Design Methods and Tools for Application-Specific Predictable Networks-on-Chip

THÈSE N^{O} 5407 (2012)

PRÉSENTÉE LE 2 JUILLET 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DES SYSTÈMES INTÉGRÉS (IC/STI)

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ciprian SEICULESCU

acceptée sur proposition du jury:

Prof. B. Falsafi, président du jury
Prof. G. De Micheli, Prof. L. Benini, directeurs de thèse
Dr F. Angiolini, rapporteur
Prof. D. Atienza Alonso, rapporteur
Prof. A. Greiner, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Abstract

As the complexity of applications grows with each new generation, so does the demand for computation power. To satisfy the computation demands at manageable power levels, we see a shift in the design paradigm from single processor systems to *Multiprocessor Systems-on-Chip* (MPSoCs). MPSoCs leverage the parallelism in applications to increase the performance at the same power levels. To further improve the computation to power consumption ratio, MPSoCs for embedded applications are heterogeneous and integrate cores that are specialized to perform the different functionalities of the application. With technology scaling, wire power consumption is increasing compared to logic, making communication as expensive as computation. Therefore customizing the interconnect is necessary to achieve energy efficiency. Designing an optimal application specific *Network-on-Chip* (NoC), that meets application demands, requires the exploration of a large design space. Automatic design and optimization of the NoC is required in order to achieve fast design closure, especially for heterogeneous MPSoCs.

To continue to meet the computation requirements of future applications new technologies are emerging. *Three dimensional integration* promises to increase the number of transistors by stacking multiple silicon layers. This will lead to an increase in the number of cores of the MPSoCs resulting in increased communication demands. To compensate for the increase in the wire delay in new technology nodes as well as to reduce the power consumption further, multi-synchronous design is becoming popular. With multiple clock signals, different parts of the MPSoC can be clocked at different frequencies according to the current demands of the application and can even be shutdown when they are not used at all. This further complicates the design of the NoC. Many applications require different levels of guarantee from the NoC in order to perform their functionality correctly. As communication traffic patterns become more complex, the performance of the NoC can no longer be predicted statically. Therefore designing the interconnect network requires that such guarantees are provided during the dynamic operation of the system which includes the interaction with major subsystems (i.e., main memory) and not just the interconnect itself.

In this thesis, I present novel methods to design application-specific NoCs that meet perfor-

mance demands, under the constraints of new technologies. To provide different levels of *Quality of Service*, I integrate methods to estimate the NoC performance during the design phase of the interconnect topology. I present methods and architectures for NoCs to efficiently access memory systems, in order to achieve predictable operation of the systems from the point of view of the communication as well as the bottleneck target devices. Therefore the main contribution of the thesis is twofold: *scientific* as I propose new algorithms to perform topology synthesis and *engineering* by presenting extensive experiments and architectures for NoC design.

Keywords: MPSoC, NoC, topology, synthesis, real-time, simulation, DRAM memory

Résumé

Comme la complexité des applications augmente avec chaque nouvelle génération, il en va de même pour les besoins en puissance de calcul. Pour satisfaire la demande en calculs tout en restant à des niveaux de puissance consommée acceptables, nous observons un changement dans le paradigme de conception depuis les systèmes monoprocesseurs aux *systèmes sur puces multiprocesseurs* (MPSoCs pour Multiprocessor Systems-on-Chip). Les MPSoCs reposent sur la parallélisation des applications pour augmenter les performances à un niveau de puissance consommée ciblé. Pour continuer à améliorer le rapport entre la puissance de calcul et la consommation d'énergie, les MPSoCs dédiés aux applications embarquées sont hétérogènes et intègrent des cœurs qui se sont spécifiques à certaines fonctionnalités de l'application. Avec l'évolution technologique et la réduction des dimensions, la consommation énergétique due aux interconnexions est en augmentation par rapport à la logique, ce qui rend la communication plus chére que le calcul. Par consèquent, l'utilisation de nouvelles structures d'interconnexions est nécessaire pour atteindre l'efficacité énergétique. La conception d'un *Réseau sur Puce* (NoC pour Network-on-Chip) optimal vis-à-vis des contraintes applicatives exige l'exploration d'un large espace de conception. La conception et l'optimisation automatique du NoC est nécessaire afin d'aboutir à un temps de conception tolérable, en particulier pour les MPSoCs hétérogènes.
Pour continuer à répondre aux exigences de calcul des futures applications, de nouvelles technologies font leur apparition. *L'intégration tri-dimensionelle* promet une augmentation du nombre de transistors grâce à l'empilement de multiples couches de silicium. Cela conduira à une augmentation du nombre de cœurs dans les MPSoCs et à des demandes accrues vis-à-vis des communications. Afin de compenser l'augmentation du délai du aux interconnexions dans les nouveaux nœuds technologiques mais aussi de continuer à réduire la consommation énergétique, la conception de systèmes multi-synchrones est devenu populaire. Avec plusieurs signaux d'horloge, les différentes parties du MPSoC peuvent être cadencées à des fréquences différentes selon le besoin immédiat de l'application et peuvent même être arrêtées lorsqu'elles ne sont

pas utilisées. Ceci complique encore la conception du NoC. Les applications exigent différents objectifs de performances du NoC afin d'assurer un fonctionnement correct. Or le trafic dans le réseau devenant de plus en plus complexes, la performance du NoC ne peut plus être prédite de façon statique. En conséquence, la conception du réseau d'interconnexion exige que ces objectifs soient tenus pendant le fonctionnement dynamique du système qui inclut l'interaction avec les principaux sous-systèmes (i.e, la mémoire principale) et non simplement l'interconnexion en elle-même.
Dans cette thèse, je présente de nouvelles méthodes pour concevoir des NoCs spécifiques à une application, qui répondent à des exigences de performance et respectent les contraintes des nouvelles technologies. Pour fournir différents niveaux de *qualité de service* (QoS pour Quality-of-Service), j'intègre des méthodes pour estimer la performance du NoC durant la phase de conception de la topologie d'interconnexion. Je présente des méthodes et des architectures pour les NoCs permettant d'accéder efficacement à la mémoire, dans le but d'obtenir un fonctionnement prévisible des systèmes du point de vue de la communication tout en considérant l'impact des périphériques les plus contraignants. Par conséquent, la thèse apporte une contribution originale sur deux fronts : *scientifique* en proposant de nouveaux algorithmes pour réaliser une synthèse topologique et *l'ingénierie* en présentant des expériences approfondies et des architectures pour la conception de NoCs.

Mots-clés : MPSoC, NoC, topologies, synthèse, temps réel, simulation, mémoire DRAM

Riassunto

Con l'aumentare della complessità delle applicazioni ad ogni generazione, si assiste ad un aumento corrispondente dei requisiti di potenza di calcolo. Per soddisfare i bisogni computazionali mantenendo un livello di consumo di potenza ragionevole, assistiamo ad una transizione dai sistemi a singolo processore verso sistemi multiprocessore (*Multiprocessor Systems-on-Chip* o MPSoCs). Gli MPSoC sfruttano il parallelismo disponibile nelle applicazioni per aumentare le prestazioni a consumo di potenza costante. Per migliorare ulteriormente il rapporto tra potenza computazionale e requisiti energetici, gli MPSoC per applicazioni embedded sono eterogenei ed integrano processori specializzati per le varie funzioni delle applicazioni. Con lo scalare della tecnologia, il consumo di potenza per l'attraversamento dei fili è in aumento rispetto a quello della logica, rendendo la comunicazione costosa quanto la computazione. Di conseguenza, l'ottimizzazione dell'interconnessione è necessaria per raggiungere la massima efficienza energetica. Tuttavia, progettare una *Network-on-Chip* (NoC) specifica per un'applicazione, che soddisfi tutti i requisiti applicativi, richiede l'esplorazione di un vasto spazio di progetto. L'automazione della progettazione e ottimizzazione delle NoC è necessaria per ottenere una rapida convergenza, soprattutto per MPSoC eterogenei.

Nuove tecnologie stanno emergendo per continuare a soddisfare i requisiti di potenza di calcolo delle applicazioni future. L'*integrazione tridimensionale* promette un aumento del numero di transistori con la sovrapposizione di più strati di silicio. Questo porterà ad un aumento del numero di processori negli MPSoC, risultando anche in un aumento dei requisiti di comunicazione. Per compensare il deterioramento dei ritardi di propagazione sui fili nei nuovi nodi tecnologici e per ridurre ulteriormente i consumi di potenza, la progettazione multi-sincrona sta diventando sempre più comune. Usando più segnali di *clock*, ogni parte di un MPSoC può essere fatta lavorare a frequenze diverse secondo gli attuali requisiti applicativi, oppure addirittura spenta completamente quando non in uso. Questo però rende più complessa la progettazione della NoC. Inoltre, molte applicazioni si attendono livelli diversi di prestazioni

garantiti dalla NoC per svolgere correttamente i propri compiti. Tuttavia, al complicarsi degli schemi di traffico, queste prestazioni non possono più essere garantite staticamente. Quindi, progettare l'interconnessione richiede che queste garanzie siano fornite a *runtime* a sistema funzionante, quando vanno tenute in conto anche le interazioni con sottosistemi chiave (per esempio, la memoria esterna) e non più solo l'interconnessione in sè.
In questa tesi, presenterò dei metodi innovativi per progettare NoC ottimizzate per specifiche applicazioni, che rispettino i bisogni prestazionali e i nuovi vincoli tecnologici. Per poter fornire diversi livelli di servizio (*Quality of Service*), mostrerò l'integrazione di metodi per stimare le prestazioni della NoC già in fase di progettazione della topologia dell'interconnessione. Presenterò poi metodi ed architetture volte ad un efficiente accesso della NoC ai sottosistemi di memoria, per poter predire il funzionamento del sistema considerando sia l'interconnessione che i colli di bottiglia imposti dai dispositivi destinatari del traffico. In sintesi, il contributo principale della tesi comprende due assi principali: *scientifico*, riguardo agli algoritmi proposti per la sintesi delle topologie, e *ingegneristico*, con riferimento ad una vasta sezione sperimentale ed alle proposte architetturali per la progettazione di NoC.

Parole chiave: MPSoC, NoC, topologia, sintesi, tempo-reale, simulazione, memoria DRAM

Acknowledgements

First of all I would like to thank Prof. Giovanni De Micheli for giving me the opportunity to do research under his supervision. Even though he gave me ample freedom to pursue my own ideas on the topic as well, his guidance was crucial for the completion of this work and for making sure that I would constantly focus on the overall goal. His advice was useful not only from the technical viewpoint, but also to help me refine other primary skills, such as technical writing and presentation. I am grateful also for the very friendly atmosphere that he creates in his laboratory, e.g. by way of the social events that he organizes.

I also need to thank Prof. Luca Benini for his continuous supervision and advice. He made sure that I would make constant progress, helping me reach the research milestone represented by this thesis. His careful review of my work was instrumental in improving the quality of my investigation and publications. His contacts with the industry and his broad scope of interests also helped make this work relevant from a practical perspective.

Then, I would like to thank Srinivasan Murali, the post-doctoral assistant supervising me, following my work and helping me from day one. I have learned a lot from him. Not only he inspired me to turn my ideas into publications from the very beginning of my PhD, he is also a very good friend. We have also spent quite a bit of time together outside of the laboratory and Srini is the one who got me started on several sportive activities in Lausanne.

I would like to thank my examination committee members, Prof. Babak Falsafi, Prof. Alain Greiner, Prof. David Atienza and Dr. Federico Angiolini for their comments and insightful advice on this thesis.

I would also like to thank my research collaborators, Dara Rahmati, Naser Khosro Pour and Stavros Volos for their contributions to some key publications, which are at the core of portions of this thesis.

I thank all my colleagues and friends at EPFL: Antonio, Pierre-Emmanuel, Jaime, Vasilis, Alena, Camilla, Shashi, Cristina, Andrea, Michele, Julien, Jacopo, Irene, Hu,

Wenqi, Eleni, Francesca and all the others that I may have missed. They have created a very nice and friendly atmosphere in the lab and we have also spent many nice moments outside of it on hikes and dinners. I thank Jacopo and Shashi for their continuous efforts to provide entertainment during lunches. I thank Antonio for organizing a few extreme hikes, which I am proud to have completed, and Cristina for other nice ones - some of which we still need to complete. My gratitude goes to our administrative secretary Christina, whose efforts have spared us from the boring tasks of filling paperwork. A special mention goes to Abhishek, as TA of the course that inspired me to choose this laboratory for the PhD, and therefore also as one of my first colleagues. I would also thank Haykel, who was a very nice traveling companion during my first conferences overseas.
I also thank my parents for the inspiration they have provided, but also their help and support throughout these years. I thank all my family and friends for their support and I would like to give special thanks to Prof. Ioan Lie for the good collaboration I have had with him during my bachelors degree. Without his collaboration I would not have had the extra skills that enabled me to come to EPFL.

Contents

List of Figures

List of Tables

1 Introduction

The field of embedded and even general purpose computing has evolved considerably as it is driven by increasingly more demanding application requirements. A look at todays applications for consumer electronics shows a great demand for computation power and high flexibility, but with great constraints on the power consumption and device size. With the introduction of *High Definition (HD)* television, video decoding alone, requires 70 to 80 *Giga Instructions Per Second (GIPS)* [33]. However TV platforms today provide much more functionality than just simple video decoding which can range from browsing the Internet or running voice over IP applications, to fetching the video stream from WiFi enabled storage server. Similarly mobile communication devices have evolved from simple phones to full-fledged computation platforms with multimedia capabilities, high-bandwidth data communication connections and even navigation capabilities. Future applications like 3D-enabled displays and high bandwidth wireless communication like 4G will increase the requirements for computation even further.

Traditionally computation power was increased by raising the operating frequency as transistors in new technology nodes were faster. The power consumption was managed by scaling the supply voltage. As supply voltage scaling is leveling off in new technology nodes, computation power can only be increased at manageable power densities by exploiting the application-level parallelism. Therefore we see a design paradigm shift from single processor systems to *Multi Processor Systems-on-Chip (MPSoCs)*. MPSoCs are communication centric as the processors communicate over a global interconnect in order to cooperatively complete the parallel application. As such the MPSoC performance is directly influenced by the efficiency of the communication infrastructure. Recent MPSoCs prototype chips [162], [166], [67] and even products [164] have adopted *Networks-on-Chip (NoCs)* as the global on-chip interconnect. NoCs originate from general networks, but have been adapted to take advantage of the locality and silicon technology. NoCs have been proposed as a solution to

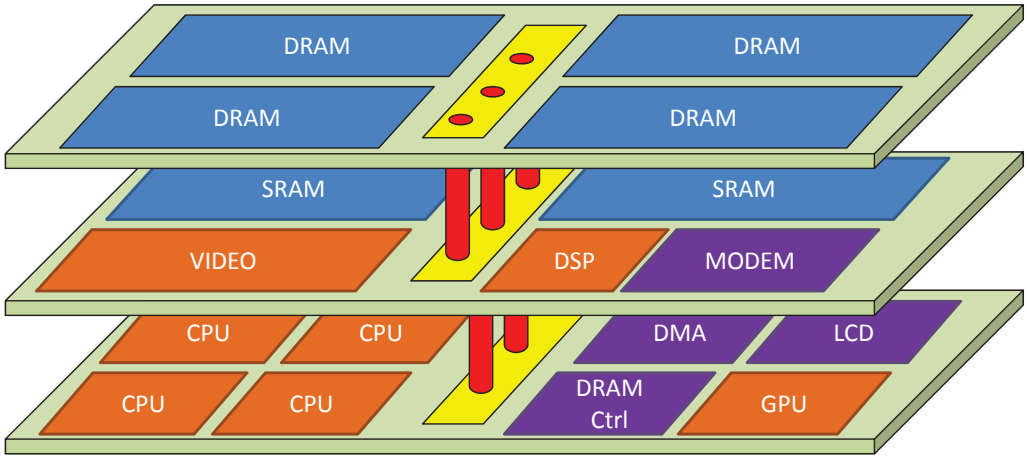

Figure 1.1: Example of a 3D MPSoC with three stacked layers

the interconnect design challenge due to predictability of design and the intrinsic scalability of the networks.

In power-constrained systems, like mobile platforms where increasing the autonomy is an important design challenge we see that programmability is traded for efficiency. Therefore the SoCs are customized for the required applications and many of the cores specialized to provide a specific functionality. The global interconnect can also be tailored according to the demands of the application. Customizing the interconnect is a must, as it can be responsible for as 30% of the SoC power consumption [162]. Already in the new-generation mobile platform chips [74] we can find NoCs as the global interconnect, not only for their scalability, but also because their modularity provides great opportunities for customization.

CMOS scaling is projected to continue at least until the end of this decade [14]. However the cost of designing and manufacturing integrated circuits in the new technology nodes is becoming prohibitive even for high-end systems. *Three Dimensional Integration* is emerging as a promising alternative for increasing the number of on-chip transistors, by stacking multiple dies in the same package. Unlike existing *Systems-on-Package, Three Dimensional Integrated Circuits* (3D-ICs) provide less restrictive vertical connectivity in the form of *Through Silicon Vias (TSVs)*. An example of a 3D SoC is shown in Figure 1.1. As it will be possible to integrate more cores in 3D-ICs to provide increased functionality, Three Dimensional Integration provides further challenges and opportunities to design an optimized on-chip interconnect. However to be able to explore all the degrees of freedom when customizing the interconnect and to improve the design productivity, tools are needed to perform architectural level exploration as well as synthesis.

1.1 Network-on-Chip a structured interconnect for Systems-on-Chip

Early MPSoC chips with few cores used a single bus for on-chip communication. However as the bus is a shared medium of communication, it cannot scale as the number of integrated cores is increased. As a consequence more complex design started using hierarchies of shared busses to exploit the parallelism in communication and to reduce collisions among cores that desire to communicate at the same time. As wire delay increases in new technology nodes due to the increased coupling capacity caused by the tall aspect ratio of the wires [5], [64], busses are becoming unpredictable from the timing point of view. The timing of the bus can only be estimated correctly after the place and route and this increases the time for achieving design closure. The introduction of crossbars in designs solves the problem of parallel access to different cores and also eliminates the long wires form the buses. Crossbar based designs do not scale though, as the crossbar is a centralized device. To solve both the problem of predictability as well as the scalability of the interconnect infrastructure, a network-based interconnect inspired from general networks designed for the super computes has been proposed as early as the year 2000 [60]. Many of the current high-end SoCs employ NoC as the global interconnect [74], [67] and even products [164].

The use of packet-switched networks to interconnect components in a computing systems has been proposed by Seitz and co-workers [152]. Networks have then been studied and used in multi processor super computes and an overview of the interconnect networks is presented in [41]. As MPSoCs became more complex integrating more cores on the same chip the use of *Networks-on-Chip (NoCs)* was first advocated by Greiner in [60] and within the Scalable Programmable Integrated Network (SPIN) project [2] and elaborated in its various facets by Benini et al. [23], [42], and Dally et al. [40] in the early part of the previous decade.

One of the most important characteristic on NoC is that they are modular. An example showing the NoC components is presented in Figure 1.2. The three basic modules to build an NoC are:

- Network Interfaces (NIs)
- Switches
- Links

Most components of the SoCs are *Intellectual Property* (IP) cores which use standard bus-like communication protocols. The role of the Network Interface is to convert the bus protocol used by the IP cores to the network protocol used by the switches. One NI is needed to connect each IP core to the NoC. NIs convert the transaction

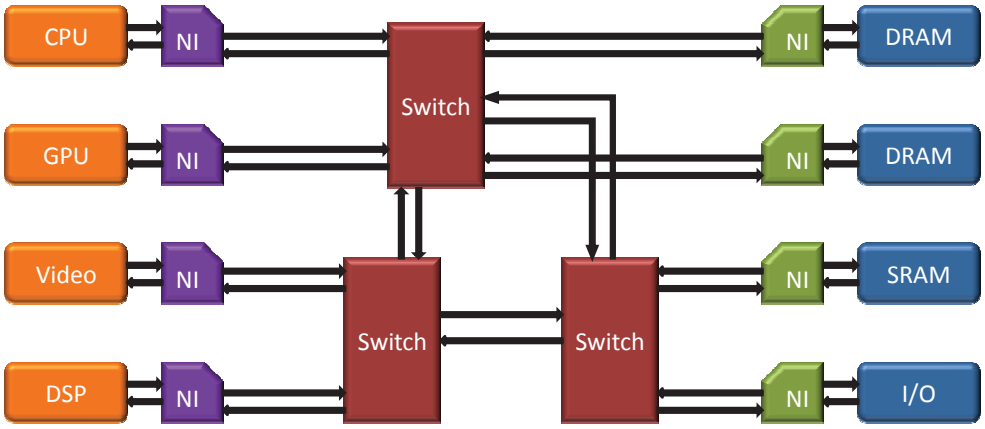

Figure 1.2: Example of a topology showing: IP cores, NIs, Switches and Links

into packets that are then sent though the network and then reassembled back into transactions by the destination NI. Packets usually contain more information bits than the number of wires that connect the NoC components. So the packets are serialized into a sequence of *FLow control unITS (flits)* before transmission, to decrease the physical wire parallelism requirements. Even though there are no standards NoC protocols for intra-network communication and are implementation dependent, most NI support standard protocols (e.g., OCP, AHB, AXI, Wishbone, OPB, PLB) toward the IP cores. Existing IPs to be connected easily to the network without requiring any modifications. The NI translation layer provides greater flexibility in connecting IPs using different protocols to the same system interconnect.

Switches provide the main infrastructure for the NoC in order to route packets from source to destination. Switches can have any number of inputs and outputs and a crossbar that allows for arbitrary connectivity between several inputs and several outputs. Of course the number of inputs and outputs is restricted by the desired operation frequency and the available area. Switches can be connected in different topologies in order to provide connectivity for many IP cores. Apart from the crossbar switches also provide buffering resources to lower congestion and improve performance. The buffers could be placed at the input ports (input-queued router), output ports (output-queued router) or at both places. In many NoCs with regular topologies where one or few cores are connected to a switch, the functionality of the NI is integrated in the switch itself. In such topologies it is common to call as a *router* the combination of the NI with the switch.

Links are used to connect NI to switches as well as provide the switch to switch connectivity. As links are point to point connections between NoC components their

timing characteristics are more predictable than those of busses. Links can represent more than just groups of parallel physical wires as they can provide pipelining in order to achieve the required timing and provide full throughput. Other components can also be added like frequency converters or size converters to further increase the flexibility of designing the interconnect in SoCs use multiple clock signals.

To summarize the NoC provide the following main advantages over bus-based design:

- Scalability - to accommodate more bandwidth, more switches can be added to increase the size of the topology.
- Predictability - as switches segment the wires and the connections are point to point, the timing characteristics can be estimated. Pipelining of the long links can be performed in order to preserve timing and throughput.
- Flexibility - the modularity of NoCs enables for the design of arbitrary topologies that can be customized to application requirements. Frequency and size converters can be seamlessly integrated to support multiple clock domains and heterogeneous IP cores.
- Interoperability - the NI protocol translation enables the connectivity of IP cores that use different communication protocols.
- Increased reliability - protocols for error detection and retransmission at link connection level within the NoC can be used to increase the reliability in critical systems or systems that are susceptible to faults.

As such NoC based design of the interconnect can result in faster design closure. In large systems there are still many parameters that need to be explored, however since the NoC design is predictable tools can be used for automatic exploration of the design space in order to achieve design closure.

1.2 Interconnect design challenges

Even though NoCs provide the scalability and predictability to design an efficient interconnect for large SoCs, there are many parameters that can be tuned. To design an efficient NoC based interconnect a vast design space needs to be explored. The main problems that need to be solved are:

- Choosing the IP cores and profiling the application to characterize the traffic.
- Decide on the technology design parameters: number of layers for a 3D-IC, the number of clock domains, the possibility to shutdown some clock domains.

- The assignment of IPs to clock domains and layers.
- Synthesis of a topology or mapping of the IPs onto a regular topology.
- Sizing the network parameters for the given application.
- Verifying correctness and performance.
- Generating the *Register Transfer Logic (RTL)* code.
- Perform RTL synthesis place and route, and post layout simulation and validation.

One of the most important challenges to design the interconnect customized for a given application is to determine the traffic characteristics. Once the IP cores and the algorithms that are to be implemented are known, architectural level simulation has to be performed in order to profile the application and to determine the traffic characteristics. The main parameters that describe the traffic patterns include: i) the average and pick bandwidth between cores, ii) the latency constraints required to offer a certain *Quality of Service (QoS)*, iii) the burstiness of the traffic, iv) the size of the transactions and v) a potential address trace. Along with the traffic characteristics the technology constraints have to be decided before the interconnect can be designed. The technology used (2D planar, 3D integration), the assignment of IP cores to clock domains or silicon layers in a 3D-IC, generate certain constraints that the interconnect has to fulfill for the SoC to function correctly. Therefore when building a customized interconnect, the analysis of the application is crucial as any mistake at this point can lead (in the best case) to a suboptimal design of the interconnect.

Designing the topology of NoC is the step that can make the most significant difference with regard to the power consumption and performance of the NoC. The parameters that have to be decided during synthesis are: choosing the number of switches, deciding the assignment of IP cores to switches, deciding the switch to switch connectivity and routing the flow such that the power is minimized under performance and technological constraints. As the design space is large there may be no optimal solution for all the criteria, but rather a set of trade-off points. *Computer Aided Design (CAD)* tools have been designed in order to explore the potential trade-off solutions and find the most suited NoC topology for the given application. For SoCs that are homogeneous and the functionality is implemented using the same IP core, mapping the tasks to IP cores on different regular topologies can also be used. However provided that the application does not change significantly and does not require the flexibility of a regular topology, synthesis of a customized topology will most likely yield a better result.

Apart from the topology other parameters can be customized like the buffer sizes, link widths, arbitration policies, routing method that also influence the performance, power consumption and area of the NoC. To efficiently tune all these parameters a combination of CAD tools, simulation tools, designer input and iteration may be needed. To reduce the time to market the automation of as many of the design steps as well as the seamless integration with simulation tools is desired, with the designer interacting only at a high level (e.g. solution evaluation).

To reduce further the design time automatic generation of the RTL code for the NoC topology is also needed. Integration with standard synthesis and back-end tools and verification and post layout simulation is also required in order to ensure that there is no performance loss between the estimation of the synthesis topology synthesis tool and the final result after place and route. In the past years much research has been perform in automatizing several is not all the steps of the NoC based interconnect design.

1.3 Tools for designing NoCs

For general purpose SoC the traffic patterns are not known at design time as they are dependent on the variety of applications that will run on the system. The topology has to provide full connectivity in such systems. As such many regular topologies have been proposed and explored. Designing the interconnect for such systems is simply a matter of choosing the topology that has the properties which best suite the SoC and sizing it using simulation and general traffic patterns that are expected to be produced. For application-dedicated SoCs on the other hand, the traffic patterns are known at design time and this enables the customization of the interconnect according to the application demands in order to reduce area and power while maximizing performance. Therefore much research has been done on tools and algorithm for automating the different steps of NoC design for application specific MPSoCs.

There are three major directions in which tools have been developed:

- architectural exploration and mapping or synthesis;
- simulation and evaluation;
- automatic RTL code generation and synthesis.

One advantage of NoCs is the ease to design regular structures by simply replicating the IP cores along with the NIs and switches. Such homogeneous SoCs are useful for applications that require greater programmability as the specifications and requirement can change during the design time. For these reason a lot of research has

focused on algorithms and tools to map tasks to IP cores interconnected by regular topologies [68], [69], [122], [123]. Murali et al. also propose in [120] a tool to map tasks considering QoS constraints as well. In systems where the application is known at design time or the traffic can be characterized well even without an explicit implementation of the application, programmability can be traded for improved efficiency. This requires the use of customized IP cores (e.g. audio accelerators, video decoders, WiFi modules) as well a customized interconnect tailored to the application requirements. Several research works propose algorithms and methods to design application specific NoC topologies [134], [65], [8], [158], [62], [181], [177], [121].

To validate the output of the synthesis tools under dynamic load or to further adjust some other parameters of the NoC components (e.g. buffer sizes), simulation tools are also needed. Alternatively analytical models have been proposed in order to estimate the NoC performance faster than simulation. Analytical models however tend to trade-off runtime for accuracy so in many cases simulation is still required for the final solution validation. Simulation as well can be done at different levels of abstraction. Architectural-level simulators like GARNET [4], SICOSYS [136], NOXIM [130] abstract away some of the details of the RTL implementation like the method for packetization or ignore the time to generate the packet. This may result in some loss of accuracy, but overall can give a good performance evaluation and verify most notwork properties faster. To close the gap between the accuracy of RTL simulation and the speed of architectural simulation or of the analytical models, there have been several proposals for emulation of the NoC hardware on *Field Programmable Gate Arrays* (FPGAs) [16]. In some cases, analytical models can evaluate properties (e.g. worst case latency) that would require extensive simulation. Several research papers have proposed models for estimating worst case latencies for real-time system implementation [139], [92]. In [61], [124], the authors present methods for application-level performance analysis and data-flow based NoC performance analysis, respectively. A data-flow based analysis can also be used to speed-up network performance estimation when the traffic patterns can be modeled accurately to fit the models.

The research and development of ×pipesCompiler [76] and NoC library [160] addressed both the support for the automatic implementation of heterogeneous NoC topologies. Both a parametrized library ×pipes and a NoC hardware compiler were presented to address the problem of synthesis and optimization for heterogeneous NoCs by means of highly-configurable network building blocks, customizable at instantiation time for a specific application domain.

In order to handle the design complexity and meet the tight time-to-market constraints, it is not sufficient to automate most of these NoC design phases. To achieve design closure, the different phases should also be integrated in a seamless manner. Several such integrated design flows have been proposed in the research commu-

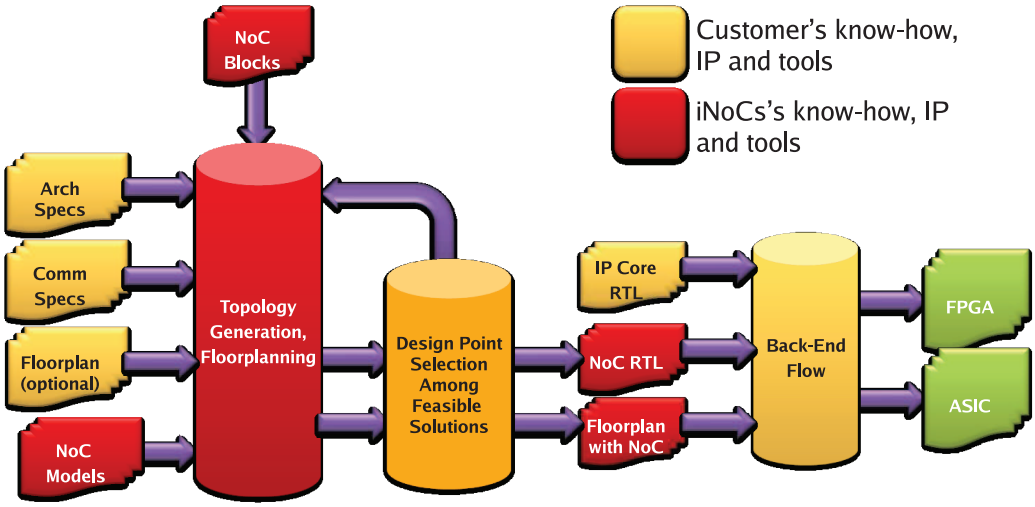

Figure 1.3: iNoCs NoC design flow [73]

nity [25], [140]. Some companies like: Arteris [170], iNoCs [73], Silistix [171] are also providing some commercial design automation tools for NoCs. As an example Figure 6 depicts the design flow from iNoCs. The tool flow starts from the application architecture, application constraints (e.g. number of IP cores) and the application communication constraints (e.g. the average bandwidth of communication between the different cores, average latency constraints). The design flow uses the provided tools to go from input specifications all the way to the RTL description of the topology. Also scripts are provided such that in conjunction with other commercial tools the RTL code can be synthesized for deployment onto *Field Programmable Gate Arrays* (FPGAs) or *Application Specific Integrated Circuits* (ASICs).

For the synthesis and architectural exploration phase there are still many opportunities that can be explored in order to improve the outcome of synthesis tools. The exiting tools do not take in to account the constraints that surface in new technologies. For example as wire delay grows in relation with the area of the die as technology continues to scale, it is not possible to design a fully synchronous SoC. Most design use multiple clock domains. To reduce the idle power IP cores in a clock domain can also be shutdown when not used. The interconnect must function correctly regardless so a synthesis tool must be aware of these properties and design the NoC accordingly. If 3D integration technology is used, further challenges need to be addressed like: limiting the number of vertical connections, assigning the NoC components to layers. Another problem with existing tools is that synthesis and performance evaluation are two decoupled steps. As such it may require many iterations before the required

Quality of Service (QoS) is achieved by the designed interconnect. By integration the performance evaluation during synthesis the number of design iteration could be dramatically reduced leading to faster design closure.

1.4 Thesis contribution

I present in this thesis methods for automatic design of NoCs considering the constraints of new technologies as well as QoS constraints. On one hand the thesis brings a strong engineering contribution as all the presented methods are oriented to solve real problems related to NoC design. The integration of the proposed algorithms into tools to automate the high level NoC design phase as well as the experimental results provided have direct applicability for designing NoCs. On the other hand the thesis has a scientific contribution as well as new algorithms are presented for NoC topology synthesis and deadlock removal.

There are two main technical contributions that this thesis brings:

1. I present methods to design efficient NoC to be used as global interconnect in SoC designed in new technologies (3D integration and multi synchronous domain) and
2. I integrate methods to evaluate the NoC performance within the synthesis algorithms such that different levels of QoS can be provided with an automated flow.

Finally I also present new architecture and methods to efficiently access bottleneck devices with high latency devices like DRAM memories.

1.4.1 Assumptions and limitations

For the rest of this work I made the following realistic assumptions:

- *Three dimensional integration adoption:* I assume that 3D integration technology will be adopted in the design of future SoC. As technology scaling is becoming more expensive with each technology node, 3D integration is a promising alternative to increase the number of on-chip transistors. This is a realistic assumption as we can already find some products (e.g. memories) that have adopted 3D integration.
- *Multi synchronous design:* Another assumption is that 3D SoCs will use multiple clock domains. Already in existing SoC designed in 2D technology it is not

possible to distribute one synchronous clock throughout the chip. It has been projected in [5] that in 35nm technologies less than 2% of the chip area can be reached in a single clock cycle at a frequency of 5GHz. Therefore it is likely that multiple clock domains will be needed in 3D as well. Also it may not even be possible to power all the cores at the same time as well [49] and considering that many IP cores provide specific functionality it is desirable to shut them down when they are not needed to save power.

- *Computation can be decoupled from communication:* The thesis focuses on designing the interconnect according to the application specifications. The application traffic characteristics can be evaluated through profiling of the applications that are executed on the IP cores. The interconnect can be designed using the obtained traffic characteristics without having the need for an explicit description of the computation IP cores. Therefore the design of the interconnect can be decoupled from the actual computation that is performed by the SoC cores.

- *NoC components are predictable:* The synthesis algorithm evaluate the NoC performance based on models of the NoC components. Therefore to decouple the high-level topology synthesis from the silicon implementation, models of the NoC components are used. The models are built from post place and route results for the given technology. The assumption is that the NoC components are predictable such that even after integration with the whole systems the NoC components will maintain the properties that they had when the models were built. This assumption is realistic as synthesis tools enables the hierarchical design, so the NoC components can be synthesized as if they were stand-alone modules. Also the links are point-to-point so their properties are easier to predict.

1.5 Thesis overview

The thesis focuses on two main aspects of NoC design. First, Chapters 3, 4 and 5 present methods for NoC design considering 3D integration technology and the use of multiple clock domains. Then in Chapters 6, 7 and 8 the thesis focuses on the integration of performance evaluation during synthesis and on design aspects to improve the Quality of Service. In the remainder of this section a detailed overview of this thesis is presented.

1.5.1 NoC design in new technologies

In Chapter 3, I present two algorithms for designing application-specific NoCs for 3D-ICs. The first algorithm is designed to produce power optimal topologies for SoCs with less restrictive constraints on vertical connections. The second algorithm is designed to produced results with small numbers of vertical connections and to guarantee direct connection only between adjacent layers in the case of more restrictive 3D integration technologies. The two algorithms are integrated in a tool together with methods to assign the NoC components to layers and to insert them in a valid floorplan. The two algorithms are fully integrated in a single tool and the tool automatically switches between the two to produce good results even under tight constraint on the number of vertical wires that can be used. As 3D integration promises to increase the heterogeneity of cores by stacking dies created with different technological processes, 3D SoCs could greatly benefit by customizing the interconnect according to the application demands. Therefore the chapter also presents an evaluation of the generated 3D topologies with respect to 2D topologies required for a similar sized 2D SoC. The benefits of application specific topologies with respect to 3D regular topologies are also explored.

As in advanced technologies it may take several clock cycles for an electrical signal to traverse the length of a chip on global wires [24], it also very expensive to distribute a synchronous clock throughout the die. Designing SoCs in new technology nodes require the use of either multiple clock domains or asynchronous global communication. In many application-oriented SoCs the IP cores are specialized for a specific functionality. When that functionality is not required, then those IP cores along with the clock domain can be shutdown to reduce the idle power consumption. Therefore the communication infrastructure must be aware of clock domain crossing and be able to provide the required functionality even when parts of the chip are turned off. In Chapter 4, I present an algorithm to design NoC topologies in the presence of multiple clock domains that have the possibility of shutting down. The algorithm is first described for designing 2D topologies and later extended to support multiple clock domains in 3D SoCs. An evaluation of the overhead of the NoC topology due to the partitioning the cores in different clock domains is described. Also a comparison of topologies for 2D and 3D SoC implementations in the presence of multiple clock domains is presented.

One important property for a NoC topology is to ensure deadlock freedom. If traffic flows are not routed with care, they could block by creating circular dependencies when reserving NoC resources. If deadlocks appear they would lead to the catastrophic failure of the communication infrastructure. In regular topologies deadlocks are prevented by restricting the routing function from using certain combinations of links. Similarly, by restricting the routing of flows from using certain sequences of

links during synthesis the resulting topology is deadlock free [121]. Alternatively deadlocks can be removed after synthesis, by adding parallel channels in the topology and changing the routes of some flows to use the new channels. In Chapter 5, I present a general algorithm for removing deadlocks for a given topology by adding a minimal number of parallel channels. This method can be used in conjunction with the synthesis algorithm to provide better results in highly-constrained environments. For example in a 3D-IC where there is a tight constraint on the number of vertical interconnect wires, synthesis could be performed first without the routing restrictions to prevent deadlocks. Then the deadlock removal algorithm is applied to the generated topologies to identify the deadlock conditions and remove them by adding parallel channels in the parts of the topology that belong to a single layer of the 3D-IC.

Thus the first part of the thesis focuses on providing new NoC topology synthesis algorithm that account for the constraints imposed by new technologies.

1.5.2 Adding QoS to NoC synthesis

In the second part, the thesis focuses in providing different levels of QoS for the synthesized topologies. I present several ways of integrating the evaluation of the NoC performance during synthesis in order to steer the synthesis algorithm toward better solution.

Some of the applications have services that require hard latency constraints for some flows (e.g. interrupts sent through the network). Many works have addressed the problem of ensuring worst case latency bounds, but they all focus on using specialized hardware for this purpose. For example the Æthereal NoC [57] uses *Time Division Multiple Access* arbitration in order to provide for guaranteed services. However as most SoCs for consumer electronics have only few services that require hard guarantees, using specialized hardware may lead to over design. In Chapter 6, I show how the analytical models from [139] for computing worst-case latency bound, can be integrated in the synthesis process. By careful design of the topology, worst case guarantees can be provided while using only best effort NoC components. This method is useful since most commercial NoC libraries [170] currently support only best effort services.

Bandwidth-intensive applications usually require only average case guarantees. For example, in video decoding, frames have to be processed at a certain rate. The frame rate has to be achieved on average and if eventually some frames are dropped, the quality of video decoding is not severely impacted. The challenge in designing the NoC topology for average case performance come from actually estimating the average case performance. Many of the analytical models proposed for estimating average

case have high error when compared to full system simulation. However full system simulation is time consuming and not suited for integration within the topology synthesis algorithms. I propose a partial simulation model to only simulate the part of the NoC that is currently designed. This model is described in Chapter 7 and can significantly speedup simulation making it suited for integration in the synthesis process. The partial simulation model is also integrated within a the synthesis flow and I present results to attest the benefits.

Providing QoS for applications does not involve only the NoC, but also the peripheral devices that are accessed need to be considered. Most of today's SoC are memory centric. To provide sufficient storage capacity *Dynamic Random Access Memory (DRAM)* is used. However DRAM can have high access latency and it is also unpredictable especially under heavy load. In Chapter 8, I analyze different NoC architecture and propose methods for efficiently accessing shared memory in the form of external DRAM.

Finally in Chapter 9, I conclude the thesis results and look at future avenues that can be pursued.

2 Background

In this chapter I will present on overview on some important topics that make a basis for the thesis. The topics include: NoC operating principles, Three dimensional integration, multi synchronous design and Quality-of-Service. With each topic I will also review the previous and current work in that field.

2.1 NoC principles

A complete description of interconnect networks can be found in [41]. In this section I will present only some of the basic operating principles of NoCs. The main topics related to NoCs include:

- hardware architecture
- topology
- switching and flow control
- routing and arbitration

To provide realistic results for the evaluation the ×pipes library [160] is used for most of the experiments. Therefore a description of this library is provided for each of the four topics.

2.1.1 NoC hardware

Many NoC architectures have been proposed. A large fraction of these are natively synchronous, such as ×pipes [160], NOSTRUM [89], Spidergon [37]; others are conceived to support asynchronous operation, such as Mango [27], FAUST [112] and ANOC [21].

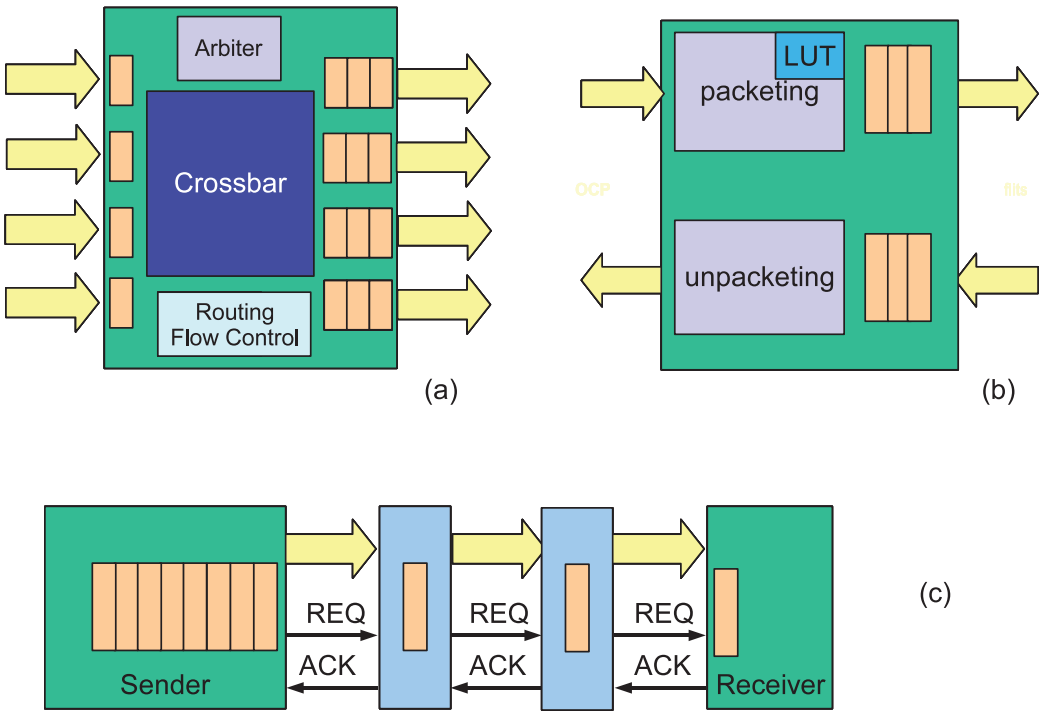

Figure 2.1: ×pipes building blocks: (a) switch, (b) NI, (c) link

Some architectures were presented to achieve predictable QoS behavior, using special hardware mechanisms, such as Æthereal [57], QNoC [29].

As an example, a brief description of the basic NoC library components, based on ×pipes [160], is presented in Figure 2.1. This library incorporates features that have been successful in many NoC designs.

There are three main components: i) *Network Interfaces (NIs)*, ii) switches and iii) links. In ×pipes, two separate NIs are defined, an *initiator* and a *target* one, respectively associated with system masters and system slaves. A master/slave device will require an NI of each type to be attached to it. The interface among IP cores and NIs is point-to-point as defined by the *Open Core Protocol OCP 2.0* [156] specification, guaranteeing maximum re-usability. NI *Look-Up Tables* (LUTs) specify the path that packets will follow in the network to reach their destination (source routing). The switches are parameterizable and can be configured at design time to meet requirements. Both the number of inputs and outputs can be set as well as the amount of buffering and the FLIT size. Similarly the amount of buffering and the FLIT size can be configured in the NI. The links are point to point connections and can be pipelined in order to meet the required cycle time. Because the library is configurable it is suited to build application specific topologies.

Other components can be used to increase the flexibility of the library. Among these, two other components are worth mentioning as they are used in this work: frequency converters and size converters. Frequency converters are needed in multi synchronous designs. When a link crosses from one clock domain to another a frequency converter is needed for a correct operation. There are several types of converters depending on the relation between the two clock domains. Mesochronous converters [102] are needed if the two clock domains have the same frequency, but different phase. This could happen in a 3D SoC as the clocks on different layers may not be aligned. If the frequency of the two clock domains is different then a dual clock FIFO is required to bridge the two domains. As the cores in an application specific SoC are heterogeneous they could have different data sizes (e.g. DRAM memory controller usually has a wide data interface to provide more bandwidth as it is shared by many IP cores). In such case FLIT size converters may be needed in order to connect components with different data size.

2.1.2 NoC topology

The topology defines how the NoC components are connected to form the on-chip interconnect. For general purpose SoCs many regular topologies have been proposed. There are two main categories of topologies: i) *direct topologies* and ii) *indirect topologies*. In direct topologies, to each switch at least one IP core is directly connected. Examples of direct regular topologies range from: mesh, torus, ring, Spidergon [37], hypercube. In indirect topologies, there are switches that are only connected to other switches. Examples of indirect topologies are: clos, butterfly, fat-tree. Graphical representations of some of this regular topologies are presented in Figure 2.2. Every topology provides different trade-offs in terms of power, area, latency, bandwidth and depending on the SoC requirements and constraints, one of the topologies may be better suited. Mesh is probably the most popular topology due to the implementation simplicity as the same switch architecture can be used for all the routing nodes. Many existing designs or prototypes use mesh or a set of meshes for the NoC topology [166], [162], [67], [164].

Customizing the topology is one of the best way to improve the performance/power-consumption trade-off, for application oriented SoCs. In custom topologies the switch sizes can have arbitrary values as depicted in Figure 2.2.g. The connectivity among the switches is also arbitrary and different number of IP cores can be connected directly to the same switch. When designing application-specific topologies the following problems have to be solved: i) find the connectivity of IP cores to switches and from switch to switch such that all communication flows can be established; ii) achieve the required performance with minimum power. Most approaches for constructing application specific topologies focus on direct topologies. This is because most of the

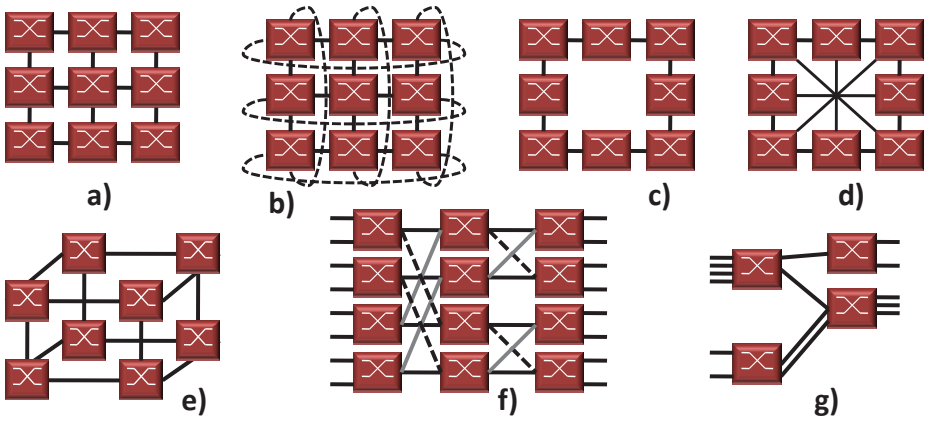

Figure 2.2: Examples of topologies: a) mesh, b) torus, c) ring, d) spidergon, e) hypercube f) butterfly and g) application specific

time topologies with fewer switches than cores can achieve the required performance with lower power consumption. In this work I will show how indirect topologies can also be built, in multi synchronous designs, in order to allow the possibility of shutdown of clock domains.

2.1.3 Switching and flow control

There are two main types of network switching technologies:

- Circuit switched (connection oriented).
- Packet switched.

Circuit-switched network are similar to the early telephone networks where a connection from source to destination has to be established prior to sending data. The advantage of such a network is that it does not require buffering and the throughput and latency are guaranteed once the connection is established. The main disadvantage of such flow control is the poor utilization of the network resources. In packet-switched networks, messages are split into packets and the packets are routed in the network from source to destination. Packets from different paths compete for network resources at different points along the paths and this allows for a much better utilization of the network resources. Even though there were some proposals of circuit-switched NoCs [48] and some proposals for hybrid circuit- and packet-switched NoC to provide guaranteed services for QoS [114], [115], most of today's implementations use packet-switched protocols. Therefore I will focus on the latter.

Several flow control protocols have been proposed for the packet switched networks as well:

- Buffer-less routing (Deflection routing);
- Buffered routing:
  - Store and forward;
  - Cut-through:
    - Wormhole flow control;
    - Virtual channel flow control;

Buffer-less routing has been proposed for networks that have low load [117], [63]. Unlike circuit switched networks the messages are divided in packets, but the right to access a channel is arbitrated at each switching point for each packet. In case the desired channel is busy than the packet is either deflected or dropped. In the latter case the packet would need to be retransmitted. However it has been shown in [109] that buffer-less networks quickly loose performance in the presence of congestion. For buffered networks two main switching methods exist: i) *store and forward* and ii) *cut-through*. In *store and forward* networks, like most Internet networks, a packet has to be completely stored in the buffer of a switch port before it is forwarded to the next switch. This flow control requires a lot of buffering capacity and has high latency as well. Therefore it is not really suited for NoCs which are area and latency sensitive.

Most NoC implement variants of the *cut-through* protocols. In *wormhole* flow control protocol a packet is split in *FLow control unITs (FLITs)*. The FLITs are sent in the network one after the other like a worm, which inspired the name of the protocol. Arbitration for resource reservation is done by the header FLIT and the resource remains reserved until all the FLITs of the packet have passed. This reduces both the buffering requirements and the latency of the packets. As packets can be blocked waiting for resources they can span across multiple switches blocking other packets. To improve the utilization of the links, *virtual channel* flow control proposes the use of multiple buffer (virtual channels) in the same port and when packets are blocked in one buffer the packets from the other buffers can be forwarded. Many NoC libraries that are dedicated to application specific NoCs like ×pipes [160] do not support virtual channels, as on-chip wiring is less expensive so multiple physical channels can be used instead to provide similar effects.

In ×pipes, switches use wormhole switching, as it is most common in NoCs, but support two variations of flow control. If *ACK/NACK* flow control is used then output buffers are required, as flits have to be retransmitted until the downstream router has sufficient capacity to store and accept them. If *ON/OFF* flow control is used,

back-pressure from the downstream switch stalls the transmission until the there is sufficient buffering capacity. In this case, output buffers can be omitted. In any case, the arbiter is required to resolve conflicts between packets when they require access to the same physical link.

2.1.4 Routing and arbitration

Routes determine the paths the packets follow to go from the source to the destination. The route is a set of switches and output ports that are used by a packet in sequence to reach the destination. Most if not all NoC implementation use *deterministic* routing. In *deterministic* routing one or more routes are defined from each source to destination and the packets are sent using one of those routes. In case of adaptive routing algorithm multiple routes are used from each source to destination and the route is selected such that certain goals are achieved (e.g. load on the NoC is balanced). Most implementation for application specific NoC use static routing to reduce the complexity of switches and NIs and because the traffic is known so it can be balanced when the routes are generated at design time.

Several mechanisms have been proposed for specifying the routes at the hardware level. The most used mechanisms are: node routing and source routing. In node routing, at each switch the next port is calculated based on the source and destination of the packet. The calculation can be done either by a hardware implementation of a routing algorithm or by a table contain the next port for all source destination combinations. Node routing is very useful in regular topologies where the next node can be calculated using a simple algorithm. In case of application specific topologies source routing is preferred as it is more flexible. For source routing, the NI holds a table with all the routes to destinations that can be reached. When a packet is generated based on the destination the route is included in the packet header. At each node the next node is selected based on the information in the packet headed. At design time the routes have to be decided and set in the NI table for the NoC to function correctly. The ×pipes [160] library uses deterministic static source routing.

When two packets need the same resource the conflict between them has to be resolved by an arbiter in the switch. Two of the most used arbitration schemes are *priority* arbitration and *round-robin* arbitration. For priority arbitration each input channel has a unique static priority and the packet on the channel with the highest priority wins. This policy is easy to implement, but it is not fair as their highest priority port can prevent the other from sending when it is congested. *Round-robin* arbitration is fair as the port that sent the packet last receives the lowest priority so all the ports that have packets to sent will get serviced. The ×pipes [160] library supports both modes. In case of virtual channels more complex allocators have been proposed, to

find the best matching between virtual channels and output port in order to maximize the throughput [41].

Other arbitration and routing schemes have been developed in order to offer support for predictable communication behavior. The *Æthereal* NoC design framework presented in [57] aims at providing a complete infrastructure for developing heterogeneous NoC with end-to-end quality of service guarantees. The network supports *guaranteed throughput* (GT) for real time applications and *best effort* (BE) traffic for timing unconstrained applications. The architecture offers so-called GT connections which provide bandwidth and latency guarantees on that connection. In order to provide bandwidth and latency guarantees, it uses a *Time Division Multiple Access* (TDMA) mechanism to divide time in multiple time slots, and then assigns each GT connection a number of slots. The result is a slot-table in each NI, stating which GT connection is allowed to enter the network at which time-slot. For traffic that has no real-time requirements, Æthereal implements Best-Effort connections.

2.2 Three dimensional integration

As computation platforms continue to miniaturize, the integration of ICs over printed circuit boards is no longer possible due to tight area constraints. To improve the area efficiency *System-in-Package (SIP)* technology was developed, where multiple dies are integrated in the same package. Connectivity among dies is provided through wire bonds at the edge of the dies. The drawback of SIP technology is that the connectivity between dies is limited to not much more than that between different packages on the printed board. *Three dimensional integration* promises to remove this limitation by priding better vertical connectivity. An example of SIP technology and 3D integration is shown in Figure 2.3.

There are two main concepts for 3D integration:

- 3D-stacking of silicon layers with *Through Silicon Vias (TSVs)* to provide connectivity.
- 3D-monolithic integration

Stacking of silicon layers has emerged as a promising solution that addresses some of the major challenges in today's 2D designs [18], [26], [58]. In the 3D stacked technology, the design is partitioned into multiple blocks, with each block implemented on a separate silicon layer. The silicon layers are stacked on top of each other. Each silicon layer has multiple metal layers for routing of horizontal wires. Unlike the 3D packaging SIP solutions, the different silicon layers are connected by means of

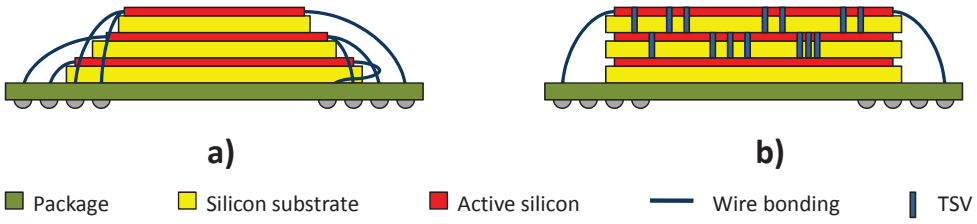

Figure 2.3: Example of: a) System-in-Package technology and b) 3D-stacking with TSV connectivity

on-chip interconnects known as *Through Silicon Vias*. TSVs are built by drilling holes in the silicon die and filling them up with metal. Even though the size of the TSVs is quite large relative to the size of the transistors, the density of TSVs is high enough to provide system level integration.

On the other hand 3D monolithic integration proposes the use of molecular bonding to add thin silicon wafers on top of already processed wafers and to create more transistors on the top layer. The density of the connection is higher and this technology is more suited for 3D integration at transistor level [19]. Therefore 3D monolithic integration can be used to enhance the performance of an IP core, while 3D-stacking allows for the integration of many IP cores (which can be processed with different integration technologies). As this work is concerned with the interconnection of IP cores, I will focus on 3D-stacking in the remainder of this section and of this work. Please note that potentially the two technologies could be used in conjunction to speedup the IP cores and to integrate more cores to provide more functionality.

The 3D-stacking technology has several major advantages:

- The footprint on each layer is smaller, thus leading to more compact chips.
- Smaller footprints lead to shorter wires within each layer. Inter layer connections are obtained using efficient vertical connections, thereby leading to lower delay and power consumption on the interconnect architecture.
- Heterogeneous integration of diverse technologies is possible. Each layer could be designed as in a different technology specific to the different IP cores (e.g. memories, analog amplifiers) resulting in better efficiency and performance

A detailed study of the properties and advantages of 3D interconnects is presented in [18] and [172].

Vertical stacking of multiple silicon layers, referred to as 3D stacking, is emerging as

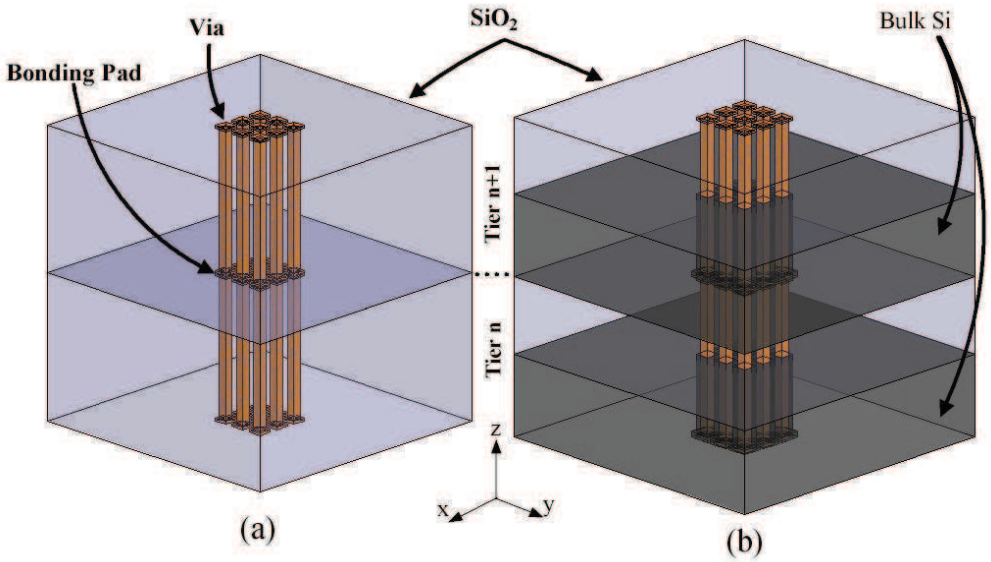

Figure 2.4: An example set of 9 vertical links [72]

an attractive solution to continue the pace of growth of Systems on Chips (SoCs) [18], [26], [58], [36], [70], [71], [96], [180]. Several technologies and methods are available in the literature for performing 3D integration. In the *Die-to-Die* bonding process, individual dies are glued together to form the 3D-IC. In the *Die-to-Wafer* process, individual dies are stacked on top of dies which are still not cut from the wafer. The advantages of these processes are that the wafers on which the different layers of the 3D stack are produced can be of different size. Another advantage is that the individual dies can be tested before the stacking process and only "known-good-dies" can be used, thereby increasing the yield of the 3D-IC. In the *Wafer-to-Wafer* bonding, full wafers are bonded together. For connection from one layer to another, a TSV is created in the upper layer and the vertical interconnect passes through the via form the top layer to the bottom layer. Connections across non-adjacent layers could also be achieved by using TSVs at each intermediate layer. The integration of the different layers could be done with either face to face or face to back technologies [101]. The face of the die is considered to the metal layers and the back is the silicon substrate. The copper half of the TSV is deposited on each die and the two dies are bonded using thermal compression. Typically, the dies are thinned to reduce the distance between the stacked layers. Several researches have addressed 3D technology and manufacturing issues [26], [58], [72], [82], [72]. In [26] the authors make a detailed analysis of performance gains for 3D technology. The authors analyze how the area, power and wire length is affected when moving from a 2D design to 3D. In [58] a

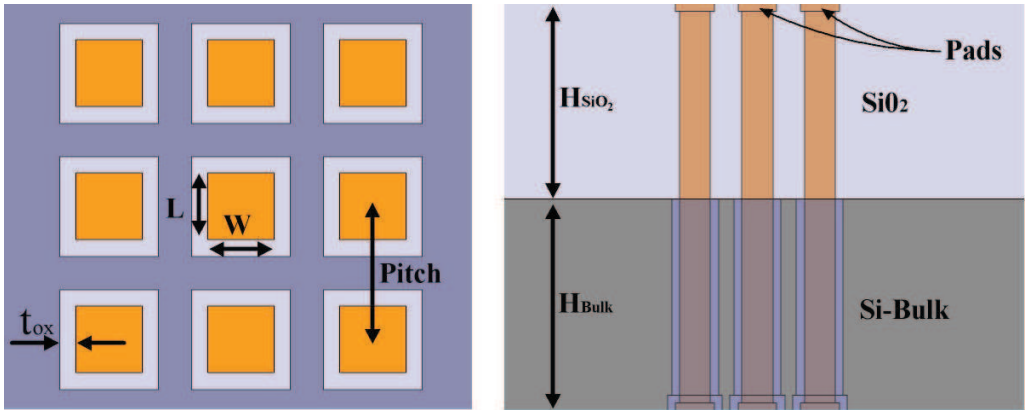

Figure 2.5: 3D bundle cross-section [72]

method for placing thermal vias is shown. The method places the thermal vias in order to keep the temperature under a desired maximum value and at the same time tries to minimize the number of thermal vias needed. Models for vertical links containing several parallel wires are presented in [72]. The authors also present methods for using the vertical links within available 2D design flows. Several industrial labs, CEA-LETI [59], IBM [163], IMEC [75] and Tezzaron [15], to name a few, are also actively developing methods for 3D integration.

In Figure 2.4, a set of vertical wires using TSVs implemented in SOI and bulk silicon technologies [72] are showed. There is also the schematic representation of a bundle of TSV vias in Figure 2.5. In [72], a 4 μm x 4 μm via cross section, 8 μm via pitch, 1 μm oxide thickness and 50 μm layer thickness are used. As can be observed from the dimensions the size of the TSVs allows for good connectivity at systems level.

A major concern in 3D chips is about managing heat dissipation. In 2D chips, the heat sink is usually placed at the top of the chip. In 3D designs, the intermediate layers may not have a direct access to the heat sink to effectively dissipate the heat generated. Several researchers have been working on all these issues and several methods have been proposed to address them. At system level, the problem of partitioning and floorplanning of designs for 3D integration has been addressed in [36], [70], [71], [96], [180] among others. In [36] a new data representation is presented that contains the necessary data for performing floorplanning using simulated annealing. The temperature of the chip is used in the objective function that is minimized. An RC model is also presented to calculate the temperature distribution that is used in the objective function. A force-directed 3D floorplanner is presented in [180]. The floorplanner efficiently takes into account physical information like temperature

gradient and the authors also present methods to transition from an unconstrained 3D assignment to a legal layer assignment without overlap. At the circuit level, use of thermal vias for specifically conducting heat across the different silicon layers has been used [58]. In [163], use of liquid cooling across the different layers is presented. Managing heat dissipation is however an orthogonal problem to interconnect design. To decouple the two problems the placement and assignment of IP core to layers in the 3D stack is assumed to be provided as input.

There are also works that address however the problem of interconnects in 3D SoCs. Multi dimensional regular topologies (like k-ary n-cubes, hypercubes) have been explored by researchers as viable interconnect solutions for chip-to-chip networks [39]. However, such standard topologies are not suitable for application specific SoCs, which are heterogeneous in nature. Synthesis of NoCs for 3D SoCs is a relatively new topic. New switch architectures for 3D have been presented in [86] and [129]. In [95], the authors present the use of NoCs as interconnects for 3D multi-processors. The electrical characteristics of vertical interconnects are analyzed in [72] and the authors also present a back-end design flow to implement 3D NoCs. Design of standard topologies for 3D is analyzed in [52] and mapping of cores on to NoC topologies is presented in [1]. Power-delay analysis of 3D interconnects is presented in [132]. However, none of these works address the issue of synthesizing custom NoCs topologies for 3D SoCs.

2.3 Multi synchronous design

In recent SoC design we can already see the use of multi synchronous design. The IP cores are not all assigned to the same clock source, but there are multiple clock sources. There are many reasons for using multiple clock signals, but the three main reasons are the following:

- Implementing a single synchronous clock in new technologies is expensive. As feature scale, the aspect ratio of wires increase the capacitance making them slower. As it can take several clock cycles a signal to propagate across the length of a die [24] it is very difficult to spread one synchronous clock signal to all the cores. The cost in power or in the operating frequency is to high for most designs.

- With the leveling of voltage scaling to maintain the power density at acceptable level not all the cores can be operated at the maximum frequency [66]. Dynamic scaling of the frequency is used to operate the IP cores at the minimum necessary frequency to perform their tasks. As in an SoC the different IP cores perform different functions they may need to operate at different frequencies. Since the power budget remains constant, as more cores will be available in future technologies (e.g. 3D-IC), it will not be able to even power all of them at the

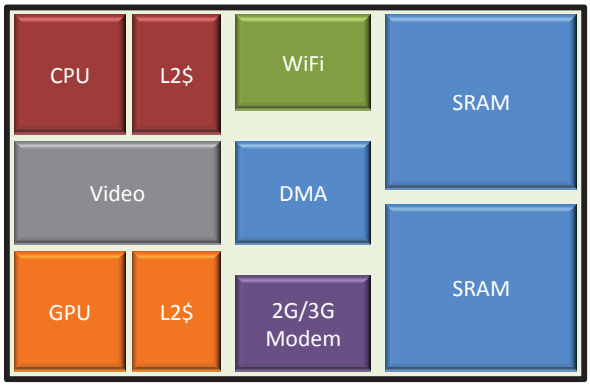

Figure 2.6: Floorplan example of an SoC with 6 clock domains

same time [49].

- With technology scaling, the leakage power consumption is increasing rapidly as a fraction of the total power consumption. In fact, leakage power can be responsible for 40% or more of the total system power [51]. Reducing the idle power is especially important in mobile devices, but it should not be neglected in devices connected to the power grid either. In application oriented SoCs many IP cores are dedicated to specific functionality. Therefore they can be shutdown when that functionality is not needed.

In Figure 2.6 there is an example of an SoC where the cores have been assigned to 6 clock domains based on functionality. The different domains are represented by different colors. The assignment of IP cores to clock domains based on functionality allows for the shutdown of those clock domains when that functionality is not used.

In order to reduce the leakage power consumption, cores that are not used by an application can be shutdown or placed in sleep mode, while the other cores can be operational. For example, power gating using sleep transistors is a popular way to shutdown cores [51]. To achieve power gating, the sleep transistors are added between the actual ground lines and the circuit ground (also called the virtual ground) [51], which are turned off in the sleep mode to cut-off the leakage path. Due to routing restrictions, separate VDD and ground lines cannot be used for each core. Instead, cores are grouped in to *Voltage and Frequency Islands (VFIs)*, with cores in an island using the same VDD and ground lines [90], [143], [165], [106]. When all the cores in an island are unused for an application, the entire island can be shutdown. For example, the IBM fabrication processes CU-08, CU-65HP and CU-45HP all support the partitioning of chips into multiple VFIs and power gating of the VFIs [155].

Power gating of designs has been widely applied in many SoCs [90]. In [90], the authors present the importance of partitioning cores in to voltage islands for power reduction. Several methods have been presented to achieve shutdown of islands [90], [143], [165], [106]. Any of these methods can be used in conjunction with the topology synthesis process to achieve the actual shutdown of cores.

There are several works that have tackled the problem of designing NoCs for multi synchronous SoCs. These works however address the hardware implementation so the use regular topologies and do not address the application specific topology synthesis problem. In [27] an architecture for *Globally Asynchronous Locally Synchronous (GALS)* NoC is presented. In [112] the authors present a physical implementation of multi-synchronous NoC. Architectures for designing NoCs using GALS paradigm are also presented in [22] and architectures for designing NoCs for GALS and DVFS operation are shown in [20]. The authors tackle the issues at the hardware level, by presenting designs for the network components, but not the architectural synthesis problem from custom NoC. In [126], the authors present a design methodology for partitioning a NoC into multiple islands and assigning the voltage levels for the islands. This work is complementary to this thesis, as I present a methodology for designing a custom NoC topology starting from an existing assignment of cores to VFIs. In [94], the authors present an approach to design NoCs with voltage islands. However, the designs produced by the method do not support the shutdown of the islands.

In [45], the authors present approaches to route packets even when parts of the NoC have failed. A similar approach can be used for handling NoC components that have been shutdown. However, such methods do not guarantee the availability of paths when elements are shutdown. Moreover, mechanisms for re-routing and re-transmission can have a large area-power overhead on the NoC [119] and are difficult to design and verify.

2.4 Quality-of-Service

Many classes of applications require some amount of determinism in the behavior of the communication flows. Providing guarantees for communication flows in NoCs is usually termed as providing *Quality of Service (QoS)* guarantees. Usually, the QoS guarantees translate into bandwidth and latency constraints for the traffic flows. The bandwidth constraints can be met by allocating enough resources, giving a bound on worst case latencies is a bigger challenge. While the guarantees that are required by applications in the multimedia domain are concerned with the average case, and infrequent cases that do not meet the requirements are allowed (also called *soft QoS*); there are applications in safety critical domains (such as automation, aerospace, military) that require hard bounds that must be always met (also called *hard QoS*).

2.4.1 QoS at NoC level

The NoC is one of the source of unpredictability as many IP cores with different traffic patterns access it. To provide QoS for application the NoC has to provide a certain degree of predictability. This can be done either through specialized hardware or through appropriate sizing of the communication resources and thorough analysis. There have been many works that provide QoS guarantees by adding specialized hardware to the NoC architecture. The Æthereal NoC, presented in [57], adds guaranteed services on top of best effort services to provide QoS support. In [88], the authors describe the MARS architecture where TDMA *(Time Division Multiple Access)* is used to provide QoS guarantees. In [153], a priority based scheme is used to provide QoS and in [131] the authors present the concept of a predictable *Time-Triggered NoC*, where QoS is ensured through communication services. In [107] the authors present a QoS NoC architecture that uses both best effort and guaranteed traffic, however requires specialized hardware and virtual channels. Many works present improvements and variations [29], [28], [30], [54], [80], [93], [108], [111], [116], [118], [137], [141], but all use specialized hardware to provide QoS. A recent survey of NoCs [47] shows that most NoC architectures do not have specialized hardware for QoS and therefore a method to design NoCs that provides guarantees on the delay of communication flows is necessary.

In [92], the authors present a method to characterize the real time behavior of communication flows in best effort wormhole general networks for parallel computing. However this method requires traffic regulation, which is not desirable in a majority of applications. In [139], we presented models to analyze and compute worst case latency bounds, even the input traffic is not regulated. In this work, we use the methods for calculating the worst-case latencies during the topology synthesis process itself, and we use the computed values to tune the instantiation of network components to produce topologies that meet the real time-constraints.

In [120], the authors consider critical streams and map them more efficiently than normal streams. However, they do not provide any bounds on the latencies. In [62], the authors present a method to synthesize NoCs for a TDMA based architecture. They also consider the design of TDMA slot tables to meet hard QoS constraints. However, the method only applies to NoCs where the underlying architecture uses additional hardware to support QoS. Moreover, with a TDMA scheme, the average performance of the system could be poorer than with a scheme where packet injection times are not so tightly constrained.

Providing soft QoS requires that average case guarantees are met. This complicates the problem over worst-case guarantees because it is harder to estimate the average case. Most often the average case is estimated through extensive simulation with

multiple variations of the benchmarks which is time consuming. Analytical models have been proposed as well [61], [124]. However analytical methods improve the estimation speed by trading-off the estimation precision and the more precise the model the closer the run time is to simulation.

2.4.2 QoS in the presence of bottleneck devices

Many of the existing multi-core architectures are DRAM centric. For example in Figure 2.7 is shown the Intel Single Chip Cloud Computer prototype processor [67]. The processor has 48 core which communicate to the main memory through only four DRAM controllers. Therefor the DRAM controller can become the bottleneck for system performance. A similar trend is visible in embedded applications. For example, in Figure 2.8 the latest generation SoC the OMAP 5 platform from Texas Instruments [74] is presented. The SoC is destined for mobile application and therefore it has many dedicated IP-core. However most cores that require large storage have to communicate with the main memory though two DRAM controllers.

Therefore to provide the QoS that is required at application level it is not enough to give guarantees on the network alone. As some peripherals are shared among multiple IP cores, the access time can be unpredictable. To ensure that the required QoS is provided, the access time to such shared devices has to be considered as well. One particular such device is the DRAM controller because it can be shared by many IP cores and the access time is unpredictable under high load. Many of today's *Systems on Chips (SoCs)* are DRAM centric as the applications running on them have large data storage requirements. In such systems many cores access the external DRAM memory through a single on-chip DRAM controller. Since there are many cores that have high communication demands to a single DRAM controller, the controller can become the bottleneck for the system performance. It can even affect the performance of cores that do not communicate with it directly. Several works have presented methods to improve the efficiency for accessing DRAM through transaction scheduling and reordering [142]. However reordering transactions makes the latency for accessing DRAM unpredictable and highly dependent on the address traces generated by the applications accessing the DRAM. The unpredictable nature of DRAM access can lead to transitory bottlenecks at the DRAM controller ports.

Several works have presented method to improve DRAM access efficiency by scheduling and reordering transactions in the DRAM controller [142], [6], [10], [91], [138], [168]. In [6], [91] are presented optimizations for multicore systems, and in [10] a predictable DRAM controller is presented. Memory scheduling is important in increasing the efficiency of DRAM access, but it is orthogonal to the scope of this paper. We use the simulator model from [168] in order to include these memory access optimizations

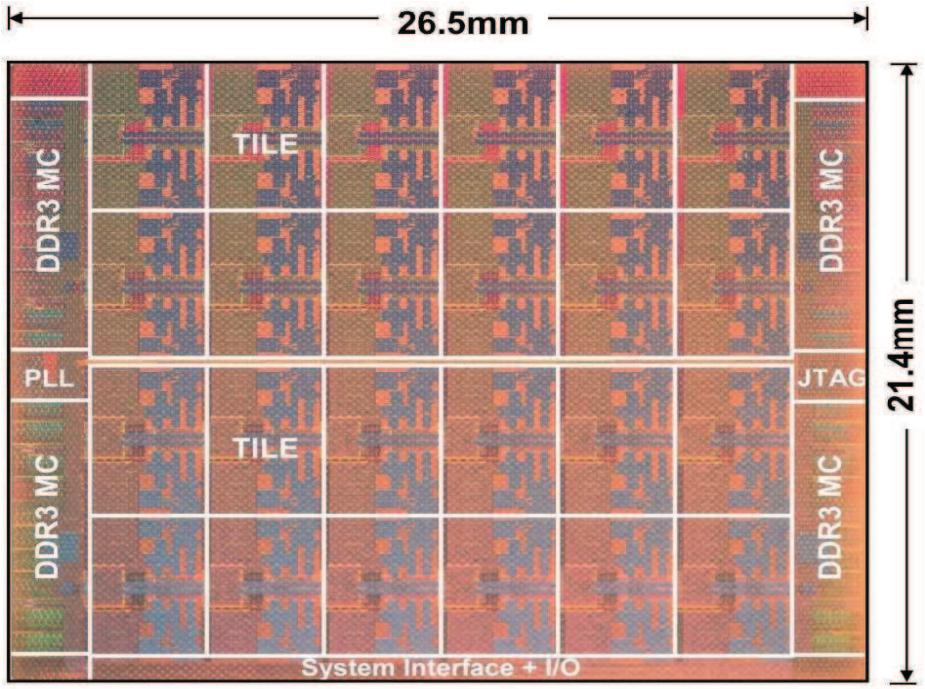

Figure 2.7: Intel Single Chip Cloud Computer [67]

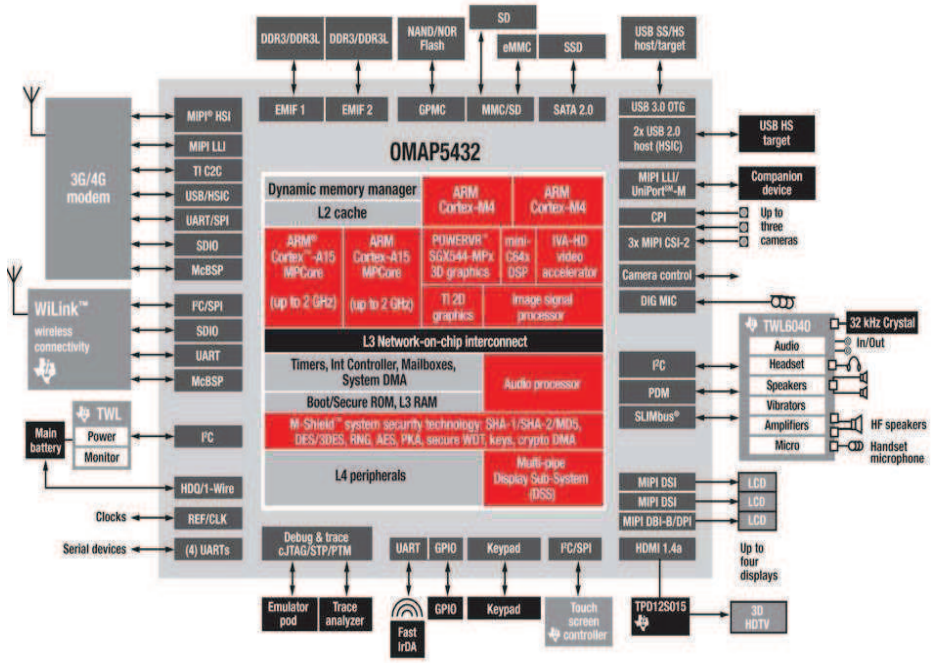

Figure 2.8: Texas Instruments OMAP 5 application platform [74]

in our simulation framework.

In [77] the authors present a way to reorder DRAM transactions while in the network and simplify the DRAM controller. However this does not solve the problem of interference with the normal traffic in the NoC. To reduce the traffic to DRAM, the authors propose in [178] to have a custom processor closely integrated with DRAM controller, in order to process complex requests and return only the results. A credit-based flow-control method is used in [167] to prevent the DRAM traffic from waiting into the network and interfering with non-DRAM traffic. In [85] the authors propose to give higher priority to transactions waiting in the DRAM queue that have to be sent back to cores in areas of the NoC which are less congested. However in custom topologies the response path is usually not congested. Memory-centric NoC architectures with real chip implementations are provided in [83] and [84]. However in these systems there are several memories which are on-chip and the communication between cores is done through these memories.

To satisfy the growing demands for memory bandwidth of current and future applications at manageable power levels, parallel access to the memory systems is required. In traditional architectures that rely on off-chip external DRAM memory, the limited number of I/O-ports prevent SoC designers from using multiple memory channels. *Three dimensional integration* is a promising technology for increasing the number of memory channels that can be accessed in parallel as well. A major advantage of 3D integration is the ability to integrate efficiently heterogeneous manufacturing technologies in a single chip stack, by putting together dies that have been processed separately. 3D-stacked memories that leverage the benefits of heterogeneous integration have been proposed as a solution to overcome the *Memory Wall* [174]. As each layer in a 3D stack is processed separately, high density DRAM memory can be stacked on top of logic to satisfy the large storage needs of applications (at low cost per MB). *Through Silicon Vias* (TSVs) are used to provide vertical connectivity and as 3D integration technology matures the density of TSVs increases. Stacked memories can benefit from the large number of TSVs to provide wide interfaces and multiple channels, such that the large bandwidth requirements can be met at low power levels. Apart from the wider data-paths that allow the memory to operate at lower frequency, while achieving the same bandwidth, power is saved in 3D stacked memories by removing the need to go off-chip through power-hungry IO ports. On the other hand, having access to multiple ports requires the SoC to be able to efficiently use them to achieve the overall bandwidth requirement.

Many research groups have focused on the technological aspects of manufacturing 3D-integrated memory systems [46], [12], [50], [175]. Others have investigated system architectures using 3D-integrated on-chip DRAM [103], [99], [100], [161]. In [174], Weis et al. present a design space exploration of SoCs with 3D-stacked DRAM and

in [11] the authors show an overview of 3D-integrated DRAMs and the challenges associated with it. In this thesis, I assume a specific 3D-stacked DRAM architecture, i.e. the WideIO JEDEC standard[87]. A study showing the advantages of using multi-channel memory systems for video encoding SoCs is presented in [7]. The study shows the impact of multiple channels on the system performance and our work is different from that as we are concerned with how to efficiently access multiple channels.

Arteris [13] and Sonics [32] offer memory interleaving support that is integrated with their interconnect solutions. However they do not provide any comparison to traditional solutions. Moreover in the white paper from Sonics [32], experiments are shown only for two channels using existing off-chip DDR3 memories and not 3D-stacked WideIO with 4 channels.

2.5 Comparison with previous work and list of publications

Many publications in literature present algorithms and tools to map tasks to IP cores interconnected by regular topologies [68], [69], [122], [123]. Murali et al. also propose in [120] a tool to map tasks considering QoS constraints as well. As SoC designs started to trade-off programmability for power efficiency and to customize the IP-cores, the NoC research works started to focus on algorithms and methods to design application specific NoC topologies [134], [65], [8], [158], [62], [181], [177], [121]. However these methods considered only 2D-integration technology and synchronous design. The work presented in Chapter 3 tries to go one step forward in the domain of application specific NoC design, and to address the challenges posed by designing the NoC for 3D-integration which is a promising new technology. As integration technology evolves new challenges arise like multi-synchronous designs and increased leakage power consumption. The existing works on application specific NoC design do not address these new technological issues. Therefore in Chapter 4, I look at methods to design the NoC facing these challenges. There has been much work on avoiding deadlocks, especialy for general networks and regular topologies [43], [44], [34], [38], [98], [55], [105]. Most works on application specific NoC design are relying on the methods from [56], [179], [159] to avoid deadlocks during the synthesis of the NoC topology. However these methods may interfere with the additional constraints that are imposed by the new integration technologies (3D-integration, voltage island design) prohibiting the synthesis tools to produce solutions when the constraints are tight. Therefore in Chapter 5, I present a new method to remove deadlocks in application specific NoC after the topology is build.

Providing Quality-of-Service has also been an important concern for researchers. Much of the research has focused though on predictable Noc architectures [57], [153], [131], [107], [29], [28], [30], [54], [80], [93], [108], [111], [116], [118], [137], [141]. How-

ever many application require predictably only for few services and average performance for the rest making the use of specialize hardware expensive in many cases. Models for estimating worst-case latencies have also been investigated [92], [139], however most of the existing synthesis algorithms are using only simple analytical models that do not take into account contention. In Chapter 6, I present a new synthesis algorithm that integrates the method from [139] to estimate the worst-case latencies and to construct the NoC topology as to meet worst-case latency requirements only with best-effort NoC hardware.

Providing QoS at system level requires to go beyond the interconnect and to improve the performance of the bottleneck-peripheral devices. One such bottleneck device is the shared DRAM controller. Much research is available in literature for improving the performance DRAM controller [142], [6], [10], [91], [138], [168]. There is also work on the technological aspects, to bringing the memory closer to the SoC, by means of 3D-integration [46], [12], [50], [175]. In my work, in Chapter 8, I analyze ways to improve the architecture of the NoC to provide better services in order to access the DRAM controller efficiently, considering WideIO 3D-integrated memory.

Most topics addressed by the thesis have also been published at conferences or in journals. The methods and algorithms for performing NoC topology synthesis for 3D-ICs, from Chapter 3, have been published in [146] and [149]. The method to perform NoC design in ICs with multiple voltage island and shutdown of voltage island, from Chapter 4 has been published in [145]. The extension of the synthesis algorithm for 3D-SoC with voltage island and the experimental comparisons between 2D- and 3D-ICs with voltage island have been published in [147]. The algorithm for removing deadlocks from Chapter 5 has been published in [148]. The design methods for constructing NoCs that meet worst-case latency constraints using best effort NoC hardware has been accepted to be published in [151]. The first part of Chapter 8 that provides architectures and analyzes ways of prevent regular NoC traffic to be affected by DRAM traffic, has been published in [150].

2.6 Summary

In this chapter I have presented some of the basic concepts that are going to be used throughout the rest of the thesis. I have introduced the general concepts describing NoCs and their functionality, 3D integration technology, multi-synchronous design and the problems related to QoS. Within each topic I have also reviewed the main publications that have contributed to some of the aspects that are relevant for this thesis.

3 Designing application specific NoCs for heterogeneous 3D-SoCs

Two-dimensional chip fabrication technology is facing lot of challenges in utilizing the exponentially-growing number of transistors on a chip. Wire delay and power consumption are dramatically increasing and achieving interconnect design closure is increasingly a challenge. Moreover, diverse components that are digital, analog, MEMS and RF are being integrated on the same chip, resulting in high complexity for the 2D manufacturing process [18]. *Three dimensional integration* is emerging as an attractive solution to continue the pace of growth of Systems on Chips (SoCs). An overview of 3D integration technology is given in Section 2.2.

As the number of IP cores will grow in 3D-SoCs, NoCs will be a necessity for 3D chips. They provide arbitrary scalability of the interconnects across additional layers, efficiently parallelize communication in each layer and help controlling the number of vertical wires (and hence TSVs) needed for inter-layer communication. The combined use of 3D integration technologies and NoCs introduces new opportunities and challenges for designers. Building power-efficient NoCs for 3D systems that satisfy the performance requirements of applications, while satisfying the technology constraints, is an important problem. To address this issue, new architectures and design methods are needed.

In this chapter, I address this important problem and present a synthesis approach for designing power efficient NoCs for 3D-SoCs that meet application performance and 3D technology constraints. Custom topologies that are tailored to meet the application performance constraints can result in large NoC power savings. The need for an application-specific topology has been well studied for 2D-ICs [123], [121]. All these advantages hold in 3D as well. The major contributions of this chapter are:

- A synthesis approach to determine the most power efficient topology for the application and for finding paths for the traffic flows that meet the TSV constraints.

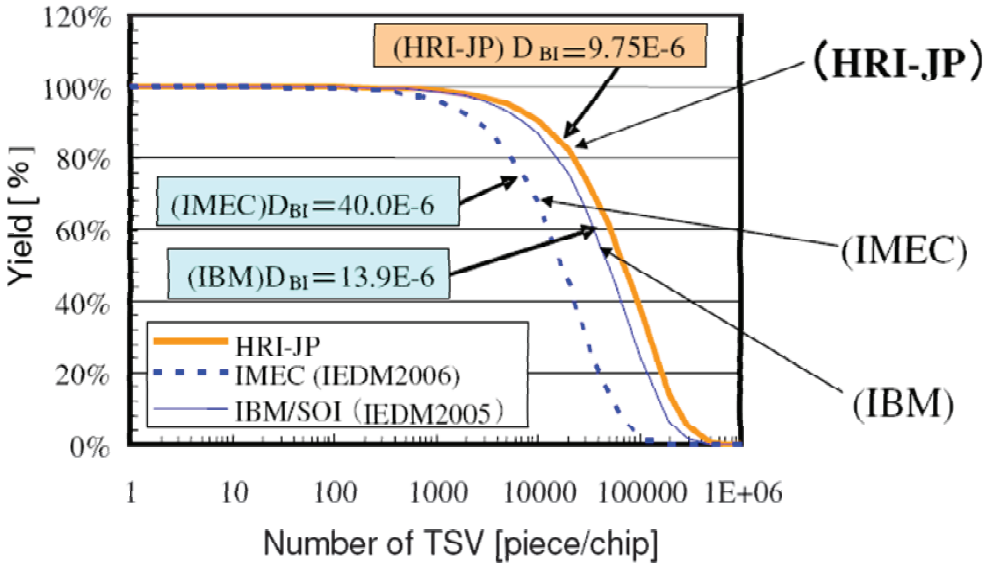

Figure 3.1: Yield vs. TSV count [113]

- Methods to assign the NoC components to the layers of the 3D-IC.
- A method to determine the optimal positions of switches in the floorplan in each layer.
- A method to insert the NoC components in the floorplan and to remove any overlap with the cores.
- The exploration of the design space of custom topologies in order to show the comparative advantages in moving to 3D technology.

Constraining the number of TSVs is important in order to control the yield of the 3D-IC. In Figure 3.1, I show the dependence of yield on the number of TSVs for different manufacturing processes [113]. For all processes there is a clear upper bound on the number of TSVs after which the yield decreases rapidly. For this reason, it is important for the synthesis process to be aware of this upper bound and should be able to ensure that the designed NoC meets this constraint, while at the same time complying with the application requirements. Also, since for the different manufacturing processes the constraint is different, it is necessary for the synthesis process to take this as an input. Another reason to try to reduce the number of TSVs is to save area. In [26], the pitch of a TSV is reported to be between 3 and 5 µm. Reserving area for too many TSVs can cause a considerable reduction in the active silicon area remaining for transistors.

The assignment of cores to the different layers and the floorplan of each layer needs to

consider several performance and technological constraints. For example cores that have I/O pins that go off chip have to be placed near the border of the die, cores that operate at the same frequency should be placed close together to share the clock tree and thermal issues should be considered as well. There are several works that address these issues [36], [70], [71], [96], [180] and this work is complementary to them. The experiments show that wires have significant power consumption and delay. Thus, the floorplan of the design should be considered during the topology synthesis process. Here, I only address the issue of designing the NoC topology and determining the placement of the NoC components. I show that using a standard floorplanner to insert the NoC components in an existing floorplan can lead to poor results. Therefore I propose a simple floorplanning method that shows a significant reduction in area (average 20%) and power consumption (average 7.5%) when compared to a constrained standard floorplanner. More complex optimization methods, like the ones from [97] could also be used to design an even better custom floorplanning routine. As the main objective is topology synthesis, developing and comparing different custom floorplanning methods is beyond the scope of my work. Another point to be noted is that a single switch or interface of a NoC has low area (few thousand gates) and power consumption (few mW at 1 GHz) overhead. Thus, the thermal properties of the system are not affected significantly when inserting the NoC components in the floorplan.

Another major contribution is a comparison between 2D and 3D SoCs in terms of the interconnect delay and power consumption. An important advantage in using 3D technology is that the wires are shorter. However, today it is still unclear the amount of power and delay gains that is achievable by 3D integration for custom interconnects for SoCs. I compare the same SoC design for the case when all cores are on the same die to the case when the cores are distributed on different dies in a 3D stack. Therefore I analyze the power reduction that is due to having shorter wires. I do not consider the case when the initial system was build using different dies in multiple packages and the power reduction is due to removing the IO pads and drivers. In this chapter, for comparative purposes, I also apply a 2D synthesis flow developed earlier by Murali et. al [121] for a corresponding 2D implementation of the benchmarks. The results show that a 3D design can significantly reduce the interconnect power consumption (38% on average) and latency (13% on average). For completeness, I also show the power consumption reduction in using a custom topology when compared to a regular topology for several benchmarks. However, detailed study of the different advantages of a custom solution is not reported here, as the analysis would be analogous to those done in [123], [121] for 2D ICs.

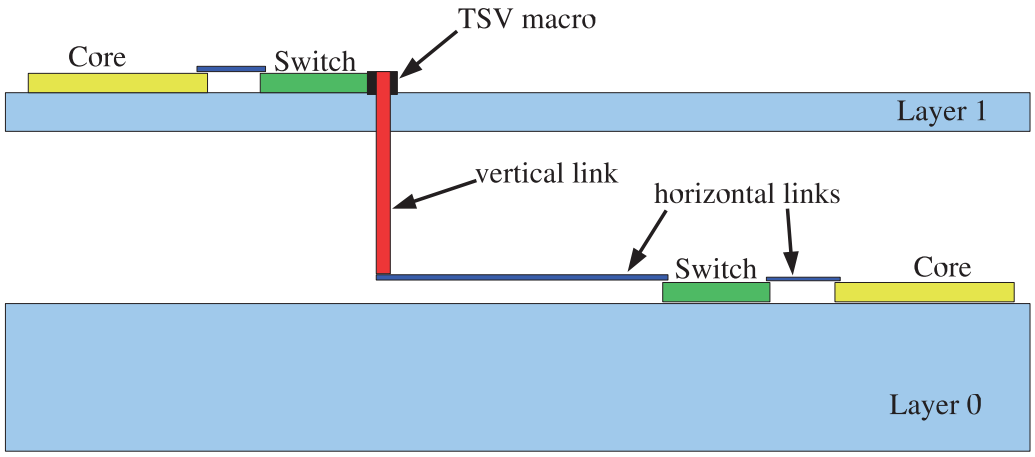

Figure 3.2: Example vertical link

3.1 Architecture

I assume a wafer-to-wafer manufacturing process, with TSVs used for the vertical interconnection wires. I present the synthesis methods for the case where each component (core and network component) is designed to span within a single layer. As the network components are small, there will not be a significant performance benefit if they were designed to span on multiple layers. In designs where a single core may span multiple layers, the network interface would still be assigned to a single layer close to where the communication port of the core is. In this case the synthesis algorithm is not affected at all and only minor extensions would be needed in the floorplanning routine to account for this fact. As this does not affect the synthesis methods, I do not present such extensions in this work. In the considered architecture, the vertical wires using a TSV between two layers uses the global metal routing of the bottom layer, therefore requiring silicon area to be reserved only on the top layer where the TSVs are drilled. Area reservation is done by placing abstractions in the floorplan called TSV macros. In Figure 3.2, I show an example where two switches on two different layers are connected using an inter-layer link.

For the inter-layer links that go through more than one layer, TSV macros are placed in the intermediate layers as well. In Figure 3.3, I show an example where there is a TSV macro in the middle layer. From the bottom layer, the link is first routed horizontally on the metal layer and then vertically. In the second layer, an intermediate TSV macro is needed to connect through the silicon. Then, the link is routed again on the metal layers in the second layer, and when aligned with the switch on the top layer, the link is fed vertically. The switch in the top layer has a TSV macro embedded for the port that is connected to this link. The area of the TSV macros for a particular link width is

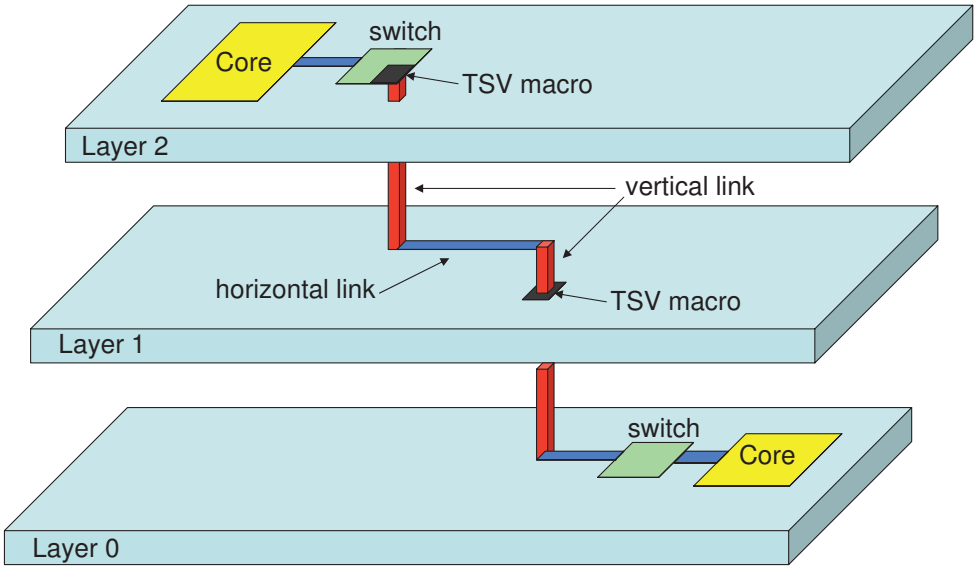

Figure 3.3: Example vertical link

taken as input. For the synthesized topologies, my tool automatically places the TSV macros in the intermediate layers and on the corresponding switch ports.

If there is a core that is connected to a switch that is in another layer, then space has to be reserved on the floorplan to place the TSVs. The network interface of the core will contain the necessary TSV macro, that reserves the space, when core and the switch are only one layer apart. The network interface is responsible to translate the core communication protocol to the network protocol. In the case when the core and the switch are several layers apart then the TSV macros, to reserve the area to place the TSVs, have to be explicitly placed on the floorplan for the intermediate layers. The explicit TSV macro is the same as in the example of the switch to switch link from Figure 3.3 in the middle layer. Active silicon area is lost every time a TSV macro is placed as the area reserved by the macro will be used to construct the TSV. In some designs redundant TSVs are used to increase reliability [104]. Adding redundant TSVs can be considered by reserving more area with the TSV macros and it is transparent for my tool. The TSV macros are placed automatically by my tool.

Even though I assumed in this chapter a wafer-to-wafer manufacturing process, also other processes can be used like die-to-wafer. The type of process weather is face-to-

face bonding or face-to-back bonding does not affect the NoC synthesis algorithm. To abstract the details of the manufacturing process, the maximum number of vertical wires using TSVs is taken as input. Models of power and latency of the TSVs are also taken as input.

Apart from TSV based 3D-integration other processes like 3D-monolithic integration can be used. In case of 3D-monolithic integration the synthesis method, that I present, requires no change. From the point of view of the NoC synthesis the important constraint the 3D-integration brings is the limited number of vertical interconnect wires which is captured by the synthesis approach. In case of 3D-monolithic integration the number of possible vertical links is higher than in TSV based integration therefore when designing the NoC in 3D-monolithic technology the designer only need to specify a looser constraint on the number of vertical links that can be used by the algorithm. Currently 3D-monolithic integration is limited to two layers (potentially as the technology matures the number of layers could be increased) so there could be hybrid approaches where 3D-monolithic integration is used to speed-up the logic, while 3D TSV-based approach is used to stack up multiple logic layers. For such a hybrid technology the synthesis algorithm would require some changes: i) have different bound on the number of vertical links that can be used between the layers of the 3D-monolithic integration and between the layers that are connected with TSVs; ii) connectivity between stacked layers of the 3D-monolithic process may have to be constrained to go through switches placed on the main silicon layer of the 3D-monolithic stack, which are connected with TSV links.

3.2 Design approach

In this section I define the inputs and outputs of the design flow. In the *core specification* file, the name of the different cores, the sizes and positions are given as inputs. The assignment of the cores to the different layers in 3D is also specified as input in the file. In the *communication specification* file, the communication characteristics of the application are specified. This includes the bandwidth of communication across different cores, latency constraints and message type (request/response) of the different traffic flows.

The technology used for 3D integration can result in two main constraints: first, to achieve high yield, the number of TSVs that can be established across two layers may need to be restricted below a threshold. Second, some 3D technologies can allow TSVs only across adjacent layers. In the rest of the paper, I model the maximum TSV number constraint by using a constraint on the number of NoC links that can cross two adjacent layers, denoted max_ill (for maximum number of inter-layer links). For a particular link width, the maximum number of links can be directly determined

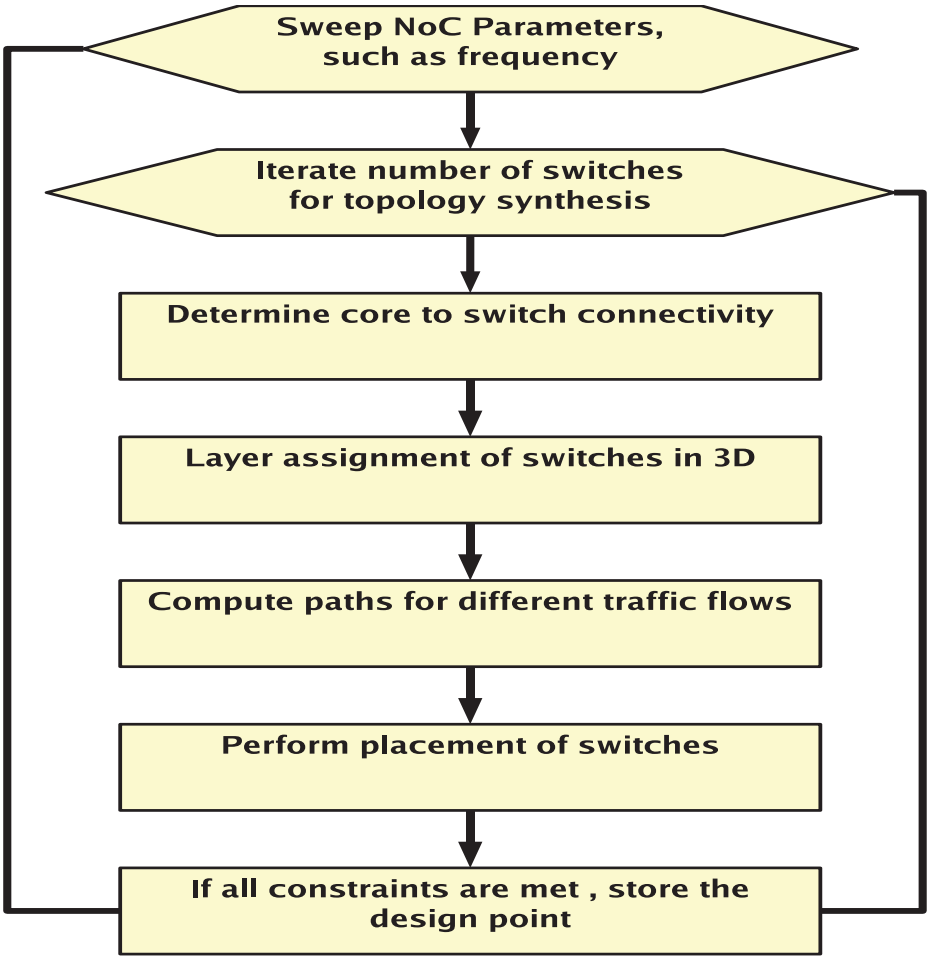

Figure 3.4: Algorithm steps

from the TSV constraints.

The objective of the topology synthesis procedure can be set by the designer to minimize NoC power consumption or latency or a combination of both.

For the synthesis procedure, the power, area and timing models of the NoC switches and links are also taken as inputs. For the experimental validation of this work, without loss of generality, I use the library of network components from [160] and the models are obtained from layout level implementations of the library components. Any other NoC library can also be used with the synthesis process. I also take the power consumption and latency values of the vertical interconnects as inputs. For this, I use the models from [72].

The output of the topology synthesis procedure is a set of trade-off points of topologies that meet the constraints, with different values of power, latency and design area. From the resulting points, the designer can choose the optimal point for the application. The synthesis procedure also produces a placement of the switches in the 3D layers and the positions of the switches.

The different steps of the synthesis algorithm are presented in Figure 3.4. As the topology synthesis and mapping problems are NP-Hard [134], I present efficient heuristics to synthesize the best topology for the design. The NoC architectural parameters, such as frequency of operation, are varied and the topology design process is repeated for each architectural point. In the following step, the number of switches needed to connect the cores is varied and different topologies are synthesized. There are some general trends that I observed during the topology design process: when the number of switches is increased, though the switches become smaller, packets need to traverse more hops and the total switch power usually increases. With more switches, the switch that is connected to a core is closer in the floorplan, thereby leading to lower core-to-switch link power consumption. However, there are also more switch-to-switch links, thereby leading to an increase in the power consumption of switch-to-switch links. In order to choose the most power-efficient topology, the combined effect of all these three trends needs to be considered. Thus, it is needed to explore designs with different number of switches, starting from one where all the cores are connected to a single switch to a design point where each core is connected to a separate switch.

For a particular switch count, in the next steps, I determine the connectivity between the switches and the cores and the 3D layer assignment of the switches. During this step, there is a degree of freedom that needs to be explored: a core in a layer of 3D design can be restricted to be connected to a switch in the same layer or could be allowed to connect to a switch in any layer. When a core is restricted to be connected to a switch in the same layer, then the traffic flowing from it to a core in another layer needs to traverse at least two switches (one in the current layer and another in the other layer), thereby increasing latency. On the other hand, if a core is allowed to be directly connected to a switch in any layer, then more inter-layer links may be needed. It is important to choose this restriction based on application characteristics. Another degree of freedom that needs to be explored is from a technology standpoint: the technology could allow vertical link across many layers (for example, a link from layer 1 to layer 3) or could allow connection only across adjacent layers.

To address these two issues, I present a two-phase method to determine the core to switch connectivity. In the first-phase (presented in Section 3.3.1), cores are allowed to be connected directly to switches in any layer. If the resulting designs do not meet the maximum inter-layer link constraints, then in the second phase (presented in

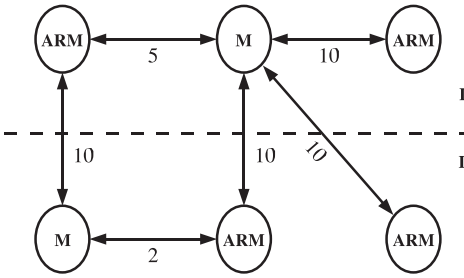

Figure 3.5: Communication graph with bandwidth demands on the edges

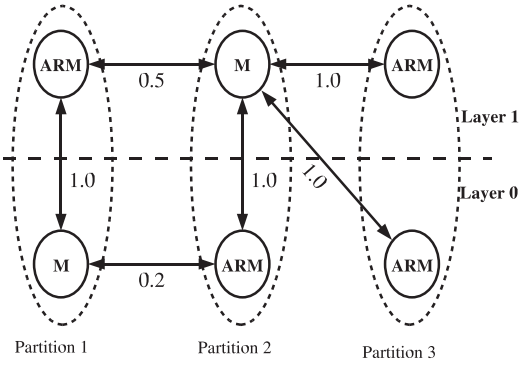

Figure 3.6: PG and the min-cut partitions

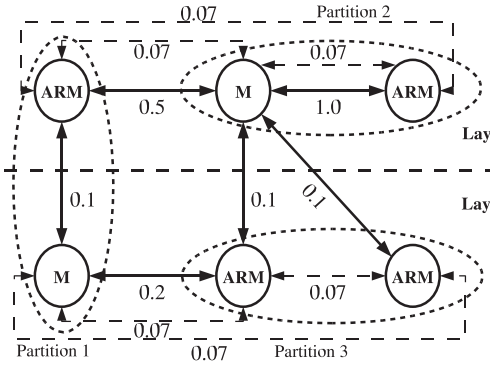

Figure 3.7: SPG and the min-cut partitions

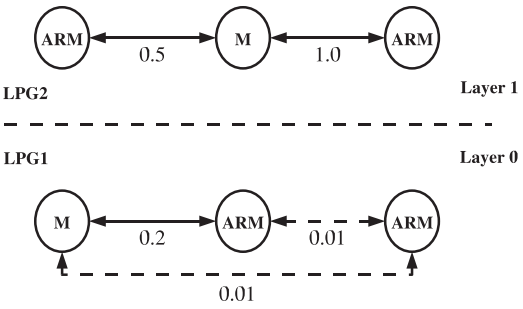

Figure 3.8: LPGs for two layers

Section 3.3.2), cores are restricted to be connected to only switches in the same layer. Also, in the second phase, vertical links are established only across adjacent layers in 3D. Thus, for systems where the underlying technology supports vertical links only across adjacent layers, the first phase can be skipped and the second phase can be used directly. Please note that Phase 2 can also be used when the objective is to reduce the number of inter-layer vertical links used.

In the next step, the paths for the inter-switch traffic flows is determined, and is explained in Section 3.4. Then, the optimal positions of the switches are determined and the switches are placed in each layer, minimally changing the input floorplan. In the last step, if the current topology meets the constraints, the design point is saved.

3.3 Methods to establish core to switch connectivity

In this section, I present methods for establishing connectivity between the cores and switches.

Definition 3.1 *Let n be the number of cores in the design. The x and y coordinate positions of a core c_i are represented by xc_i and yc_i respectively, $\forall i \in 1 \cdots n$. The 3D layer to which the core i is assigned is represented by $layer_i$.*

From the *communication specification* file, the communication characteristics of the application are obtained and represented by a graph [69], [123], [120], defined as follows:

Definition 3.2 *The communication graph is a directed graph, $G(V,E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j) representing the communication between the cores v_i and v_j. The bandwidth of traffic flow from vertex v_i to v_j is represented by $bw_{i,j}$ and the latency constraint for the flow is represented by $lat_{i,j}$.*

I define a *Partitioning Graph (PG)* as follows:

Definition 3.3 *The partitioning graph is a directed graph, $PG(U,H,\alpha)$, that has same set of vertices and edges as the communication graph. The weight of the edge (u_i, u_j), defined by $h_{i,j}$, is set to a combination of the bandwidth and the latency constraints of the traffic flow from core u_i to u_j: $h_{i,j} = \alpha \times bw_{i,j}/max\_bw + (1-\alpha) \times min\_lat/lat_{i,j}$, where max_bw is the maximum bandwidth value over all flows, min_lat is the tightest latency constraint over all flows and α is a weight parameter.*

The parameter α can be set by the designer based on the application characteristics or swept by the tool over a range of values, in order to meet the latency constraints. We use a two phased method. The first phase is more general, but requires more TSV connections. If the no topology can be built using the algorithm of phase 1, the tool switches to phase 2 where a more restrictive algorithm is used. The two phases are described next:

3.3.1 Phase 1

As the number of switches is varied in order to explore different design points, most of the times there will be fewer switches in the design than the number of cores.

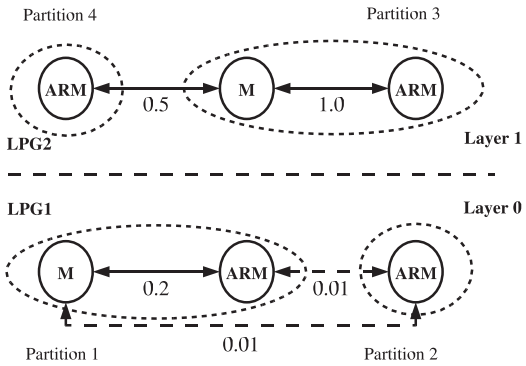

Figure 3.9: Two min-cut partitions of LPGs

Therefore multiple cores will be assigned to the same switch in most cases. The cores will be partitioned in as many blocks as there are switches and the cores in the same block will be assigned to the same switch. When using Phase 1 cores that have high communication or tight latency will be assigned to the same switch regardless of the fact that cores may be on different layers. If for a particular core to switch assignment the algorithm using Phase 1 will not be able to meet the inter-layer link constraint then the influence of bandwidth and latency of the inter-layer flows will be scaled down in steps and new partitions will be generated. The exact steps of Phase 1 are described in the following paragraphs.

In the first step of Algorithm 3.1, the partitioning graph is constructed. Then (in Step 3), the number of switches in the design is varied from 1 to the number of cores in the design. In the next step (step 5), for the current switch count, that many min-cut blocks of PG are obtained. All the cores in a block are connected to the same switch and the partitioning is done such that each block has about equal number of cores. Thus, those traffic flows with large bandwidth requirements or tight latency constraints are assigned to the same block and traverse a single hop in the network.

Example 3.1 *For the communication graph from Figure 3.5, an example partitioning graph is shown in Figure 3.6. The cores are assigned to the two layers such that highly communicating cores are placed one above the other, which is an input to the synthesis algorithm. Here, I assume* $\alpha = 1$ *and the bandwidth of the traffic flowing between cores with in a layer is lower than the traffic between the cores across the layers as denoted by the values on the edges of the communication graphs. The weights of the PG are calculated with the formula from definition of the PG graph. In the figure, I also show an example of 3 min-cut partitions of the graph. The partitioning leads to cores in different layers being assigned to the same block.*

Then (in step 7), the layer assignment of each switch is computed as an average of

Algorithm 3.1 Core-to-switch connectivity

1: Build partitioning graph PG(U,H,α)
2: $Unmet = \phi$.
3: {Vary number of direct switches in a range}
4: **for** i = 1 to $|U|$ **do**
5: Perform i min-cut partitions of PG. Let the set $Partition_j$ be set of vertices in j th partition, $\forall j \in 1 \cdots i$.
6: {Compute layer assignment for each switch:}
7: $layer\_sw_j = \frac{\sum_{\forall k \in partition_j} layer\_k}{|partition_j|}$
8: Compute paths for inter-switch flows
9: If path computation failed, add i to set $Unmet$.
10: **end for**
11: $\theta = \theta_{min}$
12: **while** (($Unmet != \phi$) & ($\theta \leq \theta_{max}$)) **do**
13: **for** Each $i \in Unmet$ **do**
14: Build scaled partitioning graph, SPG(W,L,θ)
15: PG = SPG
16: Repeat steps 5 to 8
17: If valid paths found, remove i from set $Unmet$.
18: **end for**
19: $\theta = \theta + \theta_{scale}$
20: **end while**

the layers of the cores to which the switch is connected. Alternatively, the switch could also be assigned to the layer containing the most number of cores connected to it. At this point, the intra-partition traffic flows are taken care of and I need to establish connectivity across the switches for the inter-switch traffic flows. This step is explained in the next section. Then (in step 9), the resulting designs are evaluated to see whether they meet the *max_ill* constraint and the switch counts that do not meet the constraint are stored in the set *unmet*.

In order to facilitate meeting the *max_ill* constraint for the design points in the set *unmet*, I use the *Scaled Partitioning Graph*, defined as follows:

Definition 3.4 *A scaled partitioning graph with a scaling parameter θ, SPG(W, L, θ), is a directed graph that has the same set of vertices as PG. A directed edge $l_{i,j}$ exists between vertices i and j, if $\exists(u_i, u_j) \in P$ or $layer_i = layer_j$.*

That is, in the SPG, along with the edges in PG, I define new edges between all cores in the same layer of 3D. I also reduce the edge weights of inter-layer flows, depending on the scaling parameter θ. If this scaled graph is used for partitioning, then more cores in the same layer will be in a partition, thereby reducing the inter-layer links,

at the expense of increasing the power consumption and latency of inter-layer flows. To obtain designs with lower inter-layer links, the parameter θ is varied from θ_{min} to θ_{max} in steps of θ_{scale} in the algorithm (steps 12 to 19), until the max_ill constraint is met. After several experimental runs, I determined that varying θ from 1 to 15 in steps of 3 gives good results.

In order to cluster cores in a layer that actually communicate, I also need to ensure that the newly added edges have a lower edge weight than the original intra-layer edges. Please note that if the new edges are not added, the partitioner may still cluster cores across layers, which will not lead to a reduction in the inter-layer links.

I denote the maximum edge weight in PG by max_wt. I formally define the edge weights in SPG as follows:

$$l_{i,j} = \begin{cases} h_{i,j} & \text{, if } (u_i, u_j) \in PG \,\&\, layer_i = layer_j \\ \frac{h_{i,j}}{\theta \times |layer_i - layer_j|} & \text{, if } (u_i, u_j) \in PG \,\&\, layer_i \neq layer_j \\ \frac{\theta \times \text{max\_wt}}{10 \times \theta_{max}} & \text{, if } (u_i, u_j) \notin PG \,\&\, layer_i = layer_j \\ 0 & \text{, otherwise} \end{cases} \tag{3.1}$$

From the definition, I can see that the newly added edges have at most one-tenth the maximum edge weight of any edge in PG, which was obtained experimentally after trying several values.

Example 3.2 *The SPG for $\theta = 10$ for the PG from Example 3.1 is presented in Figure 3.7. In the SPG, the inter-layer links have lower weights as they were scaled down by θ and new edges are added between cores with in the same layer. The weights of the extra edges are calculated using the equation 3.1. The 3 min-cut blocks are now different, with more cores in the same layer belonging to the same block.*

The time complexity of the Algorithm 3.1 is $O(|V|^3|E|ln(|V|))$, where $|V|^2 ln(|V|)$ corresponds to finding paths, E is the maximum number of flows. Also, the number of switches is varied from 1 to the maximum number of cores $|V|$ and that a topology is built for each switch count.

3.3.2 Phase 2

In Phase 2, I restrict cores to be connected to switches on the same layer. Also switches can only connect to switches on the same layer or on adjacent layers. For each layer a certain number of switches will be assigned. Then on each layer the cores are

partitioned and assigned to the switches on that layer considering the bandwidth and latency of the flows between the cores on that layer. Information of the inter-layer flows is ignored when the cores are assigned to switches when using Phase 2. Because of these restrictions Phase 2 can be used when a tight inter-layer link restriction is in place or when the 3D integration technologies forbids links to go across more than two layers. In this section a detailed description of Phase 2 is given.

I define the *Local Partitioning Graph* for each layer:

Definition 3.5 *A local partitioning graph, LPG(*Z*,* M*,* ly*), is a directed graph, with the set of vertices represented by* Z *and edges by* M*. Each vertex represents a core in the layer* ly*. An edge connecting two vertices is similar to the edge connecting the corresponding cores in the communication graph. The weight of the edge* (m_i, m_j)*, defined by* $h_{i,j}$*, is set to a combination of the bandwidth and the latency constraints of the traffic flow from core* m_i *to* m_j*:* $h_{i,j} = \alpha \times bw_{i,j}/max\_bw + (1-\alpha) \times min\_lat/lat_{i,j}$*, where* max_bw *is the maximum bandwidth value over all flows,* min_lat *is the tightest latency constraint over all flows and* α *is a weight parameter. For cores that do not communicate with any other core in the same layer, edges with low weight (close to 0) are added between the corresponding vertices to all other vertices in the layer. This will allow the partitioning process to still consider such isolated vertices.*

Example 3.3 *The LPGs for the two layers of the communication graph from Figure 3.5 are shown in Figure 3.8. Since the LPGs are built layer by layer, the graphs for the two layers are independent of one another. The weights of the remaining edges were calculate with the formula from the definition of the LPG graph for a value of* $\alpha = 1$*. Extra edges with low weights are added (dotted edges in the figure) from the vertices that have no connections to the other vertices of the LPG.*

The pseudo-code for establishing core to switch connectivity is presented in Algorithm 3.2. As the number of input/output ports of a switch increases, the maximum frequency of operation that can be supported by it reduces, as the combinational path inside the crossbar and arbiter increases with size. In the first step of the algorithm, for the required operating frequency of the NoC, the maximum size of the switch (denoted by max_sw_size) that can support that frequency is obtained as an input. Based on this and the number of cores in each layer, in the next steps (2-4), I determine the minimum number of switches needed in each layer. Then the local partitioning graph for each layer is constructed.

Then, the number of switches in each layer is incremented (starting from the initial count calculated in steps 2-4) every iteration, until it equals the number of cores in the layer. The term $|LPG(Z, M, j)|$ represents the number of cores in layer j. For each

Algorithm 3.2 Core-to-switch connectivity

1: Obtain maximum switch size max_sw_size for current frequency
2: **for** each layer $j \in 1 \cdots lr$ **do**
3: $ni_j = \lceil$ number of cores in $layer_j / max\_sw\_size \rceil$
4: **end for**
5: Build $LPG(Z, M, j)$ for each layer j.
6: **for** $i = 0$ to $\max_{\forall j \in 1 \cdots lr}\{|LPG(Z, M, j)| - ni_j\}$ **do**
7: **for** each layer $j \in 1 \cdots lr$ **do**
8: **if** $ni_j + i \leq |LPG(Z, M, j)|$ **then**
9: $np = ni_j + i$
10: **else**
11: $np = |LPG(Z, M, j)|$
12: **end if**
13: Obtain np min-cut partitions of LPG(Y,M,j)
14: **end for**
15: Compute paths for inter-switch flows (Section 3.4).
16: If valid paths found, save the current design point
17: **end for**

switch count, that many min-cut partitions of the LPG of the layer are obtained (step 13). The cores in the same partition are connected to the same switch.

Example 3.4 *Two min-cut blocks of the LPGs of Figure 3.8 are shown in Figure 3.9.*

The time complexity of the Algorithm 3.2 is similar to the complexity of the previous algorithm $O(|Z_{max}||V|^2|E|ln(|V|))$, where Z_{max} corresponds to the maximum number of cores in a layer, which decides the number of topologies to be explored.

3.3.3 Pruning the search space

To reduce the number of explored design points, I use several methods to prune the search space: (i) As the number of input/output ports of a switch increases, the maximum frequency of operation that can be supported by it reduces, as the combinational path inside the crossbar and arbiter increases with size. For a required operating frequency of the NoC, I first determine the maximum size of the switch (denoted by max_sw_size) that can support that frequency and determine the minimum number of switches needed. (ii) Considering all the possible combinations of the number of switches in each layer would increase the search space too much. Therefore at each iteration of Algorithm 3.2 I increment the number of switches in each layer by one. I initialize the number of switches layer by layer as explained in the previous section. Thus, the starting design point can have different number of switches in each layer and the number of switches in each layer is proportional to the number of

cores in that layer. (iii) For a particular switch count, after partitioning, I evaluate the inter-layer links used to connect the cores to the switches, before finding the paths. If the topology requires more inter-layer links than the threshold, I directly ignore the design point.

3.4 Path computation

The procedure to establish physical links and paths for traffic flows is based on the power consumption increase and latency in using the link. This cost computation in the 3D case is similar to the 2D case, such as those presented in [62], [121], but it needs to account for the max_ill and max_switch_size constraints. Here, I do not show the entire path computation algorithm, but only present the steps needed to meet these constraints. In [62], [121], the authors present methods to remove both routing and message-dependent deadlocks when computing the paths. I also use the methods to obtain paths that are free of deadlocks.

Definition 3.6 *Let nsw be the total number of switches used across all the layers and let $layer_i$ be the layer in which switch i is present. Let $ill(i, j)$ be the number of vertical links established between layers i and j. Let the $switch\_size\_inp_i$ and $switch\_size\_out_i$ be the number of input and output ports of switch i. Let $cost_{i,j}$ be the cost of establishing a physical link between switches i and j.*

In Algorithm 3.3, I show the use of hard and soft thresholds when evaluating the cost of establishing a physical link between switches i and j. In steps 3, 4, I assign a cost of INF for establishing a link across switches in non adjacent layers and for switches in layers that have reached the maximum vertical link (max_ill) threshold. To ensure meeting the maximum link constraint, I assign a very high cost (denoted by $SOFT\_INF$) for establishing links between switches that are in layers having vertical links close to the max_ill value, denoted by $soft\_max\_ill$ (steps 5, 6). From experiments, I found that a reasonable value for $SOFT\_INF$ to be 10 times the maximum cost of any flow and $soft\_max\_ill$ to be few (2 to 3) links less than max_ill value. I use a similar technique to meet the maximum switch size constraints (steps 10-12). By using these softer constraints first, I facilitate the path computation procedure to determine valid paths when compared to only using the hard constraints.

When paths are computed, if it is not feasible to meet the max_switch_size constraints, I introduce new switches in the topology that are used to connect the other switches together. These indirect switches help in reducing the number of ports needed in the direct switches.

Algorithm 3.3 CHECK_CONSTRAINTS(i,j)

```
1: for i = 1 to nsw do
2:   for j = 1 to nsw do
3:     if |layer_i − layer_j| ≥ 2 or ill(layer_i, layer_j) ≥ max_ill then
4:       cost_{i,j} = INF
5:     else if |layer_i − layer_j| = 1 and ill(layer_i, layer_j) ≥ soft_max_ill then
6:       cost_{ij} = SOFT_INF
7:     else if switch_size_inp_i + 1 ≥ max_switch_size or switch_size_out_j + 1 ≥
       max_switch_size then
8:       cost_{i,j} = INF
9:     else if switch_size_inp_i + 1 ≥ soft_max_switch_size or switch_size_out_j + 1 ≥
       soft_max_switch_size then
10:      cost_{i,j} = SOFT_INF
11:    end if
12:  end for
13: end for
```

3.5 Switch Position Computation

Once a topology for a particular switch count is obtained, the next step is to find the latency and power consumption on the wires. In order to do this, based on the input positions of the cores, the optimal position of the switches needs to be determined. For this, I model the problem as a Linear Program (LP) [31].

Let us consider a topology with nsw switches. I denote the co-ordinates of a switch i by (xs_i, ys_i), $\forall i \in 1 \cdots nsw$. The goal of the LP is to determine the values of xs_i and ys_i, for all switches in the particular topology. The sum of the Manhattan distances between a switch i and a core k is given by:

$$coredist_{i,k} = \begin{cases} |xs_i - xc_k| + |ys_i - yc_k| & \text{, if } switch_i \text{ connected to } core_k \\ 0 & \text{, otherwise} \end{cases} \tag{3.2}$$

The sum of the Manhattan distances between a switch i and switch j to which it is connected to is given by:

$$swdist_{i,j} = \begin{cases} |xs_i - xs_j| + |ys_i - ys_j| & \text{, if } switch_i \text{ connected to } switch_j \\ 0 & \text{, otherwise} \end{cases} \tag{3.3}$$

The above equations can be easily represented as a set of linear equations [31]. Let $bw\_sw2core_{i,k}$ and $bw\_sw2sw_{i,j}$ be the total bandwidth of traffic flows between switch i and core k and switches i and j, respectively. To minimize the total power consumption of the links, I need to minimize the length of the links weighted by their

bandwidth values, so that higher bandwidth links are shorter than lower bandwidth ones. Formulating the objective function mathematically, I get:

$$obj = \sum_{\forall i}\sum_{\forall k} coredist_{i,k} * bw\_sw2core_{i,k} + \sum_{\forall i}\sum_{\forall j} swdist_{i,j} * bw\_sw2sw_{i,j} \tag{3.4}$$

The LP for optimization is written as follows:

$$\begin{aligned}
&\text{minimize} && \textstyle\sum_{\forall i}\sum_{\forall k} coredist_{i,k} * bw_sw2core_{i,k} + \\
& && + \textstyle\sum_{\forall i}\sum_{\forall j} swdist_{i,j} * bw_sw2sw_{i,j} \\
&\text{subject to} \\
& && coredist_{i,k} = \begin{cases} |xs_i - xc_k| + |ys_i - yc_k| & \text{, if } switch_i \text{ connected to } core_k \\ 0 & \text{, otherwise} \end{cases} \\
& && swdist_{i,j} = \begin{cases} |xs_i - xs_j| + |ys_i - ys_j| & \text{, if } switch_i \text{ connected to } switch_j \\ 0 & \text{, otherwise} \end{cases} \\
& && xs_i, ys_i \geq 0, \ \forall i \in 1 \cdots nsw
\end{aligned} \tag{3.5}$$

I use the *lp_solve* package [135] to obtain the optimum solution for the switch co-ordinates. Even for big applications (65 cores, tens of switches), the optimal solution is obtained in few seconds. I also pipeline long links to support full throughput on the NoC and add Network Interfaces (NIs) to connect the cores to the network. The resulting design is a valid floorplan of the NoC. The TSV macros do not need to be included in the linear program as TSVs split the wires in two segments, both carrying the same bandwidth. Therefore the placement of the TSV macro is more relaxed.

However, placing the components at the ideal positions may lead to overlap with the already placed cores. To remove such overlaps, I consider one switch or TSV macro at a time. I try to find a free space near its ideal location to place it. In the case of the switches the area in which I look for free space is the same for all of the switches, as it is given as a constant to the floorplanning routine. In case of the TSV the area in which I look depends on the blocks the TSV is connected to as explained before. If no space is available, I displace the already placed blocks from their positions in the x or y direction by the size of the component, creating space. Moving a block to create space for the new component can cause overlap with other already placed blocks. I iteratively move the necessary blocks in the same direction as the first block until I remove all overlap. As more components are placed, they can re-use the gap created

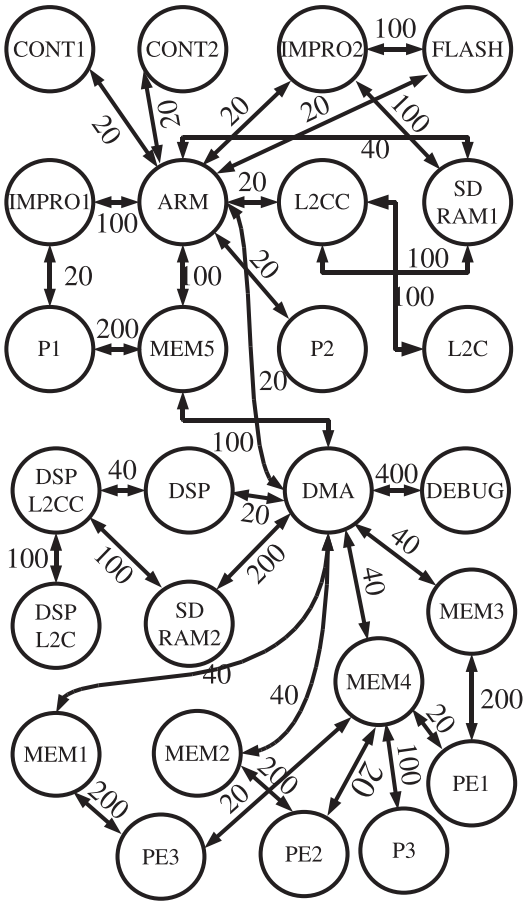

Figure 3.10: *D*26_*media* communication graph

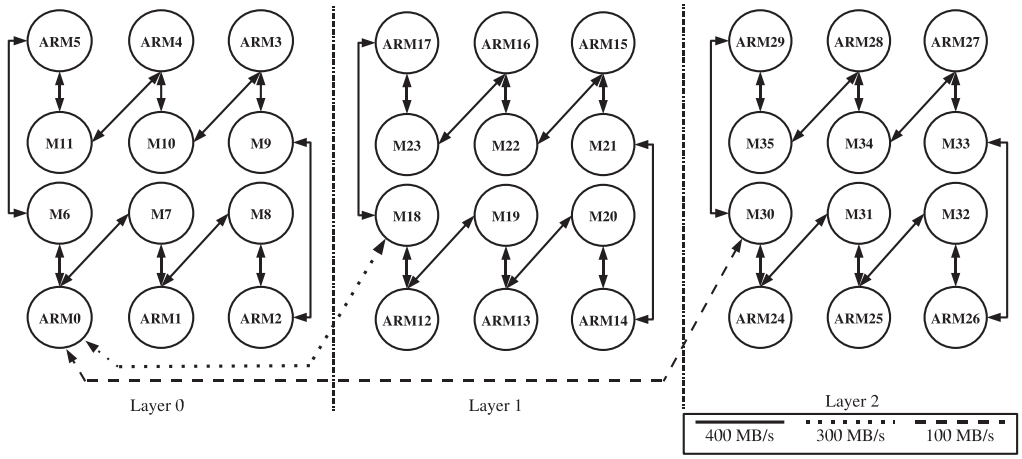

Figure 3.11: *D*36_4 communication graph

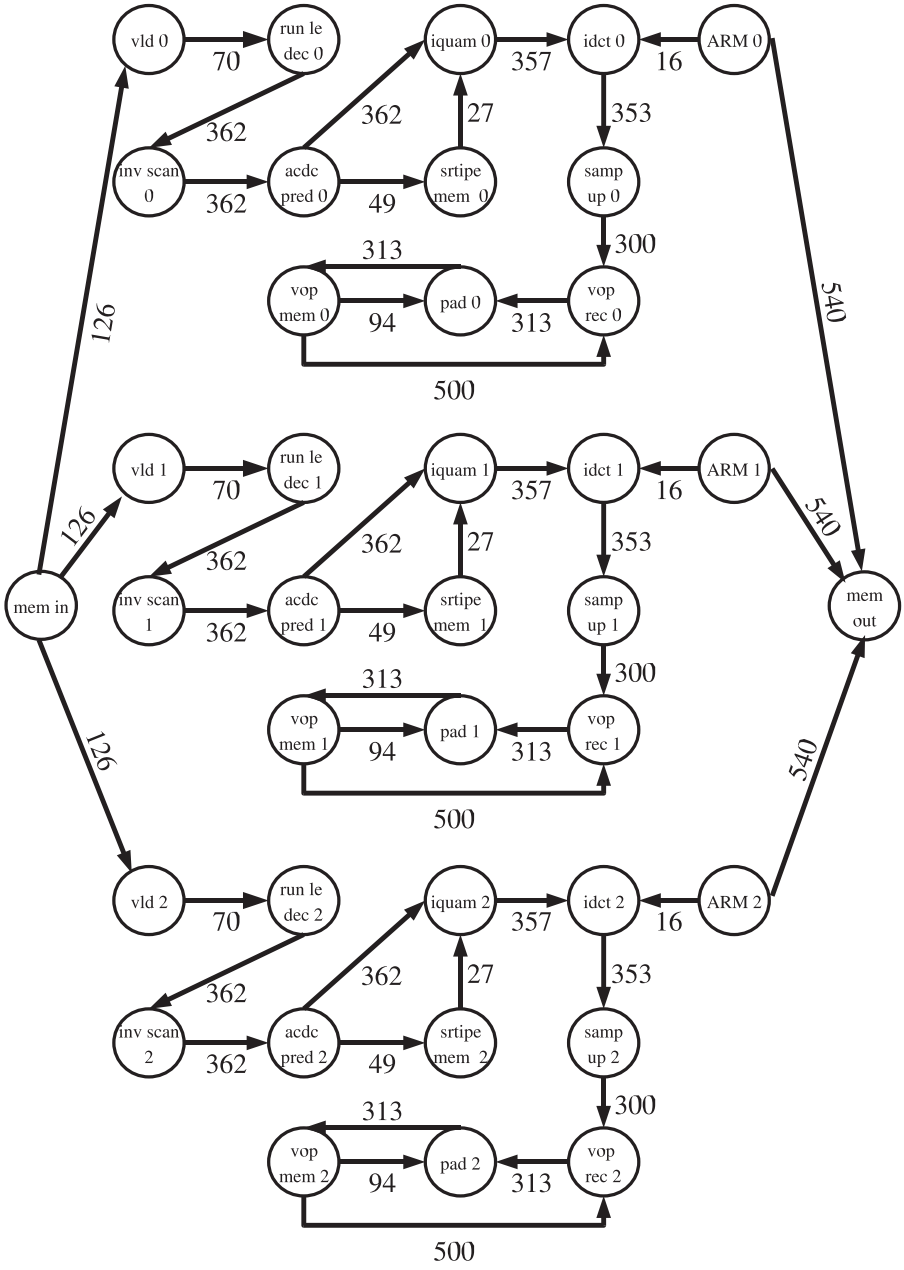

Figure 3.12: $D38\_tvopd$ communication graph

by the earlier components. As some switches can cause thermal hotspots, a more advanced thermal aware floorplanning routine can try to find position of switches near cooler components, so that the thermal distribution is more uniform. Several existing works, which are complementary to this one, present methods that address this issue [36], [70], [71], [96], [180]. However, integrating such works is beyond the scope of this work. In the experimental section I compare how this custom routine for floorplanning compares to a standard floorplanner that is constrained to only insert the network components without changing the relative placement of the initial cores. The standard floorplanner that I used to compare against is Parquet [3]. The floorplanner was used on each layer separately. I feed the core and switch positions as an input solution to the floorplanner. I allow it to move the switches around the cores, maintaining the relative positions of the cores and minimizing the movement of the switches from the optimal positions computed by the LP.

3.6 Experiments and case studies

For the experiments, the NoC component library from [160] is used. The power and latency values of the switches and links of the library are determined from post-layout simulations, based on 65nm low power libraries. The vertical interconnects using TSVs are implemented based on the models from [72]. In [72], the reported delay values for TSV placed in a tightly packed TSV bundle are $16ps$ and $18.5ps$ (for SOI and bulk silicon respectively). The considered TSVs have a diameter of $4\mu m$ and a pitch of $8\mu m$. When compared against the maximum unrepeated planar link length of $1.5mm$ in Metal 2 or Metal 3 for the same technology the authors show that the vertical links have much lower resistance and capacitance (an order of magnitude reduction for the resistance as well as for the capacitance). As a consequence even tightly packed TSVs are substantially faster and more power efficient than moderate planar links.

3.6.1 Multimedia SoC case study

I consider a benchmark of a realistic multimedia and wireless communication SoC for case-study (referred to as D_26_media). The benchmark contains 26 cores with irregular sizes, and performs based-band and multi-media processing. The communication graph of the benchmark is shown in Figure 3.10. The system includes ARM, DSP cores, multiple memory banks, DMA engine and several peripheral devices. The cores are manually mapped on to three layers in 3D. For comparisons, I also consider a 2D implementation of the benchmark. The initial positions of the cores in each layer of the 3D and for the 2D design are obtained using existing tools [3]. For fair comparisons, I use the same objectives of minimizing area and wire-length when obtaining the floorplan for both the cases. To synthesize the topologies for the 2D

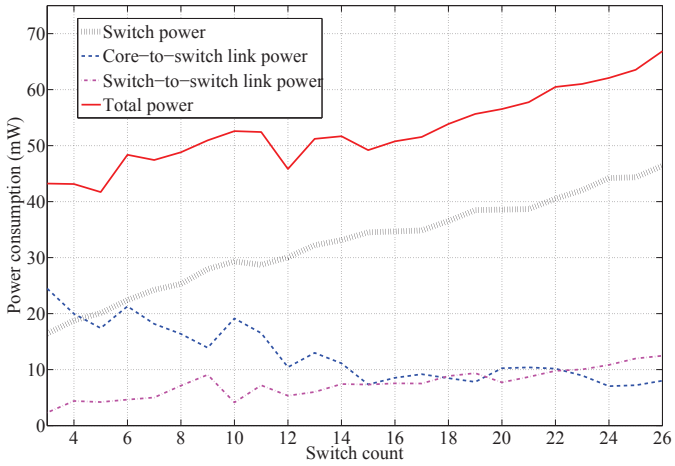

Figure 3.13: Power consumption in 2D

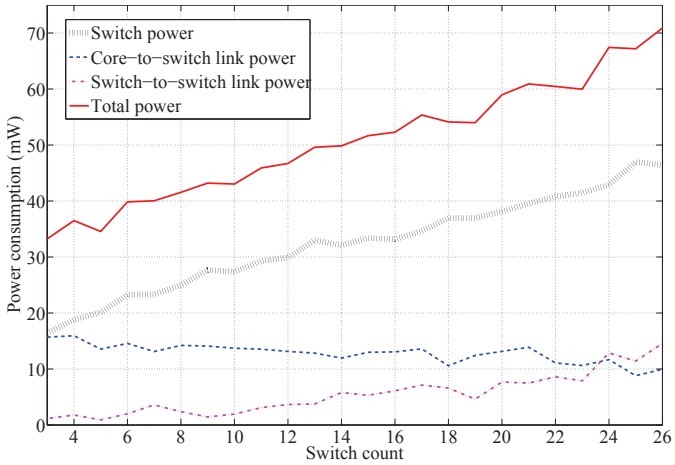

Figure 3.14: Power consumption in 3D

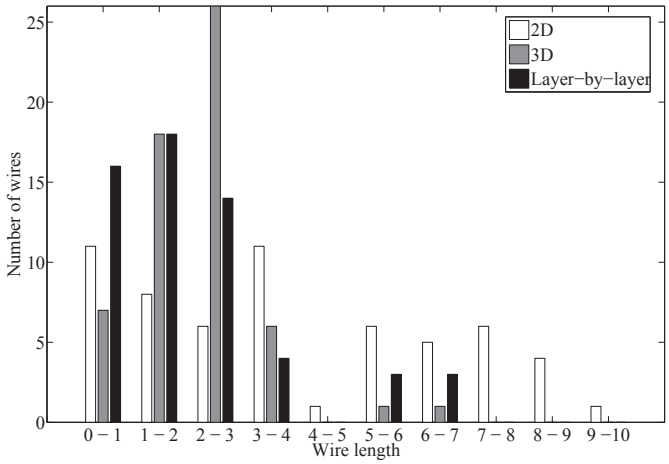

Figure 3.15: Wire length distributions

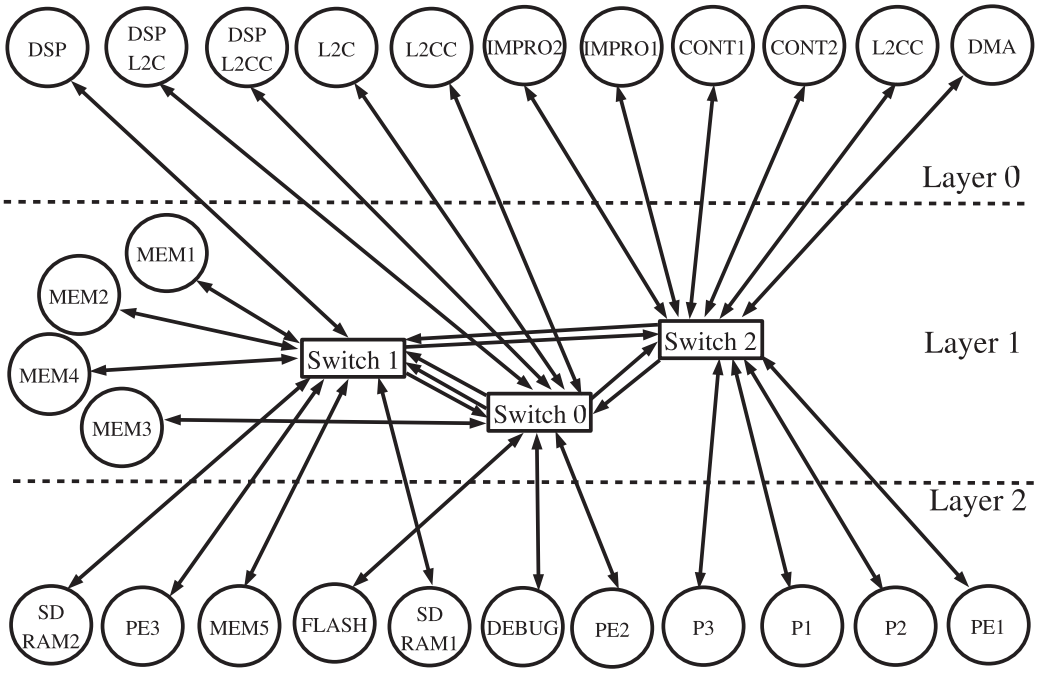

Figure 3.16: Most power-efficient topology (*Phase*1)

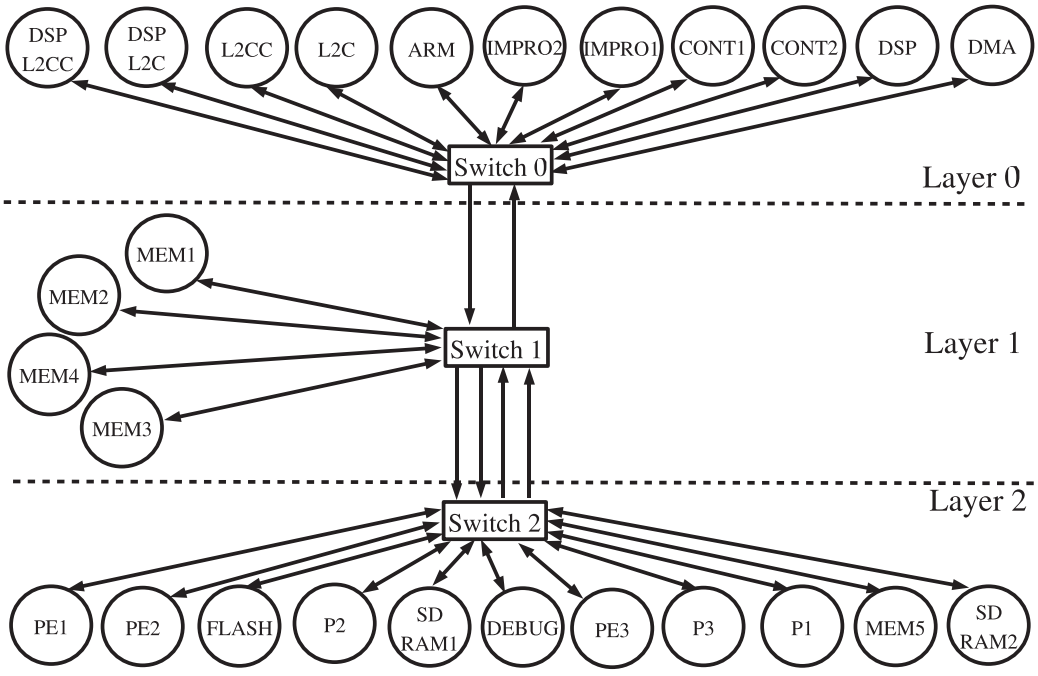

Figure 3.17: Most power-efficient topology layer-by-layer (*Phase*2)

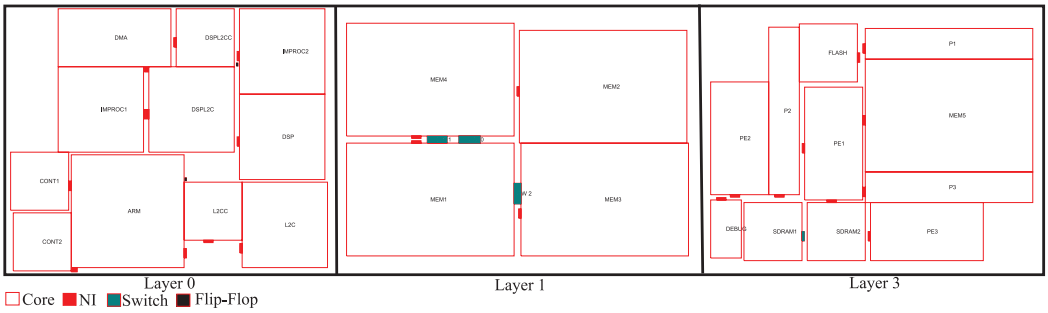

Figure 3.18: Resulting 3D floorplan with switches for the topology from Figure 3.16

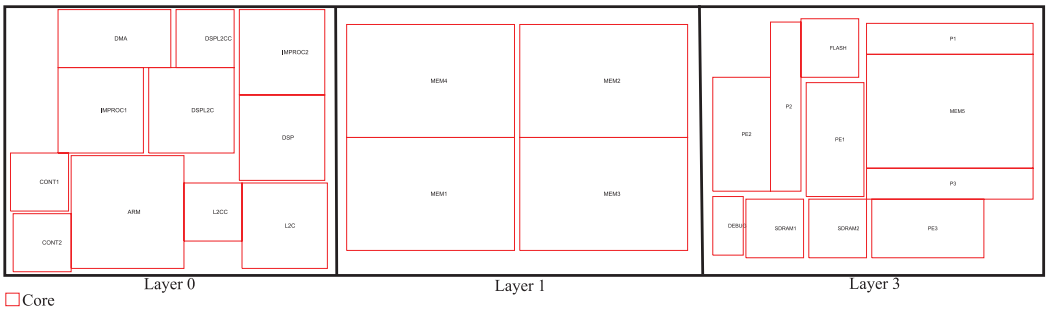

Figure 3.19: Initial positions for $D26\_media$

case, I use the synthesis flow developed earlier [121].

In Figures 3.13 and 3.14, I present the power consumption of the NoC topologies (power consumption on switches and links) synthesized by my tools for different switch counts for both cases. In all the experiments, I set the data width of the NoC links to 32 bits, to match the core data widths. The frequency for which the topologies are generated has to be given as an input. A range of frequencies can also be swept by the tool to explore more design points. However, for this benchmark the best power points are obtained for topologies designed at the lowest possible operating frequency, which was found by the tool to be 400 MHz. Higher operating frequency can be used (usually with a higher cost in power consumption). I use a *max_ill* constraint of 25 links for this and the experiments in the next sub-section. In Sub-section 3.6.5, I study the impact of varying this constraint. Please note that the frequency found by the tool is for the NoC to support the bandwidth required by the application. As the $D26\_media$ represent a mobile communication platform, the communication demands can be satisfied with the NoC running at 400 MHz. However the IP-cores may require higher operating frequencies to perform the desired functionality and generate the traffic described by the benchmark.

When very few switches are used in the design, they need to have more input/output ports, as they need to connect to more cores. A large switch can only support a low

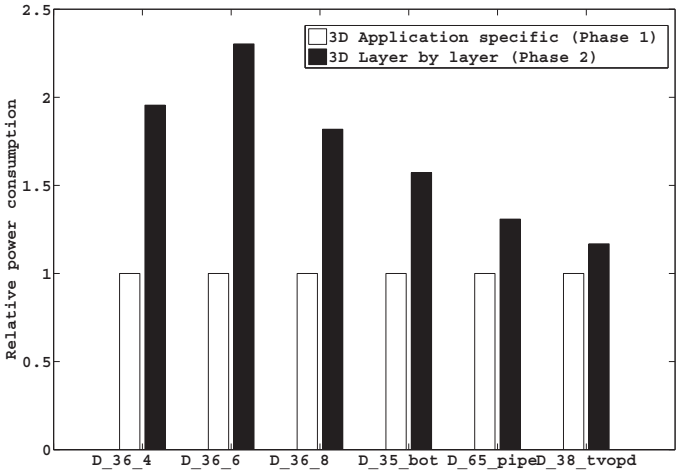

Figure 3.20: Comparison with layer-by-layer

operating frequency, as the critical path inside the switch increases with its size. In order to meet the 400 MHz requirement, I could only obtain valid topologies with 3 or more switches, thus the plots starts at 3 switches. In the plots, I show the switch, switch-to-switch link and core-to-switch link power consumption values as well. For this benchmark I can observe a power savings of 24% for the 3D case with respect to the 2D case. This is due to the fact that the long horizontal wires in a 2D design are replaced by shorter vertical wires. In Figure 3.15, I show the wire-length distribution of the links in 2D and 3D cases. From the figure, as expected, the 2D design has many long wires. In Figures 3.16 and 3.18, I present the most power efficient topology synthesized by my tool using phase 1 of the algorithm and the floorplan of the cores and network components for the 3D case. The original placement of the cores for this benchmark is shown in Figure 3.19.

In order to show how phase 2 of the algorithm performs, I constrained the tool to use the layer-by-layer approach and I ran it on the same benchmark. The topology for the best power point is presented in Figure 3.17. Even though I used the same *max_ill* constraint of 25 links as in the previous case, it can be seen from the figure that the algorithm used a lot less inter-layer links. This is also an intuitive example of why phase 2 of the algorithm is able to produce valid topologies even for tight *max_ill* constraints where phase 1 fails. There is also a price to pay for using fewer interlayer-links. In the case of the phase 2 topology, cores on different layers will have a zero load latency of at least 2 cycles as they have to go through two switches. For phase 1 cores on different layers are connected to the same switch so even if two cores are on different layer they could still have a zero load latency of just one cycle.

3.6.2 Comparison between Phase 1 and Phase 2

I applied my synthesis procedure on varied set of benchmarks to validate the gains under different application scenarios. I consider three distributed benchmarks with 36 cores (18 processors and 18 memories): D_36_4 shown in Figure 3.11, D_36_6 and D_36_8, where each processor has 4, 6 and 8 traffic flows going to the memories. The total bandwidth is the same in the three benchmarks. I consider a benchmark, D_35_bot that models bottleneck communication, with 16 processors, 16 private memories (one processor is connected to one private memory) and 3 shared memories to which all the processors communicate. I also consider two benchmarks where all the cores communicate in a pipeline fashion: 65 core (D_65_$pipe$) and 38 core designs (D_38_$tvopd$) shown in Figure 3.12. In the last two benchmarks, each core communicates only to one or few other cores.

In Figure 3.20, I show the power consumption of the topologies synthesized using Phase 2 of the algorithm, with respect to topologies synthesized using Phase 1 for the different benchmarks. Since in Phase 2 cores in a layer are connected to switches in the same layer, the inter-layer traffic needs to traverse more switches to reach the destination. This leads to an increase in power consumption and latency. As seen from Figure 3.20, Phase 1 can generate topologies that lead to a 40% reduction in NoC power consumption, when compared to the Phase 2. However Phase 2 can generate topologies with a much tighter inter-layer link constraint.

3.6.3 2D vs. 3D comparison

The power consumption for the least power design points for 2D and 3D, as well as the average latency are presented in Table 3.1. Most of the power savings obtained in 3D are due to shorter wires. For this reason, I can observe large power savings for the distributed benchmarks, where there are traffic flows to many different cores. I can also notice reasonable power savings for the bottleneck design, because the wires going to shared memories are long, though the traffic to the shared memories is smaller than to the private memories. For the pipelined benchmarks, lower savings are

Benchmark	Power (mW)						Latency (cyc)	
	Link power		Switch power		Total power			
	2D	3D	2D	3D	2D	3D	2D	3D
D_36_4	150	41.5	65	70.5	215	112	3.28	3.14
D_36_6	154.5	43.5	76.5	82	230	125.5	3.57	3.5
D_36_8	215	55.5	105	104.5	320	160	4.37	3.65
D_35_bot	68	36.2	48	43.3	116	79.5	6.04	4.2
D_65_$pipe$	106	104	63	58	169	162	2.53	2.57
D_38_$tvopd$	52.5	22.67	37	38.11	89.5	60.78	4	3.6

Table 3.1: 2D vs 3D NoC Comparison

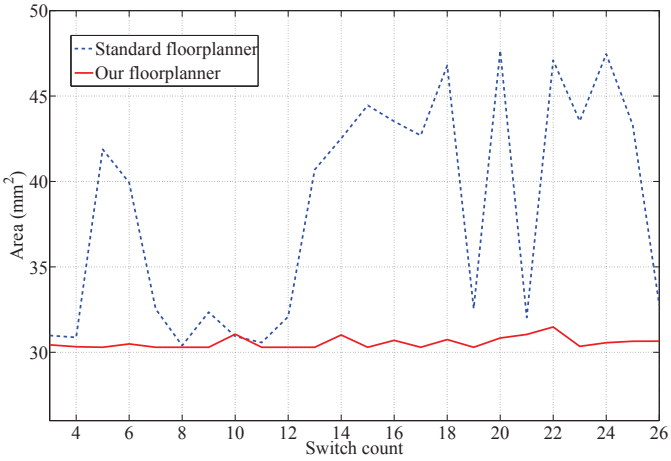

Figure 3.21: Area plot for different switch counts

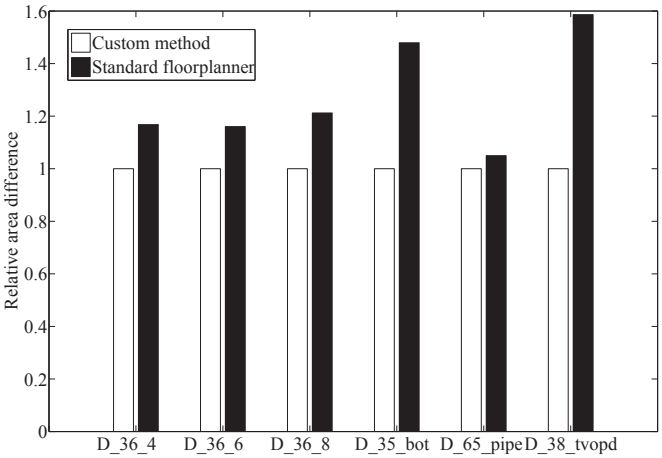

Figure 3.22: Area comparison for different benchmarks

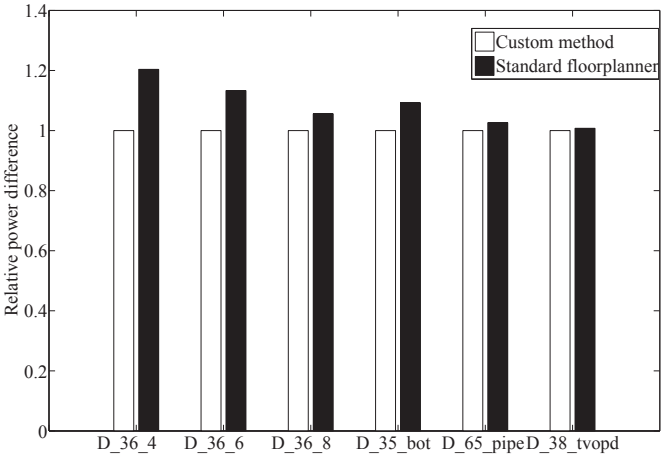

Figure 3.23: Power comparison for different benchmarks

obtained. For the different benchmarks, on average, a 38% power reduction and 13% latency reduction are obtained in the 3D case when compared to a 2D implementation.

3.6.4 Floorplanning study

To create a floorplan for a real SoC is quite a complicated process that can require several interactions between the designed and the floorplanning tool. Since the main goal of my tool is to design the NoC and not to create floorplans, I take the initial positions of the cores as input. I then only insert the NoC components as close as possible to their ideal positions in the floorplan in such a way that I minimally affect the initial positions of the cores.

Most of the time placing the NoC components at their ideal positions will result in overlap with the initially placed cores, especially if the floorplan is tightly packed. Initially I tried to use a standard floorplanner to remove the overlap. I used the floorplanner from [3] which I have modified in order to constrain it from swapping blocks, so that the relative positions of the input cores remain the same after the NoC insertion. This however leads to fairly poor results as the floorplanner has problems to remove the overlap, keep the cores close to their initial placement and not swap any of them. The floorplanner needs the ability to swap blocks in order to create good floorplans, for this reason this full floorplanner is ideal for generating the initial placement, but not for the NoC insertion. To achieve better results in inserting the NoC components and removing the overlap I designed a custom floorplanning routine which I tuned for this specific task alone.

In Figure 3.21, I show a comparison between the standard floorplanner and my custom routine for different switch counts using the D_26_media benchmark. From the figure it can be seen that for some points even the standard floorplanner performs well and that there is a better chance to get a good floorplan when fewer switches are inserted. However the behavior of the constrained standard floorplanner is unpredictable. A comparison between the best power points for the different benchmarks using the two floorplanning methods is shown in Figure 3.22. Since the area has a direct impact on the wire lengths and consequently on the power I also present a comparison of the power consumption for the considered topologies in Figure 3.23.

3.6.5 Impact of inter-layer link constraint and comparisons with mesh

Imposing a stricter constraint on *max_ill* results in topologies having more switches. When there are more switches, more cores in a layer are connected to a switch in the same layer, reducing the number of inter-layer links. However, the inter-layer traffic flows would need to traverse more switches, there by leading to higher power

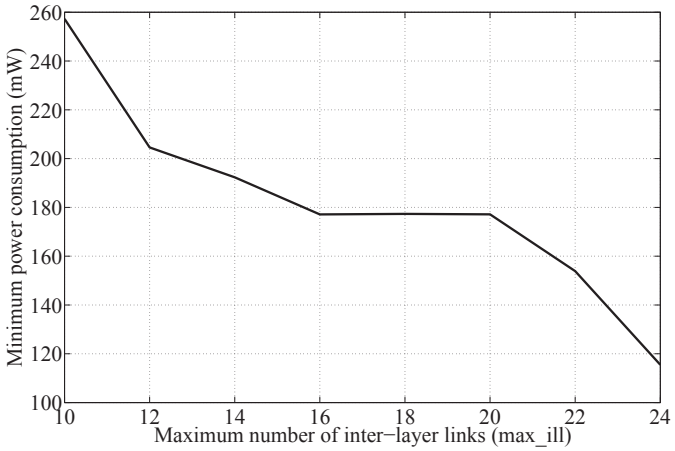

Figure 3.24: Impact of max_ill on power

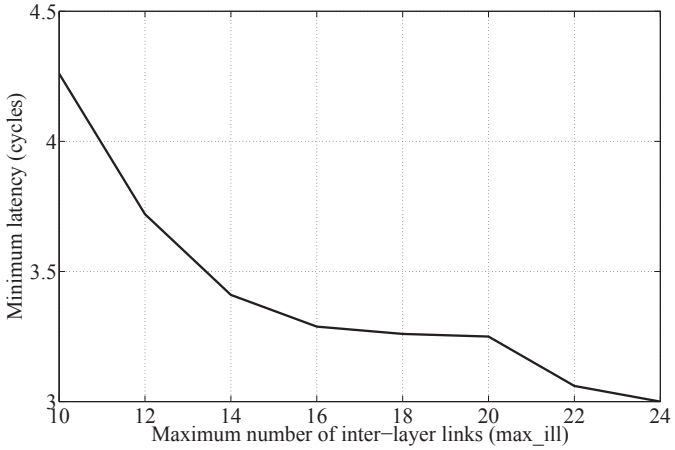

Figure 3.25: Impact of max_ill on latency

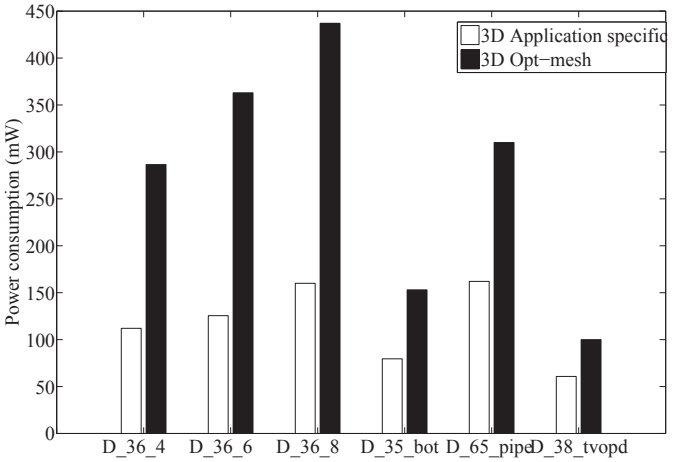

Figure 3.26: Comparisons with mesh

consumption and latency. I perform topology synthesis for the D_36_4 design with different *max_ill* constraint values, and the power, latency values for the different points are presented in Figures 3.24 and 3.25. With a tighter TSV constraint, the power consumption and latency increases significantly, as more switches are needed in the design. With less then 10 inter-layer links it is impossible to build any topology and having a *max_ill* constraint larger than 24 does not improve the results anymore. For the D_26_media benchmark, I observe a similar trend for the power consumption. However the zero load latency is not affected by tighter *max_ill* constraints, due to the nature of the traffic of this benchmark.

For completeness, I compare power consumption of the topologies generated by my procedure to a standard topology. I generate best mapping (optimizing for power, meeting the latency constraints) of the cores on to a mesh topology, and remove any unused switch-to-switch links. Compared to this optimized mesh topology, I obtain a large power reduction for the custom topologies (an average of 51%), shown in Figure 3.26. The experiments also showed that I obtain 21% reduction in latency when compared to the optimized mesh.

Even though the algorithm explores a large space of solutions, due to the use of efficient heuristics presented, all the experiments could be performed in few hours (on a system operating at 2 GHz). It takes a few seconds to build a topology with few switches and the run time can go up 2, 3 minutes for topologies with many switches (50, 60 switches). The total runtime on a benchmark depends on the frequency range and switch count range that are swept. Also, it is important to note that the synthesis algorithm has to be performed only once at design time for a system and the timing overhead is negligible.

3.7 Summary

Networks on Chips (NoCs) are necessary to achieve a scalable communication infrastructure in 3D chips. The use of NoCs in 3D ICs introduces several new and challenging problems. Building a custom NoC topology that meets the application communication requirements, as well as the 3D technological constraints, is a critical problem that needs to be addressed. In this chapter, I presented algorithms and methods for NoC topology synthesis for 3D ICs. The problems addressed are: i) topology synthesis, ii) assignment and placement of network components in the 3D layers and iii) insertion of the components in an existing floorplan. Experiments on several realistic benchmarks show that the algorithms produces topologies that result in large NoC power and latency savings (54% and 21%, respectively) when compared to standard topologies. I also presented a comparative analysis of NoCs in 2D and 3D, which shows that 3D integration can produce large interconnect power and latency

reduction (38% and 13%, respectively).

4 Designing the NoC for SoCs with VFIs and shutdown capabilities

Distribution of clock trees is a major design challenge today. With advancing technology generations, the clock frequency and design area increases and only a portion of the chip can be reached in a single clock cycle [24]. For ease of design, many complex systems are partitioned into multiple *Voltage and Frequency Islands (VFIs)*. Each island is synchronous, using the same frequency and voltage lines. With technology scaling, the leakage power consumption is increasing rapidly as a fraction of the total power consumption. In fact, leakage power can be responsible for 40% or more of the total system power [51]. To reduce the power consumption, if the cores inside a VFI are not used for a particular application, the entire VFI can be shutdown and thus saving the energy lost in leakage. The cores in a VFI can be connected using a local interconnect (a local NoC). The local interconnects are then connected using a global NoC. The global NoC can itself be synchronous or asynchronous, and the latter case is called as the *Globally Asynchronous, Locally Synchronous (GALS)* paradigm.

In today's SoCs, the NoC architecture is a bottleneck in enabling the shutdown of the islands. There are several approaches presented to synthesize application-specific NoCs [134]-[121]. However, none of them consider the issue of shutdown of VFIs. These approaches cannot be directly extended to design NoCs for SoCs with voltage islands. If the NoC is designed using such approaches, either the whole NoC should be placed in a separate VFI or the islands cannot be shutdown.

Placing the entire NoC in a separate VFI is not a feasible solution. The NoC switches are usually spread across the chip, connecting the different cores. If the entire NoC is in the same island, it is difficult to route the VDD and ground lines for the NoC across the chip. On the other hand, if all the NoC switches are physically clustered and placed in the center of the chip, then long wires are needed to connect all the cores to the NoC island. Thus, the routing congestion would be enormous and the solution is not scalable. Moreover, additional resources for routing the additional voltage and ground lines may not even be available in the design. If the switches are spread across

the different VFIs, then the switches in a VFI will share the same voltage lines as the cores in the island. However, if a VFI needs to be shutdown, then packets between cores on the other VFIs that use the switches in this VFI cannot be transmitted. This will prevent the shutdown of the entire island.

The concept of voltage island should be considered during the NoC topology synthesis phase itself. In this chapter, I present a synthesis approach to determine the best NoC topology points that are tailored to meet the application performance constraints, minimizing power consumption and supporting the ability to shutdown voltage islands. The main contribution of this chapter are:

- I present a synthesis algorithm to synthesize application specific NoC topologies for SoCs that are partitioned in VFIs.
- I present how the algorithm can be extended for 3D-SoCs that use multiple VFIs.
- I evaluate the overhead of VFI design on the NoC power consumption and latency.
- I make a comparison between NoC designed for 2D- and 3D-SoCs in the presence of VFIS.

NoC links that cross from one VFI to another change frequency domains, and therefore require a frequency converter. In 3D systems, even if the whole design is synchronous, ensuring a zero-clock skew across different layers is difficult [102]. In this case, mesochronous synchronizers are needed for the vertical links. For example, in [102], an efficient design of such synchronizers for vertical links is presented. In Chapter 3, I presented a synthesis approach for 3D systems where I assumed that the entire design was fully synchronous. In this chapter I extend the synthesis method to account for such necessary clock domain crossings as well as for the possibility of VFI design.

In such VFI based designs, the benefits of 3D integration in reducing NoC power or delay is unclear. Earlier works either made comparisons using standard topologies (such as meshes), or did not consider VFI partitioning [132], [53] and [149]. One of the objectives of this chapter is to make a comparative study of NoCs for 2D and 3D implementation of SoCs. The aim is to show quantitative benefits of the 3D technology on NoC power and delay values.

I present a detailed case-study of NoCs designed using the flow for a mobile platform. The experiments show that the support for VFI in the NoC design process only leads to a modest 3% increase in the active power consumption of the chip. This support can lead to a large reduction in the leakage and hence overall power consumption.

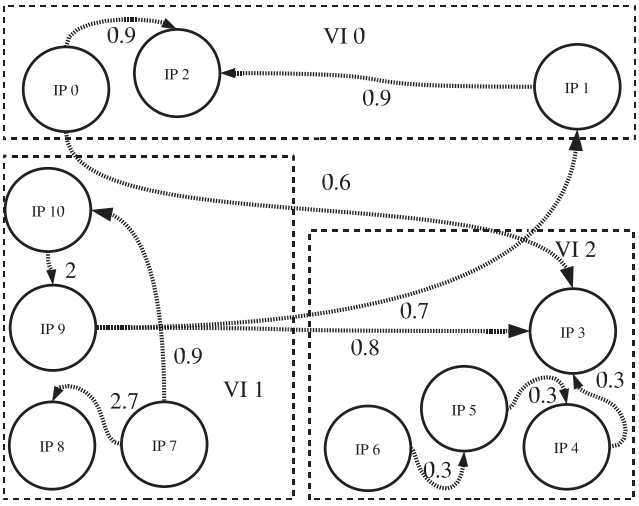

Figure 4.1: Example Input

The results show that when the whole design is synchronous, 3D designs give very low power savings (11%), as the mesochronous converters incur a lot of power overhead. As the number of VFIs increase, 3D SoCs have large NoC power reductions (up to 32%) due to reduction in wire lengths. However, after a sweet-spot, the gains fall again, as the wires get shorter in 2D due to the use of more switches and the contribution of converter power to overall power also becomes significant. The results show the need for an early architectural design space exploration of the whole space and the tools I present facilitate the same. Experiments also show that the reduction in delay is only marginal when moving from 2D to 3D systems (up to 11%), if both are designed efficiently. This is because, the number of links in 2D that are long enough to require pipelining is less and the frequency converter delay is dominant when compared to wire delay.

4.1 Problem description

In this section, I describe the architectural features of the NoC and the synthesis problem.

4.1.1 Architecture Description

An example of the architecture for which this custom NoC synthesis algorithm is designed is presented in Figure 4.1. The cores of the design are assigned to different VFIs, which is given as an input to my method. The cores in a VFI have the same operating voltage (same power and ground lines), but could have different operating

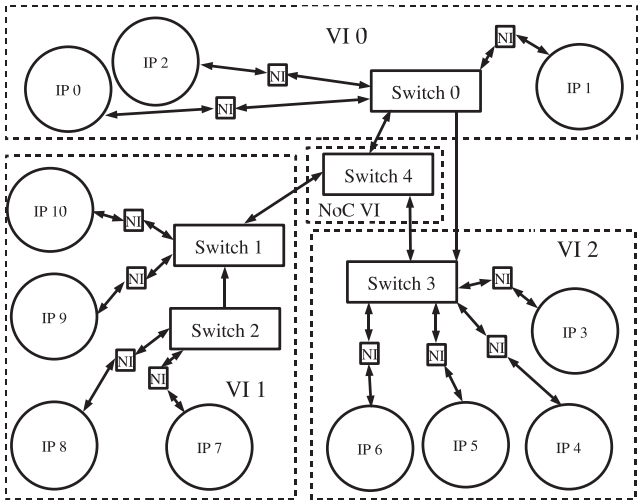

Figure 4.2: Example Architecture

frequencies. In order to have a scalable solution, I build NoC systems, where cores in a VFI are connected to switches in the same VFI. I follow an approach similar to the *GALS* approach, where the NoC components in a VFI are synchronous and operate at the same frequency. Having a locally synchronous design eases the integration of the NoC with standard back-end placement&routing tools and industrial flows. Moreover, if the different switches in a VFI operate at different frequencies, power and latency hungry synchronizers are needed to connect them.

The cores are connected to the NoC switches by means of Network Interfaces (NIs) that convert the protocol of the cores to that of the network. The NIs also perform clock frequency conversion, if the cores are running at different frequencies than the switches in the VFI. When a switch in one VFI is to be connected to a switch in another VFI, I use a bi-synchronous FIFO to connect them together. The FIFO takes care of the voltage and frequency conversion across the islands. The frequency conversion is needed because the switches in the different VFIs could be operating at different frequencies. Even if they are operating at the same frequency, the clock tree is usually built separately for each VFI. Thus, there may be a clock skew between the two synchronous islands. I use over the cell routing with unpipelined links to connect switches across different VFIs, as the wires could be routed on top of other VFIs.

The shutdown mechanism of an island could be performed in many ways, which does not affect the synthesis procedure presented in this work. In this system, I use the following mechanism: when an application does not require cores in a VFI, the power manager decides to shutdown the island. The power manager could be either implemented in hardware, or could be in the operating system. Before shutting down the island, the manager checks whether all the pending transactions in to and out of

the island are completed, by polling all the NIs. If there are no pending transactions and the cores are ready to be shutdown, it grounds the voltage lines to all the cores and network components in the island.

4.1.2 Synthesis problem

In this synthesis procedure, I generate switches in each VFI to connect the cores in the VFI. Optionally, the method can explore solutions where a separate NoC VFI can be created. I take the availability of power and ground lines for the intermediate VFI as an input, and my method will use the intermediate island, only if the resources are available. My method produces topologies such that a traffic flow across two different VFIs can be routed in two ways: (i) the flow can go either directly from a switch in the VFI containing the source core to another switch in the VFI containing the destination core, or (ii) it can go through a switch which is placed in the intermediate NoC VFI, if the VFI is available. The switches in the intermediate VFI are never shutdown. The method will automatically explore both alternatives and choose the best one for meeting the application constraints.

The objective of the synthesis method is to determine the number of switches needed in each VFI, the size of the switches, their operating frequency and routing paths across the switches, such that application constraints are satisfied and VFIs can be shutdown, if needed. The method determines if an intermediate NoC VFI needs to be used to connect the switches in the different VFIs and if so, the number of switches in the intermediate island, their sizes, frequency of operation, connectivity and paths. The synthesis method can be plugged in the design flow presented in [121] in order to generate fully implementable NoCs

An example NoC design produced by this synthesis approach is presented in Figure 4.2. As seen, the switches are distributed in the different VFIs. If an intermediate NoC island can be created, indirect switches could be generated in the intermediate island to connect the different switches together.

My method produces several design points that meet the application constraints with different switch counts, with each point having different power and performance values. The designer can then choose the best design point from the trade-off curves obtained.

Models: For a technology library, I obtain area, power and latency models for the NoC components (switches, NIs and bi-synchronous FIFOs). The models are obtained from synthesis and place&route of the RTL code of the components using commercial tools. The generated library of the NoC components is used during topology synthesis.

Inputs: The number of cores and their assignment to VFIs has to be given as an input to the algorithm. I assume that the partitioning of cores to the VFIs is based on the technological constraints, design issues and the desired floorplan. There are several research works that have addressed this issue [106], [126] and are complementary to the NoC synthesis issue addressed here. Optionally, the size and position of the cores are obtained. This floorplan information of the cores, if given, will lead to a better estimation of the wire power consumption and delays during the synthesis process. Another input is the communication description. In the communication description, for each traffic flow, the source and destination cores are specified and also bandwidth and latency constraints are given.

Outputs: Based on the inputs and models, I synthesize different topology design points. All the topologies generated will comply with the constraints given in the input description files, and may have different values for power, average latency, wire length, switch count. A detailed description of the algorithm is given in Section 4.2. If the size and initial positions of the cores were given as inputs, a floorplan with the NoC will also be generated. The floorplanning routine finds the best location for the NoC components and then inserts the NoC blocks as close as possible to the ideal positions, while minimally affecting the position of the cores given as input.

4.2 Topology synthesis approach

The synthesis algorithm is explained in detail in this section. From the input specifications, I construct the VFI communication graph. The definition of the VFI communication is similar to that from Chapter 3, but it is extended to account for the assignment of IP cores to VFIs defined as follows:

Definition 4.1 *A VFI Communication Graph (VCG(V, E, isl)) is a directed graph, each vertex $v_i \in V$ represents a core in the VFI denoted by isl and the directed edge (v_i, v_j) representing the communication between the cores v_i and v_j. The bandwidth of traffic flow from cores v_i to v_j is represented by $bw_{i,j}$ and the latency constraint for the flow is represented by $lat_{i,j}$. The weight of the edge (e_i, e_j), defined by $e_{i,j}$, is set to a combination of the bandwidth and the latency constraints of the traffic flow from core v_i to v_j: $h_{i,j} = \alpha \times bw_{i,j}/max\_bw + (1-\alpha) \times min\_lat/lat_{i,j}$, where max_bw is the maximum bandwidth value over all flows, min_lat is the tightest latency constraint over all flows and α is a weight parameter.*

The value of the weight parameter α can be set experimentally or obtained as an input from the user, depending on the importance of performance and power consumption objectives.

Algorithm 4.1 Core-to-switch connectivity

Determine the frequency at which the NoC will operate in each VFI and $max\_sw\_size_j$, $\forall j \in [1 \cdots N_{VFI}]$
$min\_sw_j = |VCG(V,E,j)| / max\_sw\_size_j$, $\forall j$
{Vary number of switches in each VFI}
for $i = 1$ to $max_{\forall j \in 1 \cdots N_{VFI}} |V_j|$ **do**
  for $j = 1$ to N_{VF} **do**
    if $i + min\_sw_j < |V_j|$ **then**
      $k = i + min\_sw_j$
    else
      $k = |V_j|$
    end if
    Perform k min-cut partitions of $VCG(V,E,j)$.
  end for
  {Vary number of switches in intermediate NoC VFI}
  for $k = 0$ to $max_{\forall j \in 1 \cdots N_{VFI}}$ **do**
    Compute least cost paths for inter-switch flows using $Check\_constraints$ procedure. Choose flows in bandwidth order and find the paths.
    If paths found for all flows save design point
  end for
end for
Ensure: $Check\_constraints$
{Check if the link between $switch_i$ or $switch_j$ can be used}
if $island(switch_i) = island(switch_j)$ **then**
  $link\_allowed$ = TRUE;
else if $island(switch_i) = src\_isl$ and $island(switch_j) = dest\_isl$ **then**
  $link\_allowed$ = TRUE;
else if $switch_i$ or $switch_j$ is in NoC VFI **then**
  $link\_allowed$ = TRUE;
else
  $link\_allowed$ = FALSE;
end if
$h = island(switch_i)$ and $k = island(switch_j)$
if $size(switch_i) >= max\_sw\_size_h$ or $size(switch_j) >= max\_sw\_size_k$ or $link\_allowed$ = FALSE **then**
  $cost_{ij} = INF$
else if $size(switch_i) >= max\_sw\_size_h - 2$ and $switch_j$ is in NoC VFI **then**
  $cost_{ij} = SOFT\_INF/2$
else if $size(switch_j) >= max\_sw\_size_k - 2$ and $switch_i$ is in NoC VFI **then**
  $cost_{ij} = SOFT\_INF/2$
else
  $cost_{ij} = SOFT\_INF$
end if

The algorithm for topology synthesis is presented in Algorithm 4.1. In the first step of the algorithm, the frequencies of operation of the switches in each of the islands are determined. In this NoC design, a core is connected to only one switch, through a NI. The links connecting the NI and the switch determine the frequency at which the NoC elements have to run in an island. The bandwidth available on a link is a product of the link data width and the frequency. In the synthesis procedure, without loss of generality, I set the data width of the NoC links to a user-defined value. Please note that it could be varied in a range and more design points could be explored, which does not affect the algorithm steps. For a fixed data width, the frequency of the switches in an island is determined by the link that has to carry the highest bandwidth from or to a core in the island.

A larger switch will have a longer critical path in the crossbar and therefore will have to operate at a smaller frequency. The frequency at which a switch has to operate determines the maximum size (number of inputs and outputs) of the switch that can be allowed, denoted as $max\_sw\_size_j$. As the switches in the different VFIs can operate at different frequencies, the maximum switch size is different for the different VFIs. Once the maximum switch sizes are determined, based on the number of cores in each VFI, the minimum number of switches required for each island is determined in step 2 of the algorithm. Let N_{VFI} denote the total number of VFIs in the design.

Example 4.1 *Consider the system depicted in Figure 4.1. I describe how the algorithm works for one design point. The design has eleven cores divided in to three islands. The first step is to determine the frequency of the NoC in each VFI and to calculate the maximum size of the switches in each VFI. In this example, IP 7 is generating the maximum traffic in the island VFI 1, with a total of 3.6 GB/s bandwidth. Let us assume that the NoC data width is set to 4 bytes. Thus, the NoC island with IP 7 should run at 900 MHz (obtained by 3.6 GB/4 B). From the ×pipes NoC [160] library that I used in 65nm, I found that a switch larger than* 3×3 *cannot operate at 900 MHz. Thus, I determine that the maximum switch size for this island is* 3×3*. As the island has 4 cores, I need at least 2 switches in the island. The minimum number of switches needed in the other islands can be calculated in a similar manner.*

In steps 4 to 10 of the algorithm, the number of switches in each island is varied from the minimum value (computed in step 2) to the maximum number of cores in the island.

Example 4.2 *Let us assume that the minimum number of switches computed in step 2 for the example in Figure 4.2 are 1, 2, 1 for VFI 0, VFI 1, VFI 2. I will generate design points with different switch counts, with each point having one more switches in each VFI,*

until the number of switches is equal to the number of cores in the VFI. For this example, I will explore the following points: 1,2,1, 2,3,2, 3,4,3, 3,4,4. As several combinations of switch counts in different VFIs are possible, I limit to this simple heuristic.

In step 11, for the current switch count of the VFI, that many min-cut partitions of the VCG corresponding to the VFI are obtained. Cores in a partition share the same switch. As min-cut partitioning is used, cores that communicate heavily or that have tighter latency constraints would be connected to the same switch, thereby reducing the power consumption and latency.

Example 4.3 *For the design point 1,2,1, 2 min-cut partitions of VCG(V,E,1) are obtained. Cores IP 9 and IP 10 communicate more and belong to the same partition. Thus, they would share the same switch. Also, all flows between cores on the same switch will be routed directly through that switch.*

At this point, the connectivity of the cores with the different NoC switches is obtained. I still need to connect the switches together and find paths for the inter-switch traffic flows. If the switches from a VFI are directly connected to the switches on the other VFIs, then several switch-to-switch links would be needed. This may lead to large switch sizes, which may lead to violation of the $max\_sw\_size_j$ constraint. By using switches in an intermediate NoC island, the number of switch-to-switch links can be reduced. These switches act as indirect switches, as they are not directly connected to the cores, but only connect other switches. If the design constraints permit the usage of another VFI, then I explore the solution space (step 14) with varying number of switches on the intermediate NoC VFI.

For each combination of direct and indirect switches, the cost of opening links is calculated and the minimum cost paths are chosen for all the flows (step 15). The traffic flows are ordered based on the bandwidth values and the paths for each flow in the order is computed. The cost of using a link is a linear combination of the power consumption increase in opening a new link or reusing an existing link and the latency constraint of the flow. The different scenarios for setting the link costs are shown in the $Check\_constraints$ procedure (step 19 to 38). When opening links, I ensure that the links are either established directly across the switches in the source and destination VFIs or to the switches in the intermediate NoC island. To enforce this constraint, a large cost (INF) is assigned to the links that are not allowed. Similarly, when the size of a switch in an island reaches the maximum value, the cost of opening a link from or to that switch is also set to INF. This prevents the algorithm in establishing such a link for any traffic flow. Also, when a switch is close to the maximum size (2 ports less then the maximum size), a larger value then the usual cost is assigned for opening

a new link, denoted by $SOFT\_INF$. This is to steer the algorithm to reuse already opened links, if possible. In order to facilitate the use of the indirect switches in the intermediate NoC VFI, the cost of opening a link between a switch that is close to the maximum size and an indirect switch is set to $SOFT\_INF/2$. Thus, when the size of a switch approaches the maximum value, more connections will be established using the switches in the intermediate NoC VFI.

Example 4.4 *Let us consider the switch assignment from the previous example. In this example, the highest bandwidth flow that has to be routed first is the one from IP 7 to IP 10. This will result in opening a link between Switch 2 and Switch 1. Now, let us assume that I have to find a path for a flow from IP 9 to IP 3. Because Switch 1 is close to its maximum size and I have other flows to other VFIs, the algorithm will use the switch in the intermediate NoC VFI. This results in opening a link from Switch 1 to 4 and another from Switch 4 to 3. The topology with the inter-switch links opened is shown in Figure 4.2.*

If for all the flows, paths that do not violate the latency constraints are found, then the design point is saved. Finally, for each valid design point, the NoC components are inserted on the floorplan and the wire lengths, wire power and delay are calculated. The time complexity of the algorithm is $O(V^2E^2ln(V))$, however in practice the algorithm runs quite fast as the input graphs typically are not fully connected.

4.3 3D architecture and design approach

I assume a 3D manufacturing process based on the wafer-to-wafer bonding technology as in Chapter 3. In this, Through Silicon Vias (TSVs) are used for establishing vertical interconnections. A vertical link requires a TSV macro on one of the layers (say the top layer), where the via cuts through the silicon wafer. In the bottom layer, the wires of the link will use a horizontal metal layer to reach the destination. For links that go through more than one layer, TSV macros are required in all the intermediate layers. However, it is important to note that the macros need not be aligned across the layers, as horizontal metal layer can be used to reach the macro at each layer as well. Stacked TSVs are not used as the alignment of the TSVs would complicate floorplanning. The area of the TSV macros for a particular link width is taken as an input. For the synthesized topologies, the tool automatically places the TSV macros in the intermediate layers and on the corresponding switch ports. The synthesis process automatically places the TSV macros at different layers for the different vertical interconnects.

Extending the algorithm from Section 4.1 to generate NoC topologies for 3D-ICs

is done by combining the previously presented algorithm with the algorithm for generating custom NoC topologies for 3D-ICs from Chapter 3. In the case of the 3D algorithm from Chapter 3, the cores are assigned to layers of the 3D silicon stack. Cores can be connected only to switches in the same layer. The 3D algorithm would explore designs with different number of switches in each layer. The concept of layer is similar to the concept of VFI, however in the original 3D algorithm all layers are assumed to be synchronous.

The extension of the 3D algorithm to support shutdown of VFIs as presented in Section 4.1 can be done under the assumption that a VFI does not span across multiple layers. This assumption makes sense because it is difficult to create a synchronous clock tree that can span on multiple layers. Therefore, even if cores on different layers operate at the same frequency and voltage level it is very likely that there will be clock skew between them and they would need to be assigned to different VFIs.

The algorithm to synthesize NoCs for 3D ICs takes as input both the assignment of cores to the silicon layers of the 3D stack as well as the assignment of the cores to VFIs. Also the maximum number of links that can cross between two adjacent layers has to be given as input. This constraint is used to limit the number of *Through Silicon Vias (TSVs)* that are needed to connect components on different layers in order to increase the yield.

4.4 Experimental results

Experiments are performed using the power, area and latency models for the NoC components based on the architecture from [160]. The models are built for 65nm technology node. I extended the library with models for the bi-synchronous voltage and frequency converters. For reference the power consumption (with 100% switching activity), area and maximum operating frequency for some of the components are presented in Table 4.1. In [72], the authors show that the power consumption of tightly packed TSVs is smaller than that of horizontal interconnect by two orders of magnitude. Therefore, the impact of power consumption and delay of the vertical links is negligible as they are very short as well (15 to 25μm). Under zero-load conditions, the switch delay is 1 cycle, an unpipelined link delay is 1 cycle and the worst case converter delay (which I use in the analysis) is 4 cycles (of the slowest clock) [35].

4.4.1 Case study on mobile platform

When the NoC has to be designed to support power gating of islands, there is an additional overhead on the dynamic power consumption of the NoC. To study the impact of the overhead and to see the impact of different core assignment to islands

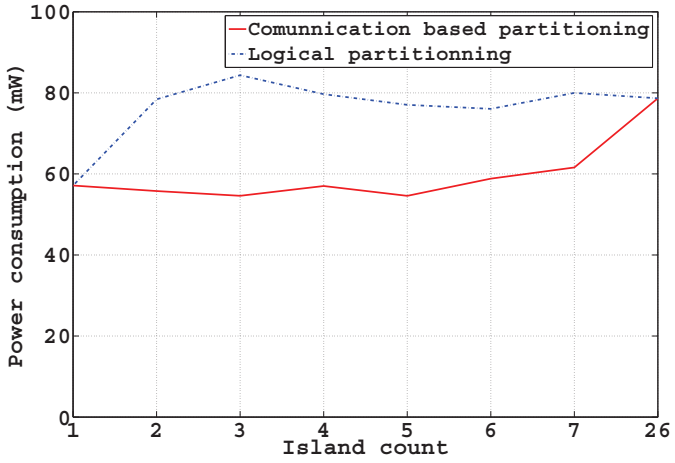

Figure 4.3: Impact of number of VFI on power

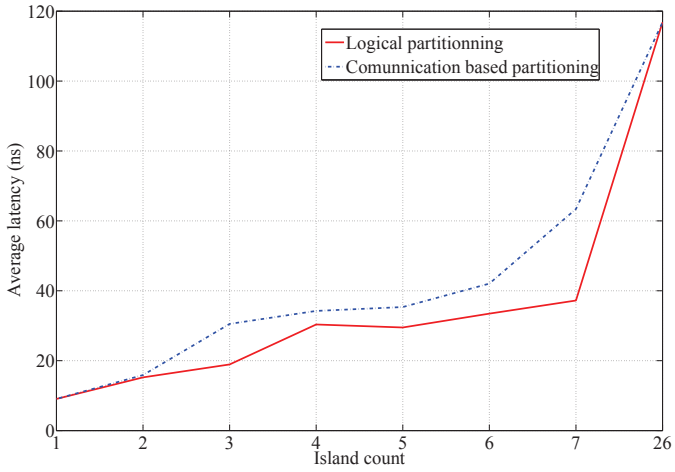

Figure 4.4: Impact of number of VFI on latency in ns

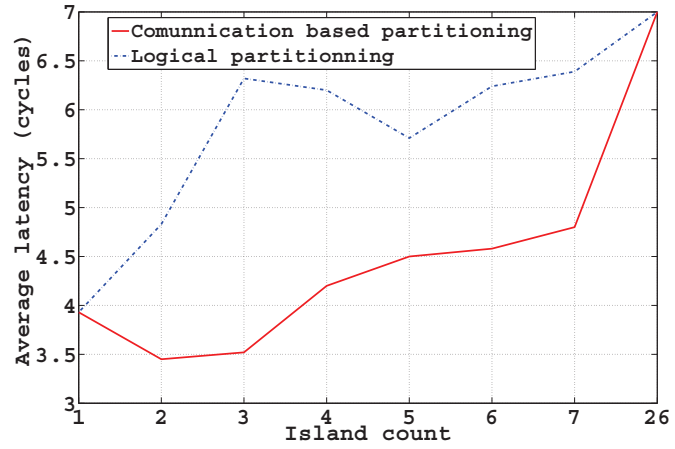

Figure 4.5: Impact of number of VFI on latency in $cycles$

and different number of islands, I consider a case-study on a realistic SoC benchmark *(D26_Media)*. The SoC design is used for mobile communication and multimedia applications. The communication graph of the benchmark is presented in Figure 3.10. The vertices in the communication graph represent actual physical cores and the weights on the edged represent the communication demands in MB/s between the physical cores. The benchmark has an ARM processor and a DSP, both having level 2 caches with controllers, several hardware accelerators, shared embedded memories and a multitude of peripherals. For the 3D design, I use 3 silicon layers to implement the benchmark. For the 3D case, the processor, the DSP and the hardware accelerators were assigned to the first layer, most of the memories to the second and the peripherals and remaining memories to the third layer. I use a data width of 32 bits for the NoC links for all the experiments, matching the data width of the cores.

I consider two ways of assigning the cores to different VFIs. One way, designated as *logical partitioning*, is based on the functionality of the cores. For example, shared memories are placed in the same VFI, as they have the same functionality and therefore are expected to operate at the same frequency and voltage. The island with the shared memories is also expected not to be shutdown, since memories are shared and should be accessible at any time and this is another reason to cluster them in the same VFI. Similar reasoning was used to partition all the cores for the case of *logical partitioning*. Another way I considered for partitioning is based on communication and is called *communication based partitioning*. In this case, cores that have high bandwidth communication with one another will be placed in the same VFI. Please note that the assignment of cores to the VFIs is an input to the synthesis algorithm.

In Figure 4.3, I show how the dynamic power consumption of the NoC varies when the cores are partitioned in to different number of VFIs. The power consumption values comprise the consumption on switches, links and the synchronizers. In the x-axis, the first point (1 island) is actually a design point with all the cores in the same island, which is the reference point. The last point on the graph corresponds to 26 VFIs, which is the point when each core is in its own island. It can be seen that in the case of *logical partitioning*, I have to pay a some overhead in NoC dynamic power, as there are more high bandwidth flows that have to go across islands. In the case of the *communication based partitioning*, the NoC consumes less power than the

	Energy ($\mu W/MHz$)	Area (μm^2)	Freq (MHz)
switch 4x4	7.2	10000	803
switch 5x5	8.4	14000	795
1 mm 32bit wire	2.72		
Converter	0.34	1944	1000

Table 4.1: NoC component figures

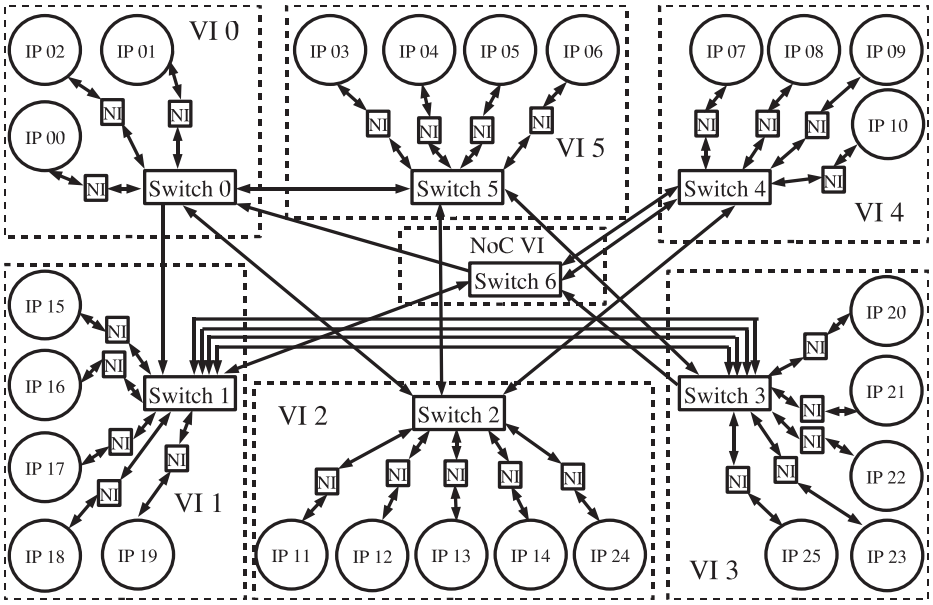

Figure 4.6: Topology example

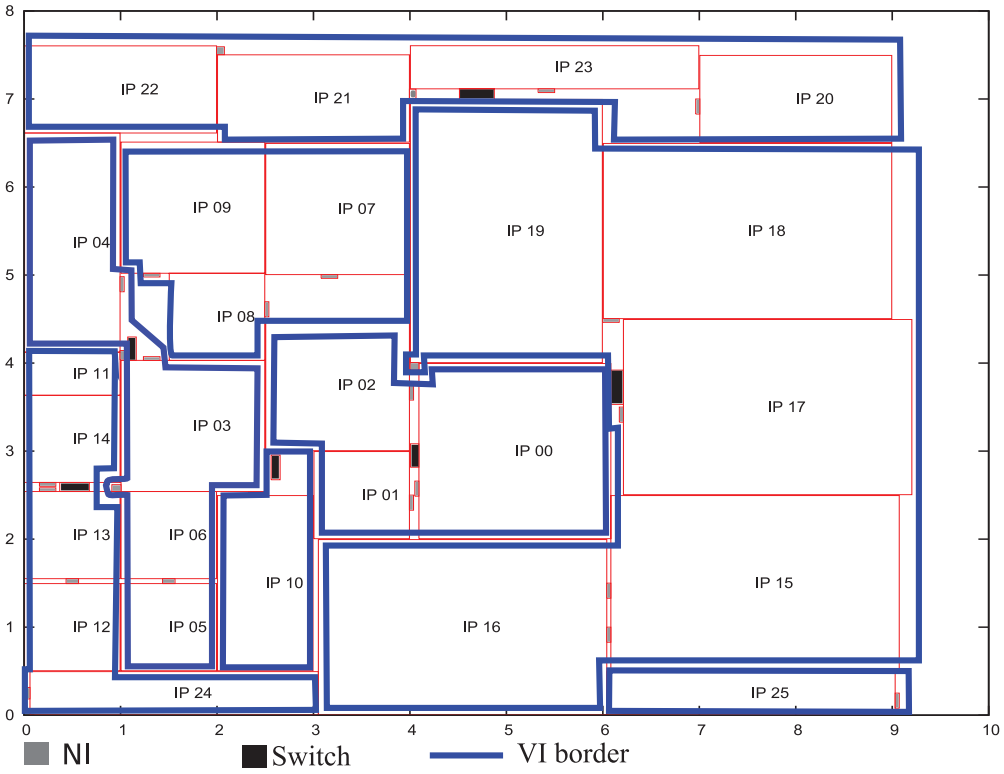

Figure 4.7: Floorplan example

reference point with 1 island, as the NoC can run at a slower frequency in some of the islands. In this case most of the high bandwidth flows are inside an island, so the power overhead is less. In Figure 4.4, I show the average packet latencies for the different design points. The latency quoted is the number of cycles needed to transfer a single chunk of the packet from the output of the source NI until the input of the destination NI under zero-load conditions. When packets cross the islands, a 4 cycle delay is incurred on the voltage-frequency converters. Thus, with increasing number of islands, the latencies increase. In Figure 4.4, I show the average packet latencies in cycles.

A topology for the 6 VFI *logic partitioning* case is shown in Figure 4.6 and a floorplan example is presented in Figure 4.7.

4.4.2 Using intermediate NoC VFI

The frequency at which a switch can operate is given by the critical path inside the switch. As the bandwidth requirements of the application start to increase, the required NoC frequencies also increase. Thus, the switches start reaching the maximum allowed sizes to meet the frequency requirements. One way to maintain the size of switches below the threshold is the use of indirect switches in the intermediate NoC VFI. However, there is a penalty in using these switches, as a flow going through them has to pass through one more set of voltage-frequency converters. In Figure 4.8, I show the number of indirect switches used for the best power points when the frequency is varied. I see that, when the bandwidth requirements are low, the intermediate island is never used. As the bandwidth requirements start scaling, more and more indirect switches in the intermediate VFI are used. The algorithm automatically explores the entire design space and instantiates the switches in the intermediate island when needed.

	No VFI		Multiple VFIs		
	Power (mW)	Latency (cycles)	Power (mW)	Latency (cycles)	VFIs
$D1$	273.3	4.10	435.5	6.31	6
$D2$	295.9	4.17	441.3	7.72	6
$D3$	448.5	5.76	561.8	7.71	6
$D4$	112.4	5.96	117.82	6.70	6
$D5$	332.9	3.25	341.64	3.40	8
$D6$	77.43	3.31	80.12	2.62	4

Table 4.2: Comparison on multiple benchmarks

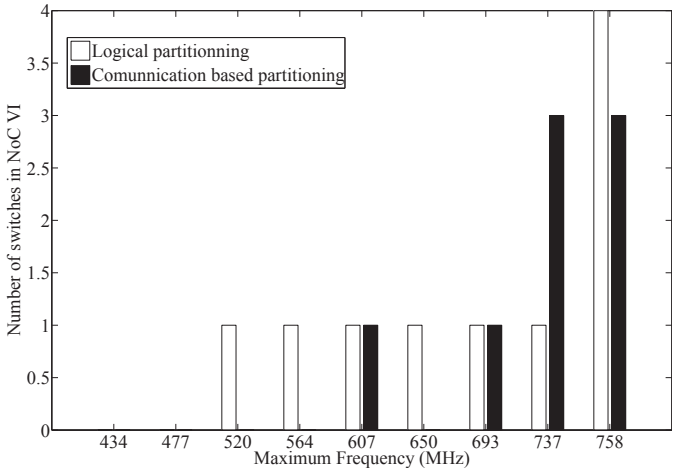

Figure 4.8: Impact of frequency on the switch count in the NoC VFI

4.4.3 Comparisons on different benchmarks

In Table 4.2, I present a comparison of power and latency between a design with no VFIs and a design with multiple VFIs for six benchmarks. Again, I report the dynamic power consumption values of the NoC. I use different number of islands in each benchmark (average of 6 islands), based on the logical characteristics of the applications. The $D1$, $D2$ and $D3$ set of benchmarks model systems with multiple shared memories on a chip. Each core communicates to 4, 6 and 8 other cores respectively with an average of 2, 3 and 4 communication flows going across the islands. In these cases, the overhead is more as there is a need for many voltage-frequency converters in order to channel all the inter-VFI communication. The $D4$ benchmark has 16 processor cores, 16 private memories and 3 shared memories. The processor and the private memory are assigned to the same VFI. Thus, there are just the low bandwidth flows going to the shared memories that have to go across VFIs. In this case, the power overhead for gating is not significant, only latency increases since some VFIs operate at a lower frequency. In case of the $D5$ and the $D6$, there are 65 cores and 38 cores respectively, communicating in a pipeline manner and therefore there are fewer links going across VFIs resulting again in a small overhead.

For the different SoC benchmarks, I find that the topologies synthesized to support multiple VFIs incur an overhead of 28% increase in the NoC dynamic power consumption. For all the benchmarks, the NoC consumes less than 10% of the total SoC dynamic power. Thus, the dynamic power overhead for supporting multiple VFIs in the NoC is less than 3% of the system dynamic power. I found that the area overhead is also negligible, with less than 0.5% increase in the total SoC area. In many SoCs, the shutdown of cores can lead to large reduction in leakage power, leading to even 25%

or more reduction in overall system power [51]. Thus, compared to the power savings achieved, the penalty incurred in the NoC design is negligible. Even though the packet latencies are higher when many VFIs are used, the synthesis approach provides only those design points that meet the latency constraints of the application. Moreover, the synthesis flow allows the designer to perform trade-offs between power, latency and the number of VFIs.

The exploration of the design points for all the benchmarks presented in this section takes only a few hours on a 2 GHz Linux machine. To be noted that the synthesis process is only run once at design time and therefor the computational time required by the algorithm is negligible.

4.4.4 Comparison of 2D and 3D topologies

To compare the power difference between a 2D and 3D design, I use the *(D26_Media)* realistic benchmark. I consider the ideal case in this section, when both the 2D design and the 3D design can be implemented in a fully synchronous manner. This experiment can be used as a baseline reference to put in perspective the following experiments where I have different number of VFIs. The frequency requirement on the NoC part of each VFI is determined by the core that has the largest bandwidth coming in or going out, as that bandwidth has to be supported by a single link. For the benchmark, for a single VFI, this minimum required frequency is calculated to be $270MHz$. The total NoC power consumption for the best power point for the 2D case is $38.5mW$ and for the 3D design is $30.9mW$. The power consumption of the 3D design is 20% lower when compared to the 2D design. Most of the power savings come from the fact that wires are shorter in the 3D design and therefore less power is required to drive them. Since all the cores are in the same VFI, there are no restrictions on the placement of the switches. This enables the placement strategy to better optimize the placement position of the switches and the there is only a 20% difference between the 2D case and the 3D case. I will show in the next section that as more constraints are added on the switch placement, even larger savings can be obtained when a 3D design is used.

4.4.5 Comparison for different number of VFIs

I assigned the cores to different number of islands (from 1 to 7) based on logic connectivity and application constraints. For example, in the case of the two islands, the processor, the DSP and the hardware accelerators were assigned to the same VFI and the memories and peripherals were assigned to the second VFI. An example of a topology for the 6 island case produced by my methods is presented in Figure 4.6 and the corresponding 2D floorplan is presented in Figure 4.7.

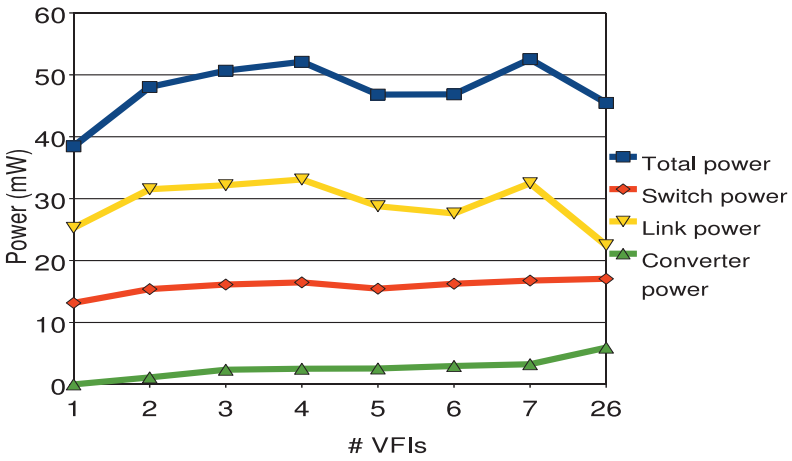

Figure 4.9: Power 2D designs

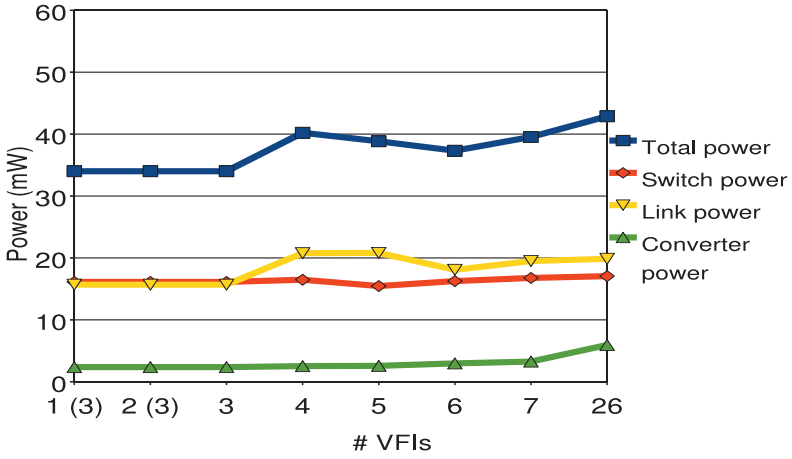

Figure 4.10: Power 3D designs

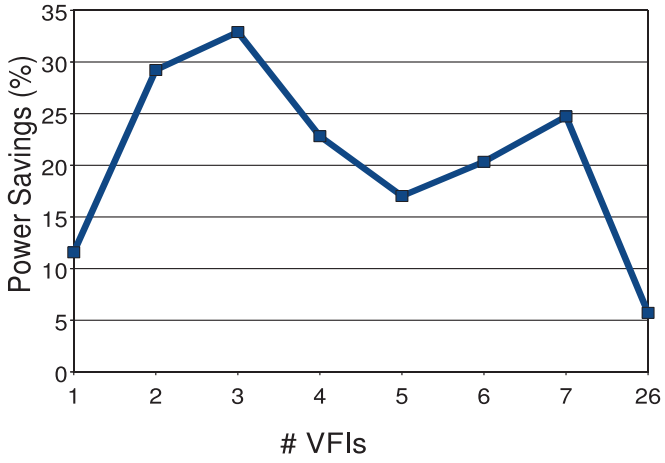

Figure 4.11: Power savings of 3D over 2D designs

As reference I consider the case where both the 2D and the 3D design are fully synchronous. The topology with the lowest power consumption in 2D uses 38.5mW and in 3D uses 30.9mW. The minimum frequency at which the NoC has to be operated to support the bandwidth requirements of the benchmark is 270MHz (power values are given for this frequency). The power consumption of the 3D design is 20% lower when compared to the 2D design. A complete analysis of 2D and 3D designs for fully synchronous designs is presented in chapter 3. I will show that with the extra constraints imposed by the VFIs even larger power savings can be obtained for the 3D designs.

The power consumption of the best power points for different number of VFIs for the 2D and 3D cases are shown in Figures 4.9 and 4.10. I show the total power consumption of the NoC as well as for the different components in the NoC (switches, links, converters). To be noted that the operation frequency in each VFI is calculated based on the bandwidth requirements in that VFI. For that reason, when more VFIs are added, some of them might operate at a lower frequency than required in a fully synchronous design. Even though increasing the number of VFIs implies adding more resources like switches, links and frequency converters, I can see that the power consumption does not go up significantly. This is due to the fact that with more VFIs in the design, a larger part of the NoC can be operated at lower frequency. The relative power savings of 3D compared to 2D are shown in Figure 4.11. In this experiment, I assume a clock skew across the different 3D layers even for a fully synchronous design, thereby leading a minimum of 3 VFIs for 3D, with one for each layer. Thus, the total power consumption plotted is the same for 1 to 3 VFIs in 3D. I obtain a maximum power saving for 3 VFIs. This is because, in 3D I have a minimum of 3 VFIs and as I increase the number of VFIs the converter power consumption increases and also wires in 2D are more segmented and shorter. The zero-load latency of the designs with different numbers of VFIs is presented in Figure 4.12. I can see that the latency goes up with the number of VFIs, because more links use frequency converters (which incur a 4 clock cycle penalty to traverse) and because more islands are operated at a lower frequency.

Apart from the total power consumption, I also show a breakdown on the power on the different NoC components. One important thing to note is that, as the number of VFIs increase, the operating frequency of certain VFIs can be lowered as the cores inside in a VFI may require less bandwidth than in another. Increasing the number of VFIs increases the number of switches in a design, as there has to be at least on switch in each VFI. However when combined with the previous effect that lowers the operating frequency, I can observe that the switch power does not have a significant increase when the number of switches is increased. A similar effect can be observed on the wire power for the cases of 3 to 7 islands in 2D, as the converters are placed closer to

the faster switch and the link is operated at the lower frequency. In 3D, however the switch to switch link power increases marginally with the number of islands, because of the increase in the number of switch to switch links. Since wire power is less in 3D due to shorter wires, the increase of the power consumption with the number of wires is more visible than for 2D topologies.

In Figure 4.11, I show the power savings in 3D compared to 2D. In this experiment, I assume a clock skew across the different 3D layers even for a fully synchronous design, thereby leading a minimum of 3 VFIs for 3D, with one for each layer. Thus, the total power consumption plotted is the same for 1 to 3 VFIs in 3D. As I increase the number of VFIs beyond 3, the converter power becomes significant, so the difference between 2D and 3D reduces. Also as the number of VFIs is increased, the wires in 2D become more segmented and shorter as well, further decreasing the power difference between the 2D and 3D designs. The zero load latency is less influenced by the wire length difference between 2D and 3D, as can be seen in Figure 4.12. The numbers plotted are the latency for a head flit of a packet to reach from the source network interface to the destination network interface. This is because, a wire has to be significantly long (7mm in this technology library) to require an additional pipeline stage. Most of the wires, also in 2D, fall below this threshold for the benchmark. Most of the delay is incurred in crossing the frequency converters, as each converter incurs a 4-cycle penalty (of the slowest clock). As the number of VFIs is increased more of the wires become inter VFI wires requiring a frequency converter (increasing the latency to cross the wire to 4 cycle). Thus, with increasing number of VFIs, the delay shoots up exponentially for both 2D and 3D systems.

In Figure 4.13, I show the power savings obtained on two other benchmarks, with different communication patterns than the multi-media system considered above. The *D35_bott* benchmark has 16 processors with 16 private memories and 3 shared memories. Most of the high bandwidth traffic is between the processors and their private memories. On the other hand the *D36_8* is a benchmark with spread traffic pattern, with each core communicating with 8 others with equal bandwidth values. As expected, these two benchmarks represent two extremes, the former providing low power savings, while the latter providing large power savings for 3D. As a reference, the topologies for *D35_bott* with 3 VFIs consume $80mW$ in 3D and $88mW$ in 2D. The previously analyzed benchmark (the *D26_Media*) is a realistic benchmark and the power savings are in between these two benchmark. Also, the power savings depend on the number of VFIs.

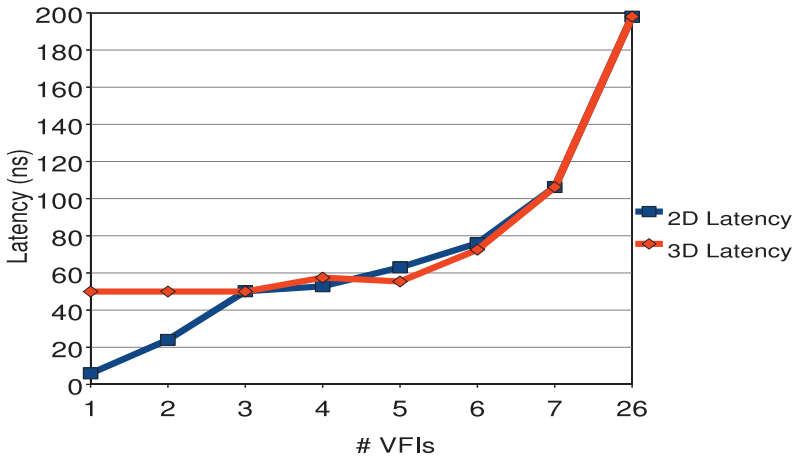

Figure 4.12: Average zero load latency of 2D and 3D designs

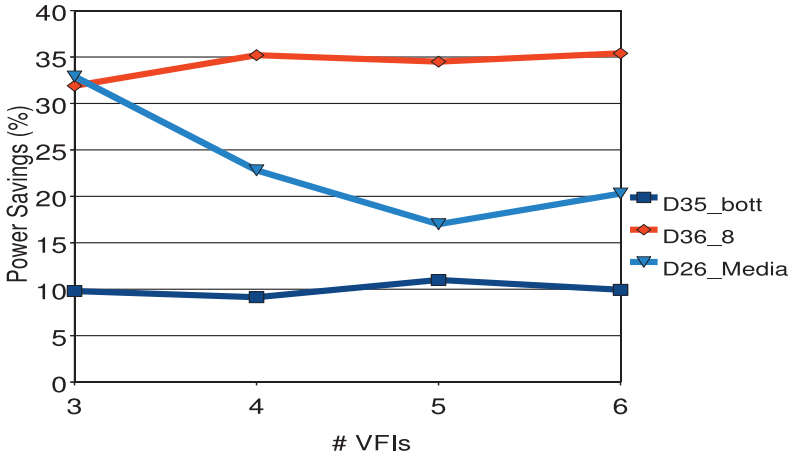

Figure 4.13: Power savings of 3D vs. 2D for different benchmarks

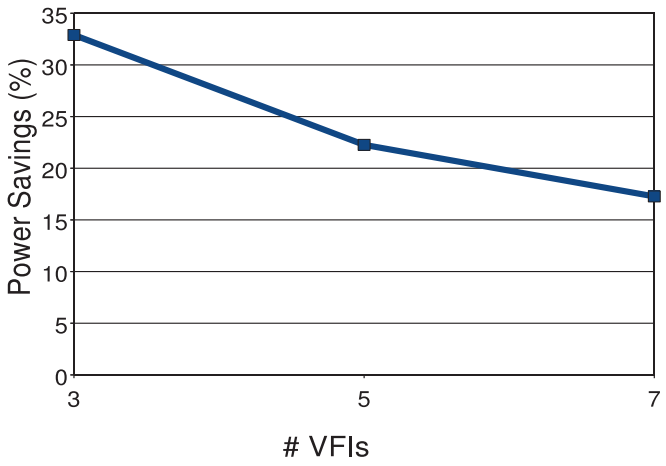

Figure 4.14: Power savings of 3D vs. 2D for different core areas

4.4.6 Comparison with different design sizes

The number of VFIs can be decided according to the design in order to achieve low power consumption by shutdown of unused VFIs. However, it can also be forced by the technology when the area of the design becomes large enough that a single clock tree cannot be designed to synchronize all the components. I performed experiments to explore the effect of increasing the number of VFIs due to an increase in the size of the design. I scale the size of the design proportional on the number of VFIs. For example, a design with 5 VFIs has an area increase of 60% over the one with 3 VFIs. The difference in power consumption between the 3D and 2D designs in percentage is shown in Figure 4.14. As the wires are longer when the benchmark is larger, I obtain more power savings in 3D, when compared to the experiments in the previous subsection.

4.4.7 Analysis of results

When the whole design is synchronous, 3D designs give very low power savings (11%), as frequency converters or mesochronous synchronizers are needed to tolerate clock skew across layers. As the number of VFIs increase, 3D SoCs have large NoC power reductions (up to 32%) due to reduction in wire lengths. However, after a sweet-spot, the gains fall again, as the wires get shorter in 2D due to the use of more switches and the contribution of converter power to overall power also becomes significant. The results show the need for an early architectural exploration of the whole design space, as the number of VFIs used play a major role in determining the power savings achieved when migrating to 3D. The experiments also show that the reduction in delay is not very significant when moving from 2D to 3D systems (up to 11%), if both are designed efficiently. This is because, the number of links in 2D that are long enough to require pipelining is less and the synchronizer delay is more dominant when compared to wire delay. The area overhead due to the insertion of TSVs in 3D is negligible, as the TSV macros occupy less than 2% area when compared to the area of the cores.

4.5 Summary

Standby and leakage power consumption of the SoC is becoming a large fraction of the total power consumption. Clustering of cores in to voltage islands and shutdown of unused islands is an effective way to reduce the leakage power consumption. The system interconnect has to be designed to ensure proper operation when shutting down voltage islands. In this chapter, I presented an approach to synthesize application specific *Networks on Chip (NoC)* interconnects that can effectively support

shutdown of islands. The topologies synthesized by my methods have negligible power and area overhead (3% power and 0.5% area, on average), in order to support shutdown. The presented approach also allows the design space exploration of NoCs with different power-performance values that meet the application constraints.I also presented a detailed comparison of NoCs for 2D and 3D, using a realistic mobile platform. I showed that, as the number of VFIs increase, 3D SoCs have large NoC power reductions (up to 32%) due to reduction in wire lengths. After a certain number of VFIs, the gains fall again due to shorter wires in 2D and more significant converter power consumption in both cases. The experiments also show that the reduction in delay is minimal when moving from 2D to 3D systems (up to 11%) because the converter delay is more dominant than wire delay.

5 Removing deadlocks in application specific topologies

To achieve high throughput and low latency, most NoCs use wormhole flow control. A major problem that arises when using wormhole flow control is the possibility of deadlocks. In wormhole flow control, as opposed to store and forward, packets are typically larger than the buffering capacity of the switches. When a packet is in flight in the network, it might span across multiple switches. The packets are split into multiple *FLow control unITs (FLITs)*. The first flit of the packet, the *header*, contains the routing information. Once the header flit reserves the output port of a switch, that port remains reserved until the last flit of the packet passes through. If a data packet has to use an output port of a switch which is already reserved to another packet, it has to wait until the packet holding the port goes through and releases the port. It is possible for a set of packets to have a cyclic dependency of resources, where each of them waits on a port held by another and none of them can advance. Such a scenario is called as a routing-level *deadlock* [41].

Deadlocks in a network can block communication between cores and can even lead to a complete network failure. Therefore it is essential to have methods to handle them. One way is to prevent the conditions that lead to a deadlock, by routing packets in such a way that the circular waiting patterns cannot occur. The other way is to detect and recover from them. Because detecting and recovering from deadlocks require specialized hardware resources, methods to prevent deadlocks are widely used in NoCs [42].

In many SoCs, the cores are heterogeneous in nature and the communication patterns are well defined. The NoC topologies and routing functions are custom designed to match the application specifications, leading to more power/performance efficient designs. For regular network topologies, deadlocks can be avoided by restricting the routing function, so that certain turns in the network are prohibited. In [144], [179], [159], methods to prohibit turns on irregular custom topologies to avoid deadlocks are presented. However, the methods have to be used during the construction of the

NoC topology, otherwise connectivity between cores cannot be guaranteed. In many cases, the topology of the interconnect is decided earlier in the design process, but as the design matures the application patterns may change. Thus, the deadlock freedom or connectivity cannot be guaranteed. Moreover, they place a severe restriction on the routes that can be used in the topology.

Most of the existing manual and automated topology synthesis methods [157] use additional virtual channels (VC) or physical links to remove deadlocks in the design. *Resource ordering* is a popular method to prevent deadlocks in custom topologies [41]. In this method, the communication channels (physical links or VCs) are assigned to different classes that are ordered. Each time a packet needs to traverse a switch, it will reserve a channel from a resource class of higher order than the one it currently occupies. In this way, no cyclic dependency between packets can be formed. However, a major disadvantage of the resource ordering method is that it incurs a large area-power overhead to remove the deadlocks, as a large number of physical or virtual channels are needed. In other works [121], [62], as well as in the methods from Chapter 3, turn-prohibition is used during the synthesis process to prevent deadlocks. While the solutions from [121] and [62] are proposed for designing 2D SoCs, turn prohibition is efficient at preventing deadlocks. However when considering 3D integration with a very tight TSV constraint, the method from Chapter 3 may fail as turn-prohibition may require too many TSV links.

Achieving deadlock free operation of custom topologies with minimal area-power overhead is a major challenge. In this chapter, I present a new method to remove deadlocks in custom NoC topologies. My method adds a minimal number of virtual channels (VCs) or physical channels to remove deadlocks, thereby incurring a very low area-power overhead. Unlike many existing methods, the proposed method can be applied on any topology and does not impose any restrictions on the routing functions that can be used. The major contribution of this chapter are:

- I present an algorithm to find and remove deadlock conditions by minimally adding VCs or physical links. Please note that the algorithm works the same whether VCs or physical links are added, so for simplicity I will only mention VCs when describing the algorithm.
- I present an evaluation of the algorithm using VCs as well as physical links.
- I show the benefits of using the algorithm to remove deadlocks after designing the topology for 3D-ICs with tight TSV constraints.

I perform experiments on several SoC benchmarks and show that the method results in a large reduction in the number of VCs needed to remove deadlocks (an average

of 88%) when compared to the resource ordering method. This translates to a large reduction in NoC area (an average of 66%) and power consumption (an average of 8.6%). In case of adding physical links the power reduction in power consumption is even larger (on average 10%). The experiments also show that the method is practical, as the total area and power overheads to remove deadlocks are less than 5% when compared to a design with no mechanism to remove deadlocks.

5.1 Background on deadlock avoidance techniques in NoCs

In [43], a necessary condition for adaptive routing functions to be deadlock free is given. In [44], the authors prove that the condition from [43] is also sufficient for the adaptive routing algorithm to be deadlock free. Deadlock free routing algorithms for different regular topologies are presented and analyzed in [34], [38], [98], [55], [105]. However, most of these methods are not applicable to custom irregular topologies.

A necessary and sufficient condition for deadlocks to appear in networks with wormhole routing is given in [41]. A survey of the deadlock free routing algorithms developed for networks with wormhole routing is presented in [125]. The method also uses the necessary and sufficient condition from [41] to detect the deadlocks.

Turn prohibition methods to remove deadlocks have been presented in [56], [179], [159]. In these methods, a certain number of turns are prohibited to be used by the routing function. The advantage is that they can be used with irregular topologies. However they have to be applied during the topology synthesis and cannot guarantee that routes can be found for existing non-deadlock free topologies. In [128], a methodology to design adaptive-deadlock free routing functions for application-specific traffic patterns is presented. The method is based on the theory from [44], but customizes the resulting routing function according to the description of the actual traffic patterns of the application. In [121], [62], the authors use the turn-prohibition method to remove deadlocks. However, all these methods are integrated with the topology synthesis process and cannot be applied to arbitrary NoC topologies and routing functions.

The most popular method to remove deadlocks in arbitrary NoC topologies with any routing function is the resource ordering method [41]. As I show in the experiments, my method results in much smaller area-power overhead when compared to this scheme.

5.2 Problem formulation

In Example 5.1, I show how a waiting pattern that can occur can lead to a deadlock.

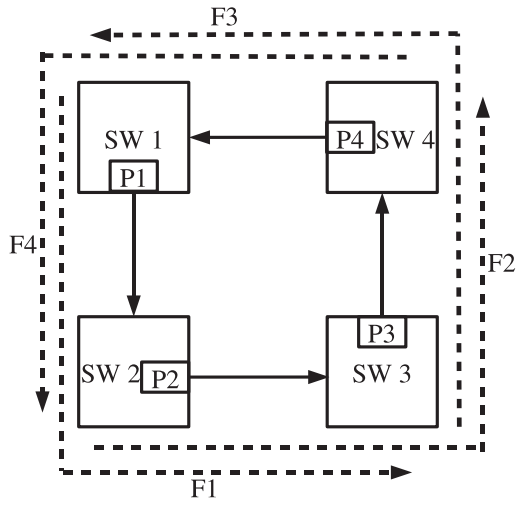

Figure 5.1: Deadlock example

Example 5.1 *In Figure 5.1, I show an example of a deadlock. Suppose a packet F1 tries to go from switch 1 to switch 3 and it holds the port P1 and waits for port P2 which is reserved by packet F2. F2 cannot advance as the needed port P3 is reserved by packet F3. F3 is blocked waiting for port P4 of switch 4 which in turn is used by packet F4 which waits for port P1 reserved by F1. In such a case none of the flows can advance and are blocked indefinitely. Any configuration in a network where packets wait on one another in a circular manner leads to permanent blocking and is called a deadlock.*

An input topology is represented as a graph, defined as follows:

Definition 5.1 *A Topology Graph $TG(S,L)$ is a directed graph where the vertexes $s_i \in S$, $i = 1 \ldots N$ represent the switches and N is the number of switches in the topology and the edged $l(s_i, s_j) \in L$ represents a physical link between switch s_i and switch s_j.*

I also need as input the description of the communication flows. The communication graph is defined in a similar manner as in Chapter 3, but it is simplified as some information is not needed here:

Definition 5.2 *The communication graph is a directed graph, $G(V,E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j) representing the communication flow between the cores v_i and v_j.*

For each communication flow, a route has to be defined which describes which of the links in the topology are used by a particular flow to reach from source to destination.

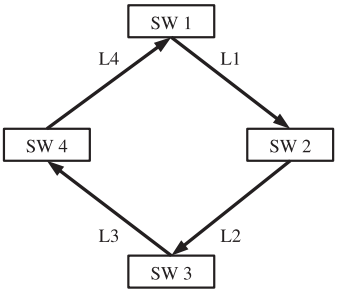

Figure 5.2: Topology example

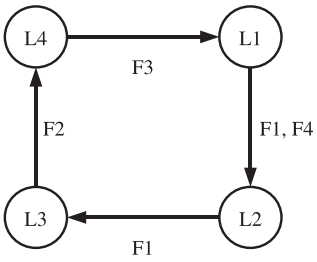

Figure 5.3: CDG example

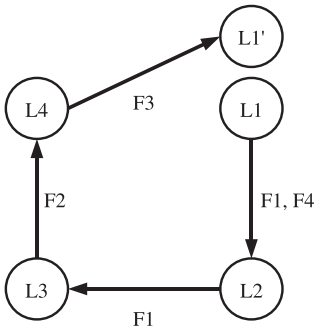

Figure 5.4: Modified CDG

I also take the description of the routes as input. A route is defined as follows:

Definition 5.3 *A Route is a set of channels (a physical link and the corresponding VC) $R = \{l_{1,0}, \ldots l_{n,k}\}$ where the channel $l_{i,j}$ uses the physical link $l_i \in L$ and the VC j. The route defines the channels that a flow will use to reach from source to destination. The order of the channels in the set is also the order in which the channels will be traversed by a packet in the network.*

From the *topology graph* $TG(S,L)$, the *communication graph* $G(V,E)$ and the set of all routes R_k corresponding to flows in $G(V,E)$ the algorithm will build the *Channel Dependency Graph (CDG)* which is used to find the possible deadlock conditions, and then to remove them. It is defined as follows:

Definition 5.4 *The Channel Dependency Graph is a directed graph $CDG(C,D)$, with each vertex $c_i \in C$ represents a channel in the topology (a physical link and the corresponding VC) and an edge $d(c_i, c_j)$ represents a dependency between the two channels. A dependency is given when there is at least one route which used channel c_i and then immediately channel c_j.*

Example 5.2 *In Figure 5.2 an example of a topology is shown. There are four switches in the topology {SW1, SW2, SW3, SW4} connected by four channels {L1, L2, L3, L4} to form a ring. I consider four communication flows F1, F2, F3 and F4 that have the following routes R1 = {L1, L2, L3}, R2 = {L3, L4} , R3 = {L4, L1} and R4 = {L1, L2}. For the topology in Figure 5.2 and the four flows, I show the CDG from Figure 5.3. The CDG has four vertices one for each channel in the topology. Flow F1 generates the edges between vertex L1 and vertex L2 and between L2 and L3, flow F2 generates the edge between*

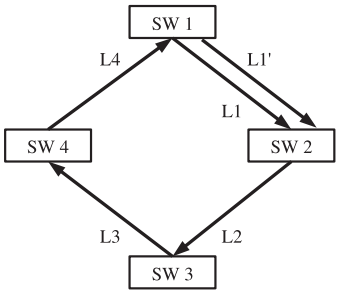

Figure 5.5: Modified Topology

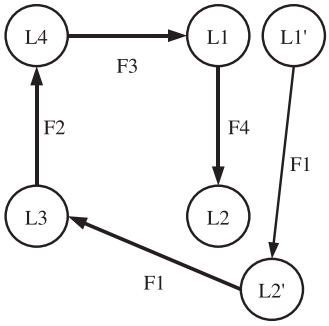

Figure 5.6: Break in forward direction

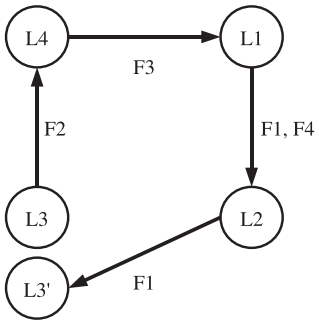

Figure 5.7: Break in backward direction

L3 and L4, flow F3 generates the edge between L3 and L4 and the flow F4 comes as the second flow to require an edge between L1 and L2.

In [41] the authors have shown that a necessary condition for a deadlock to occur in a network with wormhole routing is to have a cycle in the CDG. Cycles in the CDG can be broken by adding new vertices to the CDG and removing one or more of the edges between the vertices involved in the cycle. The corresponding operations in the network for adding vertices to the CDG is to add new VCs to create new channels between switches and to remove edges in the CDG corresponds to modifying the routes of some of the flows. In Example 5.3, I show one possible way to break a cycle and the corresponding changes in the topology.

Example 5.3 *The network with the topology from Figure 5.2 can have deadlocks because the corresponding CDG from Figure 5.3 has a cycle. To prevent deadlocks, we have to break the cycle in the CDG. We can break the cycle in the CDG by adding a new vertex L1' in the CDG and modifying the route of the flow F3 to use L1' instead of L1. The resulting acyclic CDG is presented in Figure 5.4 and the corresponding modified topology is shown in Figure 5.5. The new topology has an extra channel L1' between switch SW 1 and switch SW 2 and the route for flow F3 is now R3 = {L4, L1'}.*

The problem is formulated in following way: given a topology graph $TG(S,L)$, the communication graph $G(V,E)$, the set of all routes R_k and the corresponding $CDG(C,D)$ with $\Phi \neq \emptyset$, how to get $TG'(S,L')$ and R'_k such that the corresponding $CDG'(C',D')$ will have $\Phi' = \emptyset$ and $|L'| - |L|$ is minimal.

I present an algorithm that removes all deadlock conditions, by removing all cycles in the CDG generated for the network. A cycle is removed by adding one or more VCs in the topology (or vertices in the corresponding CDG) and re-routing some of the

Algorithm 5.1 Deadlock removal

1: {Initialize CDG using Topology and Routes}
2: Build $CDG(C,D)$ from $TG(S,L)$ and R_k $\forall k \in [1 \ldots |G(V,E)|]$
3: $C = GetSmallestCycle()$
4: **while** $C \neq \emptyset$ **do**
5: $\langle f\_cost, f\_pos \rangle = FindDependencyToBreakForward(C, \text{flows})$
6: $\langle b\_cost, b\_pos \rangle = FindDependencyToBreakBackward(C, \text{flows})$
7: **if** $f\_cost \leq b\_cost$ **then**
8: $BreakCycleForward(C, f\_cost, f\_pos)$
9: **else**
10: $BreakCycleBackward(C, b\_cost, b\_pos)$
11: **end if**
12: Update $TG(S,L)$ and R_k $\forall k \in [1 \ldots |G(V,E)|]$ from the current $CDG(C,D)$
13: C = $GetSmallestCycle()$
14: **end while**

flows on the newly created channels. The cycles are removed one by one starting from the smallest one. The algorithm terminates when all cycles are removed. The goal of the algorithm is to remove all deadlock conditions by adding the minimum number of extra VCs. The detailed explanation of all the steps is provided in Section 5.3. The output of the proposed algorithm is a minimally modified topology graph and route set, with the resulting design being deadlock free.

5.3 Deadlock removal approach

In Algorithm 5.1 the major steps of my method are presented. Given the topology graph, the communication graph and the description of the routes as input, in Step 2 of the algorithm the CDG is build for the current configuration of the network. In Step 3, I run an initial search to find the smallest cycle in the CDG. I use the heuristic of breaking the smallest cycle first, as it can also lead to breaking a larger cycle sharing some of the edges with this one. If no cycles are found in the initial search, the algorithm ends as the initial topology is deadlock free and no modifications are needed.

To find the cycles in the CDG, I run *breadth first search* starting search from every vertex of the graph. If the starting vertex is encountered during the search then a cycle is detected in the CDG and the starting vertex is part of a cycle. By running the search from every node, I find all cycles and the procedure $GetSmallestCycle$ returns the one that has the smallest length. The time complexity of the $GetSmallestCycle$ procedure for a given graph $CDG(C,D)$ is $O(C^2 \times log(D))$. Please note that more advanced methods for finding the cycles like the one presented in [173] or [78] can be used. The second referenced method has a time complexity of $O((C+D) \times (\phi+1))$

where ϕ is the number of cycles. However since the CDG most of the time is sparsely connected, I found that the simple method used runs fast enough.

In the simple case, in order to break a cycle, I need to remove an edge between any two vertices in the cycle by duplicating the vertex before or after the edge. Suppose, we duplicate the vertex after the edge, we connect the edge to the new vertex instead of the original one and the cycle can be broken. The new vertex added in the CDG corresponds to a VC that has to be added to a link in the topology graph. Connecting the edge in the CDG to the new vertex corresponds to a change in the route of the flows using the old VC of the link in the topology to use the newly added VC. Earlier, in example 5.3, I showed how a cycle can be broken by adding an extra vertex in the CDG.

However, in a general case, to break a cycle, more than one vertex may need to be duplicated. For example, in Figure 5.8, I show how adding an extra vertex still maintains the cyclic dependency. In order to break a cycle at an edge in the CDG, several vertices before or after the edge need to be duplicated. The number of vertices that need to be duplicated to remove an edge depends on the configuration of the flows relative to the vertices in the cycle and different edges can require significantly different numbers of vertices to be duplicated. It is important to choose the edge to remove from the cycle that minimizes the number of vertices to be duplicated. This problem is addressed in detail in Section 5.3.1.

Also, there are two ways in which the vertices can be replicated, relative to the position of the edge that is removed from the cycle. I say that an edge is removed in *forward direction*, if vertices are duplicated from where a flow enters the cycle up to the edge. I say that an edge is removed in *backward direction* if the vertices are duplicated from the edge to where the flow causing the edge exits the cycle. Removing an edge in the two possible directions is shown in Example 5.4. It can be seen from the example that the direction can have an impact on the number of vertices that are duplicated. Therefore, the algorithm checks the cost of breaking the cycle in both directions, using the procedures $FindDependencyToBreakForward$ and $FindDependencyToBreakBackward$. The procedures returns the edge that has the minimum cost for each of the directions. The way the cost and the position are calculated is explained in more detail in Section 5.3.1

Example 5.4 *In Figures 5.6 and 5.7, I show the two ways to remove the same edge between the vertices L2 and L3 of the CDG from Figure 5.3. The edge between the vertices L2 and L3 is created by flow F1. In Figure 5.6 the edge is removed in forward direction by duplicating the vertices from where flow F1 enters the cycle up to the edge that needs to be removed. In this case we need to duplicate L1 and L2 and we add L1' and L2' to the CDG. Then we modify the path of F1 to use L1' and L2'. In Figure 5.7 the*

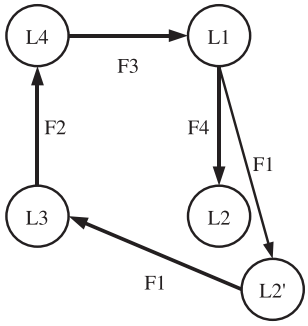

Figure 5.8: CDG with new vertex and cycle

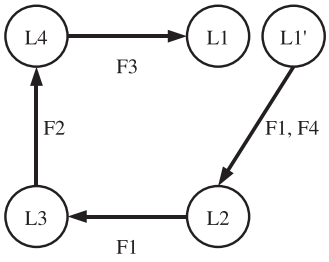

Figure 5.9: Remove edge between L1 and L2

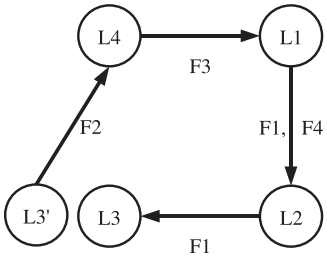

Figure 5.10: Remove edge between L3 and L4

	D_1	D_2	D_3	D_4
F1	1	2	0	0
F2	0	0	1	0
F3	0	0	0	1
F4	1	0	0	0
MAX	1	2	1	1

Table 5.1: Cost table in forward direction

edge is removed in backward direction by duplicating the vertices from where flow F1 exits the cycle down to the edge that needs to be removed. In this case we only need to duplicate the vertex L3 as the flow F1 exits the cycle at L3. We add the new node L3' to the CDG and set the path of F1 to use L3' instead of L3. As it can be seen from this example the direction in which an edge is removed can influence the number of extra VCs needed to break the cycles in the CDG.

In Step 7, the cost of breaking in the forward direction is compared with the cost of breaking in the backward direction and depending on which cost is smaller, one of the two procedures is called: $BreakCycleForward$ or $BreakCycleBackward$. The way the two procedures work is detailed in Section 5.3.2. Once the CDG is changed, in the next step the topology graph and the routes are updated accordingly. After the current cycle is broken in Step 13, the algorithm searches for the next smallest cycle in the updated CDG. The algorithm terminates when the CDG becomes acyclic.

5.3.1 Finding the edge to remove

Depending on which edge I want to break the cycle, number of vertices that need to duplicated in the CDG (and hence the number of new VCs to be added in the topology graph) can differ. Therefore it is important to find the edge that can be removed with

Algorithm 5.2 $FindDependencyToBreakForward(C,$ flows)

```
1: {Find flows taking part in cycle C}
2: F = Ø
3: for all i ∈ flows do
4:    if | path(i) ∩C| > 1 then
5:       F = F ∪ {i}
6:    end if
7: end for
8: {Calculate costs for each flow in cycle C}
9: for k = 1...|F| do
10:   cost(i)(k) = 0, ∀i ∈ C
11:   val = 1;
12:   for i = 1...|path(F_k)| do
13:      current_vertex = path(F_k)(i)
14:      if current_vertex ∈ C then
15:         cost(current_vertex)(k) = val;
16:         val = val + 1
17:      end if
18:   end for
19: end for
20: cost(i) = max(cost(i)(k)) ∀i ∈ C and ∀k ∈ 1...|F|
21: f_cost = min(cost(i)) ∀i ∈ C
22: f_pos = argmin(cost(i)) ∀i ∈ C
23: Return ‹f_cost, f_pos›
```

the least number of replication of vertices. The steps to find the best edge to break are described in Algorithm 5.2. The procedure describes how the costs are calculated and how the edge to break the cycle is found when considering breaking the cycle in the forward direction.

The algorithm starts by finding all the flows that create dependencies at each edge that is involved in the cycle. This is necessary because in order to remove one edge, the flows that are creating that dependency have to be routed on new channels that will be added. In steps 3 to 7 of the procedure, all flows in the design are checked to see if they are part of a dependency on one or more edges in the cycle. All flows that have paths that intersect the cycle are added to a new set which will be taken into consideration when breaking the cycle. The path of a flow is a set of vertices from the CDG that correspond to the set of channels from the route of the flow. The order of the vertices in the path correspond to the order of the channels in the route. In order to break a dependency created by a flow, all the channels used by the flow in the cycle prior to the dependency have to be duplicated. For example for the CDG in Figure 5.3, I show how to remove any of the four edges in the forward direction in the Figures 5.9, 5.6, 5.10 and 5.4.

In the Steps from 9 to 19, I calculate the cost of breaking the cycle considering the effects of each of the flows individually. For each of the flows, I calculate the cost in the following way: I find where the flow enters the cycle, this gives the first vertex of the cycle that is used by the current flow. To break the dependency between this first vertex and the next vertex in the cycle, the cost is one as I only have to duplicate this first vertex. If I want to break the dependency at the next edge on the cycle, two vertices need to be duplicated and the cost for this is two. I continue in a similar manner until the flow leaves the cycle. In the end, I build a table that has as many rows as there are flows involved in the cycle and has as many columns as there are edges in the cycle. In Example 5.5, I show such a cost table.

Example 5.5 *An example of a cost table for breaking the cycle in the forward direction for the CDG in Figure 5.3 is given in Table 5.1. Flow F1 has a cost of 1 on the D_1(L1, L2) as only L1 has to be duplicated to break the cycle when considering the influence of F1 alone. On the D_2(L2, L3) edge it has a cost of 2 as both L1 and L2 have to be duplicated as shown in Figure 5.6. On the other edges the cost for F1 is 0 as it has no influence on them. In a similar manner the cost for F2, F3 and F4 are computed. The last row of the table shows the compound cost of all flows. For the example, edges D1, D3, and D4 can be removed by adding 1 VC and D2 needs 2 VCs.*

The cost table gives the effect of each flow independently, so now I have to take into account the combined effect of all the flows in the cycle. For that, I calculate the combined effect at each dependency in Step 20. I take the maximum cost between all flows because it tells how many vertices have to be replicated. All flows that have smaller cost can use a subset of the vertices replicated for the flow with the highest cost. Once I have the cost that keeps track of the combined effect of all flows, I choose the dependency to break the cycle so that it has the minimum combined cost. The minimum cost and the position in the cycle of the dependency to break are returned by the function in Algorithm 5.2.

In a similar manner the function that calculates the costs in the backward direction can be obtained. In Steps 12 to 18, I look at the paths of the flows from source to destination. When calculating the costs for the backward direction, I analyze the paths in reverse order from destination to source. This is because when I break a cycle in backward direction, I replicate the vertices starting from where the dependency that I want to break is to where the flow leaves the cycle. When I consider the combined effect of all flows the minimum cost in the forward direction can differ from the minimum cost in the backward direction and for that reason I calculate in both directions and then break the cycle in the direction that has the least cost.

Algorithm 5.3 $BreakCycleForward(C, f\_cost, f\_pos)$

```
1: pos = f_pos
2: for i = 1...f_cost do
3:    pos = pos mod |C|
4:    CDG = CDG ∪ C_pos'
5:    for i = 1...|F| do
6:       if path(F_i) ∩ C_pos ≠ Ø and path(F_i) ∩ C_f_pos ≠ Ø then
7:          swap(C_pos, C_pos') in path(F_i)
8:       end if
9:    end for
10:   pos = pos - 1
11: end for
```

5.3.2 Breaking a cycle at the edge

Given a cycle, the position of the dependency in the cycle that has to be removed and the cost of removing the dependency, the cycle can be broken by duplicating vertices as described in Algorithm 5.3. The cost already indicates the number of vertices that need to be duplicated. For breaking the cycle in the forward direction, the vertices are duplicated from the edge that is to be removed (output from the Algorithm 5.2) till the vertex where the flow with the largest cost enters the cycle.

For each iteration of the loop at Step 2, the following actions are performed: first the position of the vertex in the cycle is calculated. Then, a new vertex is created and added to the CDG. Then in the Steps from 5 to 9, the paths of the flows that create the dependency that I am removing are modified to use the newly created vertices. The test in Step 6 looks if the flow uses the vertex that I have currently duplicated and also the vertex from which the dependency which I am removing starts. If the test succeeds then the path of the flow has to be modified to use the new vertex. Before completing the loop, the position of the current vertex is decremented. In a similar manner the cycle is broken in the reverse direction as well.

5.3.3 Marginal power as cost

By computing the cost for removing an edge in the cycle, as presented in Section 5.3.3, I minimize the number of extra links added in the topology. This in turn minimizes the switch area overhead. However when designing application specific NoCs for SoCs, minimizing power consumption could be an important criteria. The relationship between the power consumption of a switch and the size of the switch is non linear and there can be cases where it is more power efficient to remove a deadlock by adding two links between two smaller switches than one link between two large switches.

To use power as the optimization criteria when removing deadlocks, a simple modification has to be made in Algorithm 5.2 at Step 16. In the original algorithm the temporary cost denoted by the variable val is incremented by 1 each time a new link is added. Since an unit cost is used for every new link, the algorithm will minimize the number of extra links added in the topology graph. However if instead of the unit cost, the marginal power consumption for adding a new link is used as cost, the result is that the total switch power consumption is minimized. The marginal power consumption is calculated as the increase in power consumption by adding a new output port at the source switch and by adding a new input port at the destination switch. The power for the switches of different size can be obtained by using parametric power models like those used in [79].

5.4 Experimental results

In the experiments I generated application specific topologies for different switch counts on several realistic SoC benchmarks using an existing tool [121]. Please note that the deadlock removal method is general and the input topologies could be either manually designed or obtained using any existing synthesis tools.

5.4.1 Comparison with resource ordering

For comparisons, I also apply the *Resource ordering* technique, a popular method to avoid deadlocks in wormhole routing networks [41]. In this method the communication channels are given a resource number. After a flow uses a channel, the next channel that it acquires needs to have a resource number higher than the current channel.

As a case study, I consider a SoC design for multimedia and wireless applications, referred to as $D26\_media$. It has 26 cores, including an ARM processor, a DSP with L2 cache and cache controllers, several memory blocks and a multitude of peripherals. I use this benchmark to generate topologies for different switch counts and then apply my algorithm and resource ordering to remove deadlocks from the topologies. The results are presented in Figure 5.11. The dotted line in the plot represents the number of extra VCs that were added to the topology in order to generate the necessary number of resource classes required to use resource ordering on the routes of all the flows. The number of classes needed for a flow depends on the length of the route and that leads to considerable overhead. The solid line represents the overhead of my method. As it can be seen, for most topologies the overhead is zero because the fixed routes on the initial topology do not lead to deadlocks. Only for four design points there was one deadlock condition which was removed with one extra VC. This result is significant

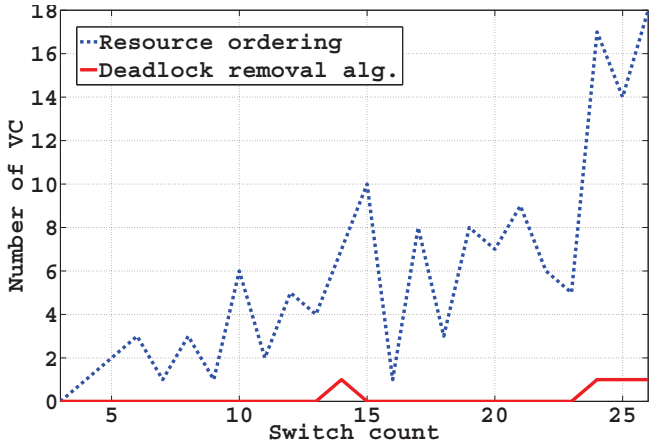

Figure 5.11: Comparison for *D*26_*media*

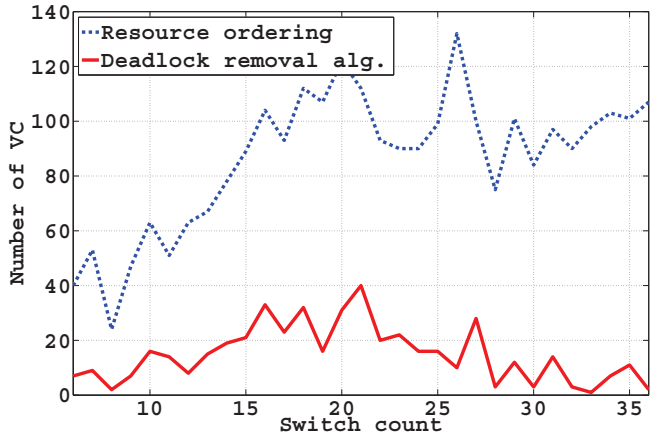

Figure 5.12: Comparison for *D*36_8

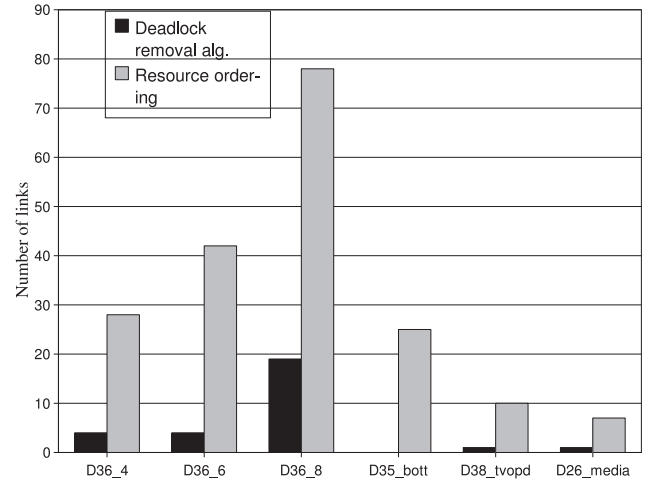

Figure 5.13: Comparison for different benchmarks

because it also shows that an application specific topology can be deadlock free even without applying restrictions on the routing function. Applying restrictions on routing can lead to overhead which in many cases may not be needed.

For a better comparison, I ran a similar test on a benchmark with more complex traffic patterns that would lead to more deadlocks in the generated topologies. The $D36\_8$ is a multi-media benchmark that has 36 processing cores. Each processing core sends data to eight other cores. There are 288 communication flows in the network. The results of the experiments are presented in Figure 5.12. In this case, all topology points have deadlock and for some a considerable number of VCs need to be added to break the deadlock. My method still has a much lower overhead than resource ordering.

To make the comparison more complete, I used four other benchmarks. The $D36\_4$ and $D36\_6$ benchmarks have spread traffic patterns and are similar benchmarks with $D36\_8$, but there are only four and six communication flows, respectively, starting from each processing core. The $D35\_bott$ benchmark has bottleneck traffic pattern. In this benchmark there are 16 processors and each communicates to one of the 16 private memories and to the three memories which are shared among all the processors. In case of the $D38\_tvopd$ benchmark, the traffic pattern is pipelined as it is the case with most multimedia applications. I present the results for all benchmarks in Figure 5.13. The results presented are for topologies with a medium switch count (14 switches) for all the benchmarks. In the case of the benchmarks with spread traffic, many of the flows have to traverse several switches which leads to more possible deadlocks, but this also increases the resource demand for resource ordering. In the bottleneck benchmark most of the flows are between processor and private memory which are on the same switch or very close, which leads to few or no deadlock conditions. For $D38\_tvopd$ there are also fewer deadlocks than in the case of the spread traffic. The experiments on these benchmarks also show the need for deadlock removal methods to be application-specific, unlike having a general restriction of turns.

5.4.2 Power comparison with VCs

For power and area estimations, I use the models for switches from [79]. The NoC power consumption for the different benchmarks is presented in Figure 5.15. The $D36\_4$ and $D36\_6$ benchmarks have spread traffic patterns and are similar benchmarks with $D36\_8$, but there are only four and six communication flows, respectively, starting from each processing core. The $D35\_bott$ benchmark has bottleneck traffic pattern. In case of the $D38\_tvopd$ benchmark, the traffic pattern is pipelined as it is the case with most multimedia applications. The plot shows the relative power consumption overhead for the resource ordering method when compared to the deadlock removal algorithm. The values reported in the plot are for topologies with 14 switches.

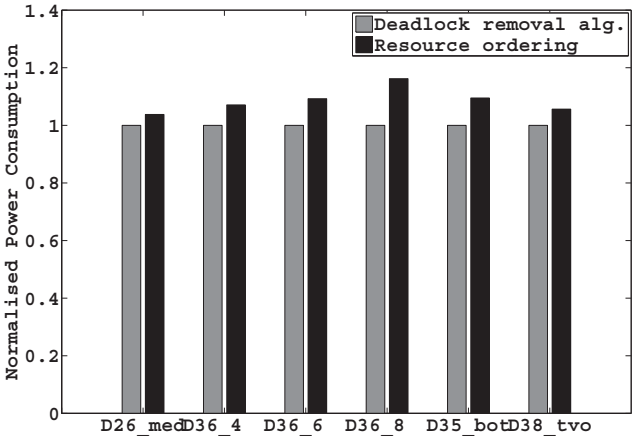

Figure 5.14: Power comparison with VC

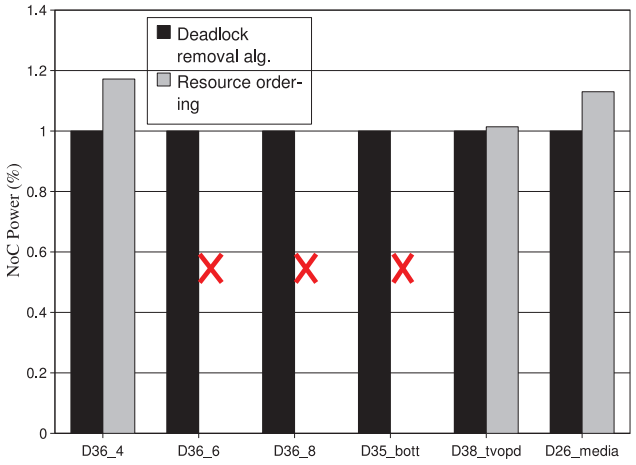

Figure 5.15: Power comparison

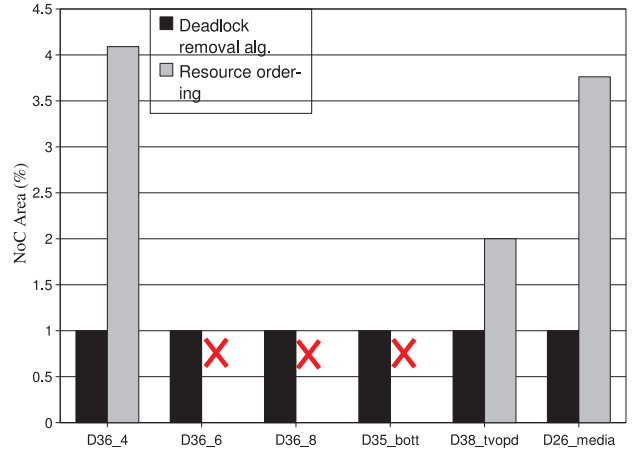

Figure 5.16: Area comparison

As can be seen from the figure, my method incurs a significant reduction in power consumption (an average of 8.6%) and in the experiments I observed a large reduction in NoC area (an average of 66%) when compared to the resource ordering method.

I also compared the power consumption of the topologies after removing the deadlocks with the original designs where deadlocks were not removed. From the experiments, I observed only a small overhead on power (of less then 5%) for the deadlock removal method.

I also ran the turn prohibition algorithm from [159] to remove deadlocks on existing topologies. The algorithm failed to find valid routes on many topologies (an average of 61% of the topologies on the different benchmarks failed). This is mostly because, when flows are re-routed to avoid turns, the bandwidth capacity on the links were violated.

5.4.3 Area and power comparison with physical channels

As many of the NoC component libraries developed for application specific NoCs do not support virtual channels [160], [154], I analyze the performance of my method by adding physical links to break the deadlock conditions as well. To compare the power and area overhead of my method to resource ordering, I used the power and area models of the switches in the NoC component library from [160]. For this set of experiments, I used the marginal power consumption as the cost metric, as described in Section 5.3.3.

The NoC power consumption for the different benchmarks is presented in Figure 5.15. The NoC area for the different benchmarks is presented in Figure 5.16. The plots show the relative power consumption and area overhead for the resource ordering method when compared to the deadlock removal algorithm. The values reported in the plot are for topologies with 14 switches as well. For the $D36\_6$, $D36\_8$ and $D35\_bott$, no values are reported for resource ordering as the method resulted in addition of a large number of links, requiring switches that were too large to meet the frequency constraints of the design. For the $D36\_4$, $D38\_tvopd$ and $D26\_media$ feasible topologies could be built with resource ordering. However, as can be seen from the figures, my method incurs a large reduction in NoC area and power consumption (10% power) when compared to the resource ordering method.

5.4.4 Benefits of deadlock removal after topology synthesis for 3D-ICs

The method I have presented in Chapter 3 to synthesize topologies for 3D-IC use turn-prohibition to prevent deadlocks. However when the constraint on the number

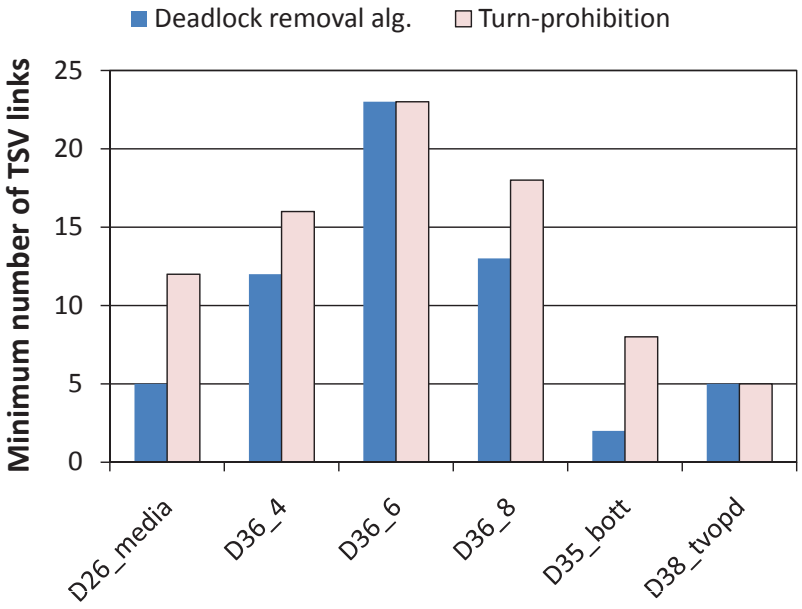

Figure 5.17: Minimum number of TSVs for topologies to be synthesized with the two methods

of TSVs is tight the method may fail as due to prohibiting some turn new TSV links need to be opened to ensure connectivity. If the TSV constraint prevents that then the topology cannot be built. However as turn-prohibition is used during the topology synthesis process and the topology is still unknown, it has to assume a fully connected switch graph. This can lead to prohibiting some turns that do not need to be as the other possible turns are not used. Therefore under tight TSV constraint the synthesis method may fail unnecessarily.

Another way to make sure that a synthesized topology is deadlock free, is to design the topology without turn-prohibition and to apply the deadlock remove algorithm described in this chapter on the resulting topology. As there is not a unique solution to breaking the cycles in the CDG, it is possible in many cases to remove deadlocks without adding extra TSV link.

In Figure 5.17, I compare the two approaches for designing topologies. The plot shows what is the minimum TSV constraint for which the two approaches have produced a topology for different benchmarks. As can be seen from the plot using the deadlock removal algorithm after topology synthesis can produce topologies for much tighter TSV constraints. Only for $D36\_6$ and $D38\_tvopd$ the result is the same. In the first case the traffic patterns of $D36\_6$ are very complex distributed requiring many TSV links and removing deadlock cannot be done without adding TSV links as well. In

the latter case the vertical communication is limited and therefore even when using turn-prohibition during synthesis topologies can be designed with a small number of TSV links.

In practice my algorithm runs fast. I ran the experiments on a 2GHz Linux machine. The method runs with in minutes even for the largest benchmark and it is scalable. As the algorithm is fast and scalable it can be added to the output of any topology synthesis design flow without incurring a significant delay compared with the original flow.

5.5 Summary

Wormhole routing is the protocol of choice for many NoCs as it provides high throughput, low latency and it does not have high buffering requirements. One important issue that arises when wormhole routing is used in a NoC is that routing deadlocks can occur during system operation.

Removing deadlocks in *Networks on Chips (NoCs)* with minimum area-power overhead is a major challenge in application-specific NoCs with custom topologies and routing patterns. In this work, I presented a method to remove the conditions that can lead to deadlocks with minimum area-power overhead. The application communication patterns are used to minimize the number VCs (or physical links) that need to be added to remove the deadlock conditions. The resulting topology is guaranteed to be deadlock free. The method can be applied any arbitrary NoC topology and routing function. The experiments show that the method leads to large reduction in NoC area and power consumption overhead when compared to existing schemes. I also found that the method has less than 5% area, power overhead when compared to designs that do not support any deadlock removal method, thereby making it very practical.

6 Meeting hard latency constraints in best-effort NoCs

Networks on Chips (NoCs) use scalable networking principles on a chip scale. They provide better performance, modularity and faster design closure when compared to bus based interconnects. Today, building NoCs that meet hard latency constraints imposed by real-time streams is a major challenge for designers. Several applications have strict requirements on latency for one or more traffic streams. For example, in radar and avionics applications, packets should be delivered from source to destination within a maximum time interval. Many applications require some level of QoS. In Section 2.4, I present an overview of the QoS flavors and requirements. In this chapter, I address the problem of meeting worst-case latency constraints.

To meet the hard real-time latency constraints, designers use NoC architectures that provide in hardware some form of guaranteed QoS support. Several different schemes are used, such as the use of TDMA *(Time Division Multiple Access)* slots [57], [88], packet priorities [153] and time-triggered communication [131]. Some of the schemes, such as the use of simple packet priorities, achieve soft QoS guarantees, where an absolute worst-case bound of latency cannot be provided. While some schemes, such as the TDMA based ones, can provide hard worst-case bounds. However, all these QoS based NoC architectures incur additional hardware overhead and/or penalize average performance to provide worst case guarantees. In fact, NoC architectures are usually classified as either "best effort" or "QoS" architectures.

In this chapter, I present a novel NoC synthesis framework to automatically build networks that meet hard latency constraints of end-to-end traffic streams. One key novelty in this approach is that I do not use special hardware mechanisms to meet the QoS constraints, but I construct the networks using simple, lean network components, identical to those used for best-effort NoC instantiation. Moreover, the worst-case guarantee is achieved with minimal impact on the average-case performance of the NoC. In other words, I show how a network that provides hard QoS guarantees on end-to-end packet delivery time, using a best-effort hardware infrastructure, can be

constructed.

While there are many works that have addressed the issue of synthesizing best-effort NoCs to meet the zero-load latency constraints [134], [65], [8], [158], [62], [181], [177], [121], none of them synthesize topologies that can meet hard latency constraints. In fact, I target the design of NoCs that use wormhole flow control, with round-robin arbitration at the switches, which are commonly used in most best-effort designs. I leverage mathematical models to compute safe worst case latency bounds on a wormhole based best effort NoC. I integrate these models with the topology synthesis process. Along with meeting the worst-case QoS constraints, the synthesized topologies also meet the average bandwidth and latency constraints, while minimizing power consumption.

In Chapter 3, as well as in several other works [121], [177], the power and area advantage of application specific NoC topologies for *Systems-on-Chip (SoCs)*, where the communication patterns are known, has been shown. Apart from improving power and area through the customization of the interconnect according to application requirements, in this work I show that other performance metrics like hard QoS requirements can be fulfilled. Therefore contributions of this chapter are twofold:

- Through experiments I show that with careful topology design the required worse-case delay bounds can be met using even best-effort NoC hardware.
- I provide an algorithm to automatically design the NoC topology to meet the required worse-case delay bounds.

I perform experiments on several *System on Chip (SoC)* benchmarks. When compared to a topology synthesis methods with no support for real-time constraints, the proposed method can produce topologies that can meet significantly tighter worst-case latency constraints (on average 44%). The results also show that the power consumption and average zero-load latency values of the topologies designed using the proposed scheme are only marginally higher as compared to a synthesis method that does not support real time constraints (on average 8.5%). Indeed, when only certain traffic streams require hard QoS guarantees, the topologies synthesized by this proposed method has negligible power consumption or area overhead.

6.1 Real-time synthesis compared to mapping onto regular topologies

In previous work [120], Murali et al. showed a method to automate tasks (cores) mapping onto custom topologies with QoS guarantees. There are three main differences

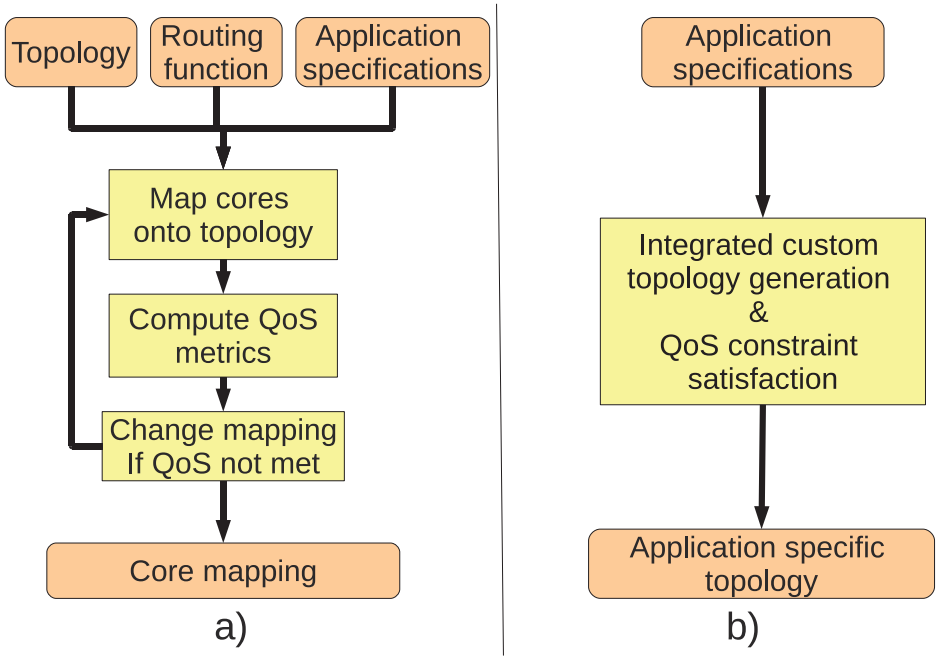

Figure 6.1: Algorithm flow: a) Task mapping with QoS, b) Real-time topology synthesis

between the mapping algorithm and the proposed application specific NoC synthesis algorithm with worst-case delay guarantees. First of all the mapping algorithm takes as input (apart from the application specification) the topology on which to map and the routing function that is to be used. As the mapping problem is constrained by the topology and the routing function, the search space is significantly reduced. When topologies are synthesized, there are no such constraints. Therefore there is a larger search space that needs to be explored and it is not trivial to go from the mapping of cores to application specific NoC topology synthesis. As such the synthesis algorithm has to solve new problems like connecting cores to switches, determine the switch to switch connectivity and find routes for flows that meet the worst-case latency requirements.

In Figure 6.1, I show the main parts of the task mapping algorithm (a) and for the real-time synthesis algorithm (b). As can be seen from the figure the another main difference comes from the way that QoS is provided. In case of the task mapping algorithm, QoS is ensured by checking the constraints for a given mapping and remapping if the constraints are not met. Also given a mapping of the cores, the routes are determined by the routing function. In the case of the topology synthesis algorithm, routing of each communication flow is integrated and with the topology building process and strongly connected to worst-case delay constraint checking. This integrated approach not only ensures better performance of the designed NoC when

compared to the mapping approach, but also give us the ability to build topologies that meet worst-case latency constraints without the need for specialized hardware. The synthesis algorithm also explores solutions with different number of switches to determine the topology that has the lowest power consumption and meets the worst-case latency constraints.

The final difference is related to the QoS model used in the two works. In the mapping algorithm an average case QoS metric is used, as the models are based on simulation. The previous work assumes a sort of soft QoS where traffic is regulated and therefore it is different from the hard QoS metrics I consider in this chapter. In the present chapter, the integrated topology design and routing with constraints checking enables the synthesis algorithm to use the extra degrees of freedom not only to design topologies that meet worst-case latency constraints, but minimal power and area as well. In the remainder of the chapter, I will describe in detail the integrated synthesis approach.

6.2 Worst-case latency models

If a fair arbitration scheme like round-robin is used in the switches, then the worst-case latency of packets is determined by the topology of the NoC and the routes chosen for the communication flows. A carefully designed NoC topology and well-selected routes can decrease the worst case latency bound and QoS can be provided without the use of extra hardware in the switches. To improve the design methods for designing application specific NoCs that can provide real time guarantees, it is important to be able to calculate the worst-case latency of the flows for a given network configuration.

For this work I assume that the switches use round-robin arbitration. For illustrative purposes, I consider an architecture with input-queued switches with no virtual channels, characteristic of many existing NoC designs [160]. The methods presented here can be easily modified for other switch architectures and to support virtual channels. A credit based (or on-off) flow control is used to provide back-pressure and to prevent the switches from forwarding the *flow control units (flits)* when the downstream buffers are full. The target cores are assumed to be ideal and eject the flits as soon as they reach the target network interface. If a target core is not ideal an end to end flow control mechanism can be used to prevent flits from entering the network when it is backlogged. Buffer size is assumed to be uniform across all the switches. In this chapter, I do not address the issue of buffer sizing, as it is beyond the scope of this paper.

In this section, I present a brief description of the mathematical model proposed in [139] to calculate worst-case latencies for a given topology, routing function and traffic flows. In the next section, I show how the model can be used to build the topology

and find paths for the given set of traffic flows between the cores of the applications.

The buffer depth of a switch is calculated as the sum of all buffers between the arbitration points of two consecutive switches. Since no traffic regulation is assumed in the model, the worst-case latency is achieved when all buffers are full and when the packet of a flow loses arbitration to all other flows that it can contend with. Under these assumptions, the upper bound on delay for a flow is given by Equation 6.1 (UB_i represent the upper bound delay for flow i) where ts_1 and ts_2 represent the packet creation and ejection times which are constant. The sum adds the contribution of the worst-case interference at every hop (u_i^j interference of other flows on flow i at switch j) from the path of the flow for which the upper bound delay is calculated. The number of hops on the path of flow i is denoted by h_i.

$$UB_i \; = \; ts_1 + ts_2 + \sum_{\forall j} u_i^j \quad with\ j = 0 \ldots h_i \tag{6.1}$$

If a core has flows going to different destinations and it is capable of having multiple outstanding transactions, then the packets generated by the same core to different destinations can also contend with one another. Source contention, as it is also called, is modeled by using a virtual switch that does not physically exists on the path of the flow. Equation 6.2 describes how to calculate the contribution of the source contention.

$$u_i^0 \; = \; MAX(U_i^0, U_{I(x)}^0) + \sum_{\forall x} U_{I(x)}^0 \quad with\ x = 0 \ldots z_0(i,0) \tag{6.2}$$

The hop delay from output buffer to output buffer is denoted by U_i^j and $I(x)$ returns the index of a flow from the pool of flows that contend with flow i at switch j. The number of flows that contend with flow i at switch j and use the output port c is denoted by $z_c(i,j)$. Using the same notation, the delays at the rest of the switches on the path of flow i can be calculated with Equation 6.3.

$$u_i^j \; = \; MAX(U_i^j, U_{I(x)}^j) + \sum_{\forall x} U_{I(x)}^j \quad with\ x = 1 \ldots z_c(i,j),\ 1 \leq j \leq h_i \tag{6.3}$$

The delay of a flow at the current switch U_i^j is calculated in a similar manner, only this

time it is based on the delays from the next switch.

$$U_i^j = MAX(U_i^{j+1}, U_{I(x)}^{j+1}) + \sum_{\forall x} U_{I(x)}^{j+1} \quad with\ x = 1 \ldots z_c(i, j+1),\ 0 \leq j \leq h_i - 1 \tag{6.4}$$

To calculate the upper bound delay, the Equations 6.2, 6.3 and 6.4 have to be calculated in a recursive manner. The recursive formulation is guaranteed to complete because the delay of any flow at the last switch in the path is fixed. The termination conditions are given by Equation 6.5:

$$U_i^{h_i} = L_i, \quad U_{l(x)}^{h_{l(x)}} = L_{l(x)} \tag{6.5}$$

The ejection time of a packet at the last switch in cycles for flow i is denoted as L_i.

In [139], the correctness of the models and the tightness of the bounds is shown. The authors also show how the models can be extended to address multiple virtual channels, buffer sizes and packet lengths.

6.3 Topology design to meet worst-case constraints

In this section, I give an example of how topology design can reduce the worst-case delay for some flows when compared to a single switch (crossbar) topology, which is non intuitive. Consequently I will give the intuition of why careful synthesis can potentially reduce the worst case delay of some flows. Intuitively one would expect the crossbar to have the lowest worst-case delay. While that is true for the average worst case delay over all flows, it is not true for individual flows. In many design there are few flows that have hard real-time constraints (e.g. interrupts) and many flows that are best effort (e.g. cache refills). Therefore a multi-switch topology can be optimized to reduce the worst case delay of only those flows that have real-time constraint in the detriment of the other that are best effort.

In Figure 6.2, I present a simple example of the worst case delay for three flows for a single switch (crossbar) and a two switch topology. Let us assume that flow 1 is a real-time flow while flow 2 and 3 are best effort flows. Also let us assume that the packet size for all flows is 5 flits and the sink is ideal so the ejection latency is 5 cycles. In Figure 6.2.a I show the case for the single switch. In this case the three flows that have the same destination will contend for the output port. In the worst case for flow

one, we have to assume that it would lose the arbitration to both flows 2 and 3. Since the ejection latency of the flows is 5 cycles and flow 1 could in the worst case wait for the other two flows, it would see a delay of 10 cycles to win the arbitration for the output port. If we add to this the ejection latency of flow 1, we can compute a worst case delay of 15 cycle for flow 1. Since the topology is symmetric, we can similarly calculate the same worst case delays for flows 2 and 3 (i.e. 15).

However since only flow 1 is a real-time flow I want to improve the worst case delay of only this flow. In Figure 6.2.b, I show how that can be done with a two switch topology. In this case flow 2 and flow 3 first contend for the output port of one switch and only the winner will contend with flow 1 for the output of the other switch. So in this case when we calculate the worst case delay for flow 1, we have to assume that in the worst case flow 1 will lose the arbitration to a packet of either flow two or 3 coming from the other switch. So the worst case delay is 10 cycles (5 cycle to win the arbitration and 5 cycles ejection latency). This will however increase the worst case delay of flow 2 and 3. When we calculate the delay of flow 2, we must assume that it will lose the arbitration with flow 3 on the first switch and with flow 1 on the next switch. Add to that the ejection delay and the total worst case delay for flow 2 is 20 cycles. Similarly flow 3 will have 20 cycles worst case delay.

If we look at the average worst case delay over all the flows we see that the single switch topology is better. However since only flow 1 has real time constraint in my example, in the two switch topology the worst case delay of flow one was reduced compared to the single switch case.

6.4 Real-time network synthesis

The goal of the real-time network synthesis algorithm is to find a power-efficient topology that meets the application requirements and the real-time constraints. Therefore the real-time network synthesis algorithm requires the following inputs. The most important input is the communication description of the SoC. The description is given in the form of a directed graph. The definition of the communication graph is similar to the one from Chapter 3, except for the latency which now represents the worst-case latency bound:

Definition 6.1 *The communication graph is a directed graph, $G(V,E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j) with $i \in 1,2...n$ and with $j \in 1,2...n$ representing the communication between the cores v_i and v_j. The bandwidth of the traffic flow from cores v_i to v_j is represented by $bw_{i,j}$ and the latency constraint for the flow is represented by $rt_{i,j}$.*

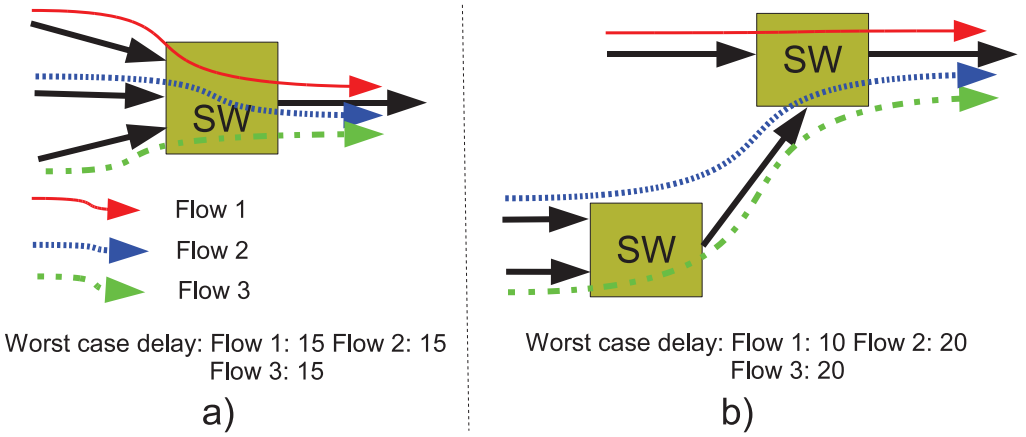

Figure 6.2: Example of topologies with worst case delays: a) single switch (crossbar); b) multi-switch

The number of switches and the values of the architectural parameters (the required operating frequency and the *flow control unit* size) are varied to explore different design points. For a given combination of the number of switches and architectural parameters the real-time network synthesis algorithm designs a topology.

I provide power, area and delay models for the NoC components to the algorithm in order to estimate the power consumption of the topology and to make sure that the architectural requirements are met. For a given technology library, synthesis and place&route of RTL code of the NoC components using commercial tools, is done to obtain the models. Floorplan information can also be provided as input in order to better estimate the wire length and the wire power consumption.

The output of the synthesis algorithm is a topology that minimizes the power consumption and that meets the worst case delay constraints given in the communication specification.

6.4.1 Real-time synthesis algorithm

The major steps of the method that finds designs the topology for a given combination of the number of switches and architectural parameters, are presented in Algorithm 6.1. Please note that the step number corresponds to the line number in the Algorithm 6.1 listing. The algorithm takes as parameters the communication graph (G), the number of switches (num_sw) and the architectural parameters.

The assignment of switches to cores is done by partitioning the cores in blocks (one partition block per switch) using a min-cut partition method. Minimum cost routes

for flows are found using Dijkstra's shortest path algorithm in conjunction with turn prohibition for routing level deadlock freedom. These steps are general and use well know algorithms and I will not describe them in detail as they are similar to similar to previous work [121]. However the customization of Dijkstra's general algorithm, for finding shortest path, to perform routing is done through the cost graph that is fed as input. One important difference from the previous work is the way that the cost are calculated that allows us to drive the path finding routine to create routes that meet the real time constraints. As flows are routed one by one, unlike in the previous work, routing a flow does not influence only its worst-case delay, but it also changes the worst-case delay of the previous flows. So another major difference from the previous work is the iterative approach to find a route that meets it own constraint but also does not cause the previously mapped flows to miss theirs. As part of the iterative approach, there are also some special cases that are treated differently during routing in order to find viable route. These new feature needed for finding routes that meet worst-case delay constraints are described in detail in the following paragraphs.

Since the number of switches is most of the time different than the number of cores, the core to switch connectivity has to be decided (step 5). The core to switch assignment is done by partitioning the cores in as many blocks as there are switches. To perform the partitioning, I use a *Partitioning Graph (PG)* as defined in Chapter 3. For completeness I present the definition below as well:

Definition 6.2 *The partitioning graph is a directed graph, $PG(U, H, \alpha)$, that has same set of vertices and edges as the communication graph. The weight of the edge (u_i, u_j), defined by $h_{i,j}$, is set to a combination dependent on α of the bandwidth and the latency constraints of the traffic flow from core u_i to u_j.*

The PG is built using the parameter α (initialized to 0 in step 1) which is varied to generate different core to switch assignments when the required worst case delays cannot be met. Initially only the bandwidth influences the core to switch assignment and as α is varied, more importance is given to the real-time constraints of the flows. I use min-cut partition and an existing tool [17] to generate the core to switch assignment. The cores in one partition block are connected to the same switch. If the parameter α reaches a certain upper bound, the algorithm exits as it is unable to find a solution.

After the core to switch assignment is decided, the flows between cores that are on different switches (inter switch flows) have to be routed. The algorithm loops through the flows (step 6) and first chooses which is the next flow to be routed. The choice of the next flow can be done using several criteria. For example the highest bandwidth flows could be mapped first, or alternatively the flows with the tightest real-time constraint could be mapped first. I use a linear combination of the bandwidth requirements

Algorithm 6.1 Real_Time_Topology_Synthesis(G, num_sw, architectural parameters)

```
1: set α=0
2: if α ≥ max_α then
3:    Exit
4: end if
5: assign_cores_to_switch(num_sw, α)
6: for i = 1 to |E| do
7:    Choose next unmapped inter switch flow
8:    set β=α
9:    build_cost_graph(β)
10:   Find min cost path
11:   if path not found or β ≥ max_β then
12:      increment α
13:      Goto step 2
14:   end if
15:   Check RT constraints for all mapped flows
16:   if previously mapped flow violates RT constraints then
17:      if destination(current flow) = destination(violated flow) then
18:         destination_contention(violated flow)
19:      else
20:         path_contention(violated flow)
21:      end if
22:      increment β
23:      Goto step 9
24:   end if
25:   if current flow violates RT constraints then
26:      increment β
27:      Goto step 9
28:   end if
29: end for
30: Save topology
```

and the real-time constraint requirement to decide the order of the flows. The linear combination is calculated in a similar way to how the weights of the edges in the partitioning graph are calculated.

Since I do not use indirect switches, I have to route the inter switch flows through the switches to which the cores are connected (the number of switches is given as input). To find routes for the flows I do the following: i) I calculate the costs to go from each switch to every other switch; ii) using Dijkstra algorithm I find a minimum cost path, which becomes a temporary route for the flow (the route lists the hops and the ports used at each hop); iii) I test the real time characteristics the found route; iv) if the found route does not fulfill the real time requirements I repeat these steps changing the cost until a path is found that meets the bound. The details for these steps are

described in the following parameters.

A parameter β is used to influence the cost calculation in order to shift the optimization criteria from minimizing power to minimize the worst case delay. In step 8, I initialize the parameter β to the current value of α. This parameter is varied locally to change the optimization criteria only for the current flow that is routed, when a valid route is not found. In step 9 a cost graph is built. The cost graph is defined as follows:

Definition 6.3 *The cost graph is a fully-connected directed graph, $C(S, L, \beta)$, where the vertices represent the switches in the topology. The weight of the edge (l_i, l_j), defined by $cost_{i,j}$, gives the cost of routing the flow from switch i to switch j.*

The way the weights of the edges in the cost graph are calculated is presented in more detail in Section 6.4.2. Based on the cost graph the minimum cost path is found in step 10. This path is used as the route for the current flow. I use Dijkstra algorithm and turn prohibition to find deadlock-free minimum cost routes as presented in [121]. If a path is not found for any flow or the parameter β has reached the maximum value, then α is incremented and I return to step 2 to retry with a different core to switch assignment.

After a valid route is found for the current flow the algorithm tests weather the real-time constraints are met (step 15). This is done using the worst-case delay models presented in Section 6.2 recursively. Not only the current flow needs to be tested, but also the previously mapped flows, because the interference of the current flow can cause the already mapped flows to violate their bounds. If the constraints are met then the algorithm proceeds to mapping the next flow.

If there is a bound violation, there are two cases that need to be considered: i) a previously mapped flow has violated its constraint or ii) the current flow has violated the constraint. If a previously mapped flow is the one that violates its worst case delay constraint, there are a further two sub-cases that need to be considered. One sub-case is if the flow that violates the bound and the current flow that I am trying to route have the same destination. This sub-case is more complex as it is not possible to fully separate the routes of the two flows and it is handled by the function $destination\_contention$ (step 18). In the other sub-case the contention is along the path of the current flow and the previously mapped flow and the function $path\_contention$ is called (step 20). The two functions annotate the states of some links so that in the next iteration (after incrementing β, steps 22, 23) the weights in the cost graph are modified such that the conditions that cause the previous flow to violate the constraint can be avoided. A more detailed description of the two functions is presented in Sections 6.4.3 and 6.4.4. If the current flow violated the worst case

Algorithm 6.2 cost($switch_i$, $switch_j$, β)

1: Find $link$ between $switch_i$ and $switch_j$ that support the required bandwidth
2: **if** $link$ found and flow_count_on($link$) $\geq$ $BOUND/\beta$ **then**
3: Goto step 1 and find next link
4: **end if**
5: **if** $link$ not found **then**
6: **if** can open new link **then**
7: $link$ = link_count($switch_i$, $switch_j$) + 1
8: cost[$switch_i$][$switch_j$] = marginal_power_new_link($switch_i$, $switch_j$, bandwidth)
9: **else**
10: cost[$switch_i$][$switch_j$] = INF
11: **end if**
12: **else**
13: cost[$switch_i$][$switch_j$] = marginal_power_existing_link($switch_i$, $switch_j$, bandwidth)
14: cost[$switch_i$][$switch_j$] += (flow_count_on($link$) / max_flow_count) * β
15: **end if**
16: **if** link_status[$switch_i$][$switch_j$][$link$] = $PROHIBITED$ **then**
17: cost[$switch_i$][$switch_j$] = INF
18: **end if**
19: **if** link_status[$switch_i$][$switch_j$][$link$] = ADD_EXTRA_COST **then**
20: cost[$switch_i$][$switch_j$] += max_cost
21: **end if**

delay constraint then I simply increment the parameter β, and I do a local iteration (steps 26, 27).

If all the inter switch flows could be routed successfully then the topology is saved (step 30) and the algorithm finishes. The power and area of the generated topology is estimated after this point.

The time complexity of the algorithm is $O(|V|^4|E|^3 ln(|V|))$, where $|V||E|^2$ is the contribution given by checking the worst-case latency constraints as presented in [139]. In practice the algorithm runs fast as valid path can be found much earlier when the constraints are not that tight.

6.4.2 Cost calculation

Calculating the weights of the edges of the cost graph is an important step, because through the cost assigned to the different edges I can drive the algorithm to find paths that meet the worst case delay constraints and that minimize the power consumption of the topology. The cost calculation for one edge is presented in Algorithm 6.2.

In the first step the function tries to find a link that exists between the two switches

and that has sufficient capacity to accommodate the bandwidth requirement of the new flow. If such a link is found it tests to see if the number of flows already mapped to that link are smaller than a certain bound determined experimentally (step 2). As the bound depends on β, it forces the algorithm to reuses fewer existing links as β is incremented in successive iterations. If the number of flows on the link is larger than the bound then it goes back to step 1 to look for the next viable link.

If an existing link that can be reused was not found, then the algorithm would need to open a new link between those switches. If a new link cannot be opened (because the size of the switches would become too large and they would not support the required frequency) the cost for that edge in the cost graph is set to a large (INF in step 10). This prevents the flow to be routed between the two switches. If the link can be opened then the cost of that link is given by the marginal power increase due to the addition of the new link (step 8). The power consumption increases when a new link is open due to an increase in the size of the switches and the increase in switching activity as the low adds the extra traffic.

If an existing link is found and it can be reused, then the cost of reusing that link is given by the marginal increase power that the new flow creates (step 13). In this case the switch size remains the same, but the switching activity is increase due to the extra traffic. In case a link is reused there are other flows that use that link. I add extra cost that is proportional to the number of flows that are already mapped on that link weighted by the value of the parameter β. In this way I drive the path computation to reuse links that will be shared by fewer flows and even to use new links as the value of β is increased.

Finally based on the status of the link that is set from a previous iteration by the $destination\_contention$ and $path\_contention$ functions I can force to prevent communication between some switches (step 17) or to increase the cost in order to favor the use of other links (step 20). The way the link status is set and the function that the link status has, is presented in detail in Sections 6.4.3 and 6.4.4.

6.4.3 Destination contention

The $destination\_contention$ function is called, if the route that was found for the current flow, would lead to a previously mapped flow to violate its constraint. Also the contention between the current flow and flow that violates the constraint happens on the output port of the last switch, just before the destination NI. Thus, I want to force the new flow to contend with a previously mapped flow that does not have real-time constraints, but that goes to the same output as the flow that violates the constraint. By doing this I increase the worst-case delay of the flow that is not RT and of the new

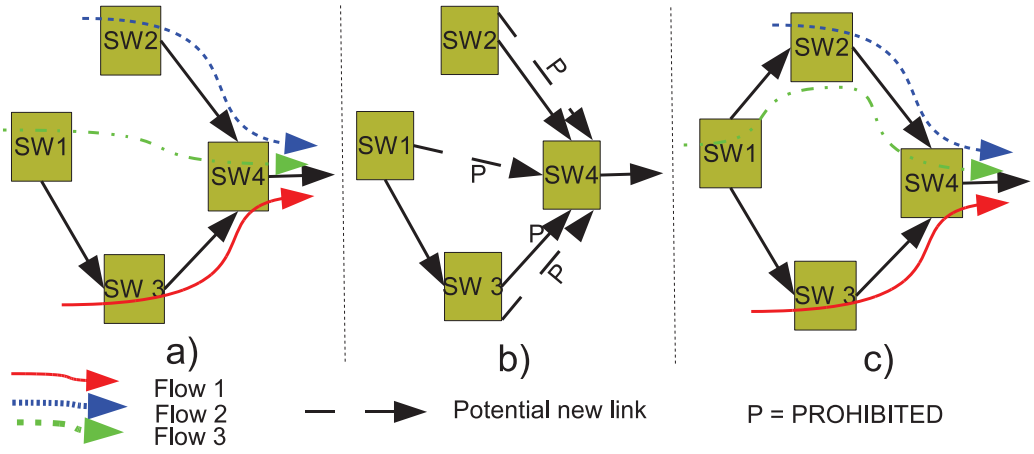

Figure 6.3: Destination contention example: a) initial topology; b) link status annotation; c)final topology

Algorithm 6.3 destination_contention(violated flow)

1: ls = last switch for the violated flow
2: **for** i = 1 to num_sw **do**
3: **if** $i \neq ls$ **then**
4: **for** j = 1 to link_count(i, ls) **do**
5: **if** link[i][ls][j] is not used by non RT flow that contends with the violated flow at the destination **then**
6: link_status[i][ls][j] = $PROHIBITED$
7: **end if**
8: **end for**
9: {Prohibit the use of new link}
10: link_status[i][ls][link_count(i, ls) + 1] = $PROHIBITED$
11: **end if**
12: **end for**

flow, but it would not modify the worst case delay of the flow that now violates the constraint. The steps to achieve this are presented in Algorithm 6.3

In step 1, I find the last switch (ls) of the flow that was mapped and which violates the constraint. I loop through all the existing switches (step 2) and see if there are existing links to the switch ls. If such links exist, it checks whether it is used by a non real-time flow that goes to the same output as the flow that violates the constraint. If no such flow, exists the link is prohibited from being used in the next iteration by setting the link status to $PROHIBITED$ (step 6). If a new link is opened then there is no flow mapped on that link. So if the current flow uses a new link it will still contend with the previously mapped flow at the destination. Therefore the use of new links toward the switch ls is prohibited (step 10).

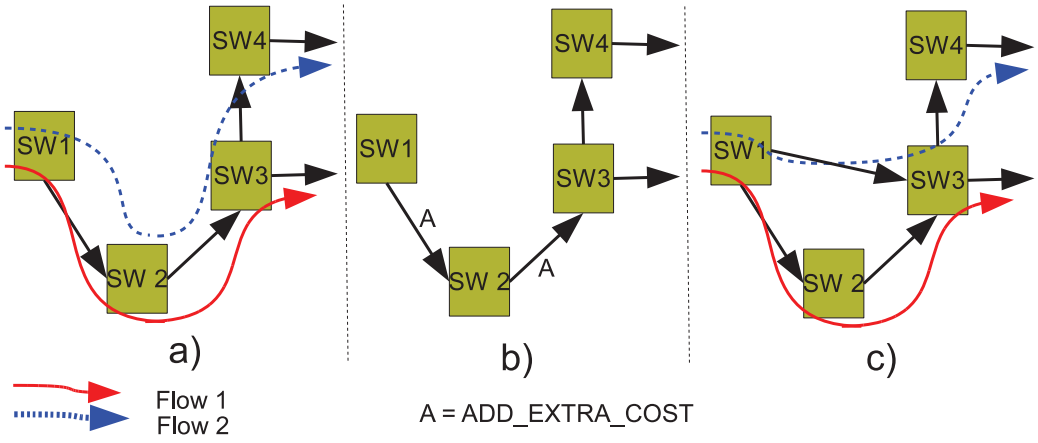

Figure 6.4: Path contention example: a) initial topology; b) link status annotation; c)final topology

Algorithm 6.4 path_contention(violated flow)

```
1: for i = 1 to num_sw do
2:   for j = 1 to num_sw do
3:     for k = 1 to link_count(i, j) do
4:       if violated flow uses link[i][j][k] then
5:         link_status[i][j][k] = ADD_EXTRA_COST
6:       end if
7:     end for
8:   end for
9: end for
```

Example 6.1 *In Figure 6.3, I show an example of how the destination contention is removed. Assume I have the topology from Figure 6.3.a and there are three flows. Flow 1 is a real time flow that is routed through switch 3 and 4. Flow 2 is a non real-time flow and is routed through switch 2 and 4. Flow 3 is the current flow that has to be routed and after the first iteration the path using switch 1 and 4 was found. Suppose that by routing flow 3 through switch 1 and 4 causes flow 1 to violate the worst case delay bound. Therefore the destination_contention function will set the state of the links as shown in Figure 6.3.b. The link between switch 3 and 4 is prohibited, and also opening new links between switch 1 and 4, 2 and 4 and 3 and 4 is also prohibited. The existing link between switch 2 and 4 can be used as it is currently used by flow 2 which does not have real-time constraint but already contends with flow 1. A potential route for flow 3 is shown in Figure 6.3.c, where flow 1 uses the existing link between switch 2 and 4 that is also used by flow 2.*

6.4.4 Path contention

If $path\_contention$ function is called, it means that the route of the current flow intersects the route of previously mapped flow before the destination switch and causes the previously mapped flow to violate its bound. In this case I want to drive the path finding algorithm to avoid to use the links that the previously mapped flow uses. If the route of the new flow does not intersect with the previously mapped flow than it will not increase the worst case delay of the previously mapped flow. To that end the links used by the flow that violate its worst case delay constraint are annotated so that in the next iteration the cost for using such a link is increased.

The steps to annotate the links are presented in Algorithm 6.4. The function loops through the existing links in the topology and if it finds a link that is used by the flow that violates the bound it set its status to ADD_EXTRA_COST (step 5). The value of maximum cost of all edges in the cost graph is added in the cost calculation function to links that have the status set to ADD_EXTRA_COST. These links will have high cost and will be avoided by the path finding routine. These links can be used (unlike in the case where the cost is INF), if due to other constraints, no other links can be used.

Example 6.2 *An example of how the path contention is avoided is shown in Figure 6.4. An example topology is presented in Figure 6.4.a and there are two flows. Flow 1 is a real-time flow that is already routed. Flow two is the current flow for which I have to find a route. Assume that in the first iteration the route that is found is from switch 1 to 2, 3 and 4. So flow 2 intersects with flow 1 at the output of switch 1. Assume that by using this route for flow 2 causes flow 1 to violate the required worst case delay bound. In this case the $path\_contention$ function is called and will set the status of the link between switch 1 and 2 and of the link between switch 2 and 3 to ADD_EXTRA_COST as shown in Figure 6.4.b. Therefore in the next iteration those links will have higher cost and will be avoided if possible. A possible solution is shown in Figure 6.4.c where a new link between switch 1 and 3 is opened and the route of flow two uses switches 1, 3 and 4, and therefore the contention with flow 1 is removed.*

6.5 Experimental results

To evaluate the real-time topology synthesis algorithm, I chose two complex benchmarks extracted from real-life SoC platforms. The first benchmark is a state-of-the-art multimedia and wireless communication SoC [133, 110, 74]. The communication patterns for this benchmark are very irregular [149], as the described SoC is composed of two subsystems. One subsystem handles the multimedia functions. It contains an ARM processor, hardware video accelerator and a complex memory system that can access off-chip SDRAM and FLASH. The second subsystem is used for the wireless communication and is built around a DSP processor and several peripherals. Commu-

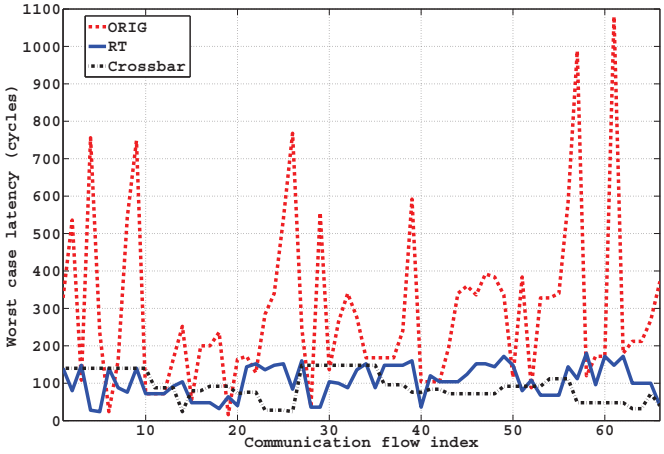

Figure 6.5: Worst case latency on each flow

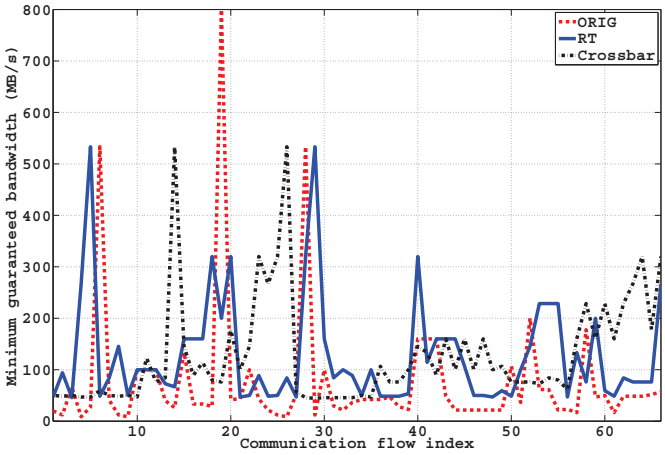

Figure 6.6: Minimum guaranteed bandwidth for each flow

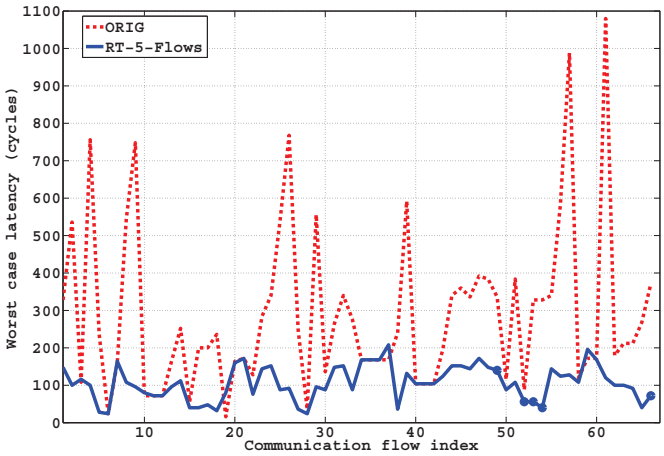

Figure 6.7: Worst case latency when only 5 flows are constrained

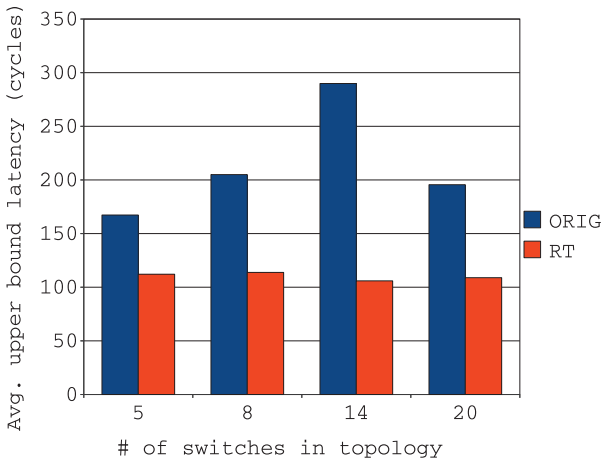

Figure 6.8: Average worst case latency for $D26\_media$

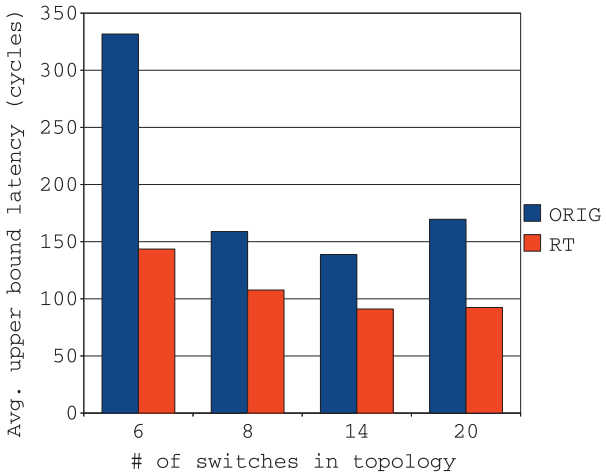

Figure 6.9: Average worst case latency for $D36\_4$

nication between the two subsystems is done with the help of a DMA core and several on-chip memories. There are a total of 26 cores that communicate in the system. The second benchmark is a SoC for high-end signal processing applications (such as those used in embedded image and radar processing). This SoC features multiple local memories, having a spread communication pattern. In this benchmark there are 36 cores and each core communicates to four other cores.

I applied the proposed synthesis algorithms on the two SoC benchmarks. For comparisons, I used an existing state-of-the-art synthesis algorithm that does not support hard QoS constraints [121] to design the topologies for the benchmarks. I use the methods described in Section 6.2 to calculate the worst case latencies for the flows for the generated topologies for both cases.

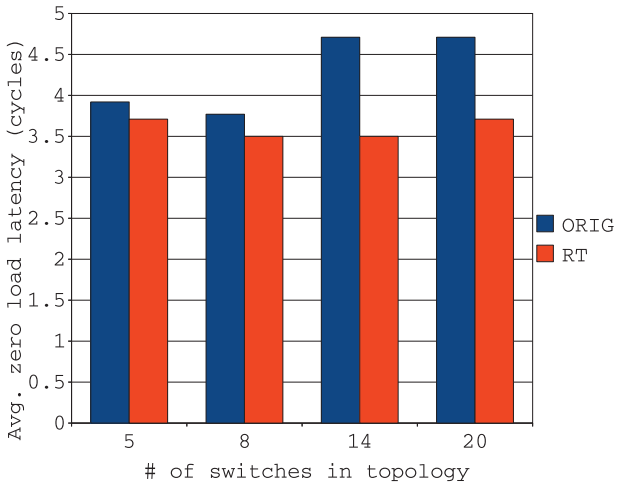

Figure 6.10: Average zero load latency for *D*26_*media*

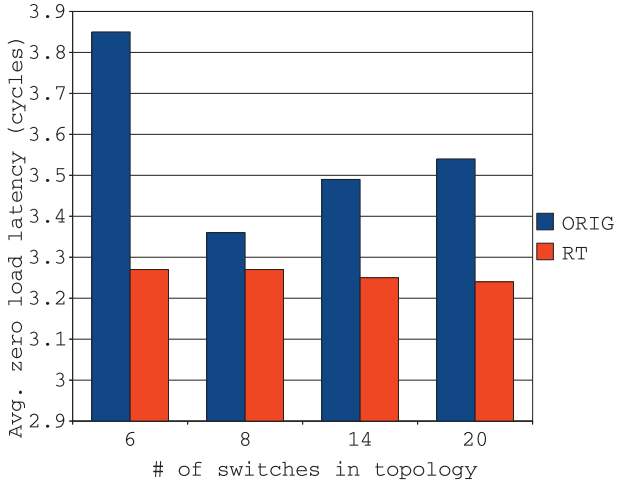

Figure 6.11: Average zero load latency for *D*36_4

6.5.1 Effect on latency

I designed four topologies using the original synthesis algorithm from [121], where real-time constraints are not considered. The different design points use different numbers of switches: 5, 8, 14, and 20 for the *D*26_*media* benchmark. I show results for 5 and 8 switches because, as I will show in the experiments, the worst case latency tends to be worse when there are fewer switches. On the other hand, these design points are important as they provide better power consumption when compared with topologies with many switches. I also give results for topologies with more switches (i.e., 14, 20) to give a clear picture of how the worst-case latency depends on the number of switches.

I then ran the RT algorithm and found the smallest latency bound for which it was possible to find solutions. While in many real applications only a subset of flows have real time constraints, I wanted to study the maximum overhead incurred by the proposed procedure. Thus, I put real time constraints on all flows. I found that for the $D26\_media$ benchmark, the tightest constraint for the upper bound delay for which feasible topologies could be build is 180 cycles. To find out how tight the constraint was, I calculated the worst case latencies of the flows only due to source and destination contentions. To perform this, I connected all the cores through a single crossbar switch and calculated the worst case latencies. On the crossbar, the average worst case latency for this benchmark was 92 cycle and the maximum value across all flows was 148 cycles. This shows that the constraint I imposed is quite tight, as it is only 1.25× the maximum value of the flows from the ideal case.

In Figure 6.5, I show the worst case latencies for the 66 flows in the $D26\_media$ benchmark. The worst case latencies are reported for the case when the crossbar is used and for the cases when a 14 switch topology is designed with the original algorithm and with the RT algorithm. As can be seen, for the topology designed with the RT algorithm, the worst case latency of the flows is in the same range as the worst case latency for the flows mapped on the crossbar. On the topology designed with the original algorithm, most flows have worst-case latency values much higher than those of the crossbar.

Another less intuitive effect that is visible in the plot is that the RT algorithm provides lower worst-case latency than the crossbar. This is because, in a crossbar, each flow will have to contend with all the other flows to the same destination. Whereas, in a multi-switch case, this may not happen. For example, if there are 3 flows to the same destination. In the multi-switch case, two of them may share a path until a point where they contend with the third flow. The third flow only has to wait for one of them (with the maximum delay) to go through. Whereas, in a full crossbar, the third flow will have to wait for both the flows, in the worst case. Thus, we can see that, when only few flows require real time guarantees a multi-switch topology can give better bounds and it is really difficult to come with the best topology directly using designer's intuition. The worst case model from [139] also gives a method to calculate the minimum guaranteed bandwidth under worst case contention in the network. In Figure 6.6, I show the calculated minimum guaranteed bandwidth for the 66 communication flows for the 14 switch topology.

So far I showed what happened to the worst-case latency when a constraint is set to all the flows. In Figure 6.7, I show the behavior of the RT synthesis algorithm when only 5 flows have worst-case latency constraints. The flows that had constraints are marked with bubbles on the figure. The latency constraints were added to flows going to and from peripherals. This is a realistic case, as many peripherals have small buffers and

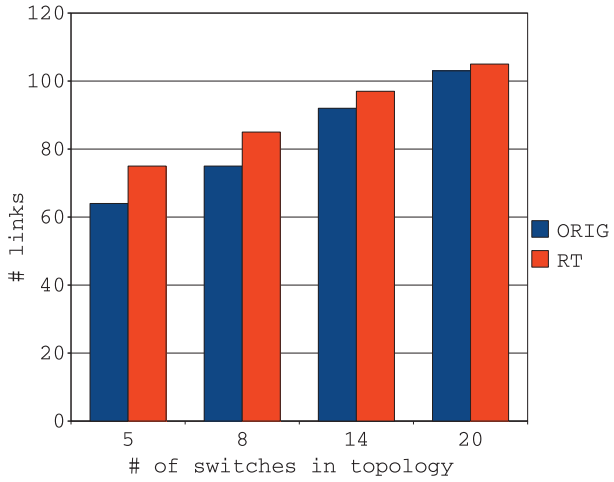

Figure 6.12: Number of links for $D26\_media$

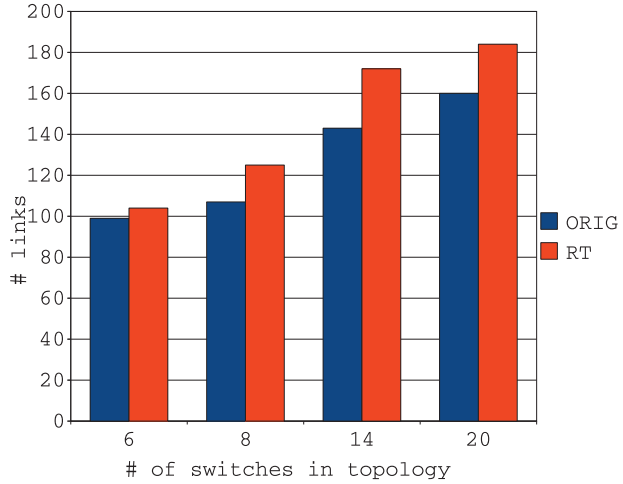

Figure 6.13: Number of links for $D36\_4$

data has to be read at a constant rate, so that it would not be overwritten. In this case, the bounds on those 5 flows could be tightened further (two flows at 160 cycles and three flows at 60 cycles). Putting these constraints also leads to a reduction in the worst case latency of other flows as well. Due to the tight constraints, the RT algorithm maps the RT flows first. Then, the unconstrained flows also have to be mapped with more care so that they do not interfere with the previously mapped ones.

In Figure 6.8, I show the worst-case latencies, averaged over all flows for both the original algorithm and the proposed RT algorithm. From the figure, it can be seen that a synthesis algorithm not considering the maximum latencies can incur a significantly higher worst case latencies than the required bound especially for low switch counts and we can observe large variations depending on the number of switches in the

topology. For the RT case, the variations are much smaller as the algorithm builds the topology in such a way so that all flows meet the constraints. One other important remark is that even in the unconstrained case, increasing the number of switches can help reduce the worst-case latency of flows. This is because, with increasing switch counts, fewer flows share each link, there by reducing the chances of contention. This is contrary to the zero load latency (the latency to go from source to destination without having any interference on the way), which grows with the number of switches in the topology, as can be observed from Figure 6.10.

For the $D36\_4$ benchmark, I designed topologies with 6, 8, 14 and 20 switches and I found that the tightest constraint for which topologies could be synthesized was 195 cycles. The average worst case latency for the different topologies designed for the $D36\_4$ benchmark are shown in Figure 6.9. The effects on the average zero load latencies are shown in Figure 6.11. Using the original algorithm, for 6 switches, the zero load latency is very high as most flows have to reuse existing links and cross more hops. For all other switch counts the zero load latency is smaller, but it constantly increases with the switch count. With the proposed RT algorithm the zero load latency dose not suffer large variations, as the algorithm is reusing fewer links in order to meet the constraints.

6.5.2 Effect on NoC components

To design topologies that meet the required worst case latency constraints, the RT algorithm may use more links and therefore additional switch ports when compared to the original synthesis method. By adding more links, contention between flows can be reduced, there by reducing worst-case latencies. But adding more links will increase the switch sizes and the power consumption of the NoC may also increase. It is important to analyze what is the overhead of the RT algorithm over the original method.

In Figure 6.12, I plot the total number of links used in the different topologies for the $D26\_media$ benchmark. As the number of switches is increased, the number of links used increases as well. With more links, the contention on each link is lower, reducing the worst case latencies. It can be seen that even though the number of links is more in the topologies synthesized with the RT algorithm, the overhead with respect to the original case is on average only 9.5%. In Figures 6.13, I show the corresponding plot on the number of links in the topology for the $D36\_4$ benchmark. Since the communication pattern of this benchmark involves more flows, there are more links in the original design as well so the bounds can be met with even smaller overhead (average 5.5%). To give a more insight on how the topologies change when they are designed with worst-case latency constraints, I present in Figure 6.14 a topology

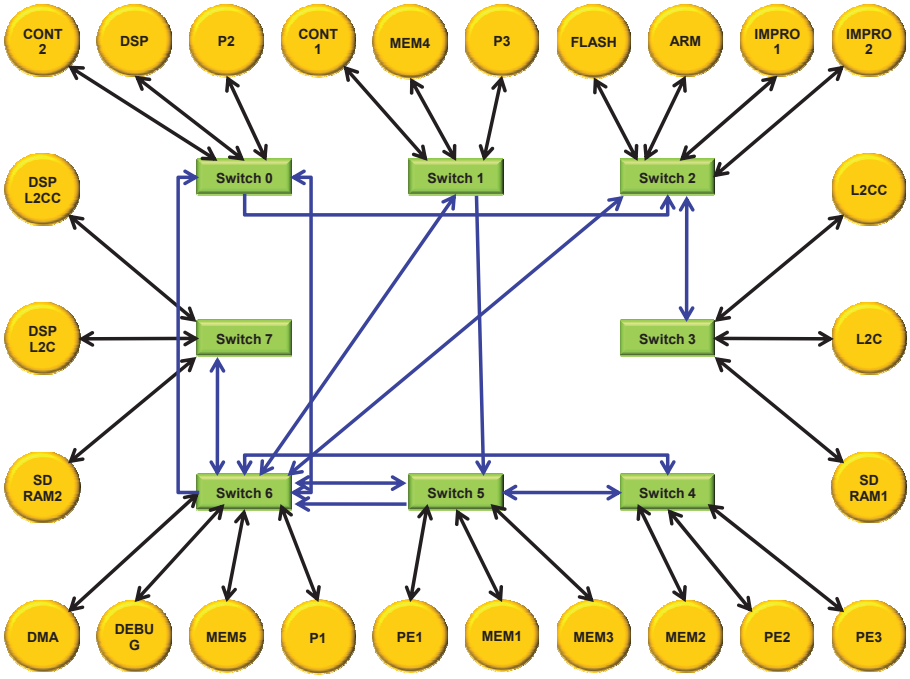

Figure 6.14: Topology with 8 switches for *D*26_*media* designed with the original algorithm

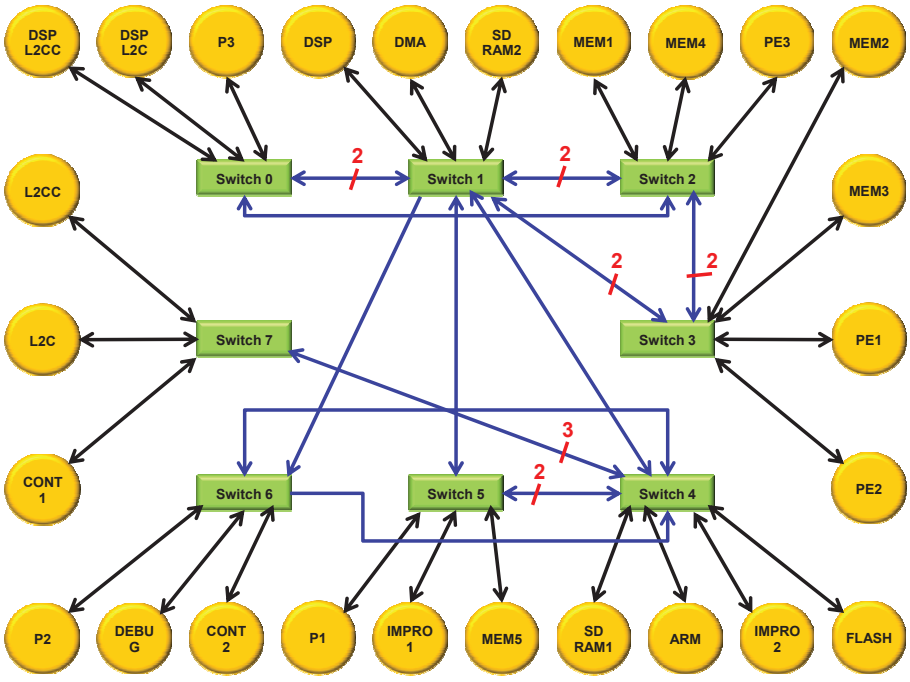

Figure 6.15: Topology with 8 switches for *D*26_*media* designed with the real-time algorithm

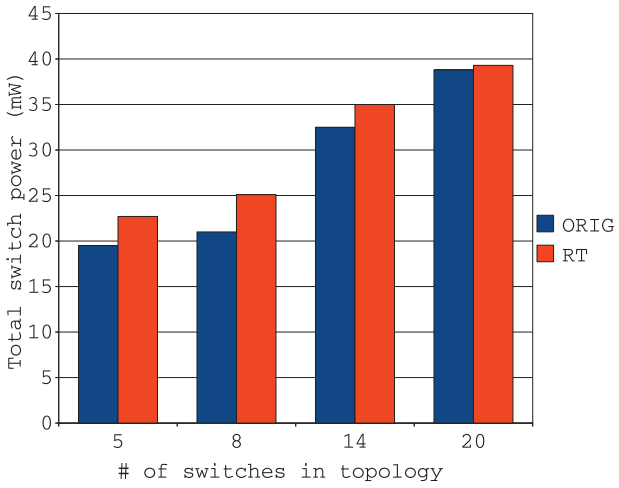

Figure 6.16: Switch power consumption for $D26\_media$

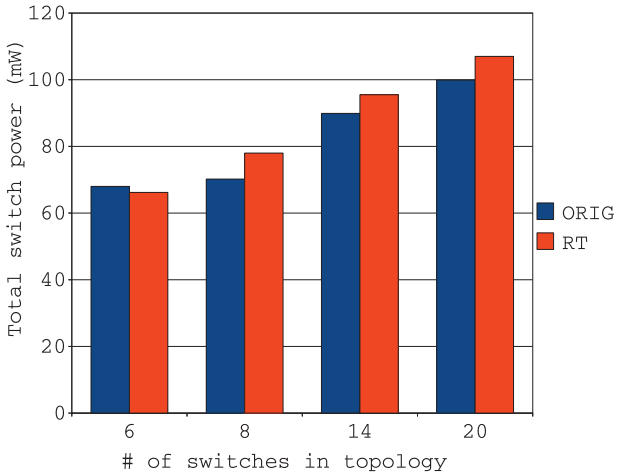

Figure 6.17: Switch power consumption for $D36\_4$

designed with the original algorithm and in Figure 6.15 the topology for the same design point, but constructed with the real-time algorithm. The topologies correspond to the design point with 8 switches for the $D26\_media$ benchmark.

6.5.3 Effect on Power Consumption

In this section, I analyze the overhead of the RT algorithm with respect to the original algorithm in terms of the power consumption of the NoC topologies. To calculate the power consumption of the designed NoC, I used the NoC component models from [160]. The power consumption of the different switches are obtained from placement&routing of the RTL designs in 65nm technology process. Switches of differ-

Bench-mark	Num of Switches	U-bound (cycles)		Z-load latency		Num links		Power (mW)	
		RT	ORIG	RT	ORIG	RT	ORIG	RT	ORIG
*D*26_*media*	5	112.1	167.3	3.7	3.9	75	64	22.7	19.5
	8	113.8	205	3.5	3.77	85	75	25.1	21
	14	105.9	289.9	3.5	4.71	97	92	35	32.5
	20	108.9	195.6	3.7	4.71	105	103	39.3	38.8
*D*36_4	6	143.5	331.7	3.27	3.85	104	99	66.2	68
	8	107.7	158.8	3.27	3.36	125	107	78	70.2
	14	91.1	138.8	3.25	3.49	172	143	95.5	89.9
	20	92.4	169.6	3.24	3.54	184	160	107	99.9

Table 6.1: Results summary

ent sizes are synthesized for different operating frequencies and switching activities. Based on the power estimates obtained after place and route, the switch power models are built. After the topology is built, based on the requirements from the communication specifications, the activity at the switches can be determined. Based on the switching activities and the power models, the actual NoC power consumption for the application is calculated.

The *RT* algorithm has the biggest impact on the switch power, because by adding more links the size of the switches is increased, leading to an increase in the power consumption. The link power is affected by the link lengths and the switching activity (the amount of bandwidth that flows on the link). The length of the links depends on the floorplan and the parallel links introduced by the *RT* procedure will have the same length. Since the bandwidth of the flows remains the same as it depends on the application, when you add more parallel links the bandwidth of the flows is distributed among all the links. Therefore in the *RT* design you have more parallel links at lower switching activity and with the same length and as such the total power for the links is similar. Also the switching activity of the switches remains constant (as the bandwidth is application dependent) even though the *RT* procedure adds more ports (to add the parallel links). However since the power of the crossbar does not grow linear with the number of inputs and outputs we see an increase in the total switch power consumption. Thus, I only plot the switch power for topologies. The power consumption values for the *D*26_*media* benchmark are shown in Figure 6.16. As can be seen, the switch power increase due to the RT process is only marginal (11% on average). In Figure 6.17, I show the power consumption for the *D*36_4 benchmark. The power overhead is slightly lower (6% on average) in this case, as fewer links need to be added to meet the constraint.

Please note that an average the switch power account for about one third of the total power of the topology, the rest being used by the links and the network interfaces (NIs). Both the power of the NIs and of the links is unaffected by the *RT* procedure and

therefore the impact of the *RT* design on the total power of the topology is minimal, while at the same time providing hard-latency guarantees for the flows.

A summary of all the results for the two benchmarks is presented in Table 6.1. The table reports average upper bound values for flow in cycles, the average zero load latency (in cycles), the number of links in each topology and the corresponding power consumption (in mW).

6.6 Summary

I have shown that guaranteeing worst-case latencies for communication flows in a best-effort NoC fabric can be done without specific hardware support, by judiciously synthesizing a QoS-aware topology. The proposed method gives worst-case latency guarantees by carefully designing the application-specific topologies for the NoC and can be used with switches that do not have any specialized hardware for QoS. The only assumption is that the switches use a fair arbitration policy, such as round robin. In the experiments, I show that my proposed algorithm can guarantee meeting real time latency constraints with little resource and power consumption overhead (average 8.5%).

7 A fast simulation model for soft QoS

Existing tools have been proposed to design application specific NoC, but the methods presented so far in this thesis do not take into account an accurate evaluation of NoCs dynamic performance during synthesis. That is partly because accurate average-case performance evaluation of the NoC requires extensive simulation. Therefore most design flows involve multiple iterations between synthesis and simulation to achieve the required performance. To achieve fast design closure there is a need for a fast and accurate evaluation model that can be integrated with the automatic design of the NoC.

To effectively tackle the design complexity, designers use a platform-based approach, where a single platform instance is used across different product versions with incremental changes across the versions. With such an approach, the platform is verified and validated once and re-used across different products. Another way to reduce the design complexity is to orthogonalize the major design issues, especially with respect to the design of the on-chip computation and communication architecture [81]. The choice of the communication architecture and performance validation is performed using abstract models of the computation architecture. For example, the computation cores are modeled as traffic generators that inject a trace or a pattern based on the application characteristics, which are used for simulating the communication system. While the designer can work on evaluating the different computational architectural choices, he/she can in parallel build the communication system. Such an approach eases the overall validation effort, as the computation and communication architectures are individually verified and put together. However, during the design phase, the computational models change according to the different choices made for the computation architecture, which can lead to incremental changes in the requirements for the communication architecture.

Simulations play a major role in the NoC design process. The congestion in a network has a significant impact on the NoC performance metrics (packet latencies and

injected bandwidths). Packet-level simulations are used to obtain accurate NoC performance metrics, as they model the dynamic effects of congestion in the network. As there is a large degree of freedom for the various NoC parameters, several simulation runs are needed before setting the different parameters.

In general, packet simulations in a NoC are performed by instantiating the entire NoC topology. That is, when the simulation model is generated, all the switches, the *Network Interfaces* (NIs) and frequency/data-width converters are instantiated. To all NIs, either the actual core or a traffic generator modeling the traffic behavior of the core is added and traffic is injected according to the application specification. After running the simulation, various performance metrics, such as the injected bandwidth and latency values of all the flows in the topology are available.

However a wide variety of usage scenarios require information only for a single traffic flow or few flows at a time. Examples include simulations performed during NoC topology synthesis, simulations after incremental design or application specification changes and for trade-off analysis. I explain some of these scenarios in detail in the following subsections. For all these cases, performing a full NoC simulation to obtain the performance metrics for just a single or few flows leads to a large timing overhead. Reducing the time-to-market is a major concern for *System on Chips* (SoCs) and a major portion of the design time is spent in simulations during the architectural design phase. Thus, achieving a fast simulation set-up is an important problem that needs be addressed and it is the subject of this chapter.

7.1 Application scenarios for fast simulation and contributions

There are several scenarios, that appear during the NoC design phases that can benefit from fast performance evaluatiation. Two frequent ones, addressed in this chapter are described in the following sub-sections. The contributions of this chapter focus on providing fast performance evaluation for these scenarios and are also detailed in a separate subsection.

7.1.1 Simulations as part of topology synthesis

In NoC topology synthesis, the number and size (number of inputs, outputs) of switches and the connectivity among them and the cores are determined. The process also determines the paths for the different traffic flows. The objective of the synthesis process is to obtain a NoC with minimum power consumption and/or area, meeting the application power-performance constraints. Many NoC synthesis algorithms map one flow at a time, where the mapping process for a flow determines the topology connectivity and paths for the flow [121], [62], [149]. The topology could also be pre-

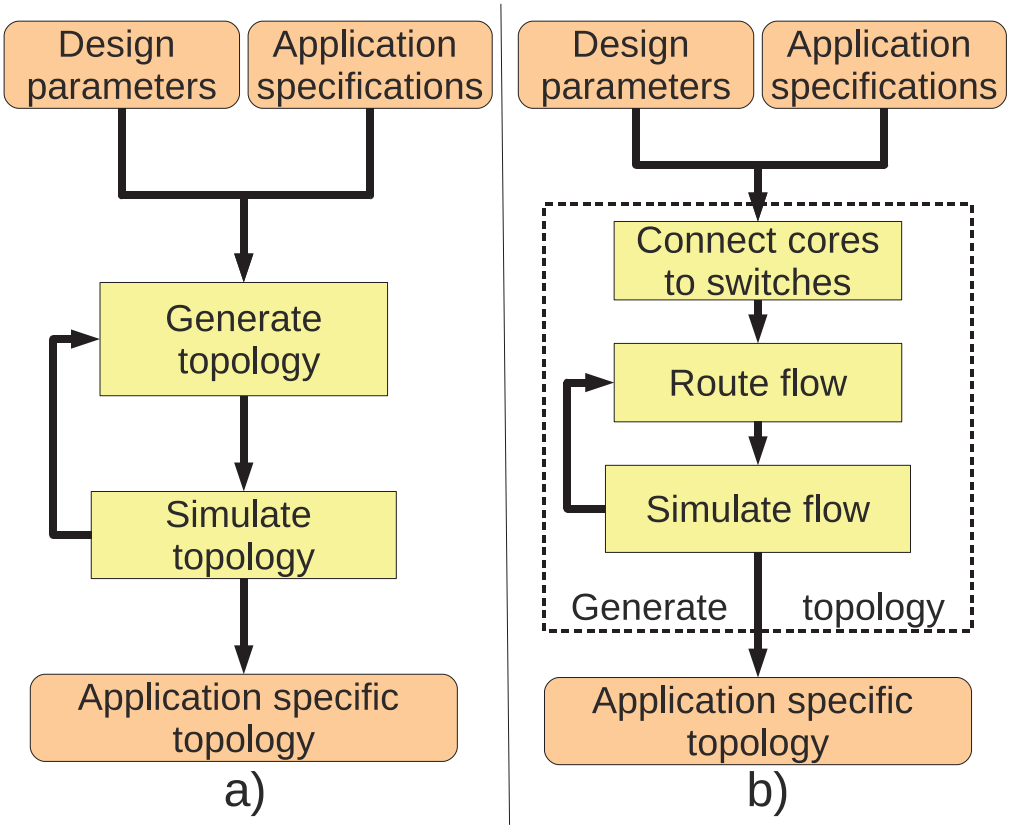

Figure 7.1: Synthesis methods: a) traditional, b) with partial simulation

defined (such as mesh), in which case the mapping process for a flow determines the path for the flow [123]. In Figure 7.1.a, I show the traditional two-step approach used for topology synthesis and mapping of flows. In the first step, the topology synthesis is performed and during the synthesis process, the injected bandwidth and latency values for flows are estimated based on analytical models. In the second phase the NoC topology is simulated, after the full network was synthesized. The major problem with such an approach is that it fails to capture the dynamic simulation effects, such as packet contention and arbitration policies that have a significant impact on the network performance during synthesis. This leads to a discrepancy between the estimated performance metrics during synthesis and the measured values when the network is simulated. If the network does not meet the performance constraints after simulation, the entire topology synthesis process needs to be performed again. Re-designing the whole NoC topology can lead to over-design in-terms of power consumption and area. Moreover, it is difficult to achieve convergence: that is ensuring that the designed network can actually meet the constraints during simulation. Since NoCs are so widespread in industrial usage now, there is a need for optimization support, to help tune and add value by differentiating.

An alternative to this approach is to perform simulations during the topology synthesis process itself, that is after mapping each flow. This is illustrated in Figure 7.1.b. After mapping a flow, a partial network is available, with some switches and connections between them and the cores. The partial network can be simulated, considering the current and previously mapped flows and accurate performance metrics can be obtained. The synthesis algorithm should also be modified to support this feature. When mapping a flow, if the current or previously mapped flows do not meet the constraints, the route as well as the connectivity described by the route, should be re-established such that the design moves closer to satisfying the performance constraints for the mapped flows. As the iteration with simulation phase is performed during the synthesis process itself, the design convergence between synthesis and simulation can be achieved more easily. However, the main challenge here is that simulating a NoC topology is a time consuming process. A full simulation of a NoC can take anywhere between few minutes to several hours, depending on the amount of traffic trace to be simulated. A typical SoC may have few tens to few hundreds of different flows in the application. If the network designed so far has to be simulated after routing each flow, it can lead to a significant timing overhead for the tool. In order to make an integrated synthesis-simulation process feasible, we need an efficient method where by the simulations can be performed in a much faster manner.

7.1.2 Simulations during design changes

The need for fast simulations also occurs in many other instances. The SoC designer may want to obtain the impact of changing certain parameters on the NoC performance. For example, in many cases, the designer would like to evaluate the impact of changing the frequency of a processing core and hence its injection bandwidth on the NoC performance. The designers also usually perform extensive simulations for trade-off analysis, where they vary the input application requirements and architecture points. In such cases, between any two design points, there is only a marginal change (such as the performance requirements of a flow). With the orthogonalization of design concerns [81], where in the communication and computation architectures are designed independently, as the design cycle progresses the requirements for the communication architecture may vary marginally. Moreover, with a platform based design approach, there is usually a marginal change in the platform when adopted to different product domains. The problem with the current NoC simulation models is that even if there is a marginal change in the application or architecture (such as only a single flow's traffic characteristics are changed), the whole network needs to be re-simulated.

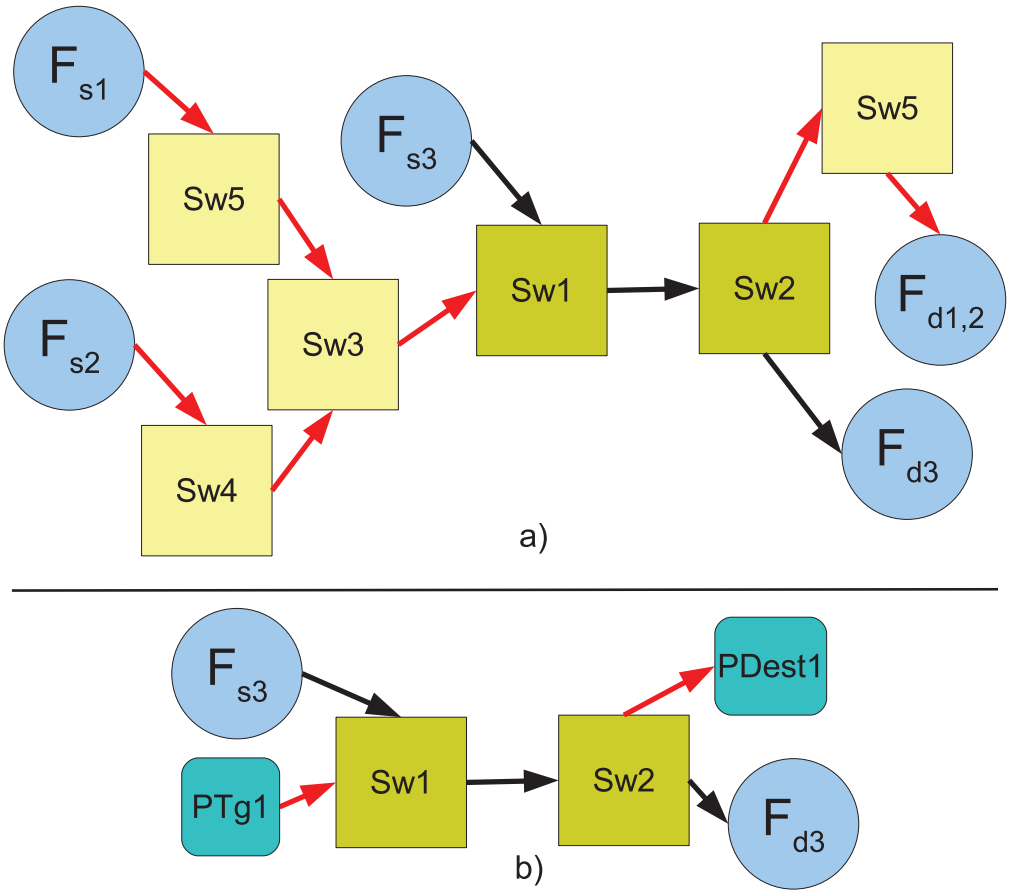

Figure 7.2: Example of: a) topology where a new flow has been added from F_{s3} to F_{d3} depicted by the black arrow and the state of the previous topology is know depicted by the red arrows; b) partial simulation setup for the new flow with the corresponding partial traffic generators and destinations that replace the rest of the topology

7.1.3 Contributions

Ideally when a new flow is added to the system or when the traffic characteristics of a flow is changed, we would like to simulate the packet transmission for only that particular flow. However, the packets of the flow considered will contend with the packets from the other flows and it is imperative to consider the contention effect to accurately measure latency. Moreover, with a back-pressure mechanism, such as the credit-based flow control, packet transmission of even flows with paths that are completely disjoint with the considered flow can affect the contention of the packets with the current flow. Note that ignoring such contention effects would not only lead to inaccurate latencies for the current flow, but also for the previously mapped and simulated flows as well. Such a secondary contention effect should also be considered.

This is the reason why in current systems the entire network with all the traffic flows need to be re-simulated, even when a single flow is added or its characteristics are changed.

In this chapter, I present a new simulation method for NoCs, which I call *Partial Simulation*. The proposed method can dramatically speed-up simulations for the cases when new traffic flows are added or when the traffic characteristics of some of the flows are changed. The method provides accurate performance estimates of traffic flows in the network, similar to that of a full network simulation. I also present a topology synthesis approach, so that the *Partial Simulation* method can be integrated within the synthesis process.

The main idea behind the proposed method is to track and build models during simulations for traffic injected/ejected from each switch input/output port and for the stall encountered on the links and use them to rapidly perform partial simulations. I track the bandwidth of the flows as seen at each input of every switch and the stall that is observed at each output of every switch during simulations. When a new traffic flow is added to the network or when a existing flow characteristics are changed, I simulate the packet injection only on the switches in the path of the network that is used by the flow. I add a traffic injector for the flow (or the actual core itself) at the source switch and a traffic ejector (or the destination core itself that consumes the data) at the destination switch. I also add injectors for modeling the traffic other traffic flows that start from the same input port of the source switch and ejectors for the output port of the destination switch. For each input port of each switch on the path of the simulated flow, but is not used by the flow, I add traffic injectors that emulate the behavior of the traffic at that port. For each output port of the switch, I add ejectors that model the traffic ejection. This set-up is then simulated. An example of this is shown in Figure 7.2.

However, due to the back-propagation of congestion, flows that do not use any of the switches of the considered flow will also be affected. We observe that in NoCs that use a flow-control mechanism, a fundamental variable that can be tracked during simulations that affects the latency of transfer is the amount of stall encountered on the links. When there is more contention among flows, more is the stall on the contending links. The more stall on the link, the more is the latency for transmission across the corresponding switch and link. With a flow-control mechanism, stall on a link at a downstream router is propagated back to the upstream routers as well. The amount of stall propagated back not only depends on the stall value at the link, but also on the bandwidth of traffic flows at the link.

I build static models for the propagation of stalls from the links in the simulated path to the links in other switches that are not part of the path. I also build models to

update the latencies of the already mapped flows based on the calculated stalls.

The major steps for performing partial simulation are presented in Figure 7.3. There are two main phases, one is to perform the pre-characterization of the network architecture and the other is running the actual simulation. The pre-characterization of the network architecture is performed to model the effects of the flow control, arbitration and back-pressure on the stall and its propagation back to the upstream links. These models are necessary to update the stall information for the flows that are affected by the partial simulation, but are not simulated in that run. I also build models for estimating changes in latency with the changes in stall. This pre-characterization step only needs to be done once for each NoC architecture. Details on how the models are built is detailed in Sections 7.2.2 and Section 7.2.5.

During the simulation phase, the entire network is simulated and the traffic injection/ejection models and stall values at the switches and links are obtained. When a new flow is added or a flow's characteristics change, the partial path is simulated as explained above and the new stall values are obtained on the links that are part of the path. The injection/ejection models at the switches of the path are also updated. Then, based on the static models built in the pre-characterization phase and the injection/ejection models built at the switches, the stall values are propagated to links that are not part of the simulated path. To calculate the latency of all flows, I track the latency of packets at each component (switch, NI). To calculate the latency of a flow it is necessary to sum up the latency of the components along the path of the flows. By keeping track of latencies at component level, the latencies of previously simulated flows that intersect with the current flow being simulated are also updated, for the switches that are simulated in path. Calculating latency is detailed in Section 7.2.5. Using the models from the pre-characterization phase, the latencies of the flows that are not part of the simulated path are updated using the propagated stall values. The partial simulation steps are detailed in Section 7.2.3.

I present experiments on two benchmarks (a realistic SOC and a bandwidth intensive synthetic benchmark) and I show that we can achieve large speedups between partial simulation and full simulation (between 4.5× to 10×) with a small error on the latency estimation (on average 8.6%). I also present how a traditional topology synthesis can be modified to account for simulations during the synthesis process itself. Such an integrated synthesis-simulation process leads to fast design convergence.

7.2 Partial simulation

Before I present the outline of the various phases in the partial simulation, I state the assumptions for the models in the following sub-section.

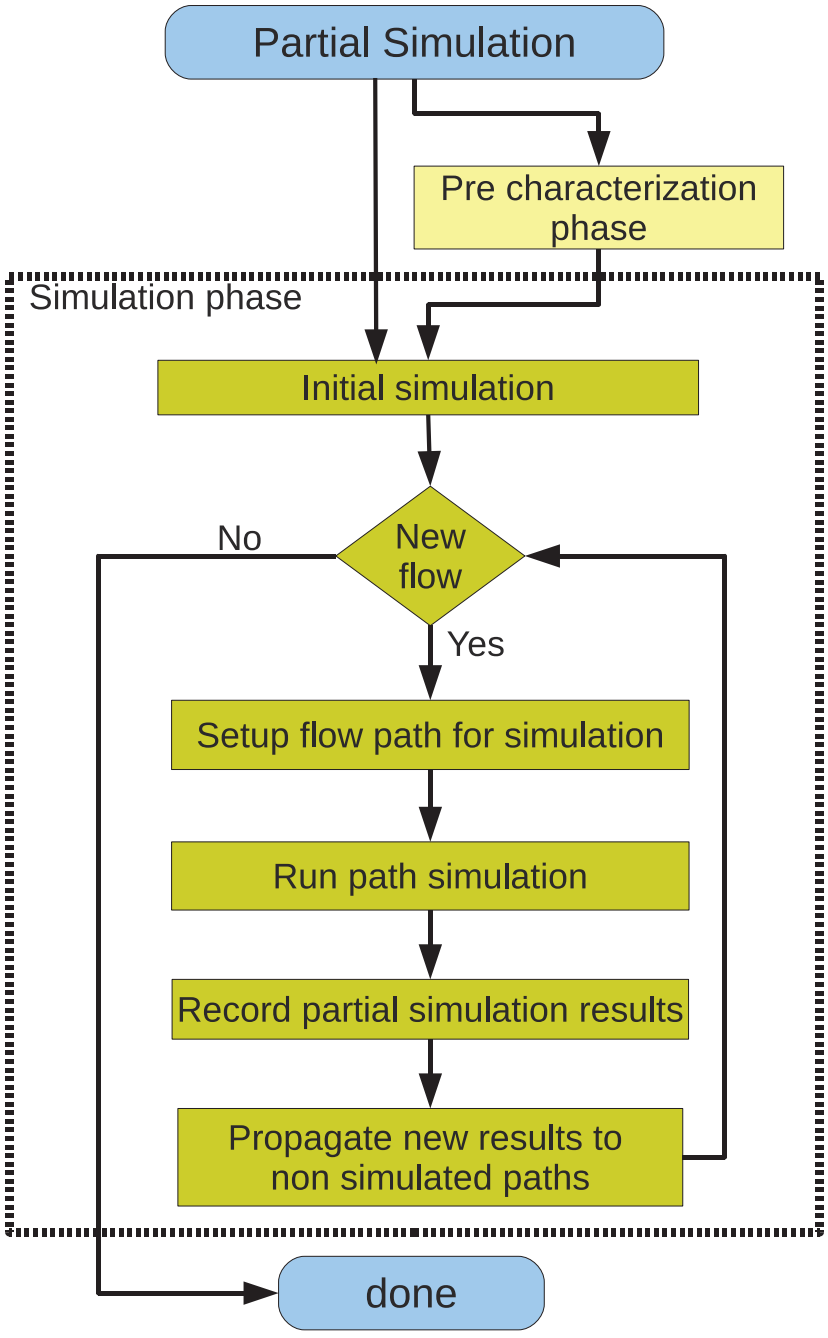

Figure 7.3: Partial simulation steps

7.2.1 Assumptions

I make the following assumptions for the NoC for this work:

- A flow control mechanism is used and packets are not dropped; this assumption holds for almost all NoC designs.
- Arbitration is pre-specified at the switches, the method can be used with several arbitration schemes (round robin, fixed priority);
- Even though the proposed simulation method is applicable to a wide variety of routing schemes including adaptive routing, in this chapter, I show the application only to static routing. In such a scheme, the paths for the traffic flows are determined at design time and all packets from a single source to a destination use the same path.
- The NoC switches have a fixed number of buffers at the input and/or output ports. The static models that track the propagation of stall and latency depends on the amount of buffering at the ports. In this chapter, I show how the models are built when the buffer sizes are uniform across all the ports. The models can be easily extended to account for variable sized buffering at the ports.

While I use a base NoC architecture based on [160] to illustrate how the model are built, the methods are generic and can be applied to any NoC architecture that satisfy the above assumptions.

In the next sub-sections, I provide a detailed description of the different phases of the partial simulation method.

7.2.2 Pre-characterization phase

To simulate only partial sections of the network (i.e. only the path of a communication flow), we need to keep the state of the network from previous simulations. To maintain the state I track the stall at the outputs of the switches in order to accurately simulate the contention. However increasing the contention in the simulated part of the network will result in an increase of the contention in the non-simulated part of the network as well. Since we cannot measure the increase of the stall in the non simulated parts, I build an analytical model to estimate the increase of the stall in the rest of the network.

When a new flow is added, the contention created by this flow will be seen as an increase of the stall at the outputs of the upstream switches that have flows that intersect. For example in Figure 7.2.a when we add the new flow 3 switch $Sw3$ will observe

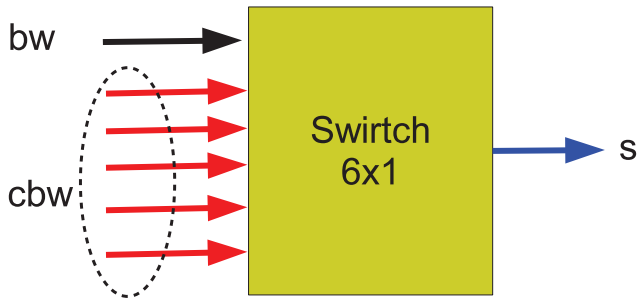

Figure 7.4: Single switch model

an increase of the stall at the output due to the contention of flow 3. This increase of the stall can be measured, but has to be propagated to the outputs of switches $Sw4$ and $Sw5$. So in the general case, we have to propagate the stall from the output of a switch to the inputs that send packets to that output. The increase of the stall cannot be propagated directly as the buffering in the switches will tend to decrease its effect. Also it will not propagate the same value to all inputs, which is dependent on bandwidth. For example a low bandwidth flow will have a higher probability to find the buffer empty so it observes a lower stall. However if the contending flows have high bandwidth it increases the chance even for a low bandwidth flow to observe a higher increase of the stall. To capture all these effects, I build a model that considers as inputs the bandwidth (bw) coming from the input to which we want to propagate, the contending bandwidth (cbw) of the other inputs and the increase of the stall (Δs) observed at the output as shown in Figure 7.4. From simulation I observed that we can consider the aggregate of the contenting bandwidth as coming from a single input, to simplify the model.

I have performed extensive simulation and I found that the second-order analytical model of Equation 7.1 (see below) can best reproduce the behavior observed in simulation. The function Δps^{buff} returns the increase of the stall that has to be propagated. The parameter $buff$ represents the amount of buffering of the switch. If there are switches with different buffer sizes than for each buffer size a model has to be build as the parameters a, b, c, d and e model the effect of buffering in the switch. These parameters are determined by fitting the general model on the experimental data that I collect from simulation on a single switch when I vary the values of bw, cbw and Δs. For example for the switch architecture used in the experiments with 2 FLIT deep input buffers I found the following parameters: a=-468.6, b=45.5, c=-191.2, d=13.44

Algorithm 7.1 Propagate_Stall(sw, op, Δs)

1: $stall_{sw,op} = stall_{sw,op} + \Delta s$
2: **for all** input ports ip of switch sw **do**
3: **if** bandwidth bw from ip to op is larger than zero **then**
4: Find previous switch sw_p and output port op_p connected to ip
5: **if** sw_p exists **then**
6: calculate contenting bandwidth cbw
7: Propagate_Stall(sw_p, op_p, $\Delta ps(bw, cbw, \Delta s)$)
8: **end if**
9: **end if**
10: **end for**

and e=-0.017.

$$\Delta ps^{buff}(bw, cbw, \Delta s) = ((a \cdot bw^2 + b \cdot bw) + (c \cdot cbw^2 + d \cdot cbw) + e) \cdot \Delta s \quad (7.1)$$

In Algorithm 7.1, I show how Equation 7.1 is used recursively to propagate stall. The function $Propagate\_Stall$ takes as input the switch and the output port to which the stall needs to be propagated and the third parameter is the increase of the stall that is propagated. The function will then propagate the stall from that output to all inputs from where flow are coming to that output. Please note that there is a one to one correspondence between an input port and the output port of a previous switch so propagating to on input actually means propagating to the output port of the previous switch. Initially the function is called for all the inputs of the switches that are simulated and are not on the path that is currently simulated as it will be showed later in Algorithm 7.2. The increase of stall that is passed to the function at the first call is calculated from the measured values from simulation. The function is then called recursively (step 7) to propagate the stall to the entire sub network that was not simulated. The function loops on all the inputs of the current switch (step 3) and if there are flows going from that input to the output on which the stall was propagated (steps 4) will attempt to find the previous switch from where the flows are coming. Provided that the switch exists (step 5) and the edge of the network was not reached, it will recursively call the function (step 7) with the new value of the stall increase calculated with Equation 7.1.

The contention created by the new flow does not only affect the observed values of the stall at the outputs of the switches, but also the latency of the intersecting flows. It is well known that the dependency between contention and latency has an exponential nature[127]. The parameters of the exponential model are dependent on the bandwidth and contention bandwidth. Since the model is exponential any error in es-

timating the parameters from the measured bandwidth will lead to even larger errors for the estimated latency. Therefore we cannot build an accurate model to propagate latency as was the case for the stall. However during the pre-characterization phase, I build an approximate model that depends only on stall to estimate the worst case latency increase. This model can then be used to estimate the worse case increase of error between the flow latencies that I can estimate and the reality. This model can be used as a trigger to run full simulation to update the latencies of the previously simulated flows. This synchronization will be discussed in Section 7.3.1. To be less conservative in estimating the error a family of models can be used for different intervals of the measured bandwidth and contention bandwidth. Since the model is used only as a trigger, I found that using a single model provides acceptable performance. The general model is represented by Equation 7.2 where a, b and c are the parameters that we need to find and s is the stall value for which we want to estimate the latency.

$$latency = a \cdot e^{b \cdot s} + c \tag{7.2}$$

From simulation, I record the value of the latency when I vary the stall for different values of the bandwidth and of the contending bandwidth. Then I fit the model and find a, b and c such that the model overestimate the increase of the latency when compared to all the curves that I got from the experimental data. Considering the same switch with 2 FLIT deep input buffers, I found the following parameter values: a=9.98, b=3.76 and c=3.21.

7.2.3 Simulation phase

To explain the way partial simulation is performed to analyze the performance of a single flow I have to define the *topology graph TG* and the *route graph RG*. The topology graph is defined in a similar manner in Chapter 5, but for completeness I redefine it as follows:

Definition 7.1 *I define the topology graph as a directed graph $TG(B,L)$ where the vertices $b_i \in B$ represent the blocks in the topology (cores and switches) and there is an edge $l(b_i,b_j) \in L$ if there is a link connecting block b_i to block b_j with $i,\ j = 1 \ldots N$ and N represent the number of cores and switches in the topology.*

Based on the definition of the topology graph I define the *route graph* as a subset of the topology. The route was also defined in Chapter 5 as a set. However since I want to capture other properties, I redefine it as a graph in this chapter:

Definition 7.2 *I define the route graph as a directed graph $RT(S, C, fl)$ as the subgraph of the topology graph $TG(B, L)$ used by flow fl. The vertices $s_i \in S$ represent the blocks from the topology used by the flow fl and the edges $c(s_i, s_j) \in C$ represent the links used by the flow. The function $ip(s_i)$ returns the input port and the function $op(s_i)$ returns the output port used at block s_i. the function $isSwitch(s_i)$ tells whether the block is a switch or a core.*

Example 7.1 *In Figure 7.2.a, I show an example of a topology graph with 11 vertices (5 cores and 6 switches) and three routed flows (flow 3 being the last one routed). In Figure 7.2.b, I show an example of a route graph for flow 3 corresponding to the topology in Figure 7.2.a. There are two cores the source F_{s3}, the destination F_{d3} and two switches $Sw1$ and $Sw2$.*

In order for the path simulation to be accurate apart from simulating the switches along the path and the NIs at the source and destination, I also need to model the traffic that intersects with the current path. The off-path traffic is simulated by attaching partial traffic generators (PTg) connected directly to the switch input ports. The partial traffic generators (PTg) model the traffic that was recorded at that switch input from previous simulations. A detailed description what information is recorded and how traffic is generated is presented in Section 7.2.4. As partial traffic is injected, we also have to model the contention observed by that traffic when exiting the simulated switches. Partial destinations ($PDest$) are added on the output ports of the switches to sink in the traffic going out. The partial destinations also have to model the stall that was recorded from previous simulations in order to improve the accuracy of the partial simulation.

Given a topology graph $TG(B, L)$ and the flow fl to simulate, the main steps for setting up the partial simulation are presented in Algorithm 7.2.

The first step is to find the cores, switches and links that are on the path of the flow to simulate and to instantiate them for simulation (step 3,4). Then, the algorithm looks at all the routed flows in the topology (step 5). If a flow intersects with the current flow that we want to simulate (step 7), then it is necessary to add a partial traffic generator (PTg) on the first switch where the intersection happens and a partial destination ($PDest$) on the last switch (steps 10 and 14). Please note that if the flow has the same source or the same destination as the current flow we simulate no PTg or $PDest$ needs to be added as the traffic is handled by the actual core that we simulate. Once the path is setup and the $PTgs$ and $PDests$ are added we run the simulation for M cycles which is given as input (step 18). After the simulation is completed, the algorithm looks at all the $PTgs$ (step 19) and propagate the recorded increase of the stall using the recursive method from Section 7.2.2 (steps 20-22).

Algorithm 7.2 Partial_Simulation($TG(B,L)$, fl, M)

1: {Get the subgraph of the $TG(B,L)$ used by fl}
2: $RT(S_{fl}, C_{fl}, fl) = \{s_i \in S_{fl}\ and\ c_i \in C_{fl}\ |$
$s_i \in B\ and\ c_i \in L\ and\ s_i, c_i\ are\ used\ by\ fl\ i = 1 \ldots |L|\}$
3: Setup the simulation for $RT(S_{fl}, C_{fl}, fl)$
4: $P = \oslash$
5: **for all** routed flow $f \neq fl$ **do**
6: $RT(S_f, C_f, f) = \{s_i \in S_f\ and\ c_i \in C_f \mid s_i \in B\ and$
$c_i \in L\ and\ s_i, c_i\ are\ used\ by\ f\ i = 1 \ldots |L|\}$
7: **if** $|RT(S_{fl}, C_{fl}, fl) \cap RT(S_f, C_f, f)| > 1$ **then**
8: $S = S_{fl} \cap S_f$
9: **if** $isSwitch(s_0)$ **then**
10: Add PTg to $ip(s_0)$ with $s_0 \in S$
11: $P = P \cup PTg$
12: **end if**
13: **if** $isSwitch(s_{|S|})$ **then**
14: Add $PDest$ to $op(s_{|S|})$ with $s_{|S|} \in S$
15: **end if**
16: **end if**
17: **end for**
18: Run simulation for M cycle
19: **for all** $ptg \in P$ **do**
20: Get switch sw and output port op replaced by ptg
21: Get recorded increase of the stall Δs from ptg
22: Propagate_Stall(sw, op, Δs)
23: **end for**

Example 7.2 *In Figure 7.2.b, I also show how the parts of the topology from Figure 7.2.a are replaced by $PTgs$ and $PDests$ when we do the partial simulation of flow 3.*

7.2.4 Traffic injection/ejection models

Traffic injection for the simulated flow is based on the average injection rate provided as input to the simulation. Traffic generation in the $PTgs$ is based on traffic rates measured from previous simulations. For that reason, during a simulation we need to collect traffic information. I refer to the data structure used to store the traffic information that I measure as the *input table*. In order to setup the $PDests$, we need to monitor the stall at the output of the switches. I refer to data structure that holds the stall information as the *output table*.

In Figure 7.5, I show for a 2×2 switch where the *input tables* and the *output tables* are attached. An *input table* is assigned to each switch input-output pair as it will store information about the packets that are going from that input to that output of the

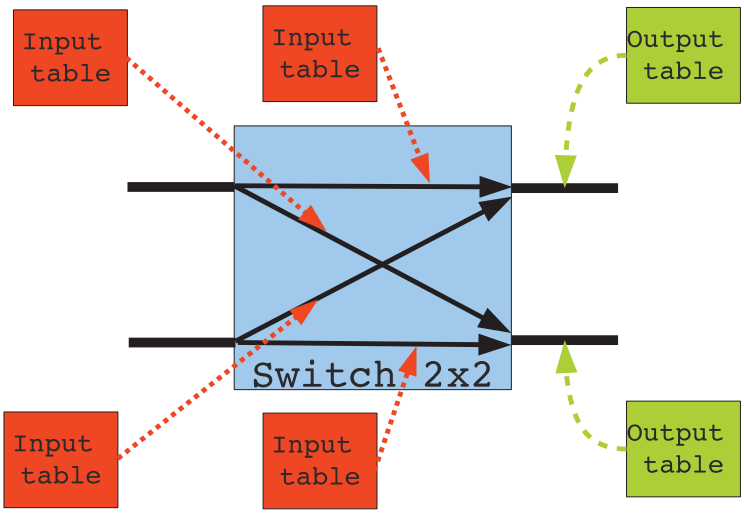

Figure 7.5: Traffic characterization

switch. Therefore for the 2 × 2 switch, we require 4 *input tables*. The *output tables* are assigned to the output ports of the switch. They measure the average stall seen at that particular port, regardless from which input the packets are coming from. Therefor we only need 2 *output tables*.

The information that is recorded in an *input table* is detailed in Table 7.1. For each flow that goes from the input to the output of the switch that is monitored by the *input table* the source, destination and size of the packet is recorded. This information is required by the PTg in order to generate valid packets that can be routed on the correct path. Since the simulation supports the use of multiple frequency island and frequency converters are required when crossing domains, I also track the average number of empty flits that the converters generate. It is important to track the empty flits separately and not include them in the packet size because in the presence of downstream contention these flits can be squashed. The rate at which the packets arrive is also recorded, as this may be different from the specification in case of high

Variable name	**Type**
Source	list
Destination	list
Size	list
Empty flits	list
Rate	list
Previous output	pointer

Table 7.1: Input table data structure

Variable name	Type
Stall	value
Previous outputs	list

Table 7.2: Output table data structure

contention, when the required bandwidth cannot be provided. The *input table* also keeps a pointer to the *output table* that corresponds to the previous output port, where the packets are coming from. This information is used by the PTg to back-propagate the stall information that it has recorded during simulation. Details on how stall is back-propagated are given in Section 7.2.2.

Base on the information from the *input table* a PTg will generate traffic for each flow with uniform distribution and with an average equal to the rate of each recorded flow. The PTg will also add empty flits to each packet (based on the average number of empty flits recorded) in order to simulate the effect of any upstream frequency converters.

To record the stall, I assign an *output table* to the output ports of every switch. The information that is recorded in an *output table* is detailed in TABLE 7.2. The *output table* is simpler as it record the percentage of time that a flit could not be transferred to the next input port due to contention. Stall information is kept global and no distinction is made based on the input ports from which the flits came. The *output table* also keeps a list of pointer of all the previous *output table* that correspond to output ports from which packets have arrived. The list of previous *output tables* is used for the propagation of stall.

The information in the *output tables* is used by the $PDests$ to simulate the stall that would occur in the full network due to contention. From the point of view of the $PDest$ time is divided in slices of equal length. In each time slice the $PDest$ will refuse to accept flits for a number of cycles. The number of cycles in which the $PDest$ does not accept flits to be transferred relative to the size of the slice is equal to the stall percentage that was recorded in the corresponding *output table* in the previous run of the simulation. For example if the stall is 30% and the time slice is 100 cycles then for 30 cycles in each time slice the $PDest$ will not accept flits to be transferred from the switch. The length of the time slice is taken as input. For the $PDest$ to have similar behavior to the full network, the start time of the blocking cycles in each time slice is assigned randomly.

7.2.5 Latency estimation

Similarly to stall, the latency of existing flows in a topology changes when a new flow is added or if the parameters of a flow are modified. While the latency of the new flow is estimated during the partial simulation of its path, estimating the latency of the other flows is also important. In order to estimate the latency of the existing flows, I keep track of the latency to cross any component (switch, NI). Therefore to each NI I attach two *latency tables*. One table is assigned to the output port of the NI and tracks the latency of the tail (or header) flit from the time it is created until the time it is sent to the switch. The other is assigned to the input port and tracks the latency for the tail flit from the time the flit was received until the corresponding transaction is delivered to the core. I use the tail flit latency as it incorporates the packetization latency of the NI. At each switch output, I attach a *latency table* that records the waiting time of the flits in the switch input buffers. The latency values of flits coming from different inputs are tracked separately. The latency of a flow is calculated by summing up the individual values of the *latency tables* along the path.

When a partial simulation is performed, intersecting flows (whose latency is affected by the new flow) are simulated by the $PTgs$. Therefore the *latency tables* of these flows are updated by the partial simulation. So the interference of the new flow is captured in the latency of the intersecting flows.

However as the latency of the intersecting flows is modified due to the interference with the new flow, a similar secondary effect appears between these flows and other flows that they are intersecting with upstream of the intersection with the new flow. Propagating the latency changes to these flows is more complicated than in the case of the stall as the dependence between latency, stall, bandwidth and contending bandwidth follows an exponential model. Therefore any small error in the input of the model leads to a large error in the propagated value. One solution to this problem is to run the full simulation from time to time in order to update correctly all the *latency table*.

7.3 The use of simulation during synthesis

As a case study to demonstrate the usefulness of the partial simulation model, I have extended the application specific NoC topology synthesis algorithm from [121] to use the partial simulation results during synthesis. In this section, I only highlight the changes made to the algorithm in order to use the partial simulation results to drive the synthesis.

The main steps of the synthesis algorithm are presented in the earlier work [121]. For a given design point the original algorithm performs the following tasks. First it assigns

Algorithm 7.3 Route_Flow($TG(B,L)$, fl, M)

1: {Use α to switch focus from power to latency}
2: **for** α = 0 to 10 (with step = 1) **do**
3: {Calculate costs $c_{ij} \in C$ to route between any switch pair}
4: **for all** switches $sw_i \in B$ **do**
5: **for all** switches $sw_j \in B$ **do**
6: Find link l from sw_i to sw_j that supports the bandwidth of fl
7: **if** l not found or $l \in critical\_links$ and number_of_flows(l) $\geq (1-\alpha)$ **then**
8: $c_{ij} = (1-\alpha) * marginal\_power(new\_link) + \alpha * max\_cost$
9: **else**
10: $c_{ij} = (1-\alpha) * marginal\_power(l) + \alpha * max\_cost$
11: **end if**
12: **end for**
13: **end for**
14: Find least cost path using cost matrix C
15: Partial_Simulation($TG(B,L)$, fl, M)
16: **if** Constraints met **then**
17: Go to step 22
18: **else if** already routed flow fp failed **then**
19: Add the links used by fl and fp to $critical\_links$ list
20: **end if**
21: **end for**
22: **if** synchronization necessary **then**
23: Run full simulation
24: **end if**

the cores to switches and then it routes the inter-switch flows one by one. In this section, I focus on the routing routine of the inter-switch flows, where I integrated the partial simulation to improve the results. The details of the routing method are presented in Algorithm 7.3.

The routine tries to find power optimal paths and then checks the latency constraints. If the constraints are not met more importance is given to latency when the costs are computed. The routine iterates using the parameter α to make this switch from power to latency (step 2). Then the costs of routing the flows between each pair of switches is computed (step 4, 5). If links can be reused, the cost is composed as a linear combination of the marginal power increase due to the added bandwidth of the new flow and the maximum cost (step 10). By increasing the contribution of the maximum cost shorter paths will be found. If no link can be reused, then a new link has to be open and the marginal power includes the increase of the sizes of the switches (step 8). A link can be reused by a flow if it is not critical or if there are fewer flows on the link than the value of the iteration counter in this case (step 7). This will drive the algorithm to open parallel links in the worst case if it cannot meet the constraint.

The list of critical links is empty in the first iteration. However if the path that was found causes older mapped flows to miss their constraint, then the intersecting links between the new flow and the old flow are added to the critical list (step 19). This is necessary as the algorithm cannot remap an old flow so it has to prevent the new flow from causing the old flow to violate its latency constraint. Once the costs are calculated then a minimum cost path is found using Dijkstra's algorithm and turn prohibition (to avoid routing level deadlock) [121]. The path is then simulated for M cycles (step 15) and the constraints are checked (step 16).

Eventually after a certain number of flows are mapped a full simulation needs to be run in order to update the partial simulation tables (step 22). The condition to run a full simulation is checked after the flows is routed (step 22) The condition used to decide when to run the full simulation is described in detail in Section 7.3.1.

7.3.1 Improving the accuracy of the estimated latency

It is well known that the dependence between latency and contention is exponential[127]. I assume an model as described in Section 7.2.2. I use this model to assess the worst case increase of the latency due to the observed increase of the stall. Based on this estimation, the algorithm decides if it is necessary to run a full simulation and updates all the tracked information.

As we partially simulate flows, the error accumulates after the simulation at each flow and after a certain number of flows are simulated we need to run a full simulation and update the NoC state. Therefore after a new flow is simulated, using the model from Equation 7.2, the algorithm gets an estimate of the latency (lat_0) for the intersecting flows with the previous stall value from before the simulation run. Based on the new value of stall, it can estimate the new values of the latency (lat_1). The condition to trigger a full simulation based on the two estimates of latency is given in Equation 7.3, where α represents the amount of error that can be tolerated. In this work, I set α to 0.1 to tolerate 10% of error.

$$lat_1 \geq lat_0 + \alpha \cdot lat_0 \quad \alpha \in [0\ 1] \tag{7.3}$$

To prevent full simulation for being run too often, I also imposed a bound on the minimum number of flow that are simulated before synchronization (in this case I set the bound at 5 flows).

7.4 Evaluation

For the evaluation I use two benchmarks, a realistic multimedia and wireless communication SoC with 26 cores which I call $D26\_media$ [149] and a 36 core synthetic benchmark with spread traffic $D36\_4$. The $D26\_media$ benchmark is a highly heterogeneous SoC that has a multitude of different cores like an ARM processor with high level cache, a DSP processor, video accelerators, a DMA core, SDRAM and FLASH controllers, several embedded memories and a multitude of peripherals. The cores are assigned to 5 voltage and frequency island (three running at 500 MHz and two running at 333 MHz). The links that cross voltage and frequency domains must pass through a frequency converter (I assume a dual-clock FIFO). I explore three topologies for this benchmark: a small one $D26\_s$ which has one switch in each frequency domain (5 in total), a medium sized one $D26\_m$ with 10 switches in total and a large one $D26\_l$ with 15 switches. The $D36\_4$ benchmark has 36 cores. I use topologies with different number of switches because small topologies have lower power consumption, while large topologies have better dynamic properties. Therefore it would be interesting for a designer to explore topologies with different number of switches. There are 18 processors and 18 memories and each processor communicates to 4 of the memories. This benchmark is fully synchronous. I also explore three topologies $D36\_s$ with 6 switches, $D36\_m$ with 16 switches and $D36\_l$ with 24 switches.

The simulation code (used for the partial and for the full simulations) is cycle accurate and written in C++. I extended the application specific NoC topology synthesis algorithm from [121] (as explained in Section 7.3) and integrated it with the partial simulation code.

7.4.1 Speedup

As the target application for partial simulation is to provide soft QoS support during synthesis, I evaluate the speedup of partial simulation under the following assumptions. After every flow is routed we need to evaluate the latency of that flow. In one setup, I do the evaluation using partial simulation and in the other setup I evaluate the flow by full simulation of the existing topology. I calculate the speedup as the sum of the times for all full simulations divided by the sum of the times of all the partial simulation. I also evaluate the speedup of partial simulation when the precision is increased by updating the simulation tables from full simulation. For reference, I use a setup where the full simulation is triggered at constant rate (after every 30 flows) and when the full simulation is triggered by the adaptive method presented in Section 7.3.1.

In Figure 7.6, I plot the speedup for three topologies for the $D26\_media$ benchmark.

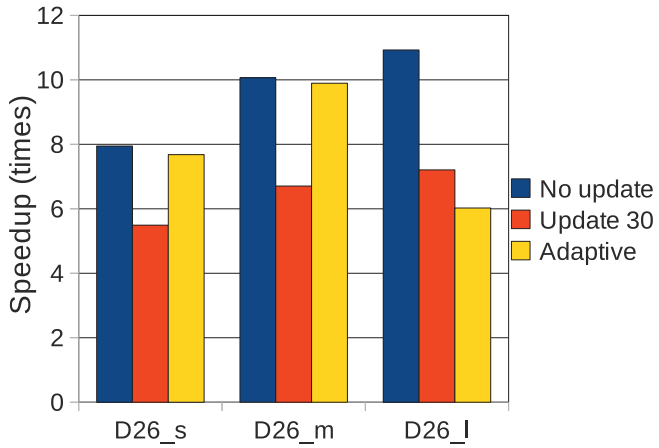

Figure 7.6: *D*26_*media* speedup

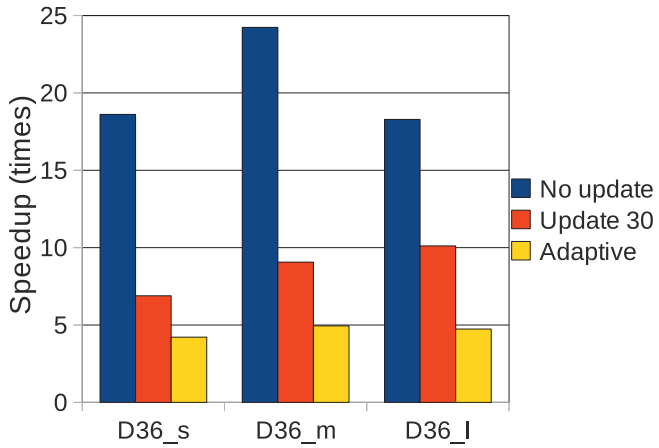

Figure 7.7: *D*36_4 speedup

As can be seen topologies with more switches benefit more from partial simulation. Using full simulation to update the partial simulation table incurs some penalty but the speedup is still significant. In case of the small and medium topologies the adaptive method has similar speedup to the partial simulation with no update, even though the full simulation is triggered more than for the constant rate method. That is because the full simulation is triggered in the beginning (as the highest bandwidth flows are mapped first), when the topology is still small and the full simulation is also fast.

In Figure 7.7, I display the speedup for three topologies for the *D*36_4 benchmark. For this benchmark as it is more complex (more cores but also more communication flows) I obtained speedups of up to 24× for partial simulation alone and up to 10× for partial simulation with full simulation update every 30 flows. Since there are more flows contending with one another the especially for the small topology with few

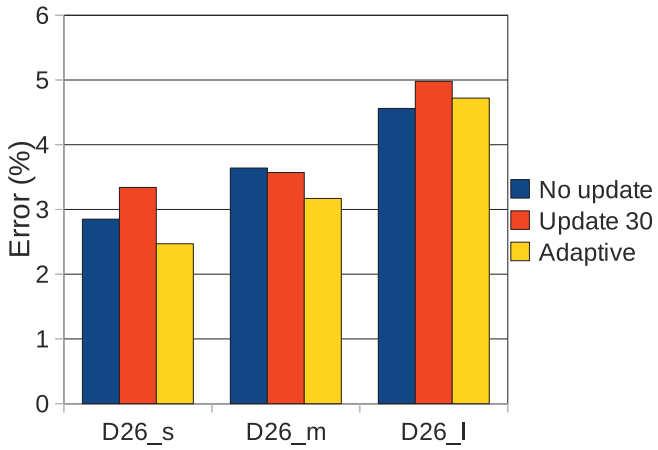

Figure 7.8: *D26_media* flow-by-flow average error

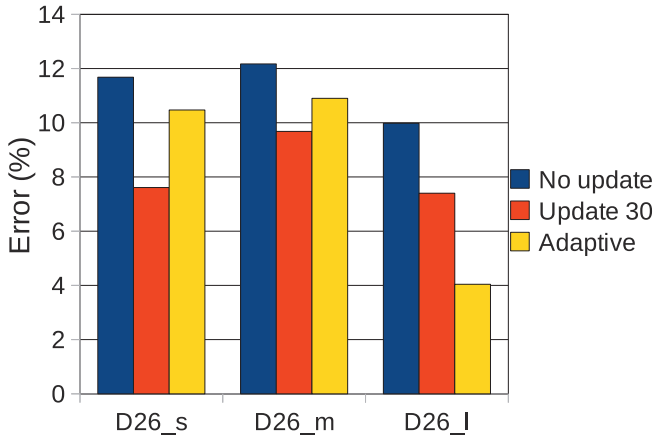

Figure 7.9: *D26_media* average error at the end

switches where many paths share the same links the adaptive method triggers the full simulation more often to reduce the error. Therefore the speedup using the adaptive method is only around 5×.

7.4.2 Accuracy

In this section, I evaluate the error between partial simulation and full simulation. I evaluate two types of errors: i) the error in latency for the simulated flow and ii) the error of the estimated latency for the previously simulated flows. In this case, I run a partial simulation and a full simulation after every flow is mapped and I compare the error flow-by-flow. After all flows are mapped I run a full simulation on the complete topology and I calculate the average error between the latency of each flow and the estimated latency calculated using the *latency tables*. Similarly, I estimate the two error

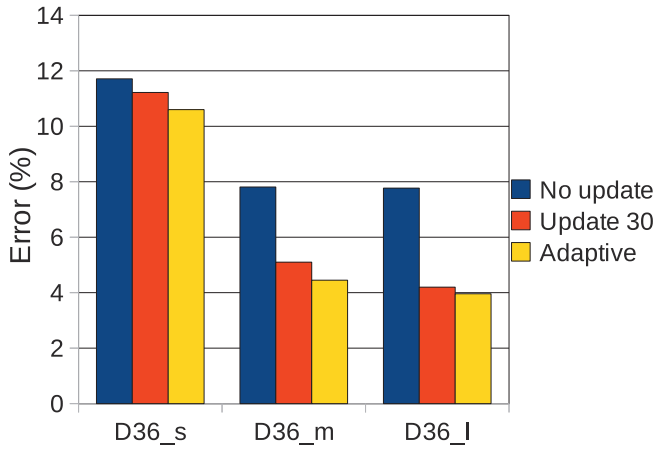

Figure 7.10: *D*36_4 flow-by-flow average error

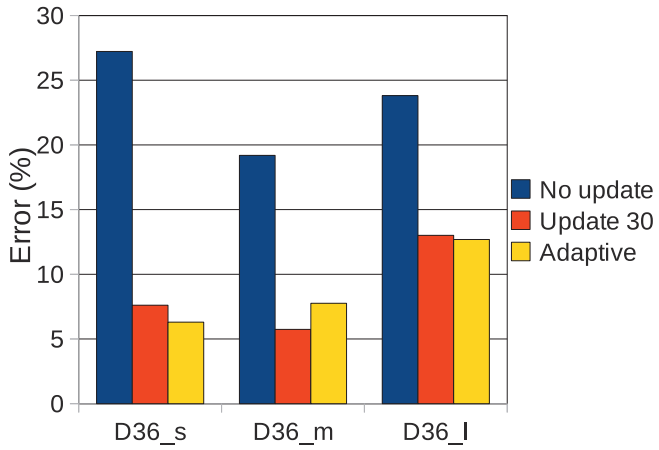

Figure 7.11: *D*36_4 average error at the end

values when I run full simulation every 30 flows and update the partial simulation tables.

In Figures 7.8 and 7.9, I show the average flow-by-flow latency error and the average estimated latency error at the end for the *D*26_*media*. If we look at the flow by flow latency error we can observe that the error is quite small (3-5%) and is slightly higher for topology with more switches where more flows are routed over multiple switches. Also synchronizing with full simulation (using the adaptive method) does not improve the results for the flow-by-flow latency error significantly. In the case of the estimated latency error we can see that for the topology with few switches the error is quite larger (12%) as there are many intersecting flows. As new flows are mapped the error in the latency estimation of the previously mapped flows grows. For topologies with more switches that have greater path diversity the error is around 10%. Using the update the error can be reduced to around 10% for the small topologies and to about 4% for

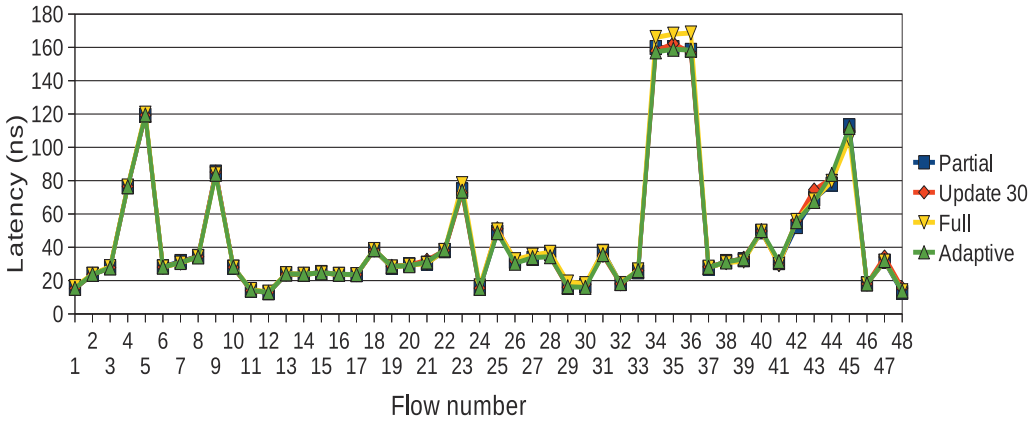

Figure 7.12: $D26\_media$ flow-by-flow error

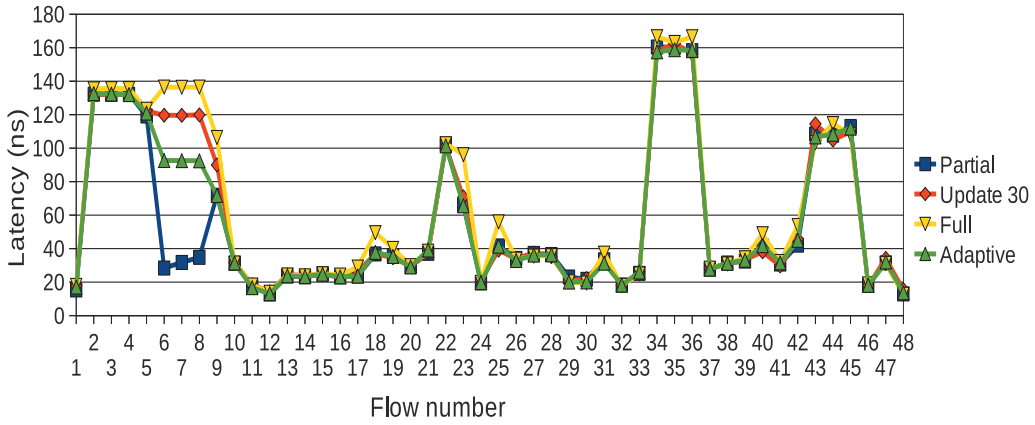

Figure 7.13: $D26\_media$ error at the end

the large topology. In case of the small topologies the adaptive method triggers after few flows are mapped only achieving better speedup than synchronizing every 30 flows, however the error at the end is larger. Please note that the goal of the adaptive method is to keep the error to 10% or less throughout the entire run.

The average flow-by-flow error and the average estimated latency error for the $D36\_4$ benchmark are plotted in Figures 7.10 and 7.11. For this benchmark as there are more communication flows that overlap the flow-by-flow error is larger and the full simulation synchronization has more effect in reducing the error for $D36\_m$ and $D36\_l$. Also the average estimated latency error is larger for topologies with more switches. As there are more flows the path diversity provided by the increased number of switches is not enough and there are still many overlapping flows leading to more error in estimating the latency. However updating the *latency tables* from full simulation still leads to a significant reduction of the error.

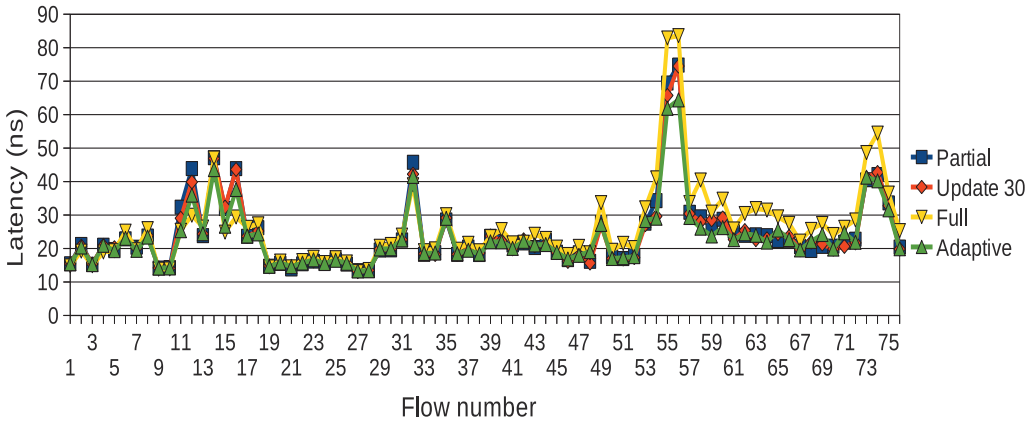

Figure 7.14: $D36\_4$ flow-by-flow error

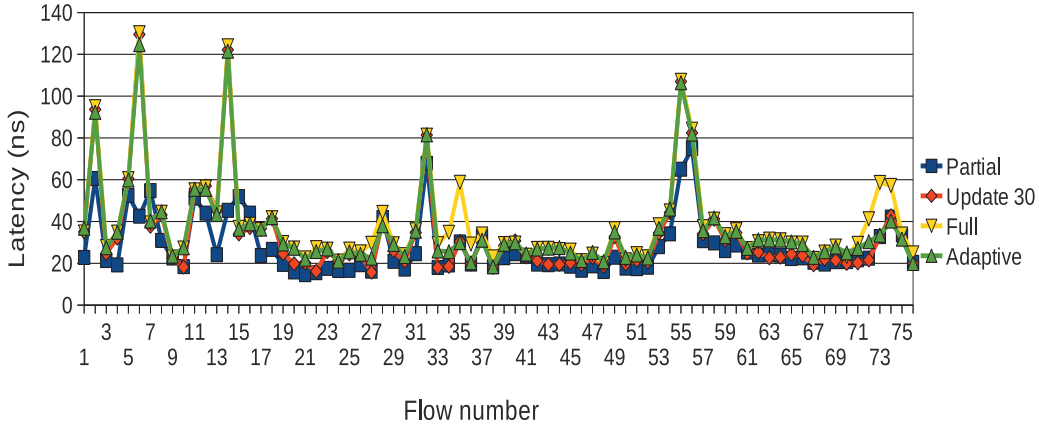

Figure 7.15: $D36\_4$ error at the end

In Figures 7.12 and 7.13, I show a detailed plot of the flow-by-flow latencies and the latency estimation for the 5 switch topology of the $D26\_media$ benchmark. It can be seen that the error of partial simulation in estimating the latency of the simulated flow is quite small in this case. However the error of the estimated latency is very large for some of the early mapped flows. That is because several of the flows mapped after that interfere with those flows. Running the full simulation and updating the partial simulation tables after 30 flows fixes these issues with the estimation of the latency of the early mapped flows. The adaptive method has more error for some of the flows that were mapped first as the last synchronization is triggered after the 10^{th} flow was mapped.

I also show a detailed plot for the flow-by-flow latencies and the estimated latency for the $D36\_s$ in Figures 7.14 and 7.15 respectively. For this benchmark as there are more traffic flows we see that the flow-by-flow latency estimation error grows as more flows

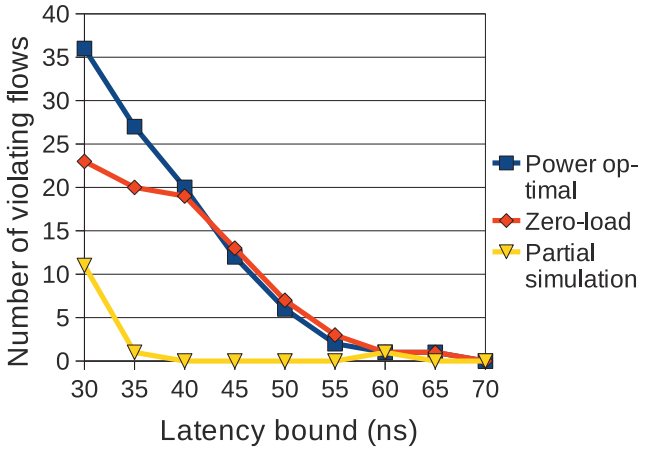

Figure 7.16: *D*26_*media* synthesis evaluation

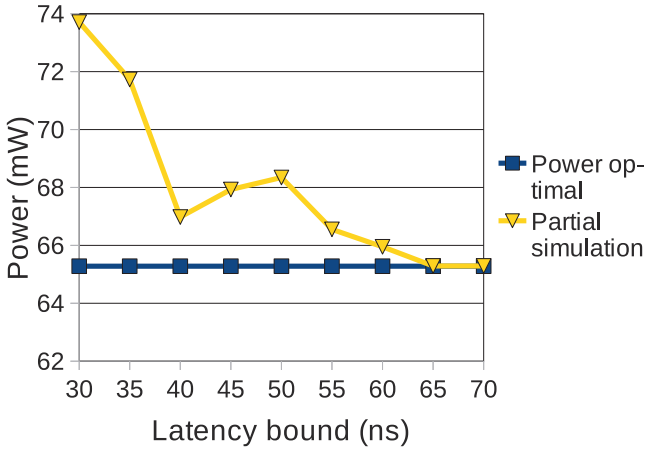

Figure 7.17: *D*26_*media* power evaluation

are mapped. That is because when the last flows are mapped there is a lot of interfering flows which are simulated by $PTgs$ and $PDests$. In case of the previously mapped flows latency estimation the error is larger for the flows mapped in the beginning, but that error is also reduced when the full simulation is run and the partial simulation tables are updated.

7.4.3 Synthesis

As one potential application of partial simulation is to be used to provide soft QoS support during topology synthesis, in this section I evaluate the proposed topology synthesis algorithm. I compare my extended synthesis algorithm, which uses partial simulation to drive the topology generation (described in Section 7.3) to the original algorithm from [121] that uses a *zero load* analytical model to estimate the latency

and drive the synthesis.

To evaluate the proposed method, I synthesize topologies for the $D26\_media$ benchmarks, with two switches per voltage island, with different latency constraints for all flows. In Figure 7.16, I show how many flows fail to meet the constraint, when I vary the latency bound from a very tight value where all algorithms fail up to a loose value where all flows meet the desired constraint. As a reference, I have synthesized a topology with no latency constraint and I counted the number of flows that would not meet the bound. As can be seen the method using partial simulation can provide good solutions for much tighter latency bound than the other. For a looser latency bound (60 ns), we see that the algorithm has a flow that fails, that is due to error in the latency estimation and because the algorithm cannot go back and remap a flow if later it realizes that a flow fails. The *zero load* driven method reduces the number of flows that fail for tight latencies when compared to the reference, but it is unable to provide good solutions. Also for looser bounds the *zero load* driven algorithm has worse results than the reference because by reducing hop count alone it increases the contention which results in more flows violating the constraint.

In Figure 7.17, I present the power overhead of the topologies designed using synthesis with partial simulation. In order to meet tight latency bounds the flow using partial simulation is forced to open more new links in order to meet the constraints. The increase of the number of links and in the switch sizes leads to some power overhead as can be seen from the plot.

7.5 Summary

System level simulations are an integral and time intensive part of the *Network on Chip (NoC)* design process. Reducing simulation times, while maintaining accuracy of the measured performance metrics can significantly speed-up the entire NoC design cycle. With increasing use of platform based designs, the NoC architecture only changes marginally across multiple simulation runs. In this chapter, I presented *a partial simulation*, a method to dramatically speed-up simulations when the traffic characteristics of only some of the flows change between subsequent simulation runs, while maintaining accurate performance results. I showed the applicability of the method to different scenarios, such as during NoC topology synthesis and design space exploration. The proposed method provides large reduction (between 4.5× to 10×) in run time when compared to full simulations. In future, I plan to extend the method to support DRAM based traffic patterns and parallel execution of multiple simulation runs.

8 Efficient memory access

Todays applications that drive the *System-on-Chip* (SoC) design require increasingly higher storage capacity and bandwidth to main memory. To provide the required storage capacity, high-density commodity DRAM is the only cost-effective main memory option, and external DRAM modules are needed. To deliver the required bandwidth when accessing external DRAM is a major challenge and main memory access is the main bottleneck for scaling computing performance, also known in literature as the *Memory Wall* [176]. There are two important aspects that we need to deal with (from the perspective of the interconnect) in the quest to provide more performance from the memory system:

- *Preventing regular traffic to be blocked by DRAM traffic:* The unpredictable nature of DRAM access can lead to transitory bottlenecks at the DRAM controller ports and the packets that backlog at the DRAM ports would queue up in the NoC and interfere with non-DRAM traffic. It is highly undesirable for non DRAM traffic to be blocked by DRAM traffic as it can take tens to hundreds of cycles to clear out the DRAM backlog.

- *Accessing multiple DRAM channels:* In mobile SoCs platforms that have strict power consumption constraints, providing the required bandwidth to main memory at manageable power levels is an even more challenging problem. Mobile SoCs rely on parallelization and specialization of the functionality in order to obtain the required performance for the application at low power level. A natural solution for providing increased memory bandwidth with low power would be to parallelize the access to the memory.

End-to-end flow control can be used to prevent DRAM traffic from blocking non-DRAM traffic. However end-to-end flow control requires the use of a specialized mechanism to notify the cores of the state of the DRAM. For example a separate

network or high priority virtual channel [167] can be used. This not only increases the design complexity, but also the buffering requirements needed to drain the messages that are already in the network. Another solution is to separate the traffic going to DRAM (DRAM traffic) from the traffic going between the other cores (non-DRAM traffic) using virtual channels or physical channels. However for most SoC the traffic patterns are known at design time and the interconnect topology can be optimized for that specific application. In such cases physical channels can be used when the NoC is designed to split the traffic. Since the bandwidth requirements to the DRAM controller are large, the ports of the DRAM can have data-paths wider than the rest of the cores. In this chapter, I show how designing a separate network to DRAM is better, as it can account for the funnel-shaped pattern communication pattern and reduce the number of data size converters.

As the complexity of the applications increases, so does their need for storage and bandwidth demands to and from the storage devices. For example most smart-phones today have capabilities to decode and playback videos. A competitive product today must be able to playback high-quality, high-definition videos. If we look at the bandwidth demands on the decoded stream alone we can see a high increase in the bandwidth requirements over the standard-definition format from few yeas back. For example the bandwidth that needs to be transfered from memory to the display driver for a standard definition video 576i is around 40MB/s. If we have a high-definition 720p stream the transfered bandwidth is around 90MB/s and for a 1080p stream is around 240MB/s. In case of upcoming 3D videos is high-definition 1080p, the required frame-rate is double (around 60 frames/second) and therefor the bandwidth requirement is around 480MB/s. And these requirements are for the decodes stream alone, the actual bandwidth required by the video decoder IP-core could be close to 3GB/s. If we consider the whole application with the video being downloaded from the Internet and played back on both the display of the device and on an external display, the bandwidth demand to and from memory can go up to 7-8GB/s. This amount of bandwidth cannot be provided by a single DRAM channel with the power constraints imposed on mobile platforms. Therefore architectures with multiple parallel DRAM channels are required.

With the main storage in the form of external DRAM memory, the limited number of available off-chip IO ports prevents the design of a parallel memory system. However with the introduction of 3D integration technology, stacked DRAM and TSV based connectivity can be used to design a parallel memory system to provide more bandwidth at acceptable power levels. Given the promise of three-dimensional DRAM stacking, industry is actively pushing standardization of TSV-based interfaces. WideIO 3D-stacked DRAM [87] is an emerging JEDEC standard. WideIO memories use TSVs to provide interfaces to 4 channels each with a 128bit wide data interface. By running

at 200MHz with single data rate the WideIO memory can provide a peak bandwidth of 12.8GB/s at lower power levels than current *Low-Power Double Data Rate* (LPDDR) DRAM. According to the JEDEC standard WideIO will provide 50% more bandwidth with 20% less power than an existing dual-channel LPDDR2 off-chip solution. In this chapter, I focus on WideIO DRAM memory integration in the SoC design and I address the problem of efficiently using the 4 channels to achieve the required application performance.

There are two existing architectures for the memory sub-system that can be adapted to access the 4 channels of the WideIO DRAM: i) a simple architecture that uses 4 DRAM controllers to independently access the four channels that rely on software to balance the traffic between the channels and ii) a complex centralized controller with multiple ports and interleaved address space to balance the traffic among the four channels. Since the TSV size is large, when compared to transistor size, the wide data interfaces that require many TSVs will be far apart on the floorplan. An example of such a heterogeneous SoC floorplan with WideIO connectivity is presented in Figure 8.1. The position of the TSV interfaces in the example was chosen to accommodate the WideIO interface from [87]. This makes the centralized controller difficult and expensive to implement. Therefore neither of these architecture are well suited to be used with WideIO DRAM.

In this context the contributions of this chapter can be summarized as follows:

- *For preventing regular traffic to be blocked by DRAM traffic* the contributions are three fold:
  - It is unclear what is the power and performance trade-off for splitting DRAM and non DRAM traffic using dedicated physical channels in the NoC. Therefore I first analyze this trade-off and also compare it to a solution using end-to-end flow control. I present results for a realistic multimedia and wireless communication SoC benchmark on application specific topologies. I use a cycle accurate NoC simulator and DRAM controller simulator [168] to compare the performance of the shared and split NoCs for the generated topologies.
  - Second I show how the network to DRAM can be optimized due to the funnel-shaped communication pattern when the cores and the DRAM controller have different data bus sizes.
  - Third I present an architecture for data size converter, capable of transferring data full bandwidth on the wider side.

  The experiments show several interesting results: i) As expected, *split* network and *end-to-end flow control* perform significantly better for non-DRAM traffic

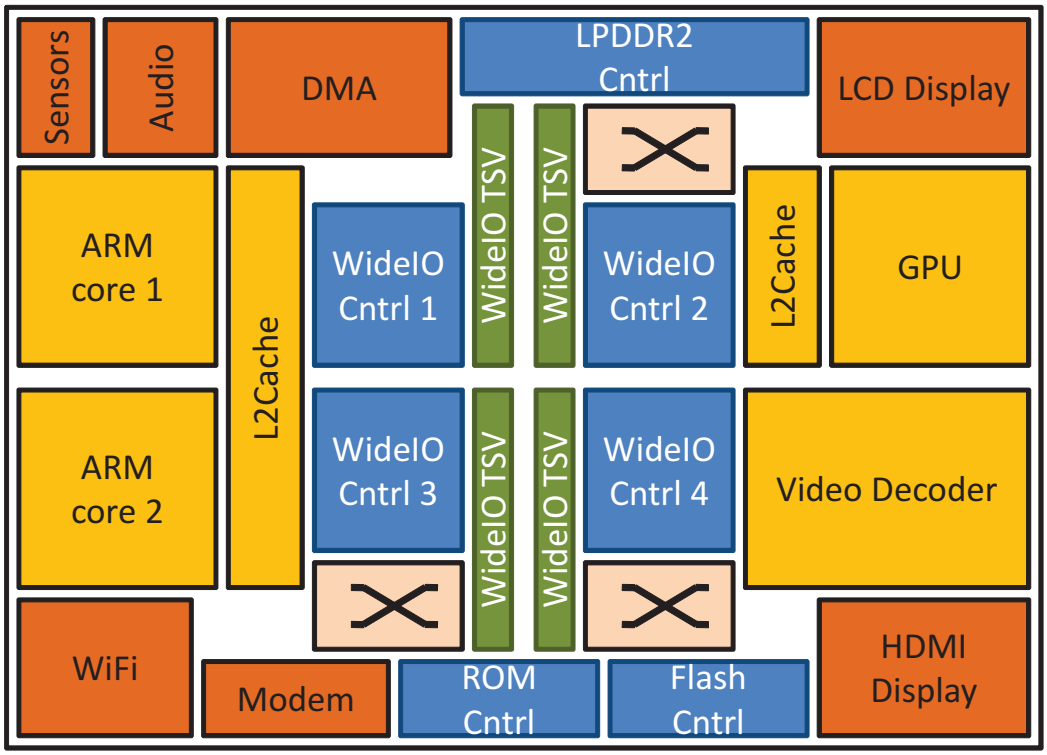

Figure 8.1: Example of SoC floorplan with WideIO TSV interfaces

(2.2× lower write latency and 1.7× lower read latency); ii) *Split* network and *end-to-end flow control* have similar power overhead; iii) All solutions perform similarly for DRAM traffic; iv) Designing a separate network to DRAM can be optimized in the presence of heterogeneous cores leading to power saving of 23%.

- *For accessing multiple DRAM channels:* I propose a new distributed interleaving method to access the memory controller that leverages the advantages of both of the previously mentioned designs, while avoiding their most serious shortcomings. On one hand it uses a simple memory interface design, as the first method, but provides an interleaved address space such that balancing the traffic to the four channels is transparent to the software. The proposed approach requires the integrated design of both the NoC and the DRAM controller, as the interleaving support is now distributed within the NoC, which also results in balancing the traffic in the NoC itself.

  I perform experiments on a realistic benchmark for a heterogeneous mobile communication and multimedia SoC platform. An important contribution is given by the advanced traffic modeling environment and by modeling the details of the NoC micro-architecture (i.e., packetization, bandwidth inflation).

From experiments, I show that the proposed distributed interleaving memory access method improves the overall throughput while minimally impacting the performance of latency sensitive communication flows. For example, this design improves the frame rate of the video decoding and display subsystem by 2 frames/second when compared to simple separated controller with the best memory allocation (42 frames/second with the worst memory allocation) and by 7 frames/second when compared to the complex controller with interleaved address space.

In the remainder of the chapter in the first part in Section 8.1, I present methods to prevent regular traffic to be blocked by DRAM traffic as well as architectures to optimize the NoC toward the DRAM. In the second part of the chapter in Section 8.2, I address the problem of efficiently accessing multiple DRAM channels in the context of WideIO 3D-stacked DRAM memory.

8.1 Preventing regular traffic to be blocked by DRAM traffic

8.1.1 System architecture

I use NoC switches and *Network Interfaces (NIs)* similar to those from [160]. I use input buffering, on/off flow control and source routing for the switches with no virtual channels. I consider a high performance DRAM controller capable of scheduling and reordering transactions. I synthesized power efficient application specific topologies using the tool from [121]. All the topologies have 4 switches for the request network and 4 switches on the response network. I chose these topologies as they were the most power efficient generated by the tool. The power models for the NoC components are based on values provided by commercial tools after place and route of the library components from [160] in 65nm technology. I analyze three designs. In the first design I synthesized a topology where the traffic that goes to DRAM and the non DRAM traffic can share channels. In the rest of the chapter I call this design as *shared* NoC. In the second case, I synthesized a topology where the DRAM traffic is physically separated from the non DRAM traffic, which I call *split* NoC. In the third design, I use the same topology as in the case of the shared NoC, but I apply *end-to-end flow control* on top, to prevent DRAM messages from queuing in the NoC when the DRAM is backlogged. I use on/off flow control and the occupancy of the DRAM controller buffers determines when core can inject packets to DRAM. This requires a specialized interconnect to give the off signal to the cores.

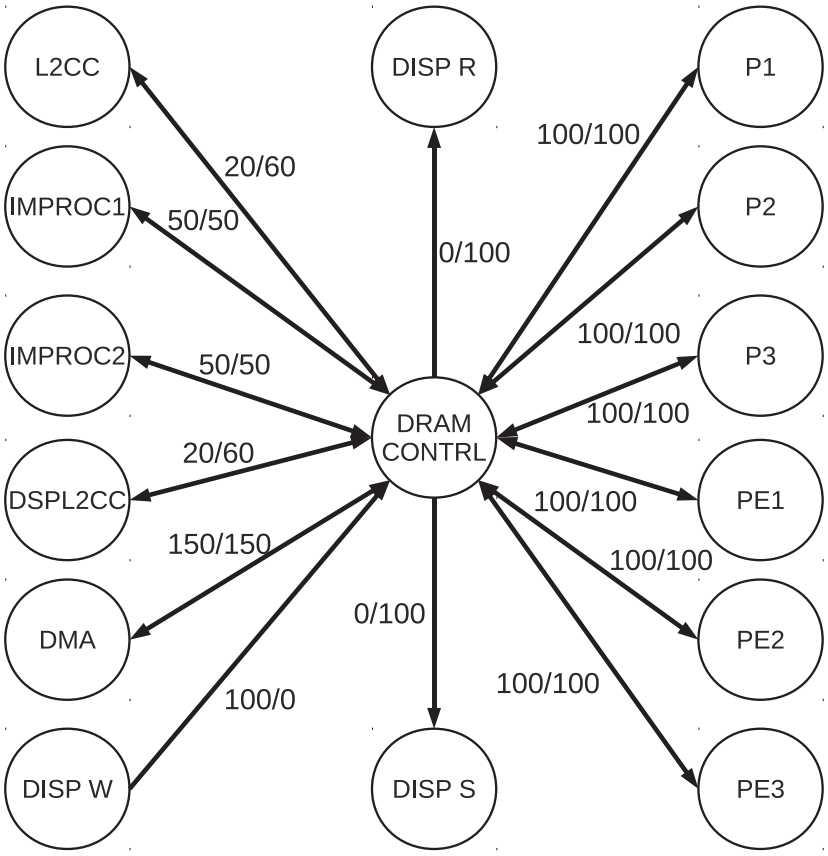

Figure 8.2: Communication specifications to DRAM

8.1.1.1 Benchmark

I use a realistic benchmark for a multimedia and wireless communication System-on-Chip. The benchmark has 28 cores, among which a processor with a separate L2 cache controller, a DSP which also has a discrete L2 cache core, a DMA, two image accelerators, a display controller, 6 clusters of peripherals, five scratch pad memories, a flash memory controller and a DRAM controller to connect to an external DRAM memory. The communication graph to and from DRAM is shown in Figure 8.2. The vertices represent the cores and the edges the communication flows between the cores. The weights on the edges represent the required communication bandwidth for write/read in MB/s. The total desired bandwidth to the DRAM controller is of 2160 MB/s. I only show the communication specifications to the DRAM controller. The non-DRAM traffic is considered to be similar to that of the D26_Media from Figure 3.10 in Chapter 3.

8.1.1.2 Simulation infrastructure

To run experiments, I use a cycle accurate NoC simulator capable to simulate custom topologies. In the NoC simulator I plugged-in DRAMsim2 [168], a cycle accurate DRAM simulator. I configured DRAMsim2 with open row scheduling policy and bank round robin. Transactions are interleaved in the banks based on the least significant bits of the address. I connect the network simulator to the DRAM simulator using a custom interface to provide support for multiple ports and to assemble the NoC messages into DRAM transactions of fixed size, as showed in Figure 8.3. I assume the DRAM to be a DDR2 device running at 333 MHz, and a 64 bit data bus from the DRAM controller to the external memory. I use the timing parameters of the Micron DDR2 of 32MB with 4 banks and 4 bit interface, 4 word burst and -3E speed grade, that is provided with the DRAMSim2 package. I design the NoC synchronous running at the DRAM controller frequency of 333 MHz.

The traffic generators have infinite injection queues. This is to capture in the measured latency the effect of the core being delayed, when they are unable to inject due to congestion the network. Traffic generators are connected to the initiator *Network Interfaces (NIs)* and inject traffic according to the bandwidth specified in the communication specification. The injected packets have a size of 32 byte of actual data for the write requests and read responses. The actual bandwidth that is injected in the network accounts for the packetization overhead that the NoC architecture creates. In Figure 8.4, I present how the NIs are modeled. In case of the *shared* NoC the DRAM and non DRAM packets can mix (Figure 8.4 a). In the *split* NoC, two NIs are used to separate the packets (Figure 8.4 b). For *end-to-end flow control* separate queues in the same NI are used (Figure 8.4 c). According to the occupancy of the DRAM buffer, the queue holding the DRAM packets can be prevented from injecting. I use a separate signal to notify the NIs when they can inject based on the DRAM buffer occupancy. The target traffic generators respond to the read requests immediately for the flows that do not go to the DRAM controller. For the DRAM traffic the read and write requests are buffer in the DRAM controller (however the DRAM controller buffer is finite), and the response is sent back through the network once the DRAM controller services the transaction.

8.1.2 Separated NoC vs. shared NoC

8.1.2.1 Power analysis

The power consumption of the NoCs is estimated by the synthesis tool that uses the power models of the components from the library in [160]. I show the breakdown of the power consumption in Figure 8.5. The *split* network has higher power as expected,

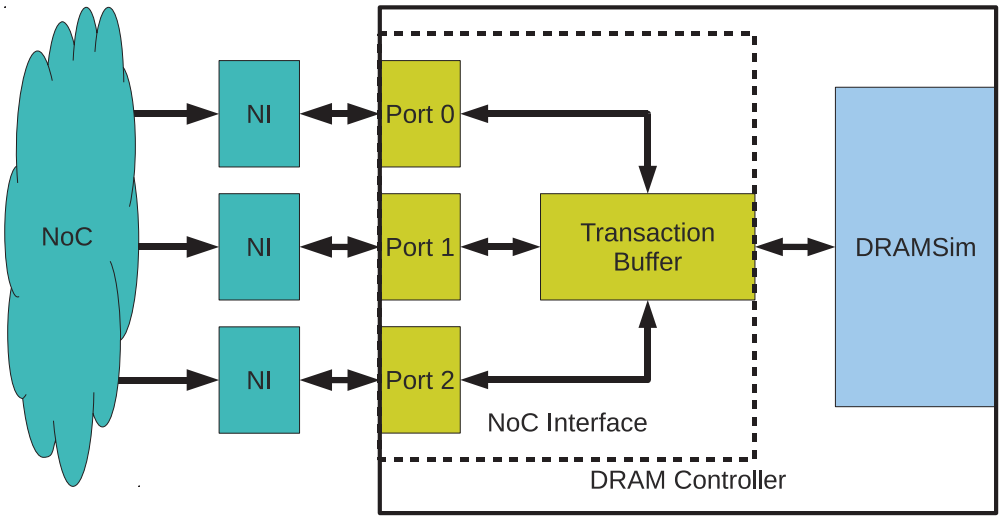

Figure 8.3: Example of DRAM controller connection

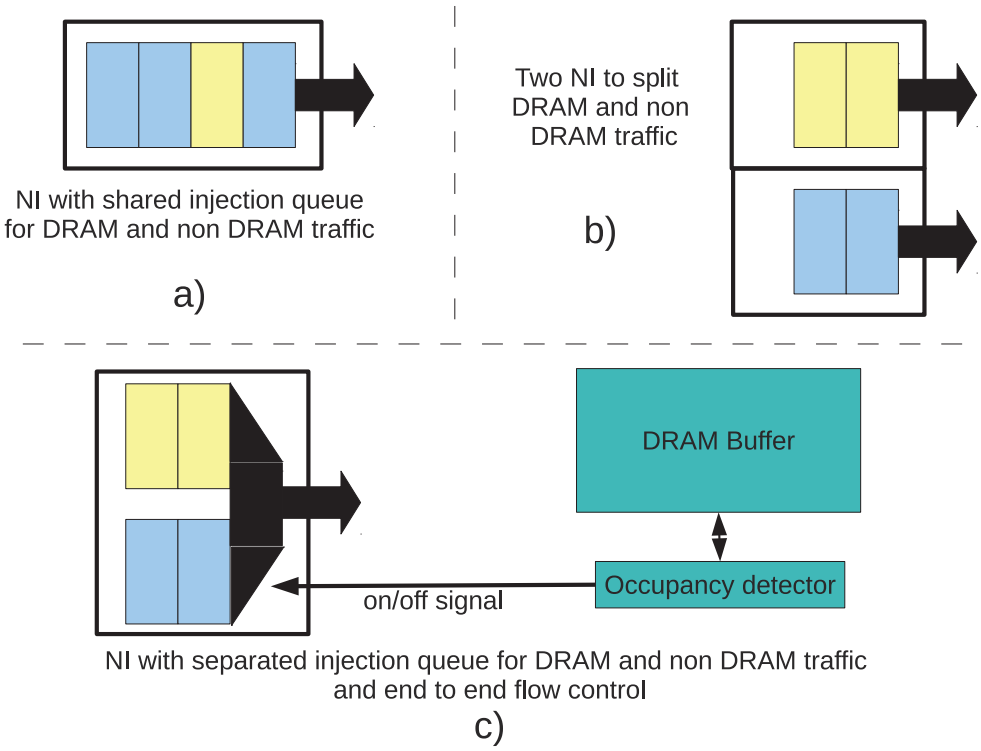

Figure 8.4: NI injection queues: a) *shared* b) *split* and c) *end-to-end flow control*

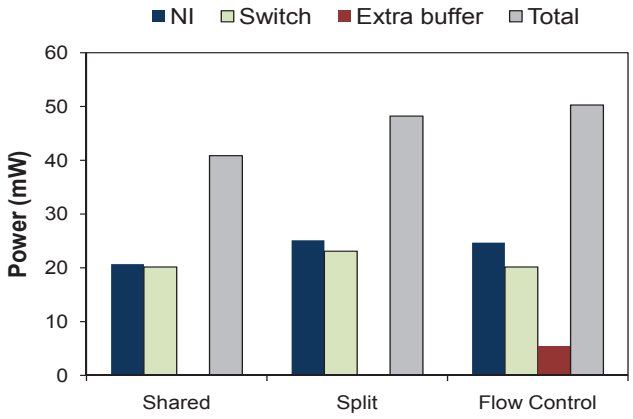

Figure 8.5: Power consumption for shared and separated NoCs

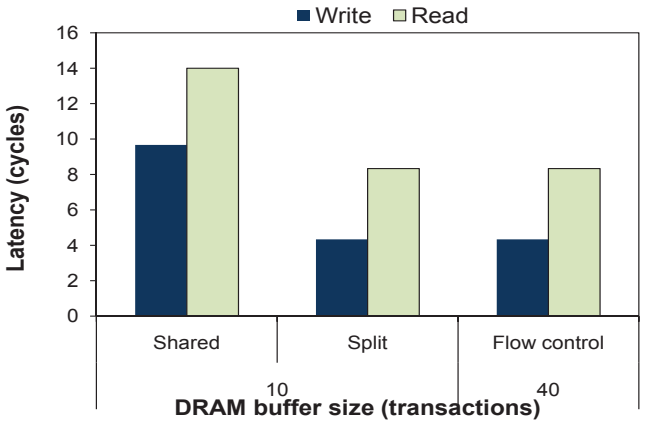

Figure 8.6: Average latency for non DRAM flows

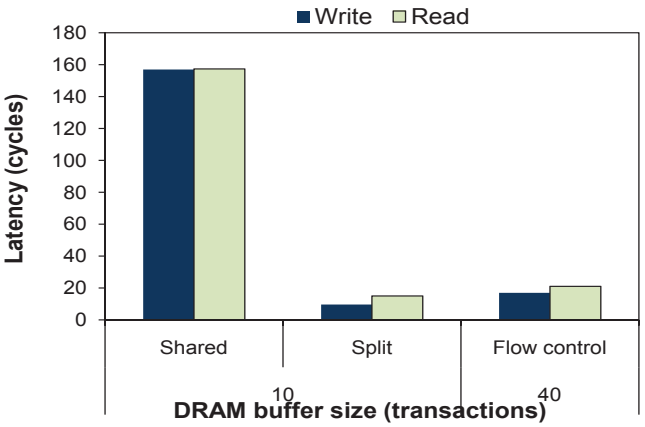

Figure 8.7: Maximum latency for non DRAM flows

as more NI are needed for cores that have both DRAM and non-DRAM traffic. Also switches are larger to connect the extra NIs. Interestingly the overhead of the switch to switch connectivity is not significant. For this benchmark the power overhead is 18%.

The *end-to-end flow control* design also has similar power overhead as *split*. The NIs of cores that have both DRAM and non DRAM traffic need separated packet queues. When the occupancy of the DRAM controller buffer reaches a threshold, the off signal is send to the injector NIs. As it takes several cycles to reach all the injectors, there can be several in flight packets going to DRAM. Therefore the buffer on the DRAM side has to be large enough to buffer these in flight DRAM packets. I use simulation data to determine the required size of the buffer. As buffering requirements in the DRAM controller are similar to those found in the network to DRAM in the *split* NoC, the power overhead is also similar. The area of the *split* NoC is 30% larger as compared to the area of the *shared* topology. Most of the increase in area comes from the extra NIs that have to be added.

8.1.2.2 Performance analysis for non-DRAM flows

In Figure 8.6, I show the average latency that the non DRAM flows have for the three designs: *shared*, *split* and *end-to-end flow control*. I present both the write and the read latency. The benchmark uses posted writes so no acknowledgment is sent back. The write latency is measured from the time that the write request is generated until the head flit of the packet reaches the destination. In case of the reads the latency is measured from the time the read request is generate until the head flit of the response packet reaches the requester. From the plot, we can see that the *shared* design has 2.2× larger write latency and a 1.7× larger read latency compared to the *split* or *end-to-end flow control* case. The increase in latency is due to the contention of the non DRAM flows with the DRAM flows when the DRAM backlogged. The traffic to the DRAM is from many initiators to one destination on the request network, and from one source to many destinations on the response network. Therefore the DRAM controller which becomes the bottleneck only creates congestion in the request network. This is the reason why the difference between the write latency is larger than that between the read latencies of the *shared* and *split* designs. The latency for the *split* NoC and *end-to-end flow control* are the same.

In the case of the maximum latency observed for the non DRAM flows, we can see a very big difference between the *shared* and *split* network as shown in Figure 8.7. For applications where the non DRAM flows have real time constraint ensuring that the worst case latency is small is very important for the performance of the system. From the plot, we can see that in the case of the shared NoC the DRAM traffic can greatly affect the latency of some packets. In between the *split* topology and the *end-to-end*

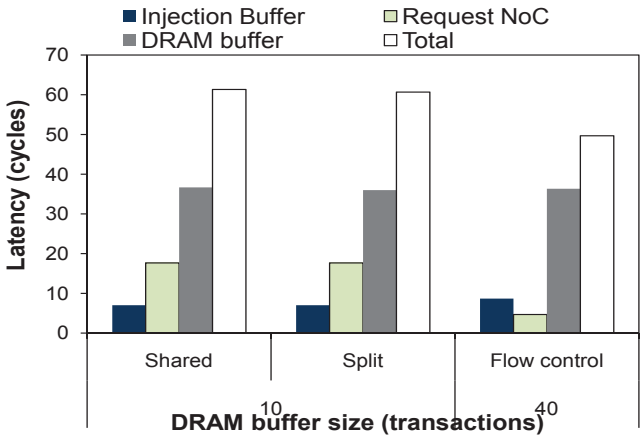

Figure 8.8: Average latency split for write DRAM flows

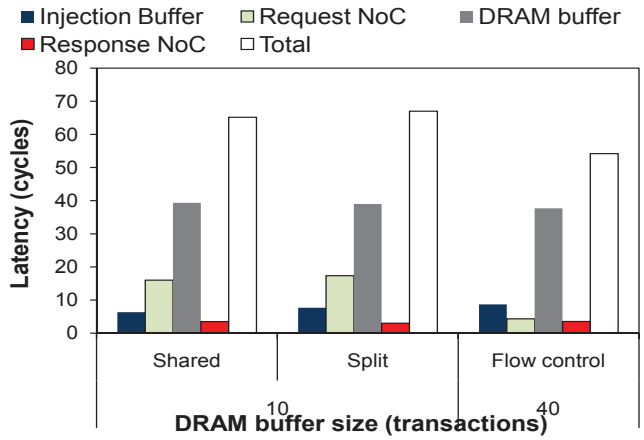

Figure 8.9: Average latency split for read DRAM flows

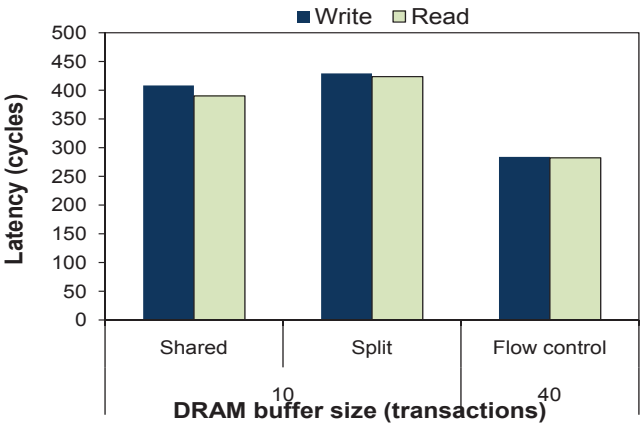

Figure 8.10: Maximum latency for DRAM flows

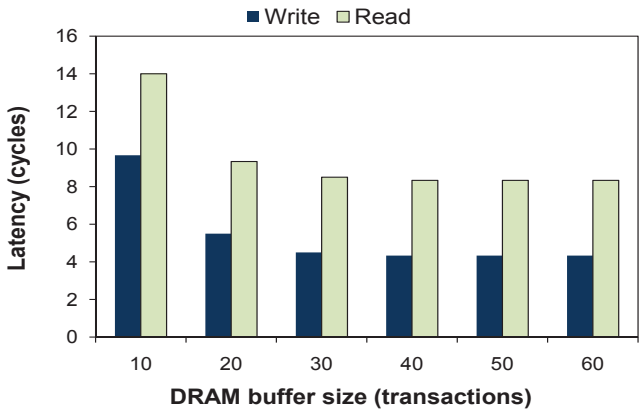

Figure 8.11: Average latency for non DRAM flows in shared NoC

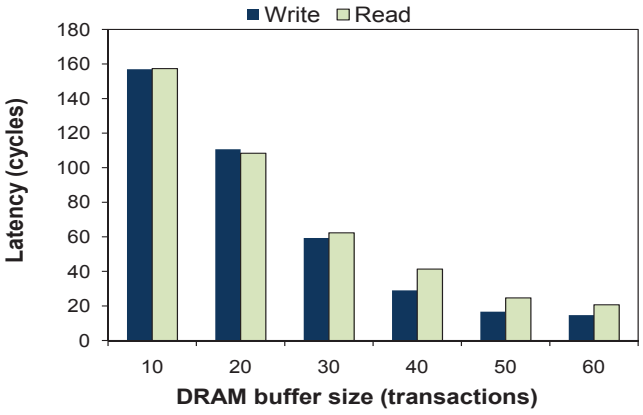

Figure 8.12: Maximum latency for non DRAM flows in shared NoC

flow control the results are quite similar, *split* being marginally better.

8.1.2.3 Performance analysis for DRAM flows

For the DRAM flows, surprisingly the three solutions perform similarly as the DRAM controller is the bottleneck. In Figure 8.8, I show a breakup of the average write latency of the DRAM flows. I show how much time the packets spend in the injection buffer of the NI, how long it takes to traverse the request network, how much time the transactions spend in the DRAM buffer (to be serviced by the DRAM controller before they can be sent back) and the total. Similarly in Figure 8.9, I present the breakup of the average read latency, where in addition the write I show the time it takes to also traverse the response network. From the experiments, we can notice that the DRAM

Number of ports	Read latency (cycles)	
	Average	Maximum
1	83	579
2	67.6	448
3	67.3	423
4	61	340

Table 8.1: Average and maximum read latency from DRAM for different number of ports

controller is the bottleneck, so most time is spent to reach the DRAM controller and for the transaction to be serviced. The latency on the response network is small, as the response network is from one to many and there is no congestion. The *end-to-end flow control* solution performs slightly better that the *split* and *shared*, because it requires a larger DRAM buffer so in turn the DRAM controller itself performs better when services the transactions. In Figure 8.10, I also show the maximum latency of the DRAM flows, for both write and read.

8.1.2.4 Observations

From the experiments presented before, we can conclude the following: i) the *shared* network preforms poorly for non DRAM flows compared to *split* and *end-to-end flow control*; ii) all designs have comparable performance for the DRAM flows; iii) the *split* and *end-to-end flow control* designs have similar power overhead. The experiments show that a *split* network can be an efficient solution as it can be implemented using existing NoC components, as opposed to *end-to-end flow control* that would require specialized hardware.

8.1.2.5 DRAM buffer size impact on non DRAM flows

I ran experiments with different buffer sizes on the DRAM controller. From the experiments I make the following observations. First having a large buffer does not make a significant difference on the latency of the DRAM flows. This is because packets are waiting in the DRAM buffer instead of the injection queue or the request NoC. Second, for the *shared* NoC we can notice that with large buffering it can achieve similar performance as the *split* network. In Figure 8.11, I show the average latency of the non-DRAM flows for different sizes of the DRAM buffer and in Figure 8.12, the maximum latency. However with the extra buffering the *shared* design would consume similar or more power than the *split* NoC.

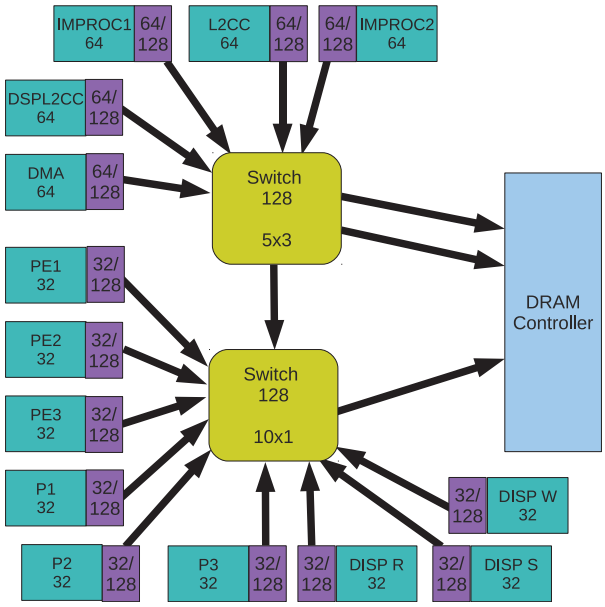

Figure 8.13: Original topology

8.1.2.6 Port count

For the previous experiments, I used three ports for the DRAM controller. However having different number of ports can influence the bandwidth that is transferred to the DRAM buffer. To see the impact, I ran experiments for different number of ports. In Table 8.1, I show how the latency is influenced by the number of ports for the *split* network. Because the DRAM memory operates at double data rate and the port data-size is the same as the bus-size between the controller and the DRAM memory, two ports are needed to support the peak bandwidth of the DRAM controller. Therefore there is a larger difference in the latency between one and two ports. When more ports are added the reduction in latency is less significant. Multiple ports are costly as the buffer is implemented as a multi-ported memory. Therefore an alternative is to increase the data size of the port. In the next section, I show how the network to DRAM can be optimized when the cores and the DRAM controller have different data sizes.

8.1.3 Network optimization for heterogeneous cores

In many SoCs, the cores are heterogeneous and they have different data sizes. Also the port of the DRAM controller could be wider than the port of the cores. If the network is designed with a single FLIT size, to support full bandwidth, the FLIT size has to be correlated with the size of the widest core port (usually the DRAM controller). In Figure 8.13, I show one topology with uniform flit size for this benchmark. The cores

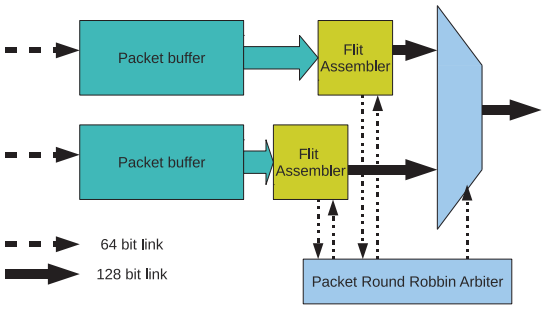

Figure 8.14: Data size converter architecture

and the switches are annotated with the data size. Some cores are 32 bit, some are 64 bit and the DRAM controller uses 128 bit wide data. In this case, the size conversion is done in the NI that connects the core to the switch. However, since the conversion is done from narrow to wide, to inject at full bandwidth the whole packet needs to be buffered in the NI before it can be sent out. In this case the size converters are at the endpoints. This requires a large number of converters and leads to wider switches. To reduce the number of converters they need to be moved after the switches. In the next section, I show a converter architecture that could be placed after the switches and be able to work at full bandwidth.

8.1.3.1 Proposed size converter architecture

In order to be able to inject a FLIT every cycle (without inserting empty FLITs) on wide link from a switch with narrow data width, I propose a new converter design. The new converter architecture is shown in Figure 8.14 for 64-bit to 128-bit conversion. Two links from the switch feed the size converter, and the packets from each link are buffered separately. This enables the converter to buffer a packet from one input link while sending data to the output link from the other buffer. The arbitration between the two input channel is done at packet level in order to prevent flits from different packets to be interleaved, in a round-robin fashion. By buffering packets from two input links, the converter can push data out on the wider output link at full bandwidth under high traffic load. Similarly the converter between a 32-bit switch and the DRAM can be designed. In this case the converter would need to buffer packets from four input links in order to be able to send data at full bandwidth to the DRAM controller. The communication flows going to the same port of the DRAM controller could be statically assigned to use one of the input links of the converter. An alternative would be to modify the allocator of the switch to send the packet of any of the flows onto one of the links going to the same converter, but where the buffer is free.

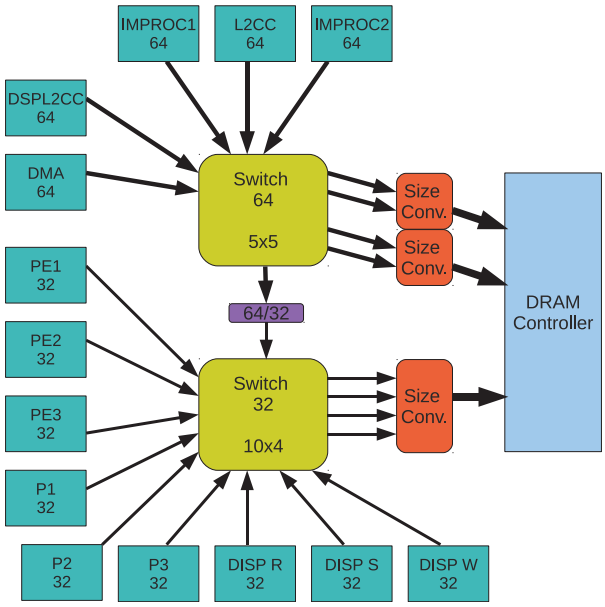

Figure 8.15: Optimized topology

8.1.3.2 Optimized NoC

To reduce the number of required packet buffers and size converters, I show in Figure 8.15 an example of how the topology could be designed using the new converter architecture. In the new case cores are clustered and connected to switches based on their data bus size. Therefore the switches can have a flit size closer to that of the cores. This removes the need for size conversion between the core and the switch. Also switches can have different sizes. Existing solutions do not consider the flow data conversion efficiency.

In the case of the new topology only 8 packet buffers are needed, two in each 64 bit to 128 bit converters and four in the 32 bit to 128 bit converter. The size converter between the 64 bit switch and the 32 bit switch is a wide to narrow converter, that in out benchmark is only used by one flow, so it does not need to buffer the packet. In this new design, I reduce the number of packet buffers by 42% which leads to an overall NoC power reduction of 23%.

8.2 Accessing multiple DRAM channels

8.2.1 NoC and controller architectures for multi-channel DRAM

WideIO DRAM memories [87] promise to fulfill the bandwidth requirements of future SoCs at power levels comparable to todays low power off-chip DRAMs. WideIO DRAM

leverages 3D integration technology to stack the DRAM die on top of the logic die. *Through Silicon Vias* (TSVs) are used to provide multiple channels (up to 4 in current specs) with wide data-paths. WideIO DRAM provides high-peak bandwidth at low power levels by using multiple channels with independent wide data-interfaces operated at low frequency. Efficiently exploiting the 4 channels of the WideIO memories, in order to satisfy the bandwidth demands of the application, is up to the SoC designer. Having multiple channels provides new opportunities for designing the memory controller to efficiently access the memory subsystem. In this section, I introduce three architectures: two traditional ones and I also propose a new architecture that moves part of the DRAM controller complexity into the NoC fabric to provide distributed parallel access to the four channels of a WideIO memory.

In this section, I describe three architectures and discuss their advantages and disadvantages:

- *Distributed controller with separate independent channels.* This is a simple solution where an independent DRAM controller is assigned to each channel.
- *Centralized controller with interleaved address space.* This is a single multi-ported controller that can access all the four channels of the WideIO memory interface. The requests from the four ports are interleaved based on the address to the four channels to balance the traffic.
- *Distributed controller with interleaving in the NI.* This is the proposed method that uses 4 simple DRAM controllers for the 4 channels, but the transactions are split and interleaved based on address within the interconnect at the *Network Interfaces* (NIs) of the NoC.

8.2.1.1 Distributed controller

The simplest solution to access the four channels is to have four independent DRAM controllers and to split the address space in four large contiguous blocks. A block diagram of such an architecture is presented in Figure 8.16. By ensuring at software level that a core does not access more than one channel at the same time, this architecture does not require any changes to existing DRAM controller or NoC architectures. To each channel a DRAM controller is assigned that only needs a bus interface and address generator, with FIFO buffers to drain the requests from the NoC and an out-of-order back-end. Designing the memory subsystem of the SoC in such a distributed fashion, with separate independent DRAM channels, is easy. The main drawback of this simple architecture is that the performance of the memory subsystem is highly dependent on the memory mapping. In the worst case, if the memory regions of all cores are mapped to the same channel (which could happen for certain periods of time in

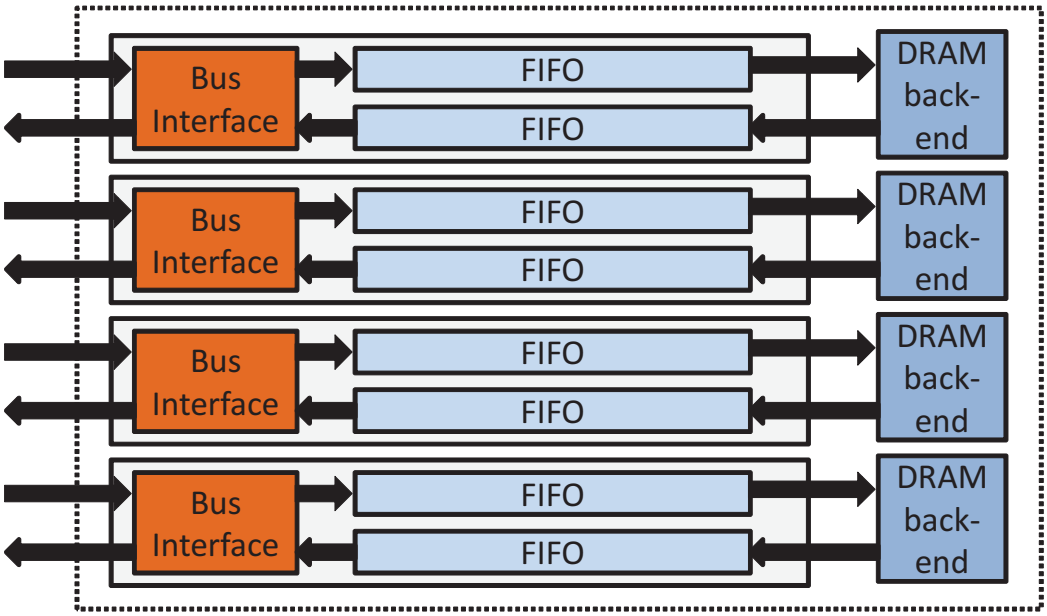

Figure 8.16: Distributed DRAM controller with 4 separate channels

systems where the memory is dynamically managed), then the peak-bandwidth of the WideIO DRAM memory is effectively reduced four times. To tackle this problem, such a system would have to expose hardware details to the software and make sure that at software level, the memory is allocated as to balance the accesses to the 4 channels. However as such an allocation, reliant on the software, would be coarse-grained, it may not be possible to efficiently balance the accesses to the channels.

To tackle this drawback of separated independent channels, a popular technique is to interleave the address space between the channels [169]. By splitting the address space in fine-grained blocks and assigning them to the channels interleaved as shown in Figure 8.18, we can make sure in hardware that the data is distributed uniformly among the channels. Accessing the memory is now balanced between the channels and it is transparent to the software. The decision to which channel a transaction (or a piece of a transaction) belongs depends on the address of that transaction. Depending on where that decision is taken, I look at two solutions: i) centralized in the DRAM controller and ii) distributed at the initiator *Network Interfaces* (NIs) of the NoC.

8.2.1.2 Centralized controller with interleaving

One way to design an interleaved memory subsystem is to have a centralized DRAM controller with multiple ports toward the NoC and multiple channels to connect to the WideIO DRAM as shown in Figure 8.17. This is similar to [9], but with different

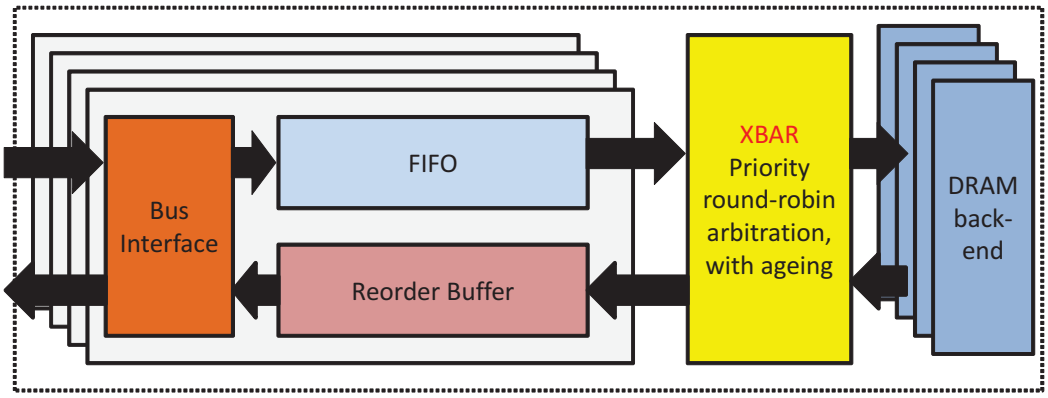

Figure 8.17: Centralized DRAM controller with interleaving at the controller side

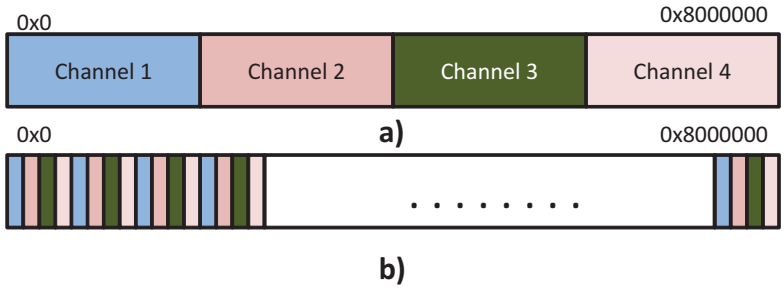

Figure 8.18: Example of a) partitioned- and b) interleaved address space

arbitration scheme though, as I do not require the same level of predictability. The advantage of this solution is that the memory controller and the NoC can be designed separately and regardless of which port you use you have access to the entire memory space. Such a centralized solution would require a crossbar to allow for parallel access between ports and the back-end controllers connected to the channels. In this architecture the bus interfaces, corresponding to each port, take the bus request and generate transactions for the DRAM back-end with the same size as the interleaving blocks. These transactions are stored in the input FIFO of the port. The arbiter takes these transactions from the ports and using the crossbar sends them to the appropriate channel according to the address. A popular schemes for arbitration (which I also assume) is priority with aging to prevent starvation. Ports can have different priorities (in this case I use two) and the transactions in the low priority ports keep an age counter. After a certain age given as parameter their priority becomes the highest and get serviced next. Please note that reordering of the transaction to be memory friendly is done in the back-end and that is beyond the scope of this work. However since transactions from the same port are sent to different back-ends they can be serviced and returned out of order, so the response buffers in the ports have to be able to reorder the transactions.

While this solution fixes the problem of application memory mapping, there are two important drawbacks of this solution. The first drawback relates to the physical implementation of the centralized controller. The WideIO uses TSVs for connectivity, which are considerably larger than other features, and has wide interfaces so requiring many such TSVs. According to the JEDEC standard the size of the 4 channels interfaces is around 0.54mm by 5.27mm. So the interfaces are far apart on the floorplan of the resulting chip as shown in Figure 8.1. Therefore designing the crossbar to connect the ports to the DRAM back-ends may be costly and slow. The second drawback is that the as there are more IP-cores that used the DRAM than ports, it is up to the designer to decide the assignment of the IP-cores to ports. As different IP-cores may be active for different application there could be cases where even a good assignment could still result in over-congesting a port and parts of the NoC. This would result in a degradation of performance.

8.2.1.3 Distributed controller with interleaving

A compromise between the two solutions discussed before is to distribute the interleaving function from the DRAM controller in the NoC. This solution would require the simpler controllers of the first solution, which could be easily implemented next to the interfaces to the WideIO memory. In this case the source NIs are responsible to implement the interleaving function (performed by the crossbar in the centralized controller). A schematic representation of this architecture is in Figure 8.19 and is called distributed controller with interleaving in the NIs. The NI needs to have a route to each port of the independent memory controllers. Based on the address the NI chooses the path to the corresponding port. In case of transactions that are larger than the size of the interleaving block, the NI is responsible to split these transactions into multiple packets and to send those packets to the corresponding controllers. In this case the responses or partial responses (as the transaction can receive the responses in multiple packets) have to be reordered and reassembled in the NI. The details describing the NI are presented in Section 8.2.2.1. To provide two level of priority as in the centralized controller case, I use two FIFO buffers per port as shown in Figure 8.20. The assignment of the transactions to the priority queue is done based on the ID of the source.

The disadvantage of this method is that it generates more bandwidth in the NoC, due to the packetization overhead in the case when transactions are split. Please note though that even in the previous case long transactions may need to be split to prevent blocking the interfering traffic for too long from accessing the memory controller as well. In this architecture the NoC and the memory controller need to be designed together. The main advantage of this architecture is that it balances the traffic in both the NoC and at the DRAM controller channels, regardless of the application memory

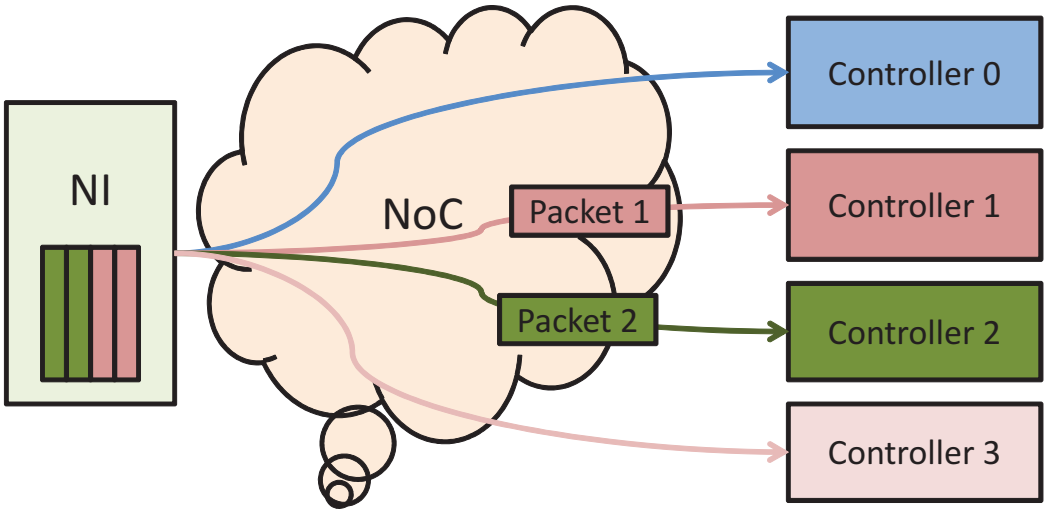

Figure 8.19: Example of distributed interleaving where the NI sends out a 4 beat burst transaction in 2 packets on different routes

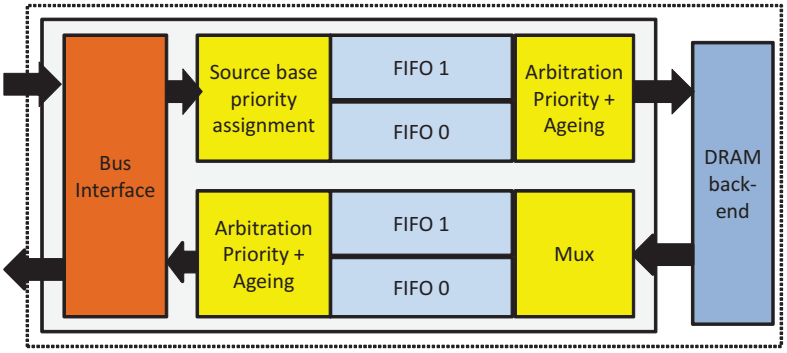

Figure 8.20: DRAM controller port with priority

mapping.

8.2.1.4 Simple reorder-buffer implementation

If the NI has support for interleaving and splitting of packets, then it needs to be able to reorder the transactions. However in cases where there are only few routes handled by the NI, the reorder buffer can be implemented in a simpler way. By leveraging the property that packets on the same route will arrive in order, then the reorder buffer can be implemented only with FIFOs and few comparators, instead of expensive CAM memories. In Figure 8.21, I show an example of such a reorder buffer for an NI that uses two routes to interleave the packets.

To track the order, the transactions are assigned an ID (consecutive values of a counter)

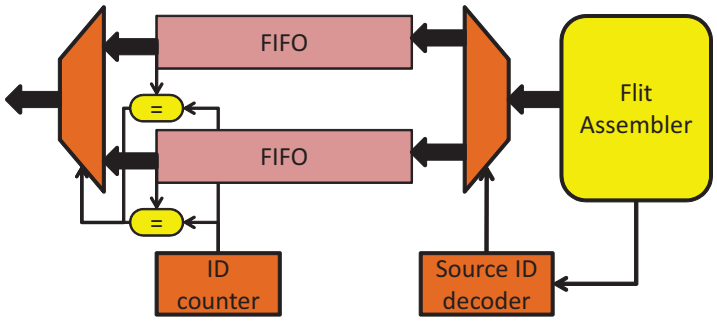

Figure 8.21: Simple reorder buffer schematic

when they are sent out to the target. The responses from the target will contain the same ID that was assigned to the request. We need as many FIFO buffers as there are routes. As the transaction responses coming from the same path arrive in-order, we can use simple FIFOs to buffer them. Based on the ID of the source (the ID of the target that responds) the transaction are stored to the corresponding FIFO. A counter at the receiver NI also indicates the ID of the transaction that has to be delivered next and it is incremented whenever a transaction is delivered from the FIFO to the bus interface. One comparator per FIFO buffer compares the ID of the transaction that is in the front of the buffer to the ID counter. If the ID of the transaction matches the ID of the counter it indicates that as the next transaction to be transfered.

8.2.2 Exploration environment

The experimental setup is based on a cycle-accurate NoC simulator. The simulator models accurately the ×pipes NoC library [160] which includes: accurate NI models that account for packetization effects, FLIT size converter models and switch models. Moreover, I extended the initiator NI models to add support for multiple outstanding transactions and for reordering responses and for transaction splitting and interleaving. I implemented three types of traffic generators: i) the initiator traffic generator, ii) the target traffic generator and iii) the DRAM target traffic generator. The initiator traffic generator supports multiple outstanding transactions and out-of-order execution and it is described in detail in the next sub-section. The target traffic generator receives the requests from the initiator traffic generators and generates the response traffic. The target traffic generator can also be configured as synchronization block. The DRAM target traffic generator uses DRAMSim 2 [168] to accurately emulate the functionality of the back-end of the DRAM controller. The DRAM traffic generator interface the NI to the DRAMSim instances and can be configured with multiple ports as well as channels in order to simulate all the architectures proposed in Section 8.2.1. The benchmarking is oriented to real-life traffic in a state-of-the-art mobile SoC. I model different types of traffic and addressing modes to emulate different IP-cores

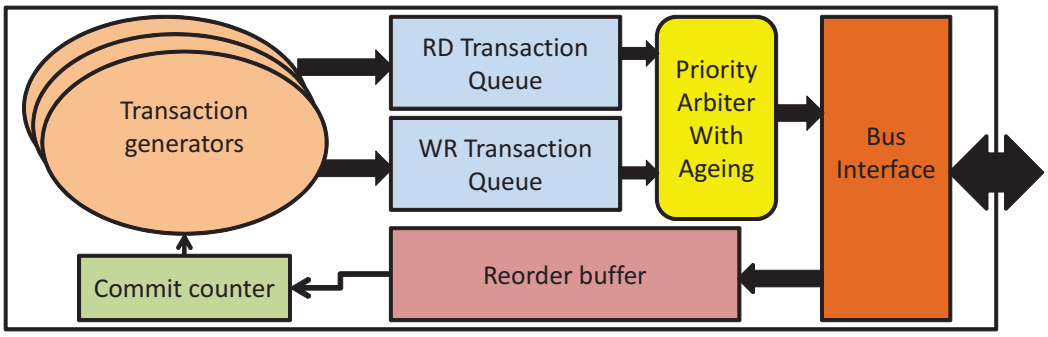

Figure 8.22: Initiator traffic generator

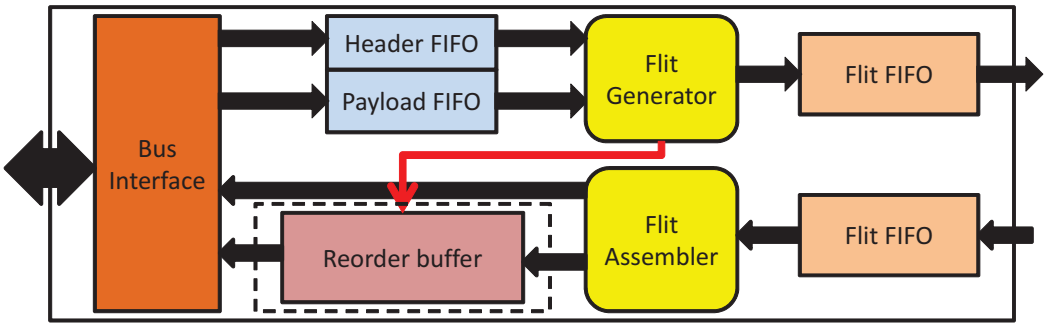

Figure 8.23: Initiator network interface

like: video and graphic accelerators, display drivers, DMAs, peripherals and not just CPUs. Hence my environment is a significant step forward with respect to homogeneous architecture modeling environments with traffic generated only by cache misses.

8.2.2.1 Traffic generators and NIs

The *initiator traffic generator* (ITGen) along with the initiator NI are some of the most important components of the simulator. The ITGen has to be able to emulate the traffic patterns generated by processors, accelerators, custom controllers, communication IP-cores. I described the ITGen in the simulator to have the structure as presented in Figure 8.22, to be able to emulate all these behaviors. The basic blocks of the ITGen are the transaction generators, the read and write queues, the arbiter between the read and the write queues, the bus interface and the reorder buffer.

The transaction generators are responsible to generate the high-level read/write transactions (e.g. cache line refill or write-backs in a cache controller). The transaction generator can be programmed to generate different patterns of transactions, with different sizes and different time intervals between them. The types of transactions I use are: i) reads, ii) non-posted writes and iii) barriers which are used for synchronization.

I only considered non-posted writes in this chapter as I assume that acknowledgments are needed to implement memory consistency protocols since data is shared among some cores. However posted writes can also have be used without impacting the results, especially since no acknowledgment traffic is necessary in that case. As the transaction generator blocks if it tries to inject a transaction in a full queue, the sequencing dependencies between transaction in the pattern are respected regardless of the type (read/write). Consequently the following transaction are delayed by the same amount of time that the transaction generator is blocked. A ITGen can have multiple transaction generators which are independent from one another. For example if a transaction generator blocks on a full read queue another transaction generator could still generate write transactions if the write queue is not full. The use of multiple transaction generator can emulate multiple threads in a multi-threaded CPU or multiple independent units in an accelerator. The arbiter selects between the read and the write queue the transaction that has the highest priority, and which will be sent next to the bus interface. The bus interface has to convert the current selected transaction to bus requests (i.e. similar to AXI bus requests), which could comprise of a burst of requests in case of writes. Once a transaction has been processed by the bus interface, it is placed in the reorder buffer to await the response from the target traffic generator. The ITGen can be configured to expect responses in-order or out-of-order. In the first case when a response it received the ITGen checks that the response corresponds to the first transaction in the reorder buffer. In the latter case the response is matched to the corresponding request transaction in the reorder buffer. The serviced transactions in the reorder buffer are committed in-order, and the latency is tracked at commit time.

One important feature of the transaction generators is that they support multiple addressing modes, which can be configured when they are instantiated. The supported addressing modes are: i) *incremental*, ii) *incremental with synchronization*, iii) *block addressing*, iv) *random block addressing* and v) *trace*. *Incremental* addressing as the name suggests generates addresses incrementally with the possibility of wrapping around after a certain number of transactions. This mode can be used to emulate IP-cores that work with arrays or that move large amounts of data (e.g. DMA). In case of the *incremental with synchronization* addressing mode at the end of the accessed address range when it wraps around or moves to another address range it requires to synchronize with other IP-cores. New transactions are not generated until the synchronization completes. This mode can be used for IP-cores like the LCD or HDMI which read a frame with incremental transaction, but require synchronization before moving to another frame. *Block addressing* is designed to emulate the addressing mode of the video decoder. The frame is viewed as an M by N matrix which is further divided in blocks of size m by n (m < M and n < N). The blocks are chosen in row order and for each block the addresses are generated in row order as well. This mode also

requires synchronization at the end of the frame. In case of *random block addressing*, the next block to be accessed is chosen in a random manner, to emulate the fact that video decoding is data dependent. For *trace* addressing, I read a trace of addresses from file. These traces can be generated beforehand with a functional simulator and can be used to have a realistic behavior of IP-core for which the behavior is harder to emulate.

The initiator NI is described in Figure 8.23. The initiator NI contains the following blocks: the bus interface, the request queues (as header information and payload data queues), the FLIT generator block, the FLIT assembler block, the FLIT FIFOs and an optional reorder buffer. The bus interface is required to connect the NI to the traffic generators. The FLIT generator and FLIT assembler blocks convert the bus transactions into NoC packets and vice versa. FLIT buffers are used toward the switch as well to improve performance. In the case when the ITGen supports multiple outstanding transaction, but it requires the responses in order, a reorder buffer is necessary. To prevent deadlocks, the FLIT generator block will reserve space in the reorder buffer when generating a packet. If there is not enough space in the reorder buffer, the FLIT generator will block until space becomes available. In case the transaction requires more space than the size of the buffer the FLIT generator will wait until the buffer is completely free before starting to generate the packet. If transaction splitting and interleaving is supported by the NI then the reorder buffer is also necessary as the different parts of a single transaction, sent as different packets, have to be reassembled in-order into the response transaction, which will be returned to the ITGen.

The target traffic generator is simpler. It only has the bus interface and a queue in which to store the incoming requests. The bus interface takes the stored requests and generates the appropriate response. The target traffic generator can be configured as a synchronization block as well. In that case it will only generate the responses when it received requests from all the ITGens that need to be synchronized. The DRAM controller simulation is done as described in Section 8.2.1. The target NI is similar to the initiator version, but it is simpler as it does not need support for reordering or splitting and interleaving.

8.2.2.2 Benchmark and use-cases

To perform experiments I use a realistic benchmark that describes a mobile multimedia platform capable of running applications on a CPU cluster with 3D acceleration, perform video decoding, support multiple high definition displays and wireless communication. The communication requirements are represented by the graph from Figure 8.24. The vertices in the graph represent the task/IP-cores. In some case sev-

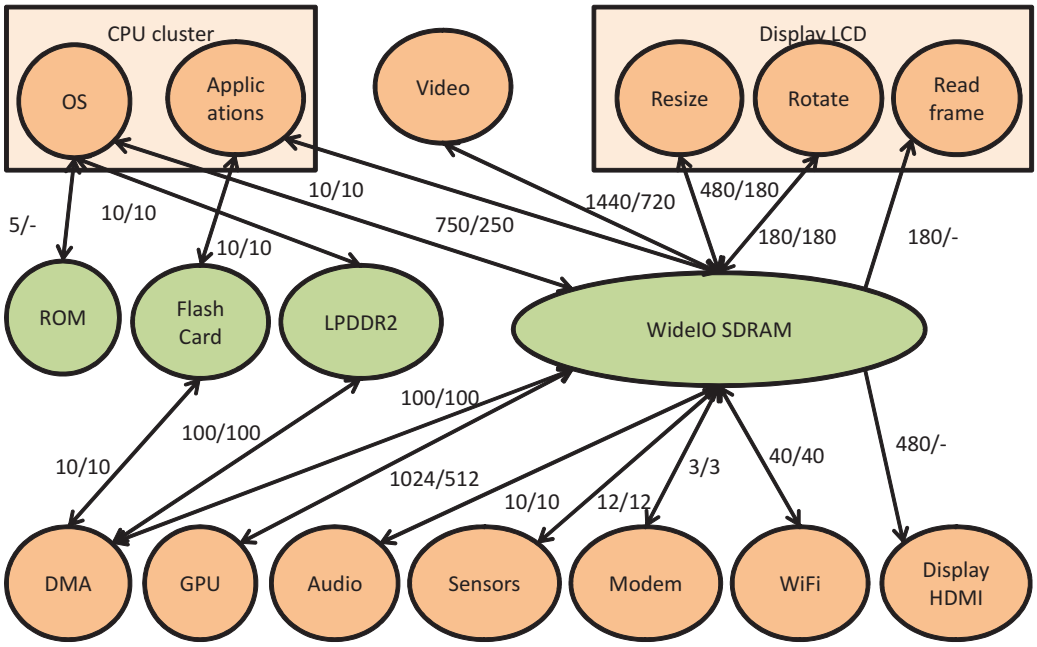

Figure 8.24: Benchmark communication graph

	CPU	GPU	Video	LCD	HDMI	DMA	Audio	Sensors	WiFi	Modem
UC1			x		x					
UC2	x	x	x		x					
UC3	x	x	x	x	x					
UC4	x	x	x	x	x	x	x	x	x	x

Table 8.2: Description of the use-cases

eral tasks are performed by the same IP-core (e.g. Display LCD and CPU cluster). The edges in the graph represent a communication flow between two IP-cores. The weights on the edges represent the read/write bandwidth demands in MB/s. As can be seen from the plot this benchmark is DRAM centric with most communication going to the WideIO SDRAM.

The communication graph describes all the possible communication flows in the SoC, however different operating modes may require only a subset of the IP-cores to be active. To emulate this behavior and to see from experiments the trends on bandwidth and latency as more flow become active, I define four use-cases. Three of the use-case activate only a subset of the flows, while the last one will use all. A description of the use-case is provided in Table 8.2.

Use-case 1 (UC1) uses only the Video decoder IP-core and the HDMI display IP-core to provide basic video playback. Use-case 2 assumes that along with the video playback,

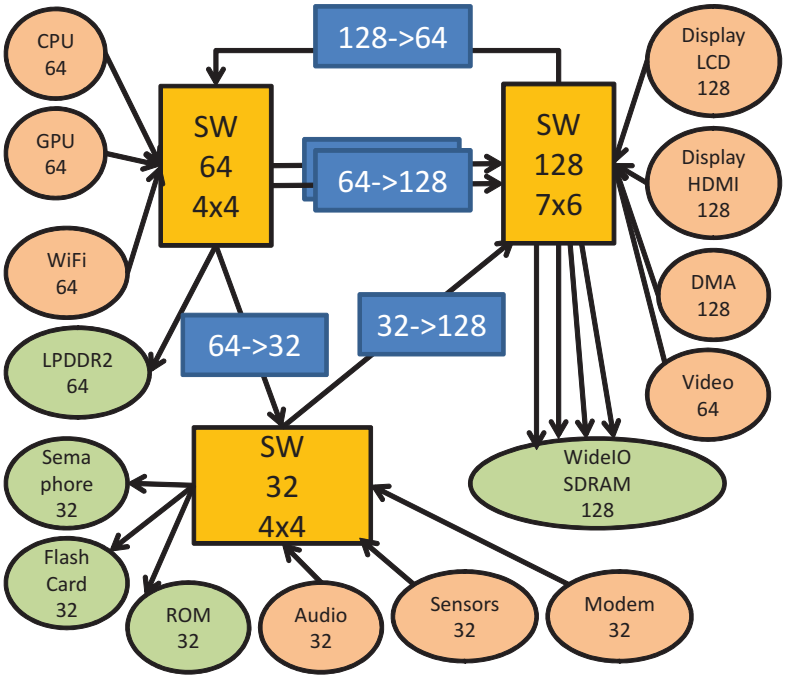

Figure 8.25: Topology

applications are also running on the CPU that require 3D graphic acceleration. In use-case 3, two displays are used. In case of the LCD display the resolution is considered to be less than 1080p so downscaling and rotation operations are needed. All the IP-cores that are active in UC3 are high bandwidth. In use-case 4, I consider the influence of the remaining IP-cores most of which are low bandwidth, but may require low latency as is the case of the Modem IP-core.

8.2.2.3 Topology

For the experiments I use a 6 switch NoC topology. To prevent message level deadlocks, I use two separate networks so 3 of the switches are used for the request network and 3 for the response network. The request network topology is presented in Figure 8.25 and the response network is symmetric, the only difference being that the data flows in the opposite direction. In the figure the round nodes represent the IP-core (in the simulator these require an instance of the traffic generator as well as an instance of an NI) and the number below the name represents the size of the data interface for that IP-core. The low bandwidth IP-cores have a 32 bit interface, while the high bandwidth IP-cores use 64 bits. Since the WideIO SDRAM has 4 channels with a data size of 128 bits, to be able to transfer full bandwidth I assume the DRAM controller interface toward the NoC is also 128 bit wide. Similarly some if the IP-core that generate large

regular transactions like the display drivers also use 128 bit data interfaces.

As can be seen from the figure the IP-cores that have the same data width are clustered on the same switch. To reduce the power consumption the switches have a FLIT size similar to the data size of the cores connected to it. The only exception is the Video IP-core which is connected directly to the 128 bit switch. When transactions are converted into packets, the bandwidth requirements increase due to the extra information that is added in each packet. Since the Video IP-core has high bandwidth demands, a link in the NoC with a 64bit FLIT width could not support the bandwidth resulting after packetization and therefore I connected it directly to the wider switch. Even though the Video IP-core generates significant traffic, it does so in small transaction and therefore I assumed that it has a 64bit data-interface. Also this setup allows us to capture the packetization effects from different data-sizes.

Since the switches have different FLIT sizes, converters need to be used on each link to change the width of the FLITS in each packet. It is important to accurately simulate the size converters as they can also increase the bandwidth when going from narrow to wide. With wormhole switching, once the head of a packet has reserved a channel, that channel remains reserved until the tail of the packet passes through it. Because of the wormhole channel reservation policy, a packet after the converter has the same number of FLITs as on the link before. Even though some of the FLITs in the wide part of the NoC are empty (there appears to be a FLIT because the channel is reserved), the resulting effect manifests itself as having higher bandwidth for the flow. However in case of downstream contention these empty FLITs can be dropped. To have accurate results, I model all these effects.

Apart from the IP-cores described in the communication graph of the benchmark, I attached another target IP-core on the narrow 32 bit switch. This core called *Semaphore* is used to synchronize the operation of the Video, LCD and HDMI IP-cores when they change from one frame to another.

8.2.3 Experimental results

In the experiments, using the benchmark described in Section 8.2.2.2, I analyze how the bandwidth and latency are affected by the presence of interfering flows at the WideIO DRAM memory. I analyze 5 setups based on the three architectures described in Section 8.2.1. The simulations setups with their corresponding names are the following:

- *Separate 1CH*: uses a distributed DRAM controller architecture with separate independent channels. This simulates the worst-case when all the active memory regions addressed by the ITGens are mapped to the same channel;

- *Separate 4CH*: uses the same distributed architecture, but in this case the active memory regions were carefully mapped to different channels. For example for the Video IP-core which generate high bandwidth traffic, the address memory regions have been distributed in all channels.
- *Controller I 3 Port*: uses a centralized DRAM architecture with interleaving at the controller. As there are more ITGens than ports in this setup several ITGens have been assigned to use the same port (only three of the 4 ports are used).
- *Controller I 4 Port*: uses the same centralized architecture, but the ITGens have been assigned to all four ports so that there is less contention at the DRAM controller ports.
- *NI I*: uses a distributed DRAM controller architecture, however the address space it is interleaved between the four channels of the WideIO memory. In this case the initiator NIs are responsible to split transactions into packets and to send the packets to the appropriate memory channel according to the address.

The simulation parameters used for the initiator traffic generators are presented in Table 8.3. For the initiator NIs, I used 2-deep bus transaction buffers and 5-deep FLIT FIFOs. For the setups using the distributed DRAM controller with independent channels and for the centralized DRAM controller, reorder buffers are not needed as there is a single route to reach the memory controller for the ITGens that require responses to be return in-order. In case of the distributed controller with splitting and interleaving at the NI, reorder buffers are not needed for the CPU, GPU and Video IP-cores. That is because the largest transaction is the same size as the interleaving size so no splitting is required. Also these traffic generators accept responses out-of-order. In case of the LCD and HDMI display controllers as well as the DMA, the transactions have to be split and therefore a reorder buffer is needed. I used 24-deep reorder buffer (each entry in the reorder buffer is a bus transaction). In case of the Audio, WiFi, Sensor and Modem IP-cores, no splitting is necessary, but the responses have to be delivered in order so, there is a reorder buffer. Since these cores are low bandwidth, I use 16 deep reorder buffers.

8.2.3.1 Throughput oriented communication

In the benchmark, I use IP-cores that are throughput sensitive (Video, HDMI, LCD) and some that are latency sensitive (CPU, Modem). In this section, I will have a closer look at two representative throughput sensitive IP-cores, namely he Video accelerator and the HDMI display controller. Both IP-cores require a significant amount of bandwidth and have to synchronize after finishing a frame. The Video controller generates small transactions, while the HDMI display controller transfers

Table 8.3: Simulation parameters for the ITGens

	Out-of-order	Maximum outstanding		Number of transaction generators	Address mode	Transaction sizes
		Reads	Writes			
CPU	yes	4	8	6	trace+incremental	64b, 512b
GPU	yes	8	8	1	incremental	512b
Video	yes	30	30	5	block + random block addressing	128b, 256b
LCD	yes	10	10	1	incremental with synchronization	1kB
HDMI	yes	10	10	1	incremental with synchronization	1kB
DMA	yes	10	10	3	incremental	128b, 1kB
WiFi	no	4	4	1	incremental	128b
Sensors	no	2	2	1	incremental	32b
Audio	no	2	2	1	incremental	128b
Modem	no	2	2	1	incremental	256b

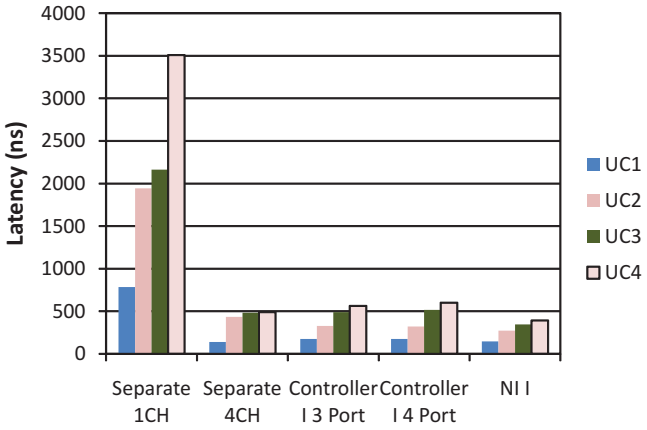

Figure 8.26: Average latency for the Video IP

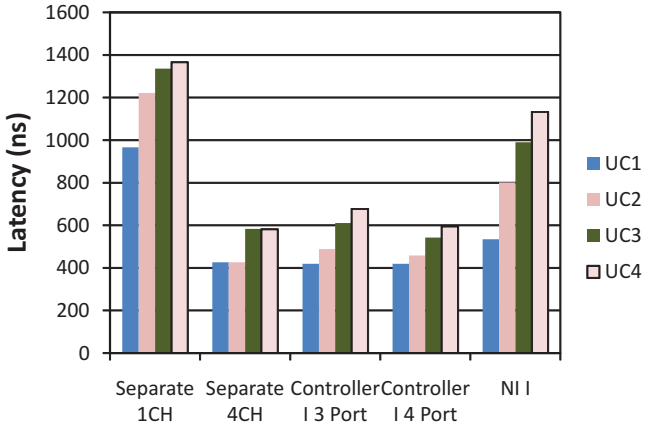

Figure 8.27: Average latency for the HDMI IP

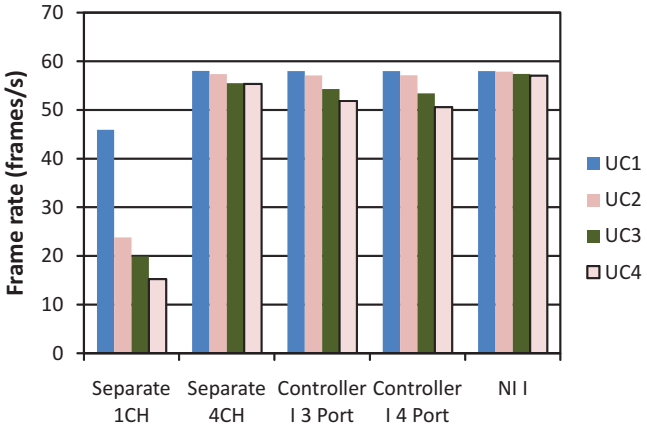

Figure 8.28: Average frame-rate

data in long 1kB burst transactions. In Figure 8.26, I show the trend of the average latency for the Video transactions as more interfering bandwidth is introduced with the more complex use-cases (in UC1 only the Video and the HDMI cores are active, while in UC4 all the IP-cores in the benchmark are active). As can be seen from the plot the distributed interleaving scheme that balances the traffic in the network as well has the lowest latency. Moreover compared to the other schemes, the distributed interleaving is also the least sensitive to the interfering bandwidth, as it has the lowest growth. The figure also shows that the mapping of data in memory has a significant impact on the performance of the distributed controller with independent channels. However with the best memory mapping as well as with the distributed controller with NI interleaving the latency is lower than in the case of the centralized controller. This is because both schemes exploit the parallelism in the NoC as well.

Similarly, in Figure 8.27, I show the average latency of the HDMI transactions. Please note that I measure latency of a transaction from the time it is generated by the ITGen and placed in the read/write buffer of the ITGen until the time it received the response and it is committed by the ITGen (in case of out-of-order ITGens the transactions are committed in-order so any completed transactions cannot be committed until all previous transactions have been completed and committed). In case of the HDMI core, data is transfered in 1kB transactions. Therefore in case of the distributed controller with interleaving in the NI, the transactions have to be split over multiple packets, in order to be sent to the appropriate channel. The different response packets have to be reordered and assembled such that the response from the NI to the ITGen is sent as a single burst of in-order bus transfers. The number of packets of the transactions that can be sent out is limited by the size of the reorder buffer. This results in a higher latency per transaction for the HDMI core as can be seen from the figure. However since a single transactions transfers 1kB of data these transfers are very efficient and the increased latency does not impact the overall performance of the system. If we look at the latency per byte for these 1kB transfers we can see that they are 11 times more efficient than the smaller transfers generated by the Video core (1.1 ns/byte for HDMI and 12.2 ns/byte for the Video in UC4), which explains why the larger latency of the HDMI core does not impact the system performance.

To analyze the overall system performance, in Figure 8.28, I show the frame rates obtained for the different use-cases in each simulation setup. The distributed DRAM controller with interleaving at the NI obtains the best frame rate and also has the lowest degradation of the frame rate as more interfering communication flows are activated. The frame rate of this setup is better even than that of the distributed controller with separate independent channels even for the best memory allocation scheme, as it balances the traffic from all the cores in the network as well. Surprisingly the centralized controller where only three ports are used has better frame rate than

when 4 ports are used. That is because in the case of the three port setup some of the interfering flows that are mapped on the same port interferer among themselves impacting their performances and in this particular case favoring the Video core. This also shows the sensitivity of the centralized controller scheme to the mapping of the communication flows to ports.

8.2.3.2 Bandwidth analysis

The benchmark is DRAM centric with all communication being between the IP-cores and the DRAM controller to be is representative of real applications. Therefore one important metric to assess the performance of the SoC is also the total read/write bandwidth to the DRAM memory. In Figure 8.29, I show the bandwidth obtained with each simulation setup for all use-cases normalized to the bandwidth required by that use-case. Please note that the reported bandwidth refers to the effective data bandwidth measured by the ITGens, the actual bandwidth transfered in the NoC is higher due to packetization and the bandwidth inflation generated by the size converters.

As expected, due to the balancing of traffic through the NoC as well, the distributed DRAM controller with NI interleaving setup achieves the best performance in terms of total average bandwidth as well (6184MB/s out of 6980MB/s demanded by use-case 4). The distributed controller with separate independent channels achieves similar performance for the most demanding use-case when the allocated memory is well distributed in the channels. However, to achieve that performance with independent channels the hardware would need to be exposed to the software and the managing operating systems should be smart enough to allocate memory such as to balance the channel usage. In case of the hardware interleaving scheme the memory access is transparent to the software, favoring the NI interleaved scheme which balances traffic in the interconnect as well.

8.2.3.3 Latency sensitive communication

So far I have analyzed the bandwidth oriented communication flows. In this section I analyze two of the latency sensitive IP-cores. One of them is the CPU where read are on the critical execution path and therefore the CPU performance is dependent on the latency of reads. On the other hand the CPU generates a significant amount of bandwidth as well. Another latency sensitive core is the Modem which has strict latency requirement imposed by the communication protocol, however the Modem core only requires little bandwidth.

In Figure 8.30, I present the average latency of the CPU for the three use-cases where

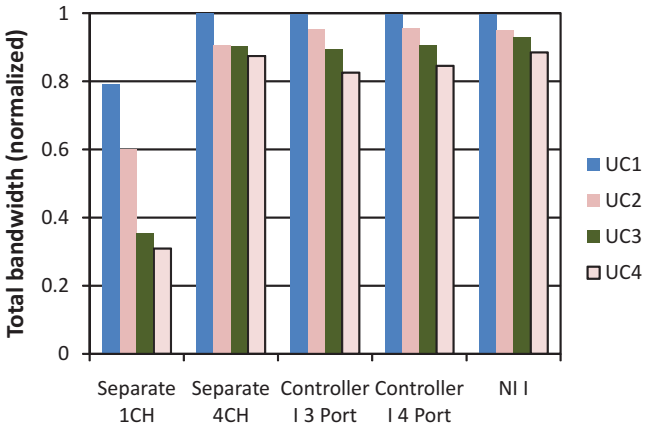

Figure 8.29: Average total bandwidth

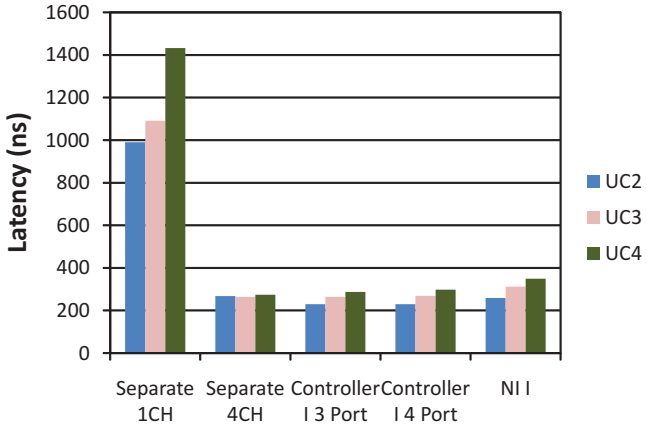

Figure 8.30: Average latency for the CPU IP

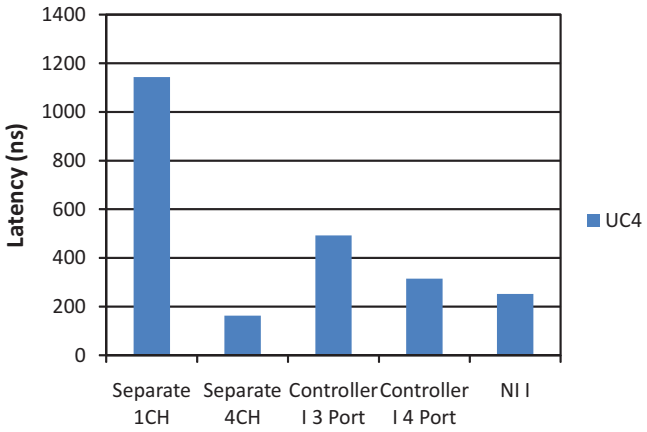

Figure 8.31: Average latency for the Modem IP

the CPU is active. As can be seen from the figure the latency of the CPU is only slightly larger for the distributed controller with interleaving at the NI when compared to the centralized controller (around 17% for UC4). That is because in the NI interleaving scheme the transaction from the CPU are distributed to all ports and they interfere with the traffic of the high-bandwidth cores at all the ports. In the other case the CPU is mapped by itself to one port. Please note however that in the case of the distributed controller with separate independent channels the mapping of memory to channels is application dependent and it may not be possible to achieve a good mapping in all cases. As can be seen from the figure for a bad mapping of the memory to channels, the controller with separated channels has very poor performance. Similarly the latency of my solution is slightly larger than that of the centralized controller. Nevertheless, it may be very expensive to physically implement the centralized controller (due to the distance between the channel interfaces), and as such the distributed interleaving method provides a good trade off. Moreover the distributed controller can provide other opportunities for the adaptation of the NoC topology (i.e., connect the different channel controller to different switches).

In Figure 8.31, I show the average latency for the Modem core. This core is latency sensitive, but has low bandwidth requirements. From the setups that perform interleaving the NI interleaved case has the lowest latency. The latency for the controller with separated independent channels, for the best mapping has the lowest latency. That is because in this case the communication of the Modem is to a separated channel where there are no high-bandwidth cores mapped. Interestingly, as the core has low bandwidth its latency is more affected by the port assignment in the case of the centralized controller.

8.3 Summary

In this chapter, I presented the power and performance trade-offs for the NoC design in the presence of a bottleneck core like the DRAM controller. I analyzed three designs and showed that physically separating the DRAM traffic from the non DRAM traffic leads to much lower latency for the non DRAM flows. I also showed that the power overhead of physically splitting the DRAM and non DRAM traffic is similar to other solutions like *end-to-end flow control*, which makes for the use of a *split* network to the DRAM controller an efficient solution. In the presence of heterogeneous cores that have different data sizes I show how the DRAM network can be optimized to reduce the number of data size converter. I also proposed a new architecture for the data size converter so that it can transfer data at full bandwidth. The optimized network leads to power savings of 23%.

3D-stacked WideIO DRAM memory promises to deliver the required bandwidth to

satisfy the demands of current and future application running on mobile SoCs at acceptable power-levels, by providing multiple channels and wide data interfaces. The challenge for SoC designers is to efficiently access the multiple channels provided by the WideIO interface, to achieve the required bandwidth and latency for communication flows. In this chapter, I also analyzed three architecture for designing the memory controller that access the WideIO DRAM memory. I proposed a new distributed interleaving method for one of the architectures. The new method capitalizes on the advantages of the two analyzed methods, to provide a simple DRAM controller design and to balance the traffic in both the NoC and at the memory channels transparent to the software. From experiments, I show that the proposed distributed interleaving memory access method improves the overall throughput while minimally impacting the performance of latency sensitive communication flows.

9 Conclusions and future directions

In this chapter, I summarize the major contribution of the thesis. I will also present a summary of the possible future avenues that can be pursued.

9.1 Summary of the thesis

In the thesis I have focused on two main aspects: i) in the first part I have presented methods to design NoCs considering the constraints and challenges of new technologies and ii) in the second part I have dealt with achieving different levels of *Quality of Service* (QoS). While the first part focuses exclusively on design aspects, the second part combines both design methods with performance estimation techniques to achieve predictable operation of the NoC topologies.

In Chapter 3, I have presented algorithms and methods to perform application specific NoC topology synthesis for SoCs designed and manufactured using 3D integration technologies. The methods not only take into account the new constraints imposed by 3D integration, but also address some new problems that do not exist in 2D integration (i.e., the assignment of switches to layers). These methods have then been extended to design NoCs for multi-synchronous SoCs in Chapter 4. They also address the possibility of shutdown of clock domains to reduce the standby power. In Chapter 5, I have presented an algorithm to detect and remove deadlocks in application-specific topologies. The algorithm minimally adds communication channels to reroute flows as to remove deadlock conditions. I have also shown how this algorithm can be integrated with the topology synthesis for 3D-NoCs, as to improve the quality of solutions in SoCs that have tight constraints on vertical connectivity.

To provide worst-case latency guarantees with best effort NoC components, in Chapter 6, I have integrated a worst case performance analytical model with the synthesis algorithm. The resulting synthesis method can realize NoC topologies that optimize the

power consumption and also satisfy hard real-time constraints. As many applications require average case performance, in Chapter 7, I have presented a fast evaluation model for the NoC performance, base on partial simulation. This model speeds up the evaluation time and still maintains the advantages of full simulation. In order to provide predictable performance, it is not sufficient to only analyze the NoC, but the bottleneck peripherals have to be evaluated as well. Therefore in Chapter 8, I have investigated different aspects related to accessing DRAM memory. I have also proposed several architectures to address the related problems with memory access.

9.2 Summary of the main contributions

With this thesis I bring two kinds of contributions: scientific and engineering. Based on the nature of the contributions, they are summarized as follows:

- *Scientific:* I presented new algorithms to perform NoC topology synthesis in new technologies (e.g. three dimensional integration and multi-synchronous design). I have also presented an algorithm to detect and remove deadlocks in application specific NoC topologies as well as to perform synthesis of predictable NoC topologies. These methods can be integrated in tools to be used by designers when developing NoCs and can also be used as a starting point to solve similar problems related to NoC design.
- *Engineering:* The engineering contribution comes from the integration of the proposed methods in CAD tools that have been used to preform extensive experiments which are presented in the thesis. The experiments can steer designers to restrict the broad design space that they need to explore when designing the interconnect. I have also performed experiments and presented architecture and results for efficiently accessing current and future multi-channel memory systems.

9.3 Future directions

There are still many different research topics that are not covered by this thesis. In this section I will mention some of the directions that can be pursued based on the work in this thesis:

- *Design of NoCs with distributed switch architectures:* The synthesis design methods that I have presented in thesis assume a NoC architecture based on centralized switches as in the ×pipes library [160]. These switches are configurable in the number of inputs and outputs. However a M by N switch can be distributed

in N switches of size M by 1. Some NoC libraries, like the one from Arteris [170], use this distributed switch architecture. A distributed switch offers different opportunities to establish the connectivity and it may require different synthesis methods to optimize the designed NoC.

- *Automatic synthesis of the DRAM network for heterogeneous IP-core data-bus sizes:* In Chapter 8, I show how a separate network to DRAM can be optimized in the presence of IP-cores with heterogeneous data-bus sizes. However, the optimization was done manually. A synthesis algorithm can be designed based on the presented observations to design automatically such NoCs toward the DRAM memory controller.

- *Detection and avoidance of message-level deadlocks:* To avoid message level deadlocks, currently message classes (e.g. requests, responses) are separated either by using different virtual channels or by using different physical channels (i.e., separate networks per message class). However in power constrained SoCs it may be useful to share channels between the message classes provided that by sharing message-level deadlocks cannot appear. The methods that I presented in Chapter 5 could be extended to take into account the dependencies between message classes and to force the usage of a separate channel only where required.

- *Extension of the worst-case latency estimation model for multi-synchronous designs:* The worst-case latency models used in this thesis to design real-time NoCs with best-effort hardware supports only synchronous operation. Therefore the synthesis algorithm can only be used to design synchronous NoCs. By extending the model to support multiple frequencies, it would enable the synthesis algorithm to design NoC with hard worst-case latency guarantees in multi-synchronous SoCs, with minimal change.

- *Fault tolerant NoC topology synthesis:* As technology scaling continues to shrink transitory sizes, devices become unreliable and can break down during operation. To prevent the complete failure of an SoC a certain amount of redundancy can be added in the interconnect to prevent it. One important tasks is to find the optimal way of adding the redundancy during the NoC design process.

- *Provide Quality-of-Service for critical flows in multi-channel DRAM systems with distributed interleaving:* In Chapter 8, I have showed the advantages of using an interleaved memory systems when there are multiple memory channels available. Future work can focus on understanding the relationship between network topology and the distributed interleaving scheme, and to analyze how to implement low overhead *Quality-of-Service* support to aggressively reduce latency for critical flows.

Even tough NoCs are scalable and can provide the required performance for the on-chip communication, accessing storage still remains an important bottleneck for system performance. Therefore in my opinion the most important direction for future work is to provide better integration between the communication infrastructure and the DRAM controllers. By removing the boundaries between the NoC and the DRAM controllers and co-designing the NoC with the memory hierarchy could potentially open different avenues for optimization and for improving the scalability of the memory-system. As 3D-integration makes it possible to have large memories on-chip and provides different opportunities to interconnect the logic to the memory, offering services, required for efficient memory access, distributed in the NoC could potentially be the key to scalability of the memory-system (both in terms of storage capacity and transfer bandwidth). On the other hand continuing the work on improving the tools for constructing predictable application specific NoCs, in order to keep up with the developments in the integration technologies, is also important. The NoC should be a transparent communication service which complies with the communication requirements, so that the SoC designers can focus on the other bottlenecks to system performance, like memory bandwidth.

Bibliography

[1] C. Addo-Quaye. Thermal-aware mapping and placement for 3-D NoC designs. In *Proc. IEEE Int. SOC Conf*, pages 25–28, 2005.

[2] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. In *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, pages 70–73, 2003.

[3] S. N. Adya and I. L. Markov. Fixed-outline floorplanning: enabling hierarchical design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(6):1120–1135, 2003.

[4] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software ISPASS 2009*, pages 33–42, April 26–28, 2009.

[5] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. 27th Int Computer Architecture Symp*, pages 248–259, 2000.

[6] Jung Ho Ahn, Mattan Erez, and William J. Dally. The Design Space of Data-Parallel Memory Systems. In *Proc. ACM/IEEE SC 2006 Conf*, 2006.

[7] Eero Aho, Jari Nikara, Petri A. Tuominen, and Kimmo Kuusilinna. A case for multi-channel memories in video recording. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 934–939, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[8] Tapani Ahonen, David A. Sigüenza-Tortosa, Hong Bin, and Jari Nurmi. Topology optimization for application-specific networks-on-chip. In *Proceedings of the 2004 international workshop on System level interconnect prediction*, SLIP '04, pages 53–60, New York, NY, USA, 2004. ACM.

[9] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 1–6, 2011.

[10] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *Proc. 5th IEEE/ACM/IFIP Int Hardware/Software Codesign and System Synthesis (CODES+ISSS) Conf*, pages 251–256, 2007.

[11] B. Akesson, Po-Chun Huang, F. Clermidy, D. Dutoit, K. Goossens, Yuan-Hao Chang, Tei-Wei Kuo, P. Vivet, and D. Wingard. Memory controllers for high-performance and real-time mpsocs requirements, architectures, and future trends. In *Proc. 9th Int Hardware/Software Codesign and System Synthesis (CODES+ISSS) Conf*, pages 3–12, 2011.

[12] R. Anigundi, Hongbin Sun, Jian-Qiang Lu, K. Rose, and Tong Zhang. Architecture design exploration of three-dimensional (3d) integrated dram. In *Proc. Quality Electronic Design Quality of Electronic Design ISQED 2009*, pages 86–90, 2009.

[13] Arteris. http://www.arteris.com/pr_22_oct_08, 2008.

[14] Semiconductor Industry Association. The international technology roadmap for semiconductors (itrs). http://www.itrs.net/, Edition 2007.

[15] Tezzaron at:. http://www.tezzaron.com.

[16] D. Atienza, P. G. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and J. M. Mendias. A fast HW/SW FPGA-based thermal emulation framework for multi-processor system-on-chip. In *Proc. 43rd ACM/IEEE Design Automation Conf*, pages 618–623, 2006.

[17] R. Leland B. Hendrickson. The Chaco Users Guide: Version 2.0. Sandia Tech Report SAND942692, 1994.

[18] K. Banerjee, S. J. Souri, P. Kapur, and K. C. Saraswat. 3-D ICs: a novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration. *Proceedings of the IEEE*, 89(5):602–633, 2001.

[19] P. Batude, M. Vinet, A. Pouydebasque, C. Le Royer, B. Previtali, C. Tabone, J.-M. Hartmann, L. Sanchez, L. Baud, V. Carron, A. Toffoli, F. Allain, V. Mazzocchi, D. Lafond, S. Deleonibus, and O. Faynot. 3D monolithic integration. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, pages 2233 –2236, may 2011.

[20] E. Beigné, F. Clermidy, S. Miermont, and P. Vivet. Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-*

Chip, NOCS '08, pages 129–138, Washington, DC, USA, 2008. IEEE Computer Society.

[21] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous NOC architecture providing low latency service and its multi-level design framework. In *Proc. 11th IEEE Int. Symp. Asynchronous Circuits and Systems ASYNC 2005*, pages 54–63, 2005.

[22] E. Beigne and P. Vivet. Design of on-chip and off-chip interfaces for a GALS NoC architecture. In *Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium on*, pages 10 pp. –183, march 2006.

[23] L. Benini and G. De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, pages 418–419, 2002.

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

[25] D. Bertozzi, A. Jalabert, Srinivasan Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 16(2):113–129, 2005.

[26] E. Beyne. The rise of the 3rd dimension for system intergration. In *Proc. Int. Interconnect Technology Conf*, pages 1–5, 2006.

[27] T. Bjerregaard, S. Mahadevan, R. G. Olsen, and J. Sparsoe. An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip. In *Proc. Int System-on-Chip Symp*, pages 171–174, 2005.

[28] Tobias Bjerregaard and Jens Sparso. A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2–3):105 – 128, 2004. <ce:title>Special issue on networks on chip</ce:title>.

[30] A. Bouhraoua and E.L. Elrabaa. A High-Throughput Network-on-Chip Architecture for Systems-on-Chip Interconnect. In *System-on-Chip, 2006. International Symposium on*, pages 1 –4, nov. 2006.

[31] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004.

[32] Phil Casini. SoC Architecture to Multichannel Memory Management Using Sonics IMT. Technical report, Sonics, Inc., 2008.

[33] Tung-Chien Chen, Chung-Jr Lian, and Liang-Gee Chen. Hardware architecture design of an H.264/AVC video codec. In *Proc. Asia and South Pacific Conf. Design Automation*, 2006.

[34] Ge-Ming Chiu. The odd-even turn model for adaptive routing. *Parallel and Distributed Systems, IEEE Transactions on*, 11(7):729–738, 2000.

[35] T. Cinotti. Progettazione di Una Unita per la Comunicazione Asincrona per Link di Network on Chip. Tesi di Laurea, DEIS, University of Bologna, 2007.

[36] J. Cong, Jie Wei, and Yan Zhang. A thermal-driven floorplanning algorithm for 3D ICs. In *Proc. ICCAD-2004 Computer Aided Design IEEE/ACM Int. Conf*, pages 306–313, 2004.

[37] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra. Spidergon: a novel on-chip communication network. In *Proc. Int System-on-Chip Symp*, 2004.

[38] Robert Cypher and Luis Gravano. Requirements for deadlock-free, adaptive packet routing. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, PODC '92, pages 25–33, New York, NY, USA, 1992. ACM.

[39] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[40] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proc. Design Automation Conf*, pages 684–689, 2001.

[41] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[42] Giovanni De Micheli and Luca Benini. *Networks on Chips: Technology and Tools; electronic version*. Elsevier, Burlington, MA, 2006.

[43] J. Duato. Deadlock-free adaptive routing algorithms for multicomputers: evaluation of a new algorithm. In *Proc. Third IEEE Symp. Parallel and Distributed Processing*, pages 840–847, 1991.

[44] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *Parallel and Distributed Systems, IEEE Transactions on*, 6(10):1055–1067, 1995.

[45] T. Dumitras, S. Kerner, and R. Marculescu. Towards on-chip fault-tolerant communication. In *Proc. Asia and South Pacific Design Automation Conf the ASP-DAC 2003*, pages 225–232, 2003.

[46] D. Dutoit and A. Jerraya. 3D integration opportunities for memory interconnect in mobile computing architectures. *Future Fab CEA-Leti MINATEC*, Issue 34:pp. 38–45, 2010.

[47] T. Hamalainen E. Salminen, A. Kulmala. Survey of Network-on-Chip Proposals. http://www.ocpip.org, March 2008.

[48] N. Enright Jerger, Li-Shiuan Peh, and M. Lipasti. Circuit-Switched Coherence. In *Proc. Second ACM/IEEE Int. Symp. Networks-on-Chip NoCS 2008*, pages 193–202, 2008.

[49] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[50] M. Facchini, T. Carlson, A. Vignon, M. Palkovic, F. Catthoor, W. Dehaene, L. Benini, and P. Marchal. System-level power/performance evaluation of 3d stacked drams for mobile applications. In *Proc. DATE '09. Design, Automation & Test in Europe Conf. & Exhibition*, pages 923–928, 2009.

[51] F. Fallah and M. Pedram. Standby and active leakage current control and minimization in CMOS VLSI circuits. *IEICE Trans. Electron. (Special Section on Low-Power LSI and Low-Power IP)*, E88-C:509, 2005.

[52] B. Feero and P. P. Pande. Performance Evaluation for Three-Dimensional Networks-On-Chip. In *Proc. IEEE Computer Society Annual Symp. VLSI ISVLSI '07*, pages 305–310, 2007.

[53] B. S. Feero and P. P. Pande. Networks-on-Chip in a Three-Dimensional Environment: A Performance Evaluation. *Computers, IEEE Transactions on*, 58(1):32–45, 2009.

[54] F. Feliciian and S.B. Furber. An asynchronous on-chip network router with quality-of-service (QoS) support. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 274 – 277, sept. 2004.

[55] Christopher J. Glass, Christopher J. Glass, Lionel M. Ni, and Lionel M. Ni. Maximally Fully Adaptive Routing in 2D Meshes. In *In International Conference on Parallel Processing, volume I*, pages 101–104, 1992.

[56] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. *J. ACM*, 41:874–902, September 1994.

[57] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, sept.-oct. 2005.

[58] B. Goplen and S. S. Sapatnekar. Placement of thermal vias in 3-D ICs using various thermal objectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(4):692–709, 2006.

[59] C. . Guedj, N. . Claret, V. . Arnal, M. . Aimadeddine, and J. P. Barnes. Evidence for 3-D/2-D Transition in Advanced Interconnects. *Device and Materials Reliability, IEEE Transactions on*, 7(1):64–68, 2007.

[60] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proc. Automation and Test in Europe Conf Design and Exhibition 2000*, pages 250–256, 2000.

[61] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 3(5):398–412, 2009.

[62] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, CODES+ISSS '05, pages 75–80, New York, NY, USA, 2005. ACM.

[63] M. Hayenga, N. E. Jerger, and M. Lipasti. SCARAB: A single cycle adaptive routing and bufferless network. In *Proc. MICRO-42 Microarchitecture 42nd Annual IEEE/ACM Int. Symp*, pages 244–254, 2009.

[64] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[65] W.H. Ho and T.M. Pinkston. A design methodology for efficient application-specific on-chip interconnects. *Parallel and Distributed Systems, IEEE Transactions on*, 17(2):174 – 190, feb. 2006.

[66] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein. Scaling, power, and the future of CMOS. In *Proc. IEDM Technical Digest Electron Devices Meeting IEEE Int*, 2005.

[67] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proc. IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, pages 108–109, 2010.

[68] Jingcao Hu and R. Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proc. Asia and South Pacific Design Automation Conf the ASP-DAC 2003*, pages 233–239, 2003.

[69] Jingcao Hu and Radu Marculescu. Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10688–, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Yun Huang, Qiang Zhou, Yici Cai, and Haixia Yan. A thermal-driven force-directed floorplanning algorithm for 3D ICs. In *Proc. 11th IEEE Int. Conf. Computer-Aided Design and Computer Graphics CAD/Graphics '09*, pages 497–502, 2009.

[71] W.-L. Hung, G. M. Link, Yuan Xie, N. Vijaykrishnan, and M. J. Irwin. Interconnect and thermal-aware floorplanning for 3D microprocessors. In *Proc. 7th Int. Symp. Quality Electronic Design ISQED '06*, 2006.

[72] Luca Benini Igor Loi, Federico Angiolini. Supporting vertical links for 3D networks-on-chip: toward an automated design and analysis flow. In *Proceedings of the 2nd international conference on Nano-Networks*, Catania, Italy, September 24-26 2007.

[73] iNoCs website at:. http://www.inocs.com/.

[74] Texas instruments. Ti's omap platform. http://focus.ti.com/omap/docs/, 2004.

[75] IMEC 3D integration at:. http://www2.imec.be/imec_com/3d-integration.php.

[76] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. ×pipesCompiler: a tool for instantiating application specific networks on chip. In *Proc. Design, Automation and Test in Europe Conf. and Exhibition*, volume 2, pages 884–889, 2004.

[77] Wooyoung Jang and D. Z. Pan. An SDRAM-Aware Router for Networks-on-Chip. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(10):1572–1585, 2010.

[78] DB Johnson. Finding all the elementary circuits of a directed graph. In *SIAM J. Comput.*, March 1975.

[79] A. B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proc. DATE '09. Design, Automation & Test in Europe Conf. & Exhibition*, pages 423–428, 2009.

[80] N. Kavaldjiev, G.J.M. Smit, and P.G. Jansen. A virtual channel router for on-chip networks. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 289 – 293, sept. 2004.

[81] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523 –1543, dec 2000.

[82] Dae Hyun Kim, K. Athikulwongse, and Sung Kyu Lim. A study of Through-Silicon-Via impact on the 3D stacked IC layout. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design - Digest of Technical Papers ICCAD 2009*, pages 674–680, 2009.

[83] Donghyun Kim, Kwanho Kim, Joo-Young Kim, Seung-Jin Lee, and Hoi-Jim Yoo. Solutions for Real Chip Implementation Issues of NoC and Their Application to Memory-Centric NoC. In *Proc. First Int. Symp. Networks-on-Chip NOCS 2007*, pages 30–39, 2007.

[84] Donghyun Kim, Kwanho Kim, Joo-Young Kim, Seungjin Lee, and Hoi-Jun Yoo. Implementation of Memory-Centric NoC for 81.6 GOPS object recognition processor. In *Proc. IEEE Asian Solid-State Circuits Conf. ASSCC '07*, pages 47–50, 2007.

[85] Dongki Kim, Sungjoo Yoo, and Sunggu Lee. A Network Congestion-Aware Memory Controller. In *Proc. Fourth ACM/IEEE Int Networks-on-Chip (NOCS) Symp*, pages 257–264, 2010.

[86] Jongman Kim, Chrysostomos Nicopoulos, Dongkook Park, Reetuparna Das, Yuan Xie, Vijaykrishnan Narayanan, Mazin S. Yousif, and Chita R. Das. A novel dimensionally-decomposed router for on-chip communication in 3d architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 138–149, New York, NY, USA, 2007. ACM.

[87] Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong-Ryol Hwang, Sooman Hwang, Byongwook Na, Joungwook Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang-Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyoon Cho, Minsoo Jang, Cheolmin Han, Jung-Bae Lee, Kyehyun Kyung, Joo-Sun Choi, and Young-Hyun Jun. A 1.2v 12.8gb/s 2gb mobile wide-i/o dram with 4x128 i/os using tsv-based stacking. In *Proc. IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, pages 496–498, 2011.

[88] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *Micro, IEEE*, 9(1):25 –40, feb 1989.

[89] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proc. IEEE Computer Society Annual Symp. VLSI*, pages 105–112, 2002.

[90] D. E. Lackey, P. S. Zuchowski, T. R. Bednar, D. W. Stout, S. W. Gould, and J. M. Cohn. Managing power and performance for system-on-chip designs using Voltage Islands. In *Proc. IEEE/ACM Int. Conf. Computer Aided Design ICCAD 2002*, pages 195–202, 2002.

[91] Kun-Bin Lee, Tzu-Chieh Lin, and Chein-Wei Jen. An efficient quality-aware memory controller for multimedia platform SoC. *Circuits and Systems for Video Technology, IEEE Trans. on*, 15(5):620–633, 2005.

[92] Sunggu Lee. Real-time wormhole channels. *Journal of Parallel and Distributed Computing*, 63(3):299 – 311, 2003.

[93] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest. Spatial division multiplexing: a novel approach for guaranteed throughput on nocs. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, CODES+ISSS '05, pages 81–86, New York, NY, USA, 2005. ACM.

[94] Lap-Fai Leung and Chi-Ying Tsui. Energy-Aware Synthesis of Networks-on-Chip Implemented with Voltage Islands. In *Proc. 44th ACM/IEEE Design Automation Conf. DAC '07*, pages 128–131, 2007.

[95] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 130–141, Washington, DC, USA, 2006. IEEE Computer Society.

[96] S. K. Lim. Physical design for 3D system on package. *IEEE Design & Test of Computers*, 22(6):532–539, 2005.

[97] Chuan Lin, Hai Zhou, and C. Chu. A Revisit to Floorplan Optimization by Lagrangian Relaxation. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD '06*, pages 164–171, 2006.

[98] D. H. Linder and J. C. Harden. An adaptive and fault tolerant wormhole routing strategy for k -ary n-cubes. *Computers, IEEE Transactions on*, 40(1):2–12, 1991.

[99] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proc. 35th Int. Symp. Computer Architecture ISCA '08*, pages 453–464, 2008.

[100] G. H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Proc. MICRO-42 Microarchitecture 42nd Annual IEEE/ACM Int. Symp*, pages 201–212, 2009.

[101] Gabriel H. Loh, Yuan Xie, and Bryan Black. Processor Design in 3D Die-Stacking Technologies. *IEEE Micro*, 27(3):31–48, 2007.

[102] I. Loi, F. Angiolini, and L. Benini. Developing Mesochronous Synchronizers to Enable 3D NoCs. In *Proc. Design, Automation and Test in Europe DATE '08*, pages 1414–1419, 2008.

[103] I. Loi and L. Benini. An efficient distributed memory interface for many-core platform with 3d stacked dram. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 99–104, 2010.

[104] I. Loi, S. Mitra, T. H. Lee, S. Fujita, and L. Benini. A low-overhead fault tolerance scheme for TSV-based 3D network on chip links. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD 2008*, pages 598–602, 2008.

[105] Pedro Lopez and Jose Duato. Deadlock-free adaptive routing algorithms for the 3D-torus: Limitations and solutions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 684–687. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56891-3-59.

[106] Qiang Ma and E. F. Y. Young. Voltage island-driven floorplanning. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD 2007*, pages 644–649, 2007.

[107] Théodore Marescaux and Henk Corporaal. Introducing the SuperGT network-on-chip: SuperGT QoS: more than just GT. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 116–121, New York, NY, USA, 2007. ACM.

[108] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Evaluation of current QoS Mechanisms in Networks on Chip. In *System-on-Chip, 2006. International Symposium on*, pages 1 –4, nov. 2006.

[109] G. Michelogiannakis, D. Sanchez, W. J. Dally, and C. Kozyrakis. Evaluating Bufferless Flow Control for On-chip Networks. In *Proc. Fourth ACM/IEEE Int Networks-on-Chip (NOCS) Symp*, pages 9–16, 2010.

[110] ST Microelectronics. ST nomadik multimedia processor. http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm, 2004.

[111] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, pages 20890–, Washington, DC, USA, 2004. IEEE Computer Society.

[112] I. Miro-Panades, F. Clermidy, P. Vivet, and A. Greiner. Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture. In *Proc. Second ACM/IEEE Int. Symp. Networks-on-Chip NoCS 2008*, pages 139–148, 2008.

[113] N. Miyakawa. A 3D prototyping chip based on a wafer-level stacking technology. In *Proc. Asia and South Pacific Design Automation Conf. ASP-DAC 2009*, pages 416–420, 2009.

[114] M. Modarressi, H. Sarbazi-Azad, and M. Arjomand. A hybrid packet-circuit switched on-chip network based on SDM. In *Proc. DATE '09. Design, Automation & Test in Europe Conf. & Exhibition*, pages 566–569, 2009.

[115] Mehdi Modarressi, Hamid Sarbazi-Azad, and Arash Tavakkol. An efficient dynamically reconfigurable on-chip network architecture. In *Proc. 47th ACM/IEEE Design Automation Conf. (DAC)*, pages 166–169, 2010.

[116] F. Mondinelli, M. Borgatti, and Z.M.K. Vajna. A 0.13 um 1Gb/s/channel store-and-forward network-on-chip. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 141 – 142, sept. 2004.

[117] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 196–207, New York, NY, USA, 2009. ACM.

[118] Robert Mullins, Andrew West, and Simon Moore. The design and implementation of a low-latency on-chip network. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 164–169, Piscataway, NJ, USA, 2006. IEEE Press.

[119] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test of Computers*, 22(5):434–442, 2005.

[120] Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 27–32, New York, NY, USA, 2005. ACM.

[121] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. Designing application-specific networks on chips with floorplan information. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 355–362, New York, NY, USA, 2006. ACM.

[122] Srinivasan Murali and Giovanni De Micheli. Bandwidth-Constrained Mapping of Cores onto NoC Architectures. *Design, Automation and Test in Europe Conference and Exhibition*, 2:20896, 2004.

[123] Srinivasan Murali and Giovanni De Micheli. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. *Design Automation Conference*, 0:914–919, 2004.

[124] A. Nelson, A. Hansson, H. Corporaal, and K. Goossens. Conservative application-level performance analysis through simulation of mpsocs. In *Proc. 8th IEEE Workshop Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 51–60, 2010.

[125] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 26:62–76, February 1993.

[126] U. Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu. Voltage-Frequency Island Partitioning for GALS-based Networks-on-Chip. In *Proc. 44th ACM/IEEE Design Automation Conf. DAC '07*, pages 110–115, 2007.

[127] U.Y. Ogras, P. Bogdan, and R. Marculescu. An Analytical Approach for Network-on-Chip Performance Analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(12):2001 –2013, dec. 2010.

[128] M. Palesi, G. Longo, S. Signorino, R. Holsmark, S. Kumar, and V. Catania. Design of Bandwidth Aware and Congestion Avoiding Efficient Routing Algorithms for Networks-on-Chip Platforms. In *Proc. Second ACM/IEEE Int. Symp. Networks-on-Chip NoCS 2008*, pages 97–106, 2008.

[129] Dongkook Park, S. Eachempati, R. Das, A. K. Mishra, Yuan Xie, N. Vijaykrishnan, and C. R. Das. MIRA: A Multi-layered On-Chip Interconnect Router Architecture. In *Proc. 35th Int. Symp. Computer Architecture ISCA '08*, pages 251–261, 2008.

[130] M. Palesi; D. Patti; and F. Fazzino. NOXIM at: http://noxim.sourceforge.net.

[131] Christian Paukovits and Hermann Kopetz. Concepts of Switching in the Time-Triggered Network-on-Chip. *Real-Time Computing Systems and Applications, International Workshop on*, 0:120–129, 2008.

[132] V. F. Pavlidis and E. G. Friedman. 3-D Topologies for Networks-on-Chip. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(10):1081–1090, 2007.

[133] Philips. Philips nexperia highly integrated programmable system-on-chip (mp-soc). http://www.semiconductors.philips.com/products/nexperia, 2004.

[134] Alessandro Pinto, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. Efficient Synthesis of Networks On Chip. *Computer Design, International Conference on*, 0:146, 2003.

[135] Linear program solver. Package available at:. http://sourceforge.net/projects/lpsolve.

[136] V. Puente, J. A. Gregorio, and R. Beivide. SICOSYS: an integrated framework for studying interconnection network performance in multiprocessor systems. In *Proc. 10th Euromicro Workshop Parallel, Distributed and Network-based Processing*, pages 15–22, 2002.

[137] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4 – 17, jan. 2005.

[138] N. Rafique, Won-Taek Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proc. 16th Int. Conf. Parallel Architecture and Compilation Techniques PACT 2007*, pages 245–258, 2007.

[139] Dara Rahmati, Srinivasan Murali, Luca Benini, Federico Angiolini, Giovanni De Micheli, and Hamid Sarbazi-Azad. A method for calculating hard QoS guarantees for Networks-on-Chip. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 579–586, New York, NY, USA, 2009. ACM.

[140] M. Reza Kakoee, F. Angiolin, S. Murali, A. Pullini, C. Seiculescu, and L. Benini. A floorplan-aware interactive tool flow for NoC design and synthesis. In *Proc. IEEE Int. SOC Conf. SOCC 2009*, pages 379–382, 2009.

[141] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip. *Computers and Digital Techniques, IEE Proceedings* -, 150(5):294–302, sept. 2003.

[142] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. 27th Int Computer Architecture Symp*, pages 128–138, 2000.

[143] A. Sathanur, L. Benini, A. Macii, E. Macii, and M. Poncino. Multiple power-gating domain (multi-VGND) architecture for improved leakage power reduction. In *Proc. ACM/IEEE Int Low Power Electronics and Design (ISLPED) Symp*, pages 51–56, 2008.

[144] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *Selected Areas in Communications, IEEE Journal on*, 9(8):1318–1335, 1991.

[145] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. Noc topology synthesis for supporting shutdown of voltage islands in socs. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 822 –825, july 2009.

[146] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. Sunfloor 3d: A tool for networks on chip topology synthesis for 3d systems on chips. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 9 –14, april 2009.

[147] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. Comparative analysis of nocs for two-dimensional versus three-dimensional socs supporting multiple voltage and frequency islands. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 57(5):364 –368, may 2010.

[148] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A method to remove deadlocks in networks-on-chips with wormhole flow control. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1625 –1628, march 2010.

[149] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. SunFloor 3D: A Tool for Networks on Chip Topology Synthesis for 3-D Systems on Chips. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(12):1987 –2000, dec. 2010.

[150] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A dram centric noc architecture and topology design approach. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 54 –59, july 2011.

[151] C. Seiculescu, D.. Rahmati, S. Murali, L. Benini, G. De Micheli, and H. Sarbazi-Azad. Designing best effort networks-on-chip to meet hard latency constraints. *accepted in ACM Transactions on Embedded Computing Systems*, expected 2012.

[152] C. Seitz. Let's Route Packets Instead of Wires. In *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 133–138, 1990.

[153] Zheng Shi and Alan Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society.

[154] D. Siguenza-Tortosa and J. Nurmi. Proteo: A New Approach to Network-on-Chip. In *CSN 02*, Sep. 2002.

[155] IBM ASIC solutions at:. www.ibm.com.

[156] Open Core Protocol specifications at:. www.ocpip.org.

[157] K. Srinivasan and K. S. Chatha. A Low Complexity Heuristic for Design of Custom Network-on-Chip Architectures. In *Proc. Design, Automation and Test in Europe DATE '06*, volume 1, pages 1–6, 2006.

[158] K. Srinivasan, K. S. Chatha, and G. Konjevod. An automated technique for topology and route generation of application specific on-chip interconnection networks. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 231–237, Washington, DC, USA, 2005. IEEE Computer Society.

[159] David Starobinski, Mark Karpovsky, and Lev A. Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Netw.*, 11:411–421, June 2003.

[160] Stergios Stergiou, Federico Angiolini, Salvatore Carta, Luigi Raffo, Davide Bertozzi, and Giovanni De Micheli. ×pipes Lite: A Synthesis Oriented Design Library For Networks on Chips. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1188–1193, Washington, DC, USA, 2005. IEEE Computer Society.

[161] Hongbin Sun, Jibang Liu, R. S. Anigundi, Nanning Zheng, Jian-Qiang Lu, K. Rose, and Tong Zhang. 3d dram design and application to 3d multicore systems. *IEEE Design & Test of Computers*, 26(5):36–47, 2009.

[162] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[163] IBM thermal cooling at:. http://www.zurich.ibm.com/st/cooling/interfaces.html.

[164] Tilera. 100 core commercial processor TILE-Gx at: http://www.tilera.com/products/processors.php.

[165] Y.-F. Tsai, D. Duarte, N. Vijaykrishnan, and M. J. Irwin. Implications of technology scaling on leakage reduction techniques. In *Proc. Design Automation Conf*, pages 187–190, 2003.

[166] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

[167] I. Walter, I. Cidon, R. Ginosar, and A. Kolodny. Access Regulation to Hot-Modules in Wormhole NoCs. In *Proc. First Int. Symp. Networks-on-Chip NOCS 2007*, pages 137–148, 2007.

[168] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33:100–107, November 2005.

[169] Feng Wang and M. Hamdi. Scalable router memory architecture based on interleaved dram. In *High Performance Switching and Routing, 2006 Workshop on*, page 6 pp., 0-0 2006.

[170] Arteris webpage at:. http://www.arteris.com/flex_noc.php.

[171] Silistix webpage at:. http://www.silistix.com/.

[172] R. Weerasekera, Li-Rong Zheng, D. Pamunuwa, and H. Tenhunen. Extending systems-on-chip to the third dimension: performance, cost and technological tradeoffs. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD 2007*, pages 212–219, 2007.

[173] Herbert Weinblatt. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. *J. ACM*, 19:43–56, January 1972.

[174] C. Weis, N. Wehn, L. Igor, and L. Benini. Design space exploration for 3d-stacked drams. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 1–6, 2011.

[175] Dong Hyuk Woo, Nak Hee Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *Proc. IEEE 16th Int High Performance Computer Architecture (HPCA) Symp*, pages 1–12, 2010.

[176] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[177] Jiang Xu, Wayne Wolf, Joerg Henkel, and Srimat Chakradhar. A design methodology for application-specific networks-on-chip. *ACM Trans. Embed. Comput. Syst.*, 5:263–280, May 2006.

[178] Junhee Yoo, Sungjoo Yoo, and Kiyoung Choi. Multiprocessor system-on-chip designs with active memory processors for higher memory efficiency. In *Proc. 46th ACM/IEEE Design Automation Conf. DAC '09*, pages 806–811, 2009.

[179] L. Zakrevski, S. Jaiswal, L. Levitin, and M. Karpovsky. A new method for deadlock elimination in computer networks with irregular topologies. In *Proc. IASTED Conf. PDCS*, 1999.

[180] Pingqiang Zhou, Yuchun Ma, Zhouyuan Li, R. P. Dick, Li Shang, Hai Zhou, Xianlong Hong, and Qiang Zhou. 3D-STAF: scalable temperature and leakage aware floorplanning for three-dimensional integrated circuits. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD 2007*, pages 590–597, 2007.

[181] Xinping Zhu and Sharad Malik. A hierarchical modeling framework for on-chip communication architectures. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 663–671, New York, NY, USA, 2002. ACM.

CURRICULUM VITAE

First name / Surname: CIPRIAN SEICULESCU

Residence Address: F.C. RIPENSIA 17, Apt.18, 300584 Timisoara, Romania
Present Address: Avenue Tivoli 70, CH-1007, Lausanne, Switzerland

Romanian Phone No. 00-40-743/008697
Swiss Phone No. 00-41-787/846294

Work email: ciprian.seiculescu@epfl.ch
Personal email: cseiculescu@yahoo.com

Date of birth: SEPTEMBER 10, 1982
Place of birth: TIMISOARA, ROMANIA
Citizenship: ROMANIAN
Marital status: SINGLE

Education:

- MSc: in Computer Science, January 2008, from Ecole Polytechnique Federale de Lausanne**.** Graduation project: "*Design Framework and Methodology for Synthesis of Networks-On-Chip on FPGA Platforms and 3D Chips*"
- BSc: in Automation and Computer Science, September 2006, from 'Politehnica' University of Timisoara. Graduation project: "*3D Multi-Material Printing: Control of the Printing Heads*". The work for the graduation project was done at Fraunhofer IPA in Stuttgart, Germany.

Main publications:

- C. Seiculescu, S. Murali, L. Benini and G. De Micheli. *SunFloor 3D: A Tool for Networks on Chip Topology Synthesis for 3D Systems on Chips*, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, num. 12, p. 1987-2000, 2010.
- C. Seiculescu, S. Murali, L. Benini and G. De Micheli. *Comparative Analysis of NoCs for Two-Dimensional Versus Three-Dimensional SoCs Supporting Multiple Voltage and Frequency Islands*, in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 57, num. 5, p. 364 - 368, 2010.
- C. Seiculescu, S. Murali, L. Benini and G. De Micheli. *A Method to Remove Deadlocks in Networks-on-Chips with Wormhole Flow Control*. Design, Automation and Test in Europe Conference (DATE 2010), Dresden, Germany, 2010.
- C. Seiculescu, S. Murali, L. Benini and G. De Micheli. *NoC Topology Synthesis for Supporting Shutdown of Voltage Islands in SoCs*. DAC 2009, San Francisco, USA, 2009.
- C. Seiculescu, S. Volos, N. Khosro Pour, B. Falsafi and G. De Micheli. *CCNoC: On-Chip Interconnects for Cache-Coherent Manycore Server Chips*. Workshop on Energy-Efficient Design (WEED 2011), San Jose, California, USA, 2011.

Skills and knowledge of tools:

- **Programming languages:** C, C++, C#, Visual Basic, Java, Assembly (x86, 8051, MSP, AVR)
- **Programming development tools:** Visual Studio .Net, Eclipse, NetBeans, QT, Matlab & Simulink.
- **Hardware description languages:** VHDL.
- **Hardware development tools:** Xilinx Embedded Development Kit (EDK), Xilinx Integrated Software Environment (ISE), Altera Quartus II, ModelSim.

General knowledge:

- Microsoft Windows / Linux
- Microsoft Office / Open Office

Languages:

- Romanian: Native speaker.
- English: fluent (Toefl 273/300 CBT)

Interests and hobbies:

- Photography
- Astronomy
- Hiking