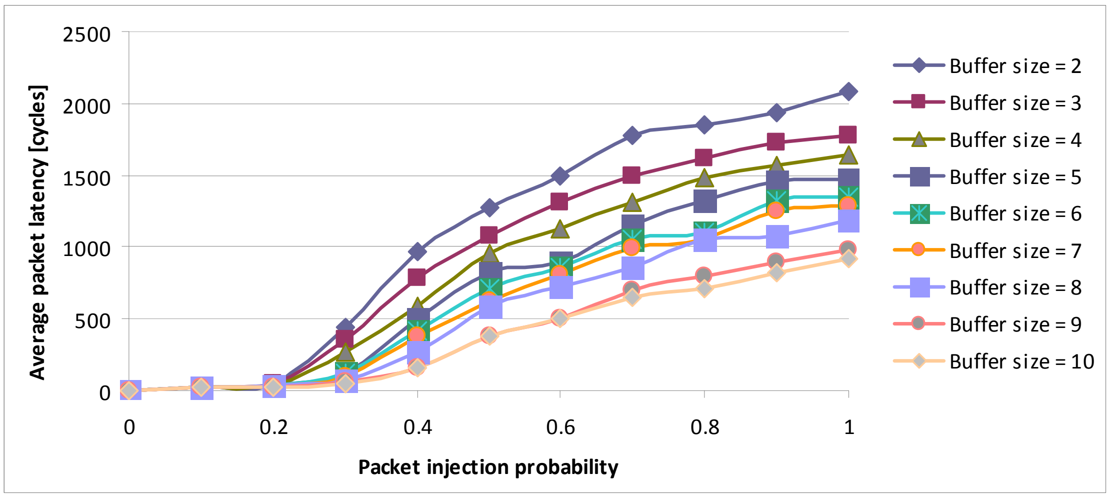

Fig. 42 The average packet latency on a hypercube NoC architecture, while the size of the input buffers varies uniformly

Like in the case of the 4x4 2D mesh, the Network-on-Chip's performance improves as the buffer size increases. We can also observe how the hypercube's increased connectivity decreases approximately four times the average packet latency, as compared to a 2D mesh with the same number of NoC nodes.

We show next how the NoC's average packet latency decreases as we increase the node degree by switching from a 2D mesh to a 3D mesh and then to a hypercube. The simulations were made using the uniform random traffic pattern.

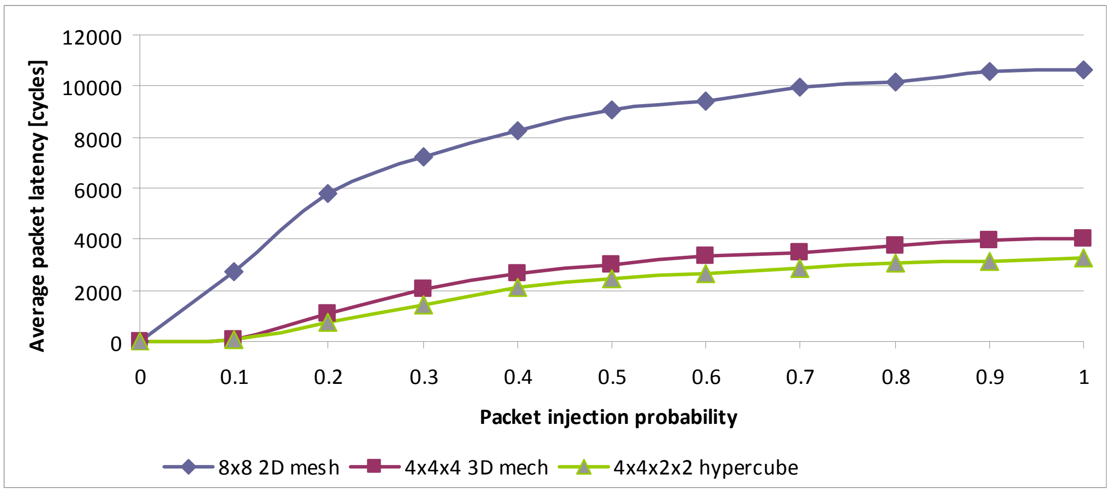

Fig. 43 Average packet latency on 64 node mesh NoCs, with 2, 3 and respectively 4 dimensions

We observe a significant increase in the NoC's performance when using a 3D mesh. The performance increases even further when placing the 64 nodes in a hypercube topology. The same behavior can be observed on torus topologies.

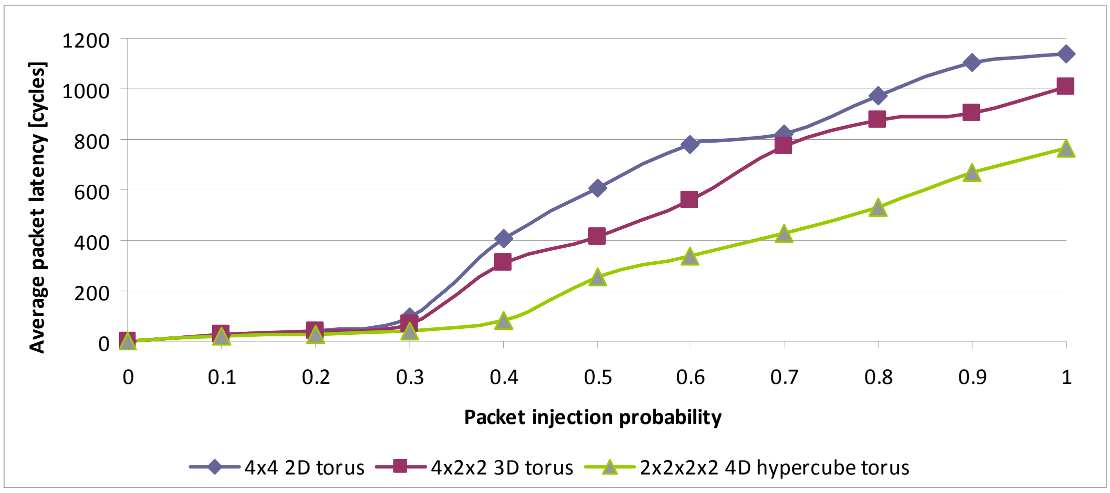

Fig. 44 Average packet latency on 64 node torus NoCs, with 2, 3 and respectively 4 dimensions

Note that since Dimension Order Routing is not deadlock-free on torus topologies and because ns-3 NoC does not yet support virtual channels, we increased the size of the input buffers until deadlock was avoided. Thus, we used buffers with a size of 55 flits (instead of 9). These big buffers explain why the average packet latency appears to be ten times lower on torus topologies than on meshes.

4.4 Summary

We presented in this chapter UniMap, our developed unified framework for the evaluation and optimization of Network-on-Chip application mapping problems. UniMap's design is flexible, modular, reusable and scalable.

Using XML schemas we defined a model for representing real applications (through Communication Task Graphs and Application Characterization Graphs) and Network-on-Chip architectures. Any researcher can easily import his application into our framework. We will show in Chapter 5 the benchmarks integrated and used by us.

UniMap allows anyone to easily implement scheduling and/or mapping algorithms. Then these algorithms can be evaluated and compared with the algorithms already implemented in our unified framework. UniMap integrated state of the art NoC application mapping algorithms, like Simulated Annealing and Branch and Bound. Using the jMetal library we also make available some of the best single-objective and multi-objective evolutionary algorithms currently available in the research community. As we will prove in Chapters 6 and 7, UniMap can also be used to optimize Network-on-Chip application mapping algorithms.

The Network-on-Chip application mappings can be evaluated in UniMap using either analytical models or a modular and scalable NoC simulator. Our developed ns-3 NoC simulator is highly parameterizable. It currently allows the user to specify: the packet size, packet injection rate, buffer size, network size, switching mechanism (Store-and-Forward, Virtual Cut-Through and Wormhole), routing protocol (XY, YX and two adaptive protocols that consider the network's load), network topology and traffic patterns. It can evaluate the simulated NoC in terms of network latency and throughput. ns-3 NoC can instantiate any k-ary d-cube network topology. It also contains a network traffic generator based on CTGs and APCGs. Our simulator can also estimate power consumption and area using the state of the art ORION 2.0 tool.

Besides the NoC simulator, UniMap can model Processing Elements using Finite State Machines. Therefore entire System-on-Chip architectures can be modeled with our UniMap framework. We will demonstrate this in Chapter 8.

UniMap is also capable of running on High Performance Computing systems. As we will show in Chapter 8, this is a very important aspect, a requirement, when addressing the Design Space Exploration problem.

For the rest of this thesis we will present how we made use of UniMap. We believe our unified framework has big potential and since it is open source it could help other researchers as well. This is even more important, knowing the Network-on-Chip research area is still at the beginning and powerful tools are still needed.

More details regarding UniMap's design are available in [85].

“Light is the task where many share the toil.”

Homer

5 Benchmarks

In the previous chapter we presented our developed unified framework for the evaluation and optimization of Network-on-Chip application mapping algorithm. UniMap uses as input traffic patterns for real applications, described through directed graphs. As stated in [45] Network-on-Chip benchmarking is still an open problem. The Open Core Protocol International Partnership (OCP-IP) is currently working to model real applications for NoC benchmarking [95].

This chapter presents the benchmarks used in this PhD thesis, for studying the Network-on-Chip application mapping problem. All benchmarks presented next describe real applications, designed for Systems-on-Chip (SoCs). These applications are modeled using Communication Task Graphs (see Section 3.1). We gathered some of the most used CTGs and APCGs by the NoC research community and integrated them in UniMap, through a common XML representation. The communication graphs are taken from the Embedded Systems Synthesis Benchmark Suite (E3S) [56] and from some of the most cited papers from the field of Networks-on-Chip. We also make our contribution to Network-on-Chip benchmarking, by proposing two new Communication Task Graphs for a H.264 video decoder.

As compared to the general CTG definition from [3], we specify that a Communication Task Graph should also have a period assigned to it. The *period* of a task graph is defined as the amount of time between the earliest start times of its consecutive executions [44]. Such information is used in E3S (see section 4.2.1.2.2). The period allows us to specify the application bandwidth requirements. Typically, the period is measured in seconds. Let us consider a CTG with period of 0.1 seconds. If a graph arc between two tasks, T_1 and T_2 has a weight of 10 bits (communication volumes are expressed in bits), it will mean that the communication between the two tasks will require a bandwidth of 100 bits/second.

We begin by presenting the Communication Task Graphs for each real application. Then we show the Application Characterization Graphs, i.e. how we group the application tasks in order to assign them to IP cores.

5.1 Embedded System Synthesis Benchmarks Suite (E3S)

This benchmark suite contains task graphs for five embedded applications: automotive/industrial, consumer, networking, office automation and telecommunications. Each application is described by a set of task graphs. There is a version of each task graph for three kinds of computing systems: distributed, wireless client-server and System-on-Chip. We use in our research only the Communication Task Graphs designed for Systems-on-Chip.

This benchmark suite is based on the Embedded Microprocessor Benchmark Consortium (EEMBC) [80] benchmarks suite. It contains 34 embedded processors that are characterized through: execution time and power consumption (measured on 47 tasks), die size, price, operating frequency and other information. E3S also contains communication resources that model different buses.

Except “src” and “sink”, each task from the E3S Communication Task Graphs represents an EEMBC benchmark. The tasks “src” and “sink” are just dummy tasks that model a data producer and, respectively, a data consumer.

We also have to note that E3S task graphs also contain hard- and/or soft-real-time deadlines. Since we do not work with real-time constraints, we disregard deadlines from E3S task graphs.

5.1.1 Automotive/industrial Application

The automotive/industrial application is made of four Communication Task Graphs: CTG 0, 1, 2 and 3.

CTG 0 models an embedded automotive system with two Controller Area Network (CAN) interfaces, a basic integer and floating point module and an actuator driven by Pulse Width Modulation (PWM) signal.

The first CAN interface (can1) simulates the processing of Remote Data Request (RDR) messages which are intended for the basic integer and floating point module (fp).

The floating point module computes the function $\arctan(x)$ using the telescoping series: $\arctan(x) = x \cdot \frac{P(x^2)}{Q(x^2)}$, where P and Q are two polynomials and $x \in [0,1]$.

The second CAN interface processes the RDR messages originated from the fp module and sends them to a PWN driven actuator. The actuator commands a stepper motor. PWN signals are simulated and it is verified, once per PWM cycle, if the motor reached the commanded position.

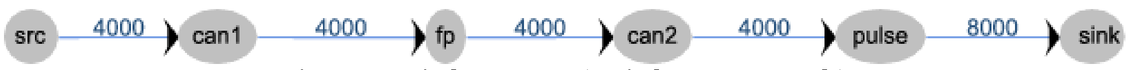

Fig. 45 auto-indust CTG 0 (period: 0.0009 seconds)

CTG 1 describes an embedded automotive system with an Infinite Impulse Response (IIR) filter and an inverse Discrete Cosine Transform (iDCT) module.

The IIR filter tests the processor’s capability of performing multiply-accumulates and rounding.

The iDCT module performs image recognition by applying an inverse discrete cosine transform on an input data matrix set using 64-bit integer arithmetic.

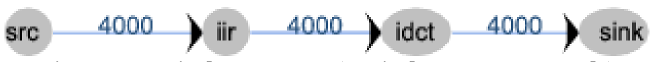

Fig. 46 auto-indust CTG 1 (period: 0.00045 seconds)

CTG 2 models a system with: Finite Impulse Response (FIR) filter, Fast Fourier Transform (FFT) module, matrix arithmetic module, Inverse Fast Fourier Transform (iFFT) module, angle to time convertor, road speed calculation and table lookup and interpolation.

The FFT module performs a power spectrum analysis of a time varying input waveform.

The matrix arithmetic module performs a LU decomposition of a square matrix, computes the determinant of the input matrix and the cross product with another matrix.

The iFFT module does a time domain analysis of an input frequency spectrum.

The angle to time convertor measures the delays between the pulses received from a toothed wheel (gear) on the crankshaft of an engine. These pulses represent the crankshaft's angle and the delay between them its angular velocity, i.e. engine's speed.

Road speed is computed based on differences between timer counter values.

Table lookup and interpolation is used by in engine controllers, anti-lock brake systems or any other system where data must be directly accessed, not computed. Sensitive data is thus stored in a table. In order to keep the size of the table small, only some data points are memorized. The rest of them are obtained through interpolation.

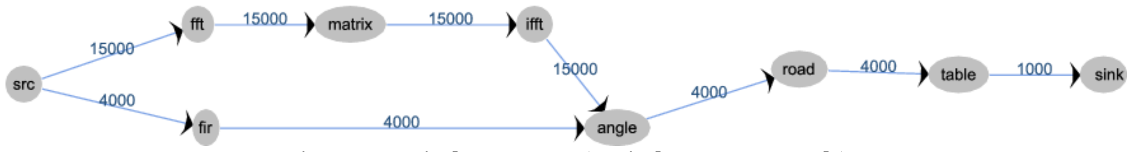

Fig. 47 auto-indust CTG 2 (period: 0.0009 seconds)

CTG 3 contains the following modules: pointer chasing (ptr), cache "buster" and tooth to spark.

Pointer chasing performs a lot of pointer manipulation: a double linked list is searched for entries matching a given items.

Cache "buster" is an application that uses data and function pointers so that data and code locality does not occur.

Tooth-to-Spark is part of the Engine Control Unit (ECU) and performs real-time processing of air/fuel mixture and ignition timing. It controls the fuel injection and ignition in the engine.

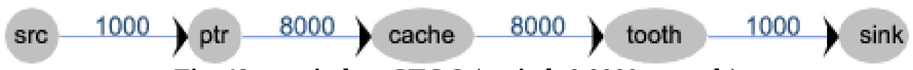

Fig. 48 auto-indust CTG 3 (period: 0.0009 seconds)

5.1.2 Consumer Application

The consumer application is composed of two modules: an image acquisition module from a video camera (CTG 0) and an image printer module (CTG 1).

The tasks from CTG 0 deal with image acquisition, filtering (applied on the red, green and blue color components of the image), color space conversion (from RGB to YIQ), compression using JPEG and finally storage.

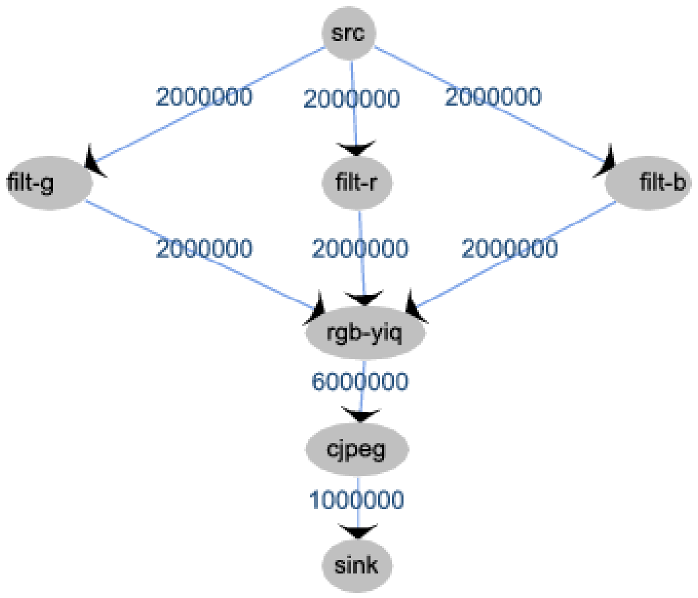

Fig. 49 consumer CTG 0 (period: 0.06 seconds)

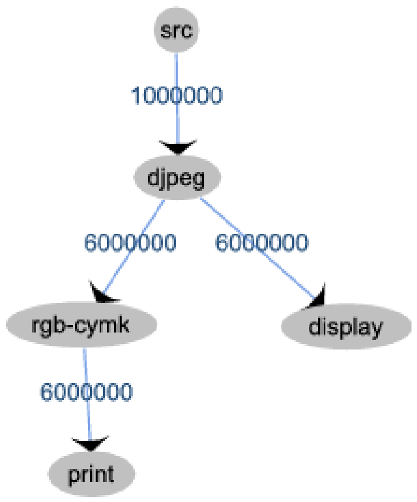

Fig. 50 consumer CTG 1 (period: 0.015 seconds)

CTG 1 models image retrieval from the data storage, JPEG image decompression, conversion from RGB color model to CMYK color model and the printing. The images to be printed may also be viewed on a display.

5.1.3 Networking Application

This application models an Internet network router. An Open Shortest First Path (OSPF) task (that represents CTG 0) implements Dijkstra's shortest path first algorithm to build a routing table. Route lookup is then performed for network packets using an IP lookup mechanism based on a Patricia Tree. The packet flow (pf) task changes the packet header to mark the passage of that packet through the router.

CTG 1 models a router through which packets with size 512 kB pass.

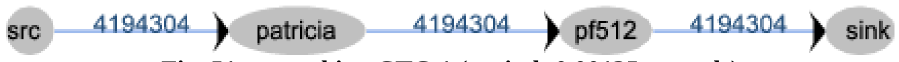

Fig. 51 networking CTG 1 (period: 0.00135 seconds)

CTG 2 models a router through which packets with size 1 MB pass.

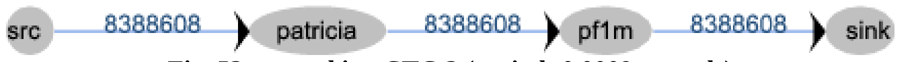

Fig. 52 networking CTG 2 (period: 0.0009 seconds)

CTG 2 models a router through which packets with size 2 MB pass.

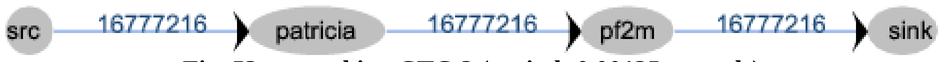

Fig. 53 networking CTG 3 (period: 0.00135 seconds)

5.1.4 Office Automation Application

The office automation application contains tasks for image and text processing, performed by a printer.

The rotate task performs a bitmap rotation algorithm to create a 90° rotation of an image. This is useful for printers, when switching between portrait and landscape modes.

The dithering task (dith) executes the Floyd-Steinberg Error Diffusion dithering algorithm to convert a grayscale image into a form ready for printing.

Text parsing is performed by the text task. It represents a printer application that parses an interpretative control language like PCL or PostScript.

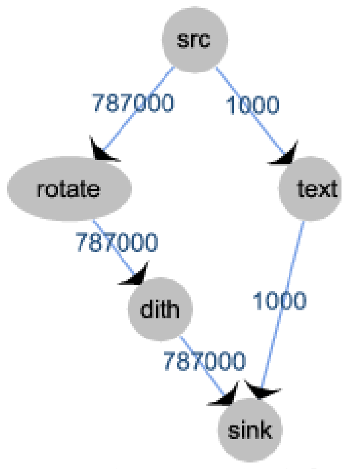

Fig. 54 office-automation CTG 0 (period: 0.03 seconds)

5.1.5 Telecommunication Application

The E3S telecommunication application is composed of nine Communication Task Graphs. The following tasks are employed: autocorrelation (ac), bit allocation (fpba), convolutional encoder (ce), fast Fourier transform (fft) and a GSM Viterbi decoder (gsm).

CTG 0 performs autocorrelation and convolutional encoding. Autocorrelation is a basic analysis tool in signal processing. In telecommunications, it is used for speech compression and recognition, channel estimation, sequence estimation, system identification and other applications. A convolutional encoder is installed at the sender. It adds redundancy to a transmitted electromagnetic signal to support forward error correction on the receiver side.

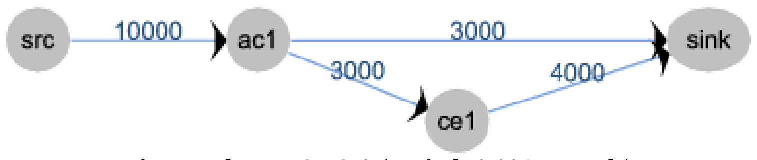

Fig. 55 telecom CTG 0 (period: 0.001 seconds)

CTGs 1 and 2 are identical. Each one of them contains the following tasks: autocorrelation, convolutional encoder, bit allocation and Fast Fourier Transform (FFT).

A bit allocation algorithm is used for Digital Subscriber Loop (DSL) modems that use Discrete Multi-Tone (DMT) technology (which partitions a channel into independent subchannels, called carriers). Such an algorithm is required to allocate a number of bits to the channel's carriers, according to each carrier's measured Signal to Noise Ratio (SNR).

Fast Fourier Transform is used for a frequency analysis of signal. In telecommunications, FFT allows for: filtering frequency-dependent noise or interference of a transmission, identifying the information content of a frequency-modulated signal etc.

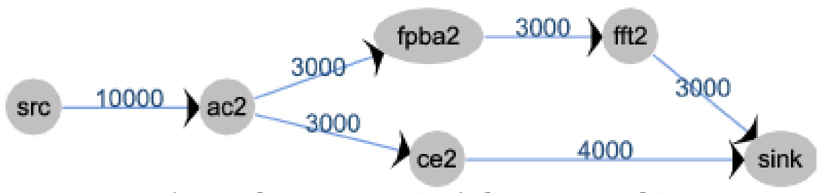

Fig. 56 telecom CTG 1 (period: 0.001 seconds)

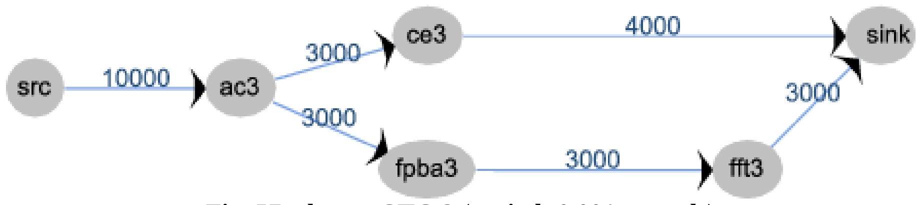

Fig. 57 telecom CTG 2 (period: 0.001 seconds)

CTG 3 is a simple graph, which contains just a Fast Fourier Transform task.

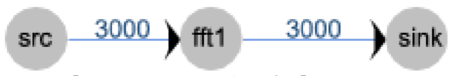

Fig. 58 telecom CTG 3 (period: 0.001 seconds)

CTG 4 is a simple graph, which contains only a bit allocation task.

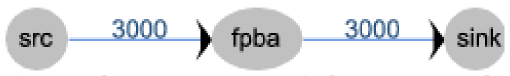

Fig. 59 telecom CTG 4 (period: 0.001 seconds)

The last Communication Task Graphs are identical. They employ a Viterbi decoder for GSM cellular telephony. The Viterbi decoder exploits a data stream's redundancy in order to recover the originally transmitted data. Such a decoder is placed on the receiver side, while on the sender side of a transmission is placed a convolutional encoder.

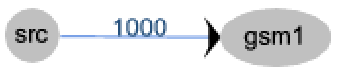

Fig. 60 telecom CTG 5 (period: 0.00033 seconds)

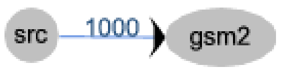

Fig. 61 telecom CTG 6 (period: 0.00033 seconds)

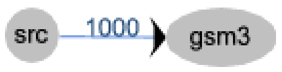

Fig. 62 telecom CTG 7 (period: 0.00033 seconds)

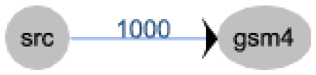

Fig. 63 telecom CTG 8 (period: 0.00033 seconds)

5.2 Audio Video Benchmarks

We present next a suite of benchmarks related to audio and/or video processing. Like E3S benchmark suite, these Communication Task Graphs are used by the research community for developing and evaluating Network-on-Chip application mapping algorithms.

5.2.1 MultiMedia System (MMS)

A generic Multimedia System (MMS) was used in [52], [40] to test the performance of a Branch and Bound mapping algorithm on a real application. MMS is an audio video system. It contains an MP3 audio encoder, an MP3 audio decoder, an H.263 video encoder and an H.263 video decoder.

The MMS application was partitioned into 40 concurrent tasks. They were assigned to 16 cores in [52] and to 25 cores in [40].

We present next the Communication Task Graphs for the MMS application, which we derived from the Application Characterization Graphs (APCGs) [3]. The communications between tasks mapped on the same cores are ignored. Therefore, our MMS CTGs are partial. They do not show all the communications between tasks but, all the communications required for building a 16 cores and 25 cores APCG are available.

CTG 0 was obtained from the MMS APCG from [52]. CTG 1 was obtained from the MMS APCG from [40]. We observe a single difference between the two CTGs: in CTG 0, task VLD (Variable Length Decoding) sends data to IDCT (Inverse Discrete Cosine Transform), while in CTG 1, VLD sends data to task IQ (Inverse Quantization). Thus, we believe the two CTGs are essentially the same. However, in order to be compatible with previous research, we use both of them, exactly like in [52], [40].

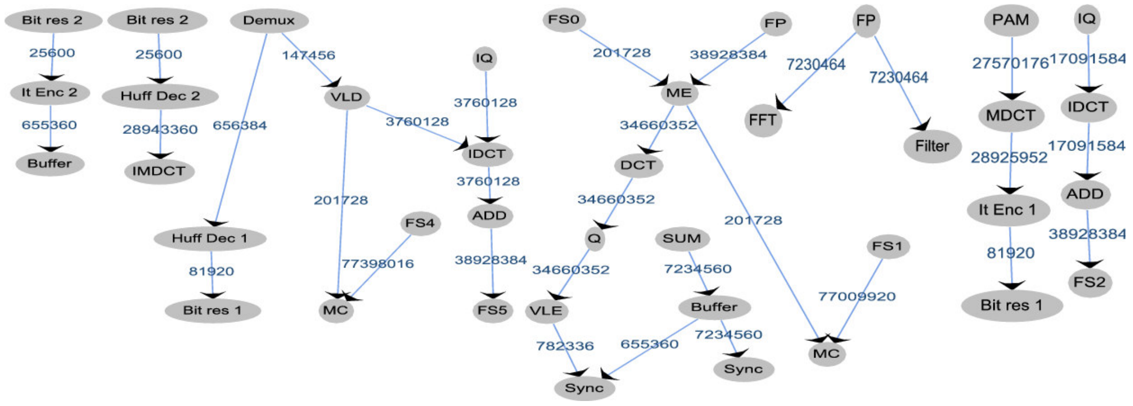

Fig. 64 MMS CTG 0 (period: 0.0009765625 seconds)

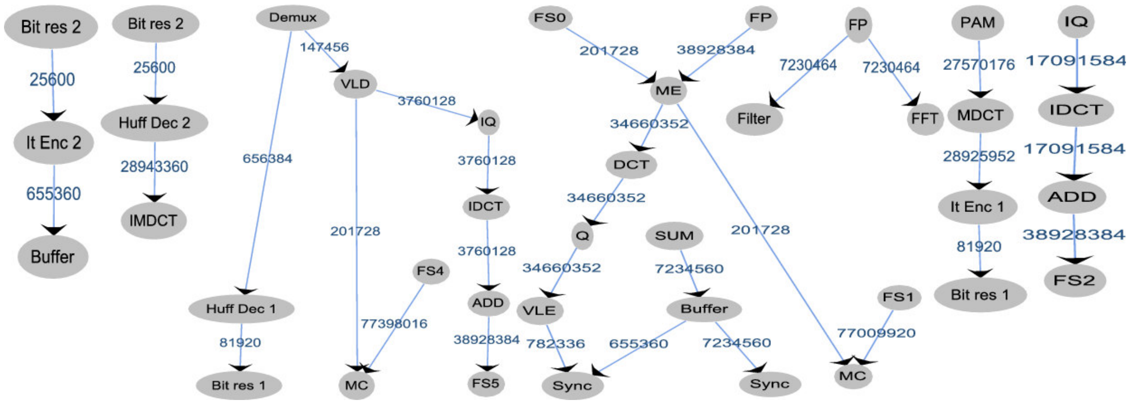

Fig. 65 MMS CTG 1 (period: 0.0009765625 seconds)

We present next some of the real applications introduced by Murali et al. with the SUNMAP tool [71] and the NMAP application mapping algorithm [61], in the field of Network-on-Chip research.

Based on their core graphs[12], we derived the Communication Task Graphs for the following applications: Picture-in-Picture (PIP), MPEG-4 decoder, Multi-Window Displayer (MWD) and Video Object Plane Decoder (VOPD).

5.2.2 Picture-in-Picture (PIP)

The core graph for the Picture-in-Picture application was obtained by Murali et al. from [99]. This application takes two video streams from an input memory (Inp mem) and superimposes them. The main video stream covers the entire screen of a TV, except a small window, where the secondary video stream is shown. Therefore, the secondary video stream is scaled horizontally and vertically (tasks hs and vs). Two advanced address generators, called jugglers, are used to write video data into the memory, at any position, with an arbitrary shape. They allow mixing the video streams. From the memory, the Picture-in-Picture video stream is output on the TV display.

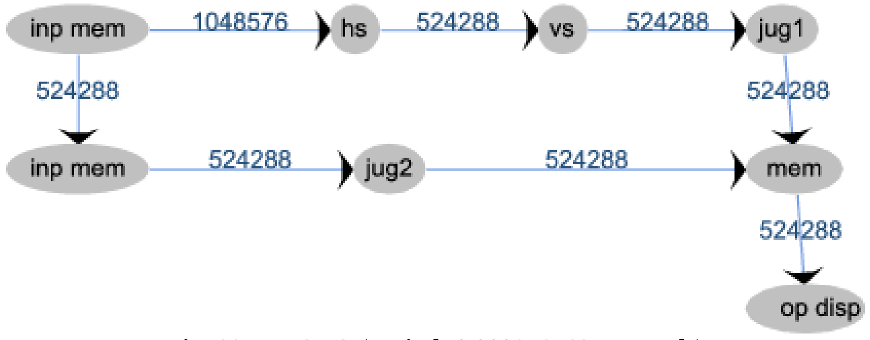

Fig. 66 PIP CTG (period: 0.0009765625 seconds)

5.2.3 MPEG-4 Decoder

The MPEG-4 decoder application was obtained from an architecture implementation of the MPEG-4 decoding system, including data transport bandwidth in MB, available in [100].

This system is made of: one Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM), two Static Random Access Memories (SRAMs), one Very Long Instruction Word (VLIW) media CPU, one Reduced Instruction Set Computing (RISC) processor, an audio Digital Signal Processor (DSP), a 3D graphics module (for rasterization), a Context Addressable Memory (CAM) used by the RISC CPU, a Binary Alpha Block (BAB), an image decompression module (for inverse discrete cosine transform and other operations) and a Motion Compensation and Estimation (MCE) module (with up-sampling and padding). A bit stream of input data is given to the system, which has two outputs: one for the decoded audio stream and another for the decoded video stream.

We observed in [100] that the RISC CPU communicates 500 MB/s with the memory and the inverse discrete cosine transform only 250 MB/s, with the same

[12] Core graph is a synonym for Application Characterization Graph (APCG)

memory. However, in the core graph used by Murali et al. the two communication volumes are reversed. Also, it is unclear to us why Murali et al. present a two-way communication between the audio and video outputs (au and vu) and the DDR SDRAM. In our opinion, data should flow only from the DDR SDRAM to the two outputs. In order to be able to reproduce the research results, we kept the graph structure like Murali et al. did.

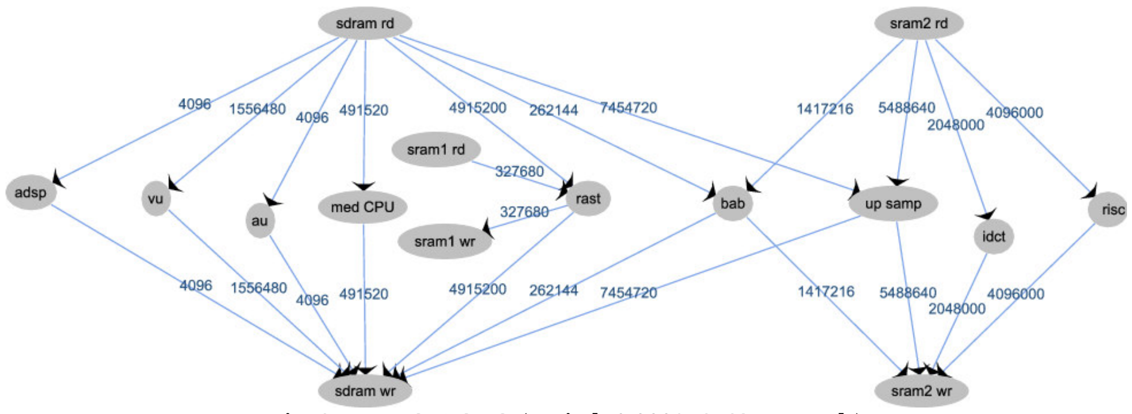

Fig. 67 MPEG-4 CTG (period: 0.0009765625 seconds)

5.2.4 Multi-Windowed Displayer (MWD)

The Multi-Windowed Displayer [99] is a more complex Picture-in-Picture application. The video background contains a zoomed-in part of the smaller video stream (presented as picture-in-picture). A small square, which may be moved and resized, allows selecting the part of the smaller video stream that will be zoomed-in. Additionally, an Internet browser is also shown on the display.

This application involves noise reduction (nr), horizontal and vertical scaling (hs, vs), mixing (jug1 and jug2), sharpness enhancement (se) and graphics blending. The noise reduction coprocessor accesses the memory to perform infinite impulse response filtering.

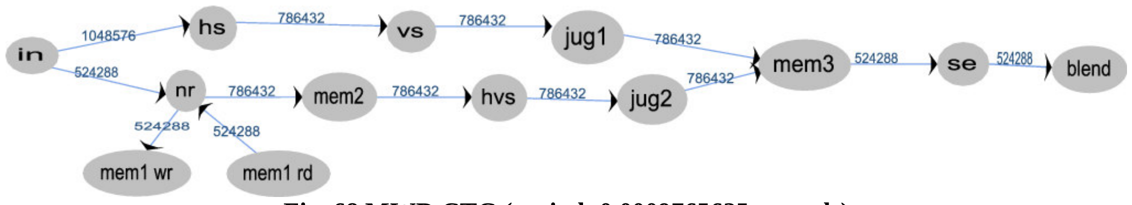

Fig. 68 MWD CTG (period: 0.0009765625 seconds)

5.2.5 Video Object Plane Decoder (VOPD)

An MPEG-4 video stream may be divided into hierarchical layers. The lowest layer is called the Video Object Plane (VOP) layer. It corresponds to a single video stream frame. The VOP is used to represent rectangular (plane) frames or an arbitrary-shaped plane.

A Video Object Plane Decoder (VOPD) is presented in [100]. A Context-based Arithmetic Decoder (CAD) parses the input stream and retrieves the context information from the shape decoder, through a local memory buffer. Then, it writes the output Binary Alpha Block (BAB) into a local memory. A shape decoder module reads the computed

BAB and the referenced BABs and calculates the context value. The newly calculated BAB is written into the same memory. The output stream obtained by variable length decoding is sent into a pipeline with inverse scan, AC/DC prediction (which uses some local memory), inverse quantization and inverse Discrete Cosine Transform. Another module provides motion compensated VOP reconstruction and padding. Additional memory is used to keep the reconstructed output and motion compensated prediction data.

A core graph for this VOPD application has been proposed by Murali et al. Based on it, we derived our VOPD CTG. We observed that compared to [100], where the stripe memory is accessed only by the AC/DC prediction module, this memory module is accessed in the core graph by both AC/DC prediction and inverse quantization modules. Our CTG respects the communication from the core graph of Murali et al. (so that we maintain compatibility with their research).

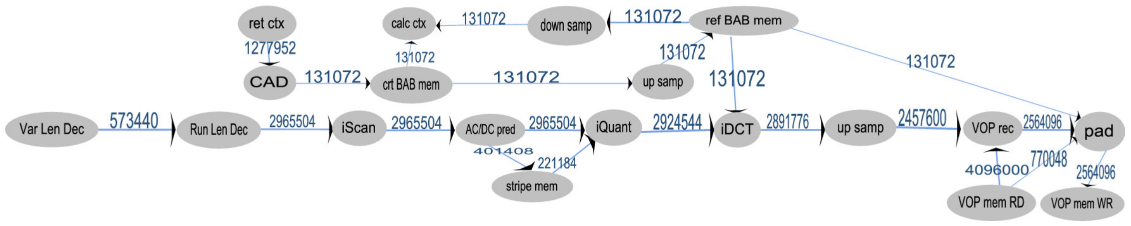

Fig. 69 VOPD CTG (period: 0.0009765625 seconds)

5.2.6 H.264 Decoder

Finally, we make our contribution to Network-on-Chip benchmarking, by proposing two new Communication Task Graphs. We obtained the two CTGs based on the research from [101], where the mapping of a H.264 decoder on a multiprocessor architecture is presented.

CTG 0 presents a H.264 decoding system that uses data partitioning: the video stream is equality divided onto more CPUs, each one of them running a H.264 decoder.

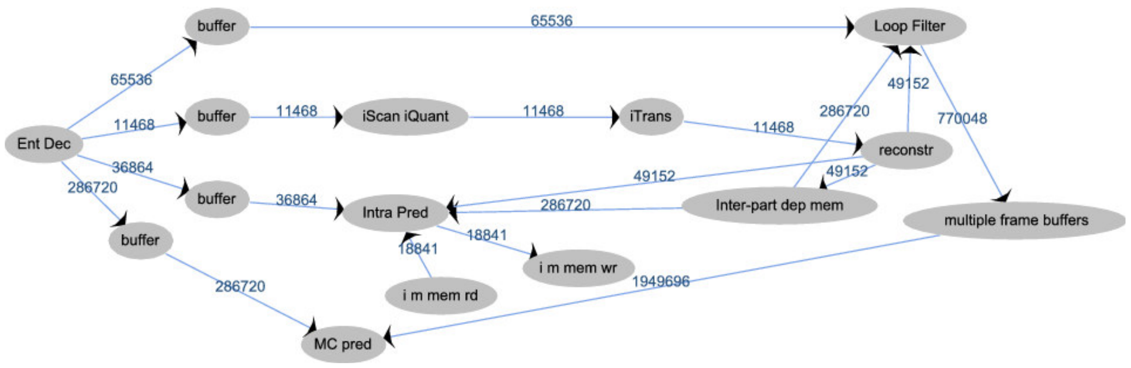

Fig. 70 H.264 CTG 0 (period: 0.0009765625 seconds)

CTG 1 presents a H.264 decoding system that uses functional partitioning: the tasks of the H.264 decoder are placed onto different CPUs.

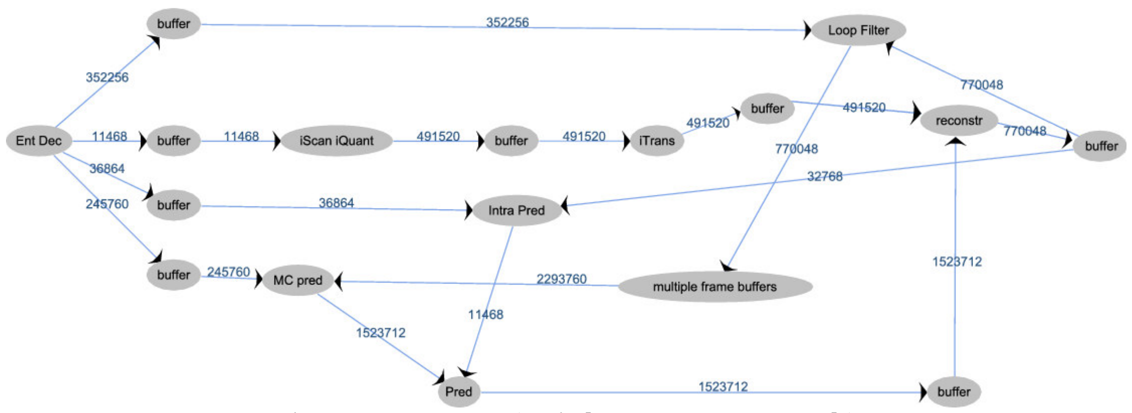

Fig. 71 H.264 CTG 1 (period: 0.0009765625 seconds)

With the functional partitioning approach, the messages between the decoder tasks are communicated. With data partitioning, data dependencies among data partitions are communicated. It is shown in [101] that, with data partitioning, a significant bandwidth reduction is obtained.

5.3 Application Characterization Graphs

For all of the above Communication Task Graphs, we have assigned the tasks to IP cores. By doing so, we obtained an Application Characterization Graph (APCG) [3], for each application. All E3S benchmarks plus Picture-in-Picture had each task assigned to a different core. For the rest of the benchmarks, at least two tasks were associated with the same IP core. We present next only the APCGs that have at least two tasks scheduled on the same processing element (marked by a dotted rectangle). Note that the communications between the tasks placed on the same IP core are neglected because they do not generate network traffic.

MMS APCG 0 has 16 IP cores and corresponds to MMS Communication Task Graph 0. Application Specific Integrated Circuits (ASICs) are used for executing tasks like: bit reservoir, iterative encoding, ME, demultiplexing and synchronization. Digital Signal Processors (DSPs) run Huffman decoding, discrete cosine transform, inverse discrete cosine transform, quantization, inverse quantization, VLD, FP, filtering, MDCT, IMDCT and SUM. Memories are used for buffering and FS (0 to 5) tasks. Finally, a general purpose processor executes MC and ADD.

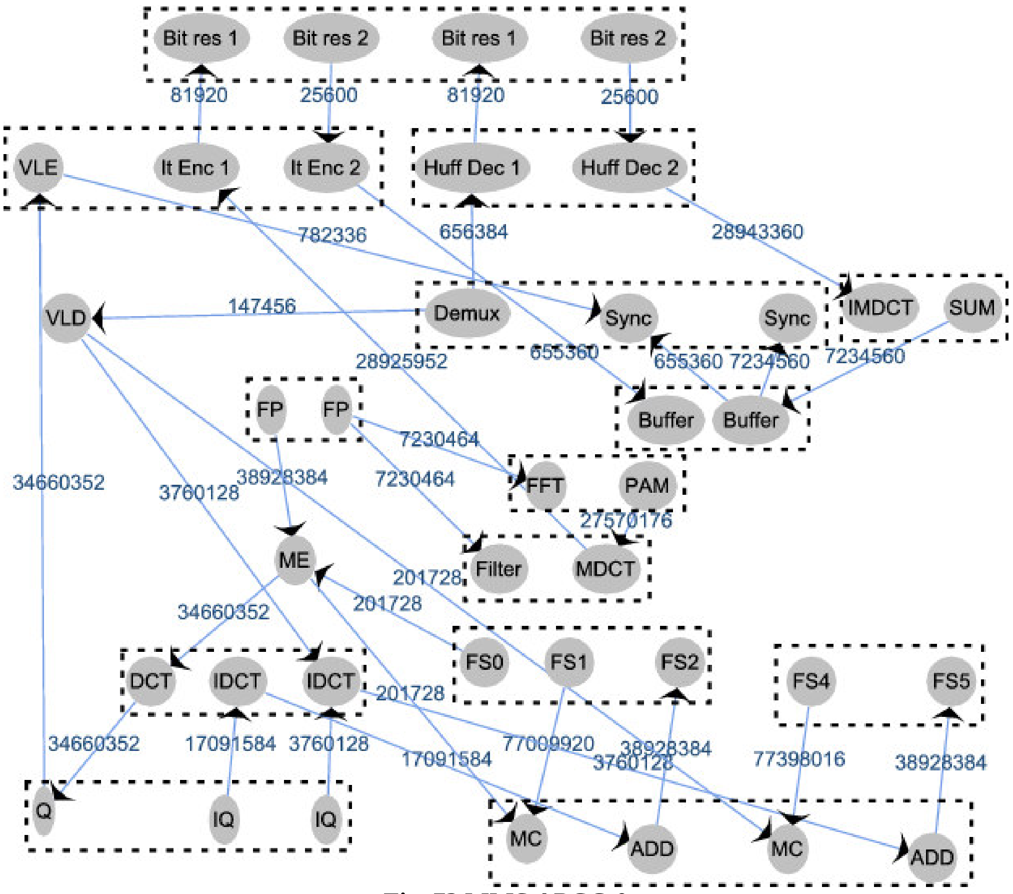

Fig. 72 MMS APCG 0

MMS APCG 1 has 25 cores that execute the tasks described by MMS CTG 1. The same kind of processors like for MMS APCG 0 are used.

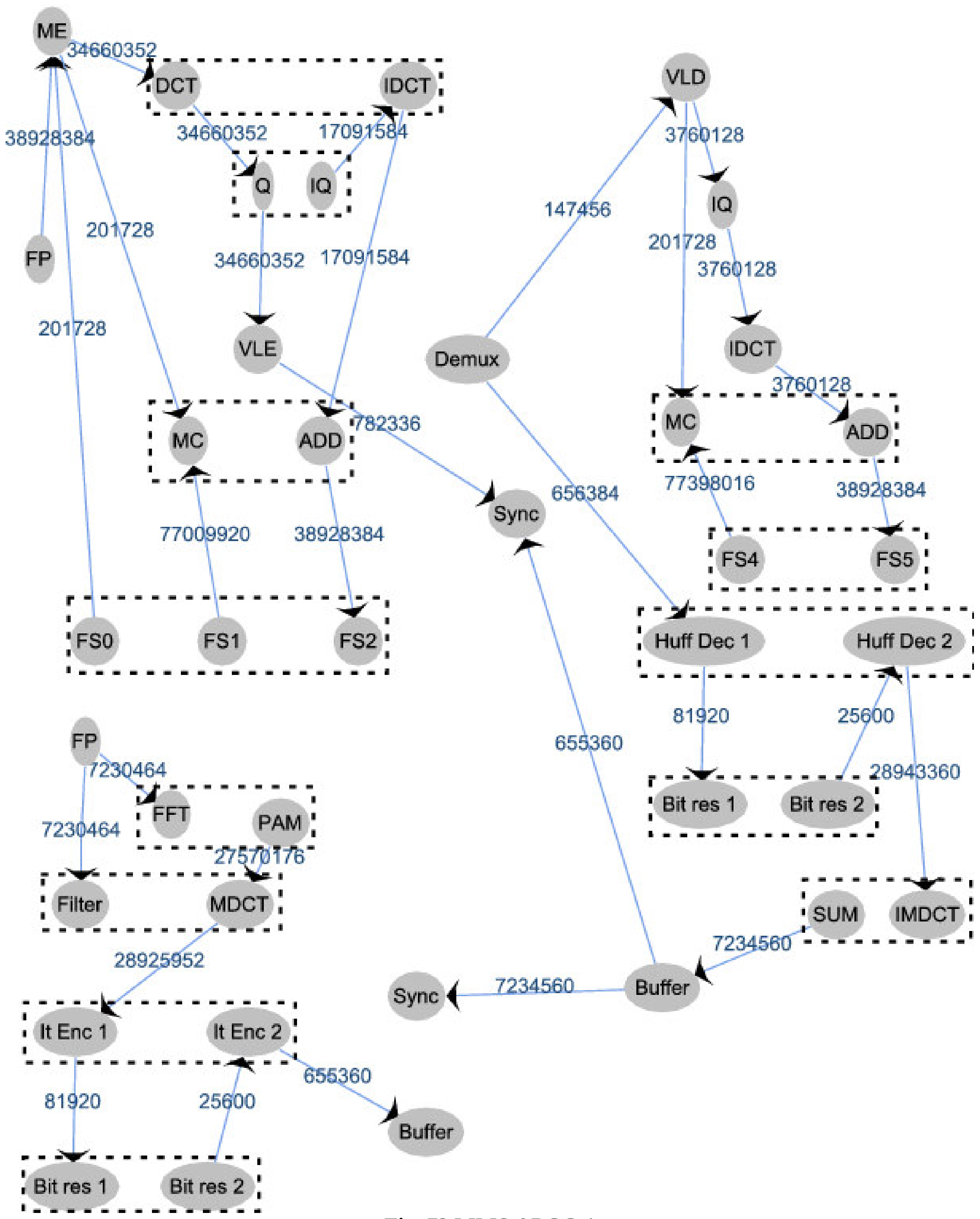

Fig. 73 MMS APCG 1

MPEG-4 APCG was obtained by grouping the tasks that deal with memory accesses. Thus, “sdram rd” task is grouped with “sdram wr”, “sram1 rd” with “sram1 wr” and “sram2 rd” with “sram2 wr”. MPEG-4 APCG has 12 IP cores.

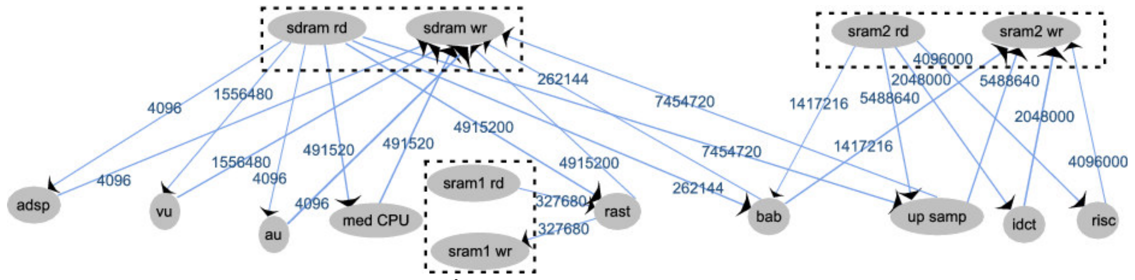

Fig. 74 MPEG-4 APCG

The Multi-Window Displayer APCG was obtained like MPEG-4 APCG, i.e. by placing on the same core the tasks which do memory reads and writes. MWD APCG has 12 IP cores.

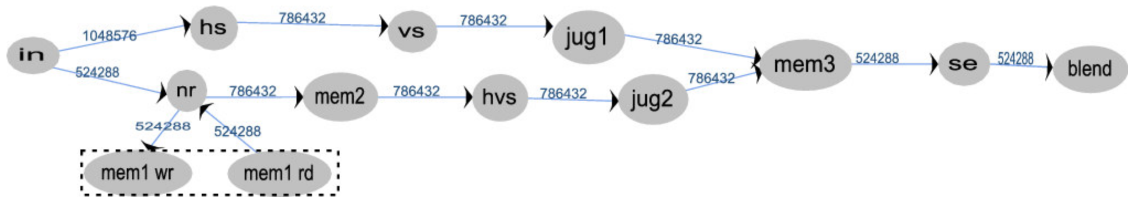

Fig. 75 MWD APCG

From VOPD CTG we obtained two APCGs: APCG 0 has 16 cores and APCG 1 has 12. The Video Object Plane memory tasks are placed on the same IP core in both APCGs. In comparison with APCG 0, APCG 1 groups more tasks from the upper part of the graph.

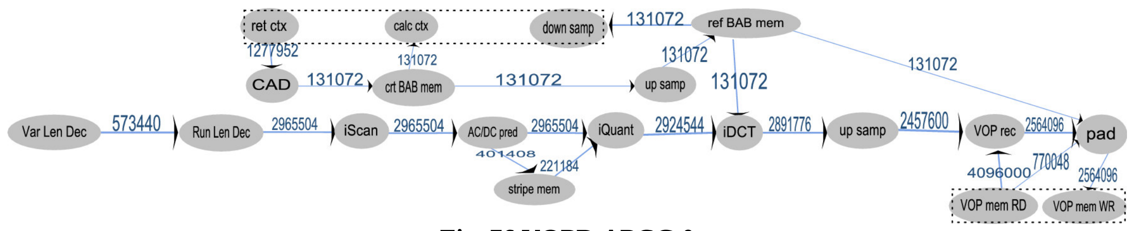

Fig. 76 VOPD APCG 0

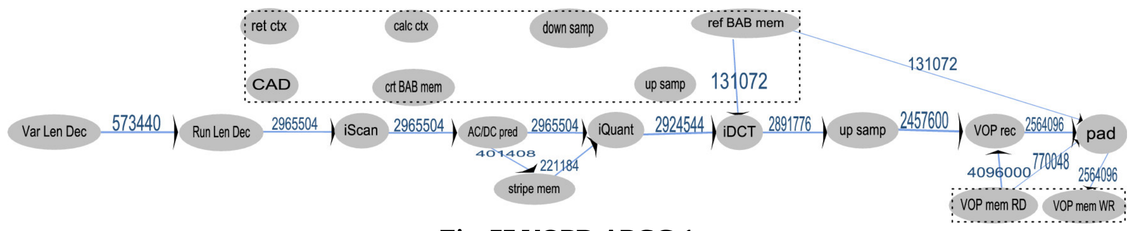

Fig. 77 VOPD APCG 1

From H.264 CTG 0 we created APCG 0, with 14 cores, by grouping the two tasks for accessing the intra mode memory.

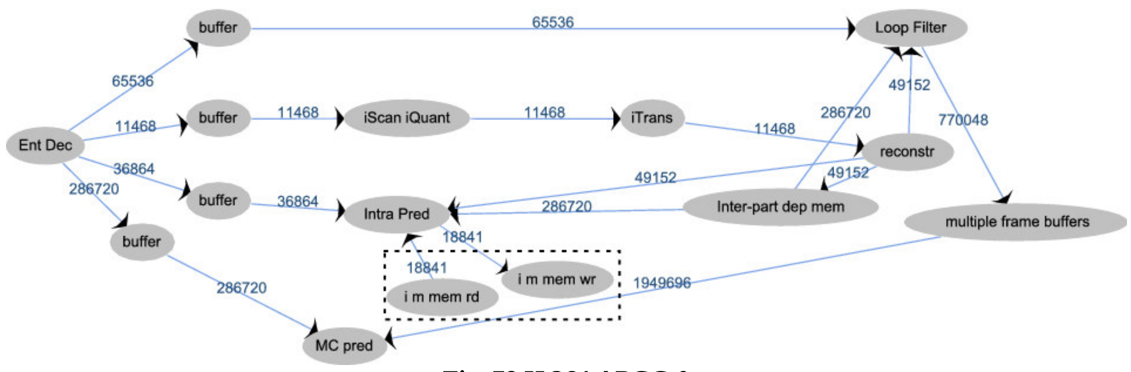

Fig. 78 H.264 APCG 0

We also obtained an APCG with 16 cores, from H.264 CTG 1.

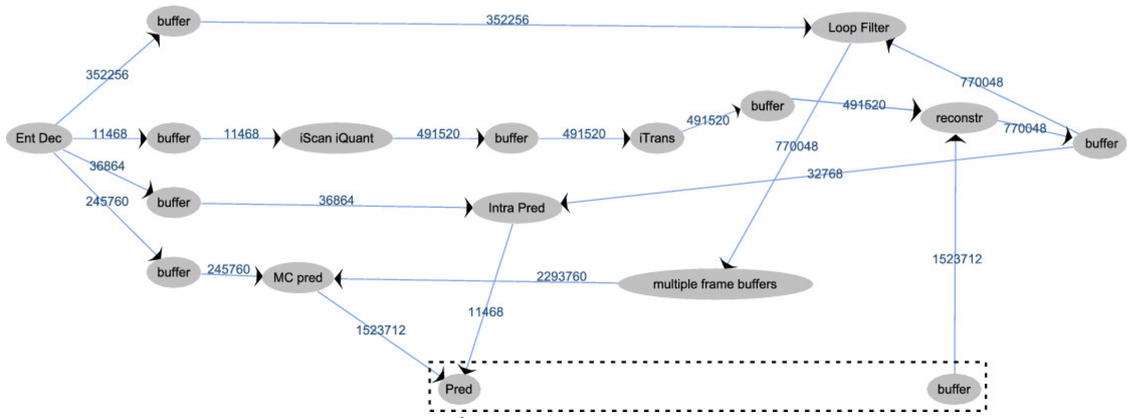

Fig. 79 H.264 APCG 1

5.4 Summary

We presented in this chapter the CTGs and APCGs used to model real applications for Network-on-Chip benchmarking. We integrated into UniMap the E3S benchmark suite and some of the currently most used benchmarks in the NoC research field. Any researcher using UniMap can now benefit from these benchmarks.

We also contributed with two versions of a H.264 video decoder application.

Our systematic approach should help other researchers that work on Network-on-Chip benchmarking.

“To climb steep hills requires a slow pace at first.”

William Shakespeare

6 Optimized Simulated Annealing for Network-on-Chip Application Mapping. A Domain-Knowledge Approach

We introduced in Chapter 3 Simulated Annealing (SA) (see Section 3.3.1) [102], one of the first heuristic algorithms used to address the Network-on-Chip application mapping problem.

The advantages of Simulated Annealing are given by its ease of implementation, its applicability to many combinatorial optimization problems and the ability to give reasonably good solutions [60]. This algorithm is used in domains like Biology, Telecommunications, Geology, Electronics, Medicine and Engineering [103], [104].

However, the parameters of the algorithm must be carefully chosen, since SA can easily run for a very long time until it gives a suitable solution. Because Simulated Annealing is a very general algorithm, several choices must be made in order to implement it for a particular problem. These choices are categorized in [105] as generic and problem-specific. A generic choice refers to SA’s input parameters: initial and final temperature, cooling schedule, number of iterations done per temperature and stopping condition. A problem-specific choice is related to the neighborhood structure, which specifies how the search space is explored.

This chapter presents a domain-knowledge approach to Network-on-Chip application mapping problem. We describe an Optimized Simulated Annealing (OSA) [106] algorithm that we designed for the topological placement of cores onto NoC nodes. OSA uses an application- and network-based exploration of the search space. Using knowledge about communication demands, the IP cores are clustered implicitly and dynamically. We compare OSA with the above mentioned simulated annealing technique and with a branch and bound algorithm, too. We focus on algorithm speed, memory consumption and solution quality.

6.1 Related Work

Simulated Annealing was one of the first algorithms used to address the NoC application mapping problem [40]. The objective was to minimize (under bandwidth constraints) the communication energy for a 2D square NoC mesh with XY routing and wormhole switching.

Hu and Marculescu compare Simulated Annealing with a Branch and Bound (BB) approach and they find out that SA is capable of finding mappings better than the ones found with Branch and Bound. However, SA has the disadvantage of speed: the simulation results from [40] show that BB is tens of times faster the SA.

Both SA and BB algorithms were further developed in [52], where the mapping problem is treated in conjunction with the routing problem. XY routing is no longer considered as the NoCs routing algorithm. The mapping algorithms are capable of generating the routing tables for each network node, in a deadlock- and livelock-free manner, using Odd-Even [31] and West-First [30] routing protocols. The same disadvantage of SA emerges from this paper too. Simulated Annealing is tens of times slower that Branch and Bound. It is also mentioned that SA required more than two hours to reach a solution for a 7x7 2D mesh. For a 10x10 NoC, the time required by SA became prohibitive: the algorithm did not finish in 40 hours.

In order to speed up Simulated Annealing for mapping cores onto NoC tiles, a Cluster-based Simulated Annealing (CSA) is proposed in [107]. CSA tries to exploit the application's communication locality so that it can identify clusters of IP cores. For each core cluster, a NoC region is allocated. At high temperatures, the annealing process occurs normally: any cores may be moved. However, when the temperature decreases enough, the annealing process is limited to the cores from the same cluster. The idea behind Cluster-based Simulated Annealing may be considered a practical implementation of the clustering proposed by Kirkpatrick et al., adapted for the NoC application mapping problem.

The clustering phase determines the initial mapping for the annealing process. First, the NoC nodes are grouped based on the number of output links and the overall distance between them.

The communication core clustering is done in a similar way, by considering the number of output communications and communication volumes for each core. However, the number of core clusters is set to the number of determined NoC node clusters and also, the number of cores from each cluster must match the number of nodes from the NoC node cluster. Therefore, the clustering is driven first by the NoC topology and secondly by the application.

In the Core-to-Node mapping phase core clusters are assigned to node clusters: core clusters are ordered by their communication demands and the clusters that communicate the most are placed onto the NoC nodes that have the most output links. The clustering is static. The purpose of this clustering technique is to reduce the number of moves that lead to worse solutions, by restricting the cores moves inside their clusters. At high temperatures, inter-cluster moves are allowed. At low temperatures, only intra-cluster moves are permitted.

After the clustering is done, the annealing process begins. The number of moves made at each temperature level is not specified. Also, the stopping condition is vaguely defined as a time threshold or a sufficient number of feasible solutions found. The initial and final temperatures are not mentioned but, the number of temperature levels is set to D, where D is the network's diameter. Each time, a core is randomly selected for swapping. For that core, it is identified the maximum network distance, d, between it and the cores from its cluster. If we consider the annealing process is currently at step D_c ($D_c \leq D$) then, if $D_c \leq d$, only moves inside the cluster are allowed. Otherwise, the core is allowed to be swapped with any core from the network. This means that inter-cluster moves become less frequently, as the annealing temperature decreases.

The authors report a 30% CSA speedup over SA by mapping the Video Object Plane Decoder (VOPD) [61] on 2D meshes with sizes 3x3, 4x4 and 8x8. For the 3x3 mesh, a partial VOPD is used and for the 8x8 mesh, the VOPD application is used four times. Hence, CSA was not extensively evaluated but, the results show that clustering is able to reduce the search space significantly and CSA is still able to find the same best solution found by SA. Clustering represents a problem-specific choice for Simulated Annealing. It considerably improves SA's speed because it uses more knowledge about the NoC application mapping problem than the general Simulated Annealing.

We have presented so far how Simulated Annealing was used until now to solve the NoC application mapping problem. Hu and Marculescu showed that SA can find good mappings but, only when the problem size is not very big (up to a 10x10 2D mesh). They show that SA is too slow for NoC application mapping and propose a Branch and Bound approach that is significantly faster because the search space is pruned. The strength of BB resides on how effective is the pruning heuristic, since it tradeoffs speed with quality of solution. However, a Branch and Bound technique can start consuming a lot of memory. Lu et al. showed that Simulated Annealing can be made faster if the Network-on-Chip is clustered and then the communicating IP cores are also clustered. Core clustering is determined by NoC node clustering. This has the advantage that the application is clustered with respect to the NoC topology characteristics. The disadvantage is that NoC node clustering is deterministic and it does not account for the application to be mapped at all. We believe a more flexible clustering, which may better adapt to the application, may yield better results. Also, we observe that CSA's clustering is explicit, fixed before the annealing process starts.

We propose next an Optimized Simulated Annealing algorithm for NoC application mapping. Compared to CSA, our algorithm performs an implicit clustering, during the annealing process and it is also significantly faster than CSA (and obviously than SA). We do not cluster the NoC nodes because we do not want to restrict the core clustering process. Only the application cores are clustered. We influence the IP core moves during the annealing process, so that the communicating cores implicitly clump together.

6.2 The Algorithm

OSA was created by continuing the work of Hu and Marculescu. Their Simulated Annealing and Branch and Bound algorithms are available through the NoCmap project [108]. We have ported their two algorithms, written in C++, into UniMap (written in Java). OSA also uses some of the best practices for Simulated Annealing applied for assigning tasks to processors [109]. We justify our approach by the fact that NoC application mapping problem is closely related to the NoC scheduling problem [43].

We present next the Optimized Simulated Annealing pseudocode, which is derived from the general Simulated Annealing from [109].

```
Require: $M_i \neq 0$
Ensure: $T_0 \geq 1$
  $M \leftarrow M_i$
  $C \leftarrow BitEnergyCost(M_i)$
  $M_{best} = M$
  $C_{best} = C$
  $T_f = 0.001$
  $R = 0$
  for $i = 0$ to $\infty$ do
    if $i \ \% \ L = 0$ then
      $R = 0$
    end if
    $T = T_0 \cdot 0.9^{\lfloor \frac{i}{L} \rfloor}$
    $M_{new} = PDFbasedSwapping(M, T)$
    $C_{new} = BitEnergyCost(M_{new})$
    $\Delta C = C_{new} - C$
    if $\Delta C < 0$ or $NormInvExpAccept(\Delta C, T)$ then
      if $C_{new} < C_{best}$ then
        $M_{best} = M_{new}$
        $C_{best} = C_{new}$
        $R = 0$
      else
        $R = R + 1$
      end if
      $M = M_{new}$
      $C = C_{new}$
    else
      $R = R + 1$
    end if
    if $T \leq T_f$ and $R = L$ then
      break
    end if
  end for
  return $M_{best}$
```

Fig. 80 Optimized Simulated Annealing

In the following sections, we show how OSA implements the most important parts of a typical simulated annealing algorithm.

6.2.1 Mapping Cost

OSA is energy-aware. It evaluates the mappings using the analytical energy model presented in Section 3.1.3.2.

6.2.2 Annealing Schedule

There are a lot of annealing schedules in literature: straight, geometric, reciprocal, logarithmic, fractional, Koch etc. [103], [104], [105]. We have chosen for OSA the geometric annealing schedule because this is the most used and recommended one [105] and because the general SA implementations use it as well.

The geometric annealing temperature schedule defines the temperature at iteration *i* as:

$$T = T_0 \cdot q^{\left\lfloor \frac{i}{L} \right\rfloor} \quad , q \in (0,1)$$

T_0 is the initial temperature and q is the geometric progression ratio. We set q = 0.9 for OSA because this value is also used in [102], [108] and usually q is set between 0.9 and 0.98 [109]. L is the number of iterations per temperature level.

OSA uses an initial temperature set to 1 by default but, this is considered a parameter of the algorithm. The final temperature is fixed to 0.001. It is correlated with the acceptance function (see Section 6.2.4). Hu and Marculescu's SA starts from a temperature of 100 and the final temperature is not bounded (their algorithm uses another stop criteria).

6.2.3 Number of Iterations per Temperature Level

The general Simulated Annealing sets $L = 100(NxN)^2$, where NxN represents the size of 2D mesh NoC. For example, for a typical 4x4 2D mesh, SA tries L = 25600 mappings at each temperature level. These mappings are randomly generated, from the current mapping, by interchanging two cores, or by placing a core into an empty NoC node. If we consider that SA runs for 100 temperature levels[13], this leads to 2560000 mappings generated. An Intel XEON processor from our HPC system [70] evaluates a mapping in ~0.04 ms. Thus, for this example, SA would run for about 102 seconds. The algorithm's speed increases with the square of the NoC topology size. For the same example but with a 10x10 mesh, SA would require more than one hour running time. However, the general SA algorithm is not bounded by the number of temperature levels, and we have observed that it easily runs for more than 150 levels of temperature for a 4x4 2D mesh. We argue SA's number of iterations per temperature levels is very high and it has a deep impact on its speed. Also, we observe that this number is only NoC aware. It is by no means application aware. For example, mapping 15 or 16 cores on a 4x4 2D mesh uses the same L.

OSA uses the following relation to compute the number of iterations for each temperature level:

$$L_{OSA} = {}_nC_2 - {}_{n-c}C_2 = \frac{n(n-1)}{2} - \frac{(n-c-1)(n-c)}{2} = \frac{c(2n-c-1)}{2} \quad ,c,n \in N^*, n \geq c$$, where:

n is the number of NoC nodes and c is the number of cores to be mapped.

This number of iterations per temperature level represents how many distinct core swappings are possible for a given NoC with n nodes and c cores placed onto these nodes, so that at most two cores change positions compared to the initial mapping.

Therefore, the first core may be moved from its node onto other (n – 1) nodes. The second core may be moved from its node onto other (n – 2) nodes (it is swapping with the first core has already been counted). The third core can do (n - 3) swappings and so on until the last core, which can be put onto (n - c) nodes.

The sum of all these single step possible swappings is therefore $\sum_{k=n-c}^{n-1} k = \sum_{k=1}^{n-1} k - \sum_{k=1}^{n-c-1} k = \frac{n(n-1)}{2} - \frac{(n-c-1)(n-c)}{2} = L_{OSA}$.

[13] This is more than possible because the 100th element of a geometric progression with ratio 0.9 and initial element 100 is $100 \cdot 0.9^{100-1} \cong 0.00295$.

It may easily be observed that $\max\{L_{OSA}\} = \frac{n(n-1)}{2} \overset{not.}{=} L_{OSA\max}$. This happens when the number of cores to be mapped is equal with the number of NoC nodes.
SA has L = 100(NxN)^2. Because we noted the number of NoC nodes with *n*, we can write that L_{SA} = 100n^2. It is obvious that $L_{OSA} < L_{OSA_{\max}} < L_{SA}$. Also, in terms of algorithm speed, we note that OSA speedup over SA is $S = 1 - \frac{L_{OSA}}{L_{SA}} \Rightarrow \lim_{n \to \infty} S = \lim_{n \to \infty}\left(1 - \frac{n(n-1)}{200n^2}\right) = 99.5\%$.
We considered for L_{OSA} the worst case, given by $n = c$. This speedup is in perfect concordance with our further experimental results.
It can easily be observed that L_{OSA} counts all mappings that are obtained by making a single modification (core swapping) compared to the given mapping (otherwise, the total possible mappings are of course ${}_nP_c > L_{OSA}$). All the mappings that derive from a given mapping in this way are what we call the mapping's immediate neighborhood. We can make an analogy with Markov chain. L_{OSA} can be associated with the number of possible single step transitions from a Markov chain, which describes the mapping state space exploration performed by OSA. Later on, we will show how OSA assigns probabilities for each single step transition, using the concept of Probability Distribution Function (PDF).

Because *n* is NoC topology characteristic and *c* is application characteristic, OSA's number of iterations per temperature level is both NoC and application aware.

Returning to the example with the 4x4 2D mesh and a Simulated Annealing algorithm that runs for 100 temperature levels, we compute L_{OSA} for mapping 16 cores to be 120. This means a run time of 0.48 seconds. Compared with how much time the general SA needs (102 seconds), we get a speedup of ~ 99.53%. For the 10x10 2D mesh, OSA's *L* is 4950. This means a runtime of 19.8 seconds. The speedup in this case is of ~ 99.5%. Obviously, if the number of cores to be mapped is less than the number of nodes from the network, L_{OSA} becomes smaller and the speedup higher.

Some criticism to this approach can be that although huge speedup can be obtained with L_{OSA}, OSA might find worse solutions because it does less exploration of the search space. We argue that the general SA can easily repeat a lot of moves without finding better solutions so that the search can be driven to qualitatively better solutions. Additionally, OSA considers the initial temperature a parameter, which can be increased so that the algorithm does more exploration. We will sustain our assessments through simulations.

6.2.4 Acceptance Function

Both general SA and CSA algorithms use the Metropolis acceptance probability function: $P(\Delta C) = e^{-\frac{\Delta C}{KT}}$. Bolzmann's constant (*K*) is set to the current mapping cost in the implementation of Hu and Marculescu (and it is unspecified in CSA's case). OSA however uses the normalized inverse exponential acceptance function because it is shown in [109] that it yields better results when it is used for task scheduling. This function is defined as $P(\Delta C) = \frac{1}{1 + e^{-\frac{\Delta C}{C_0 T}}}$. The practical difference between the two functions can be seen when we consider $\Delta C = 0$. The exponential form always accepts a new mapping that is equally good compared to the current mapping, whereas the (normalized) inverse exponential form accepts such a new mapping with a 50% probability.

OSA's acceptance function performs cost normalization by dividing the cost variance (ΔC) with the initial mapping cost (C_0). This allows the temperature T to be independent of the cost function: $T \in (0,1]$. Note that OSA sets the initial temperature to 1 and the final temperature to 0.001. However, OSA allows the initial temperature to be higher than one. With respect to the normalized inverse exponential function, this would mean the initial cost is artificially increased.

Using the normalized inverse exponential acceptance function, the temperature T can be written as: $T = -\frac{\Delta C}{C_0 \ln\left(\frac{1}{p} - 1\right)}$. For the final temperature, $\Delta C = 0.001 C_0 \ln\left(\frac{1}{p} - 1\right)$. Therefore, the cost of a new mapping that is worse than the current mapping is accepted with a certain probability until it varies with only $0.5\% C_0$ from the cost of the current mapping. Since OSA's cost function represents energy consumption, we can conclude that OSA considers the system frozen when the energy consumption varies with less than 0.5 percent.

6.2.5 PDF-based Swapping

The move function determines how Simulated Annealing explores the search space. In [40], [52] the move is a random swap: a core is selected randomly and this core is then swapped with another randomly selected core (an empty node can also be used). Note that a random swap move has two steps:

1. select a core to be swapped;
2. select another core (or empty node) for the exchange.

The same random swap is used in [107] but, in this case the moves are restricted at cluster level when the annealing temperature has decreased sufficiently. While the simple random swap is by no means problem aware, the CSA random swap is indirectly aware of the problem because it restricts a random move to the cluster of the core to be moved.

While both approaches select the core to be swapped randomly, OSA does not use a uniformly random probability when determining the core to be moved. Instead, it adapts the variable grain single move (based on probability densities and used for task mapping [109]) into a variable grain swapping move, which uses two Probability Density Functions (PDFs). OSA builds a Probability Density Function (PDF) for each core, based on the amount of data it communicates. This leads to better chances for selecting a core that communicates more data than a core which communicates less data. As the annealing temperature decreases, the probabilities uniformly equalize. Therefore, at low temperatures, all cores have a quite equal chance to get selected for swapping. Through this approach, OSA uses domain-knowledge (dynamic characteristics) to explore the search space. The following function is used:

$$P[SelectedCore = i] = \frac{1}{c} + \frac{T}{T_0}\left(\frac{coreToComm_i}{totalToComm} - \frac{1}{c}\right), \text{ where:}$$

- c is the number of cores to be mapped;

- *T and* T_0 are the current and initial temperatures;
- *totalToComm* is the total amount of data communicated by the all cores;
- *coreToComm*$_i$ is amount of data communicated by core *i*.

The second core used for swapping is selected by accounting for the communication volumes between the core to be swapped and the rest of the cores. Another Probability Density Function is built for each core. It is similar with the one above but, it does not consider only the data communicated by the core but, also the data received by the core. Also, this second PDF is not temperature dependent. Each core gets such a PDF associated before the annealing starts. This PDF is defined as $P[c_i \leftrightarrow c_j] = \frac{comm_{ij}}{totalComm}$, where:

- *comm*$_{ij}$ is the communication volume between core *i* and *j* (this value is positive if core *i* sends data to core *j*, or core *j* sends data to core *i*; otherwise, it is zero);
- *totalComm* is the communication volume of the entire application.

According to the PDF described above, the second core is selected for swapping. Then, OSA searches, in a uniformly random way, for a direct neighbor of the second selected core. This one will be swapped with the first selected core. This approach tries to make communicating cores to attract each other, to cluster themselves, in a natural manner. OSA's move function performs an implicit clustering of the communicating cores, using a stochastic approach.

Compared to CSA, our algorithm clusters the cores dynamically, during the annealing phase. OSA does not work with predetermined clusters, and it also does not cluster the NoC nodes. Network-on-Chip node clustering is not needed because OSA looks in the NoC node's neighborhood.

We call this kind of move a *PDF-based swapping move*. At every temperature level, OSA performs exactly L_{OSA} PDF-based swappings.

6.2.6 Stopping Condition

OSA uses the stopping function recommended in [109]: coupled temperature and rejection threshold. The annealing process stops when the temperature goes below the final temperature (0.001) and the last L_{OSA} tried mappings did not lead to a solution better than the best solution found so far. Therefore, OSA runs for a determined number of annealing temperatures, which can be exceeded while better solutions are found. We note OSA has a form of elitism because, during the entire running process, it stores the current best solution. This is another difference from a general Simulated Annealing technique.

Because OSA uses a geometric annealing schedule, we can write that $T_f = T_0 q^{n-1}$, where:

- T_0 and T_f are the initial and final temperatures;
- q = 0.9 is the geometric progression ratio;
- n is the number of temperature levels.

The number of temperature levels can thus be expressed as $n = \left\lfloor \log_q \frac{T_f}{T_0} \right\rfloor + 1$.

If we consider that T_0 = 1, T_f = 0.001 and q = 0.9, then OSA runs for approximately n = 65 temperature levels. Note that *n* is independent of the problem size. It can be increased if better solutions are still found, after the final temperature level has been reached, or if the initial temperature is raised. Obviously, the initial temperature must grow exponentially so that *n* increases linearly.

Because OSA's stopping condition determines a number of annealing levels independent of the problem size, the runtime of our algorithm is quasi-constant when the algorithm is run more than once in exactly the same conditions. This property does not apply to Hu and Marculescu's Simulated Annealing.

6.2.7 Summary

OSA starts from an initial mapping (M_i) that is by default randomly generated (obviously, the initial mapping can also be specified). Another input parameter can be the initial temperature, T_0, set to 1 by default. The mapping's cost is obtained using the bit energy model from [40]. We use a standard geometric annealing schedule, with L_{OSA} annealing iterations per temperature level. This number corresponds to how many mappings may be obtained from the current mapping, by moving one core. The move function is a swapping based on Probability Density Functions. We use the normalized inverse exponential acceptance function because this is the one recommended by [109]. OSA stops when the final temperature ($T_f = 0.001$) is reached and the number consecutive rejected moves, R, reaches L_{OSA}. This corresponds to the coupled temperature and rejection threshold stopping condition proposed in [109]. While in [109] R counts how many moves were rejected since the last accepted move, in OSA we use R to count how many moves were rejected, per temperature level, since the last best mapping was found. This means that while OSA requires no best mapping to be found during an entire temperature level, the general Simulated Annealing from [109] needs to wait until the number of unaccepted moves, counted from the last one accepted, reaches L. OSA's stopping condition is therefore more coupled to T_f than to R. This makes OSA's number of iterations to be independent of the NoC topology and its size. Since we consider that the energy variations are small enough when the final temperature is reached, we believe our way of computing R is more suitable for a Simulated Annealing applied to NoC application mapping.

Currently, OSA works only with 2D mesh topologies but, it can be adapted to work with other NoC topologies, too. Like Hu and Marculescu's SA, OSA is also capable to generate the routing functions, in a deadlock- and livelock-free manner, and to check if the obtained mapping meets the bandwidth constraints.

Compared to the general SA, OSA determines how many iterations to make per temperature level by considering the mappings' neighborhood size. Using Probability Density Functions, OSA performs an implicit and dynamic core clustering (CSA's clustering is explicit and static).

6.3 Simulation Methodology

This section presents the simulation methodology used to evaluate our Optimized Simulated Annealing. OSA is compared to the NoC application mapping algorithms mentioned above: Simulated Annealing and Branch and Bound [40]. We also compared our results with the ones reported for Cluster Simulated Annealing [107].

Our methodology is mainly determined by: the benchmarks used for mapping, the Network-on-Chip architecture and how and in what conditions we run our simulations.

We evaluate OSA using all the benchmarks presented in Chapter 5.

We have considered the most common Network-on-Chip architecture: a 2D mesh with regular tiles, using wormhole switching and XY routing. The NoC topology size is a simulation parameter. The NoC link bandwidth was set sufficiently high so that bandwidth constraints are always met. The energy required to transfer a bit of data was taken from NoCmap. The values were determined for 0.35 µm technology [52]: a NoC router needs 0.284 picoJoule for processing one bit of data, the link needs 0.449 picoJoule to transmit it. A buffer read operation requires 1.056 picoJoule per bit and a buffer write takes 2.831 picoJoule.

Three NoC application mapping algorithms were used to map the core graphs (benchmarks) onto 2D meshes: Simulated Annealing, Optimized Simulated Annealing and Branch and Bound.

For each benchmark, the size of the network was set as low as possible to include the number of cores from the benchmark. The following table presents the NoC 2D mesh size used for mapping each benchmark.

Benchmark	# cores	# NoC nodes	NoC size
auto-indust	24	25	5x5
consumer	12	12	4x3
networking	13	16	4x4
office-automation	5	6	3x2
telecom	30	30	6x5
PIP	8	9	3x3
MPEG4	12	12	4x3
MWD	12	12	4x3
H.264 (CTG 0)	14	16	4x4
H.264 (CTG-1)	16	16	4x4
VOPD (CTG 0)	16	16	4x4
VOPD (CTG 1)	12	12	4x3
MMS (CTG 0)	16	16	4x4
MMS (CTG 1)	25	25	5x5

In order to increase the simulations' accuracy we have mapped each application 1000 times, with each algorithm. For each simulation, the initial mapping was randomly chosen. To make the comparisons fair, we have set the seed of the random number generator so that all algorithms start from the same point in the search space, every simulation. Thus, simulation 1 works with seed 1, simulation 2 with seed 2, …, simulation 1000 with seed 1000. We note OSA uses a linear congruential random number generator [87], the same used by SA. A true random number generator could prove more useful but, we consider this aspect beyond the scope of this work.

For each mapping, we recorded the solution (i.e. the mapping), its cost (in pJoule), the runtime of the algorithm (user CPU time) and the average heap memory consumption. We did not use any application bandwidth constraints. Our experiments are fully reproducible because the algorithms and the simulation methodology are part of UniMap [69], an open-source project.

6.4 Experimental Results

In this section, we evaluate our Optimized Simulated Annealing by comparing it with Simulated Annealing and Branch and Bound. The evaluation is three folded. We account for execution runtime, memory consumption and solution quality. We show next only the most representative results. More detailed results are available in [110], [111].

We begin with a runtime comparison between OSA and SA and respectively OSA and BB. The speedups represent an average of the 1000 runtime speedups obtained for each benchmark.

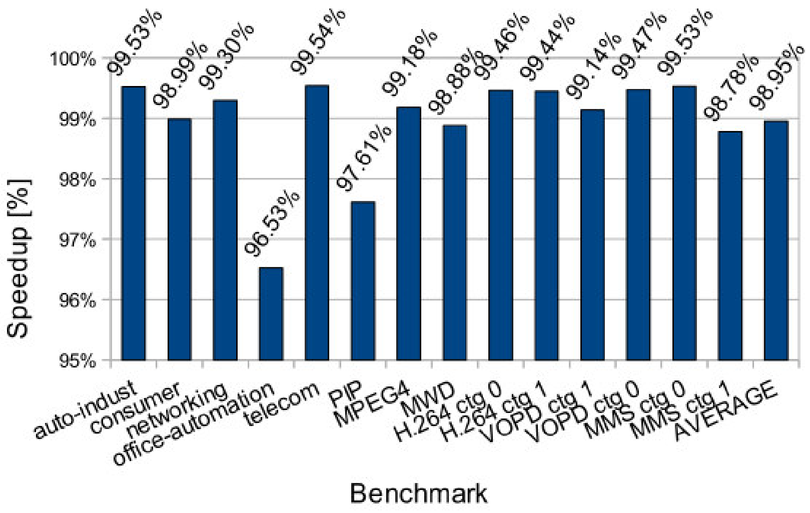

Fig. 81 OSA speedup over SA

The chart above clearly shows OSA is much faster than Hu and Marculescu's Simulated Annealing. We have obtained a 98.95% speedup on average. This is in perfect concordance with our theoretical speedup expectations from section 6.2.3. The "lowest" speedups are on *office-automation* and PIP, the benchmarks with the smallest number of IP cores. We justify this significant speed gain mainly by the way OSA computes the number of iterations per temperature level. This number takes into consideration the NoC size, the number of cores to be mapped, and it is much lower than the number used by Hu and Marculescu.

The following chart shows how fast OSA is compared to Branch and Bound.

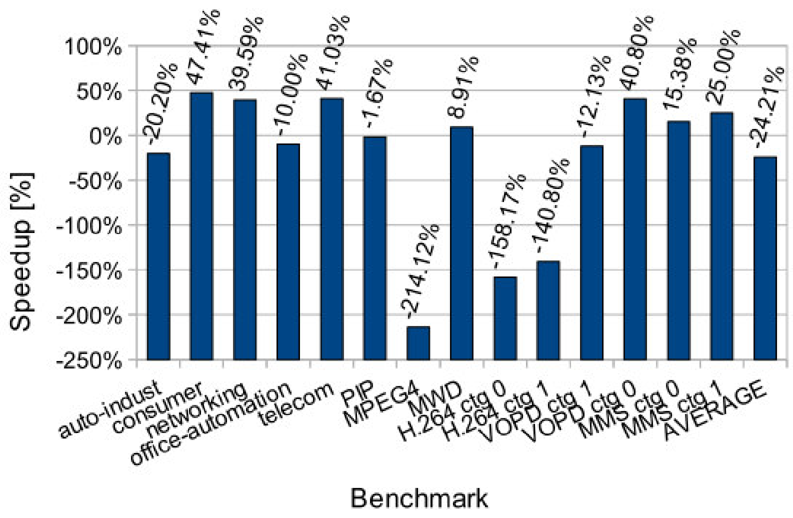

Fig. 82 OSA speedup over BB

It can be seen that OSA is slower than BB by ~ 24%, on average. However, for half of the benchmarks, OSA is faster. Compared to Branch and Bound, our algorithm obtained poor runtimes on MPEG4 (more than twice slower), H.264 (~ 1.5 times slower in both cases) and slower but similar runtimes for PIP, *office-automation*, VOPD (CTG 1) and *auto-indust*. We also observe OSA was faster on the biggest benchmarks: 25% speedup for MMS (with 25 cores) and ~ 41% speedup for *telecom* (30 cores).

Next we show how OSA's memory consumption is, compared to the memory consumed by Simulated Annealing and Branch and Bound.

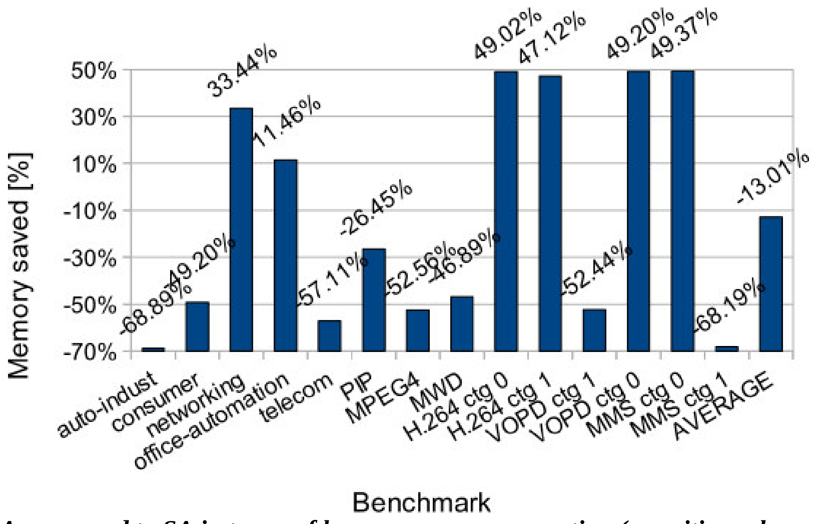

Fig. 83 OSA compared to SA in terms of heap memory consumption (a positive value means OSA consumes less memory)

Simulated Annealing consumes less memory than OSA when mapping the benchmarks with more than 16 cores. OSA manages to beat SA on several benchmarks with 16 cores but, on average, Simulated Annealing consumes with ~13% less memory than our Optimized Simulated Annealing.

However, compared to Branch and Bound, OSA takes a little bit less memory on average. This is shown in the next chart.

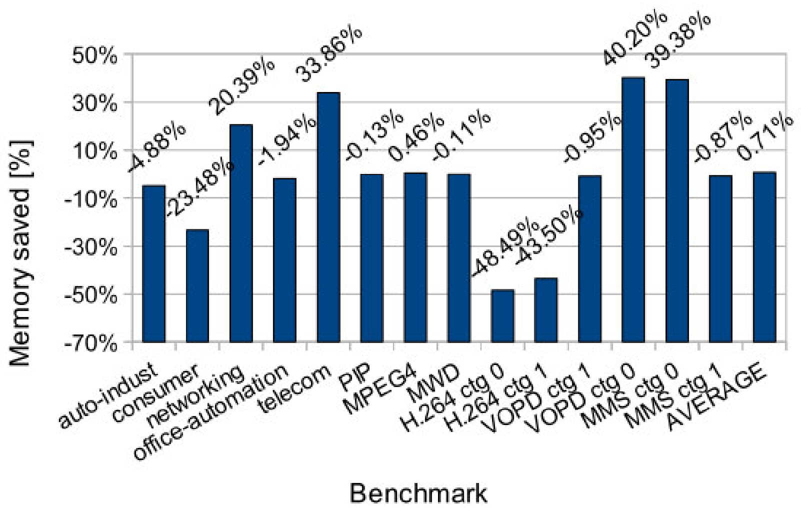

Fig. 84 OSA compared to BB in terms of heap memory consumption (a positive value means OSA consumes less memory)

Actually, this chart points out the tendency of Branch and Bound to grow its memory requirements as the problem size gets higher: OSA consumes with more than 33% less heap memory than BB, on *telecom*.

Now we present the quality of the solutions found by the three algorithms. We are interested in solutions with the smallest cost possible because the cost function we used estimates the energy consumed by the Network-on-Chip.

The following chart compares the mappings found by SA and OSA. For each benchmark, we evaluate the 1000 mappings returned by the two algorithms and count how many times one algorithm retuned mappings better (marked with "<" in the chart's legend) than the other one. Cases when both algorithms returned mappings with exactly the same cost are marked distinctively.

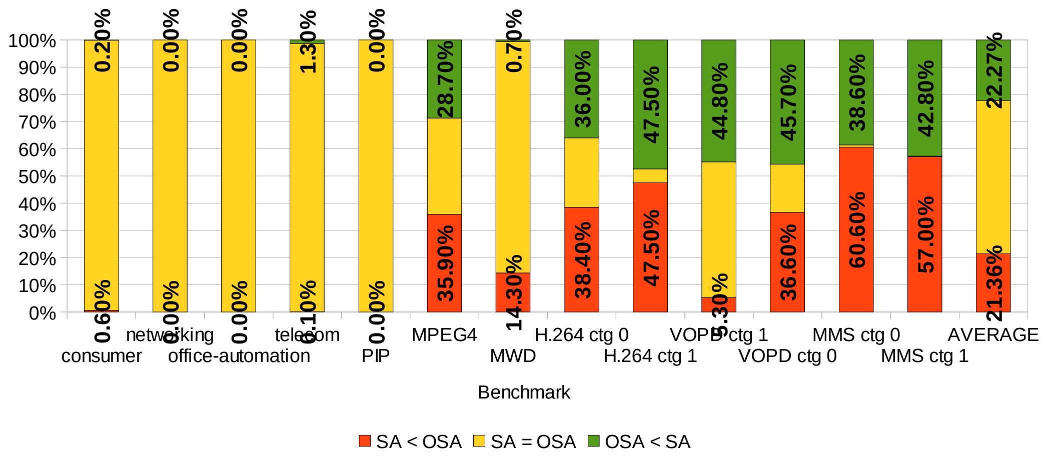

Fig. 85 OSA mapping costs, compared to SA mapping costs

We notice that both algorithms find the same "best solution", after all 1000 runs, for benchmarks: *networking*, *office-automation* and PIP. For the last two of these three benchmarks, we confirm the solution is optimal because we applied an exhaustive search. Overall, OSA finds worse solutions than SA for 6 of the 14 benchmarks used in our simulations: MPEG-4, MWD, H.264 (CTG 0), MMS (CTG 0), MMS (CTG 1) and *consumer*.
We have also found out that SA and OSA always find the same best solution. However, Branch and Bound fails to obtain a mapping that consumes at most like the best mapping found by SA and OSA in two cases: for MMS (CTG 1), the energy lost with BB's mapping would be 0.1 % and for *auto-indust*, the energy loss is ~6%.

We measured the difference between the worst and best mappings found for each benchmark by SA and OSA. With our Optimized Simulated Annealing, the variation between the worst and best mappings was not higher than 8%. However, with SA we obtained the highest variation to be 70% for MMS (CTG 1). For the rest of the benchmarks SA did not varied with more than 6%. Excluding MMS (CTG 1), the SA average variation was 1.51% and the OSA average variation was 2.56%. If we also consider MMS (CTG 1), SA had an average variation of 5.85% while OSA's value was less than half (2.53%). We conclude that the variations between the best and worst mappings are comparable for SA and OSA.

Fig. 86 shows how many times the best solution, given by all three algorithms, was found by each one of them.

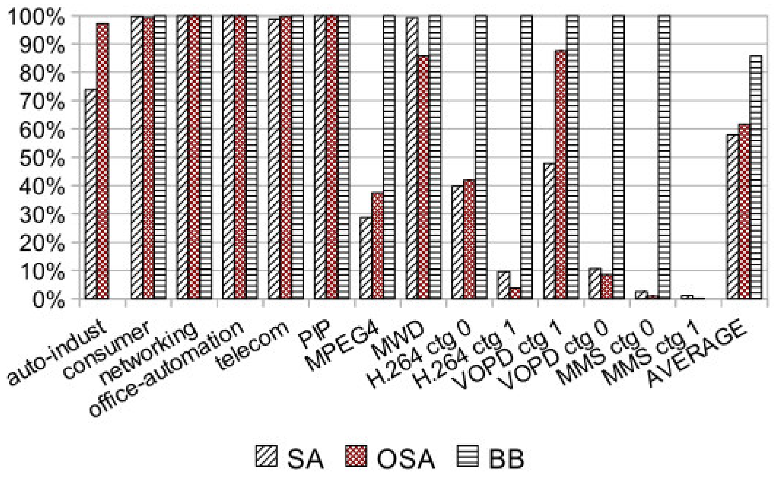

Fig. 86 Best solution percentage

This chart shows that OSA finds the best solution more often than SA for several benchmarks: *auto-indust*, *telecom*, MPEG4, H.264 (CTG 0), VOPD (CTG 1). BB outmatches OSA for the MMS benchmarks, VOPD (CTG 0), H.264 (CTG 1), MWD and *consumer*. Another observation is related to BB: it finds the best solution with probability 1 for all benchmarks, except *auto-indust* and MMS (CTG 1).

We also averaged the quality of the 1000 mappings per benchmark. Branch and Bound is the algorithm that, on average, gives the mapping with the smallest energy consumption. It fails just on *auto-indust* benchmark, where OSA provides the best average mapping cost. Optimized Simulated Annealing achieves for MMS (CTG 1) a far better average cost compared to Simulated Annealing: more than 34% energy gain is obtained with OSA. For the rest of the benchmarks, the differences between OSA and SA are less than one percent. Compared to BB, OSA provides solutions that are worse with no more than 2.5% on each benchmark, except *auto-indust*, where OSA is better with more than 6% than Branch and Bound.

Using 1000 simulations per benchmark, we have previously shown that the percentage of better solutions was lower for OSA than for SA on six benchmarks: MPEG-4, MWD, H.264 (CTG-0), MMS (both CTGs) and *consumer*. We present here our attempt of increasing OSA's quality of solution by increasing the initial temperature. We applied this technique on the benchmarks mentioned above, with the purpose of getting OSA's percentage of better solutions over SA's percentage. Increasing the initial temperature allows OSA evaluate more mappings. Also, the higher the temperature, the bigger is the probability to accept "bad" moves during the annealing process.

Through this technique the quality of solution for our Optimized Simulated Annealing got better, matching SA's quality of solution i.e., OSA's percentage of better mappings overcame the corresponding SA percentage. Still, we had one exception: we were unable to obtain the desired outcome for MMS (CTG 1). We disregard this undesired result due to the fact that in this case, on average, SA consumes with more than 34% more energy than OSA.

Note that we have increased OSA's initial temperature exponentially because, due to the OSA's geometric annealing schedule, an exponential increase of temperature leads to a linear increase of the number of temperature levels.

The following table presents OSA's speedup over SA, in terms of runtime, and the initial temperature required by OSA to beat SA.

Benchmark	**Speedup (%)**	**Initial temperature**
MPEG4	97.51	1E+10
MWD	96.76	1E+10
H.264 (CTG 0)	99.18	1E+02
MMS (CTG 0)	97.41	1E+17
MMS (CTG 1)	61.40	1E+107
consumer	98.91	2E+00

If we ignore MMS (CTG 1), we see that the speedup remained high even with the increase of initial temperature.

In order to illustrate how important OSA's clustering technique is, we present next a comparison between OSA with and without clustering. The single thing that distinguishes OSA without clustering from OSA (with clustering) is that, in the first case, the simple random core swapping is used, without any restrictions.

The following chart shows how frequently the best solution is found.

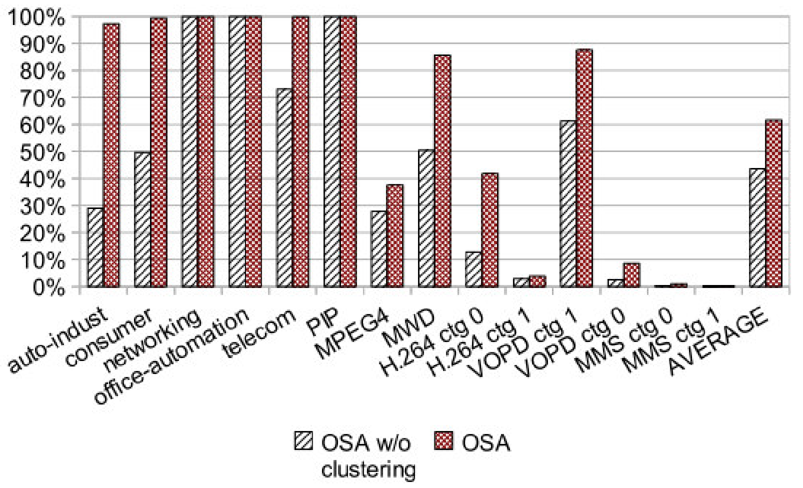

Fig. 87 The influence of OSA's clustering on best solution percentage

For all benchmarks, OSA with clustering finds the best solution more frequently than OSA without clustering. More than this, we observe significant differences for the benchmarks mapped onto the 4x4, 5x5 and 6x5 2D mesh NoCs. It is important to mention that the two OSA variants find the same best solution for all benchmarks, except MMS (CTG 1). In this case, the best solution found by OSA w/o clustering is with 0.02% worse. On average, the best solution percentage for OSA with clustering is 18% higher than for OSA without clustering. Therefore, our implicit and dynamic clustering technique helps OSA to find the best mappings more often.

The next chart shows how much energy is consumed on average by OSA without clustering (compared with OSA using clustering).

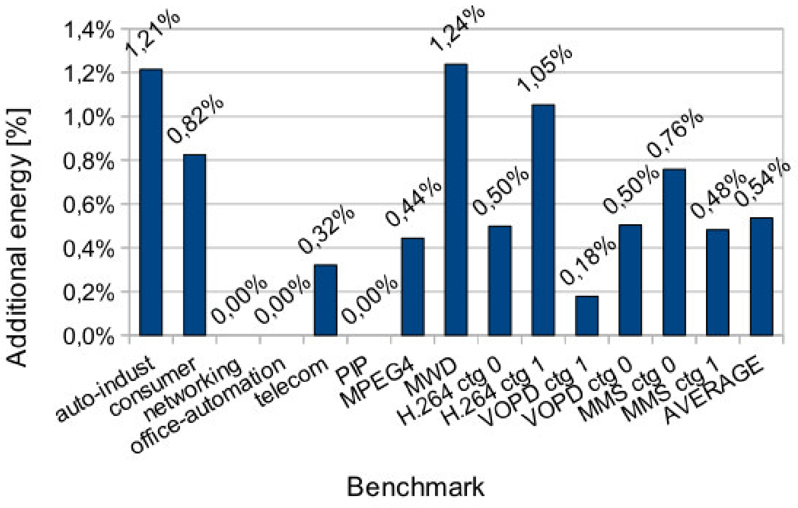

Fig. 88 Average energy consumed by the mappings obtained with OSA without clustering

It may be noticed that for each benchmark OSA without clustering found mappings that consume additional energy. The clustering technique leads to lower energy consumption with more than 1% in some cases. OSA with clustering always gives better average results than OSA without clustering.

Finally, we present the simulation results on bigger 2D meshes. We used four instances[14] of the VOPD benchmark with 16 cores and obtained a benchmark with 64 cores. Using SA, OSA and BB, we mapped it on an 8x8 2D mesh. SA was run ten times and OSA and BB run 100 times.

We obtained an average running time of ~ 12.65 hours (per simulation) for SA. OSA ran for approximately 155 seconds, while BB required just ~ 114 seconds. Averaging the results from the 100 runs, OSA was ~36% slower than BB. Still, OSA runtime is significantly lower than CSA's runtime: 4750 seconds [107].

OSA consumes with approximately 39% less memory than Branch and Bound. During the 100 simulations, OSA's peek memory consumption was 37.3 MB, while BB required a maximum memory of 85 MB.

The best mapping was found by Simulated Annealing. However, OSA's best mapping is only ~ 0.7% worse. Branch and Bound finds a best mapping that consumes around 64% more than the best mapping found by SA. Averaging the 100 mappings done by OSA and comparing them with the ones obtained with BB, we have observed that Branch and Bound obtains on average a mapping cost ~70% worse.

We have aggregated all the E3S benchmarks used in our previous simulations and obtained 84 cores that we mapped onto a 10x9 2D mesh. Again, SA run 10 times, while OSA and BB run 100 times.

SA required a very big time to run one simulation: approximately 70 hours. OSA ran for approximately 526 seconds, while Branch and Bound needed only 380 seconds. Averaging the results from the 100 runs, Optimized Simulated Annealing was ~48% slower than BB.

OSA consumed approximately the same of memory Branch and Bound required. During the 100 simulations, OSA's peek memory consumption was 62 MB, while BB required a maximum memory of 71 MB. We believe Branch and Bound manages to keep the memory consumption not growing exponentially by pruning most of the search space (we observed BB, in several simulations, to prune 85% to 93% of the explored search space).

Averaging the 100 mappings done by OSA and comparing them with the ones obtained with BB, we have observed that Branch and Bound obtains on average a mapping cost ~76% worse. Simulated Annealing found the best solution but, it is better than OSA's best solution by only 0.09%.

Using the H.264 (CTG 1), MMS (CTG 0), MMS (CTG 1), MPEG4, MWD and VOPD (CTG 0) benchmarks, we have obtained 97 cores that we mapped onto a 10x10 NoC. Because of the huge running time SA needed for mapping the previous application, we simulated these application with 97 cores only with OSA and BB (both were run ten times).

Optimized Simulated Annealing run on average approximately 15.9 minutes per simulation. Branch and Bound needed only two thirds of this time: ~15.44 minutes for each mapping simulation (OSA is only 3% slower than BB).

Branch and Bound consumed around 40 MB of memory and Optimized Simulated Annealing required approximately 45 MB.

[14] Like in [107], because applications with a high number of cores are lacking and because we preferred using real applications instead of randomly generating core graphs (like in [40], [52])

Once more, OSA found every time mappings better than the ones found by Branch and Bound. Averaging the 100 mappings done by OSA and comparing them with the ones obtained with BB, we have observed that Branch and Bound obtains on average a mapping cost ~76% worse.

By combining all non E3S benchmarks (PIP, H.264, MPEG4, VOPD, MWD, MMS), we get a benchmark with 131 cores, which we mapped onto a 12x11 Network-on-Chip. OSA and BB mapped this benchmark ten times.

OSA required, on average, approximately 51 minutes mapping this application. Branch and Bound was ~15% faster: it needed only around 44 minutes, on average.

In this case, OSA consumed less memory, 36 MB, while BB memory requirements were 14% higher.

Optimized Simulated Annealing found each time a mapping that consumes significantly less memory. On average, OSA's solutions need 79.4% less memory than BB's solutions.

Finally, we combined all of our benchmarks an obtained an application with 215 cores. We used OSA and BB to map it (ten times) onto a 15x15 NoC.

Optimized Simulated Annealing run for 8.4 hours, on average. OSA consumed on average 265 MB of memory, for each mapping.

Branch and Bound run on average 3.77 hours for each mapping. This is more than half OSA's runtime. Memory consumption was also significantly lower: only 158 MB. However, we obtained no solution from BB, after all ten mapping. All mapping attempts will Branch and Bound failed. No suitable solution was found because, each time, the algorithm pruned more than 98.7% of the search space. This severe pruning did not allow BB to finish mapping the application. This leaves us to believe that Branch and Bound's memory consumption does not grow exponentially but, the quality of solution is heavily affected, up to the point where the algorithm does not give any solution.

6.5 Summary

We have presented in this chapter an Optimized Simulated Annealing (OSA) for Network-on-Chip application mapping. Like Hu and Marculescu's Simulated Annealing, OSA is energy and performance aware. OSA uses more application knowledge that helps it at better exploring the search space. Like Clustered-based Simulated Annealing, OSA also performs clustering but, implicitly and dynamically, not explicitly and statically.

OSA proved to be much faster than SA. We have obtained an average of 98.95% runtime speedup while the quality of the mapping solution is not lost: OSA managed to find the same best solution found by SA. On the 64 cores benchmark, we found OSA is 99.97% faster than CSA.

We showed OSA is feasible for NoC meshes with size higher than 10x10. OSA is also comparable to Branch and Bound in terms of memory consumption and speed. OSA was able to map 97 cores on a 10x10 2D mesh in a time slower by only 3% than the time required by BB. However, Branch and Bound fails to find better solutions on *auto-indust* and MMS (CTG 1) benchmarks. More than this, the mapping solution given by BB is 70% worse than the one found by OSA, when mapping cores onto an 8x8 2D mesh, 76% worse on 10x9 and 10x10 2D meshes, and 79% worse on a 12x11 2D mesh. We also found that Branch and Bound is unable to map an application with 215 cores, onto a 15x15 NoC, because more than 98% of the search space is pruned.

"You have your way. I have my way.
As for the right way, the correct way, and the only way, it does not exist."

Friedrich Nietzsche

7 Designing Domain-Knowledge Evolutionary Algorithms for Network-on-Chip Application Mapping

Evolutionary Computing (EC) [112] is a part of Artificial Intelligence (AI) inspired from the evolution process encountered in Biology. The heuristic algorithms from this field of research address NP-hard optimization problems by means of natural selection and evolution mechanisms. The search space is filled with candidate solutions, called individuals.

A typical Evolutionary Algorithm (EA) starts with a population of individuals (usually randomly generated). An objective (fitness) function is used to evaluate each individual. Then, based on the individuals' fitness, a selection mechanism decides which individuals participate in the next phase: reproduction. Crossover and mutation are the typical reproduction operators that take parent individuals and produce offspring individuals. Through recombination, a new population is formed. It contains individuals from the previous population and offspring. The population size is typically constant. Thus, while some individuals from the previous population are kept, others are discarded and replaced with offspring. Each population is called a generation (of individuals). The algorithm keeps on generating populations of candidate solutions until it is considered that, from the last generation, there are no more significant improvements in the individuals' fitness. Evolutionary Algorithms are used in many research fields to address single-objective and multi-objective optimization problems, based on the concept of Pareto efficiency [113].

In this chapter[15], we use UniMap (see Chapter 4) to evaluate and optimize two evolutionary algorithms: an Elitist Genetic Algorithm (EGA) and an Elitist Evolutionary Strategy (EES). After approaching our problem with an Optimized Simulated Annealing technique, we decided to switch to evolutionary algorithms due to their intrinsic parallelism. Evolutionary techniques perform searches starting (in parallel) from multiple points in the search space. Our evaluated algorithms optimize the Network-on-Chip communication energy. We consider multiple crossover and mutation operators, specific for permutation problems, like NoC application mapping is. Using problem specific knowledge, we propose such context-aware operators. We show such operators improve the evolutionary algorithms' performance. We try to find out which crossover and mutation leads to the best solutions. We also research whether crossover or mutation helps more the evolutionary algorithms. These algorithms are compared with our Optimized Simulated Annealing (OSA) technique (see Chapter 6). Finally we approach our problem in a multi-objective way: besides minimizing NoC communication energy, we also try to obtain a mapping that is thermally balanced.

[15] The work presented in this chapter was submitted (on July 21st, 2011) to the Journal of Systems Architecture (JSA - http://ees.elsevier.com/jsa). Since July 25th, 2011, it is under review with manuscript number JSA-D-11-00103 [114].

7.1 Related Work

In our opinion, one of the most representative works, using evolutionary algorithms for Network-on-Chip application mapping, is the one developed by Ascia et al. [64], [115], [72]. The authors use a genetic algorithm to determine the mapping with the best application execution time and power consumption. The mappings are evaluated using a NoC simulator.

In [64], the SPEA2 [65] multi-objective genetic algorithm is used, with single-point-crossover and swap mutation. Both genetic operators remap hot spot cores (i.e. the cores with the highest communication volume) randomly. We question here the use of single-point-crossover because this kind of operator may lead to duplicated genes in the offspring, which are not allowed for permutation problems.

The crossover and mutation operators are redefined in [115]. The mutation operator chooses a core at random and places it onto one of the neighboring nodes of the core with which it communicates the most. The crossover operator is not aware of the NoC application mapping problem. It acts as multiple swap mutations. No useful information about the two crossed over mappings is exchanged.

The same mutation operator is kept in [72], while the crossover operator was changed: from the two parents, the one with the better mapping is chosen. Then, its hot spot core is swapped with a randomly chosen core. This crossover operator also does not exchange any information between the two parent mappings. It actually acts as a swap mutation, too.

Ascia et al. conclude that the definition of suitable genetic operators has a strong impact on the performance of the genetic algorithm. They admit future research is required in this area. We also observe that, essentially, Ascia et al. define a crossover operator that behaves like a mutation operator.

We evaluate several crossover operators for permutation problems: Position Based (PB) and Partially Mapped (PMX). We improve PB by proposing NoC Position Based crossover (NPB), a context-aware genetic operator. We also introduce a novel crossover called Mapping Similarity (MS). It identifies the similarities between the two parent mappings and propagates them to their children. In contrast with the crossover operators of Ascia et al., MS exchanges information between the parents.

As mutation operator, we integrate our developed Optimized Simulated Annealing (OSA). We show that, as compared with swap mutation, this hybrid (evolutionary – simulated annealing) approach provides better results.

Besides a genetic algorithm, we also work with an evolutionary strategy. EES is available in jMetal (a multi-objective metaheuristics library, which is integrated in UniMap – see Chapter 4). We also use the OSA algorithm as a baseline. This allows us to compare simulated annealing with evolutionary algorithms.

We evaluate our two algorithms with every crossover and mutation operators, in terms of solution quality and convergence speed. The goal is to identify the best crossover and mutation operators. We also want to find out how each genetic operator influences our algorithms.

Finally, we show how our genetic operators behave doing a multi-objective optimization, with NSGA-II [116] and SPEA2 algorithms. Besides minimizing NoC communication energy, we also aim to place the IP cores so that the NoC is thermally balanced. For the thermal balance goal we use the analytical model presented in [117].

Thermal balance means minimizing the hotspot temperatures. This is achieved by leaving a bigger distance between the IP cores that consume more power. Even if these IP cores do not communicate a lot (or even not at all), placing a bigger distance between them changes the entire mapping. A thermally balanced NoC could have the communicating IP cores not placed as close as possible. Therefore, our two objectives are contradictory. This makes the multi-objective optimization process more difficult than the one of Ascia et al., where the two objectives (application runtime and power) are not contradictory (both application execution time and NoC power consumption will decrease if the communicating cores are closer to one another).

7.2 Energy- and Performance-Aware Genetic Algorithm

We developed in UniMap an Energy- and performance-aware Genetic Algorithm (EGA). EGA is based on the Generational Genetic Algorithm (GGA) [118]. As compared to GGA, EGA implements an elitist mechanism.

EGA is developed for MxN 2D mesh NoCs but, it may be extended to work with other topologies as well. The algorithm uses a bit-energy analytical model for computing the NoC communication energy. It considers that Dimension Order Routing is employed but, it can also generate a deadlock- and livelock-free routing function using the turn [30] and odd even [31] models. Additionally, network bandwidth constraints may be considered.

Before describing how our algorithm works, we present how we represent our NoC application mapping problem genetically, in terms of genes and chromosomes.

Each IP core is uniquely identified through a positive integer number, which forms a gene. Thus, the chromosome is a one-dimensional array containing non-repetitive positive numbers (coding the IP cores) and “-1” for empty NoC tiles. Each chromosome (individual) represents a mapping of cores onto NoC tiles. For a MxN 2D NoC mesh (M columns, N rows), the ith gene ($i = \overline{1, M \cdot N}$) from the chromosome encodes the IP core placed in the $\lceil i/M \rceil$-th row and $\begin{cases} i\%M, & i\%M \neq 0 \\ M, & i\%M = 0 \end{cases}$-th column (% is the modulo operator). Thus, a chromosome may contain multiple “-1” values. Its length is of MxN genes. Internally, before applying genetic operators, we replace this “-1” values with unique identifiers so that the chromosome contains a set of non-repetitive numbers. This allows us to work with genetic operators for permutation problems. After the genetic operator is applied, the “-1” identifiers are put back.

EGA starts with a population of randomly generated individuals. The population size is an algorithm parameter. It is set by default to 100.

EGA’s fitness function is given by the bit-energy analytical model (see section 3.1.3.2). Our algorithm uses the Binary Tournament Selection [112] technique, available in jMetal. It works by selecting an individual for reproduction as the individual with the best fitness from a set of two randomly selected individuals.

Reproduction consists of applying the crossover and mutation genetic operators. Our algorithm works by default with Position Based (PB) crossover and swap mutation. The crossover and mutation probabilities are algorithm parameters. By default, we work with 90% crossover probability and *1/n* mutation probability, where *n* is the number of NoC nodes.

We have implemented and integrated into jMetal the PB crossover. Swap mutation is available in jMetal. As we will show later on, we implemented other two crossover operators and another mutation operator. We also make use of the Partially Mapped Crossover (PMX), available in jMetal.

EGA stops after a specified number of generations. It may also be stopped after a given number of evaluated mappings.

7.3 Elitist Evolutionary Strategy

Elitist Evolutionary Strategy (EES) [112] is available in jMetal. We adapted this algorithm to our problem by using the same energy-aware fitness function like in the EGA case.

EES works very similar to a genetic algorithm, with the difference that the reproduction phase uses only mutation. The algorithm works with a population of size μ. At each generation, it generates λ offspring, $\lambda \geq \mu$. For every new generation, μ best individuals are selected from the previous population and from the λ offspring individuals. In our simulations, we set $\lambda=2\mu$. We present next the pseudocode for this algorithm.

```
1.  Randomly generate a $\mu$ size population
2.  Evaluate all individuals
3.  WHILE not stopping criteria meet
4.          WHILE not $\lambda$ offspring generated
5.                  Select a parent by any selection method
6.                  Create a new offspring by using mutation operator
7.                  Evaluate the offspring
8.          END WHILE
9.          Create new population (from previous population and $\lambda$ offspring)
10. END WHILE
```

Fig. 89 Elitist Evolutionary Strategy

7.4 Developing Problem Knowledge Crossovers

This section presents the crossover operators used in this research. We work with crossover operators for permutation problems. There are many such operators in literature (order, inversion, cycle etc.) [64]. We used Position Based Crossover and Partially Mapped Crossover. Position Based Crossover (PB) [119] aims keeping absolute position information during the recombination process. Partially Mapped Crossover (PMX) [120] tries to preserve genes' order, adjacency and position as much as possible. PMX is one of the most used crossover operators for permutation problems [112].

Next, we present two new crossover operators that we propose for the Network-on-Chip application mapping problem.

7.4.1 NoC Position Based Crossover (NPB)

NoC Position Based Crossover (NPB) extends PB so that the cores that are kept fixed are not selected randomly. We rather keep fixed the hot spot cores, i.e. the cores which communicate the most data. We sustain our approach by arguing that moving hot spot cores is more likely to produce worse mappings than moving less communicating cores.

NPB basically favors less communicating cores to rearrange themselves around the hot spot cores.

NPB starts by finding the first 50% cores which communicate the most data. After we have the list of hot spot cores, we evaluate the two parents, P_1 and P_2, using the following relation: $\text{cost} = \sum_{\substack{i,j \in C \\ i \neq j}} \text{vol}(c_i, c_j) \cdot \text{distance}(c_i, c_j)$, where C is the set of hot spot cores. This relation takes into consideration the data volumes and the Manhattan distance between the hot spot cores (we assume a 2D mesh topology but, the relation may easily be adapted for other topologies as well). We use this distance metric because we assume a network with XY routing. The parent that better mapped the hot spot cores is the selected parent. Next, an offspring is created by placing the hot spot cores in the positions they are in the selected parent. The rest of the offspring's genes are filled from the other parent, like PB does.

Our approach, based on hot spots, is similar to the approach from [72]. The difference is that, we do not simply swap the hot spot core with a randomly chosen core; we rather fix the first half of the most communicating cores. While the crossover from [72] behaves as a swap mutation, NPB acts as a Position Based crossover, with context-awareness.

7.4.2 Mapping Similarity Crossover (MS)

Our developed Mapping Similarity crossover (MS) has the purpose of identifying the topological similarities between two (parent) mappings and replicating them in the offspring. MS has two phases. The first phase tries to identify the mapping similarities between the two mappings. By doing so, the common characteristics of the two mappings are identified. The cores mapped in a similar way in both parents are mapped the same in the two children: child 1 maps the similar cores like parent 1 and child 2, like parent 2. We should point out that the offspring keep the common characteristics of their parents, either good or bad. The goal of the first MS phase is to decide which genes the offspring inherit from their parents. MS attempts to improve the offspring through a secondary phase, which performs a greedy mapping for the rest of the genes. This phase tries to raise the children fitness by rearranging the cores which are not mapped similarly, hoping they will be placed better with respect to the similar cores. We present next, in detail, how MS works.

MS starts from two parent individuals P_1 and P_2. For each parent, it computes the distance array D. Its elements are given by $D[i] = \sum_{\substack{i,k \in C \\ i \neq k}} d(i,k)$. C is the set of IP cores. The term $d(i, k)$ gives the Manhattan distance between communicating cores i and k ($D[i] = 0$ when core i does not communicate with any core). Note the computation of D is topology dependent. We assume a 2D mesh NoC but, our approach may be applied to other topologies as well. Let D_1 be the distance array for P_1 and D_2 the distance array for P_2. Then, we define a similarity function S that returns a binary array so that its elements are defined as:

$$S[i] = \begin{cases} 1, & D_1[i] = D_2[i] \\ 0, & D_1[i] \neq D_2[i] \end{cases}, i \in C$$

We observe $D_1[i] = D_2[i]$ even when the cores communicating with core *i* are at different distances from *i*, in P_1 and P_2. This happens when the sum of distances is the same. This similarity function applies a transformation on the mappings' representation. From the topological graph, the similarity function allows us to only see the communicating cores between which the distances are the same. For example, consider three cores: C_1, C_2 and C_3. Let C_1 communicate with C_2 and C_2 with C_3. In both P_1 and P_2, the distance between C_1 and C_2 is 1 and between C_2 and C_3 is 2.

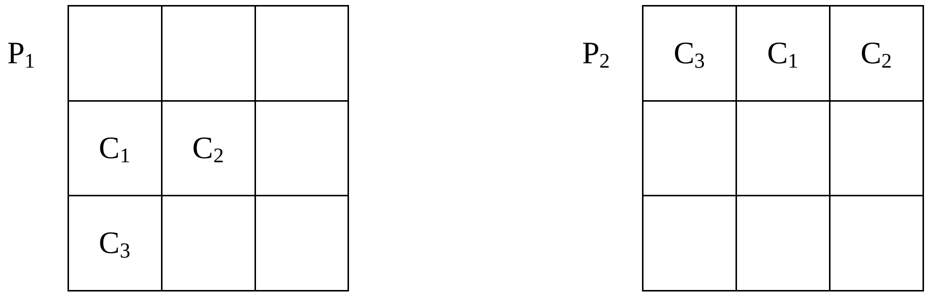

Fig. 90 Example of similar mappings, for 3 cores on a 3x3 2D mesh NoC

Our function *S* returns the array [1, 1, 1, 0, 0, 0, 0, 0, 0] (we assumed there is no similarity between the rest of the cores, not shown in the above figure). It shows the placement of these first three cores is similar in both parents, even if the cores are placed, in P_1, in the lower left corner of a 2D mesh and in P_2, in the upper part. The similarity function tries to identify which cores are mapped the same way in both parent individuals, being aware of the symmetries that exist in a NoC topology like the 2D mesh. The example above illustrates the motivation behind MS phase one: the symmetries from 2D mesh topological graph can easily lead to many different core placements which are actually having the same energy cost. A criticism to this approach might be that MS can easily propagate "bad" similarities. By "bad" similarity we understand cores that are at big distance from one another but, since they are at exactly the same distance in both mappings, they are considered similar. We argue that we should not make a distinction between "good" and "bad" similarities in our case. Firstly, we do not know for sure when a similarity is "bad". For example, even if two communicating cores are placed at five hops from each other, that might be the optimum distance between them. Secondly, we should propagate all common characteristics from parents to their children. By doing so, MS has higher chances to keep the crossover disruption rate low (obviously, the bigger the sequence of genes that is transferred from the parent to the child, the lower is the probability of a child to produce perturbations). Our crossover operator has a parameter that restricts the similarity function to work with cores that are at most *H* hops away. *H* ranges from one to infinity. When *H* is infinite, *S* is unrestricted. According to the schemata theorem, to maximize the chances of preserving the hyperplane samples, the crossover's disruptive effects should be minimized [121]. However, it is not enough to consider how often (*when*) an individual is disrupted but also *how* it will be disrupted [122]. The second MS phase addresses this issue.

The rest of the cores will then be remapped in a greedy manner. It is greedy because we want the crossover operator to be fast. We order the cores left for mapping based on how many similar cores they communicate with. Then, we position each core in an empty place, so that the Manhattan distance between it and the already placed cores is minimized. Therefore, MS second phase is actually a fast and simple partial mapping algorithm. Obviously, this greedy approach may easily be replaced with other mapping techniques, by simply restricting the similar cores to their given positions. This MS phase attempts to disrupt the hyperplane samples so that the genetic algorithm exploration process improves. The disruption rate is controlled by the first MS phase.

We argue our MS crossover operator does not simply act as a swap mutation operator like in the research of Ascia et al. [64], [115], [72]. MS instead tries to identify mapping similarities, which are inherited from both parents. This emphasizes the crossover character.

7.5 Mutation Operators

This section presents the two mutation operators used in this research. We chose to work with swap mutation, which is a very common genetic operator in permutation problems. It simply interchanges two randomly selected genes.

Using our developed Optimized Simulated Annealing algorithm as a mutation operator we obtain a hybrid algorithm: an Evolutionary Algorithm which incorporates a Simulated Annealing technique. OSA performs a context-aware mapping and outputs two cores which must be swapped. It performs an iteration each time it gets called by the Evolutionary Algorithm. When the number of iterations reaches OSA’s number of iterations per temperature level, the annealing temperature is decreased.

By using OSA as a mutation operator, we propose using hybrid algorithms for NoC application mapping. More precisely, we have a meta-heuristic, with an Evolutionary Algorithm as the main algorithm. The EA encapsulates a NoC specific algorithm, as a mutation operator (OSA). This approach allows us to benefit from the intrinsic parallelism that EAs contain. Also, the exploration has context-awareness, through the proposed mutation. The mutation may be performed by any algorithm for NoC application mapping. Any EA using a mutation operator may be used.

7.6 Multi-objective Optimization

NoC communication energy is minimized by placing the communicating IP cores as close as possible, onto the NoC tiles. Since we are interested to evaluate the performance of the genetic operators used in this research, we do a multi-objective optimization, too. Our second objective is to do a thermal-aware placement of the IP cores. Uniformly distributing the IP cores’ temperature across the network leads to the minimization of the hotspot temperature. Two IP cores that consume significant power should be placed at a greater distance from one another. However, this means our thermal balance objective is in contradiction with our energy objective.

For thermal balanced NoC design, we adopt the approach from [117]. The NoC architecture is modeled as a matrix. Each matrix element is a NoC node with an IP core. In order to measure the IP cores temperatures, we use the HotSpot tool [123]. UniMap can automatically generate a floorplan corresponding to a NoC application mapping. The floorplan is a matrix of regular NoC tiles. We consider each tile has a core and a router. The tiles size is the sum between router area and the area of the largest IP core from the E3S [56] library. This library also gives us the IP core power consumption (based on how much power a core requires to process the assigned application tasks). We used ORION

[96] to obtain the size of a typical NoC router. This is everything required by HotSpot to provide the IP cores steady state temperatures (measured in Kelvin). The NoC matrix is divided into *t x t* sub-matrices, where *t* is the heat transfer ability. The larger *t* is, the more neighboring IP cores will be affected by the source IP core. For each sub-matrix, we sum the temperatures associated with the cores from it. Considering all these sub-matrices, the goal is to minimize the maximum sum (thus, the fitness function will be $\frac{1}{\max\{submatrices\ sums\}}$). For *t = 1*, the problem is trivial: any mapping is optimal. Taking into consideration the conclusions from [124], we work with *t = 2*.

Each time the multi-objective genetic algorithm (NSGA-II or SPEA2) needs to evaluate a NoC mapping, the bit energy model is used to compute energy and HotSpot is called during the thermal balance evaluation.

7.7 Simulation Methodology

This section presents the simulation methodology used for evaluating and optimizing Evolutionary Algorithms for Network-on-Chip application mapping. We work with: our developed Energy- and performance-aware Genetic Algorithm (EGA), Elitist Evolutionary Strategy (EES) and we use OSA as a baseline. For both EES and EGA, we used two mutation operators: swap mutation and OSA mutation. For EGA, we have also worked with different crossover operators: Position Based (PB), Partially Mapped (PMX) and respectively our developed NoC Position Based (NPB) and Mapping Similarity (MS). Our multi-objective evaluations work with the jMetal NSGA-II and SPEA2 genetic algorithms, with the above genetic operators.

We use in this research all the benchmarks presented in Chapter 5. We have considered the most common Network-on-Chip architecture: a 2D mesh with regular tiles, using wormhole switching and XY routing. The NoC topology size is a simulation parameter. All benchmarks are mapped on 2D meshes with sizes exactly like in Section 6.3. Like in Chapter 6, we combined some benchmarks to be able to work with big NoC meshes. VOPD 4x has four times the VOPD (CTG 0) benchmark. *all-mocsyn* contains all the E3S benchmarks. The *97-cores* benchmark is made of: H.264 (CTG 1), MMS (CTG 0), MMS (CTG 1), MPEG-4, MWD and VOPD (CTG 0). The *131-cores* benchmark combines all non E3S benchmarks. Finally, by combining all benchmarks we get *215-cores*.

In order to increase the simulations' accuracy we have mapped each application 1000 times, with each algorithm. For each simulation, the initial mapping was randomly chosen. To make the comparisons fair, we have set the seed of the random number generator so that all algorithms start from the same point in the search space, every simulation. Thus, simulation *i* works with seed *i*, $i = \overline{1,1000}$. Benchmarks VOPD 4x and *all-mocsyn* run only 100 times, and *97-cores*, *131-cores* and *215-cores* only 10 times. This is because processing these benchmarks takes considerably more time [106].

We are interested to see how mutation probability influences the algorithms. Since we work with context-aware mutation, it is not clear to us if mutation probability should be low or high. Also, we want to find out how the proposed crossover operators perform. We argue it is also interesting to find out at what mutation probability level each crossover works best. Therefore, we varied the mutation probability, from 10% to 100%

in steps of 10%. We divided the simulations evenly: 10% simulations per mutation probability. In fact, it is unclear if crossover has higher importance than mutation to an Evolutionary Algorithm [125]. However, because we are here more interested about crossover than mutation and due to the high number of simulations, we decided to work with a constant crossover probability of 90%. We also fixed EA's population size to 100 individuals. Since we use OSA as a baseline algorithm, we limited EGA and EES to perform the same number of evaluations like OSA does.

7.8 Experimental Results

We present next only the most representative results obtained with our research on domain-knowledge evolutionary algorithms for Network-on-Chip application mapping. Our entire set of results is available in [126].

We start by measuring the mapping cost found for each benchmark. Since (in order to improve the accuracy of our results) we map the same application multiple times, we obtain an average mapping cost (energy). We get such average cost for every evolutionary algorithm, with every crossover operator and for each mutation probability. For EGA with MS crossover and OSA mutation, we also limit the similarity function to the IP cores that are one – EGA-MS-OSA (1) – or two hops away – EGA-MS-OSA (2).

We work with the metric that we call **Normalized Absolute Deviation (NAD) from the minimum** (in this case the minimum average energy). This metric is based on the statistic absolute deviation (*AD*) metric. Because we deal with a minimization problem, we consider the absolute deviation from the minimum average cost from the entire data set (X_b). Then, we normalize *AD* by dividing it to $\max\{X_b\}$ (the maximum average cost from the entire data set). Therefore, the normalized absolute deviation (of data point $x_{b_m} \in X_b$) from the minimum is $NAD_{b_m} = \frac{x_{b_m} - \min\{X_b\}}{\max\{X_b\}}$. The index b_m marks a benchmark evaluated with an algorithm with at a certain mutation probability. For each benchmark, we obtain its NAD, at every mutation rate, using the above formula. The data set X_b contains the average mapping energies obtained by all evolutionary algorithms, for the specified benchmark. At each mutation level, we average the NADs of all our benchmarks. Therefore, we have $Average\, NAD_m = \frac{NAD_{1_m} + NAD_{2_m} + ... + NAD_{B_m}}{B}$, with B being the number of benchmarks and m the mutation probability ($m \in \{10\%, 20\%, ..., 100\%\}$). We use this metric because directly comparing the average energies obtained for different applications is infeasible since each application has its own energy domain (which usually differs significantly).

Figure 91 presents how much the average mapping cost deviates from the minimum average cost, found by all algorithms. We show the results obtained only for the big benchmarks (VOPD 4x, *all-mocsyn*, *97-cores*, *131-cores* and *215-cores*) because our evolutionary algorithms perform similarly on the rest of the benchmarks (in terms of average mapping cost). For every algorithm, only the point corresponding to the mutation probability where it performed best is shown.

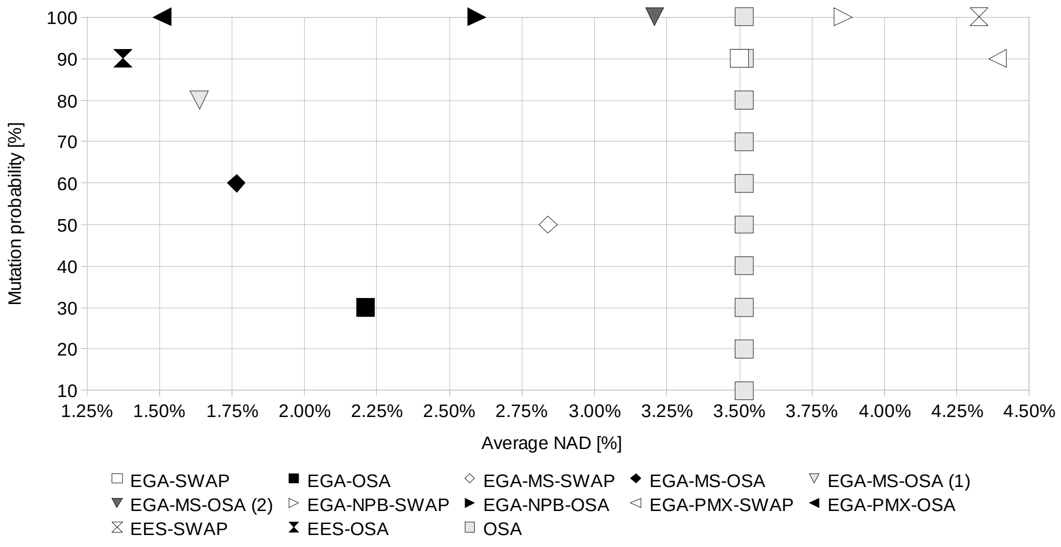

Fig. 91 Algorithms' comparison based on their average normalized absolute deviation, from their common minimum average cost (only big benchmarks)

It may be easily observed that all algorithms perform significantly better with OSA mutation than with swap mutation. EES-OSA has the smallest deviation, among all algorithms, followed by EGA-PMX-OSA. EGA-MS-OSA is the next best performing algorithm in this case. We even notice a slightly better performance for EGA-MS-OSA (1) (1.64% deviation) than for EGA-MS-OSA (1.77% deviation). However, EGA-MS-OSA (2) performs much worse (3.21% deviation, at 100% mutation probability). After EGA-MS-OSA we have EGA-OSA and EGA-NPB-OSA. EGA-OSA is the algorithm that gives the smallest deviation at the lowest mutation rate: 30%. EGA-NPB-OSA is better than EGA-MS-SWAP. Still, EGA-MS-SWAP clearly beats OSA, making MS the only crossover than outmatched OSA with both mutation operators. Mapping Similarity is the only crossover operator that performed well regardless of the mutation operator. PB crossover also does not provide bad results but, MS is clearly better (EGA-SWAP has a 3.52% deviation, with only 0.02% smaller than OSA's deviation). The performance of our other crossover operator, NPB, is not good when we compare it with PB. In both cases (OSA or swap mutations), NPB performs worse than PB. However, we observed (on all benchmarks) that NPB performed better and better as mutation grew. It performed best at 100% mutation probability (similarly to PMX). PB performed best at 80% mutation (on all benchmarks). Raising the mutation made PB perform worse. EGA with MS performed best at 50% - 60% mutation rate, on all benchmarks. We may conclude that MS is the crossover operator that contributes the most at obtaining a good average mapping cost. The rest of the operators rely significantly more on the mutation operator.

In conclusion, in terms of average mapping cost, the Elitist Evolutionary Strategy with OSA mutation performs the best. The Energy Aware Genetic algorithm has the best behavior with OSA mutation and with PMX crossover. Our developed Mapping Similarity crossover gives similar results: its normalized absolute deviation is with only 0.25% worse than the one of PMX. NPB performs worse on the big benchmarks. Its deviation is with 1% higher that the one of PMX.

Similarly to the average normalized absolute deviation metric, we also computed the minimum NAD and the maximum NAD.

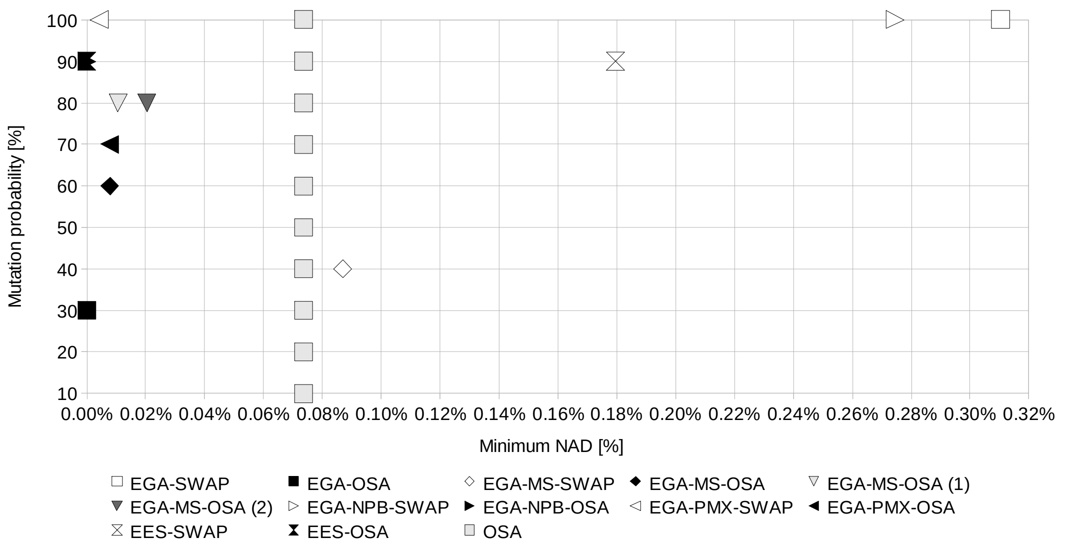

Fig. 92 Algorithms' comparison based on their minimum normalized absolute deviation, from their common minimum average cost (only big benchmarks)

In almost all cases we obtained the minimum NAD at the same mutation probability where the average NAD is. From this point of view, the biggest difference is for EGA-PMX-OSA, which has the average NAD at 100% mutation rate and the minimum NAD at 70%. EES-OSA, EGA-NPB-OSA and EGA-OSA have zero minimum NAD. This means these algorithms found for at least one big benchmark the minimum average mapping cost.

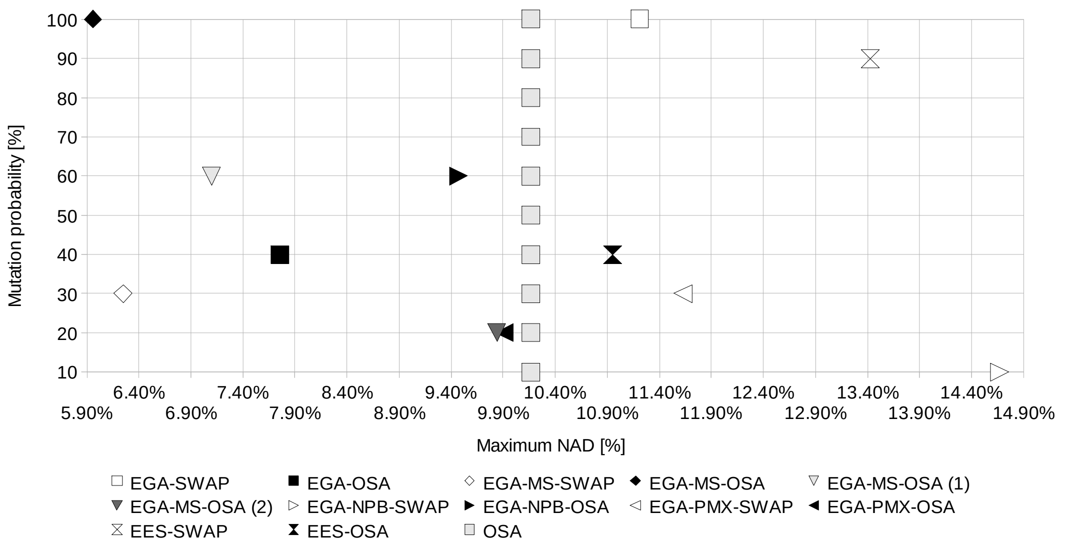

Fig. 93 Algorithms' comparison based on their maximum normalized absolute deviation, from their common minimum average cost (only big benchmarks)

The smallest maximum normalized absolute deviation was obtained with our developed MS crossover (with and without OSA mutation). EGA-MS-SWAP is the single algorithm with a maximum deviation that is lower than OSA's. EES-OSA is the only algorithm with OSA mutation that has a maximum NAD exceeding OSA's maximum deviation.

Next, we are interested to find how good are the best mappings found by each algorithm. In order to compare the best solutions found by all algorithms, we have identified for each application the best solution found by all algorithms. Then, for each application, with each algorithm and mutation probability, we have computed the additional energy (*AE*) its best mapping consumes, with respect to the best solution found by all algorithms. Using the same notation like for NAD computation, we define the Additional Energy metric as $AE_{b_m} = \frac{x_{b_m} - \min\{X_b^{'}\}}{x_{b_m}}$. In this case however, we work with a different data set. $X_b^{'}$ contains the minimum mapping energies (not the average ones, like in the previous case). Finally, like for average NAD, we averaged the additional energies for all benchmarks. The following chart presents these results. We show for every algorithm the value at the mutation level where it obtained the lowest average additional energy.

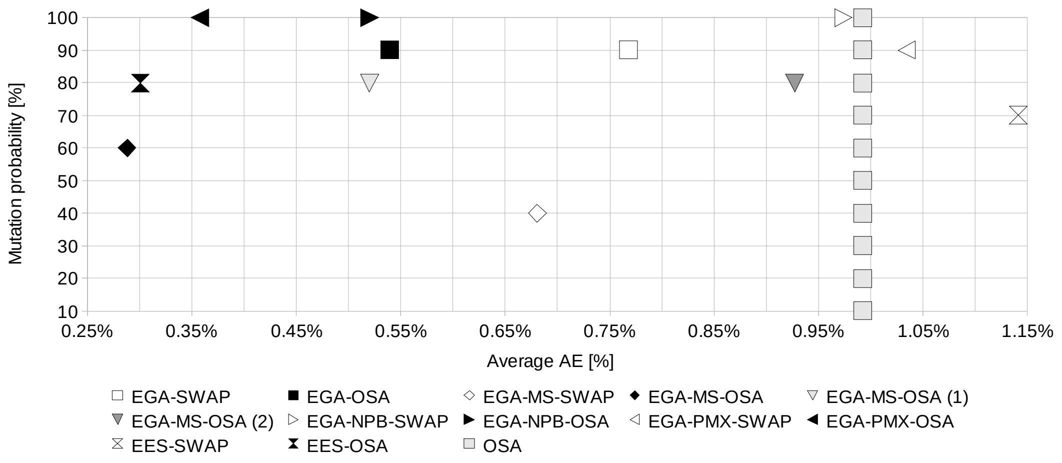

Fig. 94 Average additional energy consumed by the best mappings found by each algorithm, compared to the best mappings found by all algorithms

EGA-MS-OSA is the algorithm that has the most mappings that are the best. On average, the best mappings found with this algorithm introduce just 0.29% additional energy. Very close to this result is EES-OSA, with 0.3% additional energy. After EGA-PMX-OSA (0.36%), follow EGA-NPB-OSA and EGA-MS-OSA (1), both with 0.52% additional energy. Note that all the algorithms except EES-SWAP and EGA-PMX-SWAP find, on average, better mappings than OSA. This chart also shows that swap mutation produces worse mappings than OSA mutation, regardless the algorithm. However, there is an exception: EGA-MS-OSA (2) does not produce better mappings than all algorithms with swap mutation.

We conclude our solution quality based analysis by showing how often each algorithm manages to reach the best solution. We refer to the best solution found by all algorithms, not to the best solution each algorithm found. Hence, it is possible an algorithm has a zero best solution percentage. We define the Averaged Best Solution percentage at mutation rate *m* as $Average\,BS_m = \frac{BS_{1_m} + BS_{2_m} + \ldots + BS_{B_m}}{B}$ [%]. BS_{b_m} is the Best Solution percentage for benchmark *b*, at mutation level *m*. It represents how many times an algorithm finds the best mapping, found by all algorithms.

On the big benchmarks, OSA is unable to find the best solution. Also, not all of the evolutionary algorithms manage to reach the best solution. This may be seen in the following figure.

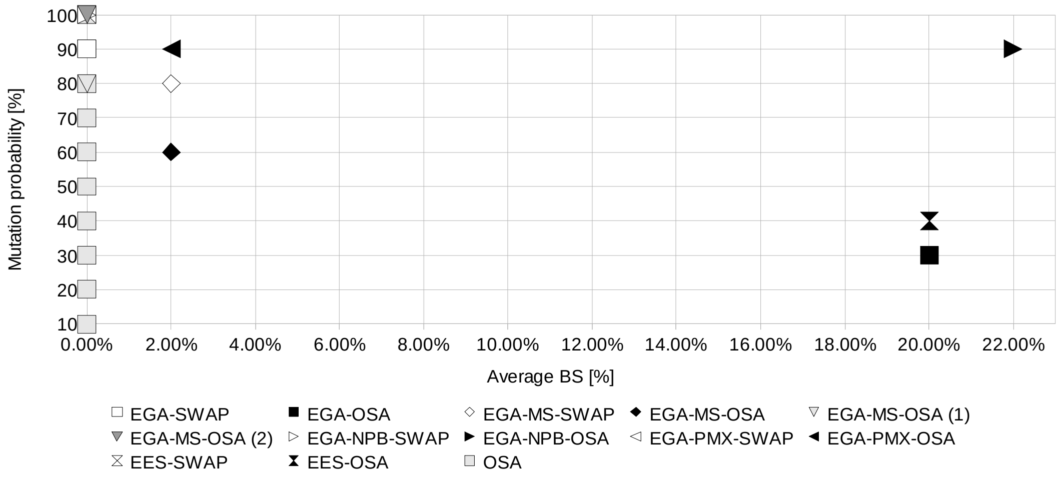

Fig. 95 Average best solution percentage on big benchmarks

EGA-NPB-OSA is the algorithm that has the highest best solution percentage, which is 22% at 90% mutation probability. EES-OSA and EGA-OSA have a value of 20%. Than, with just 2%, we have EGA-PMX-OSA, EGA-MS-OSA and EGA-MS-SWAP. We notice all the algorithms using swap mutation are unable to reach the best solution. The only exception is EGA-MS-SWAP.

Our conclusion is that NPB crossover gives the best solution percentage, on the big benchmarks. Mapping Similarity and PMX crossovers give a similar best solution percentage. OSA mutation is essential for EES because with swap mutation EES performs worse even than EGA with NPB crossover and swap mutation.

Regarding the optimal mutation probability, we observed there are algorithms, like EGA-NPB-SWAP, for which we obtained exactly the same mutation rate. However, in general there is no ideal mutation probability. Our experiments indicate the optimal mutation probability may vary from 20% up to 100%. For EGA-MS-OSA, we got the same optimal mutation probability in terms of average and best mapping cost. We conclude that mutation probability is application and algorithm dependent. The lowest mutation rate is consistently encountered when working with our developed Mapping Similarity crossover. This indicates MS is the crossover operator that relies the least on mutation to find the best NoC application mapping. We tried to limit the similarity function of MS by considering only the cores which are one or two hops apart in the NoC. Overall, we did not obtain significantly better results. EGA-MS-OSA (1) and EGA-MS-OSA (2) require a higher mutation probability to function optimally.

We present next how some of our algorithms converge in time. Since the previous results showed us that OSA mutation gives better results than swap mutation, we focus only on these algorithms: EGA-OSA, EGA-PMX-OSA, EGA-NPB-OSA, EGA-MS-OSA and EES-OSA. We ran each of the five algorithms for 1000 generations per application. To improve the accuracy of our simulations, we ran each application for 100 times (by setting the random number generator seed from 1 to 100). Finally, we averaged the energy cost of all 100 mappings per application and per generation. We worked with the mutation values determined by our average cost analysis.

Fig. 96 shows how the five algorithms converge on our biggest benchmark. We mention that for all the other benchmarks we obtained the same behavior, as we will detail next.

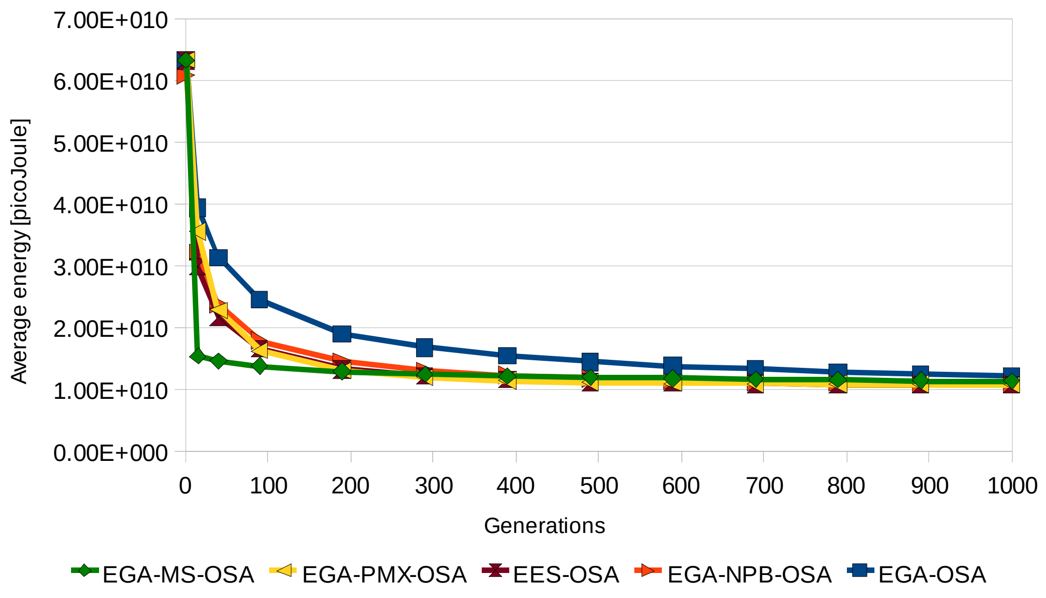

Fig. 96 Algorithms' convergence for 215-cores benchmark

All algorithms manage to reduce the mapping energy significantly, within the first 100 generations. EGA-OSA has the lowest convergence speed. EGA-PMX-OSA, EGA-NPB-OSA and EES-OSA behave approximately the same. EGA-MS-OSA is the algorithm that converges the fastest during the first generations. After that, its convergence speed decreases and it is outrun by EGA-PMX-OSA, EGA-NPB-OSA and EES-OSA. We believe this is justified by the greedy approach from the second phase of our Mapping Similarity crossover.

We measured when each algorithm reaches its best solution during the 1000 generations, for each benchmark and we averaged the results. EGA-OSA converges in 732 generations. It requires the most number of generations to obtain its mapping with the best communication energy. EGA-MS-OSA converges in 562 generations.

Algorithms EGA-PMX-OSA, EGA-NPB-OSA and EES-OSA require 475 generations. EES-OSA is the algorithm that, on average, has the fastest convergence speed (424 generations).

Finally, we switch from a single objective to multi-objective Network-on-Chip application mapping. Besides minimizing communication energy, we are now also interested in obtaining a thermal balanced NoC design. Using NSGA-II and SPEA2 genetic algorithms implemented in the jMetal library, augmented with all our genetic operators, we evaluated NoC mappings for *all-mocsyn*. This is the benchmark that contains all E3S applications. For E3S we know how much power the IP cores consume to execute a particular task. Each algorithm ran once, with each crossover – mutation combination, for 1000 generations. Each time we started from the same initial population. We used the optimal mutation probabilities determined by our average mapping cost analysis.

Fig. 97 shows, for every algorithm, the (normalized) hypervolume [113] obtained at each generation. For a minimization problem (like ours), the hypervolume is defined as the volume enclosed by the Pareto front and a reference point. The coordinates of this point are determined by the maximum values of the objectives. The values are also normalized using the (constant) volume between the coordinate systems' origin and the hypervolume reference point.

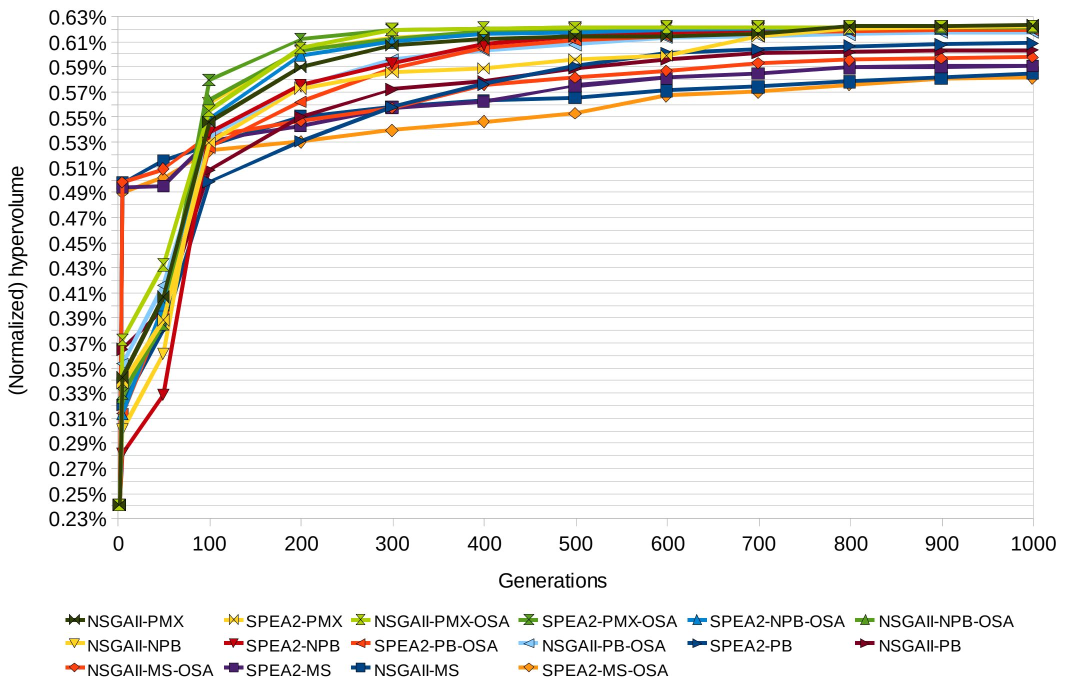

Fig. 97 (Normalized) hypervolumes, for all evaluated algorithms

The hypervolume grows significantly until the first 200 – 300 generations, for all algorithms. Then, it keeps growing slowly until the last generation. This indicates how the algorithms converge. The algorithms using our developed Mapping Similarity crossover have a very fast convergence speed within the first 100 generations. However, in the end, their hypervolumes are the smallest. This indicates MS leads to the worse performance in this multi-objective case. However, if we want fast results, then this will be a suitable crossover. Looking at the hypervolume values within the last generations, we ordered the algorithms. This order may be seen in the chart's legend. It may be observed that PMX performs the best. It is followed by NPB, PB and finally MS. We also observed that both NSGA-II and SPEA2 performed better with PMX and swap mutation. The performance with OSA mutation was worse. These multi-objective results appear to be in contradiction with our previous single-objective results. The explanation resides in the fact that our two objectives are in a mutual contradiction. OSA mutation, MS and NPB crossover work to optimize energy but, this implicitly leads to worsening the NoC mappings in terms of thermal balance. NPB is more suitable than MS (in this case) because it just identifies hot spot cores, in terms of energy. However, they may also be in terms of thermal balance because a highly communicating core might also have a higher temperature.

Fig. 98 shows the Pareto front obtained in the last generation by combining the Pareto fronts of all the evaluated algorithms. This combined Pareto front holds only the non-dominated individuals from all the merged Pareto fronts.

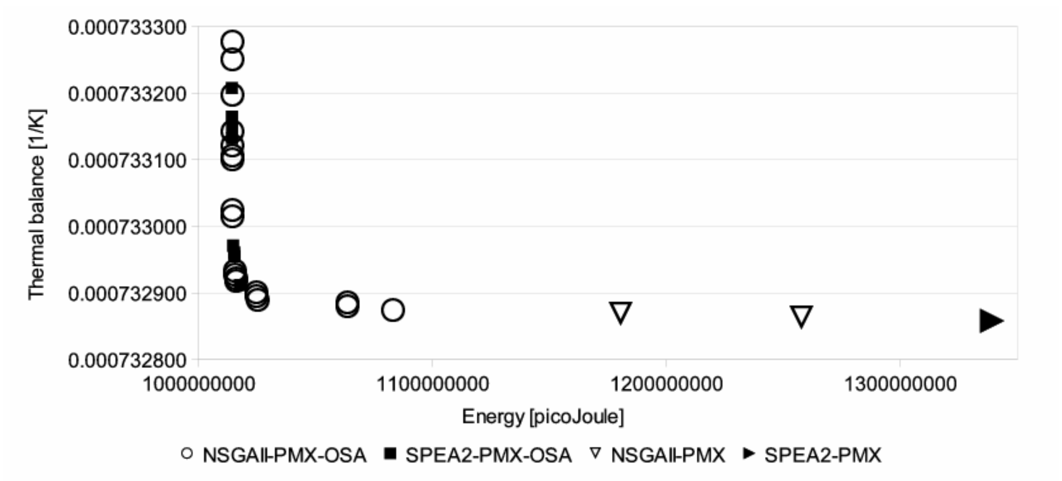

Fig. 98 Combined Pareto front (generation 1000)

It may be seen that PMX is the single crossover that leads to the best solutions, found by either NSGA-II or SPEA2. We also observe there are a lot of good solutions in terms of energy. All these mappings were found using OSA mutation. With swap mutation, we managed to find three good solutions in terms of thermal balance. The significantly higher number of good energy-biased solutions indicates the fact we tried to optimize only energy with NoC application mapping knowledge. Probably using a crossover which also optimized energy was too much bias towards a single objective. This is how we explain PMX was the best performing crossover. Anyway, PMX was one of the best performing crossovers in the single-objective case, too.

7.9 Summary

We presented in this chapter the process of design, evaluation and optimization of some efficient domain-knowledge evolutionary algorithms for Network-on-Chip application mapping. We developed in UniMap an Energy- and performance-aware Genetic

Algorithm (EGA). EGA is integrated into jMetal library, which is now part of UniMap. Along with EGA, we also evaluated the Elitist Evolutionary Strategy (EES) algorithm.

We showed all our evaluated algorithms work better with OSA mutation than with swap mutation. One may easily introduce any NoC application mapping algorithm, into any evolutionary algorithm, as a mutation operator. The influence of the introduced algorithm may be easily controlled through mutation probability.

Besides optimizing our evolutionary algorithms with context-aware mutation, we also developed two specific crossover operators: NoC Position Based and Mapping Similarity. We also worked with other two standard crossover operators for permutation problems: Position Based crossover, developed by us as a jMetal extension and Partially Mapped crossover, available in jMetal. Having four crossovers, two mutations, a genetic algorithm and an elitist evolutionary strategy, we evaluated twelve variants of evolutionary algorithms. All these algorithms were compared with the Optimized Simulated Algorithm, which served as a baseline. We found that all the best solutions were found by evolutionary algorithms (none by OSA), for the big benchmarks. This proves EAs are superior to OSA for NoCs with tens, hundreds of nodes.

We tried to find out how crossover and mutation operators influence the evolutionary algorithms. We performed an extensive solution quality analysis by measuring the average mapping energy, the best mappings' cost and the best solution percentage. We found MS to be the crossover operator that contributes the most at obtaining a good mapping. MS works best at 50% - 60% mutation probability. The rest of crossovers require higher mutation rates to get their best average mapping cost (PMX and NPB work best at 100% mutation, in terms of average cost). Mapping Similarity is the only crossover operator that performs well, on average, regardless of the mutation operator. EGA obtained the best average mapping costs with PMX and MS crossovers. By measuring the best solution percentage, we found that NPB leads EGA to the highest value, on the big benchmarks.

We also tried to improve the performance of our Mapping Similarity crossover, by limiting the similarity function to IP cores that are one or two hops away from each other. However, we did not find significantly better results.

Our developed genetic algorithm performed, in general, the best with MS and PMX crossovers. However, our best results were obtained with Elitist Evolutionary Strategy, using OSA mutation. EES-OSA was the algorithm that also converged the fastest. Although we managed to improve the genetic algorithm through our crossover operators, using an algorithm that works only with (context-aware) mutation proved to be better. Finding a suitable context-aware crossover for NoC application mapping is more difficult than finding an efficient context-aware mutation.

We also tried to find the optimal mutation probability for our algorithms. However, this depends a lot on the algorithm and on the benchmark. Our optimal mutation probabilities range from 20% to 100%.

Finally, we did a multi-objective evaluation using NSGA-II and SPEA2, improved with our genetic operators. Since our two objectives are contradictory, our developed operators did not lead to the best performance. However, we did find the best solutions, in terms of energy, with OSA mutation. A suitable crossover operator for the NoC application mapping problem is even more difficult to find if we consider multi-objective optimization.

"The greatest intelligence is precisely the one that suffers most from its own limitations."

André Gide

8 Application Driven Automatic Design Space Exploration for System-on-Chip Architectures

In this chapter we propose a method for performing an application driven automatic design space exploration for System-on-Chip (SoC) architectures. We integrate UniMap with a Framework for Automatic Design Space Exploration (FADSE) [127] with the purpose of automatically finding the best SoC design for any given application, in a multi-objective way. Our objectives are: SoC energy consumption, SoC area and application runtime.

Using UniMap's features, we simulate an entire computing system, consisting of tens of heterogeneous IP cores that are mapped onto the nodes of a Network-on-Chip.

FADSE automatically configures this System-on-Chip. It then simulates it using UniMap's simulator and gives the simulation results to the DSE algorithm that drives the search process.

We show a feasible DSE workflow that meets our requirements and we identify the most suitable SoC architectures, for a given application, in terms of energy, area and runtime. We also compare four DSE multi-objective algorithms (two genetics and two bio-inspired) with the purpose of identifying the algorithm that performs the best.

8.1 Related Work

We present next two examples that stress out the importance of an application driven automatic design space exploration for Network-on-Chip designs.

An exhaustive design space exploration of Network-on-Chip architectures is performed in [128]. The evaluated design space is obtained by considering NoC parameters like: the number of network nodes, the network topology (butterfly, flattened butterfly, fat tree, mesh and ring), the routing mechanism (static, minimal, oblivious source routing), the switching technique (store-and-forward and wormhole) and the size of messages, packets and buffers. It is shown that there is *no* NoC architecture, from the researched design space, which is optimal across all of the used traffic patterns. For example, a certain topology could best accommodate only some certain traffic patterns or, if there is a best topology, it is very likely that the network resources must be allocated differently, from workload to workload, so that the optimal performance is achieved. As such, the authors of this paper propose a configurable device, capable of emulating different NoCs. Such a device is called a *polymorphic network* because it allows the configuration of the network topology, links width and buffers for each application. It allows the instantiation of an arbitrary network, prior to application runtime. The Operating System can configure the polymorphic network through a stream of bits which contain the network configuration information for a particular application.

A polymorphic network is basically built from many buffers and crossbars. The approach has the obvious advantage of flexibility. The NoC resources can be interconnected in a lot of ways, which allows for different networks to be instantiated. The main drawback of a polymorphic network is the required area budget.

Another example is related to the size of the NoC input buffers. It is shown in [91] that using non-uniform input buffers determines a significant increase of the network's performance. This is because, especially when the network is congested, the amount of buffering resources severely affects the performance of the Network-on-Chip architecture. The authors propose an algorithm that detects the buffer which has the highest probability to be full. The size of that buffer is then increased because the channel that owns it would otherwise become a real performance bottleneck. The algorithm uses as inputs some of the parameters of the system: the traffic pattern, the routing delay and the current size of each buffer from the network.

Therefore, there is a huge number of possible NoC designs. Which is the most suitable NoC architecture for a particular application must be determined using Design Space Exploration (DSE) techniques. These methods imply heuristic algorithms, which find near optimal solutions by considering one or more objectives (e.g.: performance, energy consumption, area). Performing DSE requires a powerful, modular, flexible and highly configurable framework for automatic design space exploration.

xPipesCompiler [129] is a tool which allows the automatic instantiation of application-specific Network-on-Chip architectures. The blocks that make a NoC architecture (links, network interfaces and routers) are described in SystemC at cycle level of accuracy. One of the advantages of this tool is that it allows the user to create custom network topologies. This tool uses an input file to generate the network. The input files contain information about the cores, switches, links and the relationships between them. Routing tables for the network interfaces are also specified. This can be regarded as a disadvantage of the xPipesCompiler: the instantiated network does not actually use a routing protocol. The routing paths are specified by the user. The tool also performs optimizations of the network components. For example, if a switch has only its input link connected to a port, then the buffer and logic for an output port is not generated (because it would anyway not be needed). This allows for power and area savings. Another disadvantage of this tool is that, although it is application-specific, the network topology must be specified manually, i.e. it cannot be determined based on the characteristics of the application.

We have already presented in Section 4.1 two frameworks for Network-on-Chip design space exploration. SUNMAP is focused on NoC topology synthesis. The other DSE tool is focused on multi-objective algorithms for Network-on-Chip application mapping. UniMap, our developed unified framework for Network-on-Chip application mapping, implements several application mapping algorithms that can be used to map real applications onto different Network-on-Chip designs. The mappings can be evaluated in a multi-objective manner, using either analytical models or a NoC simulator (developed by us).

We present next an approach for automatically determining the "best" System-on-Chip (SoC) architecture, for any given (concurrent) application, from a multi-objective point of view. The model used by this approach integrates UniMap with a Framework for Automatic Design Space Exploration (FADSE) developed at our ACAPS research center in Sibiu – see http://acaps.ulbsibiu.ro. This application driven DSE for NoC architectures aims to optimally map the IP cores (running a particular application) onto the tiles of the most suitable Network-on-Chip design. We also try to find the most suitable IP cores by selecting them from a library of cores. We have three objectives: application runtime, SoC energy and SoC area. We are interested in finding that SoC design that executes the given architecture the fastest and it consumes the smallest amount of energy and it occupies the smallest area. These objectives are contradictory because improving one of them usually leads to a regression of at least one of the others.

8.2 Framework for Automatic Design Space Exploration

The Framework for Automatic Design Space Exploration (FADSE) [130], [131] is developed by Horia Calborean from “Lucian Blaga” University of Sibiu, Romania, as part of his PhD thesis [127]. FADSE is a client-server tool that includes many state of the art algorithms through jMetal [86]. It can connect to almost any existing simulator and then it can perform parallel evaluations with it. The FADSE server distributes evaluations to the available clients (the number of clients can be dynamically changed). Each client is instructed by the server to perform a simulation with a specified set of parameters. The parameter values are determined by an evolutionary algorithm, with respect to the objectives to be optimized. Since the DSE process can easily take a lot of time (weeks, months), FADSE is designed to be able to recover from situations like failing clients, network failure or even system power loss. Simulations are automatically resubmitted to other clients in case of problems and more importantly, FADSE integrates a checkpointing mechanism. This makes this tool reliable because the DSE process can easily be restarted. This approach effectively avoids restarting the design space exploration from the beginning. FADSE also uses a database for storing the simulation results. This allows the results of already simulated configurations to be reused, which reduces the time required to perform the exploration process.

FADSE was successfully used for a multi-objective, hardware-software co-design exploration of the design space for a superscalar system [132], [133]. We briefly present next the metrics and the some of the most representative multi-objective evolutionary algorithms available in FADSE.

8.2.1 Metrics

Hypervolume and coverage are some of the most important metrics implemented in FADSE for evaluating the DSE process and for comparing the search algorithms. We present them next because they are used in this chapter.

8.2.2.1 Hypervolume

For a minimization problem, the hypervolume [5] is defined as the volume enclosed by the Pareto front and a reference point. The coordinates of this point are determined by the maximum values of the objectives. The values are also normalized using the (constant) volume between the coordinate systems’ origin and the hypervolume reference point. For a maximization problem, the hypervolume is enclosed by the Pareto front and the coordinate system axes.

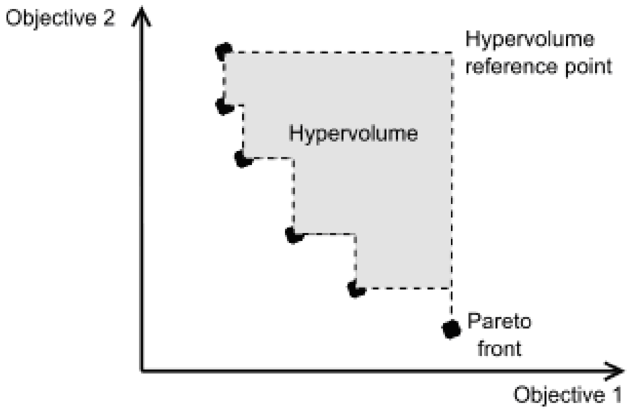

Fig. 99 Hypervolume for a two-objective minimization problem [109]

FADSE computes the hypervolume after each generation of the evolutionary algorithm. The evolution of this volume shows if the DSE algorithm makes progress. When the hypervolume saturates we can decide to stop the search process. Hypervolume is a useful metric to determine the convergence speed of an algorithm. It can also be used to compare the solution qualities of different algorithms, when the same hypervolume reference point is used for all algorithms.

8.2.2.2 Coverage

Coverage [5] is used to compare two algorithms. It is defined as the fraction of individuals, produced by the second algorithm, which are dominated by individuals produced by the first algorithm. For example, if *Coverage(Alg_1, Alg_2) = F%*, then *F* percent of algorithm's Alg_2 individuals are dominated by individuals of Alg_1 (one or more, we do not know how many). When *Coverage(Alg_1, Alg_2) =100%*, at least one individual from Alg_1 dominates all individuals produced by Alg_2.

This metric is used for determining which algorithm has a better solution quality than the other. Coverage is more suitable for evaluating solution quality than hypervolume is.

8.2.2 Multi-objective Algorithms

We present next four of the most representative evolutionary algorithms, available through FADSE, which we used in this research: NSGA-II, SPEA2, OMOPSO and SMPSO.

8.2.2.1 NSGA-II

Non dominated Sorting Genetic Algorithm (NSGA-II) [116] starts from a randomly generated initial population, which is evaluated. An offspring population is obtained by means of crossover and mutation. After the offspring is also evaluated, the two populations are composed into a single one and sorted according to the domination relationship. A fitness value is assigned to the individuals, which takes into consideration the previous sorting and a crowding measurement. The crowding specifies the density of these individuals on the Pareto front. The next population is formed by the best individuals according to their fitness.

8.2.2.3 SPEA2

Strength Pareto Evolutionary Algorithm (SPEA2) [65] is a genetic algorithm that uses an archive of non-dominated individuals. The fitness of each individual (*I*) is based on its strength, i.e. how many individuals it dominates. A raw fitness is computed as the sum of the strengths of the individuals that dominate *I*. To this raw fitness, density information is added. This density is defined as the inverse of the distance to the nearest k^{th} neighbor individual (*k* is a parameter of the algorithm). SPEA2 updates its archive by keeping only the best individuals from the offspring population and the archive. The updated archive represents the new population with which the algorithm continues.

8.2.2.4 OMOPSO

OMOPSO (Our Multi-objective Particle Swarm Optimization) [134] is a Particle Swarm Optimization (PSO) algorithm, which is inspired by the flight of birds in search for food. A swarm (population in genetic algorithms' terminology) has many particles (individuals), which "fly" through space following the best performing particle at that moment. Each particle is characterized by two parameters: position and speed. As every particle tries to get closer to the current best particle, its two parameters change. The change takes into account both the current global best and the particle's personal best solution found so far. Based on this change, the particle gets a new position and will be reevaluated. After all the particles are evaluated, the new global best particle is selected, the personal bests are updated and this entire process is reiterated. Obviously, a multi-objective PSO algorithm can have multiple global best particles (leaders). The leader is chosen using a sorting mechanism similar with the one implemented in NSGA-II. OMOPSO has an extra step in which some of the individual's genes are changed using mutation.

8.2.2.5 SMPSO

Speed constrained Multi-objective Particle Swarm Optimization (SMPSO) [135] is an algorithm similar to OMOPSO. The main difference is that the particle's speed is constrained so that it does not get too high.

8.3 Design Space Exploration Workflow

The ideal DSE workflow for our purpose is to search for the best NoC architecture for every possible mapping of IP cores onto NoC tiles. For each possible mapping we would have to search the most suitable NoC design. Each NoC design would be evaluated using UniMap's ns-3 NoC simulator. The complexity of this approach is $C_{IdealDSE} = {}_nP_c \cdot N \cdot C$, where *n* is the number of NoC nodes, *c* is the number of IP cores, *N* is the number of NoC architectures evaluated for each mapping and *C* is the number of IP core types that can be used for the *c* cores that execute the application. The first term describes the total number of possible mappings, i.e. in how many ways *c* cores can be arranged onto *n* NoC tiles. This number is factorial in size. $N = \prod_{k=1}^{p} P_k$, where *p* is the number of (ns-3) NoC parameters and P_k is the number of possible values that parameter *k* may take. Also $C = \prod_{k=1}^{c} c_k$, where c_k is the number of IP cores that can execute the tasks assigned to core $k = \overline{1, c}$ (we consider a heterogeneous system, i.e. not any core is suitable to any task). *N* and *C* are exponential numbers. It is obvious that $C_{IdealDSE}$ is a very big number. For example, for a small NoC with *n* = 16 nodes, there are 16! possible mappings of *c* = 16 IP cores. If our NoC has (only) four parameters, each with ten values, then N = 10^4. Also, let us consider that $C = 10^{16}$, i.e. for each group of tasks we have ten types of cores that can execute them. Thus, $C_{IdealDSE} = 16! \cdot 10^{20} \approx 2 \cdot 10^{33}$. If each simulation would take no more than one second (!), then we would need more than $63 \cdot 10^{24}$ years to perform an exhaustive search (on a single core system). Obviously this approach needs a tool like UniMap to heuristically search among the total number of possible mappings. A tool like

FADSE is also needed to limit the number of simulated SoC designs (we gain speed but, we find only near optimal solutions). Even so, this DSE approach would be extremely time consuming. We would still require more than 300 years for just ten thousands mappings and ten thousands SoC designs evaluated per mapping (one second per simulation).

The above DSE process is essentially a DSE in a DSE approach (an inner DSE). UniMap DSE encapsulates the FADSE. We will make the DSE workflow faster if we decouple UniMap DSE from FADSE DSE. This means we simulate each mapping on the same, single, SoC architecture. Then we use FADSE to search for the best SoC design only for mappings from the Pareto front found with UniMap DSE. The complexity of this DSE workflow is $C_{UniMapDSE\ ,FADSE} = {}_nP_c + P \cdot N \cdot C, \quad P << {}_nP_c$, where P is the number of UniMap DSE Pareto mappings. It is obvious that $C_{UniMapDSE,FADSE} < C_{IdealDSE}$ (at the expense of getting farther away from the optimal solutions). For the above example, if we consider ten Pareto mappings found with UniMap, this approach will drastically reduce the search time from 300 years to just 13 days. However, this second DSE workflow is still very time-consuming because a simulation will most like take more than just a second.

Rather than using a simulator for evaluating each mapping generated by UniMap on a default SoC architecture, we can use an analytic model to find out which mapping is better. Obviously this further reduces the accuracy of the DSE process but, it makes it faster and thus more feasible. The complexity of the DSE workflow is $C_{UniMapAnal\ ytic\ ,FADSE} = ({}_nP_c)_{analytic} + B \cdot N \cdot C, \quad B << {}_nP_c$, where B is the number of best analytic mappings. In this case, a mapping can be evaluated in less than a second. Using a bit energy model to estimate the communication energy required to send a bit of data from one NoC node to another, we evaluated a mapping in 0.04 ms on our HPC system [106]. Considering for the above example that B = 10, we get a search time of 12 days. Even if we got a similar time with the previous DSE workflow too, this estimation is obviously more realistic because previously we considered that UniMap uses a simulator for evaluating the mappings (only one second for simulating each mapping). In this case UniMap works just with an analytic model; it does not use the NoC simulator. We still work (for this rough estimation) under the assumption that a NoC simulation (generated by FADSE) takes just a second but, we mention that we can distribute simulations with FADSE on High Performance Computing systems.

It is obvious that $C_{UniMapAnalytic,FADSE} < C_{UniMapDSE,FADSE} < C_{IdealDSE}$. Because of this, we use the third DSE workflow in this research. The following figure illustrates our proposed application driven automatic design space exploration workflow for System-on-Chip architectures.

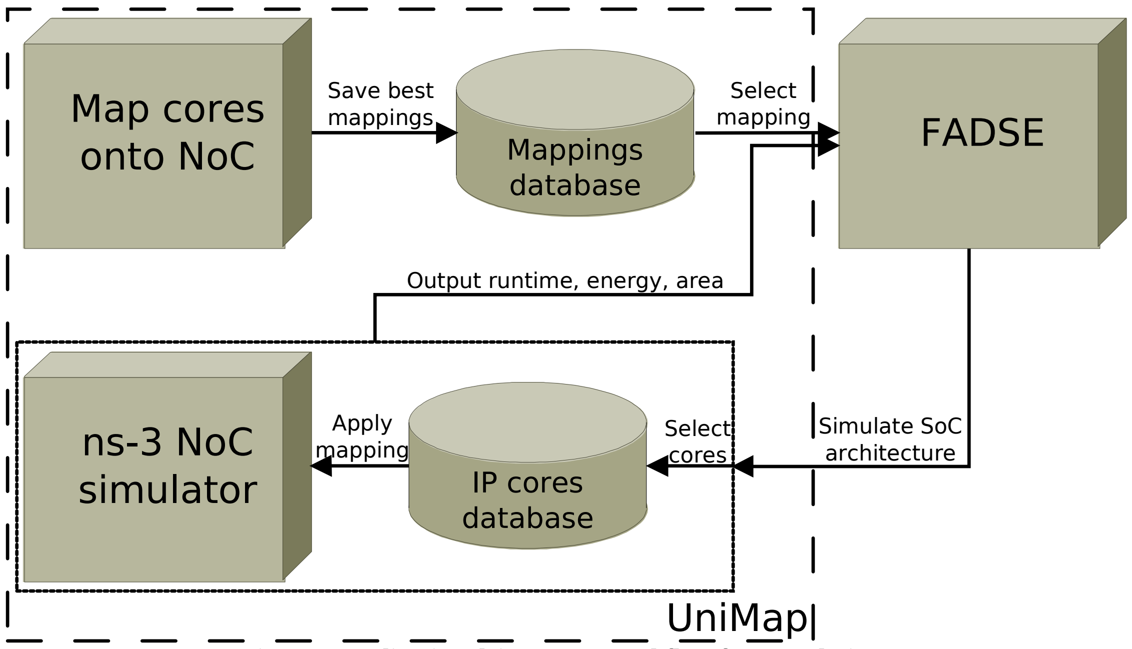

Fig. 100 Application driven DSE workflow for SoC designs

Our DSE workflow starts with mapping applications onto NoC architectures using UniMap's algorithms. The mappings are evaluated by estimating the NoC communication energy with the analytical model from Section 3.1.3.2. The best solutions found are saved into a database.

For each application, FADSE searches for the best SoC design by considering the first ten best mappings[16]. Note that we select these mappings from all best mappings found by all UniMap mapping algorithms: Simulated Annealing, Branch and Bound, Optimized Simulated Annealing and Elitist Genetic Algorithm and Elitist Evolutionary Strategy, with all their variants, evaluated in Chapter 7.

Then we configure FADSE to start a DSE process, driven by a multi-objective algorithm. FADSE evaluates different System-on-Chip architectures. Firstly, it selects the type for each IP core. The given mapping already contains information about what IP core will execute what task. However, FADSE will try with other compatible IP cores as well. Any IP core capable of executing a task is considered compatible with that task. Note that the analytical model used for obtaining the best mappings does not account for IP core types. Secondly, it instantiates a SoC architecture by placing the selected IP cores onto the nodes of a NoC that it configures. Finally, it calls UniMap's ns-3 NoC simulator. We model the tasks' execution using the approach presented in Section 4.2.2. The network communications are created using our network traffic generator (see Section 4.3.9). ns-3 NoC measures application runtime, SoC energy and SoC area. These are the three objectives of our DSE workflow.

We use the E3S [56] IP core library, which provides data about the power consumed by each core while executing a certain task and while idle and the area occupied by every core.

[16] A higher number of best mappings may be used depending on how many resources are available

For our NoC architecture, power and area metrics are measured using ORION 2.0 [96], which is integrated with UniMap's NoC simulator (see section 4.3.10). We work with the Network-on-Chip total power, which includes leakage and dynamic power for routers and links. Similarly, NoC area is the sum of routers and links area.

We measure application runtime by running the application for a specified number of CTG iterations. We determine the number of CTG iterations empirically, so that the simulations run fast enough so that our DSE process ends in a feasible amount of time.

The output of this workflow is a Pareto front with the "best" (near optimal) SoC configurations, for a particular application.

In the next section we give details about how exactly we performed the simulations, on which benchmarks, what architectural parameters we varied and how UniMap and FADSE were configured. We must point out that, during the workflow, the NoC topology is kept unchanged. This is because the topology is basically the single NoC architectural element used by the mapping algorithms. Changing it would lead to inconsistencies, i.e. doing and comparing mappings for different NoCs. Obviously, our workflow may also be applied for different NoC topologies. By doing so we could also determine the most suitable NoC topology. However, this would require adapting our application mapping algorithms for these other NoC topologies. Only then we will be able to obtain the best mappings for other NoC topologies.

8.4 Simulation Methodology

We chose to work with four of the benchmarks presented in Chapter 5: *telecom*, MPEG-4, H.264 (CTG 0) and VOPD (CTG 0). Note that we present in this thesis only some preliminary results. We plan to continue with this research. In the future we will evaluate more applications and we will give more qualitative and quantitative results.

For each of the four benchmarks, we selected the first ten best mappings that we obtained with all our algorithms from UniMap. *telecom* is a 30 cores application, mapped on a 6x5 2D mesh NoC. MPEG-4, H.264 (CTG 0) and VOPD have 12, 16, and respectively 16 IP cores. They are mapped onto 4x3, 4x4 and respectively 4x4 2D meshes.

We chose the number of CTG iterations after performing some preliminary simulations that showed us approximately how much time each simulation takes. We wanted to simulate each benchmark for just several minutes, not more than ten. Thus, we used ten CTG iterations for *telecom*, two for MPEG-4, four for H.264 (CTG 0) and one for VOPD (CTG 0).

For all benchmarks we used the E3S IP core library. Since *telecom* is an E3S benchmark, we know exactly which core types can execute each of its tasks. On average, we have 20 core types for every *telecom* task. The other three benchmarks are not from E3S benchmark suite. Therefore, we do not know what cores can execute their tasks and in what time. However, for each IP core, E3S specifies a generic task for which we have the execution time and power consumption. We considered all tasks from MPEG-4, H.264 and VOPD as generic. Thus, all E3S IP cores can be used to execute every task of these three benchmarks. The E3S IP core library contains a total of 34 cores.

We vary the following Network-on-Chip parameters: network clock frequency, input buffer size, flit size, packet size and routing protocol. The NoC clock frequency varies from 100 MHz to 1 GHz, in steps of 100 MHz. The size of the network input buffers is varied uniformly, from one to ten flits. The flit size is expressed in bytes. It is given by a geometric progression with ratio 2, the initial value 4 and the final value 256. The packet size is measured in number of flits. It varies from two flits, up to ten. Finally, we use only Dimension Order Routing but, we allow for XY or YX routing. The network bandwidth is automatically set, so that one flit can be transmitted in a single NoC clock cycle.

Therefore, the search space determined by all possible Network-on-Chip architectures has a size of *N = 12600*. The search space size given by the IP core types (*C*) is, however, much bigger.

Benchmark	Search space size
telecom	$12600 \cdot 20^{30} \approx 1.35 \cdot 10^{43}$
MPEG-4	$12600 \cdot 34^{12} \approx 3 \cdot 10^{22}$
H.264 (CTG 0)	$12600 \cdot 34^{16} \approx 4 \cdot 10^{28}$
VOPD (CTG 0)	$12600 \cdot 34^{16} \approx 4 \cdot 10^{28}$

Fig. 101 The search space size

Regarding our design space exploration tool, we configured FADSE to run with four multi-objective algorithms: NSGA-II, SPEA2, SMPSO and OMPSO. We decided to stop the algorithms after 50 generations.

We configured NSGA-II as indicated in [116]. Therefore, we used a population of 100 individuals. We employed standard genetic operators: single point crossover, bit flip mutation and binary tournament selection. Crossover probability was set to 90% and mutation probability was set to 3%. A *telecom* chromosome has the highest number of genes, 35, because we vary five NoC parameters and because it is the benchmark using the biggest NoC (30 cores placed onto a 6x5 2D mesh). Usually, the mutation rate for bit flip mutation is *1 / n*, where *n* is the number of genes (1 / 35 ≈ 3%). SPEA2 was configured identically to NSGA-II.

Similarly, SMPSO and OMOPSO used a swarm of 100 particles. The archive size is also 100.

8.5 Experimental Results

We show next some preliminary results obtained with our previously presented application driven design space exploration technique for System-on-Chip architectures. We managed to explore all ten best mappings just for the *telecom* benchmark. For the rest of benchmarks we explored only the first best mapping.

We start with the *telecom* DSE. In the next figure we use the hypervolume metric to show how our four DSE algorithms progress while searching for the best SoC designs for the *telecom* benchmark. We obtained hypervolumes for each DSE algorithm, on every one of the ten *telecom* mappings. Then we computed the average hypervolume.

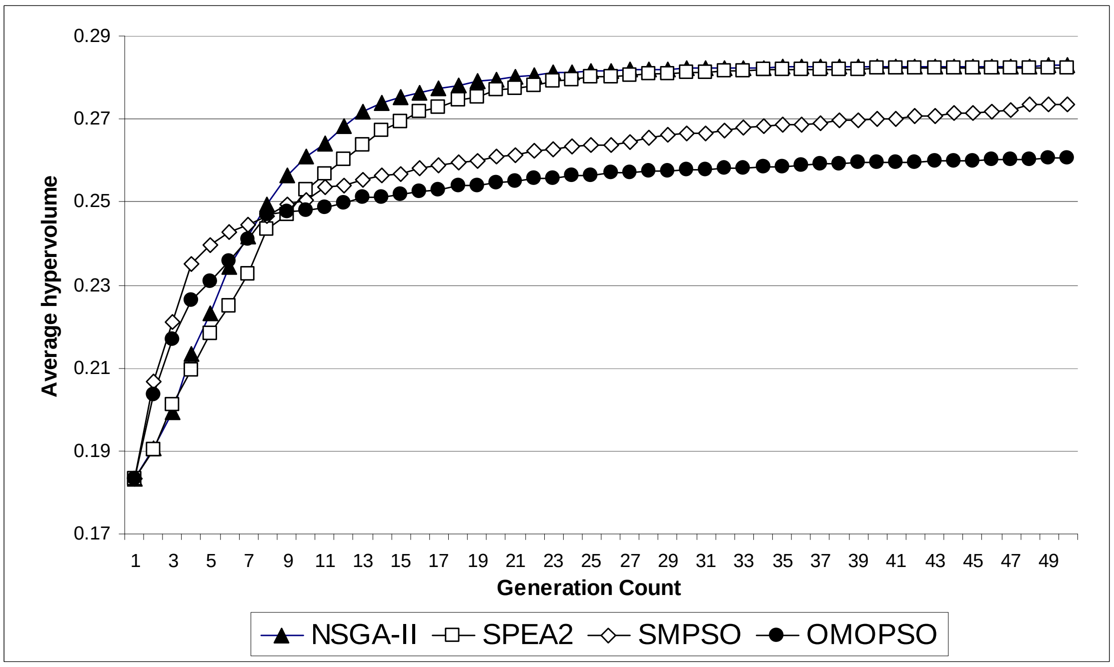

Fig. 102 Average hypervolumes over all ten best telecom mappings

It can be seen that the two genetic algorithms (NSGA-II and SPEA2) obtained the best hypervolumes. NSGA-II has a slightly faster convergence speed than SPEA2. In the last ten generations, both of them saturate; they no longer find significantly better solutions. SMPSO performs better than OMOPSO but, both PSO algorithms perform worse than the genetic algorithms in terms of solution quality (we used the same hypervolume reference point). However, they have the fastest convergence speed. Only after 8-9 generations the genetics recover and surpass the PSO algorithms.

We also compared the four algorithms using the coverage metric (results are omitted due to space constraints). We concluded that SPEA2 has the best overall results. The following figure shows the Pareto front obtained with SPEA2, by combining the Pareto fronts from all ten *telecom* mappings.

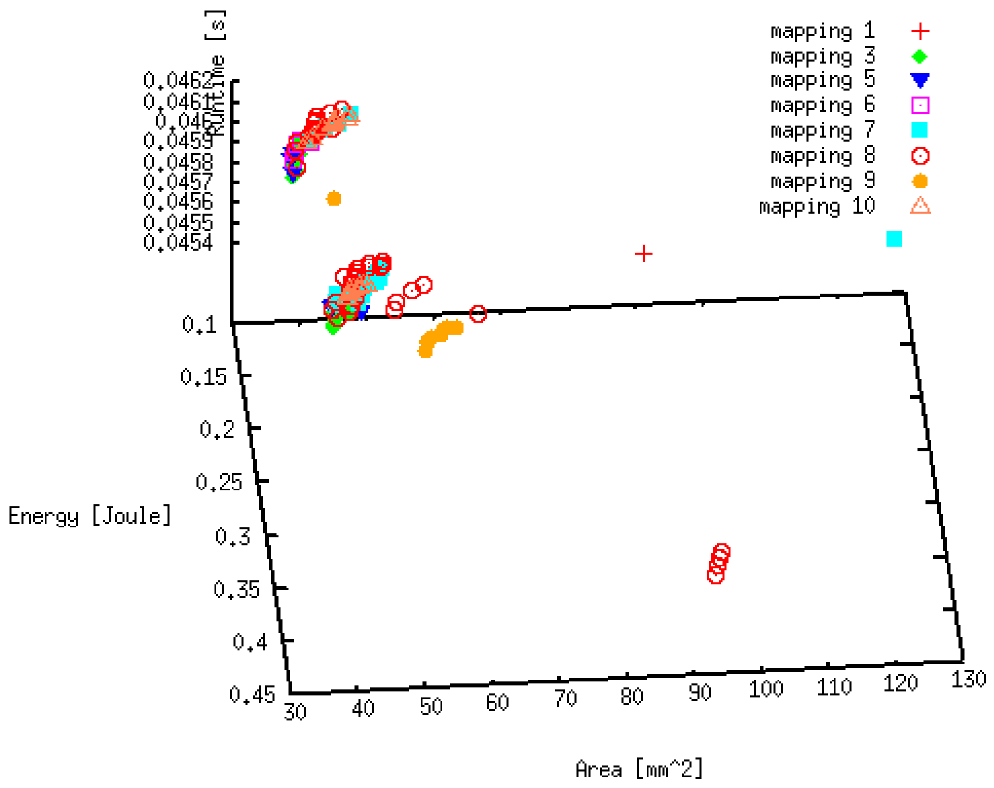

Fig. 103 SPEA2 Pareto front, for telecom

We observe that the Pareto front contains solutions from all eight of the ten best *telecom* mappings. We obtained the best energy consumption with the eight mapping. The smallest area was given by mappings three and five. With exactly the same area, the third mapping has a better energy, while the fifth has a better application runtime. Finally, the lowest application runtime was found on a SoC design corresponding to mapping eight.

It is interesting to see that we did not obtain the best energy with the first best mapping, which analytically gave us the lowest NoC communication energy. This can be due to several facts. Firstly, we analytically estimated only the NoC communication energy. With this approach we compute the entire SoC energy (IP cores energy is also included). Secondly, the analytical model is unable to capture the dynamic network effects (network congestions). Thirdly, FADSE does not obviously perform an exhaustive search. It is possible that we might get better energy results with mapping one than with mapping eight. This shows the need to perform better exploration of the design space. Using domain-knowledge to constrain the search space and applying fuzzy rules are two approaches that could improve the DSE technique [127].

Finally, we combined all the Pareto fronts obtained with all our algorithms, for all ten *telecom* mappings.

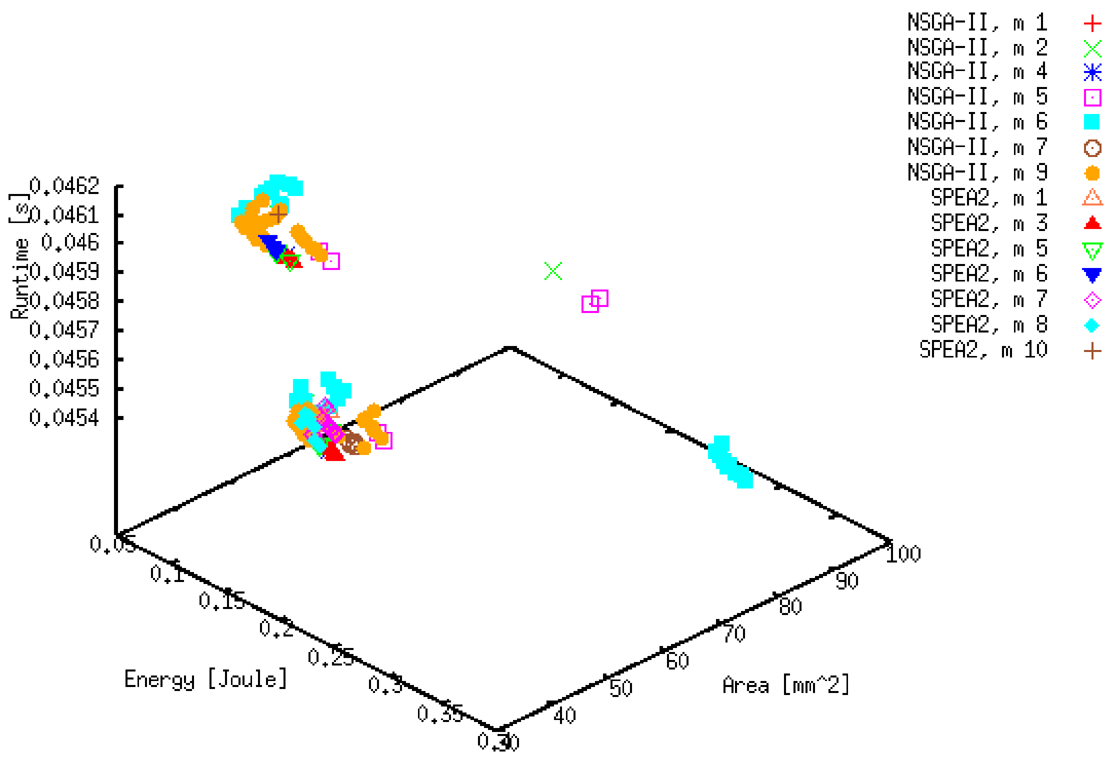

Fig. 104 Combined Pareto front for telecom benchmark

It can be observed that all the solutions found with SMPSO and OMOPSO are dominated by the solutions found with the genetic algorithms. While in terms of SoC area the best solutions are the ones found with SPEA2 (with mappings 3 and 5), in terms of energy and runtime, NSGA-II found, with mapping six, better results than SPEA2 (with mapping eight).

The following table summarizes the best SoC designs found for the *telecom* application. Due to space constraints, we do not show the 30 IP cores selected for every SoC architecture.

Objective	**Algorithm**	**Map ping**	**NoC parameters**					**SoC energy [Joule]**	**SoC area [mm^2]**	**Application runtime [ms]**
			Frequency [MHz]	**Buffer size [flits]**	**Flit size [bytes]**	**Packet size [flits]**	**Routi ng**			
Energy	NSGA-II	6	100	4	4	10	YX	0.09516	50.11	46.1144
Area	SPEA2	5	200	1	4	10	XY	0.15818	37.37	46.1132
Area	SPEA2	3	400	1	4	10	YX	0.16793	37.37	46.1111
Runtime	NSGA-II	6	900	4	32	6	YX	0.34191	81.22	45.4

The lowest energy was obtained (in accordance with our intuition) when the NoC operated at the lowest frequency allowed by our DSE workflow. The SoCs with the smallest area use some of the smallest IP cores. Also, the NoC buffers are only one flit in size. As compared with the best energy and runtime SoC designs, the two area designs use only 25% NoC buffering resources. The two designs with the smallest area essentially differ by the NoC frequency. The faster one uses a NoC that is twice faster. The SoC with the best runtime runs *telecom* with more than half a millisecond than the other three SoCs, which are differentiated in terms of speed by only a few fractions of a microsecond. The best runtime SoC architecture also requires a much faster NoC. It also operates with bigger packets. All these reflect on considerably higher energy and bigger area. Finally, we also observe that routing also influences the architecture's performance. Our best SoC designs for *telecom* use both XY and YX routing protocols.

Now we continue with the MPEG-4 DSE. The following figure presents the hypervolume of each DSE algorithm, for the best MPEG-4 mapping found analytically.

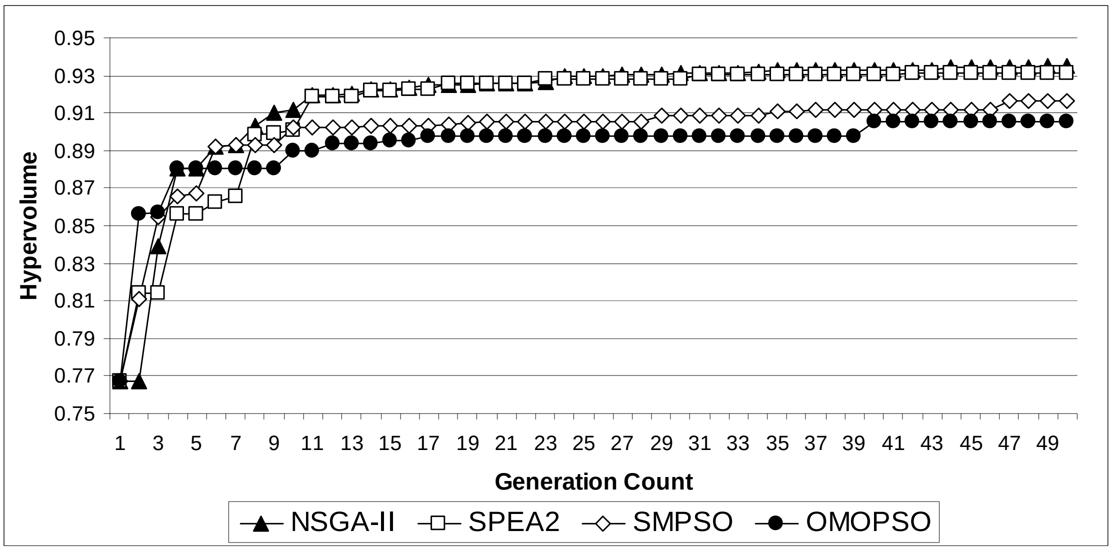

Fig. 105 Hypervolumes for the first best MPEG-4 mapping

The results obtained for *telecom* are consistent with the ones presented here. Again the two genetic algorithms perform better than the particle swarm optimization algorithms. NSGA-II converges faster than SPEA2. In terms of quality of results it seems that NSGA-II is the best. Again, SMPSO performed better than OMOPSO. Like for *telecom*, MPEG-4 results show us that it matters more the class the algorithm belongs to (evolutionary or bio-inspired), rather than the specific implementation.

We computed the coverage, trying to choose the best algorithm from each class. The results are presented in Fig. 106 and Fig. 107.

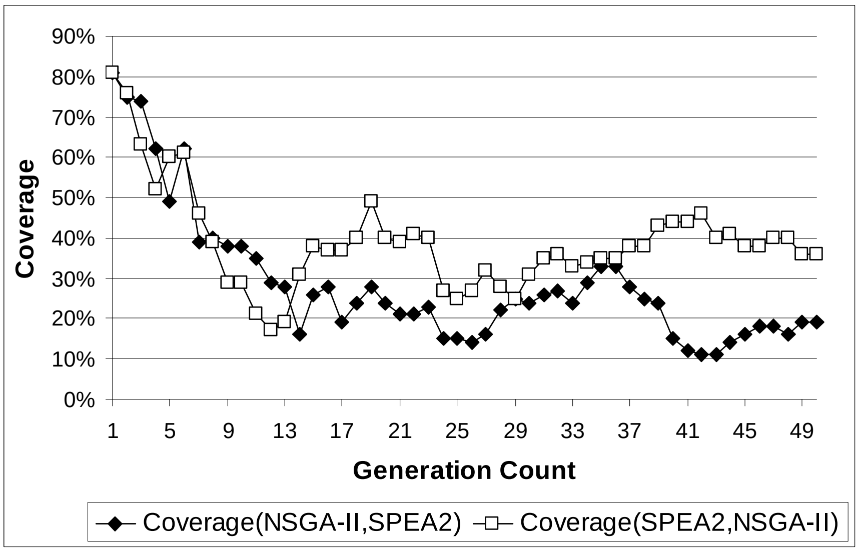

Fig. 106 Coverage comparison between NSGA-II and SPEA2, for MPEG-4

For the first generations no clear distinction can be made between the two algorithms. However, looking at the last generations, we conclude that there are more individuals produced by SPEA2 that dominate the NSGA-II individuals. This contradicts the hypervolume chart where NSGA-II seemed to perform better. We thoroughly analyzed the Pareto fronts obtained by the two genetic algorithms. Some of the solutions discovered by NSGA-II are better than the ones obtained by SPEA2 and some are worse (in accordance with the coverage metric). It is hard to establish the best one because it depends on the requirements of the designer. Still, the results obtained by NSGA-II seemed a little more spread in the objective space.

The same behavior can be observed between OMOPSO and SMPSO. SMPSO performed better from the hypervolume point of view, but here OMOPSO is the best. Again, we analyzed the Pareto fronts approximations and from our point of view SMPSO had better results. We emphasize that this is a subjective appreciation and for other designers the order might be changed.

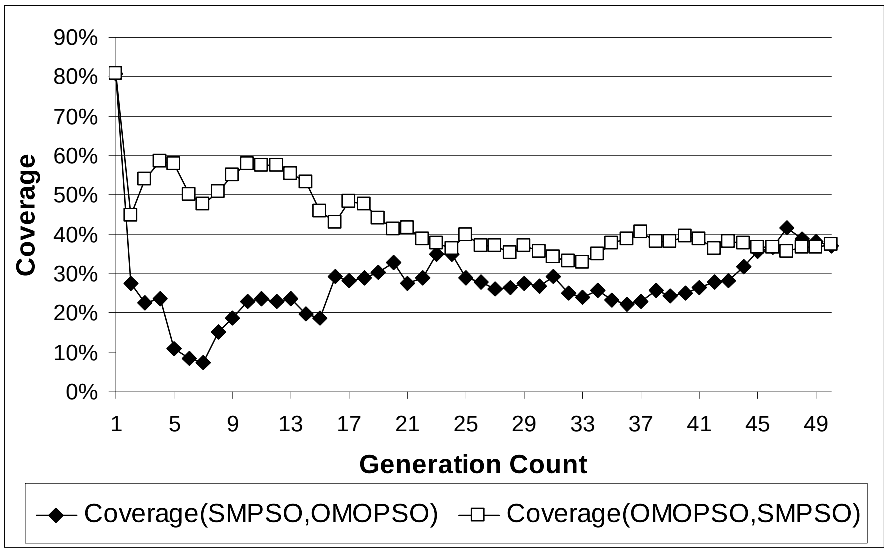

Fig. 107 Coverage comparison between SMPSO and OMOPSO, for MPEG-4

For our last comparison we selected the best algorithms from the coverage point of view: SPEA2 and OMOPSO. In the next figure we present the coverage comparison between the two algorithms. SPEA2 is clearly the best, by dominating almost 100% of the individuals found by OMOSPO. OMOSPO does not dominate almost any individuals obtained by the genetic algorithm. It is interesting to observe that OMOPSO is better for the first generations. This is because of the faster convergence speed of the PSO algorithms.

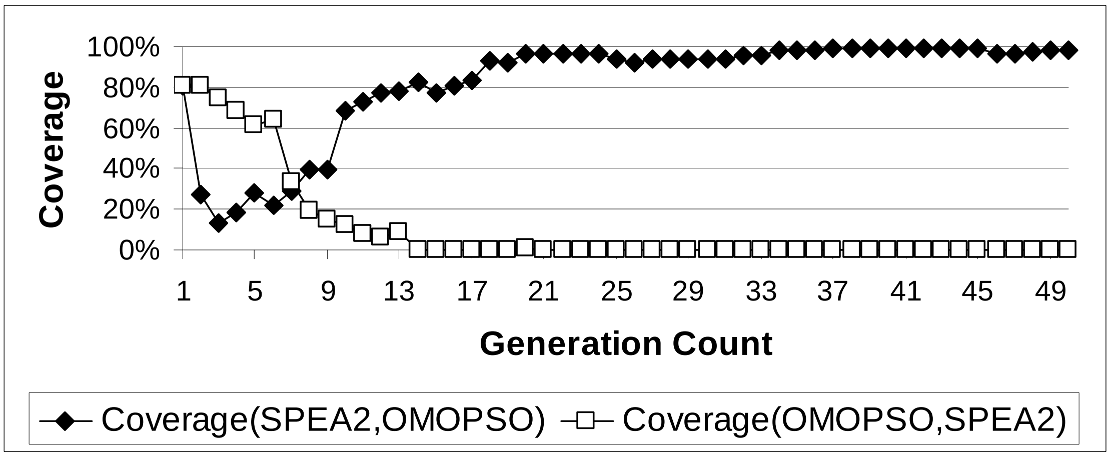

Fig. 108 Coverage comparison between SPEA2 and OMOPSO, for MPEG-4

The following figure presents the most spread Pareto front, which was obtained by the NSGA-II algorithm. Through interpolation we also obtained a surface grid that gives us a better view of the Pareto surface.

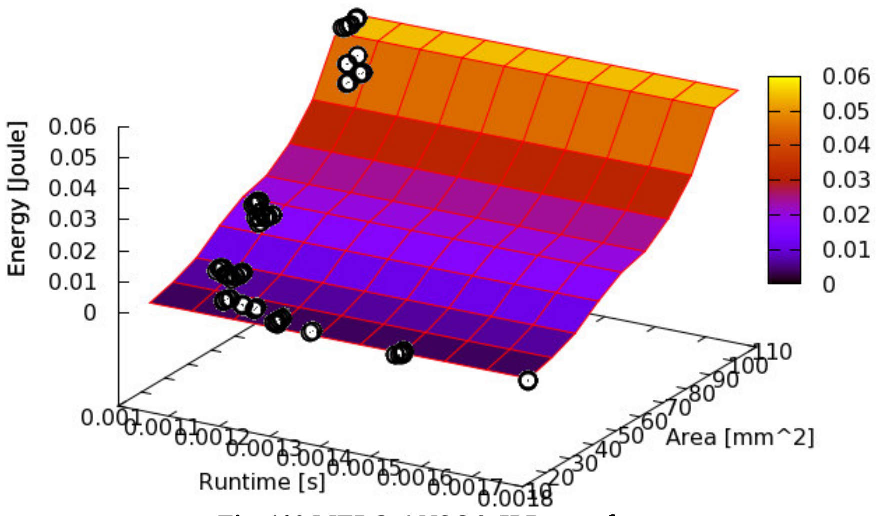

Fig. 109 MEPG-4 NSGA-II Pareto front

As expected, it can be observed that there is no SoC design for the MPEG-4 application that is best for all three objectives. The fastest designs consume more energy and occupy more area. The slowest architectures consume less energy and need less area. In between we have a lot of solutions that are better for energy and worse for area and vice versa.

We conclude this preliminary research by presenting the hypervolumes obtained for the first best analytical mapping of H.264 and VOPD benchmarks.

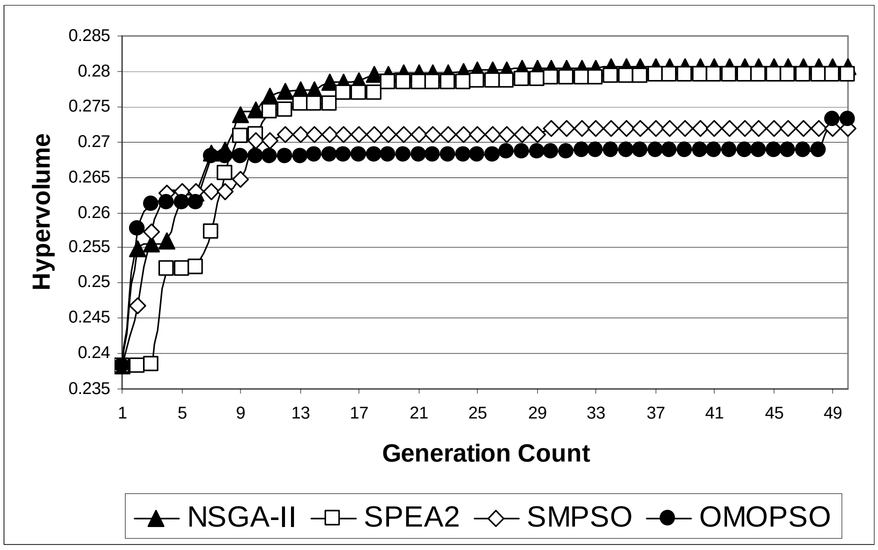

Fig. 110 Hypervolumes for the first best H.264 mapping

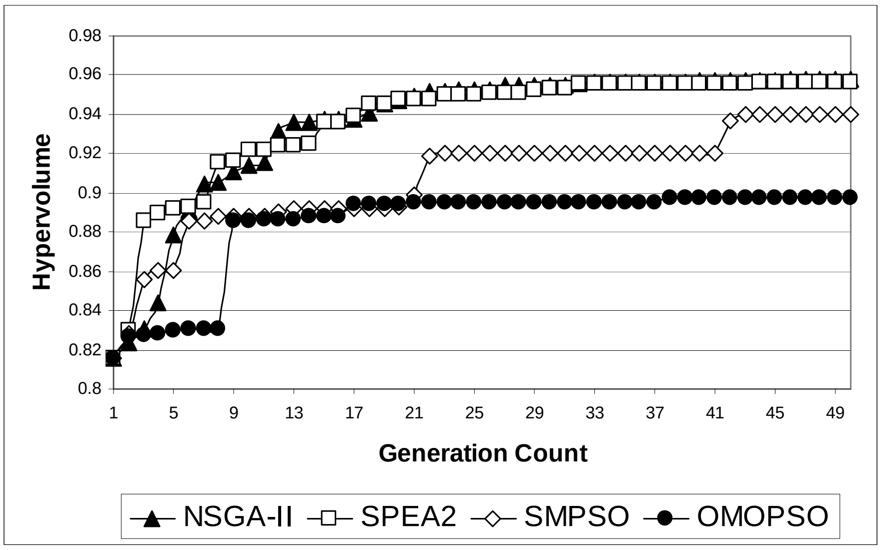

Fig. 111 Hypervolumes for the first best VOPD mapping

These H.264 and VOPD hypervolume results are in correlation with our previous results. Our conclusion is that the genetic algorithms find better solutions than the particle swarm optimization methods. The PSO algorithms manage to converge faster only for the H.264 decoder. For VOPD, SMPSO performs clearly better than OMOPSO. We also observe an unsteady convergence speed for the PSOs. For a large number of generations their evolution is insignificant. Then, they manage to find at least one significantly better individual, which makes their hypervolume grow noticeable.

8.6 Summary

We have proposed an application driven automatic DSE technique for System-on-Chip architectures. Our DSE method involves our developed UniMap and FADSE [127].

The goal was that, for a given application, to automatically determine the best SoC design, with the three objectives: SoC energy, SoC area and application runtime.

In order to speedup the exploration, we started the search from some given NoC mappings of our evaluated applications. These mappings are the best determined analytically with all UniMap's mapping algorithms. Then, using UniMap's Network-on-Chip simulator, FADSE searched for these mappings the most suitable IP cores and NoC architecture so that our three objectives are minimized.

We showed that the best analytical mappings are not necessarily the best ones when using a NoC simulator. This is because the simulator also accounts for the network dynamics. Another reason is that our simulator also models the processing elements.

We found the best energy SoC (SoC_E) for *telecom* to be only 1.5% slower than the fastest SoC (SoC_R) for *telecom*, while the energy is 2.6 times lower. However, its area is 34% bigger than the smallest SoC (SoC_A) we found. SoC_A consumes more than 66% more energy and its runtime is just a few microseconds better (than $SoC_{E)}$. SoC_R occupies the biggest area being 1.17 times bigger than SoC_A and 62% bigger than $SoC_{E.}$

Another conclusion is that the genetic algorithms were clearly more suited for our DSE workflow than the particle swam optimization methods. Still, the PSO algorithms converge faster for most of the benchmarks.

“Difficulties increase the nearer we approach our goal.”

Johann Wolfgang von Goethe

9 Conclusions and Further Work

This work addresses the Network-on-Chip application mapping problem. After we introduced the novel Network-on-Chip paradigm in Chapter 2, we focused on the mapping problem. Chapter 3 presents the problem along with a state of the art on the heuristic algorithms used to address it. In Chapter 4 we show our developed unified framework for the evaluation and optimization of Network-on-Chip application mapping algorithms. In Chapter 5 we presented the benchmarks used in our research, in this emerging NoC research field that still lacks a standard benchmarking methodology. With UniMap, we evaluated and optimized a simulated annealing algorithm using a domain-knowledge approach (Chapter 6). We also evaluated and optimized evolutionary algorithms by proposing problem aware genetic operators (Chapter 7). Finally, in Chapter 8, we proposed and used a design space exploration workflow for an application driven automatic design space exploration for Systems-on-Chip. Our algorithms’ evaluations were performed using both analytical models and simulators. We considered single and multi-objective approaches.
More precisely, this thesis makes the following contributions:

- An introduction to Network-on-Chip architectures with an emphasis on the most common network topologies and routing protocols used in this research field;
- Taxonomy for the classification of Network-on-Chip application mapping algorithms;
- State of the art regarding algorithms for Network-on-Chip application mapping;
- UniMap: a developed unified framework for the evaluation and optimization of NoC application mapping algorithms;
- UniMap runs on High Performance Computing Systems using job schedulers to automatically and optimally distribute simulations;
- Common model based on XML schemas for representing real applications and networks;
- UniMap integrates state of the art NoC application mapping algorithms like Simulated Annealing and Branch and Bound;
- UniMap integrates jMetal, a library with single objective and multi-objective state of the art evolutionary algorithms, which can be used as application mapping algorithms;
- ns-3 NoC, our developed Network-on-Chip simulator, with two router architectures, three routing protocols, three switching mechanisms and k-ary d-cube topologies;
- Network traffic generator based on communication patterns of real applications, described through Communication Task Graphs and Application Characterization Graphs;
- ns-3 NoC integrates ORION 2.0, a state of the art tool for Network-on-Chip power consumption and area estimation;
- Using ns-3 NoC, we showed that the Irvine architecture helps at decreasing the network congestion. The network is significantly less congested when data flits are transmitted faster than head flits;

- With ns-3 NoC, we showed how increasing the network buffers' size improves the NoC's average packet latency;
- Using ns-3 NoC, we evaluated different network topologies: 2D mesh, 2D torus, 3D mesh, 3D torus and hypercube. We concluded that topologies like tori and hypercube can give better NoC performance than meshes can;
- UniMap integrates the E3S benchmark suite and some of the most used CTGs and APCGs available in literature. Because NoC benchmarking is still work in progress, we effectively created our own benchmark suite;
- We propose and use for Network-on-Chip benchmarking two communication patterns taken from a H.264 decoder system available in the research community;
- Using domain-knowledge, we developed an Optimized Simulated Annealing (OSA) algorithm. It performs a dynamic and implicit core clustering and limits the number of iterations per annealing temperature based on the given application and network.
- We showed that Simulated Annealing can be feasible for NoC application mapping when domain-knowledge is used. OSA is approximately 99% faster than a generic Simulated Annealing algorithm, without losing the solution quality;
- The results obtained with OSA showed that Simulated Annealing is feasible for NoC 2D meshes larger than 10x10. Previous research stated the contrary;
- OSA is comparable to Branch and Bound in terms of memory consumption and speed. It mapped 97 cores on a 10x10 2D mesh in a time slower by only 3% than the time required by Branch and Bound;
- As the problem size increases, OSA gives significantly better solutions than Branch and Bound. The mappings found with Branch and Bound were with more than 70% worse than OSA's mappings when working with more than 64 IP cores;
- We showed Branch and Bound's limitations. This algorithm was unable to map an application with 215 cores, onto a 15x15 NoC, because more than 98% of the search space was pruned;
- We developed an Elitist energy- and performance-aware Genetic Algorithm (EGA). EGA is integrated in jMetal;
- We extended jMetal with the Position Based crossover;
- We evaluated EGA and an Elitist Evolutionary Strategy (EES) using different genetic operators (four crossovers, two mutations => 12 algorithm variants);
- We concluded that evolutionary algorithms are superior to algorithms like OSA, for NoCs with tens, hundreds of nodes. We found that, for the big benchmarks, all the best solutions were given by evolutionary algorithms (none by OSA);
- We proposed a meta-heuristic algorithm consisting of an evolutionary algorithm that uses as mutation operator a state of the art application mapping algorithm;
- EGA and EES work better with OSA mutation than with swap mutation. OSA integrated successfully into the Evolutionary Algorithms;
- We designed two problem specific crossover operators: NoC Position Based and Mapping Similarity. NoC Position Based crossover improves the standard Position Based crossover for our problem. Mapping Similarity crossover exchanges information between the parent individuals. It does not simply work as a mutation operator, like the other state of the art NoC application mapping crossover operators do;

- With NoC Position Based crossover, EGA had the best solution percentage on the big benchmarks;
- We found Mapping Similarity to be the crossover operator that contributes the most at obtaining a good mapping. It performed best at 50% - 60% mutation probability. The rest of crossovers required higher mutation rates;
- We found EES to perform better than EGA. Although we managed to improve the genetic algorithm through our crossover operators, using an algorithm that works only with (context-aware) mutation proved to be better. Finding a suitable context-aware crossover for NoC application mapping is more difficult than finding an efficient context-aware mutation;
- EES with OSA mutation was the algorithm that managed to converge the fastest;
- Using two state of the art multi-objective algorithms (NSGA-II and SPEA2) with our genetic operators, we evaluated (with analytic models) the mappings in terms of NoC communication energy and NoC thermal balance. The two objectives are contradictory and, as such, our developed operators did not lead to the best performance. However, we did find the best solutions, in terms of energy, with OSA mutation. A suitable crossover operator for the NoC application mapping problem is even more difficult to find if we consider multi-objective optimization;
- UniMap connects with the Framework for Automatic Design Space Exploration;
- We proposed an application driven automatic Design Space Exploration technique for System-on-Chip architectures. The goal is that, for a given application, to automatically determine the best System-on-Chip design, with the following objectives: SoC energy, SoC area and application runtime;
- Using our developed ns-3 NoC simulator and FADSE, we explored the NoC architectural space for different real applications;
- We showed that the best analytical mappings are not necessarily the best ones when using a NoC simulator;
- The genetic algorithms (NSGA-II and SPEA2) were clearly more suited for our design space exploration workflow than the particle swam optimization methods (SMPSO and OMOPSO). Still, the PSO algorithms converged faster.

As future work, we intend to improve UniMap. We are interested in extracting communication patterns from parallel applications. The first step will be to integrate CETA tool (see Section 4.2.1.2.3). This will allow us to obtain Communication Task Graphs from shared memory parallel programs. The second step will be to similarly use an MPI library that allows intercepting the communications from message passing parallel applications.

Another direction for extending our unified framework is to implement other state of the art Network-on-Chip application mapping algorithms. For example, the comparisons between OSA and Cluster Simulated Annealing (see Section 6.1) runtimes are very likely to be unfair. This can be due to several reasons: (1) OSA is written in Java but, we do not know yet how CSA is implemented, (2) OSA is energy aware and uses the cost function from [40], while CSA is bandwidth and latency constrained, using the cost function from [61] and (3) CSA does not specify the number of generations per temperature level.

Also, we consider further improving our developed NoC simulator. Improving the router architecture with virtual channels and allocators is an example. This will bring our router implementation closer to real router designs.

Regarding our developed crossover operators (see Section 7.4) they are suitable only for the communication energy objective. They must be adapted to work in a multi-objective case. Even OSA mutation was designed only for energy minimization. Therefore, evaluation and optimization of such algorithms, in a multi-objective context will be more difficult. Using standard crossover and mutation operators simplifies the problem a lot but, such operators are not aware of the problem.

We also plan to continue our research regarding application driven automatic design space exploration for System-on-Chip architectures (see Chapter 8). The presented results are still preliminary. We intend doing more simulations so that we identify the best SoC designs for applications other than *telecom*, too. We also intend to do a more accurate modeling of our SoC designs by increasing the accuracy with which we simulate the IP cores and by varying the NoC topology as well. As for the design space explorer, we intend to use more domain-knowledge so that we can constrain and better explore the huge architectural space. We believe our approach can be extended for performing automatic design space exploration for High Performance Computing systems.

Finally, we refer to a research niche that we identified during this PhD thesis but, unfortunately we have not had enough time to exploit it, yet. We believe that Network-on-Chip application mapping problem can be addressed using graph theory. More precisely, we refer to **graph isomorphism**, which is the problem of verifying if two graphs are actually the same. Two graphs $A = (V_A, E_A)$ and $N = (V_N, E_N)$ are isomorphic if and only if there is a *bijective* mapping $M : V_A \rightarrow V_N$, between the graph nodes, such that the following equivalence is true: $\forall e_1, e_2 \in V_A : (e_1, e_2) \in E_A \Leftrightarrow (M(e_1), M(e_2)) \in E_N$. This means a unique mapping between the corresponding edges of the two graphs is required. For weighted graphs, the condition can be extended to include the weights as well. **Subgraph isomorphism** requires the mapping M to be only *injective*. **Graph monomorphism** is a weaker type of subgraph isomorphism. The equivalence relation must be just an implication ($\forall e_1, e_2 \in V_A : (e_1, e_2) \in E_A \Rightarrow (M(e_1), M(e_2)) \in E_N$). Considering the above definitions and that the two graphs (A and N) are an Application Characterization Graph (APCG) and, respectively, a NoC topology graph, the Network-on-Chip application mapping problem can be viewed as a graph monomorphism problem. Indeed, it is mentioned in [136] that the quadratic assignment problem can be formulated as a graph monomorphism problem. Currently, there is no known polynomial-time algorithm for the monomorphism problem [137]. However, special graph types, like planar graphs, can theoretically be solved in a linear time [138]. Using the Boyer-Myrvold algorithm [139], we tested for planarity all the APCGs used in this work (see section 5.3). All of them proved to be planar graphs. We also integrated in UniMap the VF2 [140] graph matching algorithm and used it to determine if an isomorphism exists between any APCG and its corresponding NoC topology graph. We found none but, this is understandable because we should search for monomorphisms, not for isomorphisms. We found little NoC research using this idea. Graph isomorphism is used in [141] to identify the isomorphically unique NoC topology graphs. VF2 algorithm is used in [142] to perform subgraph isomorphism in order to decompose an APCG into a set of predefined communication pattern graphs. We believe approaching the NoC application mapping problem as a graph monomorphism problem is worth researching.

10 Glossary

This section presents an alphabetical list of technical terms used in this work, mainly from the field of Networks-on-Chip architectures. The definitions of these technical terms are mainly taken from [24], [1], [25], [43].

Adaptive routing	=	An adaptive routing algorithm uses information about the network's state in making routing decisions. This information may include the status of a node or link, the length of queues for network resources, and historical channel load information. Such an algorithm considers multiple paths and chooses the one which is most suitable. See also *routing, deterministic routing* and *oblivious routing*.
Bandwidth	=	The maximum amount of information that can be transmitted along a channel, in a unit of time. It is measured in bits/s. See also *channel*.
Bisection bandwidth	=	The sum of the bandwidths of the minimum set of channels that, if removed, partition the network into two equal unconnected set of nodes. See also *bandwidth, channel*.
Broadcast	=	A broadcast is a multicast in which a packet is sent to all destinations. See also *multicast*.
Channel	=	Consists of a transmitter, link and receiver. It allows the (digital) information to flow between the network interfaces attached to it. See also *link*.
Circuit switching	=	Switching mechanism by which the path from the source to the destination is established and reserved until the message is transferred over the circuit. See also *packet switching*.
(Network) collision	=	A network collision occurs when more than one device attempts to send a packet on a network segment at the same time. Collisions are resolved by discarding and resending (one at a time) the competing packets. See also *packet*.
Dateline	=	A conceptual line across a channel of a ring network (or within a single dimension of a torus).
Deadlock	=	A situation when a packet waits for an event that cannot occur; for example when no message can advance toward its destination because the queues of the message system are full and each is waiting for another to make resources available. See also *livelock*.

Deadlock-free deterministic routing	=	A deterministic routing algorithm that guarantees that a deadlock will never arise. See also *deadlock*, *routing* and *deterministic routing*.
(Node) degree	=	The number of channels entering and leaving each node. See also *channel*.
Deterministic routing	=	A routing algorithm is deterministic (or non-adaptive) if the route taken by a message is determined solely by its source and destination, and not by other traffic in the network. A deterministic routing algorithm always chooses the same path between two nodes, even if there are multiple possible paths. See also *routing*, *oblivious routing* and *adaptive routing*.
(Network) diameter	=	The maximum number of links that must be traversed to send a message to any node along a shortest path (the length of the maximum shortest path between any two nodes).
Direct network	=	Network that has each node connected to each of the other nodes. See also *indirect network*.
Design Space Exploration (DSE)	=	Design Space Exploration (DSE) is the process of searching through the possible hardware or software design points to find an optimal design. DSE can both be structural (different components) and parametric (fixed set of components of which the parameters are tuned).
Fault	=	A requirement, design, or implementation flaw or deviation from a desired or intended (logical) state. Also known as *defect*, although defect more often refers to the physical state.
Fault tolerance	=	The ability of a network to detect, isolate and recover from faulty resources. See also *fault*.
Flit	=	A **fl**ow control un**it** is the minimum unit of information that can be transferred across a link and either accepted or rejected. It may be as small as a phit or as large as a packet or message. See also *phit*, *packet* and *message*.
Flow	=	A flow is a sequence packets traveling between a single source-destination pair and is the unit at which quality of service is provided. It is possible for a source or destination to support multiple flows concurrently. See also *packet*, *quality-of-service (QoS)*.
Flow control mechanism	=	Determines when the message, or portions of it, moves along its route. Note that is common in literature [3] to say that flow control equals switching. See also *switching mechanism*.

Hard error	=	A permanent, unrecoverable error. See also *soft error*.
Hard real-time	=	A system wherein not producing the result of an operation before its deadline expires is equivalent to producing the wrong result. See also *soft real-time*.
Header	=	The front of the packet. It usually contains the routing and control information so that the switches and network interface can determine what to do with the packet as it arrives. See also *packet* and *trailer*.
Hot-spot	=	A hot-spot resource is one whose demand is significantly greater than other, similar resources. For example, a particular destination terminal becomes a hot-spot in a shared memory multicomputer when many processors are simultaneously reading from the same memory location (for example, a shared lock or data structure).
Indirect network	=	Network that has nodes connected only to a specific subset nodes, which form the edges of the network. See also *direct network*.
Jitter	=	The maximum difference in the latency between two packets within a flow. Low jitter is often a requirement for video streams or other real time data for which the regularity of data arrival is important. The jitter times the bandwidth of a flow gives a lower bound on the size of buffer required. See also *packet*, *flow* and *latency*.
Latency	=	The time required to deliver a unit of data (usually a packet or message) through the network, measured as the elapsed time between the injection of the first bit at the source to the ejection of the last bit at the destination. See also *packet* and *message*.
Link	=	A bundle of wires or fibers that carries an analog signal.
Livelock	=	A situation when the routing of a packet never leads to its destination (can only occur with adaptive non-minimal routing). See also *deadlock*.
Load balance	=	The measure of how uniformly resources are being utilized in a network. A network is load-balanced if all the (expensive) resources tend to saturate at the same offered traffic.
Message	=	A message is the logical unit of data transfer provided by the network interfaces. Because messages do not always have a bounded length, they are often broken into smaller packets for handling within the network. See also *packet*.

Minimal routing	=	Routing algorithm that considers only the shortest path routes from source to destination. See also *routing* and *non-minimal routing*.
Multiprocessor System-on-Chip (MPSoC)	=	A system-on-chip that uses multiple, often heterogeneous, processors. See also *SoC*.
Multicast	=	A multicast packet can be sent to multiple destinations. See also *packet* and *unicast*.
Network-on-Chip (NoC)	=	A Network-on-Chip (NoC) consists of a number of interconnected heterogeneous devices (e.g. general or special purpose processors, embedded memories, application specific components, mixed-signal I/O cores) where communication is achieved by sending packets over a scalable interconnection network. No global wiring is used by a Network-on-Chip. Wiring resources are shared by the communicating devices.
Non-minimal routing	=	Routing algorithm that considers all the possible paths from source to destination. See also *routing* and *minimal routing*.
Oblivious routing	=	It includes deterministic routing. The route taken by a message is determined solely by its source and destination but, the same path is not chosen always. For example, a random algorithm that uniformly distributes traffic across all of the paths is an oblivious algorithm. See also *routing*, *deterministic routing* and *adaptive routing*.
Packet	=	A self-delimiting sequence of digital symbols that logically consists of three parts: a header, a payload and a trailer. See also *header*, *payload* and *trailer*.
Packet switching	=	Switching mechanism by which the message is broken into a sequence of packets. Packets are individually routed from the source to the destination. See also *circuit switching*.
Payload	=	The part of the packet containing the data transmitted across the network. See also *packet*.
Phit	=	A **ph**ysical un**it** is the minimum size datagram that can be transmitted in one link transaction. See also *flit*.
Quality-of-Service (QoS)	=	The bandwidth, latency, and/or jitter received by a particular flow or class of traffic. A QoS policy differentiates between flows and provides services to those flows based on a contract that guarantees the QoS provided to each flow, provided that the flow complies with restrictions on volume and burstiness of traffic.

Real-time	=	A real-time system is a system whereby the correctness or quality of the output not only depends on the produced value, but also on when this value becomes available. See also *hard real-time* and *soft real-time*.
Reliability	=	The probability that a network is working at a given point in time.
Router	=	Network component which drives the information though the network.
Routing algorithm	=	The routing algorithm of a network determines which of the possible paths, from source to destination, are used as routes and which route is taken by each particular packet.
Saturation	=	A resource is in saturation when the demands being placed on it are beyond its capacity for servicing those demands. For example, a channel becomes saturated when the amount of data that wants to be routed over the channel exceeds its bandwidth. See also *bandwidth*.
Self-adaptive system	=	A self-adaptive system contains control logic to let the system modify its execution based on the input and environment without requiring external intervention.
Soft error	=	A soft error is a transient error. Hardware continues to work correctly after the soft error occurred, but the data may be corrupted permanently. See also *hard error*.
Soft real-time	=	In a soft real-time system the result of a calculation that becomes available only after its deadline has passed will at worst cause graceful degradation, but otherwise the system will keep functioning correctly. See also *hard real-time*.
Store-And-Forward (SAF) switching	=	Packet switching mechanism where the entire packet is received by a switch before it is forwarded to the next link. See also *packet switching*.
Switch	=	Network component that provides the means to route information from the source node to the destination node.
Switching mechanism	=	Determines how and when the data in a message traverses its route. See also *circuit switching*, *packet switching* and *flow control mechanism*.
System-on-Chip (SoC)	=	A System-on-Chip is a system whereby all components of the entire computing system have been integrated on a single integrated circuit.
Throughput	=	The amount of traffic (in bits/s) delivered to the destination terminals of the network.

Topology	=	The physical interconnection structure of the network. It specifies the connection pattern of the network’s nodes.
Traffic	=	The sequence of injection times and destinations for the packets being offered to the network. This sequence is often modeled by a static traffic pattern that defines the probability a packet travels between a particular source-destination pair and an arrival process.
Trailer	=	The end of the packet. It typically contains error-checking code. See also *packet* and *header*.
Unicast	=	A unicast packet has a single destination terminal (as opposed to multicast). See also *multicast*.
Virtual Cut-Through (VCT) switching	=	Packet switching mechanism which does not buffer data at the output. It allows data to cut through to the input of the next router, before the whole packet was received at the current router. It can block entire routing paths in the network when a message gets blocked somewhere in an intermediate node. In such a case, VCT switching falls back to SAF switching. See also *store-and-forward* switching and *wormhole switching*.
Virtual channel	=	A group of multiple buffers associated to the same physical channel.
Virtualization	=	Basic technique that separates workloads from the physical hardware. It allows for running legacy software on new hardware, for dynamically adapting applications to changing hardware resources, and for isolating software domains (to do dedicated resource provisioning, or for security).
Worst-Case Execution Time (WCET)	=	The Worst-Case Execution Time (WCET) is the maximal execution time necessary for a part of a program to perform its task assuming the worst possible circumstances.
Wormhole switching	=	Packet switching mechanism which, compared to virtual cut-through, buffers the flits in the nodes where they currently are. See also *virtual cut-through switching* and *store-and-forward switching*.

11 References

[1] M. Duranton et al., “The HiPEAC Vision,” *HiPEAC Roadmap*, 2010. [Online]. Available: http://www.hipeac.net/system/files/LR_3910_hipeac_roadmap-2010-v3.pdf.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*, 4th ed. Morgan Kaufmann, 2006.

[3] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, “Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 1, pp. 3-21, 2009.

[4] G. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114-117, Apr. 1965.

[5] G. E. Moore, “Excerpts from a conversation with Gordon Moore: Moore’s Law, 2005,” *URL ftp://download. intel. com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore. pdf.*

[6] L. Vinţan N., *Arhitecturi de procesoare cu paralelism la nivelul instrucțiunilor*. Editura Academiei Române, Bucureşti, 2000.

[7] L. Vinţan N., *Prediction Techniques in Advanced Computing Architectures*. Matrix Rom Publishing House, Bucharest, 2007.

[8] A. Florea and L. Vinţan N., *Simularea şi optimizarea arhitecturilor de calcul în aplicaţii practice*. Editura Matrix Rom, Bucureşti, 2003.

[9] L. Vinţan N., “Direcţii de cercetare în domeniul sistemelor multicore,” *Revista Română de Informatică și Automatică, ICI București*, vol. 19, no. 3, 2009.

[10] **C. Radu**, H. Calborean, A. Crapciu, A. Gellert, and A. Florea, “An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture,” in *The 6th EUROSIM Congress on Modeling and Simulation*, 2007, p. 58.

[11] A. Florea, **C. Radu**, H. Calborean, A. Crapciu, A. Gellert, and L. Vintan, “Designing an Advanced Simulator for Unbiased Branches Prediction,” *Proceedings of 9th International Symposium on Automatic Control and Computer Science*, 2007.

[12] A. Florea, **C. Radu**, H. Calborean, A. Crapciu, A. Gellert, and L. Vinţan, “Understanding and Predicting Unbiased Branches in General-Purpose Applications,” *Buletinul Institutului Politehnic Iasi, Tome LIII (LVII), fasc. 1-4, Section IV, Automation Control and Computer Science Section*, pp. 97-112, 2007.

[13] **C. Radu**, “Implementing a multicore Shared Memory Architecture using Transaction Level Modelling with UNISIM,” Diploma project (Bachelor), “Lucian Blaga” University of Sibiu, Romania (in Romanian, supervisor Professor Lucian Vintan, PhD), Sibiu, Romania, 2008.

[14] **C. Radu**, H. Calborean, A. Florea, A. Gellert, and L. Vintan, “Exploring Some Multicore Research Opportunities. A First Attempt.,” in *Advanced Computer Architecture and Compilation for Embedded Systems*, Terrassa (Barcelona), Spain, 2009.

[15] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of Network-on-chip,” *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[16] K. Asanovic et al., “The landscape of parallel computing research: A view from berkeley,” Citeseer, 2006.

[17] P. Guerrier and A. Greiner, “A generic architecture for on-chip packet-switched interconnections,” *Proceedings of the conference on Design, automation and test in Europe*, pp. 250–256, 2000.

[18] A. Hemani et al., “Network on chip: An architecture for billion transistor era,” in *Proceeding of the IEEE NorChip Conference*, 2000, pp. 166–173.

[19] W. J. Dally and B. Towles, “Route packets, not wires: on-chip inteconnection networks,” in *Proceedings of the 38th annual Design Automation Conference*, Las Vegas, Nevada, United States, 2001, pp. 684-689.

[20] D. Wingard, “Micronetwork-based integration for SOCs: 673,” *Proceedings of the 38th annual Design Automation Conference*, p. 677–, 2001.

[21] E. Rijpkema, K. Goossens, and P. Wielage, “A Router Architecture for Networks on Silicon,” *IN PROCEEDINGS OF PROGRESS 2001, 2ND WORKSHOP ON EMBEDDED SYSTEMS*, p. 181--188, 2001.

[22] S. Kumar et al., “A network on chip architecture and design methodology,” in *isvlsi*, 2002, p. 0117.

[23] G. de Micheli and L. Benini, “Networks on Chip: A New Paradigm for Systems on Chip Design,” *Proceedings of the conference on Design, automation and test in Europe*, p. 418–, 2002.

[24] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. Morgan Kaufmann, 1998.

[25] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, 1st ed. Morgan Kaufmann, 2004.

[26] J. Duato, S. Yalamanchili, and L. M. Ni, *Interconnection Networks: An Engineering Approach*, 1st ed. Institute of Electrical & Electronics Enginee, 1997.

[27] J. P. Bowen, “Hypercubes,” *[[Practical Computing magazine|Practical Computing]]*, vol. 5, no. 4, pp. 97–99, Apr. 1982.

[28] J. Balfour and W. J. Dally, “Design tradeoffs for tiled CMP on-chip networks,” in *Proceedings of the 20th annual international conference on Supercomputing*, Cairns, Queensland, Australia, 2006, pp. 187-198.

[29] J. Kim, W. J. Dally, and D. Abts, “Flattened butterfly: a cost-efficient topology for high-radix networks,” in *Proceedings of the 34th annual international symposium on Computer architecture*, San Diego, California, USA, 2007, pp. 126-137.

[30] C. J. Glass and L. M. Ni, “The turn model for adaptive routing,” *J. ACM*, vol. 41, no. 5, pp. 874-902, 1994.

[31] Ge-Ming Chiu, “The odd-even turn model for adaptive routing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 7, pp. 729-738, Jul. 2000.

[32] A. A. Chien and J. H. Kim, “Planar-adaptive routing: low-cost adaptive networks for multiprocessors,” *J. ACM*, vol. 42, no. 1, pp. 91-123, 1995.

[33] E. Salminen, A. Kulmala, and T. D. Hamalainen, “Survey of Network-on-chip Proposals,” *OCP-IP*, Mar-2008. [Online]. Available:

http://ocpip.org/uploads/documents/OCP-IP_Survey_of_NoC_Proposals_White_Paper_April_2008.pdf.

[34] J. Hu and R. Marculescu, “DyAD: smart routing for networks-on-chip,” in *Proceedings of the 41st annual Design Automation Conference*, San Diego, CA, USA, 2004, pp. 260-263.

[35] “The Odd-Even Turn Model for Adaptive Routing.” .

[36] A. Agarwal, C. Iskander, and R. Shankar, “Survey of Network on Chip (NoC) Architectures & Contributions,” *Journal of Engineering, Computing and Architecture*, vol. 3, no. 1, 2009.

[37] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, and M. Pedram, “An Empirical Investigation of Mesh and Torus NoC Topologies Under Different Routing Algorithms and Traffic Models,” in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2007, pp. 19-26.

[38] H. Wang, L.-S. Peh, and S. Malik, “A Technology-Aware and Energy-Oriented Topology Exploration for On-Chip Networks,” in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, 2005, pp. 1238-1243.

[39] L. Bononi, N. Concer, M. Grammatikakis, M. Coppola, and R. Locatelli, “NoC Topologies Exploration based on Mapping and Simulation Models,” in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2007, pp. 543-546.

[40] J. Hu and R. Marculescu, “Energy-aware mapping for tile-based NoC architectures under performance constraints,” in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, Kitakyushu, Japan, 2003, pp. 233-239.

[41] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co. New York, NY, USA, 1979.

[42] I. Walter, I. Cidon, A. Kolodny, and D. Sigalov, “The era of many-modules SoC: revisiting the NoC mapping problem,” in *2nd International Workshop on Network on Chip Architectures, 2009. NoCArc 2009*, 2009, pp. 43-48.

[43] U. Y. Ogras, J. Hu, and R. Marculescu, “Key research problems in NoC design: a holistic perspective,” in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Jersey City, NJ, USA, 2005, pp. 69-74.

[44] R. P. Dick and N. K. Jha, “MOCSYN: multiobjective core-based single-chip system synthesis,” in *Proceedings of the conference on Design, automation and test in Europe*, Munich, Germany, 1999, p. 55.

[45] C. Grecu et al., “Towards Open Network-on-Chip Benchmarks,” in *Proceedings of the First International Symposium on Networks-on-Chip*, Princeton, NJ, 2007, p. 205.

[46] R. P. Dick, D. L. Rhodes, and W. Wolf, “TGFF: task graphs for free,” in *Proceedings of the 6th international workshop on Hardware/software codesign*, Seattle, Washington, United States, 1998, pp. 97-101.

[47] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Computing Surveys (CSUR)*, vol. 31, pp. 406–471, Dec. 1999.

[48] D. Towsley, “Allocating programs containing branches and loops within a multiple processor system,” *IEEE Transactions on Software Engineering*, vol. 12, pp. 1018–1024, Oct. 1986.

[49] H. El-Rewini and H. H. Ali, “Static scheduling of conditional branches in parallel programs,” *Journal of Parallel and Distributed Computing*, vol. 24, pp. 41–54, Jan. 1995.

[50] A.-H. Liu and R. P. Dick, “Automatic run-time extraction of communication graphs from multithreaded applications,” in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, Seoul, Korea, 2006, pp. 46-51.

[51] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 1st ed. Prentice Hall, 1995.

[52] J. Hu and R. Marculescu, “Energy- and performance-aware mapping for regular NoC architectures,” *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 24, no. 4, p. 551--562, 2005.

[53] J. Hu and R. Marculescu, “Communication and task scheduling of application-specific networks-on-chip,” *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 5, p. 643, 2005.

[54] **C. Radu**, “The Current Stage in Developing some Automatic Design Space Exploration Algorithms for Networks-on-Chip,” Computer Science Department, “Lucian Blaga” University of Sibiu, PhD Technical Report no. 2, Oct. 2010.

[55] T. Lei and S. Kumar, “A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture,” in *Proceedings of the Euromicro Symposium on Digital Systems Design*, 2003, p. 180.

[56] “The Embedded System Synthesis Benchmarks Suite (E3S) website.” [Online]. Available: http://ziyang.eecs.umich.edu/~dickrp/e3s/.

[57] R. Pop and S. Kumar, “A survey of techniques for mapping and scheduling applications to network on chip systems,” *School of Engineering, Jonkoping University, Research Report*, vol. 4, p. 4, 2004.

[58] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, pp. 671-680, May 1983.

[59] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 1st ed. Prentice Hall, 1995.

[60] M. A. . Elmohamed, P. Coddington, and G. Fox, “A comparison of annealing techniques for academic course scheduling,” *Practice and Theory of Automated Timetabling II*, p. 92, 1998.

[61] S. Murali and G. D. Micheli, “Bandwidth-Constrained Mapping of Cores onto NoC Architectures,” in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, 2004, p. 20896.

[62] K. Srinivasan and K. S. Chatha, “A technique for low energy mapping and routing in network-on-chip architectures,” in *Proceedings of the 2005 international symposium on Low power electronics and design*, San Diego, CA, USA, 2005, pp. 387-392.

[63] C. M. Fiduccia and R. M. Mattheyses, “A Linear-Time Heuristic for Improving Network Partitions,” in *19th Conference on Design Automation, 1982*, 1982, pp. 175- 181.

[64] G. Ascia, V. Catania, and M. Palesi, “Multi-objective mapping for mesh-based NoC architectures,” in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Stockholm, Sweden, 2004, pp. 182-187.

[65] M. Laumanns, L. Thiele, and E. Zitzler, “SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization,” *Evolutionary Methods for Design Optimisation and Control*, pp. 95-100.

[66] R. Tornero, V. Sterrantino, M. Palesi, and J. M. Orduna, “A multi-objective strategy for concurrent mapping and routing in networks on chip,” in *Proceedings of the 2009 IEEE International Symposium on Parallel\&Distributed Processing*, 2009, pp. 1-8.

[67] **C. Radu** and L. Vinţan, “UNIMAP: UNIFIED FRAMEWORK FOR NETWORK-ON-CHIP APPLICATION MAPPING RESEARCH,” *Acta Universitatis Cibiniensis Technical Series*, May 2011.

[68] **C. Radu** and L. Vinţan, “Towards a Unified Framework for the Evaluation and Optimization of NoC Application Mapping Algorithms,” in *ACACES 2010 Poster Abstracts*, Terrassa (Barcelona), Spain, 2010, pp. 163 - 166.

[69] **C. Radu**, “Unified Framework for Network-on-Chip Application Mapping,” *unimap - Project Hosting on Google Code*. [Online]. Available: https://code.google.com/p/unimap/. [Accessed: 04-Feb-2011].

[70] “ULBS HPC cluster.” [Online]. Available: http://zamolxe.hpc.ulbsibiu.ro/. [Accessed: 07-Feb-2011].

[71] S. Murali and G. D. Micheli, “SUNMAP: a tool for automatic topology selection and generation for NoCs,” in *Proceedings of the 41st annual Design Automation Conference*, San Diego, CA, USA, 2004, pp. 914-919.

[72] G. Ascia, V. Catania, and M. Palesi, “A Multi-Objective Genetic Approach to Mapping Problem on Network-on-Chip,” *JUCS*, vol. 22, p. 2006.

[73] R. Marculescu and U. Y. Ogras, “‘It’s a small world after all’: NoC performance optimization via long-range link insertion,” *Ieee Transactions On Very Large Scale Integration Vlsi Systems*, vol. 14, no. 7, pp. 693-706.

[74] “A Survey of Network-On-Chip Tools,” 31-Mar-2011. [Online]. Available: http://scholarlyexchange.org/ojs/index.php/IJRRCS/article/view/8207. [Accessed: 22-Aug-2011].

[75] “The ns-3 network simulator website.” [Online]. Available: http://www.nsnam.org.

[76] F. Fazzino, M. Palesi, and D. Patti, “Noxim: Network-on-chip simulator,” *URL: http://sourceforge. net/projects/noxim [24.06. 2008]*.

[77] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, “Kilo-NOC,” in *Proceeding of the 38th annual international symposium on Computer architecture - ISCA ’11*, San Jose, California, USA, 2011, p. 401.

[78] Sheng Li, Jung Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*, 2009, pp. 469-480.

[79] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparso, and J. Madsen, “A Network Traffic Generator Model for Fast Network-on-Chip Simulation,” in

Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, 2005, pp. 780-785.
[80] “The EEMBC website.” [Online]. Available: http://www.eembc.org.
[81] “Simics.net - The technical support site for the Virtutech Simics full system simulation platform,” *https://www.simics.net/*, 2010. [Online]. Available: https://www.simics.net/.
[82] “Tachyon Parallel / Multiprocessor Ray Tracing System,” *http://jedi.ks.uiuc.edu/~johns/raytracer/*, 2010. [Online]. Available: http://jedi.ks.uiuc.edu/~johns/raytracer/.
[83] E. Ort and B. Mehta, “Java Architecture for XML Binding (JAXB),” *Java Architecture for XML Binding (JAXB)*. [Online]. Available: http://www.oracle.com/technetwork/articles/javase/index-140168.html.
[84] “XSD: XML Data Binding for C++,” *CodeSynthesis XSD - XML Data Binding for C++*. [Online]. Available: http://www.codesynthesis.com/products/xsd/.
[85] **C. Radu**, “Developing Network-on-Chip Architectures for Multicore Simulation Environments,” Computer Science Department, “Lucian Blaga” University of Sibiu, PhD Technical Report no. 1, Jun. 2010.
[86] J. J. Durillo, A. J. Nebro, and E. Alba, “The jMetal Framework for Multi-Objective Optimization: Design and Architecture,” in *CEC 2010*, Barcelona, Spain, 2010, pp. 4138-4325.
[87] D. E. Knuth, *The Art of computer programming: Seminumerical algorithms*. Addison-Wesley, 1981.
[88] D. H. Besset, *Object-Oriented Implementation of Numerical Methods: An Introduction with Java & Smalltalk*, First Edition. Morgan Kaufmann, 2000.
[89] S. Schlingmann, “Selbstoptimierendes Routing in einem Network-on-a-Chip,” Augsburg, Germany, 2007.
[90] S. E. Lee and N. Bagherzadeh, “Increasing the throughput of an adaptive router in network-on-chip (NoC),” in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, Seoul, Korea, 2006, pp. 82-87.
[91] Jingcao Hu, Umit Y. Ogras, and Radu Marculescu, “System-Level Buffer Allocation for Application-Specific Networks-on-Chip Router Design,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 2919-2933, 2006.
[92] A. Gancea, “Simulator pentru proiectarea, evaluarea și optimizarea unor rețele de interconectare tip NoC,” Diploma project (Bachelor), “Lucian Blaga” University of Sibiu, Romania (in Romanian, supervisor Professor Lucian Vintan, PhD), Sibiu, Romania, 2011.
[93] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.
[94] W. Trumler, S. Schlingmann, T. Ungerer, J. H. Bahn, and N. Bagherzadeh, “Self-optimized Routing in a Network-on-a-Chip,” presented at the 20th IFIP World Computer Congress, Milano, Italy, 2008.
[95] E. Salminen, K. Srinivasan, and Z. Lu, “OCP-IP Network-on-chip benchmarking workgroup,” *OCP-IP*, Dec-2010. [Online]. Available: http://www.ocpip.org/uploads/dynamic_areas/Cv8XdaKTKDztFpWKPqsl/6189/NoC%20Working%20Group%20Overview%20WP.pdf.

[96] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, “ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 3001 Leuven, Belgium, Belgium, 2009, pp. 423–428.

[97] D. Albonesi, “Power- and Reliability-Aware Microarchitecture,” in *ACACES 2000 Course*, Terrassa (Barcelona), Spain, 2009.

[98] **C. Radu** and L. Vinţan, “Optimizing Application Mapping Algorithms for NoCs through a Unified Framework,” in *Roedunet International Conference (RoEduNet), 2010 9th*, Sibiu, Romania, 2010, pp. 259 - 264.

[99] E. G. T. Jaspers and P. H. N. de With, “Chip-set for video display of multimedia information,” *IEEE TRANS. CONS. ELECTR*, vol. 45, p. 706--715, 1999.

[100] E. B. Van Der Tol and E. G. T. Jaspers, “Mapping of MPEG-4 decoding on a flexible architecture platform,” *MEDIA PROCESSORS 2002*, vol. 4674, p. 1--13, 2002.

[101] E. B. van der Tol, “Mapping of H.264 decoding on a multiprocessor architecture,” in *Proceedings of SPIE*, Santa Clara, CA, USA, 2003, pp. 707-718.

[102] S. Kirkpatrick, C. Gelatt, and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, pp. 671-680, May 1983.

[103] *Simulated Annealing, Theory with Applications*. Sciyo, 2010.

[104] *Simulated Annealing*. InTech, 2008.

[105] D. Fouskakis and D. Draper, “Stochastic Optimization: a Review,” *International Statistical Review*, vol. 70, no. 3, pp. 315-349, Jan. 2007.

[106] **C. Radu** and L. Vinţan, “Optimized Simulated Annealing for Network-on-Chip Application Mapping,” in *Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS-18)*, Bucharest, Romania, 2011, vol. 1, pp. 452–459.

[107] Z. Lu, L. Xia, and A. Jantsch, “Cluster-based Simulated Annealing for Mapping Cores onto 2D Mesh Networks on Chip,” in *Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, Washington, DC, USA, 2008, pp. 1–6.

[108] SLD:: System Level Design Group @ CMU, “NoCmap: an energy- and performance-aware mapping tool for Networks-on-Chip,” *SLD:: System Level Design Group @ CMU*, 2010. [Online]. Available: http://www.ece.cmu.edu/~sld/wiki/doku.php?id=shared:nocmap.

[109] H. Orsila, E. Salminen, and T. D. Hämäläinen, “Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems,” in *Simulated Annealing*, I-Tech Education and Publishing KG, 2008, pp. 321-342.

[110] **C. Radu** and L. Vinţan, “Domain-Knowledge Optimized Simulated Annealing for Network-on-Chip Application Mapping,” *Submitted to an Elsevier journal*, Sep. 2011.

[111] **C. Radu**, “Optimized Simulated Annealing for Network-on-Chip Application Mapping,” Computer Science Department, “Lucian Blaga” University of Sibiu, PhD Technical Report no. 3, Jun. 2011.

[112] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer, 2008.

[113] C. A. C. Coello, D. A. V. Veldhuizen, and G. B. Lamont, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 1st ed. Springer, 2002.

[114] **C. Radu**, S. Mahbub, and L. Vinţan, “Developing Domain-Knowledge Evolutionary Algorithms for Network-on-Chip Application Mapping,” *Journal of Systems Architecture (in review since July 25th, 2011).*

[115] G. Ascia, V. Catania, and M. Palesi, “Mapping cores on network-on-chip,” *International Journal of Computational Intelligence Research*, vol. 1, no. 1-2, pp. 109–126, 2005.

[116] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II,” *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, vol. 6, p. 182--197, 2000.

[117] W. Hung, C. Addo-Quaye, T. Theocharides, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, “Thermal-Aware IP Virtualization and Placement for Networks-on-Chip Architecture,” in *Proceedings of the IEEE International Conference on Computer Design*, 2004, pp. 430-437.

[118] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Addison-Wesley Professional, 1989.

[119] G. Syswerda, “A Study of Reproduction in Generational and Steady State Genetic Algorithms,” in *Foundations of Genetic Algorithms*, 1990, pp. 94–101.

[120] D. E. Goldberg and R. Lingle, “Alleles, Loci, and the Traveling Salesman Problem,” in *Proc.\ of the International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, pp. 154-159.

[121] D. Whitley, “A Genetic Algorithm Tutorial,” *STATISTICS AND COMPUTING*, vol. 4, p. 65--85, 1994.

[122] L. J. Eshelman, R. A. Caruana, and J. D. Schaffer, “Biases in the crossover landscape,” in *Proceedings of the third international conference on Genetic algorithms*, George Mason University, United States, 1989, pp. 10–19.

[123] K. Skadron, M. R. Stan, W. Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and D. Tarjan, “Temperature-aware microarchitecture,” in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings*, 2003, pp. 2- 13.

[124] C. C. N. Chu and D. F. Wong, “A matrix synthesis approach to thermal placement,” in *Proceedings of the 1997 international symposium on Physical design*, Napa Valley, California, United States, 1997, pp. 163–168.

[125] W. M. Spears, E.-mail Pears, and A. N. N. Mil, “Crossover or Mutation?,” *FOUNDATIONS OF GENETIC ALGORITHMS 2*, vol. 2, p. 221--237, 1992.

[126] **C. Radu**, “Evolutionary Algorithms for Network-on-Chip Application Mapping,” Computer Science Department, “Lucian Blaga” University of Sibiu, PhD Technical Report no. 4, Jun. 2011.

[127] H. Calborean, “Multi-Objective Optimization of Advanced Computer Architectures using Domain-Knowledge,” PhD Thesis, “Lucian Blaga” University of Sibiu, Romania, 2011 (PhD Supervisor: Prof. Lucian Vintan, PhD), Sibiu, Romania, 2011.

[128] M. M. Kim, J. D. Davis, M. Oskin, and T. Austin, “Polymorphic On-Chip Networks,” *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 101-112, 2008.

[129] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, “×pipesCompiler: a tool for instantiating application specific networks on chip,” in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, pp. 884-889.

[130] H. Calborean and L. Vintan, “An Automatic Design Space Exploration Framework for Multicore Architecture Optimizations,” in *Proceedings of The 9-th IEEE RoEduNet International Conference*, Sibiu, Romania, 2010, pp. 202-207.

[131] H. Calborean and L. Vinţan, “Framework for Automatic Design Space Exploration of Computer Systems,” *Acta Universitatis Cibiniensis Technical Series*, May 2011.

[132] R. Jahr, T. Ungerer, H. Calborean, and L. Vintan, “Automatic Multi-Objective Optimization of Parameters for Hardware and Code Optimizations,” in *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, 2011, pp. 308 – 316.

[133] H. Calborean, R. Jahr, T. Ungerer, and L. Vintan, “Optimizing a Superscalar System using Multi-objective Design Space Exploration,” in *Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania*, Calea Grivitei, nr. 132, 78122, Sector 1, Bucuresti, 2011, vol. 1, pp. 339–346.

[134] J. J. Durillo, J. García-Nieto, A. J. Nebro, C. A. C. Coello, F. Luna, and E. Alba, “Multi-Objective Particle Swarm Optimizers: An Experimental Comparison.,” in *EMO’09*, 2009, pp. 495-509.

[135] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. A. Coello Coello, F. Luna, and E. Alba, “SMPSO: A new PSO-based metaheuristic for multi-objective optimization,” in *ieee symposium on Computational intelligence in miulti-criteria decision-making, 2009. mcdm ’09*, 2009, pp. 66-73.

[136] D. E. Ghahraman, A. K. C. Wong, and T. Au, “Graph Optimal Monomorphism Algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 10, pp. 181-188, 1980.

[137] “Graph Monomorphism Algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 10, pp. 189-196, 1980.

[138] G. S. Lueker and K. S. Booth, “A Linear Time Algorithm for Deciding Interval Graph Isomorphism,” *Journal of the ACM*, vol. 26, pp. 183-195, Apr. 1979.

[139] “On the Cutting Edge: Simplified O(n) Planarity by Edge Addition.” [Online]. Available: http://academic.research.microsoft.com/Publication/1734993. [Accessed: 07-Sep-2011].

[140] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “Performance evaluation of the VF graph matching algorithm,” pp. 1172-1177.

[141] N. K. Bambha and S. S. Bhattacharyya, “Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, pp. 99-112, Feb. 2005.

[142] U. Y. Ogras and R. Marculescu, “Energy- and Performance-Driven NoC Communication Architecture Synthesis Using a Decomposition Approach,” pp. 352-357.