Reconciling performance and predictability on a noc-based mpsoc using off-line scheduling techniques

Manel Djemal Fakhfakh

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité Informatique

(École Doctorale Informatique, Télécommunication et Électronique)

Présentée par MANEL DJEMAL

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

RÉCONCILIER PERFORMANCE ET PRÉDICTIBILITÉ SUR UN MANY-COEUR EN UTILISANT DES TECHNIQUES D'ORDONNANCEMENT HORS-LIGNE

Soutenue le 27 juin 2014, devant le jury composé de

Mme.	FLORENCE MARANINCHI	Verimag, Grenoble	Rapporteur
M.	RENAUD SIRDEY	CEA, Saclay	Rapporteur
M.	BERTRAND GRANADO	UPMC, LIP6, Paris	Examinateur
M.	FRANÇOIS IRIGOIN	CRI - MINES ParisTech	Examinateur
M.	LOUIS MANDEL	Collège de France, Paris	Examinateur
M.	FRANÇOIS PÊCHEUX	UPMC, LIP6, Paris	Examinateur
Mme.	ALIX MUNIER-KORDON	UPMC, LIP6, Paris	Directeur de thèse
M.	DUMITRU POTOP-BUTUCARU	INRIA, Rocquencourt	Encadrant de thèse

PH.D. THESIS OF THE UNIVERSITY PIERRE AND MARIE CURIE

Department : COMPUTER SCIENCE AND MICRO-ELECTRONICS

Presented by: MANEL DJEMAL

Thesis submitted to obtain the degree of
DOCTOR OF THE UNIVERSITY PIERRE AND MARIE CURIE

RECONCILING PERFORMANCE AND PREDICTABILITY ON A NOC-BASED MPSOC USING OFF-LINE SCHEDULING TECHNIQUES

Defence on 27 June 2014, Committee:

Mme.	FLORENCE MARANINCHI	Verimag, Grenoble	Reviewer
M.	RENAUD SIRDEY	CEA, Saclay	Reviewer
M.	BERTRAND GRANADO	UPMC, LIP6, Paris	Examiner
M.	FRANÇOIS IRIGOIN	CRI - MINES ParisTech	Examiner
M.	LOUIS MANDEL	Collège de France, Paris	Examiner
M.	FRANÇOIS PÊCHEUX	UPMC, LIP6, Paris	Examiner
Mme.	ALIX MUNIER-KORDON	UPMC, LIP6, Paris	Advisor
M.	DUMITRU POTOP-BUTUCARU	INRIA, Rocquencourt	Co-Advisor

Remerciements

Je tiens à remercier en premier lieu Alix Munier, ma directrice de thèse, qui a toujours été à mon écoute et a veillé sur le bon déroulement de ma thèse.

Je remercie également Dumitru Potop-Butucaru, mon encadrant de thèse, pour m'avoir proposé un sujet passionnant, pour avoir eu la patience de m'encadrer pendant tout ce temps et pour les discussions fructueuses et les conseils avisés qil a su me prodiguer durant tout ce travail.

Ensuite, toutes mes remerciments à Florence Maraninchi et Renaud Sirdey d'avoir accepté la lourde tâche d'être rapporteurs de cette thèse, leurs commentaires ont augmenté mon recul par rapport aux domaines traités. Je remercie de même Bertrand Granado, François Irigoin, Louis Mandel et François Pêcheux de m'avoir fait l'honneur de faire partie de mon jury.

Mes remerciments s'adressent plus particulièrement à François Pêcheux, Frank Wajsburt et Zhen Zhang avec qui j'ai eu le plaisir de travailler sur certains aspects de ma thèse. Le temps n'aurait pas passé si vite s'il n'y avait pas eu Meriem Zidouni, Thomas Carle, Cécile Stentzel sans oublier leur soutien pendant la rédaction du manuscrit.

Je voudrais remercier aussi ma famille et mes amis qui m'ont soutenu, pour tout ce que je leur ai fait subir pendant tout ce temps, chacun à sa façon m'a aidé à traverser ces trois ans de ma vie.

Enfin, je remercie mon mari qui m'a soutenu, encouragé et supporté. Sans lui, cette thèse n'aurait été qu'une thèse et certainement la vie n'aurait pas été si belle. Sans oublier biensur mon ange Nour.

Résumé

Les réseaux-sur-puces (NoCs) utilisés dans les architectures multiprocesseurs-sur-puces posent des défis importants aux approches d'ordonnancement temps réel en ligne (dynamique) et hors-ligne (statique). Un NoC contient un grand nombre de points de contention potentiels, a une capacité de bufferisation limitée et le contrôle réseau fonctionne à l'échelle de petits paquets de données. Par conséquent, l'allocation efficace de ressources nécessite l'utilisation des algorithmes da faible complexité sur des modèles de matériel avec un niveau de détail sans précédent dans l'ordonnancement temps réel. Nous considérons dans cette thèse une approche d'ordonnancement statique sur des architectures massivement parallèles (Massively parallel processor arrays ou MPPAs) caractérisées par un grand nombre (quelques centaines) de coeurs de calculs. Nous identifions les mécanismes materiels facilitant l'analyse temporelle et l'allocation efficace de ressources dans les MPPAs existants. Nous déterminons que le NoC devrait permettre l'ordonnancement hors-ligne de communications, d'une manière synchronisée avec l'ordonnancement de calculs sur les processeurs. Au niveau logiciel, nous proposons une nouvelle méthode d'allocation et d'ordonnancement capable de synthétiser des ordonnancements globaux de calculs et de communications couvrants toutes les ressources d'exécution, de communication et de la mémoire d'un MPPA. Afin de permettre une utilisation efficace de ressources du matériel, notre méthode prend en compte les spécificités architecturales d'un MPPA et implémente des techniques d'ordonnancement avancées comme la préemption pré-calculée de transmissions de données. Nous avons évalué notre technique de mapping par l'implantation de deux applications de traitement du signal. Nous obtenons dans les deux cas de bonnes performances du point de vue de la latence, du débit et de l'utilisation des ressources.

Mots clés: Multiprocesseurs-sur-puce (many-coeur), réseau-sur-puce (NoC), ordonnancement hors-ligne, ordonnancement temps réel

Titre en anglais: Reconciling performance and predictability on a NoC-based MPSoC using off-line scheduling technique.

Abstract

On-chip networks (NoCs) used in multiprocessor systems-on-chips (MPSoCs) pose significant challenges to both on-line (dynamic) and off-line (static) real-time scheduling approaches. They have large numbers of potential contention points, have limited internal buffering capabilities, and network control operates at the scale of small data packets. Therefore, efficient resource allocation requires scalable algorithms working on hardware models with a level of detail that is unprecedented in real-time scheduling. We consider in this thesis a static scheduling approach, and we target massively parallel processor arrays (MPPAs), which are MPSoCs with large numbers (hundreds) of processing cores. We first identify and compare the hardware mechanisms supporting precise timing analysis and efficient resource allocation in existing MPPA platforms. We determine that the NoC should ideally provide the means of enforcing a global communications schedule that is computed off-line (before execution) and which is synchronized with the scheduling of computations on processors. On the software side, we propose a novel allocation and scheduling method capable of synthesizing such global computation and communication schedules covering all the execution, communication, and memory resources in an MPPA. To allow an efficient use of the hardware resources, our method takes into account the specificities of MPPA hardware and implements advanced scheduling techniques such as pre-computed preemption of data transmissions. We evaluate our technique by mapping two signal processing applications, for which we obtain good latency, throughput, and resource use figures.

Keywords: Chip-multiprocessor (Many-core), On-chip network (NoC), Off-line scheduling, Real-time scheduling

English Title: Reconciling performance and predictability on a NoC-based MPSoC using off-line scheduling technique.

Table of Contents

Remerciements **2**

Résumé **3**

Abstract **4**

1 Introduction **8**
1.1 Thesis motivation 8
1.1.1 The advent of many-cores 8
1.1.2 The advent of Networks-on-Chips 12
1.1.3 Many-cores for hard real-time applications 13
1.1.4 Mapping applications onto NoC-based many-cores 15
1.2 Thesis contributions 16
1.2.1 The DSPINpro programmable Network-on-Chip 17
1.2.2 The Automatic real-time mapping and code generation 17
1.2.3 An environment for virtual prototyping of MPPA applications . . 19
1.3 Outline 21

2 State of the art **22**
2.1 Network-on-Chip design 23
2.1.1 NoC building blocks 23
2.1.2 NoC topology 24
2.1.3 NoC switching 24
2.1.3.1 Routing 25
2.1.3.2 Switching method and buffering policy 26
2.1.3.3 Arbitration/Scheduling 29
2.1.4 Existing Network-on-Chip architectures 33
2.1.4.1 DSPIN 33
2.1.4.2 Æthereal 33
2.1.4.3 Nostrum 35
2.1.4.4 Kalray MPPA NoC 36
2.1.4.5 The scalar interconnect of MIT RAW 38
2.1.4.6 Other NoC architectures 39
2.1.4.7 Comparison with our work 40
2.2 Massively parallel processor arrays 40
2.2.1 Tilera TILE*Pro*64 40
2.2.2 Kalray MPPA-256 42

2.2.3 Adapteva Epiphany . . . 43
2.2.4 Intel SCC . . . 45
2.2.5 ST Microelectronics STHORM . . . 46
2.2.6 TSAR . . . 46
2.2.7 Academic MPSoC architectures with TDM-based NoC arbitration 47
2.3 Static application mapping . . . 47
2.3.1 Off-line real-time multi-processor scheduling . . . 49
2.3.1.1 The AAA/SynDEx methodology . . . 49
2.3.2 The StreamIt compiler for the MIT RAW architecture . . . 52
2.3.3 Compilation of the ΣC language for the Kalray MPPA256 platform . . . 54
2.3.4 Other mapping approaches . . . 55

3 Tiled MPPA architectures in SoCLib 57
3.1 MPPA structure . . . 57
3.2 Memory organization . . . 59
3.2.1 Distributed shared memory . . . 59
3.2.2 Address structure . . . 60
3.2.3 Global memory organization . . . 61
3.2.4 Tile memory organization . . . 62
3.2.5 Hardware/software interface . . . 63
3.3 Improving the timing predictability of the SoCLib tile . . . 64
3.4 SystemC simulation and compilation support . . . 68

4 Programmable NoC arbitration 70
4.1 The case for programmed arbitration . . . 71
4.1.1 The principle . . . 71
4.1.2 Target application classes . . . 72
4.1.3 The cost of programmability . . . 73
4.2 Programmable DSPIN . . . 74
4.2.1 NoC router extensions . . . 74
4.2.2 Area overhead . . . 77
4.3 A simple example in depth . . . 78
4.4 Case study: the FFT . . . 84
4.4.1 FFT algorithm description . . . 85
4.4.1.1 Mapping onto the MPPA architecture . . . 86
4.4.1.2 Traffic injection configuration . . . 87
4.4.2 Evaluation of the slow-down due to traffic injection . . . 88
4.4.3 Removing the slow-down through NoC programming . . . 89

5 Off-line mapping of real-time applications using LoPhT 94
5.1 Background: AAA using the Clocked Graphs formalism . . . 95
5.1.1 The Clocked Graph formalism . . . 95
5.1.1.1 Functional specification . . . 95
Clocks . . . 96
Clocked graphs . . . 97
Example . . . 98

Support of a clock . . . 99
Well-formed properties . . . 100
5.1.1.2 Non-functional specification . . . 101
Platform model . . . 101
Non-functional properties . . . 101
5.1.2 Off-line scheduling of CG specifications . . . 102
5.1.2.1 Scheduled clocked graphs . . . 102
5.1.2.2 Real-time scheduling problem . . . 103
5.1.2.3 Consistency of a scheduled clocked graph . . . 103
Notations . . . 103
Consistency properties . . . 104
5.1.2.4 Makespan-optimizing scheduling algorithm . . . 106
5.2 Static (off-line) mapping onto MPPA architectures . . . 109
5.2.1 AAA for NoC-based MPPA: The problem . . . 109
Limitations . . . 110
5.2.2 Extension of the CG format . . . 111
5.2.2.1 Modeling of MPPA resources . . . 111
NoC ressources: . . . 112
Tile ressources: . . . 113
5.2.2.2 Memory footprint specification . . . 113
5.2.2.3 Non-functional properties . . . 113
Worst-case computation durations . . . 113
Worst-case communication durations . . . 114
5.2.3 Makespan-optimizing scheduling . . . 115
5.2.3.1 Mapping NoC communications . . . 117
5.2.3.2 Multiple reservations . . . 119
5.3 Automatic code generation . . . 121
5.3.1 Tile code generation . . . 121
5.4 Experimental results . . . 124

Conclusion **127**

List of Publications **129**

Bibliography **130**

Chapter 1

Introduction

Contents

1.1 Thesis motivation 8
1.1.1 The advent of many-cores 8
1.1.2 The advent of Networks-on-Chips 12
1.1.3 Many-cores for hard real-time applications 13
1.1.4 Mapping applications onto NoC-based many-cores 15
1.2 Thesis contributions 16
1.2.1 The DSPINpro programmable Network-on-Chip 17
1.2.2 The Automatic real-time mapping and code generation 17
1.2.3 An environment for virtual prototyping of MPPA applications . 19
1.3 Outline 21

1.1 Thesis motivation

1.1.1 The advent of many-cores

Due to advances in circuit technology and performance limitation in wide-issue, super-speculative processors, Chip-Multiprocessor (CMP) or multi-core technology has become the mainstream in CPU designs. [Peng et al., 2007]

The number of transistors in micro-processor chips upholds today its historic trend of exponential growth, known as Moore's law, which is expected to continue until at least 2020 [Gordon E. Moore, 1965, Gordon E. Moore, 2005], as pictured in Fig. 1.1.

Until the years 2000, this growth mostly translated into micro-architecture gains aimed at improving mono-processor performance. Combined with a steady increase in operating frequencies, this allowed the continued use of a von Neumann computing paradigm [John von Neumann, 1945] where a sequential processor offers the performance needed by most general-purpose applications.

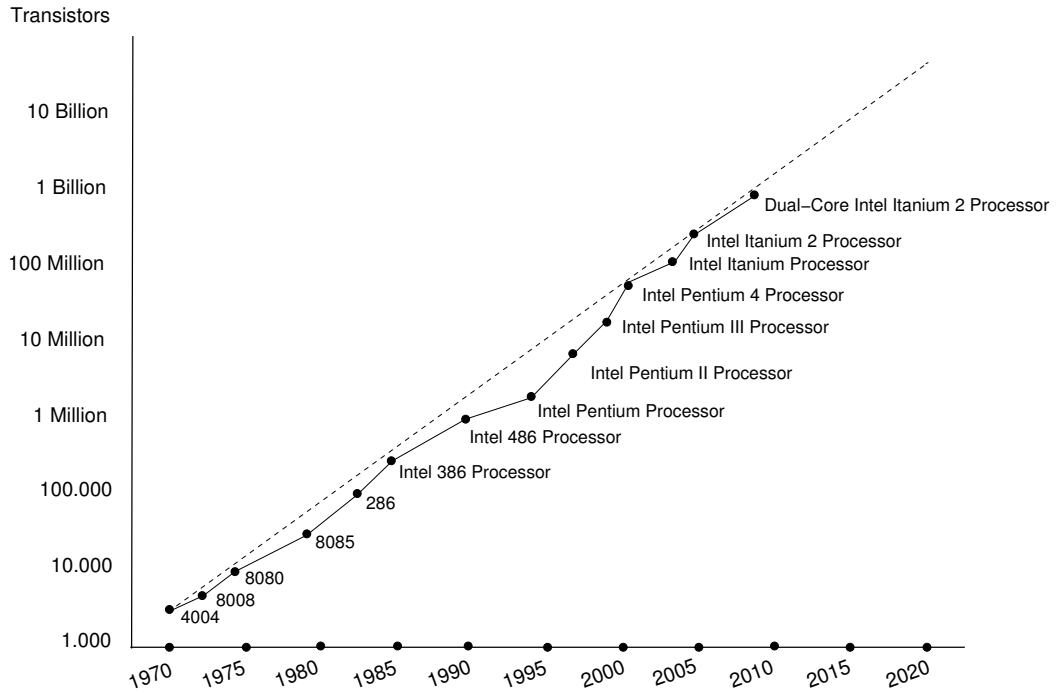


Figure 1.1: Moore's law (cf. [Held et al., 2006])

However, this is no longer possible. First of all, the last decade brought the end of the fast operating frequency increases which were the dominant cause of processor performance gains [J.Flynn, 2004]. Second, performance increase by micro-architectural advances alone follows the (empirical) Pollack's rule [Pollack, 1999] which states that the performance increase is roughly proportional to the square root of the increase in complexity (complexity in this context means processor logic, *i.e.* transistor count). In other words, doubling the logic in a processor core delivers only 40% more performance, while doubling the number of processors can almost double the speed for many applications.

The end of fast performance scaling for sequential processors led to an industry-wide shift towards parallel computing. While parallel architectures already existed, mainly in the high-performance and embedded computing fields, parallelism now entered the mainstream of general-purpose computing under the form of multi-core, and then many-core architectures. This trend started in 2001, when the first general-purpose processor that featured multiple processing cores on the same die was released by IBM: the POWER4 processor [IBM, 2001]. Since then, all major hardware vendors started shipping multi-core processors. Today, most personal computers, workstations and servers are based on multi-core chips.

Multi-core processors have clear benefits, such as scalable performance, improved reliability, or an easier energy and thermal management. But these benefits come at the price of as many challenges (we only list here a few):

- **Scalable performance**: Our computing environments offer more and more potential for parallelism, coming from either the nature of the applications (*e.g.* video

streaming) or the fact that multiple applications are run in parallel. But exploiting this parallelism requires significant changes in the way systems are built, in both software and hardware.

In software, the main difficulty is that of producing correct and efficient parallel programs. These programs must expose parallelism at the good level, and also provide guarantees of correctness in the presence of concurrency. To facilitate multi-core programming, a variety of languages and formalisms have been proposed in the past years, and significant effort has been invested in the software engineering of such applications [OpenMP, 2008, Khronos, 2011] and in the automatic parallelization of previously-existing sequential code [Beletska et al., 2011, Cetus, 2004, Irigoin et al., 2012].

In hardware, as the computing cores are counted in dozens and hundreds, the classical shared-bus communication approaches no longer scale. This leads to major bottlenecks in the memory hierarchy, in the inter-core communication network, and in the access to external data sources. Solving these issues requires a significant improvement of the on-chip and the off-chip interconnects.

But the most challenging issues are the complex design decisions concerning both hardware and software. Indeed, designing a complex hardware/software system consists in solving a multi-criteria optimization problem having as parameters efficiency, facility of programming, predictability, hardware complexity, price, etc. Solving such an optimization problem involves complex trade-offs. For instance, choosing a multi-core architecture with support for cache coherency facilitates shared memory programming, but leads to a poor temporal predictability and an increased hardware complexity. On the contrary, having no hardware support for cache coherency improves predictability but requires software control of memory consistency. Clearly, these architectural choices have a profond influence on both the specification and mapping of parallel applications onto multi-core platforms.

- **Energy efficiency and thermal management**: As the number of transistors and the computation power increase, power and temperature management becomes more and more critical and difficult, and needs to be addressed in either hardware or a combination of both hardware and software [Hanumaiah and Vrudhula, 2012].

  Multi-core architectures usually provide at least one of two classical hardware mechanisms that facilitate power and temperature management: The possibility of turning off unused processor cores and the possibility of running cores at optimized supply voltages and frequencies. Furthermore, load balancing among the processor cores can be used to distribute thermal dissipation across the die.

- **Fault-tolerance and reliability**: The evolution of silicon technologies results in a steady increase of transistor densities. At the same time, the increase in transistor counts means that chip sizes do not decrease. The combination of the two results in a significant increase of the probability of hardware defects per chip [Furber, 2006].

> Coupled with the need for reasonable yields, the increase in transistor count will bring an end to current design practices where only chips that are 100% functional are accepted. However, tolerating defects in chips requires support in both hardware and software, through redundancy and support for failure isolation and (re-)configuration [Zhang, 2011].

Intense research and industrial developments have resulted in the definition of several classes of multi-core processors corresponding to different contexts of use and programming styles. Among them:

- General-purpose multi-cores, such as AMD Phenom or Intel Core.
- Application-specific System-on-Chip (SoC) platforms, such as TI OMAP or Qualcomm SnapDragon [Texas Instruments, 2009, Qualcomm, 2011], which have emerged from the embedded computing community.
- Graphics Processing Units (GPUs) [Nvidia, 1999], which have emerged from the image processing and high-performance computing community, and which are optimized for Single Instruction Multiple Data (SIMD) execution.

But in the past few years, a clear and consistent convergence can be seen between the historically isolated communities of general-purpose, embedded, and high-performance computing. All three communities have moved towards so-called *many-core* platforms characterized by:

- *Large numbers of simpler cores*. The number of cores ranges here from a few tens to a few hundreds in production architectures, and to thousands of cores in research platforms.
- *Novel memory architectures* that can deliver higher bandwidth access through the use of multiple memory banks localized near the processors. Data localization often requires that the memory hierarchy is exposed, at least in part, to the programmer.
- *New interconnect types*, like the Network-on-Chips (NoCs), that provide higher performance and/or scalability than classical interconnects such as buses and crossbars.

As graphically illustrated in Fig. 1.2, this hardware-level convergence between the general-purpose, embedded, and high-performance computing communities has several causes:

- In the general-purpose computing community, the use of the *massively parallel processor array (MPPA)* chips [1] such as Kalray MPPA [MPPA, 2012], Adapteva Epiphany [Adapteva, 2012], Tilera TilePro [Tilera, 2008], or Intel SCC and MIC [Howard and al., 2011, MIC, 2010] improves scalability and/or energy efficiency.

[1] We will come back with an in-depth description of MPPA architectures in the following chapters.

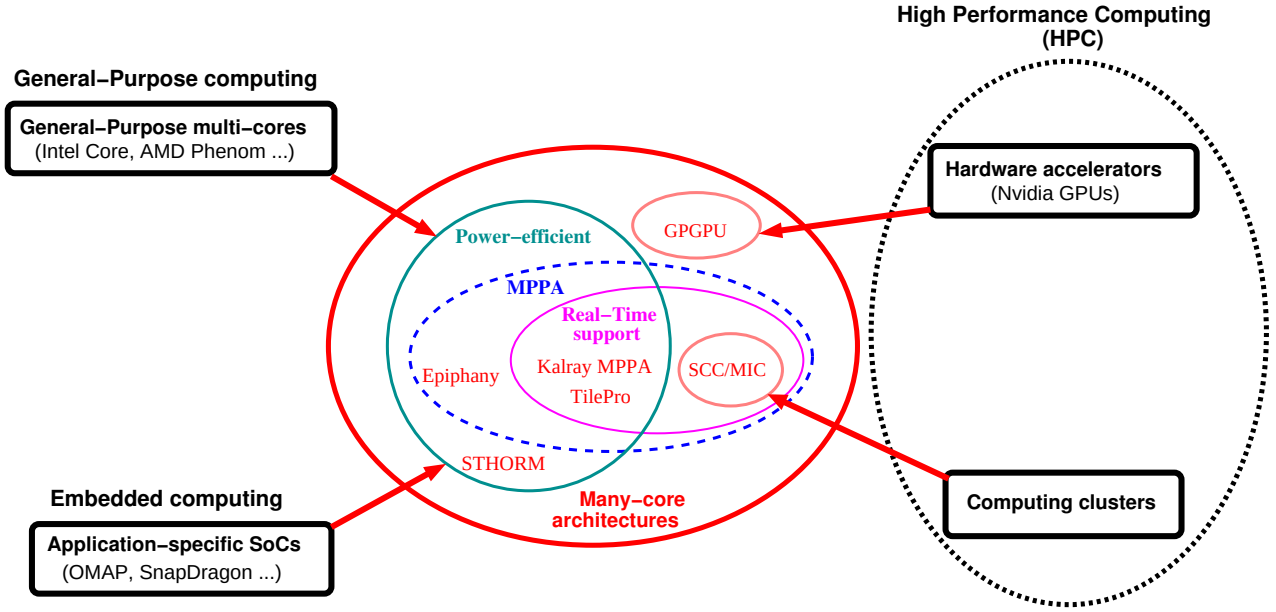


Figure 1.2: The many-core convergence

- Embedded applications also benefit from improved scalability and energy efficiency, like in the ST STHORM [Benini, 2010] platform. In addition, they often require some support for real-time implementation, such as mechanisms for resource reservation or spatial and temporal separation, as offered by the Kalray MPPA and Tilera TilePro chips.

- In the high-performance computing community, hardware accelerators like the GPUs evolved towards General-Purpose GPUs (GPGPUs) architectures whose more expressive instruction sets provide more flexibility to the programmer. Energy-efficient architectures such as Adapteva Epiphany also position themselves as accelerator chips. Finally, many-core chips, such as Intel SCC and MIC [Howard and al., 2011, MIC, 2010], are the result of yet anothor tradition. Here energy efficiency is not an objective *per se*, but appears as a by-product of the need to parallelize high-performance applications through methods previously used on computing clusters.

1.1.2 The advent of Networks-on-Chips

The first multi-core architectures used bus-based interconnects such as ARM's AMBA [ARM, 1999] and IBM's CoreConnect [PLB, 2001]. However, an on-chip bus can only perform one communication at a time. As the number of cores increased, the buses became major contention points and performance bottlenecks, and new interconnect paradigms were investigated.

From a computation speed point of view, the ideal choice is that of crossbar interconnects where every two components are directly linked. A crossbar introduces no contention point in addition to the ones associated with the resources connected to it (RAM

banks and other peripherals). But the hardware cost of a crossbar increases quadratically with the number of resources connected to it. Thus, crossbars can only be used in systems or sub-systems with a small number of components.

Finding a trade-off between performance (*e.g.* speed), scalability, and system cost led to the development of the Network-on-Chip (NoC) paradigm, which takes inspiration from classical computer networks [Benini and De Micheli, 2002, Sgroi et al., 2001]. Like a classical computer network, a NoC is formed of standardized point-to-point links and NoC routers that can be composed following simple rules to form complex interconnect graphs (regular or not). The use of multiple links avoids the creation of global bottlenecks. At the same time, the number of links is kept at a reasonable level (usually linear in the number of computation and storage components) guaranteeing scalability in large designs.

1.1.3 Many-cores for hard real-time applications

The objective of this thesis is to investigate the use of many-core architectures using a NoC interconnect in the implementation of complex real-time applications. We are targeting two types of applications:

- Hard real-time applications, like those used in embedded control systems in the avionics and automotive industries. In such applications, the non-respect of deadlines may lead to catastrophic results like the loss of life or serious damage to the environment.

- Highly regular signal/image processing applications. In such applications, a tight control of timing, like the one we propose, helps in improving computation speed.

In such applications, timing guarantees should be obtained before execution, ideally by static analysis methods (as opposed to measurement-based ones). Furthermore, the timing guarantees should be *precise* in order to avoid the waste of computing resources. But determining precise timing guarantees is inherently difficult on many-core platforms due to the large number of potential contention points. It is imperative to identify (and then eliminate or control) the sources of timing impredictability at all levels of the many-core architecture[2] and the application software.

From existing work on WCET analysis for classical single-core and multi-core architectures, we know that certain microarchitectural features make timing analysis more difficult [Wilhelm and Reineke, 2012, Hardy and Puaut, 2008]. Such features are the shared caches and the cache/memory coherency mechanisms. We shall assume that neither of them is used in the many-core platforms considered in this thesis. One particular consequence of this assumption is that *all* communications and synchronizations between different processor cores are performed through one or more NoCs.[3]

[2]Comprising the processor cores, the memory sub-system, the communication network, and the I/O devices.

[3]As opposed to using dedicated communication devices such as the complex memory hierarchies of TSAR [TSAR, 2008].

But eliminating microarchitectural sources of impredictability is not enough, as the NoCs themselves raise serious problems when the objective is to ensure efficiency and predictibility. Indeed, a NoC has large numbers of potential contention points: Whenever a NoC router is connected to at least 3 links, contentions are possible when data arriving on 2 links must be sent onto the third. In common NoC architectures, this is the case of all routers, and arbitration is needed to control the access to every NoC link.

Furthermore, one communication often traverses several NoC routers, and can be subject to contentions at the level of each one. This is why the use of fair arbitration policies at the level of NoC routers is not good when the objective is to provide tight timing guarantees for NoC communications. Previous work on NoC-based architectures for the real-time subject have explored the use of arbitration policies similar to those used for ensuring Quality-of-Service (QoS) in computer networks. Several approaches have been proposed: circuit switching, bandwidth reservation, priority-based scheduling, and programmed arbitration.

In NoCs based on *circuit switching* [Hilton and Nelson, 2006], communications are performed along a set of *circuits*. Each circuit is a sequence of point-to-point NoC links. The fundamental constraint is that two circuits cannot share a link. The absence of resource sharing lowers utilization of NoC resources and increases costs. However, once a connection has been established, it can use the full bandwidth of all its links and real-time guarantees are easily computed.

Virtual circuit switching is an evolution of circuit switching which allows resource sharing between circuits. Resource sharing requires the use of arbitration, and several types of arbitration techniques have been proposed: TDM-based, bandwidth management-based, and priority-based.

Most interesting from the point of view of timing predictability are NoCs where arbitration is based on *time division multiplexing (TDM)*. TDM-based NoCs [Goossens et al., 2005, Millberg et al., 2004b] allow the computation of precise latency and throughput guarantees. The same type of latency and throughput guarantees (albeit less precise) can be obtained in NoCs relying on bandwidth management mechanisms, such as Kalray MPPA [Harrand and Durand, 2011].

But NoCs using TDM arbitration or bandwidth management have certain limitations. The most important is that they largely ignore the fact that the needs of an application may change during execution, depending on the *state* of the application. One way of taking into account the application state is by using NoCs with support for *priority-based arbitration*. But priority-based arbitration requires the use of costly virtual channel mechanisms [Howard and al., 2011, Miro Panades et al., 2006], which limits applicability to systems supporting only a few priority levels. The alternative to priority-based approaches is to use a NoC allowing *programmed NoC arbitration*, such as MIT RAW [Waingold et al., 1997] or the architecture proposed in this thesis. Programmed arbitration allows the enforcement of static arbitration and routing patterns of data transmissions, as demanded by the application.

1.1.4 Mapping applications onto NoC-based many-cores

The introduction of NoC-based architectures was accompanied by the definition of novel mapping techniques targeting NoC-based MPSoCs. Indeed, if paralelism is recognized as the only way of providing scalable performance, this scalability comes at the price of increased complexity of both the software and the software mapping (allocation and scheduling) process.

Part of this complexity can be attributed to the steady increase in the *quantity* of software that is run by a single system. But there are also significant *qualitative* changes concerning both the software and the hardware. In software, more and more applications include *parallel* versions of classical signal or image processing algorithms [Aubry et al., 2013, Gerdes et al., 2012, Villalpando et al., 2010], involving potentially complex synchronizations between the sequential programs executed on the various cores. Such applications are best modeled using data-flow models (as opposed to so-called *independent tasks* that are common in classical real-time).

Designing parallel software is difficult in itself, relying on notoriously hard disciplines such as parallel programming [Kwok and Ahmad, 1999] and multi-processor scheduling [Ramamritham et al., 1993]. The picture is further complicated when considering real-time aspects. Providing functional and real-time correctness guarantees requires an accurate control of the functional and temporal interferences due to concurrent use of shared resources. Depending on the hardware and software architecture, this can be very difficult [Wilhelm and Reineke, 2012]. In our case, there are two main reasons to this: The first one concerns the NoCs: as the tasks are more tightly coupled and the number of resources in the system increases, the on-chip networks become critical resources, which need to be explicitly considered and managed during real-time scheduling. Recent work [Shi and Burns, 2010, Kashif et al., 2013, Nikolic et al., 2013] has determined that NoCs have distinctive traits requiring significant changes to classical multi-processor scheduling theory [Goossens et al., 2003]. The second reason concerns automation: the complexity of many-cores and of the (parallel) applications mapped on them is such that the *allocation and scheduling must be largely automated*.

Efficient and real-time implementation of applications onto NoC-based systems remains largely an open problem, with the issue of best mapping of computation parts (threads, tasks,...) onto processing resources amply recognized, while the issue of best use of the interconnect NoC to route and transfer data still less commonly tackled.

In the most general case, dynamic allocation of applications and channel virtualization can be guided by user-provided information under various forms as in OpenMP for Open Multi-Processing [OpenMP, 2008], CUDA for Compute Unified Device Architecture [Nvidia CUDA, 2006], and OpenCL for Open Computing Language [Khronos, 2011]. But there is no clear guarantee of optimality. Conversely there are consistent efforts, in the domains of embedded and HPC computing, aiming at automatic parallelization, compile-time mapping and scheduling optimization. They rely on the fact that applications are often known in advance, and deployed without disturbance from foreign applications, and

without uncontrolled dynamic creation of tasks.

The results of this thesis fit in this "static application mapping" case. We focus on mapping techniques where *all* allocation and scheduling decisions are taken off-line. In theory, off-line algorithms allow the computation of *scheduling tables* specifying an *optimal* allocation and real-time scheduling of the various computations and communications onto the resources of the MPPA. In practice, this ability is severely limited by 3 factors:

1. The **application** may exhibit a high degree of dynamicity due to either environment constraints or to execution time variability resulting from data-dependent conditional control.[4]

2. The **hardware** may not allow the implementation of optimal scheduling tables. For instance, most MPPA architectures provide only limited control over the scheduling of communications inside the NoC.

3. The **mapping** problems we consider here are NP-hard. In practice, this means that optimality cannot be attained, and that efficient heuristics are needed.

1.2 Thesis contributions

In our work, we are interested in a specific sub-class of many-core architectures: the *massively parallel processor arrays (MPPAs)* characterized by:

- A large number of processing cores, ranging in current silicon implementations from a few tens to a few hundreds. The cores are typically chosen for their area and energy efficiency, instead of raw computing power.

- A regular internal structure where processor cores and internal storage (RAM banks) are divided among a set of identical *tiles*, which are connected through one or more NoCs with regular structure (*e.g.* mesh or torus topologies). In this thesis, we are focussing on 2D-Mesh micro topologies.

- The capability of executing in parallel a different task on each core. Known as *task parallelism* or *multiple-instruction, multiple-data (MIMD)*, this parallel computing paradigm is also that of classical distributed computing and multi-processor real-time scheduling.

My contributions in this thesis concern the **hardware** and the **mapping** technique. On the hardware design side, I extend an existing state-of-the-art NoC architecture to allow programmed arbitration and thus provide the best support for off-line scheduling. On the mapping side, I explain how low-level details of NoC-based MPPA architectures can be taken into account in a scalable way to allow the synthesis of schedules with

[4]Implementing an optimal control scheme for such an application may require more resources than the application itself, which is why on-line scheduling techniques are often preferred.

unprecedented timing precision. To evaluate our novel NoC architecture and mapping technique we build a cycle-accurate model of a NoC-based MPPA and a new tool for the automatic real-time mapping and code generation, called LoPhT.

1.2.1 The DSPINpro programmable Network-on-Chip

Our purpose is to investigate how the underlying architecture should offer the proper infrastructures to implement optimal computation and communication mappings and schedules. We concretely support our proposed approach by extending the DSPIN 2D mesh network-on-chip (NoC) [Panades, 2008] developed at UPMC-LIP6. In the DSPIN NoC, we replace the fair arbitration modules of the NoC routers with static, micro-programmable arbiters that can enforce a given packet routing sequence, as specified by small programs. We advocate the desired level of expressiveness/complexity for such simple configuration programs. The result is named Programmable DSPIN, or *DSPINpro*.

To improve the efficiency and predictability of our MPPA architecture, and thus facilitate real-time mapping, we also constrained and standardized the structure of the computing tiles connected to the *DSPINpro* NoC, as well as the software architecture of our implementations. On the hardware side, we constrain the type and number of tile components (CPUs, RAM banks, DMA units ...).[5] On the software architecture side, we constrain the memory organization, we impose that all computations are performed on local tile data, with specific "send" operations being in charge of all inter-tile data transfers (along with a software lock mecanism), and we require the explicit placement of input and output data of the tile on memory banks.

1.2.2 The Automatic real-time mapping and code generation

We propose a technique and tool for the automated mapping of real-time applications onto MPPA architectures based on 2D mesh NoCs. Our tool is named LoPhT, for **Lo**gical to **Ph**ysical **T**ime Compiler.

The global flow of our mapping technique, pictured in Fig. 1.3, is similar to that of the AAA methodology and the SynDEx tool [Grandpierre and Sorel, 2003]. It takes as input functional specifications provided under the form of data-flow synchronous specifications *à la* Scade/Lustre [Caspi et al., 2003]. Our input formalism allows the specification of conditional execution and execution modes, which are common features in complex embedded control specifications.

To map such specifications onto MPPA architectures and provide hard real-time guarantees, we rely on *off-line* allocation and scheduling algorithms. These algorithms also take as input a description of the MPPA platform, and non-functional constraints including allocation constraints and conservative upper estimates for the:

[5] We have also developed a tool allowing the automatic synthesis of the corresponding SystemC models and memory maps from simple architecture specifications defining the NoC size and the type and number of tile components.

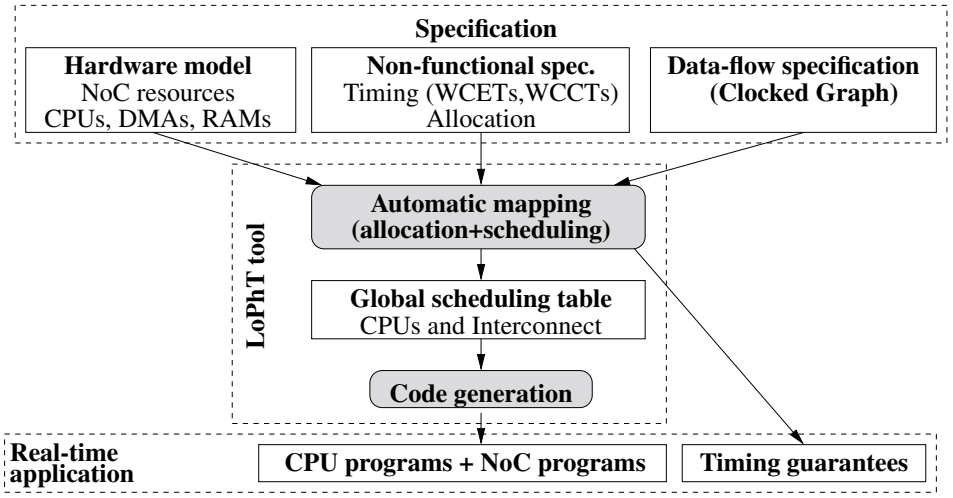


Figure 1.3: Global flow of the proposed mapping technique

- Worst-case execution times (WCETs) of the data-flow blocks (seen as atomic computations).
- The worst-case size of data transmitted through the data-flow arcs, needed to determine the worst-case communication time (WCCT) of basic communication/synchronization operations.

Starting from these inputs, our algorithms build *reservation tables* (also called *scheduling tables* or simply *time tables*) specifying for each resource of the platform its use by various computations or communications. Reservation tables are then converted into sequential code ensuring the correct ordering of operations on each resource and the respect of the real-time guarantees.

Our main contribution was to provide new mapping algorithms that explicitly take into account MPPA-specific features. The first problem here is that NoCs are very different from the communication buses used in classical real-time scheduling work. NoCs are composed of *multiple communication resources* that must be considered separately during mapping. However, reservation of a communication path along the NoC requires the *synchronized reservation* of resources along the path, due to the limited amount of buffer memory inside the NoC. The second problem is that the large number of computation and communication resources requires the use of scalable, yet precise mapping algorithms.

To provide tight real-time guarantees, our mapping heuristics seek to achieve a good parallelization of the application while *ensuring that concurrent computations and communications do not interfere (functionally or temporally)* with each other outside of functionally-needed synchronization points. The absence of interferences reduces the pessimism of the worst-case timing analysis, and limits resource over-allocation. Achieving such functional and temporal isolation can be done with low overhead in MPPA architectures that provide the programmer with good control over the memory hierarchy and the interconnect.

We evaluated our hardware extensions and mapping technique by automatically implementing two signal processing applications: A model of an automotive embedded control application and an implementation of the Fast Fourier Transform (FFT). These two examples provide a good illustration of how abstract data-flow communications between compute operations have to be organized according to crossroad traffics at routers, once computations have been mapped to processing elements. For both applications, NoC arbitration programming reduces contentions, communication time, and therefore global execution time. We obtain in both cases good latency, throughput, and resource use figures.

1.2.3 An environment for virtual prototyping of MPPA applications

Together, our extension of the DSPIN NoC and the development of a novel mapping technique and tool define a new environment for the virtual prototyping of real-time MPPA applications. The expression "virtual prototyping" is used here to mark the fact that hardware execution is simulated in software, as opposed to "emulated" on an FPGA target after full hardware synthesis. The SoCLib library [LIP6, 2011] allows both. We preferred the simulation-based approach because it facilitated both hardware design and the precise timing measures needed to evaluate our real-time mapping technique on the resulting platform. Note that no difference exists between simulation and emulation from the point of view of software or the real-time properties, given that simulation is of *cycle-accurate, bit-accurate (CABA)* type.

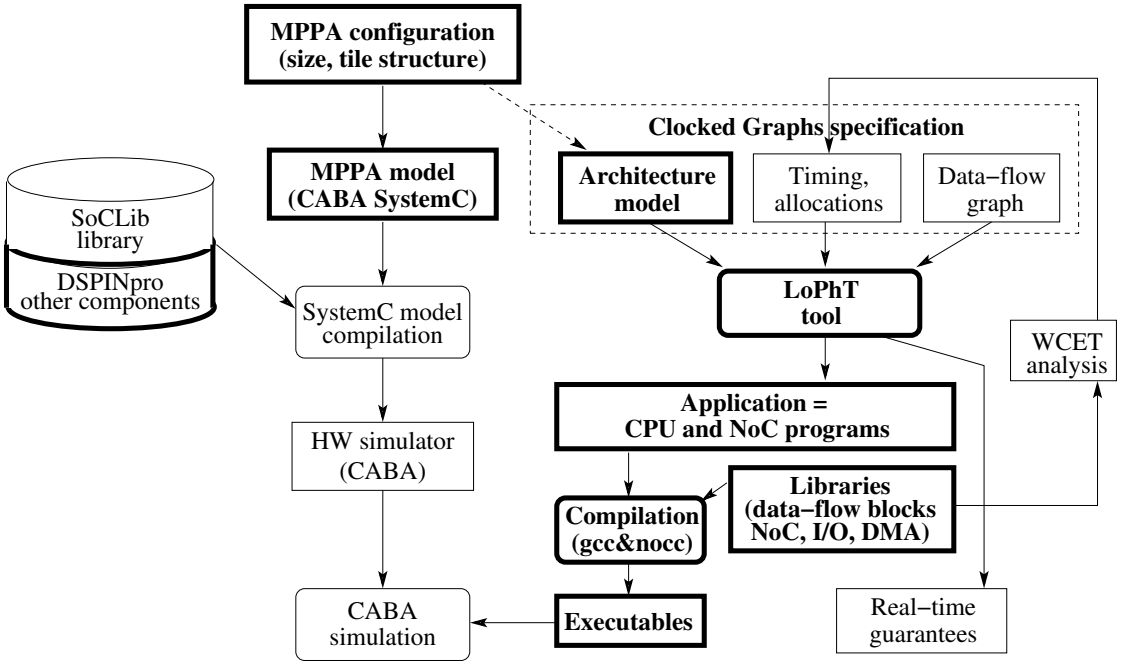


Figure 1.4: An environment for virtual prototyping of MPPA applications

In our environment, described in Fig. 1.4, objects with thick borders are artefacts or

transformations that were created or significantly modified as part of this thesis. In our environment, we start by choosing the high-level configuration parameters of the MPPA platform, which include the size (number of tiles) of the MPPA, the structure of the standard tile, and the positioning of I/O devices. Once configuration is fixed, we can instantiate the MPPA SystemC model. This model uses standard components from the SoCLib library, but also the DSPINpro network-on-chip and other hardware components developed on top of SoCLib as part of our effort to build an MPPA platform reconciling timing predictability and efficiency. The model is then compiled to obtain the hardware simulator.

The configuration parameters are also used to build the architecture model taken as input by our mapping tool LoPhT. LoPhT also takes as input a functional specification provided under the form of a data-flow synchronous specification and a non-functional specification defining the worst-case durations of all data-flow blocks and communications, and possible allocation constraints. Worst-case durations are obtained through WCET analysis of the C code associated with the data-flow blocks and other library functions (but this aspect will not be covered in this thesis, the interested reader is invited to read [Puaut and Potop-Butucaru, 2013]). The LoPhT tool takes this input specification and transforms it into statically scheduled code for the CPU cores and the NoC routers. This code is compiled, separately for each sequential resources, using either `gcc` (the C code of the CPUs) or with `nocc` (designed and implemented by us) for the NoC programs. Resulting code is executed on the hardware simulator, which allows us to verify that the real-time guarantees computed by LoPhT are respected.

Of course, this environment is the result of highly collaborative work started between my team (INRIA AOSTE) and the SoC team of the UPMC/Lip6 laboratory (led at the time by A.Munier). From the SoC team, the main participants were François Pêcheux, Franck Wajsburt and Zhen Zhang, which have carried out most of the hardware design. From the AOSTE team, in addition to myself, Thomas Carle helped in the implementation of the scheduler, and Robert de Simone provided major insights on the high-level architectural modeling. My personal contributions are the following:

- I participated in the definition of the DSPINpro NoC and the MPPA platform by specifying what services the hardware should provide in order to allow efficient and predictable real-time implementation. Actual definition of the hardware was carried out by the Lip6 team and D. Potop.

- I defined the architecture model taken as input by LoPhT.

- Starting from off-line mapping algorithms originally defined by T. Carle and D. Potop, I have extended them to cover the NoC-specific and MPPA-specific aspects, such as the synchronized reservation of NoC resources.

- I have defined the code generation scheme which ensures that the real-time guarantees computed by LoPhT are respected in the running implementations. This

scheme takes into account the low-level detail of the MPPA platform to allow for low-overhead synchronization and for high precision in resource allocation.

1.3 Outline

My thesis is organized as follows:

Chapter 2 summarizes the state of the art in NoC design and in the mapping and scheduling for multi-core and many-core platforms.

Chapter 3 presents the detail of the DSPIN NoC and the SoCLib-based MPPA architecture on which our work is based. It also presents the changes brought to the tiles of the SoCLib-based MPPA to allow predictable and efficient implementation.

Chapter 4 presents the concept of programmed arbitration and its implementation in the DSPINpro NoC. It also presents an evaluation of the gains obtained through semi-automatic mapping of two applications onto the new platform.

Chapter 5 presents the LoPhT tool for automatic real-time mapping of embedded control specifications onto MPPAs.

Chapter 2

State of the art

Contents

2.1 Network-on-Chip design 23
2.1.1 NoC building blocks 23
2.1.2 NoC topology 24
2.1.3 NoC switching 24
2.1.4 Existing Network-on-Chip architectures 33
2.2 Massively parallel processor arrays 40
2.2.1 Tilera TILE*Pro*64 40
2.2.2 Kalray MPPA-256 42
2.2.3 Adapteva Epiphany 43
2.2.4 Intel SCC 45
2.2.5 ST Microelectronics STHORM 46
2.2.6 TSAR 46
2.2.7 Academic MPSoC architectures with TDM-based NoC arbitration 47
2.3 Static application mapping 47
2.3.1 Off-line real-time multi-processor scheduling 49
2.3.2 The StreamIt compiler for the MIT RAW architecture 52
2.3.3 Compilation of the Σ*C* language for the Kalray MPPA256 platform 54
2.3.4 Other mapping approaches 55

Our work extends the state of the art in three fields: Network-on-Chip (NoC) design, Massively Parallel Processor Array (MPPA) design, and mapping techniques for multi/many-cores. We start this section with a general introduction of NoC-related concepts, and then we identify and compare the hardware mechanisms supporting precise timing analysis and efficient resource allocation in existing NoC architectures. Most important, Section 2.1.3 reviews existing NoC switching paradigms and their support for

real-time application mapping. Section 2.2 presents some examples of existing industrial and academic MPPA platforms. Finally, we review related work on application mapping for multi-core, distributed, and many-core architectures.

2.1 Network-on-Chip design

The Network-on-Chip (NoC) paradigm has been defined by its authors as a "layered-stack" approach to the design of on-chip interconnect [Sgroi et al., 2001], inspired by the classical layered models for computer networks [Zimmermann, 1988]. In this paradigm, the communication functions of the NoC are organized in a set of logical layers, each layer using the services of the lower layers in order to provide higher-level services.

At the highest abstraction level, a NoC can be viewed as a classical telecommunication network offering lossless communication services with some QoS properties between a set of computation and storage elements (CPU cores, RAM banks, I/O devices, etc). These services rely on lower-level routing and switching algorithms controlling the behavior of the NoC building blocks: links, routers, and interfaces. We briefly describe these elements here. Our description insists on the traffic management mechanisms supporting real-time implementation.

2.1.1 NoC building blocks

A Network-on-Chip (NoC) is formed of components of three types: links, routers, and Network Interface Controllers (NICs).

- **Links**: The communication links are the central data transmission media. They interconnect all the routers and the NICs. Links are unidirectional, point-to-point media. Bidirectional point-to-point communication lines can be obtained by pairing two links, one for each communication direction. One fundamental characteristic of links is their buffering capability, which will be discussed later.

- **Routers**: They implement the flow control policies (routing, arbitration) and define the overall strategy for moving data through the NoC. As shown in Fig. 2.1, each NoC router is composed of *buffers*, which provide temporary storage for incoming and outgoing data, *routing components*, which are essentially *demultiplexers* (labeled **D**) with some control logic, and *scheduling/arbitration components*, which are essentially multiplexers (labeled **M**) with some control logic.

- **Network Interface Controllers (NICs)**: Like in a classical computer network, storage and computing components of a many-core are often grouped in a number of sub-systems, called Processing Elements (PEs). The NICs provide the logic

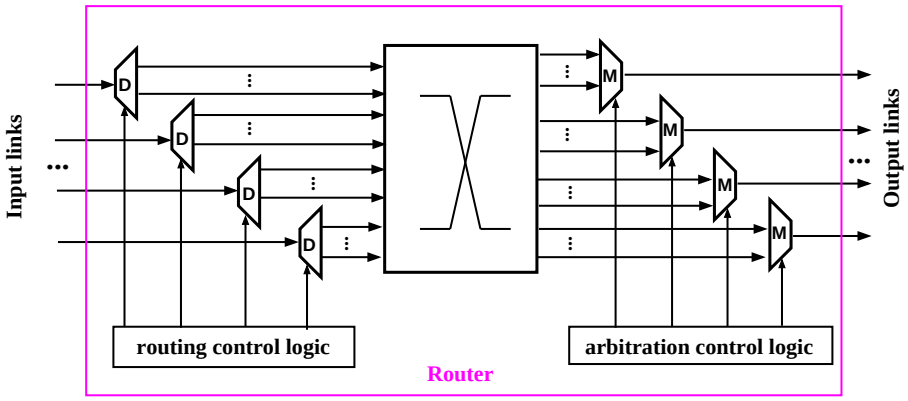


Figure 2.1: Generic router model

connection between the PEs and the NoC. Its main task is protocol conversion between the one used locally within the PE and the one of the NoC. In particular, NICs are responsible for building the data packets transferred through the NoC.

2.1.2 NoC topology

The topology of a NoC is usually modeled by an adjacency graph describing how the routers, NICs, and PEs are interconnected using NoC links. NoC topologies can be either regular or irregular. The most common regular topologies are presented in Fig. 2.2. In this figure, the routers are represented with squares and the PEs with circles. Each PE contains exactly one NIC. The arcs represent either unidirectional links, or bidirectional pairs of links.

Various comparisons between various regular topologies (in terms of latency, throughput, and energy dissipation) can be found in [Pande et al., 2005]. In this thesis we focus on 2D-Mesh NoCs like the ones used in the Adapteva Epiphany [Adapteva, 2012], Tilera Tilepro [Tilera, 2008], or DSPIN [Panades, 2008]. The structure of a router in a 2D mesh NoC (which is a specialization of the general router structure of Fig. 2.1) is presented in Fig. 2.3. It has five bidirectional connections (labeled North, South, East, West, and Local) to the bidirectional links leading to the four routers next to it and the local PEs. Note that mesh and torus NoCs are often used in so-called *tiled many-core* architectures, where most PEs have a standard form (approximated with a rectangle) resulting in a regular, tiled organization of the many-core chip. This is why in tiled many-cores the PEs are usually called *computing tiles* or simply *tiles*.

2.1.3 NoC switching

The allocation of NoC resources to the various data transmissions is governed by a set of design choices and algorithms commonly known as the *switching policy* of the NoC. This

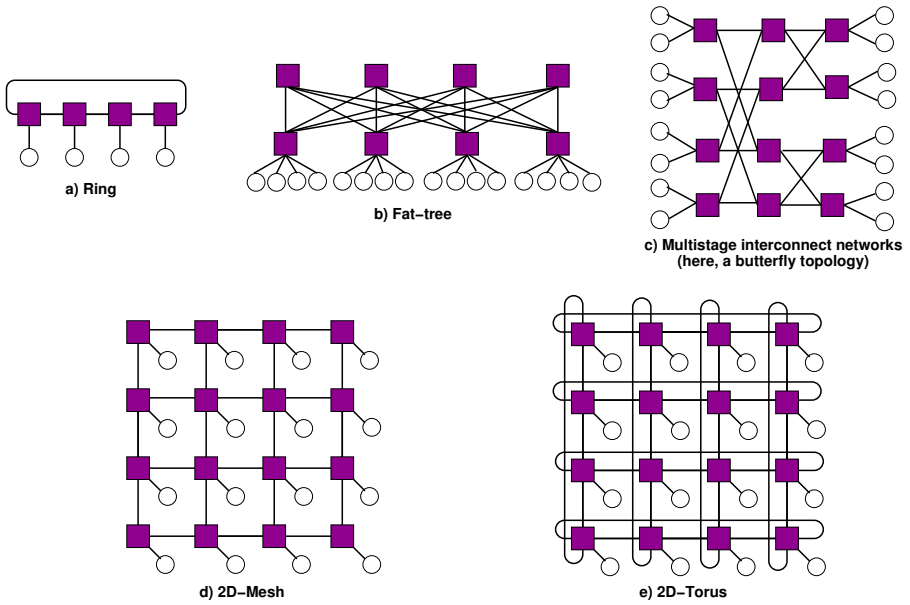


Figure 2.2: Some regular NoC topologies

very general notion covers all aspects of NoC data transmission: the organization of data into transmission units, routing, buffering, and arbitration/scheduling. We briefly discuss these aspects.

2.1.3.1 Routing

The routing algorithm, implemented by the NoC router demultiplexers, defines how data is routed from its source towards its destination. It must decide at each intermediate router which output link(s) are to be selected for each incoming data packet. Routing algorithms can be classified according to various criteria. According to the number of destinations of individual data transmission operations, routing algorithms can be *unicast*, when each data transmission operation has a single destination, *multicast*, when a transmission can have several destinations, or *broadcast*, when each data is transmitted to all PEs. In this thesis, we use unicast routing. In unicast routing each data message arriving at a router through an incoming link must be forwarded through only one of the output links.

Depending on *where* routing decisions are taken, routing algorithms can be divided into *source* routing and *non-source* routing algorithms. In source routing, the whole path is fixed by the sender of the data and explicitly encoded in the headers of the data packets. The path information is read and used at each router traversed by the packet. In non-source routing, each router makes its own decisions locally, depending on parameters such as the final destination of the packet.

Non-source routing algorithms are either *static* (sometimes called deterministic) or *adaptive*. When static routing is used, all traffic between given source and destination fol-

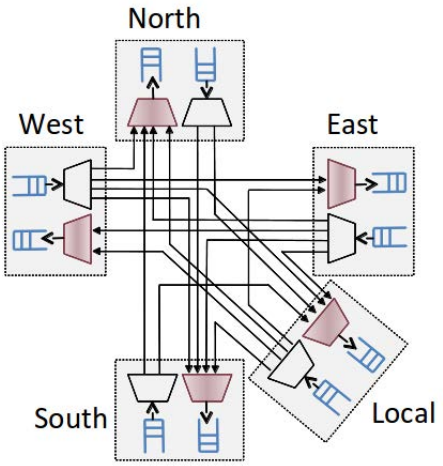


Figure 2.3: Generic router for a 2D mesh NoC with X-first routing policy

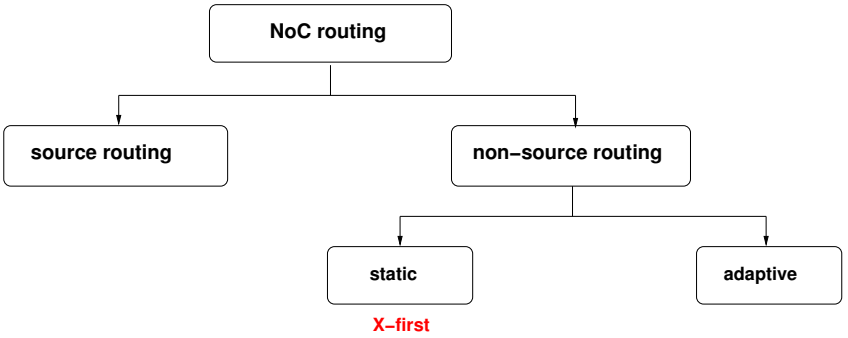


Figure 2.4: Classification of NoC routing algorithms

lows the same route (path). Adaptive algorithms can dynamically modify the routing path depending on network conditions such as the presence of faulty or congested channels. In an adaptive NoC, data packets sent from a given source towards a given destination can follow different paths and arrive in an order different from the sending order. Adaptive routing can reduce congestion situations but the dynamic nature of adaptive routing means that it makes timing analysis more difficult, and thus complicates implementation when hard real-time guarantees are needed. Existing NoC architectures with support for real-time scheduling do not employ adaptive routing, and we make the same choice. In this thesis, we will use the classical X-first policy.

2.1.3.2 Switching method and buffering policy

All NoC switching methods belong to one of two basic switching paradigms: circuit switching or packet switching. In *circuit switching* [Hilton and Nelson, 2006], communications are performed through dedicated communication channels (called circuits) that connect the source and destination PEs. A circuit consists in a sequence of point-to-point

physical links going all the way from the source PE to the destination PE. Two channels cannot share a physical link. This is achieved by statically fixing the output direction of each demultiplexer and the data source of each multiplexer along the channel path. Timing interferences between circuits are imposible. Thus, throughput is guaranteed and latency is predictible, but NoC use is usually low.

In *packet switching*, data is divided into small *packets* that are transmitted independently. Each packet is formed of a sequence of *flits*, where a flit is the amount of data that a link can transmit at the same time (in one clock cycle, for a synchronous NoC). Each packet must contain, in its *header flits* all the information needed to perform its communication, such as destination, priority, *etc.* The packetization of data allows a link to be used by multiple data transitions at the same time, by interleaving the transmission of the packets coming from different sources. This is why NoC use figures are usually improved when comparing to circuit switching approaches.

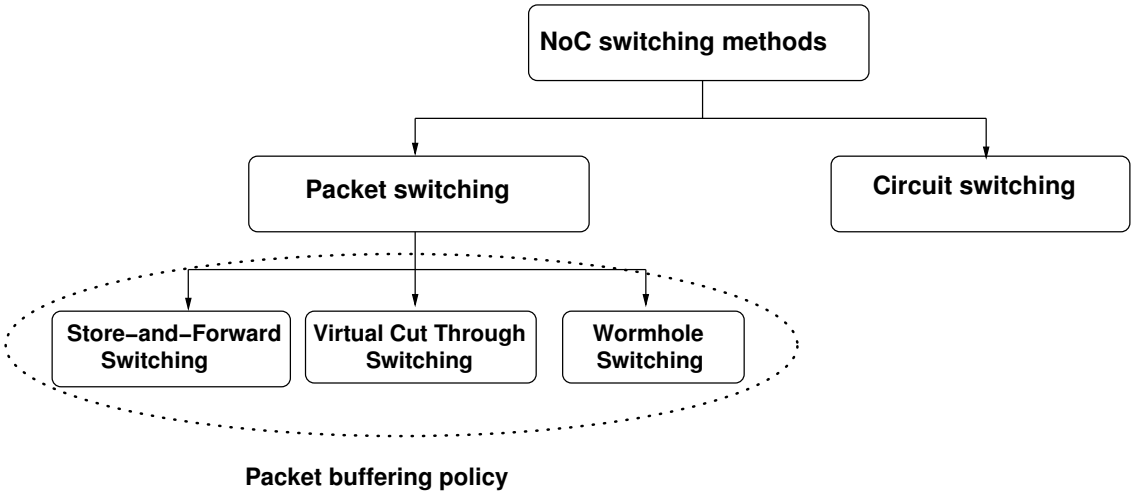


Figure 2.5: Classification of NoC switching techniques (part 1)

NoCs are complex communication networks where each data transmission traverses at least 2 routers. Achieving a good transmission throughput over such a network requires the use of some buffering in its routers and links. As pictured in Fig. 2.5, commonly-used buffering policies are *store-and-forward*, *virtual cut-through* and *wormhole*.

Store-and-forward This is the simplest buffering policy, used in most telecommunication networks. It requires that a router receives and stores a full data packet before forwarding it to the next router or NIC. To allow the storage of full packets, this method requires a large amount of buffering space in each router. This is why it is seldom used in NoCs [Kumar et al., 2002].

Wormhole In this buffering policy, a router makes its routing and arbitration decisions as soon as the header flits of a packet arrive. These flits are needed because they

may contain information such as the packet destination, or its priority, which are required by the routing and arbitration algorithms. Once the decisions are made, transmission can start as soon as the needed outgoing link allows transmission. This policy reduces communication latency and only requires a small buffering capacity. This is why it is the most common buffering policy in NoCs, used in all architectures described in the remainder of this thesis.

Virtual cut-through This approach is intermediate between the store-and-forward and wormhole buffering policies. Like in wormhole switching, forwarding can start as soon as one flit has been received. But forwarding can only start when signalling ensures that the next router on the path can receive the full packet [Sadawarte et al., 2011]. Thus, if a packet is blocked waiting for a link to be freed, it will be stored entirely on one router without blocking others. By comparison, in wormhole routing a blocked packet can stretch over several routers, blocking resources in all of them. Virtual cut-through is used in the NoC of the Intel SCC many-core chip [Howard and al., 2011].

A second classification criterion divides NoC switching techniques into *connection-oriented* and *connection-less* ones (*cf.* Fig. 2.6): Connection-oriented techniques rely on dedicated (logical) connection paths established prior to the actual transmission of data. Connections can be created and destroyed using specific operations that typicaly carry a large timing penalty, but once a connection is established, communication along it is facilitated. In connectionless switching techniques, the communication occurs in a dynamic manner with no prior connection-oriented resource allocation. By definition, circuit switched communication is connection-oriented, wheras packet switched communication can be either connection-oriented (based on virtual circuit approaches detailed below), or connection-less.

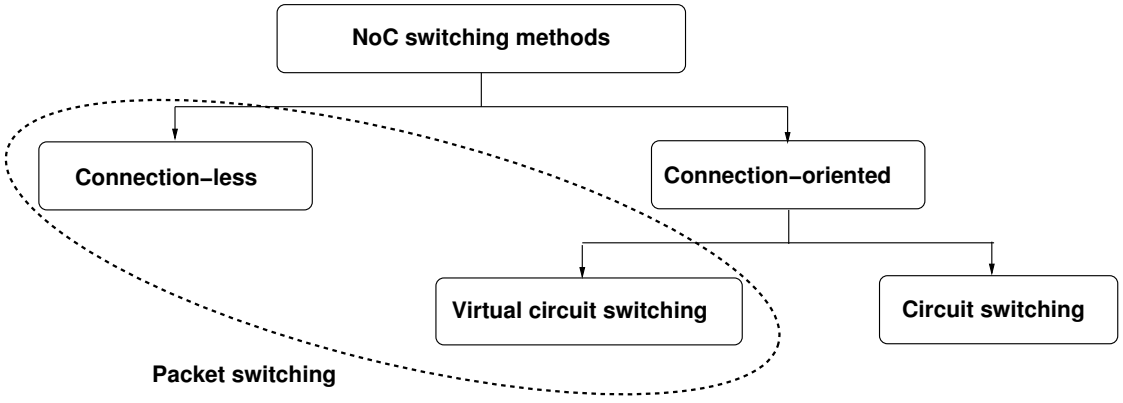


Figure 2.6: Classification of NoC switching techniques (part 2)

2.1.3.3 Arbitration/Scheduling

In packet switching NoCs a link can be shared by several data transmissions, and arbitration is needed to determine how packets belonging to multiple data transmissions are interleaved. Arbitration is realized at the level of NoC multiplexers.[1] As pictured in Fig. 2.7, several arbitration mechanisms are used in practice, each offering different levels of support for real-time implementation.

The simplest and most common arbitration technique is fair arbitration. In this approach, if two or more packets arriving from different sources request at the same time to pass through a NoC multiplexer, a fair arbitration policy such as Round Robin is used to decide which one passes first. The process is repeated whenever such a contention occurs. Along with a limitation on NoC packet sizes, the use of fair arbitration ensures that the NoC resources are evenly distributed among the data transmissions using them, with good NoC utilization factors and guaranteed (albeit possibly low) transmission throughputs for each transmission. Fair arbitration is used in the industrial NoC-based platforms Adapteva Epiphany [Adapteva, 2012], Kalray MPPA256 [MPPA, 2012], and ST STHorm [Benini, 2010]. In the Tilera TilePro64 [Tilera, 2008] chip, fair arbitration is used in 5 out of the 6 NoCs. In research prototypes, fair arbitration can be found from the early NoC architectures, such as SPIN [Guerrier and Greiner, 2000], to the more recent designs, where it is used in conjunction with other types of arbitration, such as priority-based, time division multiplexing-based, or programmed arbitration, as described below.

When designing real-time systems, the objective is to respect the real-time requirements. Achieving this goal on resource-constrained architectures usually amounts to achieving the best possible *predictable efficiency*. Given the large number of potential NoC contention points (router multiplexers), and the synchronizations induced by data transmissions, providing *tight* static timing guarantees is only possible if some form of system-level flow control mechanism is used. The tightest timing control is provided by circuit switching approaches. As explained earlier in this section, in circuit switching NoCs all communications are performed through dedicated communication channels formed of point-to-point physical links. Channels are set up so that they share no NoC resource, which makes timing interferences impossible. Once a channel is set up, communication latency and throughput are the best possible and timing analysis is easy. But the absence of resource sharing is also the main drawback of circuit switching, resulting in low utilization of the NoC resources. Even more important, the number of channels that can be established is limited by the number of NoC links, which severely limits application mapping choices.

This is why most NoCs use a packet switching approach. In this case, four types of

[1] Arbitration mechanisms will be presented in greater detail in Chapter 4

NoC control mechanisms have been proposed to improve the real-time properties: Time Divison Multiplexing (TDM), bandwidth reservation, priority-based arbitration, and programmed arbitration. As shown in Fig. 2.7, more than one arbitration scheme can be supported in a NoC architecture, like in Æthereal (TDM-based, fair, and priority-based) or DSPIN (fair and priority-based).

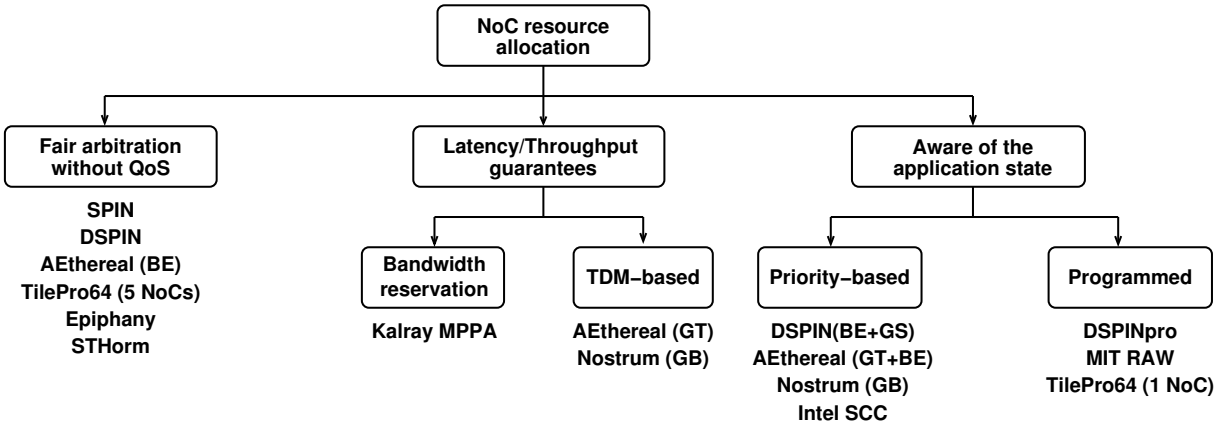


Figure 2.7: Classification of NoC resource allocation policies in packet-switched NoCs

TDM-based and bandwidth reservation resource allocation techniques are often used in the context of *virtual circuit switching* communication protocols. Virtual circuit switching is an evolution of circuit switching which allows NoC link sharing between circuits, but ensures that each circuit has configurable and guaranteed latency and throughput.

The most popular way of providing these guarantees is by relying on *time division multiplexing (TDM)* arbitration in the NoC routers. Such NoCs relying on TDM arbitration are Æthereal [Goossens et al., 2005], Nostrum [Millberg et al., 2004b], and others [Sorensen et al., 2012]. In a TDM NoC, all routers share a common time base (the hardware clock). The point-to-point links are reserved for the use of the virtual circuits following a fixed cyclic schedule (a scheduling table). The reservations made on the various links ensure that communications can follow their path without waiting. TDM-based NoCs allow the computation of *precise latency and throughput guarantees*. They also ensure a strong *temporal isolation* between virtual circuits, so that changes to a virtual circuit do not modify the real-time characteristics of the other.

When no global time base exists, the same type of latency and throughput guarantees can be obtained in NoCs relying on *bandwidth management* mechanisms such as Kalray MPPA [MPPA, 2012, Harrand and Durand, 2011]. The idea here is to ensure that the throughput of each virtual circuit is limited to a fraction of the transmission capacity of a physical point-to-point link, by either the emitting tile[2] or by the NoC routers. Two or

[2]In which case the actual NoC arbiters can be simple fair arbiters.

more virtual circuits can share a point-to-point link if the sum of their transmission needs is less than what the physical link provides.

But TDM and bandwidth management NoCs have certain limitations: One of them is that latency and throughput are correlated [Shi and Burns, 2010], which may result in a waste of resources. But the latency-throughput correlation is just one consequence of a more profound limitation: TDM and bandwith management NoCs largely ignore the fact that the needs of an application may change during execution, depending on its *state*. For instance, when scheduling a dependent task system with the objective of reducing task graph makespan, it is often useful to allow some communications to use 100% of the physical link, so that they complete faster, before allowing all other communications to be performed. One way of taking into account the application state is by using NoCs with support for *priority-based scheduling* [Shi and Burns, 2010, Panades, 2008, Howard and al., 2011]. In these NoCs, each data packet is assigned a priority level (a small integer), and NoC routers allow higher-priority packets to pass before lower-priority packets. To avoid *priority inversion* phenomenons, higher-priority packets have the right to preempt the transmission of lower-priority ones. In turn, this requires the use of one separate buffer for each priority level in each router multiplexer, a mechanism known as *virtual channels* in the NoC community[Panades, 2008].

The need for virtual channels is the main limiting factor of priority-based arbitration in NoCs. Indeed, adding a virtual channel (VC) is as complex as adding a whole new NoC[Yoon et al., 2010, Carara et al., 2007], and NoC resources (especially buffers) are expensive in both power consumption and area [Moscibroda and Mutlu, 2009]. To our best knowledge, among existing silicon implementations only the Intel SCC chip offers a relatively large numbers of VCs (eight) [Howard and al., 2011], and it is targeted at high-performance computing applications. Industrial MPPA chips targeting an embedded market usually feature multiple, specialized NoCs [Tilera, 2008, Adapteva, 2012, Harrand and Durand, 2011] without virtual channels. Other academic NoC architectures feature low numbers of VCs. Two VCs is a popular choice, in which case the channels are often labeled “guaranteed service”, for the high-priority one, and “best effort”, for the low-priority one [Goossens et al., 2005, Panades, 2008]. Current research on priority-based communication scheduling has already integrated this limitation, by investigating the sharing of priority levels [Shi and Burns, 2010].

The second limiting factor related to priority-based NoCs is that the associated scheduling theory mainly focuses on independent task systems. However, we have already explained that the large number of computing cores in an many-core architecture means that applications are likely to include parallelized code which is best modeled by large sets of relatively small *dependent tasks* with predictable functional and temporal behavior [Villalpando et al., 2010, Aubry et al., 2013, Gerdes et al., 2012]. Such timing-

predictable dependent task systems are those that can *a priori* take advantage of an off-line scheduling approach, as opposed to on-line priority-based scheduling. But *efficient* off-line mapping requires NoCs with support for *static communication scheduling* [Tilera, 2008, Djemal et al., 2012, Waingold et al., 1997]. The idea here is to determine an efficient (possibly optimal) global computation and communication schedule, represented with a scheduling table, and then enforce it through synchronized *sequential computation and communication programs*. Computation programs run on processor cores to sequence task executions and the initiation of communications. Communication programs run on specially-designed micro-controllers that control each NoC multiplexer to fix the order in which individual data packets are transmitted. Synchronization between the programs is ensured by the data packet communications themselves.

Like in TDM NoCs, the use of global scheduling tables allows the computation of very precise latency and throughput estimations. Unlike in TDM NoCs, static communication scheduling allows NoC resource reservations dependent on the application state. Global clock synchronization is not needed, and existing NoCs based on static communication scheduling do not use it[Tilera, 2008, Waingold et al., 1997, Djemal et al., 2012]. Instead, global synchronization is realized by the data transmissions themselves (which eliminates some of the run-time pessimism of TDM-based approaches).

The microcontrollers that drive each NoC router multiplexer are similar in structure to those used in TDM NoCs to enforce the TDM reservation pattern. The main difference is that the communication programs are usually longer than the TDM configurations, because they must cover longer execution patterns. This requires the use of larger program memory (which can be seen as part of the tile program memory[Djemal et al., 2012]). But like in TDM NoCs, buffering needs are limited and no virtual channel mechanism is needed.

From a mapping-oriented point of view, determining exact packet transmission orders cannot be separated from the larger problem of building a global scheduling table comprising both computations and communications. By comparison, mapping onto MPPAs with TDM-based or bandwith reservation-based NoCs usually separates task allocation and scheduling from the synthesis of a NoC configuration independent from the application state [Lu and Jantsch, 2007, Aubry et al., 2013] .

Under static communication scheduling, there is little run-time flexibility, as all scheduling possibilities must be considered during the off-line construction of the global scheduling table. For dynamic applications this can be difficult. This is why existing MPPA architectures that allow static communication scheduling, including the one we propose in this thesis, also allow communications with dynamic (Round Robin) arbitration.

2.1.4 Existing Network-on-Chip architectures

To provide a better understanding of the NoC switching concepts introduced above, we review several existing NoC architectures before explaining, in the next section, how they are used in existing many-core architectures.

2.1.4.1 DSPIN

The DSPIN NoC (for Distributed Scalable Predictable Interconnect Network) [Panades, 2008] was designed at the LIP6 laboratory by a team led by Alain Greiner. It is part of the SoCLib virtual prototyping library [LIP6, 2011] and has been physically implemented by ST Microelectronics [Miro Panades et al., 2006]. DSPIN extends concepts of the previously-defined SPIN on-chip interconnect. The main objective of the extension were to facilitate the design of tiled MPSoC architectures through the use of a regular interconnect topology, and to allow the design of globally asynchronous, localy synchronous (GALS) MPSoCs where each tile can have its own local clock, different in frequency and/or phase from the clocks of other tiles.

DSPIN has a regular 2D mesh topology. It uses a static X-first routing algorithm, and follows a wormhole packet switching approach. Arbitration is based on a combination of priority-based and fair algorithms. There are two priority levels, labeled guaranteed service (GS) and best effort (BE). When GS traffic reaches a NoC router multiplexer it interrupts ongoing BE traffic (if any). Interruption is done with flit granularity, meaning that the transmission of a BE packet is stalled, to be continued only after all GS packets have passed. Arbitration among packets of the same priority level is fair, using a Round Robin algorithm.

The programmable NoC developed as part of this thesis is based on DSPIN. This is why we provide an in-depth review of DSPIN in Chapter 3 (except for the priority-based arbitration features which we do not use).

2.1.4.2 Æthereal

The Æthereal NoC was developed at Philips Research Laboratories by a team lead by K. Goossens [Goossens et al., 2005]. Like DSPIN, Æthereal uses a mix of arbitration policies to support two types of NoC traffic named guaranteed service (GS) and best effort (BE). Like in DSPIN, fair arbitration is used when multiple BE traffic flows arrive at a router multiplexer at the same time, and priority-based arbitration is used to let GS traffic interrupt BE traffic. Unlike in DSPIN, GS traffic is subject here to TDM-based arbitration, under a form dubbed *contention-free routing*.

We only detail here the TDM-based arbitration of GS traffic. To allow TDM-based arbitration, Æthereal ensures that all NoC components share a global time basis (the NoC

is globally synchronous). In this time basis, NoC time is divided into *slots* of equal duration. Slots of different NoC routers are aligned between different NoC components (there is no phase shift).

Each NoC router multiplexer contains a *slot table* of fixed size. During execution, this table is cyclically traversed from beginning to the end to determine which traffic will be accepted at each time slot. More precisely, if T is the slot table associated with some router multiplexer, and if its length is n, then during slot s the multiplexer accepts input from the direction described by $T[s \bmod n]$. This direction is either one of the input links of the local router, or a special value specifying that the slot is to be left unused by GS traffic.

To allow contention-free routing, the slot tables of the NoC must satisfy a coherency property. More precisely, Æthereal requires that if some data reaches a NoC router at slot s (through one of the input links), then it leaves the router at slot $s+1$. This mechanism is illustrated in Fig. 2.8, which is reproduced from [Goossens et al., 2005]. This figure depicts 3 routers of a 2D mesh implementation of Æthereal. To simplify presentation, links to the local tiles are not represented. Thus, each router is connected with the routers next to it in the directions West, North, East, and South. In each of these directions, two links are used (one in each sense). In each router, the 4 multiplexers controlling the output links are labelled with $o_0 \ldots o_3$, and the 4 input links with $i_0 \ldots i_3$.

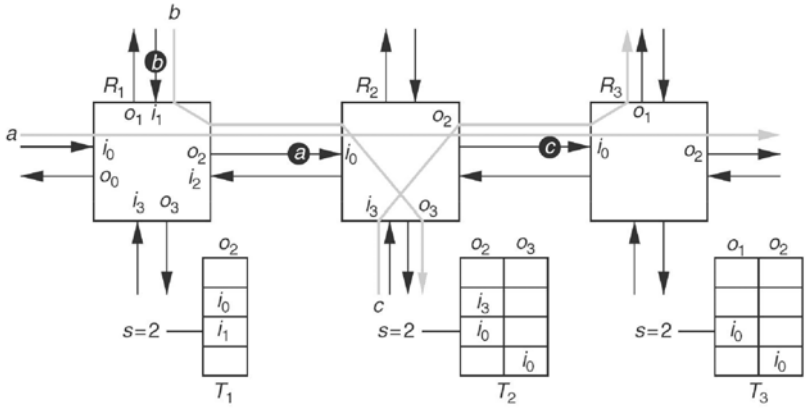


Figure 2.8: Æthereal contention-free routing (reproduced from [Goossens et al., 2005])

Fig. 2.8 only provides the slot tables of multiplexers o_2 of router R_1, o_2 and o_3 of router R_2 and o_1 and o_2 of router R_3. The slot tables have all size 4, and encode transfers along the 3 routes a, b, and c represented with light gray arrows. Slots in the slot tables are numbered from 0 (top one) to 3 (bottom one). Execution is globally synchronous, so all multiplexers move from one slot to the next at the same time. For instance, when in slot $s = 2$, the multiplexer o_2 (East) of router R_1 will accept data that came in the previous slot from i_1 (North). Our figure pictures the case where all multiplexers are on

slot 2. The labelled black bullets represent data that arrived during slot 1 and must be transmitted during slot 2. For instance, during slot 1 the output o_2 of R_2 has transmitted data belonging to c and coming from input i_3. This data, which has arrived in R_3 through i_0 will be transmitted through o_1 in slot 2. Note that the link going from R_2 to R_3 is multiplexed. In slot 1 it transmits data belonging to c, whereas in slot 2 it transmits data belonging to a.

In our example the communications of a, b, and c are of unicast type, but the TDM mechanism described above allows multicast communications.

Note that TDM arbitration is not work conserving, which means that slots may be empty even through GS data is waiting to be transmitted over the NoC. Free slots are used to transmit low-priority best-effort (BE) traffic.

2.1.4.3 Nostrum

The Nostrum NoC was developped by the Laboratory of Electronics and Computer Science (LECS) at the Royal Institute of Technology in Sweden [Millberg et al., 2004b]. The objective of Nostrum was to reduce the need for buffering resources through innovative routing and arbitration mechanisms, not replicated elsewhere.

The first choice of Nostrum is the use of *deflective routing* [Feige and Raghavan, 1992], which requires that all data arriving at a NoC router is immediately forwarded through some output link, even if the output link normally required by the packet to reach its destination is occupied. Deflective routing requires the use of an adaptive routing algorithm, but NoC routers do not need to store packet data, as packets are constantly in flight. A packet can be denied access to the NoC, but once accepted it is never blocked waiting for some other transmission to complete. Each packet has exactly one flit (so that it can be transmitted in one clock cycle).

The deflective routing mechanism alone provides no guarantees on the duration of a data transfer, and data packets may arrive at their destination in an order different from the one in which they were sent. This is why this basic service is dubbed “Best Effort” (BE). On top of it Nostrum provides a “Guaranteed Bandwidth” (GB) service designed to offer timing and packet ordering guarantees. To provide the GB service, NoC traffic is prioritized. There are two priorities: The lower one is that of BE traffic, and the higher one the GB traffic. GB traffic is organized in a set of virtual circuits. In each virtual circuit so-called *containers* [Millberg et al., 2004a] loop on a periodic basis. A container is a packet-size reservation. Once created, a container is looping arround its virtual circuit. At each cycle, it can either be used (if data is available in the source PE for transmission), or it can be left unused. The reservation made for an empty container is lost. The transmission of containers along the virtual circuits is synchronized at NoC level, so that no contention occurs between containers belonging to different circuits. The result is similar to that

obtained in Æthereal through the use of slot tables. Like in Æthereal, the construction of the virtual circuits and of their interleaving is a major problem, that must be solved through global optimization approaches.

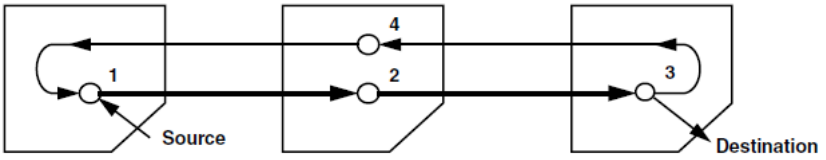


Figure 2.9: Nostrum looping container (reproduced from [Millberg et al., 2004a])

Fig. 2.9 depicts an example of container looping around a virtual circuit spanning over 3 routers: one connected to the tile sending data, one connected to the destination tile, and an intermediate one. This containter is tracked during four clock cycles. In the first cycle, the empty container arrives to the switch 1 (the GB source). The container is loaded with the GB traffic and sent off the east switch. In the second cycle, the container and its load is routed along its predefined path. In the third cycle, the container reaches its destination, the information is unloaded and the container is sent back empty, possibly, with some new information loaded. In the fourth cycle the empty container traverses the intermediate router.

2.1.4.4 Kalray MPPA NoC

The MPPA256 architecture of Kalray [Harrand and Durand, 2011, MPPA, 2012] is a tiled many-core with a 2D torus NoC. The topology of the interconnect is detailed in Fig. 2.10. This figure depicts the routers of the 16 computing tiles (the 4x4 central square) and the 16 routers connected to the various I/O devices. To facilitate description, our figure presents only the (unidirectional) links along one vertical line of tiles. The full NoC is obtained by repeating this pattern along each horizontal and vertical line of tiles.

NoC arbitration is done under a fair policy. Communications are performed along virtual channels, each (unidirectional) channel connecting one source tile with one destination tile. Setting up a channel amounts to assigning it a route and a latency/throughput budget. It is required that for each physical link of the NoC, the sum of the transmission requirements of all the virtual channels using this link is less than the transmission capacity of the link. Under this condition, all channels will provide the latency/throughput guarantees assigned to them. Ensuring that a virtual channel does not attempt to take more resources than its budget allows is realized through bandwidth management mechanisms. More precisely, each tile and I/O device in the chip contains a configurable bandwith limiter device. For each virtual channel, the respect of its latency/throughput budget is enforced by the bandwidth limiter of the source tile or I/O device.

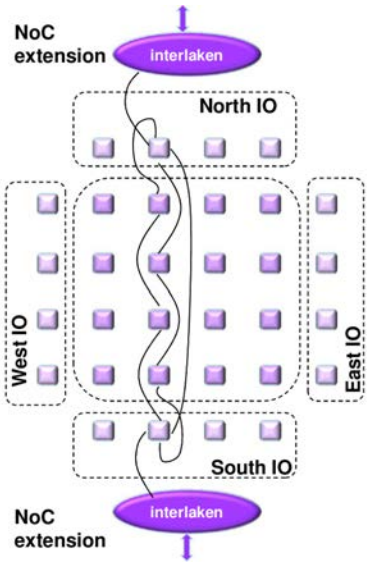


Figure 2.10: Kalray MPPA NoC architecture (reproduced from [MPPA, 2012])

To give an example of how bandwidth allocation is done, we consider the example in Fig. 2.11 (the example is borrowed from [Harrand and Durand, 2011]). The complex torus topology of the Kalray NoC complicates graphical representation. To simplify our presentation, we use in Fig. 2.11 a 2D mesh NoC topology with bidirectional links, which largely simplifies graphical representation.[3] In our example, the NoC is traversed by 5 virtual channels, whose configuration information is listed in Table 2.1. For each channel,

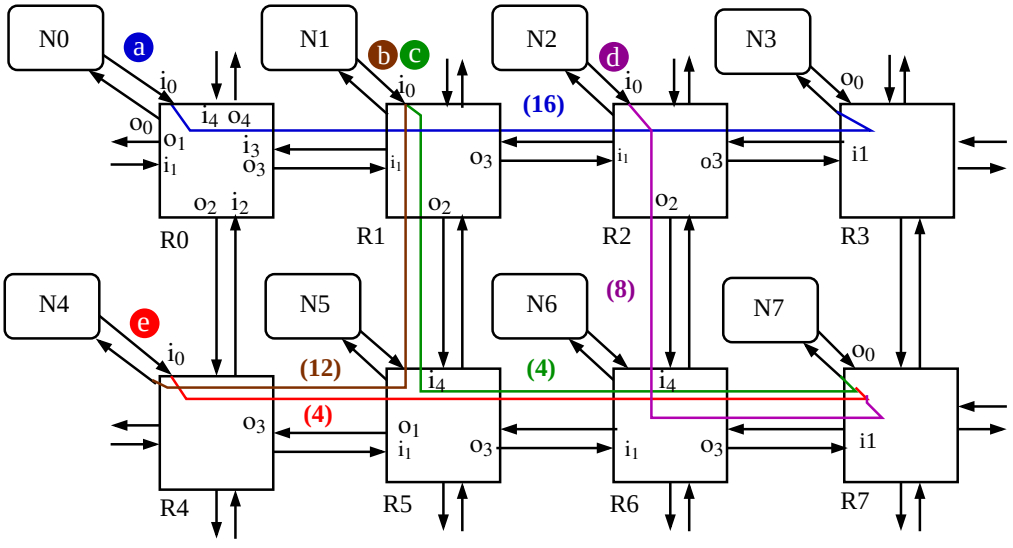


Figure 2.11: Example of throughput allocation in meshed network

this table provides its full route, including source and destination tiles, as well as the al-

[3]The creators of the Kalray NoC do the same in their patent application [Harrand and Durand, 2011].

Channel name	Source PE	Dest. PE	Route	Bandwidth allocation
a	T00	T03	00,01,02,03	16/16
b	T01	T10	10,11,10	12/16
c	T01	T13	01,11,12,13	4/16
d	T02	T13	02,12,13	8/16
e	T10	T13	10,11,12,13	4/16

Table 2.1: Virtual channels in the example of Fig. 2.11

located bandwidth, provided under the form of the transmission budget per time unit. For instance, virtual channel *a* starts in tile T00, traverses the routers of processing elements (tiles) 00, 01, 02, and 03, and transmits 16 data units per time unit. Bandwidth limiter devices are implemented with counters that only allow transmission for a fixed amount of time during each time unit. We assume that the transmission capacity of each NoC link is of 16 data units per time unit.

2.1.4.5 The scalar interconnect of MIT RAW

The NoCs we already described in this section transfer data with packet granularity, and reducing the cost of data transfers usually amounts to ensuring that data is grouped into packets that are as large as the NoC architecture and the application allow.

In this respect, the interconnect of the RAW many-core chip is very different, as data transfers are done with word granularity, hence the name *scalar operand network* given to the resulting many-core architecture. Allowing communications with word granularity enables the MPSoC-wide use of compilation techniques that exploit Instruction Level Parallelism (ILP) [Taylor et al., 2004] and a very fine grain, very efficient scheduling of computations and communications.[4]

The interconnect of RAW is also the only one among production NoCs to support programmed arbitration.[5] As pictured in Fig. 2.12, in the RAW architecture the router of each tile (labeled "Switch") contains a program memory (labeled "SMEM") allowing the storage of a single sequential communication program that enforces a pre-computed static communication order concerning all 4 connections to the neighboring routers. This program is executed by the switch processor, not figured here, which has a very simple instruction set allowing only data send operations, branching, and nops [Waingold et al., 1997].

In addition to static communication scheduling, RAW also provides a dynamic communication mechanism that is used whenever the compiler is unable to determine a precise

[4]Such techniques were initially designed to exploit the parallelism between functional units of superscalar and VLIW microprocessors [Taylor, 2003].

[5]The tiled TilePro64 many-core chip from Tilera uses this interconnect as one of the 5 on-chip networks [Tilera, 2008].

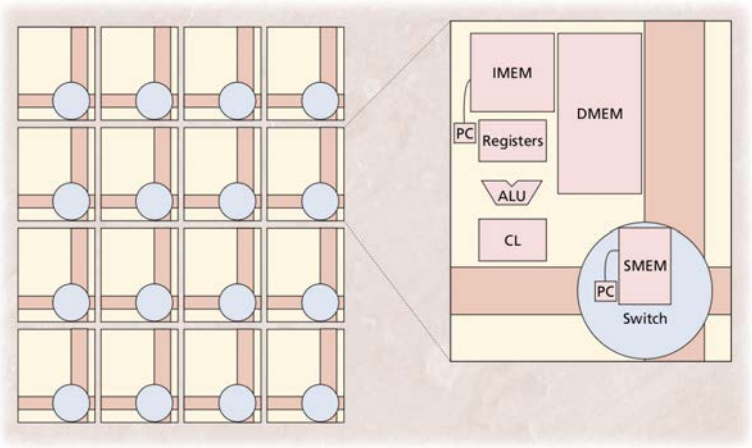


Figure 2.12: Organization of the tiled RAW many-core processor, and structure of a tile (reproduced from [Waingold et al., 1997])

static schedule of the application communications.

2.1.4.6 Other NoC architectures

This section already presented in detail a number of NoCs that were chosen to exemplify the implementation of various arbitration mechanisms in architectures for which extensive implementation documentation exists.

Of course, many other NoCs have been defined, each with its own originality points. Under an arbitration-focused point of view we have already mentioned some of these NoCs in Section 2.1.4. We briefly mention here three more NoCs whose originality does not concern arbitration/scheduling:

The ACROSS many-core platform[Salloum et al., 2013] has been designed to support the implementation of safety-critical systems, and its NoC provides mechanisms for ensuring the isolation (non-interference) between applications running on it. The NoC uses a classical TDM arbitration mechanism allowing the definition of virtual channels with fixed transmission budgets. To ensure that no unprivileged application running on a tile can alter the virtual channel communication of the NoC (and thus interfere with other applications), NoC access is controlled for each tile by a trusted hardware component, called the TISS, for Trusted Interface SubSystem. The reconfiguration of the TISS components can only be done by a secure Trusted Resource Manager (of which only one exists).

Another metric in the design of NoCs is the area footprint of the communication subsystem, and significant work has been dedicated to reducing it. We mention here only two approaches where area reduction was a key objective. First, in the design of the *asynchronous arrays of simple processors*, the communication system is reduced to registers shared between processors of neighboring tiles [Yu et al., 2008]. The result is not a NoC

in a classical sense, being even simpler than the interconnect of RAW. The second approach is a full-fledged NoC providing high-level communication services and real-time guarantees through the use of a TDM arbitration approach [Sørensen et al., 2012].

2.1.4.7 Comparison with our work

In RAW, the objective is to allow the MPSoC-wide use of compilation techniques that exploit Instruction Level Parallelism [Taylor et al., 2004] and a very fine grain, very efficient scheduling of computations and communications. The main difference in our case is that we aim for a coarser level of control in both the NoC hardware (transmission of packets instead of mere scalar values), and the software control of the NoC (which is performed through components such as cache controllers and DMA units). While losing in NoC routing flexibility and timing precision, our approach allows the use of a classical programming model, general-purpose development tools, and existing applications (like in DSPIN-based platforms). It also reduces the complexity of NoC programs.

Our intent of allowing NoC resource reservations to improve temporal (or other) properties parallels that of existing work on NoC architectures based on TDM arbitration or bandwidth reservation mechanisms. The main difference we see here is that previous work use TDM arbitration and bandwith reservation as just a way of providing *quality of service (QoS)*-like guarantees such as fixed throughput and latency. Our NoC, based on programmed communication, allows for allocating the NoC resources in a way that is synchronized with the fine-grained application needs. We allow communications to start as soon as the data to be sent is computed, and we can grant the transmission the exclusive use of all communication resources along its path for a fixed time duration.

2.2 Massively parallel processor arrays

While the previous section reviewed existing NoC architectures, we focus now on the structure of a full many-core, of which the NoC is only one component. We start by reviewing existing industrial MPPA chips and MPPA-like platforms. Then, we present a few research architectures that exhibit different compromises between performance and predictability.

2.2.1 Tilera TILE*Pro*64

The TILE*Pro*64 ™ [Tilera, 2008] is the second generation of many-core processors produced by Tilera, the company created by the conceptors of the RAW processors. As pictured in Fig. 2.13, it contains 64 processing cores, organized into as many identical computing tiles that are arranged in an 8×8 two-dimensional (2D) array. The tiles are

interconnected and connected to the I/O devices via 6 independent NoCs.

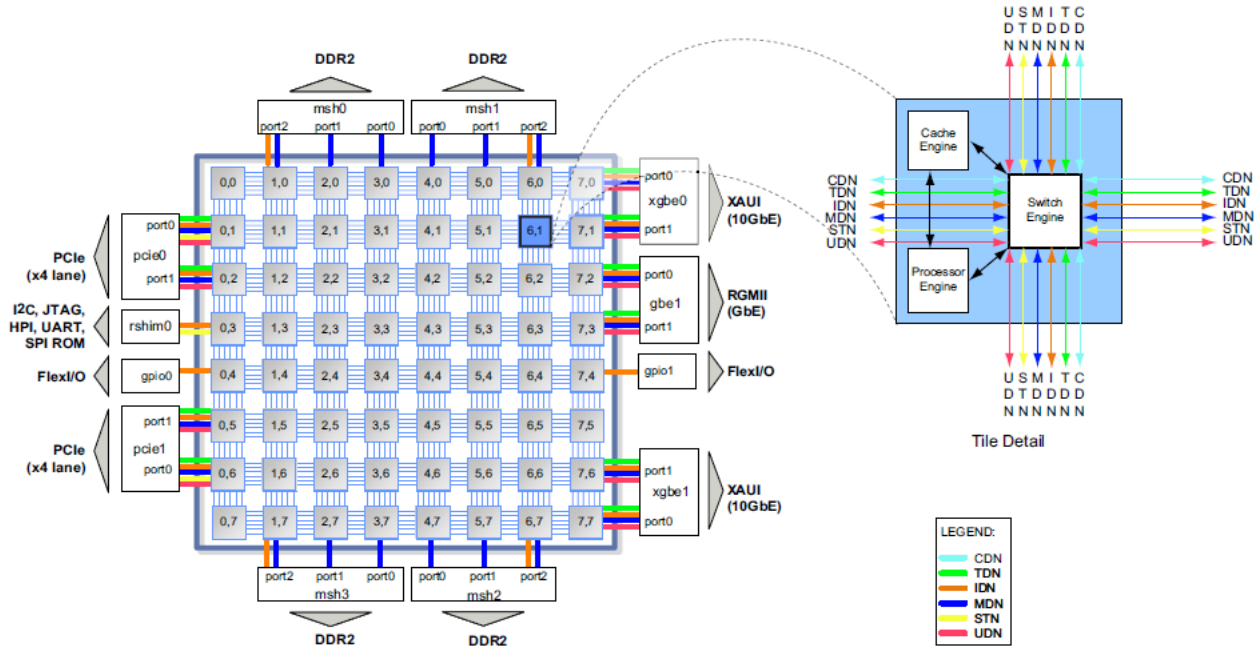


Figure 2.13: The 64-core TILE*Pro*64™ Tile processor (reproduced from [Tilera Corporation, 2013]).

Each tile is formed of three main components labeled *processor engine*, *cache engine* and *switch engine*, which are interconnected through registers.

- The processor engine is a conventional very long instruction word (VLIW) processor [Fisher, 1983] with three instructions per instruction word and full memory management.
- The cache engine contains the tile's translation lookaside buffers (TLBs), caches, and cache sequencers. Each tile has 16KB of L1 instruction cache, 8KB of L1 data cache, and a 64KB unified, 4-way set associative L2 cache. The L2 caches can be coherently shared among tiles, and thus the set of L2 caches can be viewed as a large L3 cache. A non-coherent and non-cached memory access mode is also supported.

  The cache engine also contains a *DMA engine* that is responsible for orchestrating memory data streaming between tiles and external memory, and among the tiles.
- The switch engine performs routing and arbitration for the 6 NoCs, as discussed below.

On-chip communication is performed through 6 independent NoCs with 2D mesh topology. Of these, 3 are exposed to the programmer through specific APIs that allow the definition of application-level streaming and messaging:

- The Static Network (STN) is an implementation of the programmable MIT RAW interconnect presented above. It switches scalar data between tiles with very low latency.
- The User Dynamic Network (UDN) uses a fair arbitration algorithm. Most user-defined inter-tile data transfers are expected to use this NoC.
- The I/O Dynamic Network (IDN) is used primarily for transfers between I/O devices and tiles, and between I/O devices and memory.

The remaining 3 NoCs are only used by specific hardware devices and cannot be accessed otherwise.

- The Memory Dynamic Network (MDN) is used for memory data transfers between the tiles and the 4 memory controllers of the chip. Only the Cache Engine has a direct hardware connection to the MDN.
- The Tile Dynamic Network (TDN) is also dedicated to memory traffic. It is used for transferring data between the caches of the tiles. Again, only the Cache Engine has a direct hardware connection to the TDN.
- The Coherence Dynamic Network (CDN) is also dedicated to memory traffic. It only carries cache-coherence invalidate messages between the tiles.

All NoCs with the exception of STN use wormhole packet switching, a static routing policy (X-first or Y-first), and fair (round robin) arbitration [Tilera Corporation, 2013].

2.2.2 Kalray MPPA-256

The Kalray MPPA-256 [MPPA, 2012] chip addresses both the high performance and embedded markets. As such it has good energy efficiency and provides support for hard real-time implementation in both the NoC (through bandwidth reservation mechanisms) and the computing tiles (described below).

The chip integrates 288 cores, of which 272 are grouped into 16 computing tiles arranged into a 4×4 2D array. The remaining 16 cores are evenly distributed among the 4 I/O sub-systems, placed on the 4 sides, like in the TILE*Pro*64 architecture.

The tiles and the I/O sub-systems are inter-connected through two NoCs, one for data (D-NoC), and the other for control (C-NoC). Both NoCs have the topology and arbitration mechanisms presented in Section 2.1.4.4. The only differences between C-NoC and D-NoC concern the amount of buffering in NoC routers and the way NoC traffic is generated from the tiles. The D-NoC is dedicated to high bandwidth data transfers. It implements the QoS mechanisms described in Section 2.1.4.4. The C-NoC is dedicated to peripheral

D-NoC flow control, to power management, and to application software messages. Given the relatively small amount of traffic, no QoS mechanism is used.

The structure of a computing tile is provided in Fig. 2.14. Each tile contains 17 CPU cores. Of these, 16 are dedicated to data computations, and are labeled with $C0 \ldots C15$. The 17^{th} processor is referred to as the *system core* and performs only resource management tasks, such as scheduling computations onto the other 16 cores and driving tile I/O. Each computing core has its own separate instruction and data caches (2-way set associative, 8kbytes each). Cache coherency is not enforced in hardware, and the cores themselves are timing compositional in the sense of [Wilhelm et al., 2009].

In addition to the computing cores, the tile contains 2Mbytes of shared memory distributed in 16 banks of 128kbytes each. Memory access can be configured to be either interleaved, or not. In interleaved access, successive memory addresses are placed onto different memory banks. More precisely, the memory word of address n^{th} will be placed on memory bank n **mod** 16. Such a memory access technique ensures a good spread of memory accesses over the memory banks in the absence of contention-minimizing memory allocation techniques, thus reducing the worst-case cost of contentions. In non-interleaved access, each memory bank is assigned a full range of addresses. This facilitates the explicit allocation of variables to memory banks, which can also be used to minimize the number of memory access contentions.

The MPPA chip also features a DMA unit, a synchronization unit allowing the definition of synchronization barriers, and network interfaces for connecting to the NoCs. The processors, memory banks, DMAs, *etc.* are interconnected through a crossbar local interconnect. To minimize the size of the crossbar, the processing CPU cores are grouped by 2 using fair arbitration.

Each of the four I/O subsystems contains a traditional 4-core symmetric multi-processor (SMP) with its own cache, static memory, and external DDR access. They operate controllers for the PCIe, Ethernet, Interlaken, and other I/O devices, and are meant to execute an operating system such as Linux or a real-time OS.

2.2.3 Adapteva Epiphany

The Epiphany many-core chip from Adapteva [Adapteva, 2012] makes the choice of simplicity, power efficiency, and extensibility. It comes in two sizes, featuring either 16 or 64 microprocessor cores. The cores are distributed in as many computing tiles which are arranged in a 4×4 or 8×8 2D array. In both cases, the interface of the chips allows the tiling of multiple Epiphany chips into 2D meshes which provide the equivalent of larger tile arrays.

In addition to its processor, each tile contains 4 memory banks for a total of 32kbytes of program and data RAM, DMA and interrupt units, and the NoC interface. Memory is

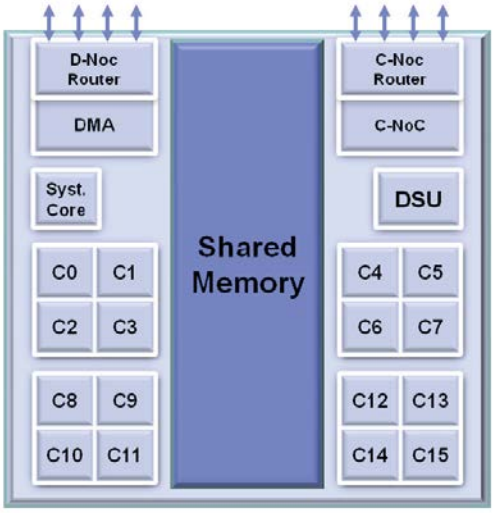


Figure 2.14: MPPA-256 computing tile (reproduced from [de Dinechin et al., 2013])

organized under a distributed shared memory paradigm where each processor can access the RAM banks of all tiles. Processor cores are cacheless. Local tile interconnect is of full crossbar type [Epiphany, 2012].

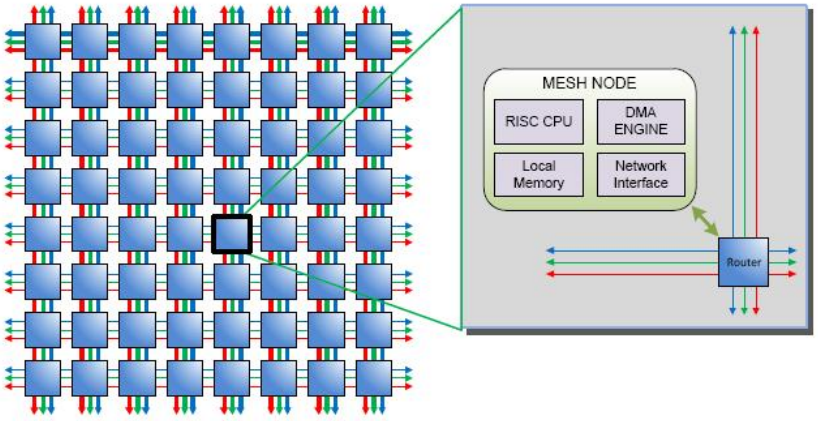


Figure 2.15: The Epiphany architecture (reproduced from [Epiphany, 2012])

The tiles are interconnected and connected with the chip I/O through 3 NoCs labeled *cMesh*, *rMesh*, and xMesh. The *cMesh* and *rMesh* NoCs have a classical 2D mesh topology and follow a static X-first or Y-first routing algorithm and fair (round robin) arbitration. The *xMesh* NoC only allows a packet to travel in one direction (up, down, left, or right) without changing direction. The *cMesh* NoC has a high bandwidth and is used for write transactions between computing tiles. The *rMesh* NoC has lower bandwidth is used for read transactions between tiles or between one tile and the exterior of the Epiphany

chip. The *xMesh* NoC is used for write transactions involving the exterior of the Epiphany chip. The xMesh and rMesh NoCs allow an array of Epiphany chips to be connected in a mesh structure without glue logic.

2.2.4 Intel SCC

The experimental Single-chip Cloud Computer (SCC) many-core chip [Howard and al., 2011] from Intel was the result of a very different trade-off between power efficiency and computing power. In this architecture, the processor cores are feature-rich Pentium core variants, with their IA-32 instruction set slightly enhanced to improve software control over cache coherency. The difficulty is that of providing the interconnect (a NoC) and the power management mechanisms (of dynamic voltage and frequency scaling type) allowing a less power-hungry implementation.

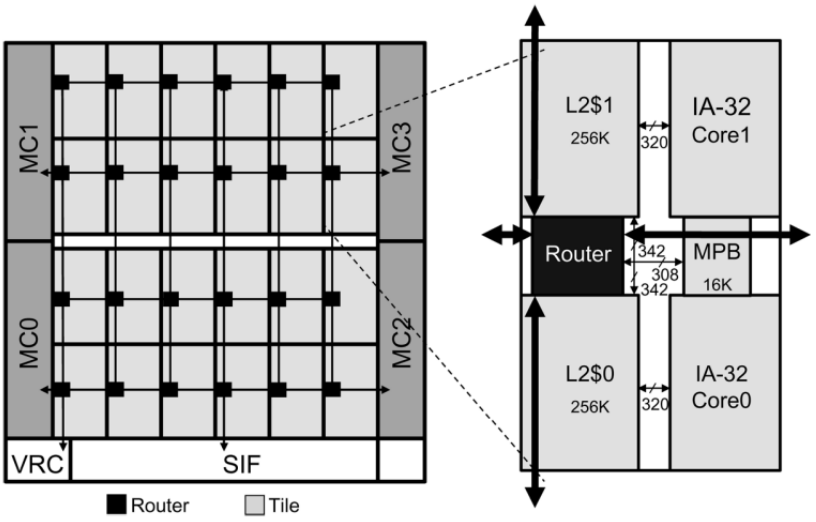


Figure 2.16: The SCC full-chip and tile architecture (reproduced from [Howard and al., 2011])

Each SCC chip intergrates 48 IA-32 cores and the innovations in power management allow it to attain power consumptions 125W to as low as 25W. The 48 cores are arranged in a 6×4 on-die mesh of tiles with two cores per tile. L1 instruction and data caches of each core have been upsized to 16KB, support 4-way set associativity, and both write-through and write-back modes. As pictured in Fig. 2.16, each core also has a unified L2 cache of 256kbytes.

At the periphery of the SCC chip, four DDR3 memory channels connected to the NoC provide access to up to 64 GB of system memory. Additionally, an 8-byte bidirectional high speed I/O interface is used for all off-die communication.

Communication between computing cores is realized through an approach that mixes shared memory and message passing mechanisms. The basic communication mechanism follows a traditional shared memory model where all communications are done through reads and writes of the main memory. However, communications of less than 16 kbytes can be transmitted directly from the L1 cache of the sender processor core to the L1 cache of the destination processor.

To limit complexity and power consumption, the SCC removes hardware cache/memory coherency mechanisms, and in particular hardware-managed coherency traffic over the NoC. Instead, the user must ensure coherency management using different synchronization strategies.

The NoC has a 2D torus topology with bidirectional links between routers. It uses static X-first routing and a virtual cut-through switching technique. The NoC routers feature 8 virtual channels used to implement 8 priority levels among packets traversing the NoC. However, these priority levels are not user-defined. Instead, their allocation follows a scheme [Tamir and Chi, 1993] meant to improve communication fairness at chip level with respect to the more traditional router-level round robin arbitration.

2.2.5 ST Microelectronics STHORM

STHORM [Melpignano et al., 2012], initially called "Platform 2012" or P2012, was not designed as a many-core chip, but rather as a *platform* for the design of embedded many-cores for multi-media applications. It features a tiled organization, but the contents and size of each tile can be varied depending on the design needs. The platform was designed by ST Microelectronics and the CEA.

To improve power efficiency and facilitate power management, tiles are connected via a fully asynchronous NoC called ANoC, designed by the CEA-Léti [Thonnart et al., 2010]. The NoC uses a source routing algorithm and provides QoS control under the form of 2 packet priority levels.

A typical computing tile, as used in the test chip produced by ST Microelectronics, consists in 17 processing cores (of which 1 is dedicated to resource allocation/scheduling), several memory banks, power management, DMA, and synchronization units, and a special unit with support for data-flow programming.

2.2.6 TSAR

TSAR (Tera Scale ARchitecture) [TSAR, 2008] is an academic MPPA architecture designed by a team lead by A. Greiner. In its design, the main objectives were scalability, efficiency, and facility of programming. By comparison with a chip like Kalray MPPA, where the need for temporal predictability requires that memory hierarchy is simple and

exposed to the programmer and that no cache coherency is used, TSAR takes the opposite choices. It uses a complex memory hierarchy, but hides it as much as possible and implements in hardware the cache coherency protocols.

TSAR integrates up to 1024 computing tiles interconnected by the DSPIN NoC presented in the previous section. Each computing tile contains up to 4 processors cores, each with its own program and data L1 caches. The tiles contain no local RAM banks. Instead, each tile contains one *memory cache* unit that acts as an L2 cache with cache coherency manager. The global address space is partitioned between the tiles, each tile being associated a range in the global space. The memory cache of a tile will only cache addresses in this range. Requests from the local processors for addresses not in this range are routed through the NoCs towards the tile that owns the address. Requests for addresses in the range of the tile pass through the memory cache and (in case of a miss) are routed through another level of cache (L3) towards the main memory through a NoC, called the RAM network, which has a specific tree topology. Inter-tile traffic are transferred through 2 other NoCs of DSPIN type, one dedicated to memory access traffic, and the other to cache coherency traffic.

Application mapping onto TSAR is managed by an operating system that was developed especially for TSAR: ALMOS [Almaless and Wajsbürt, 2012], for Advanced Locality Management Operating System. ALMOS aims at hiding as much architectural detail as possible from user applications, but while preserving efficiency. To this end, it uses a client/server scheduler design allowing the kernel to offer scalable inter-thread synchronization mechanisms. Moreover, it implements a kernel-level affinity technique named *Auto-Next-Touch* allowing the kernel to transparently and automatically migrate physical memory pages in order to enforce the locality of thread's memory accesses.

2.2.7 Academic MPSoC architectures with TDM-based NoC arbitration

Various research projects such as T-Crest, ACROSS, or CompSoC have recently proposed NoC and many-core architectures with strong support for real-time embedded implementation. Most of them [Brandner and Schoeberl, 2012, Salloum et al., 2013, Goossens et al., 2012] rely on time division multiplexing (TDM) NoC arbitration, as described in Section 2.1.4.2.

2.3 Static application mapping

In the previous sections we have provided a comprehensive classification of existing work on NoC and MPPA design from a hardware architecture point of view. This was made possible by the fact that relatively few teams have endeavored to fully define such hardware architectures, and that among this set of architectures we could identify a smaller

representative subset. But such a comprehensive presentation is no longer possible for mapping techniques. Indeed, in the past few years virtually every mapping (allocation and scheduling) technique proposed in at least 3 research fields (compilation, real time scheduling, parallel programming) has been adapted for multi-core or many-core architectural targets.

We therefore needed to limit the scope of our state of the art presentation. We chose to include in it:

- A detailed presentation of works that have been direct inspiration sources in our developments.
- A brief presentation of other mapping approaches where all the elements of the flow are tailored to fit a given scheduling paradigm (just like we do).

We have already explained in Section 2.1.3.3 that our work aims at reaching the best possible timing precision and predictability in the mapping of parallelized versions of embedded control applications. Such applications are best described as dependent task systems, and best modeled using deterministic data-flow specification languages. Two large classes of such data-flow formalisms exist:

- The dataflow synchronous languages, such as Scade/Lustre/Heptagon [Heptagon, 2013, Halbwachs et al., 1991], Signal [Guernic et al., 2003], or SynDEx [Grandpierre and Sorel, 2003].
- The synchronous dataflow (SDF, CSDF) formalisms [Lee and Messerschmitt, 1987, Parks et al., 1995].

For both classes, extensive previous work covers their mapping onto multi-processor architectures [Grandpierre and Sorel, 2003, Fohler and Ramamritham, 1995], and much of this work has concentrated on defining off-line mapping techniques. Indeed, these formalisms facilitate the programming of deterministic and regular applications, which in turn facilitates the construction of potentially optimal scheduling tables defining the allocation and real-time scheduling of the various computations and communications onto the hardware resources.

This thesis has two objectives:

- To modify the hardware platform so that it provides better support to the implementation of static computation and communication schedules.
- To define novel algorithms for the construction of static computation and communication schedules for the new architectures.

In doing this, our work has drawn significant influence from two lines of existing work:

- Off-line distributed real-time mapping (allocation and scheduling) of data-flow applications, and in particular work on the AAA/SynDEx methodology [Caspi et al., 2003, Grandpierre and Sorel, 2003, Potop-Butucaru et al., 2009].

- Compilation for MIMD microprocessor architectures such as the VLIW processors [Fisher, 1983] or the *scalar operand networks* presented in Section 2.1.4.5 [Lee et al., 1998, Amarasinghe et al., 2005].

2.3.1 Off-line real-time multi-processor scheduling

Our work is closely related to classical results on the off-line real-time mapping of dependent task systems onto multiprocessor and distributed architectures [Eles et al., 2000, Fohler and Ramamritham, 1995, Xu, 1993, Grandpierre and Sorel, 2003]. The objective is the same as in our case: the synthesis of optimized scheduling tables defining time-triggered execution patterns. But there are also significant differences. As explained in the previous section, most NoCs rely on wormhole routing, which requires synchronized reservation of the resources along the communication paths. By comparison, the cited papers either use a store-and-forward routing paradigm [Fohler and Ramamritham, 1995, Grandpierre and Sorel, 2003, Eles et al., 2000] that is inapplicable to NoCs, or simply do not model the communication media [Xu, 1993].

A second difference concerns the complexity of the architecture description. MPPAs have more computation and communication resources than typical distributed architectures considered in classical real-time. Moreover, resource characterization has clock cycle precision. To scale up without losing timing precision, we need to employ scheduling heuristics of very low computational complexity, avoiding the use of backtracking [Fohler and Ramamritham, 1995], but taking advantage of low-level architectural detail concerning the NoC.

A third difference concerns fine-grain resource allocation and scheduling operations, such as the allocation of data variables into memory banks or the scheduling of DMA commands onto the processor cores. These operations are often overlooked in classical distributed scheduling because they are usually delegated to operating systems. But an OS introduces timing overheads that are significant at the precision level of our algorithms. (hundreds or thousands of clock cycles), and which can have a major impact on execution durations on an MPPA platform. This why we need to explicitly consider them at off-line mapping time.

2.3.1.1 The AAA/SynDEx methodology

SynDEx [AOSTE-INRIA,] is a system level CAD software based on the algorithm architecture adequation (AAA) methodology [Sorel, 1994] for rapid prototyping and optimized

real-time implementation of embedded control applications onto architectures with multiple CPUs and communication lines (buses or shared RAMs). It has been designed and developed in the INRIA Paris-Rocquencourt Research Center France, by a team led by Yves Sorel.

Fig. 2.17 describes the flow of the AAA methodology and of the SynDEx tool. SynDEx takes as input a functional specification, an architecture specification, and some non-functional (allocation, timing) requirements. It performs two tasks:

- The off-line real-time scheduling of the functional specification on the architecture under the given non-functional constraints.
- The generation of correct-by-construction C code implementing the computed off-line schedule.

The functional specification is written in a high level synchronous data flow language similar to Scade/Lustre [Halbwachs et al., 1991], and importers exist for other synchronous languages such as Signal [Guernic et al., 2003] or Scicos [Campbell et al., 2006].[6] The specification of the architecture (hardware and basic software) is provided under the form of a bipartite graph whose nodes are the processing elements (CPUs, accelerators) and the communication media (buses, shared RAM banks). The arcs of the graph describe the connections of processing elements to the communication media. The non-functional constraints allow the specification of worst-case durations for the various computation and communication operations over the various processors and communication media, as well as allocation constraints.

Starting from these 3 inputs, SynDEx produce a *reservation table* describing the real-time execution of one cycle of the synchronous functional specification. To do so, it uses greedy heuristics that perform at the same time the allocation and scheduling of the various computations and communications. The various computation and communication operations are mapped one by one, and once a decision is taken for one operation it is never changed (there is no backtracking).

For each computation and communication operations, the resulting reservation table specifies the resource (processor or bus) that will perform it, the starting date, and the time duration starting from the start date where the resource is (exclusively) dedicated to the operation. To cover conditional behaviors, each reservation can also have an *execution condition*.[7] The same resource can be allocated at the same date to operations with exclusive execution conditions.

In a second phase, the reservation table is translated into executable code where the

[6] We do not cover in this document the multi-periodic extensions of SynDEx's specification language.

[7] Also called *clock* in the jargon of synchronous languages.

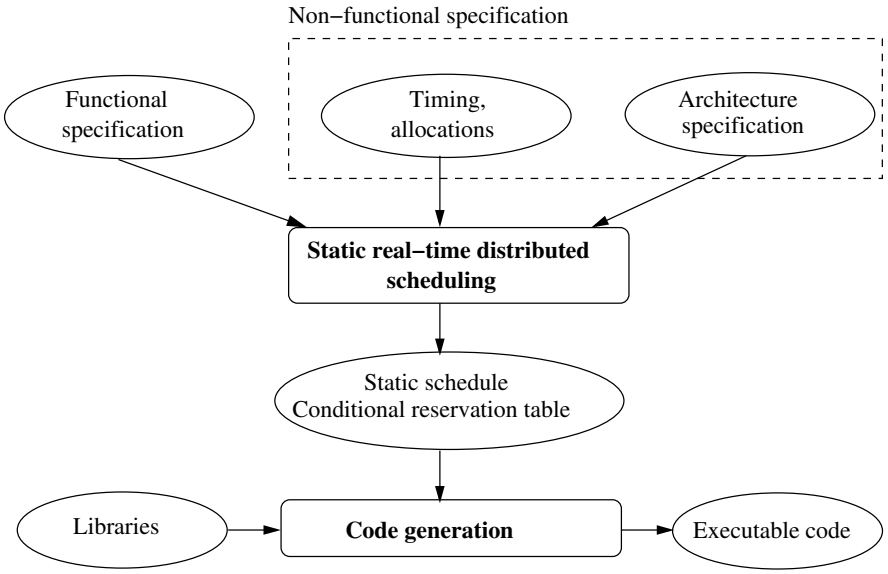


Figure 2.17: The global flow of the AAA methodology and the SynDEx software

primitives of the execution platform are used to implement the needed timing and reservation mechanisms.

The work on the AAA methodology and SynDEx has been extended in various ways. Of particular interest for this thesis are previous works on improving the handling of execution conditions. Indeed, SynDEx requires, at both specification and algorithmic level, that all communications between a component and its environment to take place at the same speed (each input and output is read or written at each activation of the component). This may result in redundant communications slowing down an application.

To overcome this limitation, a new formal model, called *Clocked Graphs (CG)* has been proposed in [Potop-Butucaru et al., 2009]. On one hand, this format allows the faithful representation of specifications written in high-level synchronous languages such as Scade/Lustre, Signal, or discrete Scicos (in particular, without the constraints imposed by SynDEx). On the other hand, CG specifications are close enough to the target machine code to allow fine manipulations such as scheduling, allocation, and optimization (e.g. software pipelining [Carle and Potop-Butucaru, 2011]). We give a detailed description of this formalism and the associated scheduling algorithms in Section 5.1.

Our work in this thesis is completely subsumed to the AAA methodology (and to the closely-related *platform-based design* paradigm [Sangiovanni-vincentelli and Martin, 2001]). It directly builds upon the toolset based on the Clocked Graphs formalism. Compared with previous work on the AAA methodology [Grandpierre and Sorel, 2003], our work shares the objective of real-time mapping of dataflow synchronous specifications onto multi-processor platforms and the use of greedy heuristics without backtracking. The originality of our approach is given by:

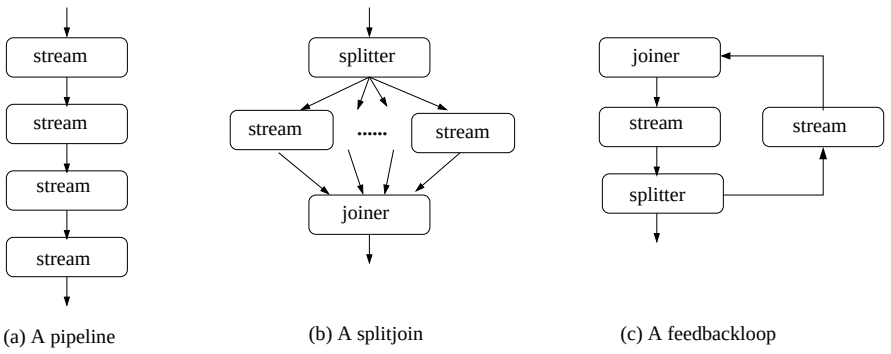
- The handling of complex architectural components, such as the NoC and the multi-bank RAMs, which require complex manipulations on a large number of resources. Scaling up to deal with large number of resources is only possible by exploiting the homogeneity and regularity of MPPA architectures.
- The use of code optimizations, such as *software pipelining*, in the presence of conditional execution, as explained in [Carle and Potop-Butucaru, 2011].
- The use of (pre-computed) preemption for NoC communications.

2.3.2 The StreamIt compiler for the MIT RAW architecture

The second main source of inspiration in the definition of our mapping technique is previous work on mapping StreamIt programs onto the MIT RAW architecture presented above [Waingold et al., 1997]. Like in AAA/SynDEx, StreamIt mapping is realized using table-based static scheduling techniques. But there are also significant differences. First of all, the StreamIt input language belongs to a different class of data-flow formalisms. Its execution model is similar to that of synchronous data flow (SDF) [Goossens et al., 2012], but:

- It includes features facilitating the programming of real-life streaming applications (*e.g.* hierarchy, peek operations, *etc.*).
- The topology of the data-flow graphs is significantly restricted. More precisely, the construction of the graph is done through hierarchic composition by employing the 3 elementary combinators pictured in Fig. 2.18: *pipeline*, *splitjoin*, *feedbackloop*.

Along with constraints on the form of elementary data-flow blocks (known as *filters*), these restrictions allow the real-life programming of complex applications and at the same time facilitate their compilation into executable code.



(a) A pipeline (b) A splitjoin (c) A feedbackloop

Figure 2.18: StreamIt data-flow combinators (*cf.* [Gordon, 2010]).

The flow of the StreamIt compiler is pictured in Fig. 2.19. It involves 3 separate steps:

- *Coarsening the granularity of the data flow graph*. To reduce the complexity of subsequent transformations, state-less data-flow blocks are fused together into larger blocks.

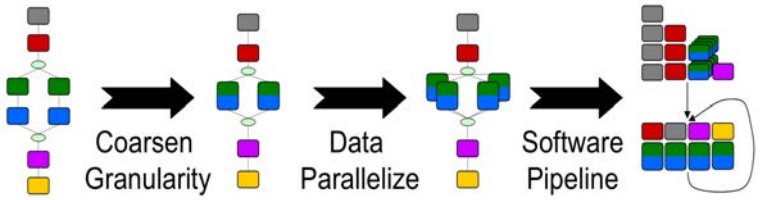


Figure 2.19: The flow of the StreamIt compiler (reproduced from the PhD presentation of [Gordon, 2010])

- *Data parallelization*: In this phase, data-flow blocks are data parallelized using a so-called *judicious fission* algorithm [Gordon et al., 2006] that uses load balancing arguments in order to decide how much data parallelism is introduced.

- *Coarse-Grained software pipelining*: The actual mapping of the coarsened and data-parallelized graph onto the RAW processor is done using classical software pipelining techniques.

The compilation flow proposed in this thesis shares with the one of StreamIt the use of scheduling tables as an internal compiler representation and the use of optimizations such as software pipelining to improve resource use. There are, however, significant differences. First, our objective is to provide worst-case hard real-time guarantees, while compilers usually aim for average-case performance. For instance, the StreamIt compiler uses incomplete timing information coming from simulations to guide the mapping process, not taking into account timing interferences due to the mapping itself. By comparison, our tool uses safe WCET/WCCT bounds and maintains throughout the mapping process a strict control of the timing interferences due to concurrent memory or NoC accesses.

The second difference is the optimization objective. Classical software pipelining algorithms attempt to optimize computation *throughput*, thus maximizing the use of the hardware resources. But for embedded control applications the objective is most often to optimize not the shere speed, but the *latency* between input acquisition and corresponding output production. Our algorithms are therefore tailored for optimizing both latency and throughput, with latency being prioritary.

Finally, our mapping tool can exploit conditional execution to improve resource allocation, whereas StreamIt cannot. Our mapping problem is therefore harder, the only mitigating factor being the coarser grain of control of our MPPA architecture: In RAW,

all inter-processor communications are done with scalar granularity. When the statically-scheduled NoC is used (to ensure temporal predictability) one routing statement must be executed for each scalar value (word) traversing a tile. By comparison, our MPPA architecture uses packetized communications, and each tile contains multiple CPUs, functioning as a classical *symmetric multiprocessor (SMP)* machine (but with a memory subsystem with better support for timing analysis).

2.3.3 Compilation of the ΣC language for the Kalray MPPA256 platform

The Σ*C* [Goubier et al., 2011] programming language has been jointly developed by CEA LIST and Kalray as a way to program Kalray's MPPA256 manycore processor.

Σ*C* belongs to the same class of data-flow formalisms as StreamIt, but is far more expressive:

- The formal model of Σ*C* is an extension of cyclo-static data-flow networks (CSDF), which is more expressive than StreamIt's SDF.
- Σ*C* imposes no constraint on the topology of the data-flow graph.

The authors of the language also mention some limited amount of data-dependent control, but it is unclear from existing papers what can be expressed and if knowledge of the data-dependent control conditions can be used to improve implementations. Nevertheless, Σ*C* provides a very rich development formalism while retaining the decidability of problems such as the absence of deadlocks and the implementability in bounded memory.

The compilation of a Σ*C* program starts, like the compilation of StreamIt, with a series of transformations aimed at adapting the parallelism grain of the application to the one of the execution platform (in this case the MPPA256 chip). This first phase is called *parallelism reduction*. On the resulting CSDF specification a series of 4 scheduling and allocation steps are then applied. The first one uses classical CSDF scheduling algorithms to determine minimal buffer sizes (associated to the data-flow arcs) ensuring the absence of deadlocks [Aubry et al., 2013]. The next step refines these buffer size bounds by taking into account required throughput constraints. The third step performs the mapping of the data-flow blocks to the computing tiles of MPPA256. Mapping is done using affinity-driven algorithms in such a way as to ensure that no tile is assigned more computations than it can perform. The final allocation step performs communication routing, ensuring that no link is charged at more than 100%. This phase defines the virtual channels needed to configure the MPPA256 NoC, and their bandwidth, needed to configure the bandwidth limiters.

Note that the scheduling phases of the compilation process are not aimed at producing static scheduling patterns, but more at dimensioning the memory allocation to provide

liveness guarantees and improve the throughput. It is important to note that mapping on MPPA256 platforms is not fully static. More precisely, allocation is static (each task is assigned its tile and each communication its route), but scheduling is not: Inside a tile, a task instance can be placed on one processor or another, depending on its execution context. Thus, the output of the scheduling process is the partial order of tasks that must be cyclically executed on each tile.

The final compilation stage is the so-called *link edition*, which generates the executable C code.

In conclusion, even though the MPPA256 architecture provides significant support for real-time implementation, the compilation process does not take advantage of it to compute or enforce application-level timing guarantees. The focus is less on scheduling and more on allocation (tasks on tiles, communications on the NoC, I/O buffers on the memory banks) in order to improve the throughput of the application.

2.3.4 Other mapping approaches

The idea of organizing a development environment and flow around a scheduling paradigm is not new. We already saw that static table-based scheduling was the basis of the RAW/StreamIt approach. We know of two other attempts.

The first one is the *CompSoC* platform [Goossens et al., 2012], which relies on a compositional scheduling and timing analysis approach where applications are assigned latency and throughput budgets on the computation and communication resources (such an approach can be generalized towards the use of full-fledged real-time calculus, like in [Bacivarov et al., 2013]). The respect of these budgets is enforced using time division multiplexing (TDM) mechanisms on the various resources, such as the NoC, but the fine-grain synchronization between these TDM mechanisms is not required, nor used during timing analysis (only the latency/throughput budgets are used). By comparison, the mapping approach proposed in this thesis allows the tight synchronization of computation and communication schedules, which improves timing precision and guaranteed performance, but requires a more static execution model than CompSoC.

The second one is based on the use of a priority-preemptive scheduling paradigm [Shi and Burns, 2010]. However, its target application class is very different from ours. Indeed, the cited paper considers the case of independent tasks, whereas our main focus is on dependent task systems.

More generally, our work is related to all previous work on application mapping onto many-core architectures, be it at application level [Bebelis et al., 2013, Genius et al., 2013, Zhai et al., 2013, Bhattacharyya et al., 2013] or aiming just some detail such as the allocation or the configuration of TDM tables [Lu and Jantsch, 2007] . In all these cases, the main difference with respect to our approach is given by the integrated approach we use

and by the statically scheduled NoC communications which ensure high timing precision and efficiency for the chosen class of applications.

Chapter 3

Tiled MPPA architectures in SoCLib

Contents

3.1 MPPA structure 57
3.2 Memory organization 59
3.2.1 Distributed shared memory 59
3.2.2 Address structure 60
3.2.3 Global memory organization 61
3.2.4 Tile memory organization 62
3.2.5 Hardware/software interface 63
3.3 Improving the timing predictability of the SoCLib tile 64
3.4 SystemC simulation and compilation support 68

This chapter reviews the MPPA architecture we used as basis for the developments of this thesis, as well as the modifications we brought to its computing tiles to improve timing predictability. We provide here fine technical detail of the platform which will be needed in the following sections. The complexity of the platform, apparent in this chapter, explains in part why hardware design, hardware configuration, and software mapping problems are so complex.

3.1 MPPA structure

We are using a tiled many-core architecture built using the components of the SoCLib hardware library [LIP6, 2011]. The central element of this architecture is the DSPIN NoC [Panades, 2008], which links together the computing tiles. In our MPPA, all inter-tile communication and synchronization are realized through the DSPIN NoC. In particular, no complex memory hierarchy exists to allow communication and synchronization

through shared caches. Thus, we avoid the complex problem of ensuring timing predictability in the presence of such shared caches [Wilhelm and Reineke, 2012], [Hardy and Puaut, 2008].

As pictured in Fig. 3.1, SoCLib allows the construction of many-core architectures where the computing tiles are organized in a 2D array and interconnected by two DSPIN NoCs. Like in all MPPA architectures, the tiles are largely identical. As pictured in Fig. 3.2, each tile may contain:

- A set of processor cores with the associated instruction and data L1 caches. SoCLib provides a variety of CPU cores, including MIPS32, PPC405, ARM7 and ARM9.
- RAM and ROM memory banks.
- DMA controllers.
- Programmable interrupt units which allow both the generation of interrupts upon command, or the programming of timers.
- I/O units, hardware locks units, *etc.*

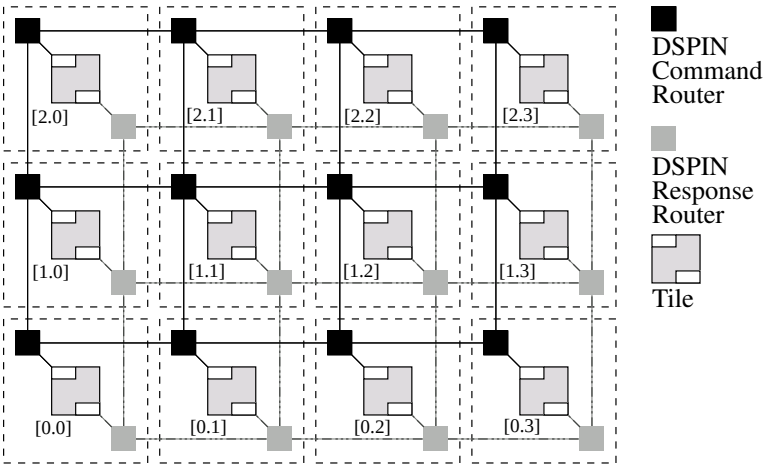


Figure 3.1: A typical DSPIN NoC based MPPA architecture. Dark rectangles are the routers of the command sub-network. Light rectangles are the routers of the response network

All tile components are connected to a *local interconnect*, which is linked to the DSPIN NoCs through a Network Interface Controller (NIC). SoCLib provides a choice of several topologies for the local interconnect. Among them, full crossbar and ring.

In the original MPSoC architecture proposed by SoCLib each tile contains one RAM bank and at most 4 CPU cores, as experimentations showed that placing more than 4 CPUs per tile results in no performance gain due to memory bandwidth saturation.

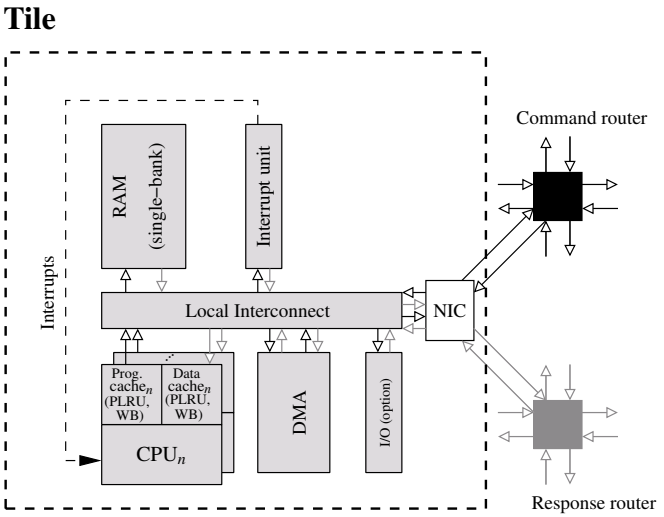


Figure 3.2: The computing tile structure in the original SoCLib many-core

3.2 Memory organization

3.2.1 Distributed shared memory

SoCLib-based tiled many-cores follow a *distributed shared memory* model where all memory banks are assigned addresses in a single address space. To this shared memory space are also mapped the programming registers of all peripheral devices, allowing a uniform programming paradigm and the use of general-purpose development tools (the GNU compiler suite, in our case).

The implementation of the distributed shared memory is based on the VCI/OCP protocol [Alliance, 2001]. This protocol follows a master/slave model. Masters are called VCI initiators, and slaves are called VCI targets. A CPU is a typical VCI initiator, while a RAM is a typical VCI target. Some components act as both VCI initiators and VCI targets. For instance, a DMA acts as a target to let the CPU write its programming registers, but acts as a master when reading data from one RAM bank and writing it to another.

The VCI/OCP protocol organizes on-chip communications into *transactions*. A transaction takes place between a VCI initiator and a VCI target, which exchange *packets* through one or more layers of interconnect, according to the locations of the initiator and

target. A transaction consists of a command packet issued by the VCI initiator and going all the way to the target, followed by the corresponding response packet emitted by the VCI target. The commands and the responses are transmitted through distinct input and output ports and through two distinct physical networks (the command sub-network and the response sub-network). This helps avoiding deadlock conditions between commands and responses and allows transaction pipelining. This explains the presence of two DSPIN NoCs in our architecture. The local interconnect of each tile is also formed of two separate networks.

3.2.2 Address structure

Routing of commands along the two-level interconnect formed of the command DSPIN NoC and the command local interconnect is done according to static routing mechanisms. The same is true for the routing of responses along the response interconnect. To this end, all VCI targets and initiators are assigned fixed addresses. One component may have multiple VCI source and target addresses. For each address it must have a separate interface with the other components.

To facilitate the definition of the static routing patterns and reduce the complexity of the routing hardware, VCI target and initiator addresses used in the SoCLib-based MPPA architecture have a particular structure, where the most significant bits identify the destination tile inside the NoC (through its Y and X coordinates) and the least significant bits form the so-called *local address* identifying the destination VCI target or initiator inside the tile. The number of bits dedicated to the encoding of the Y, X, and local address for both the command and response networks are parameters of the SoCLib platform which must be fixed during configuration.

Figure 3.3: Memory address decoding

Each hardware component acting as a VCI target is assigned a contiguous block of memory addresses of the global address space. All memory accesses targeting addresses in this block will be routed to and handled by the associated VCI target component. To simplify the association between VCI target addresses and memory addresses, it is required that the most significant bits of a memory address are the VCI address. Therefore, memory addresses have the structure of Fig. 3.3.

Addresses are encoded on 32 bits. The most significant 8 bits are dedicated to the encoding of the Y and X coordinates of the target tile (4 bits for each coordinate). This should allow the construction of NoCs of at most 16x16 tiles, but in practice only 16x15 tiles can be used due to reserved address blocks (defined next). Both the address size and the number of bits allocated to tile coordinates are set by configuration, but they will not change throughout this thesis. The Y and X coordinates are known together as the *global index*, and are decoded and used by the NoC routers and NICs.

The number of bits allocated to the local target address depends on the number of VCI targets, and in particular memory banks, of each tile, which depends itself on configuration, as explained below. The local target address, also known as the *local index*, is decoded by the local interconnect of the target tile to route the command packet to the proper target. If `T` is the number of target components in the tile, then the minimal number of bits that must be assigned to the local index is $\lceil \log_2(T) \rceil$.

The remaining bits of the address are used to encode the offset inside the memory block associated to the VCI target. This limits the number of memory addresses inside a VCI target to 2^O, where O is the number of bits used in addresses to encode the offset.

3.2.3 Global memory organization

The structure of memory addresses, defined above, implies that the global address space is divided into blocks of size 2^N, where N is the number of bits allocated to the encoding of the local address and offset. For each `Y` and `X` ranging over $0..F$ one such block exists starting at address `0xYX000000` and terminating at address `0xYXFFFFFF`.

As pictured in Fig. 3.4 (left), most address blocks are uniquely assigned to the tiles according to their Y and X coordinates. Any of these address blocks can be freely accessed from any tile on the MPPA. The only exception to this rule is the top-most memory block, which starts at address `0xFF000000`. Addresses in this block are used in each tile for *private addresses* that are never accessed through the NoC. When an access to this memory range is issued in a tile, it never leaves the tile. Such addresses can be assigned to the stack RAM banks, program RAM/ROM, DMA programming registers, *etc.* The use of private addresses significantly reduces the number of unique VCI identifiers, and thus the footprint of the VCI address encoding.[1] It also helps in ensuring memory protection between different tiles.

The drawback of reserving one address block to private VCI targets is that no tile can

[1] A similar problem exists in telecommunication networks, where it lead to the use of *loopback* network interfaces.

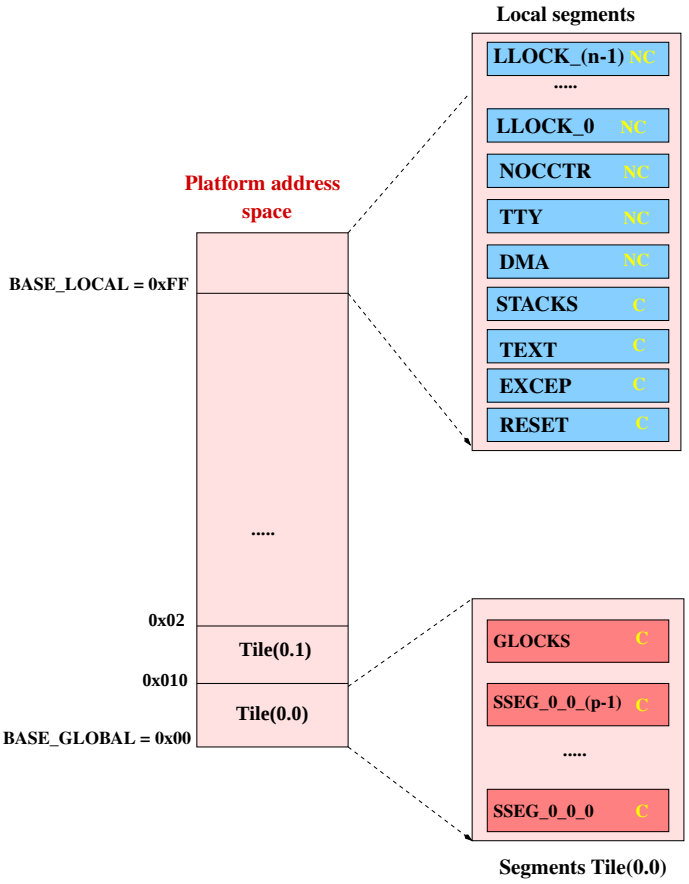


Figure 3.4: Memory organization in the modified MPPA platform (tile structure described in Section 3.3). Global address space segmentation is on the left, tile address space segmentation (public and private) on the right.

be assigned the Y/X coordinates `0xFF`. This is why in this thesis we consider MPSoCs of size 16x15, and not 16x16, and why the global address space of Fig. 3.4 contains an unused space, which corresponds to unused Y/X coordinates of the form `0xFX`, with X ranging from `0x0` to `0xE`.

3.2.4 Tile memory organization

The private and public memory blocks of each tile are allocated to the various VCI targets of the tile. We pictured on the right side of Fig. 3.4 the tile memory allocation pattern we employ throughout this thesis.

The public memory block is divided between the main data memory of the tile and the public programming registers of the hardware lock unit (such a public programming interface is needed on the lock device to allow inter-tile synchronization). To improve both performance and predictability, the main data memory is divided into multiple banks,

each being assigned a separate VCI target. However, we require that the multiple banks cover a contiguous memory address space, so that they can store data structures larger than a single bank. In turn, this requires that each memory bank stores exactly 2^O octets, where O is the number of bits used in addresses to encode the offset. Thus, the desired memory size constrains the construction of the memory and VCI target addresses. The public programming registers of the lock unit are stored on a single VCI target.

The private memory block is divided between the private tile memory banks and the private programming registers of the various tile devices. One memory bank (one VCI target) stores the program and the static program data of all the CPU cores of the tile. For each processor core, one separate memory bank (one VCI target) stores the execution stack. The remaining VCI targets contain the programming registers of the various devices described in the next section. Of particular interest is the lock unit, which uses N private VCI targets, where N is the number of processors on the tile, in order to ensure that no timing interference exists between CPU accesses to the lock unit (as explained in the next section).

The MPPA processors allow both cached and uncached memory accesses. The type of access is determined by the VCI target address. More precisely, a (configurable) cacheability mask over the local index bits determines whether an access is cacheable or not.

3.2.5 Hardware/software interface

The previous sections have discussed memory organization from a hardware point of view. But the software has another view of memory organization, and the two views (hardware and software) must be compatible.

Executable code for our platform is generated using the `gcc` compiler toolset, in the ELF executable format [ELF, 1995]. An ELF file defines the *memory segments* of the application. For specific segment types (program, read-only data, *etc.*) the ELF file also provides the content of the segment. On our MPPA platform, the ELF file contains *non-relocatable* code. Each segment of such a file is assigned a fixed memory address where the segment must be placed at execution time. Before execution can start, each segment that contains data must be *loaded* at the prescribed address.

Traditional compilation using `gcc` for an embedded single-processor platform will place all executable code in 3 segments: `.text` for the program code, `.excep` for the interrupt handling code, and `.reset` for the (re-)boot code. In our case, one set of

executable code segments is generated for each tile of the MPPA. The processors of a tile share the same program memory and program segments, but use CPU identifier tests to determine which parts of the code to execute. All 3 executable code segments of a tile are placed in the private program memory. The stack and heap of the program are set by the boot code to point to the stack memory banks associated to the various processor cores. The private data segments are also placed on these memory banks.

Enforcing the needed memory organization at compiler level is realized through the use of custom `ldscript` loader configuration files which are synthesized by our platform configuration scripts. This allows the platform, at execution/simulation time, to load the segments of the ELF file into the prescribed memory banks before starting the execution on the processor cores (as specified by a *mapping table* included in the hardware description).

3.3 Improving the timing predictability of the SoCLib tile

We have described in Section 3.1 the structure of the original MPPA computing tile, as it was provided by SoCLib. To improve both timing predictability (needed in real-time applications) and performance, we have modified the structure of the tile as described in this section. Our modifications retain the global organization of the many-core, and in particular its distributed shared memory model which allows programming using general-purpose tools. The modified tile is pictured in Fig. 3.5.

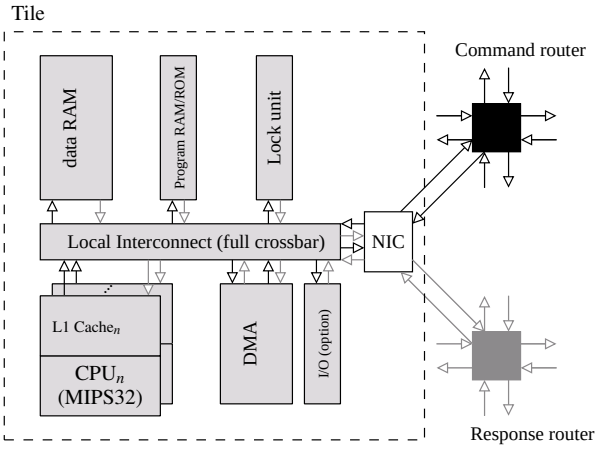


Figure 3.5: Modified computing tile structure of our architecture (with fair arbitration)

The memory sub-system: Our objective was to improve timing predictability by eliminating contentions. In our experiments with the original SoCLib-based many-core, the second most important source of contentions (after the NoC) is the access to the unique RAM bank of each tile. As a solution to this problem, we decided to follow the example of existing industrial many-core architectures [Benini, 2010, Harrand and Durand, 2011], and replace the single RAM bank of a tile with several (up to 32) memory banks that can be accessed independently.

To facilitate timing analysis, we separate data (including stack) and program memory. One RAM/ROM bank is used in each tile to store the program of all the CPUs of the tile. As explained above, data and stack are stored on several RAM banks. Each bank of the data RAM has a separate connection to the local interconnect. Explicit allocation of data onto the memory banks, along with the use of lock-based synchronization and the local interconnect presented below allow the elimination of contentions due to concurrent access to memory banks.

Note that the use of a multi-bank data RAM also removes a significant performance bottleneck of the original architecture. Indeed, a single RAM bank can only serve 4 CPUs (placing more than 4 CPUs per tile result in no performance gain because the RAM access is saturated). Having multiple RAM banks per tile removes this limitation. Our test configurations use a maximum of 16 CPU cores per tile and two data RAM banks per CPU core, for a maximum of 4Mbytes of RAM per tile.

The local interconnect is chosen in our design so that it cannot introduce contentions due to its internal organization. Contentions can still happen, for instance, when two CPUs access concurrently the program memory. However, accesses from different sources to different targets never introduce contentions. Interconnect types allowing this are the full crossbars and the multi-stage interconnection network [Aydi et al., 2011] such as the omega networks, the delta networks, or the related logarithmic interconnect [Kakoee, 2012]. The experiments of our work use a full crossbar interconnect.

The CPU core we use is a single-issue, in-order, pipelined implementation of the MIPS32 ISA with no speculative execution. We did not change this, as it simplifies timing analysis and allows small-area hardware implementation. However, significant work has been invested in designing a cycle-accurate model of this core inside a state-of-the art WCET analysis tool [Puaut and Potop-Butucaru, 2013].

The caches have been significantly modified. The original design featured caches with a pseudo-LRU (PLRU) replacement policy and with a writing policy that is intermediate between write-through and write-back.[2] Furthermore, memory accesses from the data and instruction caches of a single CPU were multiplexed over a single connection to the local interconnect of the tile. Both these choices are known to complicate timing analysis and/or to reduce its precision [R. Wilhelm et al., 2008, Hardy and Puaut, 2008], and thus we revert to more conservative choices: We use the LRU replacement policy, a fully write-through policy, and we let the instruction and data caches access the local tile interconnect through separate connexions. Note that the use of a write-through policy reduces the processing speed of each CPU. This is the only modification we made on the MPPA architecture that decreases processing speed.

Synchronization: To improve temporal predictability, and also speed, we do not use interrupt-based synchronization. Indeed, interrupt signaling by itself is fast, but handling an interrupt usually requires accesses to program memory which take supplementary time. Furthermore, imprecision related to interrupt arrival dates and modifications of the cache state mean that it is difficult to accurately account for interrupt-based synchronization overhead during execution time analysis.

To avoid these performance and predictability problems, we never use the interrupt units, and therefore remove them from our hardware descriptions. Instead, we include in each tile a hardware lock component which allows synchronization with very low overhead (1 non-cached local RAM access) and without modifications of the cache state.

The lock unit follows a simple request/grant protocol. A lock unit can theoretically implement 2^O locks, where O is the number of bits assigned to offset encoding. Each lock is visible under the form of a memory address, and lock operations are performed through regular (uncached) memory access operations on this address. A lock grant operation consists in writing a non-zero value at this address. A lock request operation consists in reading the address. When the read command packet produced by the CPU cache arrives at the lock unit, if the value of the lock is non-zero, then it is set to 0 and the response packet is immediately sent back to the CPU. If the lock value is 0, then the response packet is sent back only after the lock has been granted by some other CPU or DMA. This effectively blocks execution on the CPU until the lock has been granted.

Multiple processors may be blocked waiting for locks at the same time, and each read transaction that is blocked will block (in the SoCLib architecture) a VCI interface.

[2] Consecutive writes inside a single cache ligne were buffered.

Therefore, each lock unit has multiple VCI interfaces. It has one interface for each CPU core and one for the NIC. Thus, it allows contention-less simultaneous access from all these directions. Given that VCI interfaces correspond to different address ranges, each lock will have one address on each interface, but all these addresses share the same address offset, which is known as the `lockid`, and which uniquely identifies the lock.

Buffered DMA: To minimize NoC usage during *large* data transfers, we perform them using direct memory access (DMA) units controlled by the CPU of the sending tile. Transferring data directly through CPU operations would mean that the packet construction and sending is controlled by the CPU cache, which generates one packet for each transmitted word.

The traditional DMA unit used in the original MPPA architecture requires significant software control to determine when a DMA operation is finished so that another can start. This is either done using interrupt-based signaling, which has the inconvenients mentioned above, or through polling of the DMA registers, which requires significant CPU time and imposes significant constraints on CPU scheduling.

To avoid these problems, we use DMA units allowing the buffering of transmission commands. A CPU can send one or more DMA commands while the previous DMA operation is not yet completed. Furthermore, the DMA unit can be programmed so that it not only sends out data, but also signals the end of the transmission to the target tile by granting a lock. Thus, all inter-tile communication and synchronization can be performed by the DMA units, in parallel with the data computations of the CPUs and without requiring significant CPU time for control.

The high-level interface we use to program the new DMAs is the `dma_send` library routine which has the following prototype:

```
void dma_send(SRC,DST,SIZE,LOCK_ADDR)
```

where:

- `SRC` is the address of the data to send (local tile address).
- `DST` is the address where data must be copied.
- `SIZE` is the amount of data to send (in octets).
- `LOCK_ADDR` is the address of a lock in the target tile. The DMA grants this lock immediately after sending the last data packet. Setting the lock address to `0xFFFFFFFF` means that no lock synchronization is needed.

3.4 SystemC simulation and compilation support

Executing code over our DSPIN-based many-core platform requires two distinct executables:

- The cycle-accurate hardware simulator itself, compiled and linked with the SystemC and SoCLib libraries, using the `soclib-cc` command.
- The multi-threaded software that will run on the MPPA platform.

The hardware simulator is written in SystemC, and is of cycle-accurate bit-accurate (CABA) type. The strict CABA modeling rules followed in the definition of the various architecture components allow the use of the optimized simulation engine SystemCASS [Buchmann et al., 2004].

Using a simulation-based approach, as opposed to using silicon hardware, had two significant advantages: First, it facilitated our experimentation with various degrees of control over the interconnect, which implied significant changes in the hardware. Second, by instrumentation of our the cycle-accurate hardware simulator we were able to perform precise timing measures.

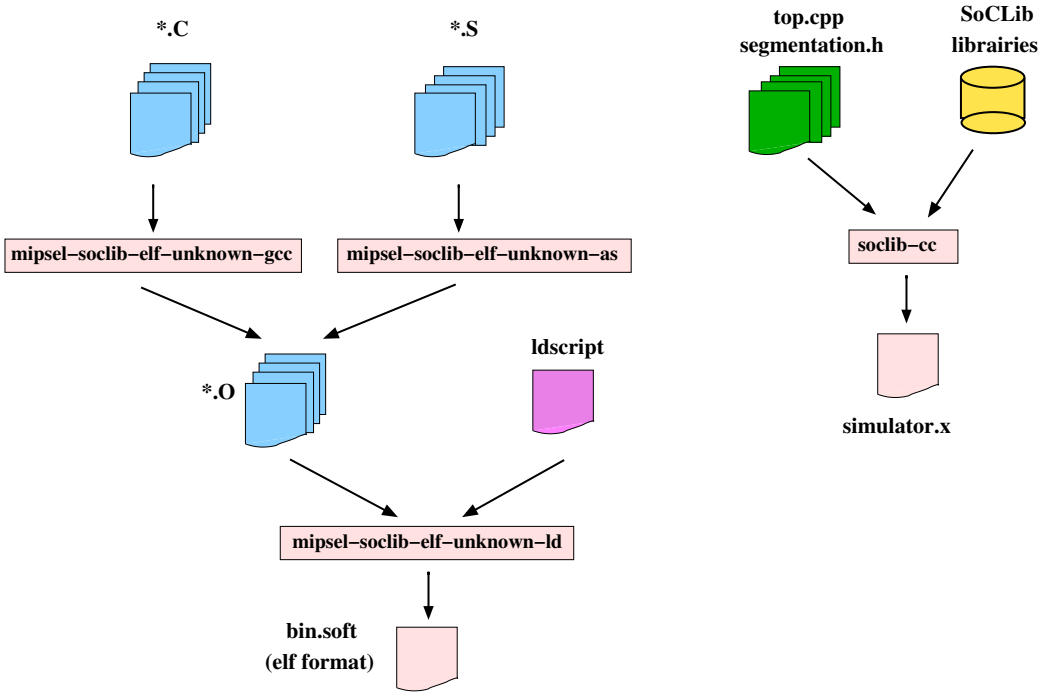


Figure 3.6: Building the MPPA simulator and executable code

The code executed on top of our hardware simulator is built from C sources and some assembly code for low-level device control and boot code. Compilation is done using a

`gcc` cross-compiler (for the MIPS32 architecture), unmodified, but with automatically generated loader script files.

The output of the linker is a single ELF file that contains the code and data for all the tiles of the MPPA.

We have developed a tool allowing the automatic synthesis of the corresponding systemC models and memory mapping files from a high-level hardware specification defining:

- The number of rows and columns of tiles in the MPPA.
- The number of CPU cores in each tile.
- The amount of RAM per tile.
- The number of RAM memory banks in each tile.
- The configuration of I/O devices.

Chapter 4

Programmable NoC arbitration

Contents

4.1 The case for programmed arbitration 71
4.1.1 The principle 71
4.1.2 Target application classes 72
4.1.3 The cost of programmability 73
4.2 Programmable DSPIN 74
4.2.1 NoC router extensions 74
4.2.2 Area overhead 77
4.3 A simple example in depth 78
4.4 Case study: the FFT 84
4.4.1 FFT algorithm description 85
4.4.2 Evaluation of the slow-down due to traffic injection 88
4.4.3 Removing the slow-down through NoC programming 89

In the previous chapter we have presented the original SoCLib-based MPPA architecture we use as base for our work, and then explained how we modified its computing tiles to improve the overall timing predictability.

But the main originality of our work, on the hardware side, concerns the NoC. We explore in this chapter how the NoC architecture should offer the proper infrastructure to implement optimal communication schedules that are synchronized with the scheduling of computations on processors. Our thesis is that optimal data transfer patterns should be encoded using simple programs configuring the router nodes, each router being then programmed to act its part in the globally concerted computation and communication scheme.

We concretly support our thesis by extending the DSPIN 2D mesh NoC. We replace the fair arbitration modules of the NoC routers with micro-programmable modules. We justify our choice by its use in reducing communication time, and in reducing contentions in two case studies: A simple embedded control application and an implementation of the Fast Fourier Transform (FFT).

4.1 The case for programmed arbitration

4.1.1 The principle

In the original MPPA architecture proposed by SoCLib, the use of fair arbitration inside the NoC routers ensures good utilization factors for the resources of the DSPIN NoC. However, when implementing embedded control or consumer applications the objective is usually not to improve NoC usage, but to improve application-level characteristics such as speed, power consumption, predictability, *etc.*

The following example shows to what extent fair routing may slow down communications, and thus the overall application. Fig. 4.1 pictures the case where the "East" output of a DSPIN router is concurrently traversed by two bursts of data, each formed of n packets of equal length m numbered from 1 to n. Each burst transmits a single piece of data, and processing cannot start at the destination until all packets of a burst have arrived. In the worst case, the first packets of the two bursts arrive at the router in the same clock cycle, and we assume that the current state of the arbiter leads to "Local" passing first. Then, the fair routing policy of DSPIN results in the interleaved transmission of the "Local" burst and the "West" burst having respectively lengths $(2*n-1)*m$ and $2*n*m$. The passing order can be represented by the $(WL)^n$ regular expression.

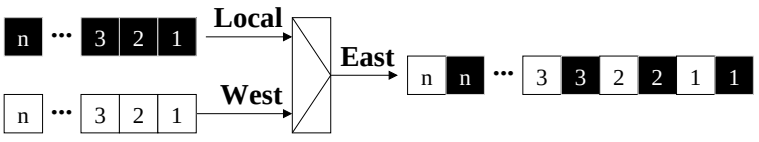


Figure 4.1: Round Robin communication interleaving:$(WL)^n$

Our objective is to allow for the better packet interleaving of Fig. 4.2. We assume there that the "West" burst is needed by a computation located on the critical path of the application running on the NoC, so that accelerating the burst transfer will result in a faster

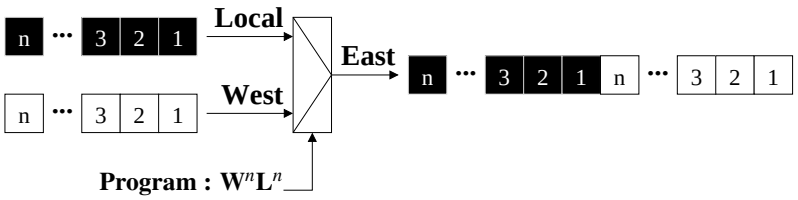


Figure 4.2: Programmed communication interleaving:W^nL^n

application. Therefore, we let the entire "West" burst pass before the entire "Local" burst, even when "Local" arrives first. The transmission durations will then be respectively $2*n*m$ and $n*m$. The passing order is represented by the W^nL^n regular expression inside the multiplexer. This expression is an abstract view of the router program specifying that n packets from the "West" direction should pass before the n packets from the "Local" direction of the NoC router.

4.1.2 Target application classes

We are therefore advocating for applying a static scheduling principle for the arbitration of NoC communications. This should be done under a global optimization approach in conjunction with static scheduling of the computations on the CPUs. Two types of applications can benefit from from such an implementation approach:

- Signal and image processing algorithms, which often have a very regular control structure, and for which static scheduling yields optimal results. Such algorithms, like the FFT, form a significant part of embedded control software, in terms of computational resource use.

- Applications with high functional and temporal determinism requirements, like those used in safety- or mission-critical embedded systems, where predictability is imposed using specific language or OS constructs[Henzinger and Kirsch, 2007, Lickly et al., 2008, AUTOSAR, 2009, ARINC653, 2005].

The term "application" should be considered here in a large sense. Thus, a global optimization principle can be applied at the level of a sub-system, localized in space and/or time. Improving the properties of the sub-system indirectly improves in many cases the properties of the global system.

Note that in our simple example, the optimal communication schedule (*i.e.* letting the West burst pass before the Local one) can also be obtained using more classical priority-based arbitration mechanisms. However, this is no longer true for more complex applications and architectures.[1] Our focus on "regular" applications justifies our choice of static scheduling and NoC programming mechanism. For larger classes of applications, other types of arbitration mentioned in Chapter 2 may provide best results.

4.1.3 The cost of programmability

As we shall see in the examples of the next sections, the static ordering of packets at router multiplexers allows speed gains and a balanced use of NoC resources, by precisely allocating free time slots on the NoC to in transit packets. It also allows for the construction of applications with very good timing predictability and enhanced determinism.

However, these gains come at a price. One part of this price is the need for *programmable NoC routers*, described in the next section, which increase the silicon surface of the NoC.

The second part is the need for temporal predictability. Indeed, the router programs are computed based on the expected execution order of the various operations (computations and communications). In turn, the order depends on operation durations, and better precision in computing these durations results in better routing programs. Therefore, programming NoC packet orders should only be done on architectures where all other architectural elements (*i.e.* the MPPA tiles) provide strong support for ensuring temporal predictability, as described in the previous chapter.

The third and possibly most important part of this price is the need for automation: the complexity of many-cores and of the applications running on them is such that allocation and scheduling, including the construction of the router programs, must be largely automated.

[1] Proof sketch: Optimal (static) scheduling of such applications may require scheduling algorithms that are not of as-soon-as-possible (ASAP) type. However, priority-based scheduling is a form of ASAP scheduling, hence not optimal in the general case.

4.2 Programmable DSPIN

To evaluate the costs and benefits of programmed arbitration, we have modified the DSPIN NoC of the previously-defined MPPA architecture by allowing the programming of all arbiters of the command NoC.

4.2.1 NoC router extensions

The purpose of DSPIN router programming is to fully control the arbitration between incoming input packets at each of the router's multiplexers. Adding programmability to the NoC means adding program to each router multiplexer. This is done by introducing new signals (wires) that control each multiplexer (Fig. 4.3), which are controlled by the new hardware components called *router controllers*, as pictured in Fig. 4.4.

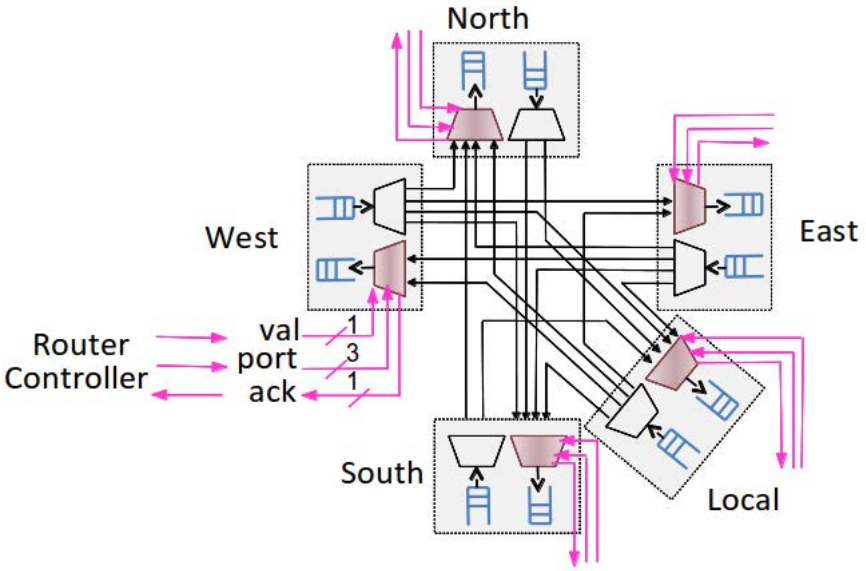


Figure 4.3: The programmable DSPIN router architecture

There are 3 new signals, named VAL, PORT and ACK, which use a classical FIFO protocol to transmit the sequence of routing orders to the router multiplexer:

- The PORT signal is set by the router controller. It tells the router multiplexer from which input direction to accept a packet for transmission. This signal is set while the previous packet is still transmitted, but does not affect its transmission (we do not allow packet transmission interruption).

- Signal VAL is set by the router controller to signal the presence of a valid value on PORT. It is reset upon reception of ACK.

- Signal ACK is used by the router multiplexer to acknowledge that the current PORT value has been taken into account. It is set for one clock cycle only when the first flit of the corresponding packet passes.

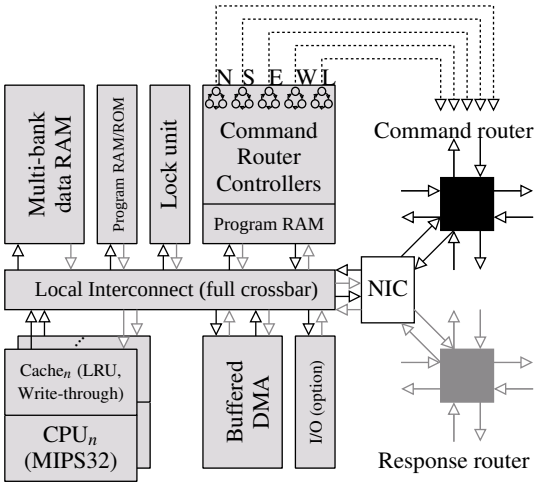


Figure 4.4: Tile structure in our architecture with programmable routers

One router controller is added to each of the router multiplexers of the command NoC. We therefore use 5 independent micro-programmable router controllers per tile, grouped together into a LocalRouterController component connected to the local interconnect of the tile to allow programming, as shown in Fig. 4.4.

Each of the 5 router controllers contains:

- 8 16-bit local registers named R0 to R7, which allow a compact encoding of regular expressions through the use of counters.

- 256 32-bitword (1kbyte) local memory for micro-programs.

- 2 memory-mapped registers that allow CPU control over the router controllers. One of these registers allows for commuting between the programmed and fair (Round Robin) arbitration policies. The other is the router controller program counter, which can be read to determine the current execution point.

Instruction	Function
`loadimm` reg imm	Load a 16-bit immediate value (imm) to register reg
`write` port	Send a new arbitration value (through port and val) to the corresponding router output. The controller is blocked until reception of ack.
`dec` reg	Decrement the register
`bnz` reg label	If the value of REG is not zero, jump to the target instruction marked by the label
`jump` label	Jumps to the target instruction marked by label

Table 4.1: The router controller micro-instruction set.

The controllers have 5 micro-instructions, whose assembly language representations are presented in Table 4.1.

The interaction of the tile router controllers with the router multiplexers is realized as pictured in Fig. 4.5. Each multiplexer of the command network is controlled by its own controller running a separate program. Starting and stopping the router controller is realized through the VCI interface of the router controller. When the controller is stopped, the fair arbitration of DSPIN takes control. In Fig. 4.5, the arbitration program will cyclically accept 26 packets from the Local direction, then 26 packets from the West direction.

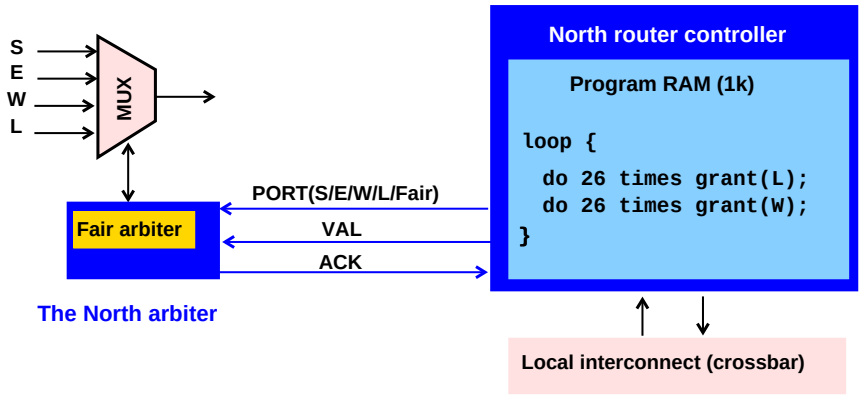


Figure 4.5: Programmed arbitration in a NoC router multiplexer

This specific architecture and instruction set allows an efficient (compact) encoding of routing patterns such as the one of Fig. 4.2 through the use of counters. A full example of router program for a simple application is provided in Section 4.3 and Listing 4.2. More generally, the use of multiple registers and general decrement and test statements allows

a simple encoding of complex loop nests.

The instruction set proposed above has been chosen for its simplicity, and was sufficient for our tests. However, it can be improved in a variety of ways, of which we mention only 2:

- Given the way router controllers are used, instructions `write`, `dec`, and `bnz` can be grouped together, thus reducing program sizes and decoding time.
- Using simple packet inspection techniques allows the use of data-dependent control in router programs.

Note that we only modified the arbitration function of the DSPIN NoC, but not its routing function. This means that NoC programming can introduce deadlocks, but cannot change the function computed by the code executing on processors.

4.2.2 Area overhead

Adding programmability to the network-on-chip routers has its cost. In hardware, this cost is measured in supplementary circuit area. Given the exploratory nature of this work, we did not fully synthesize, place, and route our circuit,[2] so that we were not able to perform an *exact* evaluation of the area cost of programmability.

Instead, we relied on the observation that area increases are mostly due to added memory elements, meaning that the largest penalty in our case comes from the 5Kbytes of micro-program memory added to each tile by the 5 router controllers. We have compared this area with the global memory footprint of the tile, which is mainly due to the multi-bank RAM. We let this RAM size range between values proposed in current industrial many-core architectures: 256 Kbytes [Benini, 2010] and 2 Mbytes[Harrand and Durand, 2011]. We thus determined that the area overhead due to network-on-chip programmability is under 2% of the total chip area.

A second remark here is that router program memory should be accounted for as program memory, and considered in view of the application efficiency (speed, power) optimizations it enables.

[2]Note that synthesizable VHDL versions exist for most components of our platform, as part of the SoCLib library.

To limit the area overhead, we only add programmable multiplexers and their respective controllers on the command network part of the NoC, leaving unchanged the fair arbiters on the response network. To avoid uncontrolled contentions on the response network, all large transfers of data, represented by packet bursts, should be performed with write operations. This way, the response network only transfers 2-flit acknowledge packets with negligible contention cost attached.

4.3 A simple example in depth

To showcase the previous developments, we consider a simplified and pipelined version of the image processing part of the platooning application for the CyCab electric car [Pradalier et al., 2005]. The CyCab application has a cyclic behavior represented by the data-flow synchronous specification of the Fig. 4.6. At each cycle, the system acquires an image. On this image is then applied an edge-detecting Sobel filter. The output of the Sobel filter is used to detect the front car by using a histogram search. The position of the front car is then used to correct the speed and steering of the CyCab electric car to ensure that it follows the front car at the prescribed distance. The histogram, detection and correction function may also change the parameters of image acquisition, which explains the feedback loop of length 2.

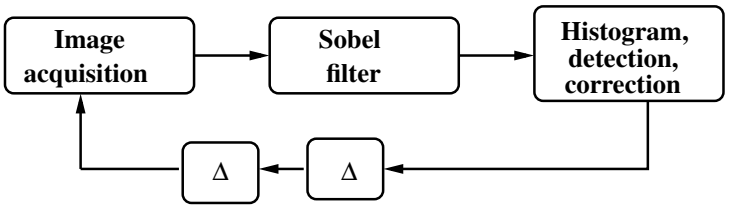


Figure 4.6: Data-flow specification of the CyCab

To allow parallel implementation on our MPPA platform, we start by putting in evidence specification-level parallelism, which results in the data-flow specification of Fig. 4.7. We give detailed description of this formalism in Section 5.1.1.1. The control of image acquisition is performed here by function F (acquisition and data transfer is realized in hardware, F only contains the code performing the configuration of the acquisition device and the synchronization). Functions $S1$ and $S2$ each compute the Sobel filter on half of the input image. On the output ($s1$ and $s2$) of the Sobel filters, functions $H1$ and $H2$ per-

form the histogram search, detection, and correction computation. The histogram search is split in two to allow computations to start before the reception of *S*2's result. The position of the delays is changed through retiming to facilitate allocation and scheduling by putting in evidence the parallelism between the image capture of one cycle and the Sobel computation of the previous cycle.

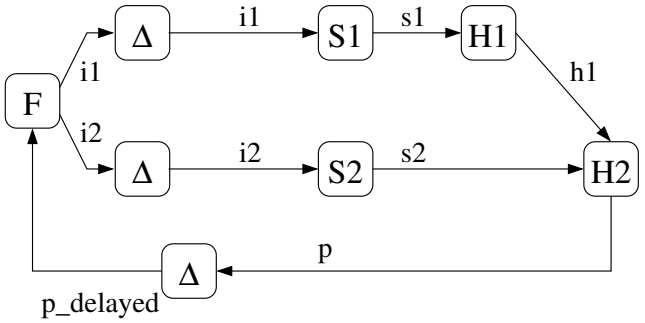


Figure 4.7: Simple data-flow specification

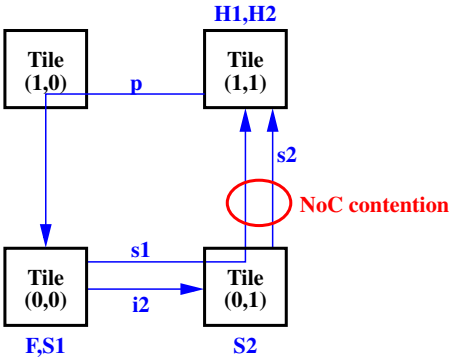


Figure 4.8: Mapping of the example in Fig. 4.7

We implement this specification on an MPPA of size 2x2 as pictured in Fig. 4.8. To simplify presentation, we assume that the durations of the 5 computation blocks are respectively 200, 2100, 2100, 500 and 500 clock cycles. We also assume that the transmission of either *s1* or *s2* uses NoC packets whose combined length is 500 flits. If we also assume that the tiles and the NoC have the same clock speed (fully synchronous), transmitting any of these data will consume only a little more than 500 cycles (with small variations due to instruction cache state and issues related to the internal state of the DMA unit). We also assume that the communication of other data requires only one packet whose length is 10 flits.

We also assume that the data acquisition must be done by F on tile (0,0) and that $H1$ and $H2$ must be executed on tile (1,1). Based on this specification, our automatic allocation, scheduling and code generation tool Lopht, presented in Chapter 5, generates for this example the allocation of Fig. 4.8 (F and $S1$ on (0,0), $S2$ on (0,1), and $H1$, $H2$ on (1,1)), the static scheduling represented by the reservation table of Fig. 4.9 and the code of Listing 4.1.[3]

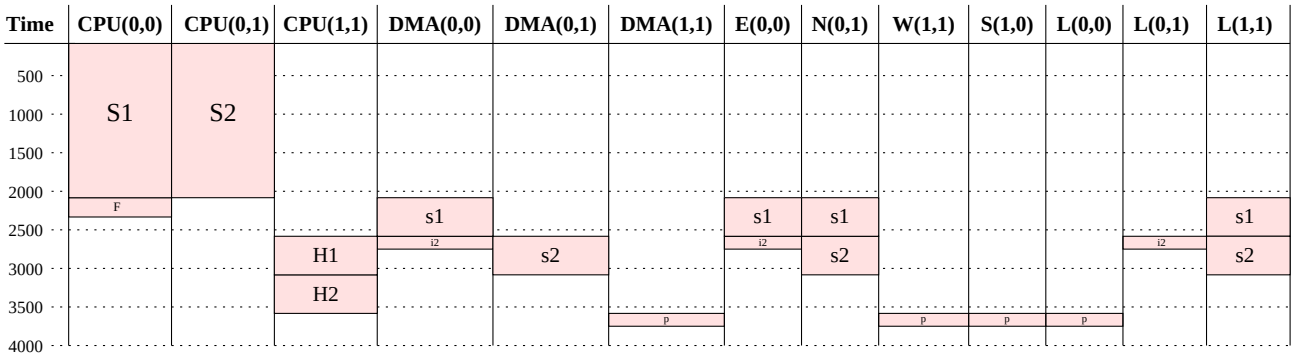


Figure 4.9: Reservation table for our simple application. Only resources that are used are pictured. $CPU(x,y)$ is the unique CPU of tile (x,y), $DMA(x,y)$ is its DMA unit, and the remaining resources are the NoC router multiplexers

The reservation table of Fig. 4.9 represents the static scheduling of the operations inside one execution cycle of the data-flow specification. The infinite execution of our system is a succession of execution cycles, where one new cycle can start as soon as the data dependencies between cycles and resource availability allow it. Pipelining is possible. For instance, a new cycle can start with the execution of $S1$ and $S2$ as soon as F has completed. Each data-flow block is assigned exactly one reservation on one CPU (the reservations made for delays were not represented for simplicity). In our simple example, each communication is assigned one reservation on each resource along its path. For instance, the communication of $s1$ from tile (0,0) to tile (1,1) uses the DMA of tile (0,0) and the NoC router outputs E(0,0), N(0,1), and L(1,1). Since NoC buffering resources are very limited, reservations for one communication must be synchronized. For instance, the reservation for $s1$ on DMA(0,0) starts at date 2100, the reservation on E(0,0) starts at date $2100+\delta$, the reservation on N(0,1) at date $2100+2*\delta$, and the reservation on L(1,1) at date $2100+3*\delta$, where δ is the number of cycles needed to traverse one NoC resource

[3]The reservation table and the code are slightly simplified for presentation reasons.

(in our case, $\delta = 3$). The small increments corresponding to δ are not visible in Fig. 4.9.

```
void main() {
  //Global addresses of all memory objects (data items and locks).
  //Memory objects stored on Tile (1,1)
  ImageType  s1_in_1_1 =
    (ImageType*)(build_address(1,1,ID_GRAM));
  ImageType  s2_in_1_1 =
    (ImageType*)(build_address(1,1,ID_GRAM+1));
  CorrectionType  p_out =
    (CorrectionType*)(build_address(1,1,ID_GRAM+2));
  HistogramType  h1_out =
    (HistogramType*)(build_address(1,1,ID_GRAM+3));
  lock_t s1_in_1_1_lock = build_lock_address(1,1,0);
  lock_t s2_in_1_1_lock = build_lock_address(1,1,1);

  //Memory objects stored on Tile (0,1)
  ImageType  i2_in_0_1 =
    (ImageType*)(build_address(0,1,ID_GRAM));
  ImageType  i2_delayed_in_0_1 =
    (ImageType*)(build_address(0,1,ID_GRAM+1));
  ImageType  s2_out =
    (ImageType*)(build_address(0,1,ID_GRAM+2));
  lock_t i2_in_0_1_lock = build_lock_address(0,1,0);
  lock_t i2_delayed_in_0_1_lock = build_lock_address(0,1,1);

  //Memory objects stored on Tile (0,0)
  ImageType  i1_delayed_in_0_0 =
    (ImageType*) (build_address(0,0,ID_GRAM));
  CorrectionType  p_in_0_0 =
    (CorrectionType*)(build_address(0,0,ID_GRAM+1));
  CorrectionType  p_delayed_in_0_0 =
    (CorrectionType*)(build_address(0,0,ID_GRAM+2));
  ImageType  i2_out =
    (ImageType*)(build_address(0,0,ID_GRAM+3));
  ImageType  i1_out =
    (ImageType*)(build_address(0,0,ID_GRAM+4));
  ImageType  s1_out =
    (ImageType*)(build_address(0,0,ID_GRAM+5));
  lock_t i1_delayed_in_0_0_lock = build_lock_address(0,0,0);
  lock_t p_in_0_0_lock = build_lock_address(0,0,1);
  lock_t p_delayed_in_0_0_lock = build_lock_address(0,0,2);

  switch (cpuid) {
  case 0x00:// CODE FOR TILE (0,0)
    i1_delayed_in_0_0 = i1_init;
    lock_grant(i1_delayed_in_0_0_lock);
    p_delayed_in_0_0 = p_init;
    lock_grant(p_delayed_in_0_0_lock);
    do {
      if(TRUE) {
        lock_request(i1_delayed_in_0_0_lock);
        //execute S1
        S1(i1_delayed_in_0_0,s1_out);
      }
      //send s1 to (1,1)
```

```
    dma_send(s1_out,s1_in_1_1,s1_in_1_1_lock,sizeof(ImageType));
    if(TRUE) {
      lock_request(p_delayed_in_0_0_lock);
      //execute F
      F(p_delayed_in_0_0,i1_out,i2_out);
    }
    //send i2 to (0,1) (and signal it using the lock)
    dma_send(i2_out,i2_in_0_1,i2_in_0_1_lock,sizeof(ImageType));

    if(TRUE) {
      i1_delayed_in_0_0 = i1_out;
      lock_grant(i1_delayed_in_0_0_lock);
    }
    if(TRUE) {
      lock_request(p_in_0_0_lock);
      p_delayed_in_0_0 = p_in_0_0;
      lock_grant(p_delayed_in_0_0_lock);
    }
  } while(1);
  break;
case 0x01:// CODE FOR TILE (0,1)
  i2_delayed_in_0_1 = i2_init;
  lock_grant(i2_delayed_in_0_1_lock);
  do {
    if(TRUE) {
      lock_request(i2_delayed_in_0_1_lock);
      //execute S2
      S2(i2_delayed_in_0_1,s2_out);
    }
    //send s2 to (1,1)
    dma_send(s2_out,s2_in_1_1,s2_in_1_1_lock,sizeof(ImageType));
    if(TRUE) {
      lock_request(i2_in_0_1_lock);
      i2_delayed_in_0_1 = i2_in_0_1;
      lock_grant(i2_delayed_in_0_1_lock);
    }
  } while(1);
  break;
case 0x11:// CODE FOR TILE (1,1)
  do {
    if(TRUE) {
      lock_request(s1_in_1_1_lock);
      //execute H1
      H1(s1_in_1_1,h1_out);
    }
    if(TRUE) {
      lock_request(s2_in_1_1_lock);
      //execute H2
      H2(s2_in_1_1,h1_out,p_out);
    }
    //send p to (0,0)
    dma_send(p_out,p_in_0_0,p_in_0_0_lock,sizeof(CorrectionType));
  } while(1);
  break;
default: do {} while(1); break;
```

```
  }
}
```

Listing 4.1: C code for our simple data-flow application.

In order to simplify code generation using `gcc`, the same program text is shared by all the CPUs of the system. The `main` function is divided in two distinct sections. The declaration section defines the addresses of all memory-mapped objects (variables and locks). These addresses are fixed by the LoPhT tool. They could be hard-coded in the function calls below under the form of numeric constants in order to minimize the memory footprint. Here, we maintained them for clarity.

The second part of the `main` function consists in a `switch` statement that decides which code is executed by each CPU depending on its identifier `cpuid`. By construction, this identifier encodes the CPU tile coordinates (y and x) and the CPU identifier within the tile. This last part is missing in our example because we only used one CPU per tile. The `default` branch of the `switch` statement provides the code for CPUs without allocated operations, such as the CPU of tile (1,0) in our example. This code is an infinite empty loop.

As explained in Chapter 3, synchronization between tiles is realized using the lock units through the `lock_request` and `lock_grant` primitives. In our example, most `lock_grant` calls are performed by the `dma_send` routines that perform all inter-tile data transfers using the DMA units. Note that unlike the scheduling table, the generated code makes no reference to time, as all needed synchronizations are performed using the locks.

The data $s1$ and $s2$, respectively produced by $S1$ and $S2$, become available for transmission over the NoC at the same date (2100). Since they must both transit through the North router of tile (0,1), they exhibit the phenomenon described in Section 4.1.1. In our case, the theoretical slowdown of both transmission will amount to 480 clock cycles, a value closely matched by simulations.

```
//
// Micro-code for the output North
// of Router 0,1
//

// 26 Packets from WEST to NORTH
LOOP:   LOADIMM R1 26
W0:     WRITE WEST
        DEC R1
```

```
        BNZ R1 W0

// 26 Packets from LOCAL to NORTH
        LOADIMM R1 26
L0:     WRITE LOCAL
        DEC R1
        BNZ R1 L0
        JUMP LOOP
```

Listing 4.2: Assembly code for the North router of cluster (0,1) of our simple application.

This slowdown can be eliminated by choosing the order in which packets are transmitted by the North router of tile (0,1) (also denoted N(0,1)). To determine the needed program, we fix the maximal packet length for DMA data transmissions to 20 flits. As the transmission of $s1$ is on the critical path of the application, and not $s2$, the best throughput is ensured by the routing sequence $(W^{26}L^{26})^*$, which allows all the 26 packets from the West input (the transmission of $s1$) to pass before all the packets from the Local input (the transmission of $s2$). From each direction, the first 25 packets correspond to the `dma_send` call, and the 26^{th} corresponds to the lock update. The sequence of 52 packets is repeated indefinitely, once for every computation cycle of the data-flow program. The exact controller assembly program corresponding to this routing sequence is provided in Listing 4.2. This program must be placed on the controller of the North output of the (0,1) tile router. All the other router outputs of the NoC can be left unprogrammed (under fair routing) because no contentions must be arbitrated.

Note that the optimization involved no changes to the C code, which can be executed over a programmed or non-programmed NoC.

4.4 Case study: the FFT

The previous example showed the details of our platform and presented the form of the code generated by our automatic allocation and scheduling tool LoPhT. Fully presenting this tool here would be in Chapter 5.

Instead, we present here a different approach to code generation, where allocation and scheduling are not fully automated. To illustrate this approach, we chose the Fast Fourier Transform [Johnson and Frigo, 2008, Milder et al., 2007] because of its widespread use in signal processing and embedded control and because its characteristics (at least in the variant we considered) raise optimization questions that go beyond the classical speed

optimization criterion. We use the FFT algorithm proposed in [Bahn et al., 2008] for execution on MPPA architectures with 2D mesh NoC interconnect. On the default DSPIN-based MPPA, this algorithm features a strong domination of computation over communication, and a clear organization into successive computation phases separated by strongly synchronizing communication phases.

As we show in Chapter 5, the FFT, considered alone or as part of a larger signal processing application with regular structure, can be efficiently mapped onto our architecture in a fully automatic fashion. However, a large MPPA usually executes several applications which are often only loosely synchronized or even fully asynchronous. In such cases, the LoPhT tool cannot be used at a global level. Each application such as the FFT is then allocated and scheduled separately on a subset of the MPPA tiles. When doing this, it is often necessary to allow that the MPPA area dedicated to one application such as the FFT is traversed by NoC traffic belonging to other applications. As we shall see in the case of the FFT, this external traffic may significantly slow down the application: up to 26% under realistic architectural choices if the standard (non-programmable) fair routers are used. We will also show in this section that *the use of programmable routers allows us to fully remove this slow-down*.

4.4.1 FFT algorithm description

We work on the first FFT algorithm proposed in [Bahn et al., 2008]. It encodes a 1D radix-2 decimation in time FFT. Recall that such an FFT is organized as a set of binary "butterfly" operations, as shown in Fig. 4.10. Computing the FFT on a data vector of size $N = 2^n, n \geq 1$ takes $n \times 2^{n-1}$ butterfly operations (complexity $O(N * log(N))$).

We have evaluated the asymptotic cost of one "butterfly" operation by dividing the global duration of the FFT transform on a single CPU by the number of butterfly operations, provided above. The result, for FFT data vectors ranging in size from 2^4 to 2^{16}, is presented in Fig. 4.11a. The small increase (and stabilization) of the butterfly operation duration corresponds to the moment where FFT data no longer fits inside the data cache of the CPU.

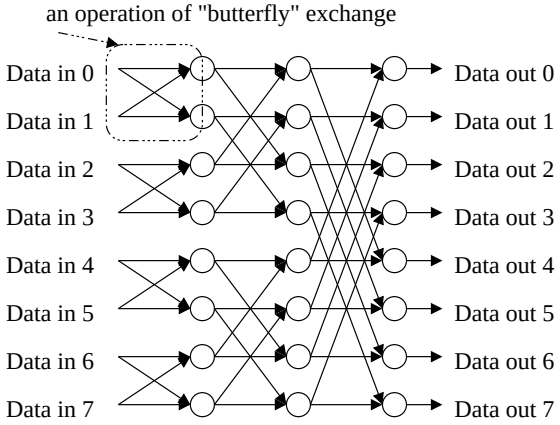


Figure 4.10: A FFT of size 2^3

(a)

(b)

Figure 4.11: (a) Average duration of a butterfly exchange. (b) Average duration of communications on the MPSoC platform

4.4.1.1 Mapping onto the MPPA architecture

For our parallelization work, we will assume that the FFT data vector has size $N = 2^n$, and that the number of tiles is $M = 2^m$. The parallel FFT algorithm works by dividing the data vector in M vectors of size 2^{n-m} each. Data distribution is performed by one of the tiles (detailed below). After reception of its data, each tile computes an FFT on its own data vector of size 2^{n-m}. Once this local FFT is computed, each tile engages in sequence of m stages. During each stage, the tile exchanges its data (through DMA transfers) with another tile and performs 2^{n-m} butterfly operations. At the end, the partial FFT results of all tiles are centralized.

We have performed our experimentations with FFTs of sizes $N = 2^{14}, 2^{15}, 2^{16}$ on an MPPA architecture with 16 tiles organized in a 4x4 square. We made simulation for architectures with 1, 2, 4, 8, and 16 CPUs per tile. Data distribution and data centralization are realized by tile of coordinates x=0,y=0 (the bottom left tile). Thus, there are 6 computation and communication stages, as pictured in the figure Fig. 4.12 (Stage 0 is data distribution, Stage 5 is result centralization).

We have evaluated the duration of the data transmissions, as presented in Fig. 4.11b.

Asymptotically, this value is of approximately 2.69 cycles/data (we work on double word data), but on smaller data sets the duration increases significantly (unlike the computation time) due to the large DMA transfer initiation time. As mentioned before, communications between tiles are performed in bursts, meaning that the NoC resources remain largely unused. However, the bursty nature of the communications lead to significant contentions, even if the FFT is the only application running on the MPPA. These contentions occur in stages 2, 4, and 5, in the places where communication routes in Fig. 4.12 intersect/join.

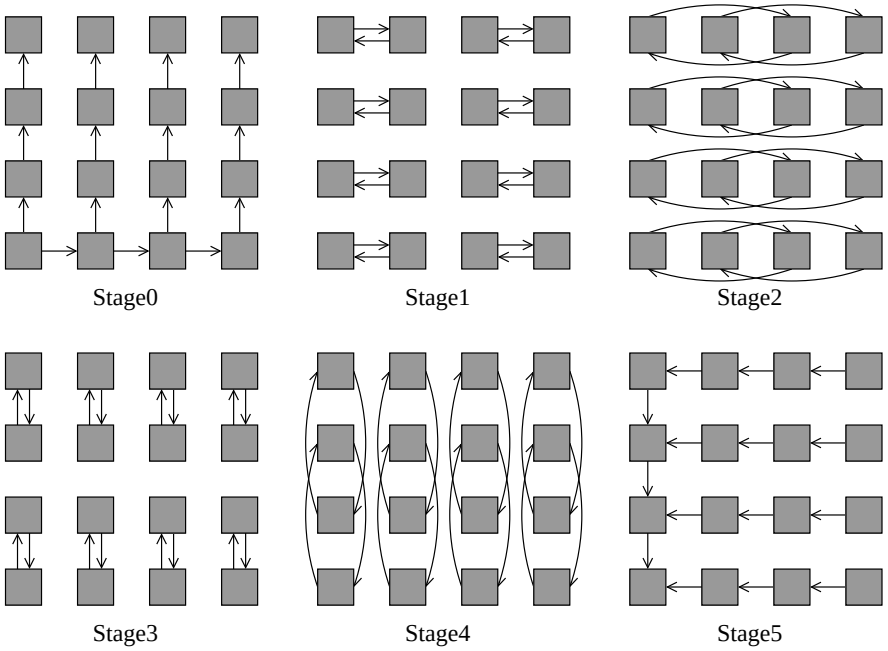


Figure 4.12: The 6 FFT communication stages

4.4.1.2 Traffic injection configuration

The data provided in the previous section concerns the execution of the FFT, in isolation, on a 4x4 tile MPPA. But our objective here is to allow the execution of several applications (not just the FFT), at the same time, on a single, large MPPA like the one in Fig. 4.13. Throughout this work, we assume that application mapping respects the following rules: Each application is assigned a set of tiles that it can use during execution, and no tile is assigned to more than one application. We assume that the FFT is assigned a 4x4 tile area (the dark gray tiles in Fig. 4.13) on which its mapping is the one defined in the previous section.

To evaluate the *worst-case* impact of traffic coming from outside the FFT-dedicated area onto the FFT computation speed, we shall use traffic generators placed in the tiles adjacent to the FFT-dedicated area. These tiles have a light gray color in Fig. 4.13. Traffic is simulated through sustained data transmissions between the generator tiles, in the fixed directions pictured with arrows in the figure. This form of traffic simulates the worst-case effect of East-West and North-South transfers. We believe the approach is realistic due to the bursty nature of the FFT communication, which means that the same slow-down, described below, can be obtained with much less traffic (but occurring during peak NoC use by the FFT).

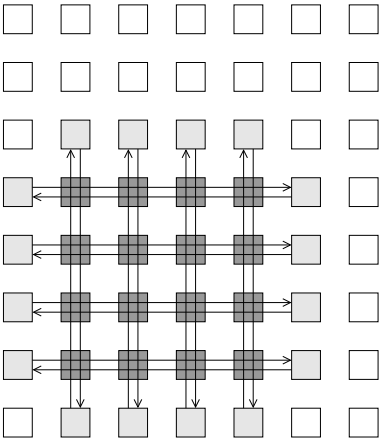


Figure 4.13: Adding traffic not due to the FFT itself

4.4.2 Evaluation of the slow-down due to traffic injection

As expected, traffic injection in the absence of network router programming does slow down the execution of the FFT. To evaluate this slowdown, we have measured the execution time of the FFT with and without traffic injection in the various configurations described above: FFT of size 2^{14}, 2^{15}, and 2^{16}, and MPPA configurations with 1, 2, 4 , 8 and 16 processors per tile. The results are pictured in Fig. 4.14.

The left side tables present here the raw figures (in CPU cycles), and the slowdown induced by traffic injection to the execution time of the FFT. The slowdown is computed using the formula:

$$Slowdown = \frac{\text{“FFT+Traffic”} - \text{“FFT alone”}}{\text{“FFT alone”}}$$

The right side graphs plots the evolution of the speedup obtained through parallelization while the number of processors per tile changes from 1 to 2, 4, 8, and 16. The formula for computing the speedup is:

$$Speedup = \frac{\text{“FFT duration on one CPU”}}{\text{“Parallel FFT duration”}}$$

where the parallel FFT duration is considered for the cases where the FFT is executed alone (solid line), and with traffic injection (dashed line). For instance, a configuration with 16 CPUs/tile (for a total of 256 CPU cores) computes the 2^{16} size FFT 79.62 times faster than a single processor. But the same configuration only results in a 63.22 times speedup if external traffic is injected, which amounts to a significant 26% slowdown of the FFT execution time.

The results without traffic injection show that the MPPA architecture we consider supports well the parallelization of the chosen FFT algorithm. Indeed, for large data sizes, each doubling of the number of processors results in an acceleration close to the theoretical optimum (1.8 in our largest examples, asymptotically 2). When the processors are allocated small data vectors, the slow-down of the communications identified in Fig. 4.11b results in the plateau effect that can be observed in the graph associated with the 2^{14} FFT in Fig. 4.14.

Note that the 26% slowdown mentioned above happens while the FFT still uses only a small part of the NoC bandwidth (as explained below). However, the fact that this use is concentrated in short, highly synchronized bursts means that contentions at those points in time have a very significant effect, and show the importance of reserving not only bandwidth, but bandwidth at specific points in time, as our architecture allows.

4.4.3 Removing the slow-down through NoC programming

Our objective here is to show that NoC programming allows us to maintain FFT speed while allowing the FFT-dedicated MPPA area to be traversed by NoC traffic originating outside of it.

Like the LoPhT tool (Chapter 5), the approach we propose here relies on static scheduling and the use of reservation tables. But the way reservation tables are used is different from that of Lopht. The objective of Lopht is to build a reservation table “from scratch”, and thus produce a static schedule for the application *in isolation* (in our case the FFT).

Figure 4.14: FFT slow-down due to traffic injection

Data Size of FFT (2^{14})
(Single processor execution: 12809139 cycles)

FFT duration in 4×4 tiles			
CPU /tile	FFT alone CPU cycles	FFT+traffic CPU cycles	Slowdown of Time
1	1040453	1057503	1.64%
2	564973	597043	5.68%
4	348246	389988	11.99%
8	263112	312042	18.60%
16	266411	326989	22.74%

Data Size of FFT (2^{15})
(Single processor execution: 27435983 cycles)

FFT duration in 4×4 tiles			
CPU /tile	FFT alone CPU cycles	FFT+traffic CPU cycles	Slowdown of Time
1	2189483	2222736	1.52%
2	1202232	1264318	5.16%
4	695706	780041	12.12%
8	484784	576635	18.95%
16	418092	527041	26.06%

Data Size of FFT (2^{16})
(Single processor execution: 58507893 cycles)

FFT duration in 4×4 tiles			
CPU /tile	FFT alone CPU cycles	FFT+traffic CPU cycles	Slowdown of Time
1	4598214	4663732	1.42%
2	2503537	2626790	4.92%
4	1448886	1608880	11.04%
8	945211	1115779	18.05%
16	734862	925365	25.92%

On the contrary, we shall assume here that the FFT application is already fully allocated and scheduled on the MPPA area dedicated to it, and that the use of MPPA resources by the FFT in isolation is represented using a reservation table covering all computation, communication, and storage resources of the FFT-dedicated area. Such a reservation table can be obtained in two ways:

- If the allocation and scheduling of the application is realized using the LoPhT tool, then LoPhT automatically provides this table. This approach can be applied not only on fully regular applications such as the FFT, but also on applications featuring some data-dependent conditional control.

- If the application was allocated and scheduled by other means (*e.g.* manually) and if it features no data-dependent conditional control, then a simulation of the application *in isolation* on the MPPA simulation platform allows the recording of the (worst-case) use of the various resources under the form of a trace. This trace can be directly used as a reservation table.

Regardless of the way it was generated, the reservation table includes the routing operations performed by all the routers of the NoC, including their starting dates and durations. We can then determine for each NoC route when it is free from FFT traffic. *The free time intervals of each route can then be used to schedule the transfer of NoC traffic originating outside the FFT-dedicated MPPA area.* The resulting scheduling table, when projected on each router output, provides the program of the router. Collectively, these programs ensure the preservation of the FFT speed while allowing the transfer of packets not belonging to the FFT.

Note that external traffic is statically scheduled. This scheduling fixes the maximum amount of resources that can be dedicated to the transmission of external traffic along every NoC route. Such an approach is best suited for cases where the external traffic is either statically scheduled (like that of the FFT), or when it can be divided among a set of fixed virtual circuits for which the maximal needs (throughput, latency) are known, like in the TDM-based NoCs reviewed in Chapter 2.[4]

[4]In this latter case the router controller instruction set of Table 4.1 must be extended to allow the `write` instruction to timeout while waiting for packets belonging to external traffic. This allows us to account for the case where a virtual circuit does not use all its reservations.

Figure 4.15: The tables present the network use by packets not belonging to the FFT (in % of all transiting packets). The figures show the FFT packet-ratio.

Data Size of FFT (2^{14})

CPU/tile	Non-programmed	Programmed	Loss
1	98.51%	97.47%	1.04%
2	97.36%	95.33%	2.02%
4	95.97%	92.45%	3.52%
8	94.94%	89.95%	4.98%
16	95.18%	90.10%	5.08%

Data Size of FFT (2^{15})

CPU/tile	Non-programmed	Programmed	Loss
1	98.58%	97.60%	0.99%
2	97.51%	95.62%	1.89%
4	95.97%	92.44%	3.53%
8	94.54%	89.14%	5.41%
16	94.01%	87.33%	6.69%

Data Size of FFT (2^{16})

CPU/tile	Non-programmed	Programmed	Loss
1	98.76%	97.91%	0.86%
2	97.80%	96.15%	1.65%
4	96.43%	93.38%	3.04%
8	94.81%	89.78%	5.03%
16	93.80%	86.97%	6.83%

We have written a tool that automatically allocates a maximal amount of statically scheduled communications from outside the FFT-dedicated area. Our tool works in cases where external traffic is done along virtual circuits that share no NoC component, so that no arbitration is needed between the virtual circuits. This tool allowed us to schedule the external traffic generated as described in Section 4.4.1.2, and to prove that our approach preserves the FFT speed. The flow of our tool is presented in Fig. 4.16. A key point of this flow is the optimization of the generated router programs by identification of repetitive patterns in the routing orders provided by the previously-defined process.

The simulation of the FFT with external traffic injection allowed us to determine how NoC programmation affects the amount of resources allocated to packets originating outside the FFT-dedicated area. Table 4.15 shows the number of FFT-generated packets transiting the FFT-dedicated MPPA area, computed as a percentage of all transiting packets. We can see that programming reduces permeability to external traffic, but not significantly. In the worst case (bottom line), more than 86% of all transiting packets do not belong to the FFT, and the loss of permeability due to programming is of less than 7%. In all cases, the programmed NoC allows the FFT to run as if no external traffic existed.

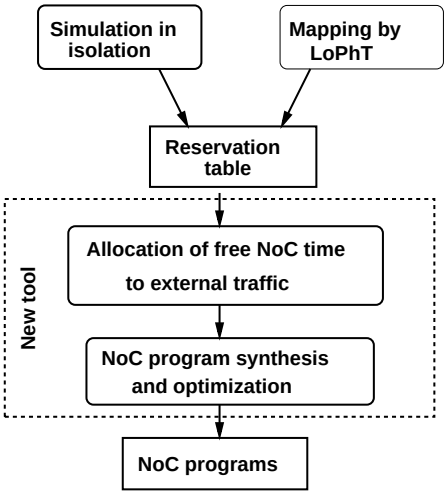


Figure 4.16: NoC program generation flow

Chapter 5

Off-line mapping of real-time applications using LoPhT

Contents

5.1 Background: AAA using the Clocked Graphs formalism 95
5.1.1 The Clocked Graph formalism 95
5.1.2 Off-line scheduling of CG specifications 102
5.2 Static (off-line) mapping onto MPPA architectures 109
5.2.1 AAA for NoC-based MPPA: The problem 109
5.2.2 Extension of the CG format 111
5.2.3 Makespan-optimizing scheduling 115
5.3 Automatic code generation 121
5.3.1 Tile code generation 121
5.4 Experimental results 124

In this chapter, we define our technique and tool, called LoPhT, for automatic off-line real-time mapping of synchronous data-flow specifications onto MPPA architectures such as the previously-defined one, where the NoC supports static communications scheduling.

Our off-line mapping and code generation technique drew significant inspiration from previous work on the AAA methodology [Grandpierre and Sorel, 2003], whose principles have been defined in Section 5.1. More precisely, we defined our techniques as an extension of a particular implementation of the AAA methodology, namely the one built around the Clocked Graphs (CG) intermediate representation formalism [Potop-Butucaru et al., 2009].

This is why we fully dedicate the first section of this chapter to a detailed technical description of the existing CG formalism and CG-based mapping algorithms. This section borrows material from [Potop-Butucaru et al., 2009]. Sections 2 and 3 of this chapter

provide our original contribution. They first explain why the CG formalism and existing mapping techniques are not fully adapted to our many-core implementation problem (mainly due to issues related to NoC communication mapping). Then, they define our off-line scheduling algorithms and the code generation technique. The last section provides some quantitative evaluation.

5.1 Background: AAA using the Clocked Graphs formalism

5.1.1 The Clocked Graph formalism

The CG formalism has been introduced in [Potop-Butucaru et al., 2009] as an intermediate representation to be used during the (multi-processor, real-time) implementation of synchronous specifications. This format allows the faithful representation of specifications written in high-level synchronous languages like Scade/Lustre [Halbwachs et al., 1991], Signal [Guernic et al., 2003], or discrete-time Scicos [Campbell et al., 2006], including some structural information that can be used for efficient code generation purposes. At the same time, the CG format is close enough to the target machine code to allow fine grain manipulations such as scheduling, allocation, and optimization.

Following classical industrial design practices, a CG specification is formed of a *functional specification* and a *non-functional specification*. The functional specification is provided under data-flow synchronous program with a cyclic execution model. The non-functional specification includes a description of the multi-processor architecture (topology and component types) plus the real-time characteristics of the data-flow nodes and the allocation constraints.

5.1.1.1 Functional specification

Synchronous languages [Halbwachs, 1993, Benveniste et al., 2003, Potop-Butucaru et al., 2005] are modeling and programming formalisms used in the specification and analysis of safety-critical embedded systems. They comprise (synchronous) concurrency features, and are based on the Mealy machine paradigm: Input signals can occur from the environment, possibly simultaneously, at the pace of a given *global clock*. Output signals and state changes are then computed before the next clock tick, grouped as one *atomic reaction*,

also called *execution instant*.

Among synchronous languages, the CG representation is characterized by a strict separation of computations, under the form of a *data-flow graph*, from control, represented with *clocks* which identify the synchronous execution instants where the data-flow elements are executed. The two parts are interconnected as all computations and communications are associated a clock defining their execution condition, while clocks depend on values computed by the data-flow.

Clocks *Clocks* are logical activation conditions defining the sequence of synchronous execution instants where some computation or communication is performed, or when some data is available to be used in computations. To each execution instant a clock associates a value of 1 (*true*, active) or 0 (*false*, inactive). A computation or communication whose clock is *clk* will be executed in execution instants t with $clk(t) = true$.

The CG formalism defines two types of clocks: elementary clocks and composed clocks. Elementary clocks are:

- The constant clocks *True* and *False* defined by $True(t) = true$ and $False(t) = false$ for all instant t.
- The *Test* clocks are Boolean predicates over the output ports of data-flow nodes (defined below). For example, if o an integer output port of data-flow node n, $o = 2$ is the clock defining the execution instants where the value of o is 2. Similary, $o_1 = o_2$ is the clock defining execution instants where o_1 equals o_2.

Composed clocks are obtained from the elementary ones by composing them using:

- The Boolean combinators $\wedge$, $\vee$, and $\neg$: For example, $clk_1 \wedge clk_2$ is true at execution instants where both clk_1 and clk_2 are true. We also denote $c_1 \backslash c_2 = c_1 \wedge \neg c_2$ the difference operators on Booleans and clocks.
- The sub-clock operator $clk_1.clk_2$, which evaluates clk_2 only on instants where clk_1 is true.

Clocks are partially ordered by $\leq$, where $clk_1 \leq clk_2$ if at each execution instant t $clk_2(t)$ is true whenever $clk_1(t)$ is true.

instant	1	2	3	4	5	6	7	8	9	10	11
output port x	5	4	3	2	1	0	-1	-2	-3	-4	-5
True	1	1	1	1	1	1	1	1	1	1	1
$x > 0$	1	1	1	1	1	0	0	0	0	0	0
$\neg(x > 0)$	0	0	0	0	0	1	1	1	1	1	1
$(x > 0) \wedge (x \leq 3)$	0	0	1	1	1	0	0	0	0	0	0

Figure 5.1: Examples of clocks.

Fig. 5.1 shows examples of clocks. The first line of the table indexes execution instants, and the second provides the value of output port x.

Clocked graphs A clocked graph is a pair $\mathcal{G}$=($\mathcal{N}$,$\mathcal{A}$) formed of a set of data-flow *nodes* $\mathcal{N}$ and a set of *arcs* $\mathcal{A}$. Each node $n \in \mathcal{N}$ has a set of *named input ports* $\mathcal{I}(n)$, a set of *named output ports* $\mathcal{O}(n)$, and a *clock* $clk(n)$. The name of a port p is denoted $name(p)$ and the ports of a node have all different names. The port of name *name* of node n is denoted with $n.name$. Each input and output port p is assigned a data type (domain) denoted $\mathcal{D}_p$.

Each data-flow arc $a \in \mathcal{A}$ connects one output port denoted $src(a)$ to one input port denoted $dest(a)$. Each arc has a communication condition (a clock) denoted $clk(a)$, which determines at which instants the data transfer takes place. Formally:

$$\mathcal{A} \subseteq \left(\bigcup_{n \in \mathcal{N}} \mathcal{O}(n) \right) \times \left(\bigcup_{n \in \mathcal{N}} \mathcal{I}(n) \right) \times \mathcal{C}$$

where $\mathcal{C}$ denotes the set of all clocks that can be defined using the previously-defined syntax. Each arc $a \in \mathcal{A}$ is assigned a data type (domain) denoted $\mathcal{D}_a$ and for each $a \in \mathcal{A}$:

$$\mathcal{D}_a = \mathcal{D}_{src(a)} = \mathcal{D}_{dest(a)}$$

The Clocked Graph formalism defines two types of data-flow nodes: *function* nodes and *delay* nodes. Function nodes represent atomic stateless computations. They perform computations following a simple, cyclic read-compute-write semantics where all input ports are read and all ouput ports are computed at each cycle where they are activated. The state of the system is maintained by the delays. A delay $\delta \in \mathcal{N}^{\Delta}$ allows data to be propagated between the successive execution instants (*i.e.* from one global execution cycle to the next), where $clk(\delta)$ is active. Each delay δ has a data domain denoted $\mathcal{D}_\delta$, one input port i of type $\mathcal{D}_\delta$, one output port o of type $\mathcal{D}_\delta$, a depth denoted $depth(\delta)$ which is a

strictly positive integer, and a list of initial values of length $depth(\delta)$: $\delta_0, \ldots, \delta_{depth(\delta)-1} \in \mathcal{D}_\delta$. The function computed by a delay block is (t indexes here execution instants):

$$\delta.o(t) = \begin{cases} \delta_t & \text{if } t < depth(\delta) \\ \delta.i(t - depth(\delta)) & \text{if } t \geq depth(\delta) \end{cases}$$

The set of function nodes is denoted $\mathcal{N}^{\mathcal{F}}$, and the set of delay nodes is denoted $\mathcal{N}^{\Delta}$, and we have $\mathcal{N} = \mathcal{N}^{\mathcal{F}} \cup \mathcal{N}^{\Delta}$.

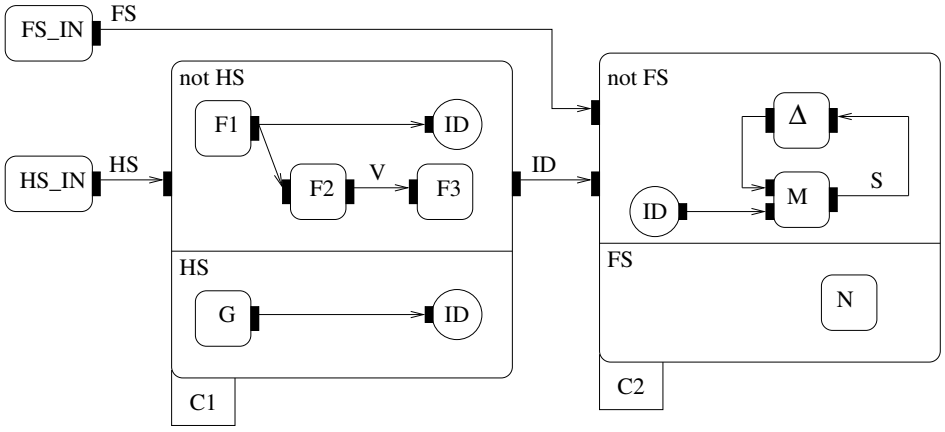


Figure 5.2: Graphical model of a SynDEx synchronous specification

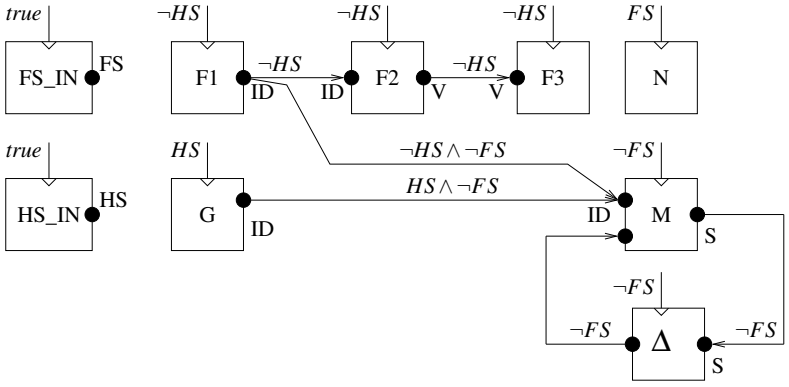


Figure 5.3: Clocked Graph representation of the example in Fig. 5.2

Example We provide in Fig. 5.3 the intermediate representation corresponding to the simple SynDEx synchronous specification of Fig. 5.2. The specification represents a system with two switches (Boolean inputs) controlling its execution: high-speed (HS) vs. low-speed ($\neg HS$), and fail-safe (FS) vs. normal operation ($\neg FS$). In the low-speed mode, more operations can be executed, whereas in the fail-safe mode the operation that gets

executed (N) does not use any of the inputs, because the sensors or treatment chain are assumed to be faulty (control is done using default values).

The clocked graph has 9 data-flow nodes (8 function nodes and 1 delay node) and 6 arcs. Clocked Graphs is a textual language, for which we use here an intuitive graphical representation. The predicate on top of each data-flow node and arc is its clock.

The behavior of this CG representation is: Nodes FS_IN and HS_IN have clock *true*, so they are executed at each execution cycle to read FS and HS. If HS =*false* then clock $\neg HS$ is *true* for the instant, which triggers the execution of $F1$, followed by $F2$ and $F3$. Otherwise, G is executed (on clock HS). Clock dependencies, such as clock HS depending on the output port $HS\_IN.HS$, are not explicit. Both $F1$ and G are computing through their output ports named ID the SynDEx-level output value ID of the hierarchical conditional node $C1$. The execution of N can start as soon as we can determine that FS is *true* for the instant. The execution of M (on clock $\neg FS$) can start after ID is received from either $F1$ or G. It also uses the value S produced by M in the previous execution cycle (we assume that $depth(M) = 1$).

Data-flow blocks having no dependency between them can be executed in parallel. For instance, if FS =*true* then N can be executed as soon as FS is read, independently of the execution of $F1$, $F2$, $F3$, or G. On the contrary, the computation of M must wait until both FS and ID have arrived.

Support of a clock We have previously defined clocks as predicates over the output ports of data-flow blocks, or Boolean combinations thereof. However, this definition is not sufficient to allow a precise treatment of causality. We extend it here with the notion of *support* of a clock, which defines the set of all the output ports used in its computation, along with the clocks defining the instants where these output ports are needed for the computation. In the CG specifications considered in this thesis, a support is specified for the clocks of all nodes and arcs.

Formally, the support of a clock c is a list of pairs $o@c_o$, where o is an output port of some node n, and c_o is a clock defining the instants where the value of o is used in the computation of c. Intuitively, the support of a clock gives sufficient data for some algorithm to compute the clock. For instance, a correct support for the clock $c = (o_1 = 3) \wedge (o_2 = 5)$ is $\{o_1@true, o_2@(o_1 = 3)\}$ which corresponds to the following computation

of c:

```
c = false ;
read(o1) ;
if(o1 == 3){
  read(o2) ;
  if(o2 == 5) c = true
}
```

The notion of correct support is formally defined in [Potop-Butucaru et al., 2009], under the name of *endochrony*. Intuitively, endochrony requires that the clock c of an element $o@c$ of a support list can be computed using the previous elements of the support list.

Note that a given clock can have several supports. For instance, the clock c defined above also accepts the support $\{o_2@true, o_1@(o_2 = 5)\}$.

Well-formed properties To ensure compliance with the synchronous hypothesis and the atomicity assumption for node executions, we require CG specification to satisfy the following properties:

- **No causality cycle**: All cycles in the data-flow graph contain a delay node. Moreover, acyclicity must be preserved when the dependencies related to clock computation (through the supports) are taken into account.
- **No uninitialized data**: All the input ports of a node n receive a value in instants where n is activated. Formally:
  - For all $n \in \mathcal{N}$ and for all $i \in \mathcal{I}(n)$ we have:

$$clk(n) \leq \bigvee_{a \in \mathcal{A}, dest(a)=i} clk(a)$$

  - For all $a \in \mathcal{A}$ we have $clk(a) \leq clk(src(a))$.
- **Single assignment**: An input port of a node n receives at most one value at each execution cycle (no write conflict is possible). Formally, we require that for all $a_1, a_2 \in \mathcal{A}$ with $dest(a_1) = dest(a_2)$ we have $clk(a_1) \wedge clk(a_2) = false$.

5.1.1.2 Non-functional specification

Platform model Formally, in the original CG formalism, a piece of architecture is a pair $Arch = (Comm(Arch), Procs(Arch))$ formed of a communication medium connecting a finite set of sequential processors $Procs(Arch)) = \{P_1, P_2, ..., P_n\}$.

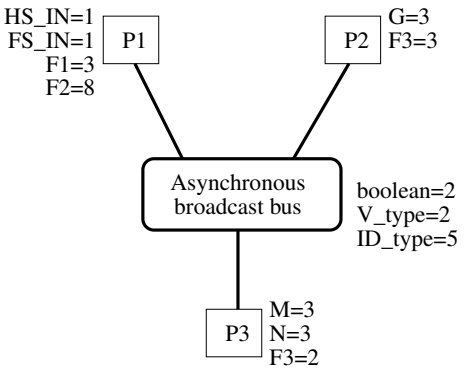


Figure 5.4: Example of a hardware architecture [Potop-Butucaru et al., 2009].

The hardware architecture on which our example is implemented is pictured in Fig. 5.4. This architecture has 3 processors (P_1, P_2, and P_3) connected to a asynchronous broadcast bus. The architecture model is decorated with some timing and allocation information presented below and collectively referred to as *non-functional properties*.

Non-functional properties In the CG formalism used in this thesis, non-functional properties describe duration of the various computation and communication operations and the allocation constraints. For each function node $n \in \mathcal{N}^F$ and each processor $P \in Procs(Arch)$ we provide the *duration* of n on P. We assume this value is obtained through a *worst-case execution time (WCET)* analysis, and denote it $WCET(n,P)$. This value is set to ∞ when execution of n on P is not possible. Similary, for each data type we provide the *worst-case communication time* needed to transmit one such value over the bus. This value is denoted $WCCT(\mathcal{D})$. We assume that this value is always finite.

Note that the $WCET$ information may implicitly define *allocation constraints*, as $WCET(n,P) = \infty$ prevents n from being allocated on P. Such allocation constraints are meant to represent hardware platform constraints, such as designer-imposed placement constraints. For instance, in Fig. 5.4, while the function HS_IN can only be executed on P_1, function $F3$ can be executed (with different costs) on both P_2 and P_3. We assume that

local communications (that do not use the bus) take no time.

Variations of our mapping algorithms can also handle other non-functional constraints, such as periodicity, start date, and deadline requirements [Carle et al., 2012], but we do cover these aspects here.

5.1.2 Off-line scheduling of CG specifications

Given a CG specification, the off-line real-time scheduling problem we consider is that of synthesizing a *scheduling table* (also known as *reservation table*) that assigns real-time dates and execution resources (computation or communication resources) to each element of the CG. The scheduling table describes the real-time scheduling of one execution instant of the synchronous specification (possibly pipelined). A full execution is obtained by indefinitely repeating the pattern provided by the scheduling table.

5.1.2.1 Scheduled clocked graphs

The scheduling tables output by our algorithms are represented using *scheduled clocked graphs*, which are an extension of the CG formalism. Given a CG specification, a scheduled clocked graph $\mathcal{S}$ over it defines the following supplementary structures:

- An allocation of the delays to processors determining on which processor the delay value is stored between execution cycles, and at which date the bookkeeping operations associated with the delay are performed: $\mathcal{S}_\Delta : \mathcal{N}^\Delta \rightarrow \mathcal{P} \times \mathbb{N}$
- A set of scheduled functions assigning a processor and a real-time date (an integer) to each computation node of the data-flow: $\mathcal{S}_\mathcal{F} : \mathcal{N}^\mathcal{F} \rightarrow \mathcal{P} \times \mathbb{N}$
- A set of scheduled communications assigning to each arc of the data-flow the sender processor, a real-time date, and an effective communication clock (recall that in this section architectures have a single broadcast bus):

$$\mathcal{S}_\mathcal{A} : \mathcal{A} \rightarrow \mathcal{P} \times \mathbb{N} \times \mathcal{C}$$

- For each clock c of a data-flow element (node or arc), a set of scheduled communications assigning to each element of its support $supp(c)$ a sender processor, a real-time date, and an effective communication clock:

$$\mathcal{S}_x : supp(x) \rightarrow \mathcal{P} \times \mathbb{N} \times \mathcal{C}$$

We say that the scheduled CG $\mathcal{S}$ is partial when any of its defining functions is partial. This may be the case during scheduling, when all data-flow elements have not been mapped.

5.1.2.2 Real-time scheduling problem

The real-time mapping problem we consider is a bi-criteria optimization problem: Given a correct CG specification consisting of a functional specification and a non-functional specification, synthesize a scheduling table (a scheduled clocked graph) that minimizes the *makespan* and maximizes *throughput* with priority given to makespan
[Carle and Potop-Butucaru, 2011]. Makespan in this context means the time difference between the start of the first scheduled operation and the end of the last scheduled operation in one execution cycle, while throughput means the number of execution cycles started per time unit. It can be different from the inverse of the makespan because we allow one cycle to start before the end of previous ones, provided that operation dependencies are satisfied.

The allocation and scheduling problem being NP-complete, we do not aim for optimality. Instead, we rely on low-complexity heuristics that allow us to handle large numbers of resources with high temporal precision.

Mapping and code generation is realized in three steps: The first step produces a makespan-optimizing scheduling table using the algorithms described in Section 5.2.3. The second step uses the software pipelining algorithms of [Carle and Potop-Butucaru, 2011] (not detailed here) to improve the throughput while not changing the makespan. Finally, once a scheduling table is computed, it can be implemented in a way that preserves its real-time properties.

5.1.2.3 Consistency of a scheduled clocked graph

Notations Given a scheduled clocked graph $\mathcal{S}$, we denote with:

- t_x the real-time date associated by $\mathcal{S}$ to any computation node, arc, or element of a support list x.
- $Res(x)$ the processor associated to each node, arc, or element of a support list x (the execution processor for data-flow functions, the storage processor for delays, and the sender processor for arcs and support elements).

- $e\_clk(x)$ the effective communication clock associated with an arc or element of a support list x.
- $WCCT(x)$ the duration of the scheduled communication of an arc or element of a support list x. Note that $WCCT(x) = WCCT(\mathcal{D}_x)$.

The execution condition defining the execution cycles where the value of an output port o is sent on the communication media before date t is denoted $clk^{\mathcal{S}}(o,t,Comm(Arch))$, and is formally defined as the union of all the clocks $e\_clk(x)$, where e ranges over:

- the arcs $a \in \mathcal{A}$ with $src(a) = o$ that have been scheduled such that $t_a + WCCT(\mathcal{D}_a) \leq t$.
- the elements $o@c$ of $supp(y)$ (for some arc or node y) that have been scheduled ($\mathcal{S}_x(o@c)$) such that $t_{o@c} + WCCT(o@c) \leq t$.

Obviously, $clk^{\mathcal{S}}(o,t,Comm(Arch))$ is the execution condition giving the cycles where o is available system-wide at all dates $t' \geq t$.

Assuming o is a port of node n, we also define the execution condition $clk^{\mathcal{S}}(o,t,P)$ defining the cycles where o is available on P at date t:

- If n is not allocated on P ($Res(n) \neq P$), then $clk^{\mathcal{S}}(o,t,P) = clk^{\mathcal{S}}(o,t,Comm(Arch))$
- If n is a delay node allocated on P ($Res(n) = P$), then $clk^{\mathcal{S}}(o,t,P) = clk(n)$, meaning that the value is available from the beginning of all execution instants of n.
- If n is a computation node allocated on P at date t_n, then $clk^{\mathcal{S}}(o,t,P)$ is $clk(n)$ if $t \geq t_n + WCET(n,P)$, and *false* if not.

If c is a clock with $c \leq clk(n)$, then we denote with $ready\_date(P,o,c)$ the minimum t such that $clk^{\mathcal{S}}(o,t,P) \geq c$, and with $ready\_date(Comm(Arch),o,c)$ the minimum date t such that $clk^{\mathcal{S}}(o,t,Comm(Arch)) \geq c$.

Note that $clk^{\mathcal{S}}(o,\infty,R)$ is the clock giving the instants where o becomes available at some point on resource R.

Consistency properties The scheduling tables we build satisfy a number of properties ensuring that executable code can be generated for them that is both functionally and temporally correct:

- **Causality:** To ensure causal correctness the schedule must ensure in a static fashion that when an operation (computation or communication) is using the value of an output port o at time t on execution condition (clock) c, the port value has been either computed locally or transmitted on the communication media at a previous date, and on a greater clock. Formally:

  - If $\mathcal{S}_{\mathcal{F}}(n) = (P,t)$ is defined for a function node n, then:
    - $clk^{\mathcal{S}}(o,t,P) \geq c$ for all $o@c \in supp(n)$
    - $clk^{\mathcal{S}}(o,t,P) \geq c$ for all $o@c \in supp(a)$ if $dest(a)$ is an input port of n
    - $clk^{\mathcal{S}}(src(a),t,P) \geq clk(a)$ for all a with $dest(a)$ being an input port of n
  - To derive the rule ensuring that a delay has enough input at the end of an instant, we simply set in the previous rules the date to ∞. More precisely, if $\mathcal{S}_{\Delta}(n) = P$ is defined for a delay node δ, then:
    - $clk^{\mathcal{S}}(o,\infty,P) \geq c$ for all $o@c \in supp(\delta)$
    - $clk^{\mathcal{S}}(o,\infty,P) \geq c$ for all $o@c \in supp(a)$ if $dest(a)$ is an input port of δ
    - $clk^{\mathcal{S}}(src(a),\infty,P) \geq clk(a)$ for all arc a with $dest(a)$ being an input port of δ
  - If $\mathcal{S}_{\mathcal{A}}(a) = (P,t_a,c_a)$ is defined for an arc a with $c_a \neq false$ and if n_a is the source node of a, then:
    - $clk^{\mathcal{S}}(o,t,Comm(Arch)) \geq c$ for all $o@c \in supp(a)$
    - $\mathcal{S}_{\mathcal{F}}(n_a)$ is defined and $Res(n_a) = P$ and $t_{n_a} + WCET(n_a,P) \leq t_a$

- **Sequential use of resources:** Two reservations of the same resource must not overlap in time, unless their corresponding computations or communications have exclusive execution conditions. Formally:

  - *On processors*: if n_1 and n_2 are different scheduled function nodes with $clk(n_1) \wedge clk(n_2) \neq false$, then either $t_{n1} \geq (t_{n2} + WCET(n_2,P))$ or $t_{n2} \geq (t_{n1} + WCET(n_1,P))$.
  - *On communication media*: if x_1 and x_2 are different scheduled arcs or elements of support lists with $e\_clk(x_1) \wedge e\_clk(x_2) \neq false$, then either $t_{x1} \geq (t_{x2} + WCCT(x_2))$ or $t_{x2} \geq (t_{x1} + WCCT(x_1))$.

- **Worst-case reservations:** We reserve for each data-flow node a time interval having a length equal to its WCET (provided in the non-functional specification), plus the worst-case time needed to compute its execution condition. Similarly, we reserve for every communication, on each resource along the communication path, a time interval having a length equal to the worst-case communication time (WCCT) of the communication.

5.1.2.4 Makespan-optimizing scheduling algorithm

The algorithms of this section are used to transform a CG specification into a scheduled CG specification. This basically consists in building a scheduling table. For instance, Fig. 5.5 provides the graphical representation of the scheduling table produced for the example in Figures 5.3 and 5.4. Our tools can further improve the throughput of this table through software pipelining methods [Carle and Potop-Butucaru, 2011].

The scheduling algorithm, whose top-level routine is Procedure 1, follows a classical list scheduling approach. The nodes of the data-flow graph are considered one by one, in an order consistent with the partial order determined by the data dependency arcs not originating in an output of a delay node. When a node is considered it is scheduled, along with the communication operations needed to bring its input data on the processor where it is executed. The allocation and scheduling decisions taken for a node and for the associated communications are never changed during the scheduling of subsequent nodes (there is no backtracking).

The body of the **while** loop allocates and schedules a single node n, along with the communications needed to gather the input data of n. Scheduling follows a classical ASAP (as soon as possible) policy by mapping each operation at the earliest possible date. Allocation is performed automatically by attempting to schedule n on each of the processors that can execute it. Among all the possible allocations of n, Procedure 1 chooses the one minimizing the date at which n terminates. The length of the final scheduling table gives the worst-case duration of one cycle (*i.e.* the makespan). Function **MapDelayCommunications** maps the communications of the outputs of delay blocks.

Scheduling a node n on a given processor P is realised by Procedure 2. The first call to function **ScheduleSupport** (whose pseudocode is provided in [Potop-Butucaru et al., 2009]) schedules the bus communications of the data needed for the computation of $clk(n)$ (*i.e.*

Procedure 1 SchedulingDriver

Input: $\mathcal{G}$: Clocked Graph $(\mathcal{N},\mathcal{A})$
$Arch$: Architecture description
Output: S : Scheduled CG (full schedule of the application)
1: $S \leftarrow \emptyset$
2: **while** there exists $n \in \mathcal{N}$ not yet scheduled **do**
3: choose $n \in \mathcal{N}$ unscheduled whose predecesors have already been scheduled
4: **for all** $P \in Procs(Arch)$ with $WCET(n,P) \neq \infty$ **do**
5: (S_P, End_P) := **ScheduleNodeOnProcessor**(S,n,P);
6: **end for**
7: Assign to S the S_P with minimal End_P
8: **end while**
9: S := **MapDelayCommunications**(S);

its support $supp(clk(n))$, as defined in Section 5.1.1.1) and which is not yet present on P. The **forall** loop schedules the communications related to the acquisition of the data-flow inputs of n (the clock governing its transmission and the data itself). Once communications are scheduled, we schedule the node n at the earliest date after the date where all needed data is available using function **ReserveFirstAvailable**. Function $ReserveFirstAvailable(\mathcal{S},R,t,clk,d)$ reserves the first slot of duration d (time interval) available on resource R after date t and on the condition clk. To allow an efficient search of the first available interval to reserve, the data structure storing the partial schedules also stores the set of free intervals of each resource R.

Procedure 2 ScheduleNodeOnProcessor

Input: S : Scheduled CG (partial schedule)
P : Target processor
n : Data-flow node to schedule
Output: S: Scheduled CG (partial or not)
$EndDate$: The date on P where n completes its execution
1: $(S, EndDate)$:= **ScheduleSupport**$(S,P,supp(n))$;
2: **for all** incoming arc a of n **do**
3: (S,t) := **ScheduleSupport**$(S,P,supp(a) \cup src(a)@clk(a))$;
4: $EndDate$:= **max**$(EndDate,t)$;
5: **end for**
6: $(S, EndDate)$:= **ReserveFirstAvailable**$(S,P,EndDate,clk(n),WCET(n,P))$;
7: $EndDate := EndDate + WCET(n,P)$;

The data structure used by the actual scheduling algorithms not only deal with real-time scheduling, but also associate software variables to the data-flow output ports and delay nodes, so as to help with code generation. For instance, for each output port of a node, and for each processor where the value of this port is used, one variable is allocated during scheduling. This is why the algorithms defined in the following sections structure

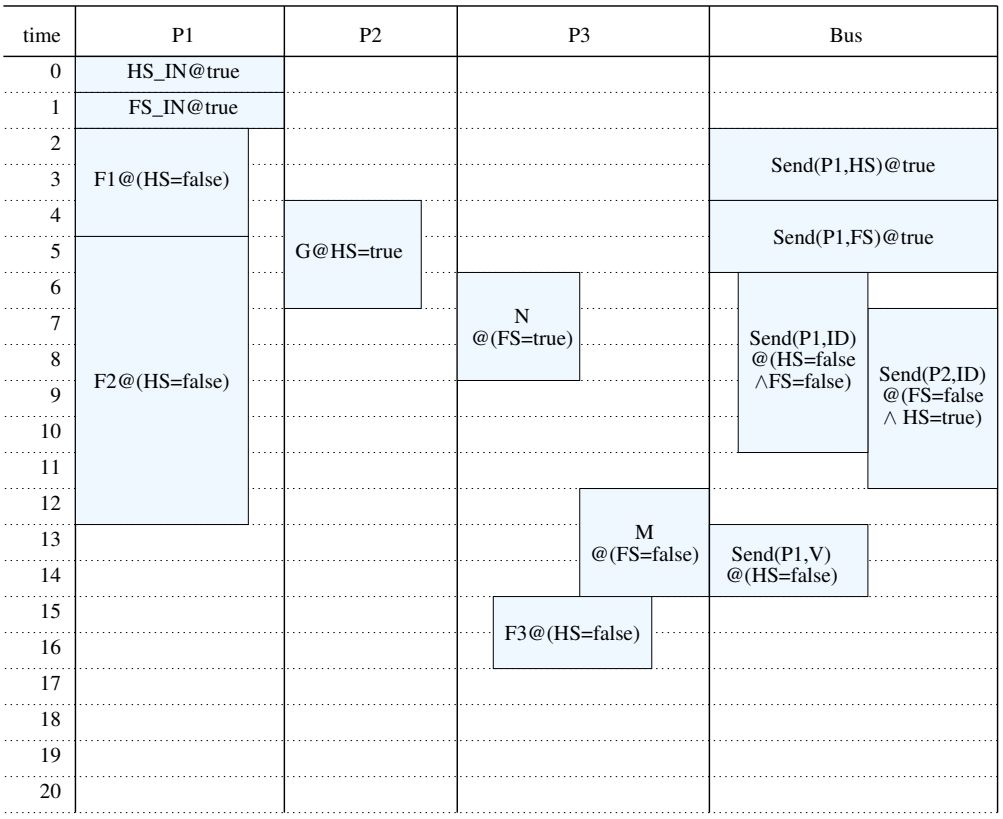


Figure 5.5: Real-time schedule table generated by these algorithms for the example in Figures 5.3 and 5.4

a Scheduled CG specification S as a tuple formed of 7 data structures:

$$S =< ScheduleLength, VariableAllocations, ProcessorSchedules,$$
$$ProcessorsFreeIntervals, CommunicationSchedule, CommunicationFreeIntervals >$$

where:

- *ScheduleLength* is the current length of the scheduling table. Scheduling starts with an empty scheduling table of length 0, which is incrementally filled as the computations and the associated communications are reserved time intervals on the various resources. There is no limit on the length of the scheduling table, so the mapping process cannot fail.

- *VariableAllocations* associates to each output port of a node the set of processors on which a variable corresponding to the port must be allocated.

- *ProcessorSchedules* is the set of reservations made on each processor $P \in Procs(Arch)$, *i.e.* the set of data-flow nodes that were already scheduled on each processor.

- *ProcessorsFreeIntervals* is the set of free (not yet allocated) time intervals on each processor schedule. The intervals of this set can be used to schedule other data-flow nodes.

- *CommunicationSchedule* is the set of reservations made on the communication bus $Comm(Arch)$.

- *CommunicationFreeIntervals* is the set of free (not yet allocated) time intervals on $Comm(Arch)$, that can be used to schedule other communications.

5.2 Static (off-line) mapping onto MPPA architectures

5.2.1 AAA for NoC-based MPPA: The problem

Existing implementations of the AAA methodology – SynDEx and the CG-based scheduling algorithms described above – do not allow the efficient allocation and scheduling of applications onto NoC-based many-cores, regardless of the type of NoC arbitration and routing. This is mainly due to limitations in the modeling and handling of the complex communication media. In particular:

- Existing implementations of the AAA methodology only consider *sequential* communication media, which can be modeled as classical sequential resources for scheduling purposes. Meanwhile, a NoC is formed of a large number of resources working in *parallel* and allowing multiple communication flows to traverse the NoC at the same time.

- SynDEx allows the modeling of architectures with multiple communication resources, where a piece of data may need to be routed along several communication resources in order to reach its destination. However, it assumes that processors possess unlimited local storage capabilities, which allows the use of a *store-and-forward* buffering policy along multi-hop routes, which simplifies scheduling. Meanwhile, NoCs have limited internal buffering, and usually rely on *wormhole*

buffering policies which, as explained in Section 2.1.3.2, result in significant synchronization between resource reservations on different NoC resources.

- Both SynDEx and the existing CG toolset consider the mapping of coarse-grain parallel applications onto classical multiprocessor/distributed architectures. But we already explained in Section 1.1.4 that when mapping onto MPPAs, both application and architecture usually feature a finer-grain parallelism (specification- and architecture-level). This requires a tighter control of timing, concerning both NoC control and the execution inside a computing tile. In turn, this requires precise control over the contention points, such as NoC routers, memory banks, *etc.* and a fine-grain allocation of these resources.

The remainder of this thesis is dedicated to extending the CG formalism and CG-based implementation of the AAA methodology to allow the efficient off-line real-time scheduling and code generation onto the MPPA architecture presented in Chapter 4. In doing so, we face 2 main difficulties:

- The definition of a hardware specification model that takes into account the specificities of the target architecture, including the timing model by determining the cost (in clock cycles) of the various NoC operations.

- Extending the existing scheduling and code generation algorithms to take into account the new hardware description. These algorithms must have very low complexity, so that they scale up to the large numbers of resources of a typical NoC, while at the same time retaining a high timing precision (and guaranteeing functional and temporal correction).

The resulting AAA flow is pictured in Fig. 1.3.

Limitations This PhD thesis focuses on the modeling and handling of NoC resources for real-time scheduling. Our main objective was to show that table-based off-line scheduling heuristics have the potential to scale up, allowing the use of architecture models exposing all possible contention points of the NoC.

To achieve this objective in only 3 years, we have made some simplifications to the scheduling problem: The main simplification is that we do not take into account data-

dependent conditional control. Its implementation in hardware is not completed, as explained in Section 4.2.1. Also, given the absence of hardware support, we have also preferred not to include conditional execution treatment in the scheduling algorithms defined later in this chapter. This has the advantage of simplifying the presentation of the algorithms.

The second limitation of our work concerns the modeling of the computing tiles, which detailed in the next section.

5.2.2 Extension of the CG format

5.2.2.1 Modeling of MPPA resources

To allow off-line mapping onto our architectures, we need to identify the set of *abstract* computation and communication resources that are considered during allocation and scheduling. The choice of resources must allow a precise timing characterization while preserving tractability of the scheduling problem.

Formally, we represent an MPPA architecture with a pair $Arch = (NoC(Arch), Tiles(Arch))$. The set $NoC(Arch)$ contains the communication resources of the NoC, and $Tiles(Arch)$ contains the computation resources associated with tiles. We picture in Fig. 5.6 the resources of a 3x3 MPPA.

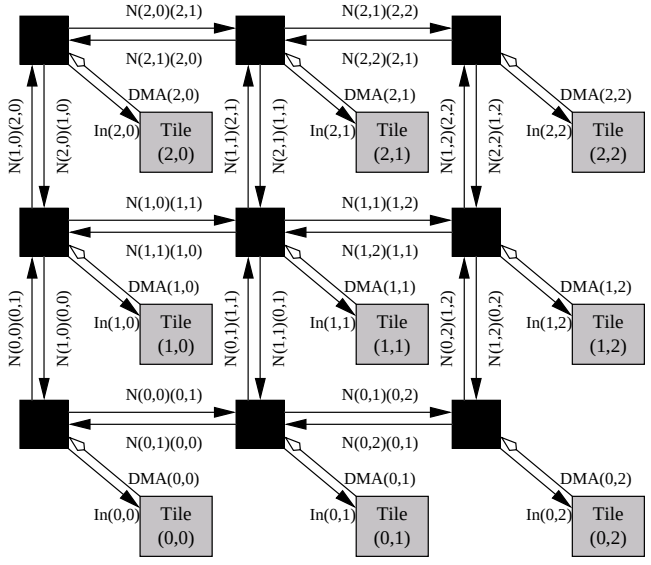


Figure 5.6: Hardware resource modeling for our architecture

NoC ressources: Classical communication media (*e.g.* buses, shared RAMs) considered in previous work [Grandpierre and Sorel, 2003, Potop-Butucaru et al., 2009] have two properties facilitating mapping:

- Each media can be seen as a sequential communication resource.
- When a communication follows a route involving multiple buses, data can be temporarily stored between two routing steps (store-and-forward buffering policy).

These two hypotheses are not true in NoCs. First, the transmission time for a given amount of data depends on the distance between the source and destination tiles, which in turn depends on the number of multiplexers that are crossed during transmission. For this reason, we need to associate one communication resource to each of the multiplexers of NoC routers, and to each DMA unit allowing command queuing. We use the term *segment* to refer to these communication resources. As pictured in Fig. 5.6, our architecture model contains 3 types of segments:

- Inter-router segments correspond to the links between NoC routers and their command multiplexers. We denote with $N(i,j)(k,l)$ the segment going from tile (i,j) to tile (k,l).
- Tile input segments correspond to the links going from routers to their local tiles (and their command multiplexers). We denote with $In(i,j)$ the tile input segment of tile (i,j).
- Tile output segments correspond to the links going from tiles to local routers. These links are not controlled by multiplexers, but by the DMA of the local tile. The tile output segment of tile (i,j) is denoted $DMA(i,j)$.

Under this resource model, a data transfer between two tiles is performed by a set of segments called the *communication path*. All communication paths are formed according to the X-first routing policy. Recall that modeling is done only for command NoC resources. No resource modeling, nor scheduling analysis, is needed for the response NoC. Each NoC segment has a buffering capability of only 3 flits.

The set $NoC(Arch)$ is formed of all the segments of the NoC. In Fig. 5.6, the NoC description consists in 42 segments.

Tile ressources: This PhD thesis focuses on the modeling and handling of NoC resources. To this end, we consider a resource model that simplifies as much as possible the representation of the computing tiles.

The main simplification is to consider each tile as a single computing resource. All the CPUs of the tile (1 upto 16 in our evaluations) are seen as a very fast computing resource. This means that computation operations (data-flow nodes) will be allocated to the tile as if it were a sequential processor, but each operation is in fact parallel code running on all the processors of a tile. In Fig. 5.6 there are just 9 tile resources representing 144 CPUs. This simplification largely reduces the complexity of our presentation, and also satisfies our evaluation needs, given that the 2 applications used as examples can be organized into operations that are easily parallelized.

While not explicitly represented in the CG hardware model, memory organization is considered for code generation.

Thus, in our model, $Tiles(Arch) = \{T_0, ..., T_{Y \times X - 1}\}$ where X and Y are the dimensions of the MPPA.

5.2.2.2 Memory footprint specification

Our MPPA architecture features a complex memory organization including multi-bank RAMs, which we must take into account through explicit allocations of data and code onto the RAM banks.

To allow this, we extend the existing CG functional specification as follows:

- To each data type $\mathcal{D}$ a CG specification associates a worst-case size $sizeof(\mathcal{D})$. This value will be used in Section 5.2.2.3 to compute the worst-case communication time for a piece of data of that type.

- To each data-flow function we associate the number of RAM banks it needs in order to allow parallel execution while respecting the provided WCET figure.

5.2.2.3 Non-functional properties

Worst-case computation durations For each data-flow node n and each MPPA tile T the CG specification defines $WCET(n,T)$, which must be a safe upper bound for the

WCET of n on T. Note that the WCET values we require are for *parallel* code running on all the 16 processors of a tile, which can be computed using the technique of [Puaut and Potop-Butucaru, 2013].

Even though the tiles of our MPPA architecture are largely identical, we made the choice of defining one WCET value per tile. This allows a simple expression of allocation constraints which specify on which tiles a given data-flow block can be executed. Allocation constraints can be used to confine an application to part of the MPPA, leaving the other tiles free to execute other applications. Furthermore, using one WCET value per tile allows our algorithms to handle heterogenous many-cores (but we did not investigate this issue).

Worst-case communication durations The previous section explained that a worst-case size $sizeof(\mathcal{D})$ is provided by the CG specfication for each data type $\mathcal{D}$.

All inter-tile data transmissions are performed using the DMA units. If a transmission is not blocked on the NoC, then its duration at the sender side only depends on the size of the transmitted data. The exact formula is

$$d = sizeof(\mathcal{D}) + \lceil sizeof(\mathcal{D})/MaxPayload \rceil * PacketHeaderSize$$

where d is the duration in clock cycles of the DMA transfer from the start of the transmission to the moment where a new transmission can start, $\mathcal{D}$ is the type of the transmitted data, *MaxPayload* is the maximum payload of a NoC packet produced by the DMA (in 32-bit words), and *PacketHeaderSize* is the number of cycles that are lost for each packet in the chosen NoC. These values are architecture constants. For instance, the architecture used in this thesis has *MaxPayload*=16 flits=64 bytes and *PacketHeaderSize*=4 flits=16 bytes.

In addition to this transmission duration, we must also account in our computations for:

- The DMA transfer initiation, which consists in 3 uncached RAM accesses plus the duration of the DMA reading the payload of the first packet from the data RAM. This cost is over-approximated as 30 cycles.
- The latency of the NoC, which is the time needed for one flit to traverse the path from source to destination. This latency is of $3 * n$, where n is the number of NoC

segments on the route of the transmission. The constant 3 corresponds here to the number of clock cycles needed to traverse a NoC segment in the absence of contentions.

5.2.3 Makespan-optimizing scheduling

The makespan-optimizing scheduling routine we use on the MPPA is a variant of the one described in Section 5.1.2, extended with support for scheduling NoC communications, but without support for conditional control.

Like its predecessor, it works by building a global scheduling table covering all MPPA resources (NoC segments and tiles). It uses a non-preemptive scheduling model for the data-flow nodes, because preemptions would introduce important temporal imprecision (through the use of interrupts). At the same time, it uses a preemptive scheduling model for NoC communications, because data communications over the NoC are naturally divided into packets that are individually scheduled by the NoC multiplexer programs, allowing a form of pre-computed preemption.

For each data-flow node our scheduling routine reserves exactly one time interval on one of the tiles. For every communication between two tiles, it reserves one or more time intervals on each segment along the communication path between the two tiles, starting with the DMA of the source tile, and continuing with the NoC multiplexers (recall that the route is fixed under the X-first routing policy). Scheduling is done under an ASAP (as soon as possible) policy.

The top-level scheduling routine is Procedure 3. It is very similar with the original routine Procedure 1 of page 107. The single difference is that the choice between possible allocations of a given data-flow node depends not only on the end date of the node, but on a more elaborate cost function. This cost function, which we seek to minimize, should be chosen so that the final length of the scheduling table is minimized (this length gives the execution cycle makespan). Our choice of cost function combines the end date of the node in the schedule (with 95% weight) and the maximum occupation of a CPU in the current scheduling table (with 5% weight). We found it to produce shorter scheduling tables than the cost function based on end date alone (as used in [Grandpierre and Sorel, 2003, Potop-Butucaru et al., 2009]). This is due to the fact that our cost function discourages the scattering of computations onto a large number of processors, which ultimately reduces

synchronization cost.

Procedure 3 SchedulingDriverMPPA

Input: $\mathcal{G}$: Clocked Graph $(\mathcal{N},\mathcal{A})$
$Arch$: Architecture description
Output: S : Scheduled CG (full schedule of the application)
1: $S \leftarrow \emptyset$
2: **while** there exists $n \in \mathcal{N}$ not yet scheduled **do**
3: choose $n \in \mathcal{N}$ unscheduled whose predecesors have already been scheduled
4: **for all** $T \in TilesArch$ with $WCET(n,T) \neq \infty$ **do**
5: (S_T, End_T) := **ScheduleNodeOnTile**(S,n,T);
6: $Cost_T := \frac{95}{100} * End_T + \frac{5}{100} * MaxTileOccupation(S_T)$;
7: **end for**
8: Assign to S the S_T with minimal $Cost_T$
9: **end while**
10: S := **MapDelayCommunications**(S);

Similarly, Procedure 4 has the same global structure as Procedure 2, page 107, but adapted to our needs through the handling of NoC communications and through the simplification concerning the absence of data-dependent control. Indeed, no reference is made here to the scheduling of clock supports. Instead, we need to determine for each piece of data that needs to be transmitted on the NoC which path the transmission must take (following the X-first routing policy), and then schedule the communication over the resources of this route. Once all needed data is present on the tile, the node is scheduled at the earliest possible date.

Procedure 4 ScheduleNodeOnTile

Input: S : Scheduled CG (partial schedule)
T : Target tile
n : Data-flow node to schedule
Output: S: Scheduled CG (partial or not)
t: The date on T where n completes its execution
1: $t \leftarrow 0$
2: **for all** incoming arc a of n **do**
3: let n' be the (already mapped) data-flow node producing $src(a)$
4: let T' be the tile on which n' has been allocated
5: let t' be the end date of n' on T'
6: **if** $T' \neq T$ **then**
7: $Path \leftarrow$**GetXFirstPath**(P',P)
8: (S,t') := **MapCommunicationOnPath**$(S, Path, t', sizeof(\mathcal{D}_a))$
9: **end if**
10: $t := max(t,t')$
11: **end for**
12: (S,t) := **ReserveFirstAvailable**$(S,T,t,true,WCET(n,T))$;
13: $t := t + WCET(n,T)$;

5.2.3.1 Mapping NoC communications

The most delicate part of our scheduling routine is the communication mapping function **MapCommunicationOnPath**. When a node is mapped on a tile, this function is called once for each of the input dependencies of the node, if the dependency source is on another tile and if the associated data has not already been transmitted.

Procedure 5 MapCommunicationOnPath

Input: S : Scheduled CG (partial schedule)
Path : list of NoC segments (the communication path)
StartDate : date after which the data can be sent
DataSize : worst-case data size (in 32-bit words)
Output: S: Scheduled CG (partial or not)
EndComm : the end date of the communication

1: **for** i := 1 to ***length***(*Path*) **do**
2: /*Identify the unreserved time intervals on segment i and
3: compensate for the delays induced by segment buffers. */
4: *ShiftSize*[i] := $(i-1) * \underline{SegmentBufferSize}$;
5: *FreeIntervalList*[i] := ***GetIntervalList***(S, *Path*[i], *StartDate*);
6: *FreeIntervalList*[i] := ***ShiftLeftIntervals***(*FreeIntervalList*[i],*ShiftSize*[i]);
7: **end for**
8: /* Determine time intervals that are free along the path. */
9: *PathFreeIntervalList* :=***IntersectIntervals***(*FreeIntervalList*);
10: /* Reserve intervals for the transmission of data and lock. */
11: (*IntervalsForData*,*NewFreeIntervalList*,*NewScheduleLength*) :=
ReserveIntervals(*DataSize*,*PathFreeIntervalList*, ***length***(S));
12: (*IntervalForLock*,*NewIntervalList*,*NewScheduleLength*) :=
ReserveIntervals(1,*NewFreeIntervalList*, *NewScheduleLength*);
13: *ReservedIntervals* := ***AppendToList***(*IntervalsForData*,*IntervalForLock*);
14: **for** i := 1 to ***length***(*Path*) **do**
15: /*Remove the compensation added in the beginning of the algorithm.
16: Separately for each segment. */
17: *SegmentReservedIntervals*[i] := ***ShiftRightIntervals***(*ReservedIntervals*,*ShiftSize*[i]);
18: **end for**
19: /*If reservations go past the current end of the scheduling table,
20: update the scheduling table with the new length. */
21: **if** *NewScheduleLength* $>$ ***length***(S) **then**
22: S := ***IncreaseLength***(S,*NewScheduleLength*);
23: **end if**
24: *EndComm* := *NewScheduleLength*;
25: /*Update the lists of reservations and the lists of free intervals
26: for all segments along the path. */
27: S := ***UpdateSchedulingTable***(S,*Path*,*SegmentReservedIntervals*);

Fig. 5.7 presents a (partial) scheduling table produced by our mapping routine. We shall use this example to give a better intuition on the functioning of our algorithms. We assume here that the execution of operation f produces data x which will be used by g. Our scheduling table shows the result of mapping operation g onto $Tile(2,2)$ (which also

requires the mapping of the transmission of x) under the assumption that all other computation operations (f, h) and data transmissions (y, z, u) were already mapped. Fig. 5.7 uses a lighter color to identify reservations made as part of the mapping of g.

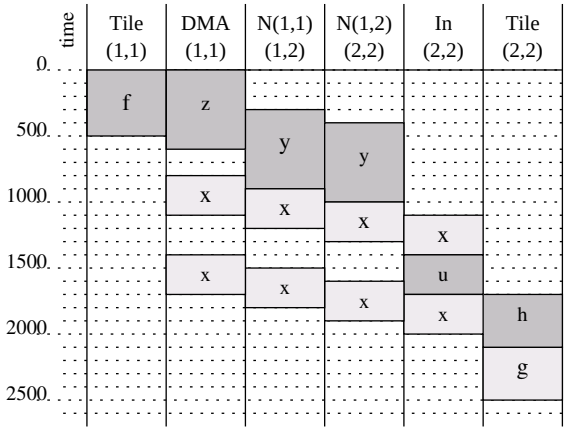


Figure 5.7: Scheduling table covering one communication path on our NoC. Only the 6 resources of interest are represented (out of 70)

As part of the mapping of g onto $Tile(2,2)$, function **MapCommunicationOnPath** is called to perform the mapping of the communication of x from $Tile(1,1)$ to $Tile(2,2)$. The parameters of its call are the schedule itself, *Path*, *StartDate*, and *DataSize*. Parameter *Path* is the list formed of resources $DMA(1,1)$, $N(1,1)(1,2)$, $N(1,2)(2,2)$, and $In(2,2)$ (the transmission route of x under the X-first routing protocol). Parameter *StartDate* is set to be the end date of node f (in our case 500), and *DataSize* is the worst-case size of the data associated with the data dependency (in our case 500 32-bit words). Time is measured in clock cycles.

To minimize the overall time reserved for a data transmission, we shall require that it is never blocked waiting for a NoC resource. For instance, if the communication of x starts on segment $N(1,1)(1,2)$ at date t, then on segment $N(1,2)(2,2)$ it must start at date $t + \underline{SegmentBufferSize}$, where $\underline{SegmentBufferSize}$ is a platform constant defining the time needed for a flit to traverse one NoC segment. In our NoC this constant is 3 clock cycles (in Fig. 5.7 we use a far larger value of 100 cycles, for clarity).

Building such synchronized reservation patterns along the communication routes is what function **MapCommunicationOnPath** does. It starts by obtaining the lists of free time intervals of each NoC segment along the communication path, and realigning them by subtracting $(i-1) * \underline{SegmentBufferSize}$ from the start dates of all the free intervals of

the i^{th} resource, for all i. Once this realignment is done on each segment by function ***ShiftLeftIntervals***, finding a reservation along the communication path amounts to finding time intervals that are unused on all resources. To do this, we start by performing (in line 9 of function **MapCommunicationOnPath**) an intersection operation returning all realigned time intervals that are free on all resources. In Fig. 5.7, this intersection operation produces (prior to the mapping of x) the intervals [800,1100) and [1400,2100]. The value 2100 corresponds to the length of the scheduling table prior to the mapping of g.

We then call function ***ReserveIntervals*** twice, to make reservations for the data transmission and for the lock command packet associated with each communication. These two functions produce a list of reserved intervals, which then need to be realigned on each resource. In Fig. 5.7, these 2 calls reserve the intervals [800,1100), [1400,1700), and [1700,1704). The first 2 intervals are needed for the data transmission, and the third is used for the lock command packet.

5.2.3.2 Multiple reservations

Communications are reserved at the earliest possible date, and function ***ReserveIntervals*** allows the fragmentation of a data transmission to allow a better use of NoC resources. In our example, fragmentation allows us to transmit part of x before the reservation for u. If fragmentation were not possible, the transmission of x should be started later, thus delaying the start of g, potentially lengthening the reservation table.

Fragmentation is subject to restrictions arising from the fact that communications are packetized. More precisely, an interval cannot be reserved unless it has a minimal size, allowing the transmission of at least a packet containing some payload data.

Function **ReserveIntervals** performs the complex translation from data sizes to packets and interval reservations. We present here an unoptimized version that facilitates understanding. This version reserves one packet at a time, using a free interval as soon as it has the needed minimal size. Packets are reserved until the required *DataSize* is covered. Like for tasks, reservations are made as early as possible. For each packet reservation the cost of NoC control (under the form of the *PacketHeaderSize*) must be taken into account.

If the current scheduling table does not allow the mapping of a data communication, function **ReserveIntervals** may lengthen it.

Procedure 6 ReserveIntervals

Input: *DataSize* : worst-case size of data to transmit
FreeIntervalList : list of free intervals before reservation
ScheduleLength : schedule length before reservation
Output: *ReservedIntervalList* : reserved intervals
NewIntervalList : list of free intervals after reservation
NewScheduleLength : schedule length after reservation

```
1:  NewIntervalList := FreeIntervalList
2:  ReservedIntervalList := ∅
3:  /* Consider the free intervals one by one and reserve as much as
4:     possible of each until there is no more space or the
5:     communication need has been covered. */
6:  while DataSize > 0 and NewIntervalList≠ ∅ do
7:     ival := GetFirstInterval(NewIntervalList);
8:     NewIntervalList := RemoveFirstInterval(NewIntervalList);
9:     if IntervalEnd(ival)==ScheduleLength then
10:       /* ival can be extended indefinitely along with the schedule length. */
11:       RemainingIvalLength := ∞;
12:    else
13:       RemainingIvalLength := length(ival);
14:    end if
15:    ReservedLength := 0;
16:    while RemainingIvalLength > MinPacketSize and DataSize > 0 do
17:       /*Reserve place for data packets, one at a time
18:         (clear, but suboptimal code).*/
19:       PacketLength := min(DataSize + PacketHeaderSize, RemainingIvalLength,MaxPacketSize);
20:       RemainingIvalLength -= PacketLength;
21:       DataSize -= PacketLength - PacketHeaderSize;
22:       ReservedLength += PacketLength
23:    end while
24:    ReservedInterval := CreateInterval(start(ival), ReservedLength);
25:    ReservedIntervalList := AppendToList(ReservedIntervalList,ReservedInterval);
26:    if length(ival) - ReservedLength > MinPacketLength then
27:       NewIntervalList := InsertInList(NewIntervalList,
                CreateInterval(start(ival)+ReservedLength, length(ival)-ReservedLength));
28:    end if
29:    NewScheduleLength := max(ScheduleLength,end(ival));
30: end while
```

5.3 Automatic code generation

The final step in our CG-based flow is code generation, which transforms the scheduling table, where synchronization is time-based, into multi-threaded C code with lock-based synchronization, plus the communication programs that control the behavior of the NoC.

Executable code is generated as follows: One sequential execution thread is generated for each tile and for each NoC segment corresponding to a NoC multiplexer (resources $N(i,j)(k,l)$ and $In(i,j)$ in the architecture model of Section 5.2.2.1).

The resulting programs strictly enforce the operation *ordering* computed for each resource in the reservation table, but allow for some start date elasticity at execution time to take advantage of execution/communication times shorter than the WCETs/WCCTs. This elasticity does not compromise the timing guarantees computed by the mapping tool. Indeed, if inputs are acquired periodically, with a period equal to the length of the reservation table (to model the periodic acquisition of an input), then the computed cycle times are fully respected.

Listings 4.1 and 4.2 provide the full application code synthesized by our tool for the example of Figures 4.7 and 4.9 on an architecture with 1 CPU per tile. Tile code is plain C code, whereas NoC router code is written in the assembly language defined in the previous chapter.

In the absence of conditional execution, the generation of assembly code for the NoC router controllers is straightforward, and we do not present it here. Instead, the generation of tile code involves complex issues related to the multiplexing of CPU and DMA commands in a single sequential thread.

5.3.1 Tile code generation

Each tile thread is an infinite loop that executes the (computation or communication) operations scheduled on the associated resource in the order prescribed by their reservations. The tile thread code is generated by the **GenerateTileThread** procedure of page 122. Recall that each tile may contain up to 16 CPUs, but is reserved as a single sequential resource, parallelism being hidden inside the data-flow blocks. The sequential thread of a tile runs on CPU 0 of the tile, but the code of each data-flow block can use all the processors.

Procedure 7 GenerateTileThread

Input: *ProcSchedule* : The scheduling table of processor (y,x)
DMASchedule : The scheduling table of the DMA segment of tile (y,x)
VariableAllocations : The set of variable allocations

```
 1: CurrDMAOp := 0 ;
 2: for i:=0 to length(ProcSchedule)-1 do
 3:    NodeEndDate := GetEndDate(ProcSchedule[i]);
 4:    while (CurrDMAOp <length(DMASchedule)) and (GetStartDate(DMASchedule[CurrDMAOp])<
       NodeEndDate) do
 5:       /* Make sure that writing is allowed. */
 6:       PrintWriteLockRequest(DMASchedule[CurrDMAOp],VariableAllocations);
 7:       /* DMASend also grants the read lock. */
 8:       PrintDMASend(DMASchedule[CurrDMAOp],VariableAllocations);
 9:       CurrDMAOp := CurrDMAOp+1 ;
10:    end while
11:    /* Make sure that input data has arrived. */
12:    PrintReadLockRequests(ProcSchedule[i],VariableAllocations) ;
13:    if IsFunctionNode(ProcSchedule[i]) then
14:       PrintFunctionCall(ProcSchedule[i],VariableAllocations);
15:    else
16:       PrintDelayCode(ProcSchedule[i],VariableAllocations);
17:    end if
18:    /* Node completion is the end of lifetime for some variables. */
19:    PrintWriteLockGrants(ProcSchedule[i],VariableAllocations);
20: end for
21: /* Remaining DMA commands. */
22: while CurrDMAOp <length(DMASchedule) do
23:    PrintWriteLockRequest(DMASchedule[CurrDMAOp],VariableAllocations);
24:    /* DMASend also grants the read lock. */
25:    PrintDMASend(DMASchedule[CurrDMAOp],VariableAllocations);
26:    CurrDMAOp := CurrDMAOp+1 ;
27: end while
```

No separate thread is generated for the DMA resource of a tile. Its operations are instead initiated by the tile thread. This is possible because the DMA allows the queuing of DMA commands. Code generation is done as follows: It is assumed that the scheduling table is sorted by reservation starting date. Each iteration of the top-level **for** loop of Procedure 7 generates code for one scheduled data-flow node. Code generation (lines 13-17 of the procedure) depends on the type of node (function or delay). The statements immediately before and after this code are synchronization code ensuring that:

- Needed input data has already been transmitted (line 12).
- Variables that have reached the end of their lifetime can be written again by DMA transfers (line 19).

The remaining code (lines 3-10) generated the DMA initiation code. The initiation of all DMA operations starting during the execution of a computation node is realized before the node starts execution. Lines 22-26 will generate the DMA initiation code for the DMA operations starting after all computations of the tile have completed. The example of Fig. 5.7 emphasizes the two cases where DMA code generation applies. First of all, DMA initiation for the sending of z and the first part of x is performed before the execution of k. The code initiating the sending of the second part of x is executed after k.

As explained in Section 5.2.2.3, DMA initiation code has very low duration. However, it is not accounted for during the scheduling phase, so the real-time guarantees provided by the scheduling table must be amended by taking this cost into account.

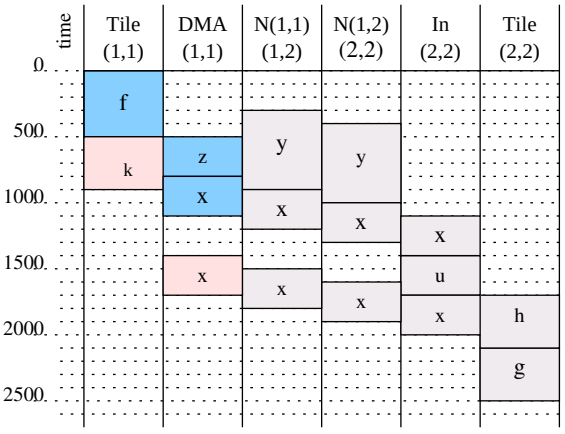


Figure 5.8: Scheduling table example for DMA code generation

5.4 Experimental results

We have evaluated our mapping and code generation method on two applications featuring no conditional execution but using classical signal processing filters: The platooning application described in Fig. 5.9, and a parallel Cooley-Tukey implementation of the integer 1D radix 2 FFT over 2^{14} samples [Bahn et al., 2008]. We chose these two applications because they allow the computation of tight lower bounds on the execution cycle makespan and because (for the FFT) an MPPA mapping already exists. This allows for meaningful comparisons, while no tool equivalent to ours exists to provide another basis for evaluation.

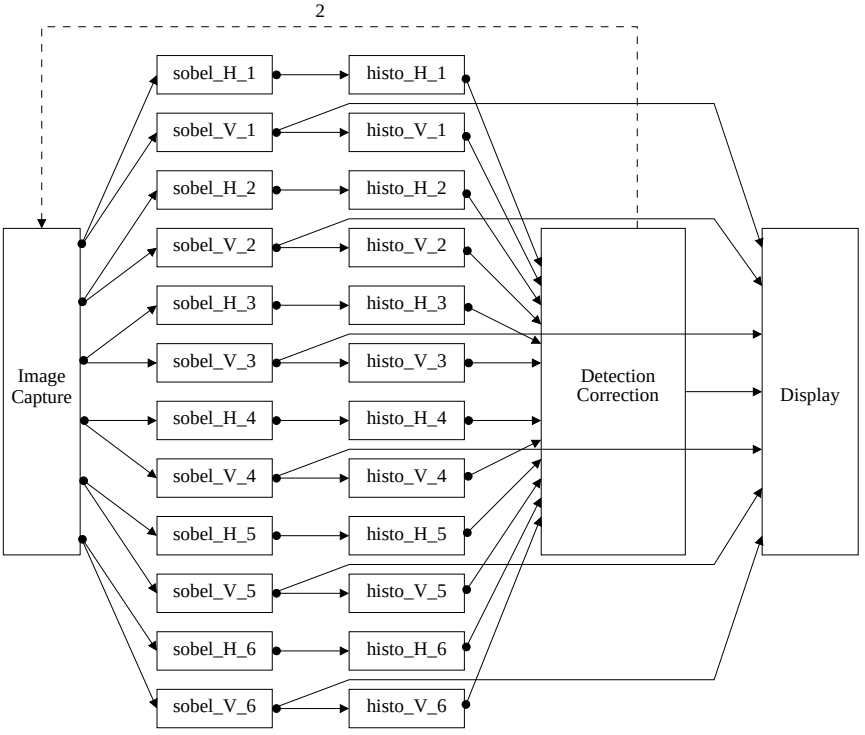


Figure 5.9: Dependent task system of a platooning application

The two applications were described using our data-flow formalism. This (manual) data-flow modeling phase chooses the degree of parallelism that can be exploited by our algorithms. In the platooning application, each block is a computation node, solid arcs are simple dependencies, and the dashed arc is a delayed dependency of depth 2. The application is run by a car to determine the position (distance and angle) of another car moving in front of it. It works by cyclically capturing an input image of fixed size. This image is

passed through an edge-detecting Sobel filter and then through a histogram search to detect dominant edges. This information is used by the detection and correction function to determine the position of the front car. The whole process is monitored on a display. The delayed dependency represents a feedback from the detection and correction function that allows the adjustment of image capture parameters. The Sobel filter and the histogram search are parallelized. Each of the *Sobel_H* and *Sobel_V* functions receives one sixth of the whole image (a horizontal slice).

For the FFT, we followed the parallelization scheme used in [Bahn et al., 2008], with a block size of 2^{11}, resulting in a total of 32 computation nodes. Evaluation is done on the 3x4 MPPA, where we assume that input data arrives on $Tile(0,0)$ and the results are output by $Tile(2,3)$.

For both applications, after computing the WCET of the tasks and the WCCT of the data transmissions, the mapping tool was applied to build a running implementation and to compute execution cycle makespan and throughput guarantees. Then, the code was run, and its performances measured. This allowed us to check the functional correctness of the code and to determine that our tool produces very *precise timing guarantees*. Indeed, the difference between predicted and observed makespan and throughput figures is less than 1% for both examples, which is due to the precision of our mapping algorithms and to the choice of a very predictable execution platform.

The generated off-line schedule (and the resulting code) has *good real-time properties*. For both the CyCab and the FFT, we have manually computed lower bounds on the execution cycle makespan.[1] The lower bounds computed for the CyCab and FFT examples were lower than the makespan values computed by our algorithms by respectively 8.9% and 3.4%. For the FFT example, we have also compared the measured makespan of our code with that of a classical NoC-based parallel implementation of the FFT [Bahn et al., 2008] running on our architecture. For our code, the NoC was statically scheduled, while for the classical implementation it was not. Execution results show that our code had a latency that was 3.82% shorter than the one of the classical parallel FFT code. In other words, **our tool produced code that not only has statically-computed hard real-time bounds**

[1] To compute these lower bounds we simplify the hardware model by assuming that the resources $N(i,j)(k,l)$ generate no contention (*i.e.* they allow the simultaneous transmission of all packets that demand it). We only take into account the sequencing of operations on processors and DMAs and the contentions on resources $In(i,j)$.

(which the hand-written code has not) but is also faster.

Our mapping heuristics favor the concentration of all computations and communications in a few tiles, leaving the others free to execute other applications (as opposed to evenly spreading the application tasks over the tiles). The code generated for Cycab has a tile load of 85%-99% for 6 of the 12 tiles of the architecture, while the other tiles are either unused or with very small loads (less than 7%). Using more computing tiles would bring no latency or throughput gains because our application is limited by the input acquisition speed. In the FFT application the synchronization barriers reduce average tile use to 47% on 8 of the 12 MPPA tiles. Note that the remaining free processor and NoC time can be used by other applications.

Finally, we have measured the influence of static scheduling of NoC communications on the application latency, by executing the code generated for Cycab and the FFT with and without NoC programming. For Cycab, not programming the NoC results in a speed loss of 7.41%. For the FFT the figure is 4.62%.

We conclude that our tool produces global static schedules of good quality, which provide timing guarantees close to the optimum.

Conclusion

The thesis we defend in this manuscript is that efficient parallel execution on a NoC-based MPPA requires better synchronization between computations and NoC data traffic, which could be obtained by compile-time static (off-line) real-time scheduling of both computations and communications. In turn, this means that global compiling processes should target together the processing elements and the programmable NoC routers.

Optimal NoC usage should result from a global optimization principle, as opposed to a collection of local optimization of individual connections. Indeed, various data flows with distinct sources and targets need to be highly concerted, both in time and space, like in a classical pipelined CPU, where the use of registers (replaced in our case with a complex NoC-based interconnect) is strongly synchronized with that of the functional units.

One main problem in applying such a global optimization approach is to provide the proper hardware infrastructures allowing the implementation of optimal computation and communication mappings and schedules. Our thesis is that optimal data transfer patterns should be encoded using simple programs configuring the router nodes (each router being then programmed to act its part in the globally concerted computation and communication scheme).

On the hardware design side, we concretely supported our proposed approach by extending a state-of-the-art NoC to allow programmed arbitration and offer the best support for off-line scheduling. In this NoC we have replaced the fair arbitration modules with static, micro-programmable modules. This allows us to establish effective static scheduling and routing of data transmissions as required by the application. Router programs are the result of a global compilation process which targets the NoC and the individual cores altogether. We have advocated the desired level of expressiveness for such configuration programs, and provide experimental data (coming from cycle-accurate simulations) supporting our choices. We also wrote an architecture synthesis tool that allows simple

architectural exploration of MPPAs using the new programmable NoC.

On the software side, we have proposed a novel allocation and scheduling method capable of synthesizing such global computation and communication schedules covering all the execution, communication, and memory resources in an MPPA. Our method allows static (table-based) scheduling of synchronous data-flow specifications. To allow an efficient use of the hardware resources, our method takes into account the specificities of the MPPA hardware and implements advanced scheduling techniques such as software pipelining and pre-computed preemption of data transmissions. Our tool synthesizes code for processing elements and NoC routers with static real-time guarantees that runs faster than (simple) hand-written parallel code.

Future work

The first point we wish to address in the future is the extension of both hardware and mapping technique so that they support data-dependent conditional control, and thus are able to consider a larger class of applications.

The main hardware-related question is that of finding a good balance between NoC router complexity and efficiency gain, seen in a broad sense. Solutions here range between a solution where the routers incorporate more and more features such as software-defined routes or multicast/broadcast, and the minimalist solution presented here, where these features must be implemented through software protocols.

On the software side, the mapping technique should be extended so that it directly takes into account the internal architecture of each tile (CPUs, memory banks, *etc.*), instead of seeing them as single resources. Our first objective here is to perform memory resource allocation at scheduling time (and not during code generation). More generally, our mapping technique could benefit from/to previous work on the scheduling of data-flow specifications and on compilation, but complex evaluation is needed to determine which algorithms scale up to take into account the low-level architectural detail.

Finally, it is important to explore the integration of on-line and off-line mapping techniques for efficient mapping of complex applications onto NoC-based MPPAs.

List of Publications

1. Djemal, M., Pêcheux, F., Potop-Butucaru, D., de Simone, R.,Wajsbürt, F., and Zhang, Z. **Programmable routers for efficient mapping of applications onto NoC-based MPSoCs**. In *Proceedings of the IEEE international conference on Design and Architectures for Signal and Image Processing (DASIP'12)*, Karlsruhe, Germany

2. Carle,T., Djemal,M., Potop Butucaru,D., de Simone,R.,Zhang,Z. **Static mapping of real-time applications onto massively parallel processor arrays**. In *Proceedings of the IEEE international conference on Application of Concurrency to System Design (ACSD'14)*, Tunis, Tunisia

3. Carle,T., Djemal,M., Potop Butucaru,D., de Simone,R.,Zhang,Z., Pechêux,F., and Wajsbürt, F. **Reconciling performance and predictability on a many-core through off-line mapping**. In *Reconciling Performance and Predictability Workshop (REPP'14)*, Grenoble, France

4. Carle,T., Djemal,M., Genius,D., Pechêux,F., Potop Butucaru,D., de Simone,R., Wajsbürt,F., and Zhang,Z. **Reconciling performance and predictability on a many-core through off-line mapping**. In *Reconciling Performance and Predictability Workshop (ReCoSoC'14)*, Montpellier, France

5. Djemal, M., Pêcheux, F., Potop-Butucaru, D., de Simone, R.,Wajsbürt, F., and Zhang, Z. **Programmable routers for efficient mapping of applications onto NoC-based MPSoCs**. *Poster In Colloque GDR SOC-SIP 2012*, Paris, France

Bibliography

[Adapteva, 2012] Adapteva (2012). The Epiphany many-core architecture. Online `http://www.adapteva.com/products/epiphany-ip/epiphany-architecture-ip/`. 11, 24, 29, 31, 43

[Alliance, 2001] Alliance, V. (2001). VCI: Virtual Component Interface Standard (OCB 2 2.0). Online at: http://www.vsi.org. 59

[Almaless and Wajsbürt, 2012] Almaless, G. and Wajsbürt, F. (2012). On the scalability of image and signal processing parallel applications on emerging cc-NUMA many-cores. In *Proceedings DASIP'12*, Karlsruhe, Germany. 47

[Amarasinghe et al., 2005] Amarasinghe, S., Gordon, M. I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R., and Thies, W. (2005). Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2). 49

[AOSTE-INRIA,] AOSTE-INRIA. SynDEx:System-Level CAD Software for Distributed Real-Time Embedded Systems. Online at: http://www.syndex.org/. 49

[ARINC653, 2005] ARINC653 (2005). ARINC 653: Avionics application software standard interface. `www.arinc.org`. 72

[ARM, 1999] ARM (1999). Amba specification rev 2.0. ARM Limited. 12

[Aubry et al., 2013] Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., de Dinechin, B. D., Galea, F., Goubier, T., Harrand, M., Jones, S., Lesage, J.-D., Louise, S., Chaisemartin, N. M., Nguyen, T. H., Raynaud, X., and Sirdey, R. (2013). Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In *Proceedings of the First International Workshop on Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems (ALCHEMY 2013)*, Barcelona, Spain. 15, 31, 32, 54

[AUTOSAR, 2009] AUTOSAR (2009). Autosar (automotive open system architecture), release 4. `http://www.autosar.org/`. 72

[Aydi et al., 2011] Aydi, Y., Baklouti, M., Abid, M., and Dekeyser, J.-L. (2011). A multi-level design methodology of multistage interconnection network for mpsocs. *IJCAT*, 42(2/3):191–203. 65

[Bacivarov et al., 2013] Bacivarov, I., Haid, W., Huang, K., and Thiele, L. (2013). Methods and tools for mapping process networks onto multi-processor systems-on-chip. In *Handbook of Signal Processing Systems*. Springer. 55

[Bahn et al., 2008] Bahn, J. H., Yang, J., and Bagherzadeh, N. (2008). Parallel fft algorithms on network-on-chips. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*. 85, 124, 125

[Bebelis et al., 2013] Bebelis, V., Fradet, P., Girault, A., and Lavigueur, B. (2013). A framework to schedule parametric dataflow applications on many-core platforms. In *Proceedings CPC'13*, Lyon, France. 55

[Beletska et al., 2011] Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., and Siedlecki, K. (2011). Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. *Parallel Computing*, 37(8):479 – 497. 10

[Benini, 2010] Benini, L. (2010). Programming heterogeneous many-core platforms in nanometer technology: the p2012 experience. Presentation in the ARTIST Summer School, Autrans, France. Online at: http://www.artist-embedded.org/artist/Videos.html. 12, 29, 65, 77

[Benini and De Micheli, 2002] Benini, L. and De Micheli, G. (2002). Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78. 13

[Benveniste et al., 2003] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83. 95

[Bhattacharyya et al., 2013] Bhattacharyya, S., Deprettere, E., Leupers, R., and Takala, J., editors (2013). *Handbook of Signal Processing Systems*. Springer. 2nd edition, in particular chapter. 55

[Brandner and Schoeberl, 2012] Brandner, F. and Schoeberl, M. (2012). Static routing in symmetric real-time network-on-chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12. 47

[Buchmann et al., 2004] Buchmann, R., Pétrot, F., and Greiner, A. (2004). Fast cycle accurate simulator to simulate event-driven behavior. In *Proceedings of the International Conference on Electrical, Electronic and Computer Engineering*, pages 35–38. 68

[Campbell et al., 2006] Campbell, S., Chancelier, J.-P., and Nikoukhah, R. (2006). *Modeling and Simulation in Scilab/Scicos*. Springer. 50, 95

[Carara et al., 2007] Carara, E., Calazans, N., and Moraes, F. (2007). Router architecture for high-performance nocs. In *Proceedings SBCCI*, Rio de Janeio, Brazil. 31

[Carle and Potop-Butucaru, 2011] Carle, T. and Potop-Butucaru, D. (2011). Throughput Optimization by Software Pipelining of Conditional Reservation tables. Rapport de recherche RR-7606, INRIA. 51, 52, 103, 106

[Carle et al., 2012] Carle, T., Potop-Butucaru, D., Sorel, Y., and Lesens, D. (2012). From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. Research report RR-8109, INRIA. 102

[Caspi et al., 2003] Caspi, P., Curic, A., Magnan, A., Sofronis, C., Tripakis, S., and Niebert, P. (2003). From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings LCTES*, San Diego, CA, USA. 17, 49

[Cetus, 2004] Cetus (2004). Cetus: A source-to-source compiler infrastructure for c programs. Online `http://cetus.ecn.purdue.edu/`. 10

[de Dinechin et al., 2013] de Dinechin, B. D., de Massas, P. G., Lager, G., Léger, C., Orgogozo, B., Reybert, J., and Strudel, T. (2013). A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18(0):1654 – 1663. International Conference on Computational Science. 44

[Djemal et al., 2012] Djemal, M., Pêcheux, F., Potop-Butucaru, D., de Simone, R., Wajsbürt, F., and Zhang, Z. (2012). Programmable routers for efficient mapping of applications onto NoC-based MPSoCs. In *Proceedings of the IEEE international conference on Design and Architectures for Signal and Image Processing (DASIP)*. 32

[Eles et al., 2000] Eles, P., Doboli, A., Pop, P., and Peng, Z. (2000). Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on VLSI Systems*, 8(5):472–491. 49

[ELF, 1995] ELF (1995). Tool interface standard (tis) committee: Executable and linking format (elf) specification. version 1.2. Online `http://refspecs.linuxbase.org/elf/elf.pdf`. 63

[Epiphany, 2012] Epiphany (2012). Epiphany architecture reference manual. Online `http://www.adapteva.com/uncategorized/e3-reference-manual/`. 44

[Feige and Raghavan, 1992] Feige, U. and Raghavan, P. (1992). Exact analysis of hot-potato routing. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, SFCS '92, pages 553–562, Washington, DC, USA. IEEE Computer Society. 35

[Fisher, 1983] Fisher, J. (1983). Very long instruction word architectures and the eli-512. In *Proceedings ISCA*. 41, 49

[Fohler and Ramamritham, 1995] Fohler, G. and Ramamritham, K. (1995). Static scheduling of pipelined periodic tasks in distributed real-time systems. In *In Procs. of EUROMICRO-RTS97*, pages 128–135. 48, 49

[Furber, 2006] Furber, S. (2006). Living with failure : Lessons from nature ? In *Proceedings of the 11th IEEE European Test Symposium (ETS)*, pages 4–8. 10

[Genius et al., 2013] Genius, D., Kordon, A. M., and el Abidine, K. Z. (2013). Space optimal solution for data reordering in streaming applications on noc based mpsoc. *Journal of System Architecture*. 55

[Gerdes et al., 2012] Gerdes, M., Kluge, F., Ungerer, T., Rochange, C., and Sainrat, P. (2012). Time analysable synchronisation techniques for parallelised hard real-time applications. In *Proceedings DATE'12*, Dresden, Germany. 15, 31

[Goossens et al., 2003] Goossens, J., Funk, S., and Baruah, S. (2003). Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205. 15

[Goossens et al., 2012] Goossens, K., Azevedo, A., Chandrasekar, K., Gomony, M., Goossens, S., Koedam, M., Li, Y., Mirzoyan, D., Molnos, A., Nejad, A. B., Nelson, A., and Sinha, S. (2012). Virtual execution platforms for mixed-time-criticality applications : the CompSoC architecture and design flow. In *Proceedings of the 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pages 23–30, San Juan, Puerto Rico. 47, 52, 55

[Goossens et al., 2005] Goossens, K., Dielissen, J., and Radulescu, A. (2005). Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5). 14, 30, 31, 33, 34

[Gordon, 2010] Gordon, M. (2010). *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology. 52, 53

[Gordon et al., 2006] Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162. 53

[Gordon E. Moore, 1965] Gordon E. Moore (1965). Moore's law. Online `http://en.wikipedia.org/wiki/Moore's_law`. 8

[Gordon E. Moore, 2005] Gordon E. Moore (2005). Excerpts from a conversation with gordon moore: Moore's law. Online `http://download.intel.com/museum/Moores_law/Video-transcripts/excepts_a_Conversation_with_gordon_Moore.pdf`. 8

[Goubier et al., 2011] Goubier, T., Sirdey, R., Louise, S., and David, V. (2011). σc: A programming model and language for embedded manycores. In *Proceedings ICA3PP'11 (LNCS 7016)*, Melbourne, Australia. 54

[Grandpierre and Sorel, 2003] Grandpierre, T. and Sorel, Y. (2003). From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings MEMOCODE*, Mont Saint-Michel, France. 17, 48, 49, 51, 94, 112, 115

[Guernic et al., 2003] Guernic, P. L., Talpin, J.-P., and Lann, J.-C. L. (2003). Polychrony for system design. *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design*. 48, 50, 95

[Guerrier and Greiner, 2000] Guerrier, P. and Greiner, A. (2000). A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 250–256, New York, NY, USA. ACM. 29

[Halbwachs, 1993] Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer academic Publishers. 95

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320. 48, 50, 95

[Hanumaiah and Vrudhula, 2012] Hanumaiah, V. and Vrudhula, S. (2012). Temperature-aware dvfs for hard real-time applications on multicore processors. *IEEE Trans. Comput.*, 61(10):1484–1494. 10

[Hardy and Puaut, 2008] Hardy, D. and Puaut, I. (2008). Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*. 13, 58, 66

[Harrand and Durand, 2011] Harrand, M. and Durand, Y. (2011). Network on chip with quality of service. United States patent application publication US 2011/026400A1. 14, 30, 31, 36, 37, 65, 77

[Held et al., 2006] Held, J., Bautista, J., and Koehl, S. (2006). From a Few Cores to Many: A Tera-scale Computing Research Overview. Technical report. 9

[Henzinger and Kirsch, 2007] Henzinger, T. and Kirsch, C. (2007). The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems*, 29(6). 72

[Heptagon, 2013] Heptagon (2013). Heptagon/bzr manual. Online `http:http://bzr.inria.fr/pub/bzr-manual.pdf`. 48

[Hilton and Nelson, 2006] Hilton, C. and Nelson, B. (2006). Pnoc: a flexible circuit-switched noc for fpga-based systems. *Computers and Digital Techniques, IEE Proceedings* -, 153(3):181 – 188. 14, 26

[Howard and al., 2011] Howard, J. and al. (2011). A 48-core ia-32 processor in 45nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1). 11, 12, 14, 28, 31, 45

[IBM, 2001] IBM (2001). The POWER4 processor introduction and tuning guide. Online `http://www.redbooks.ibm.com/redbooks/pdfs/sg247041.pdf`. 9

[Irigoin et al., 2012] Irigoin, F., Amini, M., Ancourt, C., Coelho, F., Creusillet, B., and Keryell, R. (2012). Polyèdres et compilation. *Technique et Science Informatiques*, 31(8-10):987–1019. 10

[J.Flynn, 2004] J.Flynn, L. (2004). Intel halts development of 2 new microprocessors. *New York Times*. 9

[John von Neumann, 1945] John von Neumann (1945). Von neumann architecture. Online `http://en.wikipedia.org/wiki/Von_Neumann_architecture`. 8

[Johnson and Frigo, 2008] Johnson, S. G. and Frigo, M. (2008). Implementing FFTs in practice. In Burrus, C. S., editor, *Fast Fourier Transforms*, chapter 11. Connexions, Rice University, Houston TX. 84

[Kakoee, 2012] Kakoee, M. R. (2012). *Reliable and Variation-tolerant Interconnection Network for Low Power MPSoCs*. PhD thesis, Universitá di Bologna. Online at `http://amsdottorato.unibo.it/4407/1/phdthesis.pdf`. 65

[Kashif et al., 2013] Kashif, H., Gholamian, S., Pellizzoni, R., Patel, H., and Fischmeister, S. (2013). Ortap: An offset-based response time analysis for a pipelined communication resource model. In *Proceedings RTAS*. 15

[Khronos, 2011] Khronos (2011). The open standard for parallel programming of heterogeneous systems. Online `https://www.khronos.org/opencl`. 10, 15

[Kumar et al., 2002] Kumar, S., antsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., and Hemani, A. (2002). A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '02, pages 117–124, Washington, DC, USA. IEEE Computer Society. 27

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471. 15

[Lee and Messerschmitt, 1987] Lee, E. A. and Messerschmitt, D. G. (1987). Synchronous data flow. In *Proceedings of the IEEE*, pages 1235–1245. 48

[Lee et al., 1998] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S. (1998). Space-time scheduling of instruction-level parallelism on a raw machine. *SIGOPS Oper. Syst. Rev.*, 32(5):46–57. 49

[Lickly et al., 2008] Lickly, B., Liu, I., Kim, S., Patel, H., Edwards, S., and Lee, E. (2008). Predictable programming on a precision timed architecture. In *Proceedings CASES'08*. 72

[LIP6, 2011] LIP6 (2011). SoClib: an open platform for virtual prototyping of multiprocessors system on chip. Online at: http://www.soclib.fr. 19, 33, 57

[Lu and Jantsch, 2007] Lu, Z. and Jantsch, A. (2007). Tdm virtual-circuit configuration for network-on-chip. *IEEE Trans. VLSI*. 32, 55

[Melpignano et al., 2012] Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., and Dutoit, D. (2012). Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA. ACM. 46

[MIC, 2010] MIC (2010). Intel Many Integrated Core Architecture. Online at: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. 11, 12

[Milder et al., 2007] Milder, P., Franchetti, F., Hoe, J., and Püschel, M. (2007). FFT compiler: From math to efficient hardware. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 84

[Millberg et al., 2004a] Millberg, M., Nilsson, E., Thid, R., and Jantsch, A. (2004a). Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, Washington, DC, USA. IEEE Computer Society. 35, 36

[Millberg et al., 2004b] Millberg, M., Nilsson, E., Thid, R., Kumar, S., and Jantsch, A. (2004b). The nostrum backbone - a communication protocol stack for networks on

chip. In *Proceedings of the 17th International Conference on VLSI Design*, VLSID '04, Washington, DC, USA. IEEE Computer Society. 14, 30, 35

[Miro Panades et al., 2006] Miro Panades, I., Greiner, A., and Sheibanyrad, A. (2006). A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach. In *NanoNet International Conference on Nano-Networks*, pages 1–5. 14, 33

[Moscibroda and Mutlu, 2009] Moscibroda, T. and Mutlu, O. (2009). A case for bufferless routing in on-chip networks. In *Proceedings ISCA-36*. 31

[MPPA, 2012] MPPA (2012). The MPPA256 many-core architecture. `www.kalray.eu`. 11, 29, 30, 36, 37, 42

[Nikolic et al., 2013] Nikolic, B., Ali, H., Petters, S., and Pinho, L. (2013). Are virtual channels the bottleneck of priority-aware wormhole-switched noc-based many-cores? In *Proceedings RTNS*, 2013. 15

[Nvidia, 1999] Nvidia (1999). Geforce 256: The world's first gpu. Online `http://www.nvidia.co.uk/page/geforce256.html`. 11

[Nvidia CUDA, 2006] Nvidia CUDA (2006). Cuda. Online `http://www.nvidia.com/object/cuda_home_new`. 15

[OpenMP, 2008] OpenMP (2008). The openmp api specification for parallel programming. Online `www.openmp.org`. 10, 15

[Panades, 2008] Panades, I. (2008). *Conception et implantation d'un micro-réseau sur puce avec garante de service*. PhD thesis, Université Pierre et Marie Curie. 17, 24, 31, 33, 57

[Pande et al., 2005] Pande, P. P., Grecu, C., Jones, M., Ivanov, A., and Saleh, R. (2005). Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040. 24

[Parks et al., 1995] Parks, T. M., Pino, J. L., and Lee, E. A. (1995). A comparison of synchronous and cycle-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, ASILOMAR '95, pages 204–, Washington, DC, USA. IEEE Computer Society. 48

[Peng et al., 2007] Peng, L., Peir, J.-K., Prakash, T. K., Chen, Y.-K., and Koppelman, D. M. (2007). Memory performance and scalability of intel's and amd's dual-core processors: A case study. In *IPCCC'07*, pages 55–64. 8

[PLB, 2001] PLB (2001). 32-bit processor local bus architecture specification version 2.9. IBM Corporation. 12

[Pollack, 1999] Pollack, F. J. (1999). New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, Washington, DC, USA. IEEE Computer Society. 9

[Potop-Butucaru et al., 2009] Potop-Butucaru, D., de Simone, R., Sorel, Y., and Talpin, J.-P. (2009). Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings EMSOFT*, Grenoble, France. 49, 51, 94, 95, 100, 101, 106, 112, 115

[Potop-Butucaru et al., 2005] Potop-Butucaru, D., Simone, R. D., and pierre Talpin, J. (2005). The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*. 95

[Pradalier et al., 2005] Pradalier, C., Hermosillo, J., Koike, C., Braillon, C., Bessière, P., and Laugier, C. (2005). The CyCab: a car-like robot navigating autonomously and safely among pedestrians. *Robotics and Autonomous Systems*, 50(1). 78

[Puaut and Potop-Butucaru, 2013] Puaut, I. and Potop-Butucaru, D. (2013). Integrated worst-case execution time estimation of multicore applications. In *Proceedings WCET'13*, Paris, France. to appear. 20, 65, 114

[Qualcomm, 2011] Qualcomm (2011). Snapdragon S4 Processors: System on a Chip Solution for a New Mobile Age. Technical report. 11

[R. Wilhelm et al., 2008] R. Wilhelm et al. (2008). The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7(3). 66

[Ramamritham et al., 1993] Ramamritham, K., Fohler, G., and Adan, J. M. (1993). Issues in the static allocation and scheduling of complex periodic tasks. In *In Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*. 15

[Sadawarte et al., 2011] Sadawarte, Y. A., A.Gaikwad, M., and M.Patrikar, R. (2011). Implementation of virtual cut-through algorithm for network on chip architecture. *IJCA Proceedings on International Symposium on Devices MEMS, Intelligent Systems and Communication (ISDMISC)*, (1):5–8. Published by Foundation of Computer Science, New York, USA. 28

[Salloum et al., 2013] Salloum, C. E., Elshuber, M., Höftberger, O., Isakovic, H., and Wasicek, A. (2013). The {ACROSS} {MPSoC} – a new generation of multi-core processors designed for safety–critical embedded systems. *Microprocessors and Microsystems*, 37(8, Part C):1020 – 1032. Special Issue on European Projects in Embedded System Design: {EPESD2012}. 39, 47

[Sangiovanni-vincentelli and Martin, 2001] Sangiovanni-vincentelli, A. and Martin, G. (2001). Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33. 51

[Sgroi et al., 2001] Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J., and Sangiovanni-Vencentelli, A. (2001). Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 667–672, New York, NY, USA. ACM. 13, 23

[Shi and Burns, 2010] Shi, Z. and Burns, A. (2010). Schedulability analysis and task mapping for real-time on-chip communication. *Real-Time Systems*, 46(3):360–385. 15, 31, 55

[Sorel, 1994] Sorel, Y. (1994). Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy. 49

[Sorensen et al., 2012] Sorensen, R., Schoeberl, M., and Sparso, J. (2012). A light-weight statically scheduled network-on-chip. In *Proceedings NORCHIP*. 30

[Sørensen et al., 2012] Sørensen, R. B., Schoeberl, M., and Sparsø, J. (2012). A light-weight statically scheduled network-on-chip. In *Proceedings of the 29th Norchip Conference*, Copenhagen. 40

[Tamir and Chi, 1993] Tamir, Y. and Chi, H.-C. (1993). Symmetric crossbar arbiters for vlsi communication switches. *Parallel and Distributed Systems, IEEE Transactions on*, 4(1):13–27. 46

[Taylor, 2003] Taylor, M. B. (2003). *The Raw Processor Specification*. MIT. Available online `http://groups.csail.mit.edu/cag/raw/documents/RawSpec99.pdf`. 38

[Taylor et al., 2004] Taylor, M. B., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., and Kim, J. (2004). Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. *SIGARCH Comput. Archit. News*, 32(2). 38, 40

[Texas Instruments, 2009] Texas Instruments (2009). Omap 4: Mobile applications platform. Online `http://focus.ti.com/lit/ml/swpt034/swpt034.pdf`. 11

[Thonnart et al., 2010] Thonnart, Y., Vivet, P., and Clermidy, F. (2010). A fully-asynchronous low-power framework for gals noc integration. In *Proceedings DATE'10*, pages 33–38, Dresden, Germany. 46

[Tilera, 2008] Tilera (2008). The TilePro64 many-core architecture. Online `http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf`. 11, 24, 29, 31, 32, 38, 40

[Tilera Corporation, 2013] Tilera Corporation (2013). Tile processor architecture-overview for the tilepro series. Online `http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf`. 41, 42

[TSAR, 2008] TSAR (2008). Tera-scale architecture. Online `https://www-asim.lip6.fr/trac/tsar/wiki`. 13, 46

[Villalpando et al., 2010] Villalpando, C., Johnson, A., Some, R., Oberlin, J., and Goldberg, S. (2010). Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander. In *Proceedings of the IEEE Aerospace Conference*. 15, 31

[Waingold et al., 1997] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., and Agarwal,

A. (1997). Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86 –93. 14, 32, 38, 39, 52

[Wilhelm et al., 2009] Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. (2009). Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978. 43

[Wilhelm and Reineke, 2012] Wilhelm, R. and Reineke, J. (2012). Embedded systems: Many cores – many problems (invited paper). In *Proceedings SIES'12*, Karlsruhe, Germany. 13, 15, 58

[Xu, 1993] Xu, J. (1993). Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *Software Engineering, IEEE Transactions on*, 19(2):139–154. 49

[Yoon et al., 2010] Yoon, Y., Concer, N., Petracca, M., and Carloni, L. (2010). Virtual channels vs. multiple physical networks: a comparative analysis. In *Proceedings DAC*, Anaheim, CA, USA. 31

[Yu et al., 2008] Yu, Z., Meeuwsen, M. J., Sattari, O., Lai, M., Webb, J. W., Work, E. W., Truong, D., Mohsenin, T., and Baas, B. M. (2008). Asap: An asynchronous array of simple processors. IEEE Journal of Solid-State Circuits. 39

[Zhai et al., 2013] Zhai, J. T., Bamakhrama, M., and Stefanov, T. (2013). Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *Proceedings DAC*. 55

[Zhang, 2011] Zhang, Z. (2011). *On the field Detection, De-activation and Reconfiguration (ODDR) mechanism for Permanent Fault-Tolerance of Network-on-Chip*. PhD thesis, Université Pierre et Marie Curie. 11

[Zimmermann, 1988] Zimmermann, H. (1988). Innovations in internetworking. chapter OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, pages 2–9. Artech House, Inc., Norwood, MA, USA. 23