Modeling flexible Networks On-Chip

L. Pieralisi

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

␣␣␣␣␣␣␣␣␣␣␣␣

THESE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : « Micro et nano électronique »

préparée au laboratoire TIMA
dans le cadre de **l'Ecole Doctorale « *E.E.A.T.S.* »**

présentée et soutenue publiquement

par

Lorenzo Pieralisi

le 07/07/2006

Titre :

Modélisation de réseau de communication flexible pour les systèmes monopuce

---

Directeur de thèse : Ahmed Amine JERRAYA

JURY

M. Frédéric Pétrot	, Président
M. Amara Amara	, Rapporteur
M. Alain Greiner	, Rapporteur
M. Ahmed Amine Jerraya	, Directeur de thèse
M. Marcello Coppola	, Co-encadrant
M. Eric Flamand	, Examinateur

Résumé

Les systèmes monopuce deviennent de plus en plus complexes, intégrant composants à la fois logiciels et matériels dans le but de procurer une capacité de calcul croissante aux applications embarquées. L'interconnexion des composants devient un élément crucial de la conception ; il fournit aux concepteurs des fonctionalités avancées telles qu'opérations atomiques, transactions parallèles et primitives de communication permettant des systèmes sécurisés.
Le concept de réseau sur puce s'impose comme élément de communication pour les architectures d'interconnexion des systèmes de la prochaine génération. Le rôle des réseaux sur puce consiste à remplacer les bus partagés dont la mise à l'échelle comporte de sérieux problèmes de conception et représente un goulot d'étranglement pour le système global.
La modélisation d'un réseau sur puce est une tâche extrêmement complexe ; ces modèles doivent être à la fois rapides en terme d'exécution, précis et ils doivent exporter des interfaces standard afin d'en améliorer la réutilisation.
Les principales contributions de cet ouvrage sont représentées par : (1) le développement d'un simulateur de réseaux sur puce complet, précis au cycle près, basé sur OCCN, un logiciel de simulation libre disponible sur « sourceforge » à l'adresse http://occn.sourceforge.net , (2) l'intégration de plusieurs environnements de simulation hétérogènes en plate-formes très complexes utilisées pour étudier des systèmes monopuce réels produits par STMicroelectronics et (3) une connaissance complète des concepts sous-jacents aux réseaux sur puce qui a apporté une contribution importante au développement de STNoC™, la nouvelle technologie d'interconnexion de STMicroelectronics développée au sein du laboratoire Advanced System Technology (AST) de Grenoble. L'environnement de modélisation réalisé a été utilisé pour l'étude de deux systèmes monopuce réels développés par STMicroelectronics orientés vers la télévision numérique à très haute définition (HDTV).

Mots-clés : systèmes monopuce, qualité de service, modélisation au niveau transactionnel, co-simulation, SystemC, plate-forme, interface, POSIX, réseaux sur puce, couches réseaux, commutation par paquet.

Abstract

The Multi-Processors Systems on a chip (MPSoC) era is bringing about many new challenges for systems design in terms of computation and communication subsystems complexity. Interconnection systems became a pivotal component of the overall design, providing designers with advanced communication features such a split transactions, atomic operations and security adds-on. Momentum is building behind Networks on-chip (NoC) as future on-chip interconnection technology. Networks on-chip role is about to take over shared busses whose scalability properties are already a major bottleneck for system design.
Modeling of on-chip network is an exacting work; networks models must be fast, accurate and they have to sport standard interfaces. The main contributions of this work to networks on-chip design and implementation are: (1) the development of a brand new, full-fledged network on-chip simulator based on OCCN, an open-source framework for NoC modeling developed within sourceforge available at http://occn.sourceforge.net, (2) the successfull integration of heterogeneous simulation environments in extremely complex platforms used to benchmark real STMicroelectronics SoC and (3) a thorough understanding and contribution to the design of STNoC™, the new interconnection technology developed within AST Grenoble lab of STMicroelectronics for future generation systems. The modeling environment has been used to benchmark two STMicroelectronics systems on-chip for High Definition digital Television (HDTV).

Keywords: system on-chip, quality of service, transaction level modeling, co-simulation, SystemC, platform, interfaces, POSIX, networks on-chip, packet switching, layers.

"Don't be afraid to attempt the impossible.
Simply knowing what is impossible is
useful knowledge – and you may well find,
in the wake of some unexpected success,
that not half of the things we call impossible
have any right at all to wear the label."
– Michael Abrash

To mum and dad, with love

Acknowledgements

This is the part of my thesis I was raring to write, in order to thank all of the people involved directly and indirectly on its development. First and foremost, I want to thank Marcello and Master Jerraya for the simple reason that, without their immense help, I would not have achieved any of this, which actually means a lot to me. I have come a long way and I learned a lot from you. Believe me, it was such a pleasure, and the forthcoming years will prove what I am talking about. Frankly, thank you very much. A fond wink to the STNoC™ cabal (alphabetical order), Ennio (STNoC™ guestlist), Giuseppe, Marcela, Marcello, Michael, Miltos, Nicola, Payal, Philippe, Riccardo, Saurin, Valerio and all the guys who contributed to its development, including ST Tunis and ST Catania OCCS mates. Benchmarking made us toil, working together made us friends, and you know, at the end of the day, the acquired knowledge will make us win.
Great many thanks to Thesis committee in alphabetical order, Professor Amara AMARA, Engineer Eric FLAMAND, Professor Alain GREINER and Professor Frédéric PÉTROT, for accepting to review my PhD thesis and their willingness to attend its defence. A sad farewell from TIMA/SLS guys; working with you was a privilege and your kindness, liveliness and sympathy made my life ways better. I want to thank the Linux community for developing a top-notch operating system and for distributing it as open source. It is a monument of knowledge that must not be taken for granted. Let me thank Marcello, Stéphane and Gianluca for giving me the possibility to help them develop open-source software (OCCN). When you contribute to open source software your work is of small but immediate benefit to other people, which, after all, is the most enjoyable pleasure. Always remember, freedom does matter.
Last but not least, I want to dedicate this work to my beloved parents Bruno and Giancarla, and Ramona chérie, without whose help and love none of this would have been possible. I love you.

Lorenzo

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Présentation de la thèse

1.1 Problématique

Au cours des trois décennies écoulées, depuis 1975, l'industrie des semiconducteurs a été gouvernée par la loi de Moore, qui affirme que le niveau d'intégration des circuits double tous les 18 mois [34]. Ceci a amené à la possibilité d'intégrer dans une seule puce l'ensemble des fonctions d'un système informatique : acquisition des données depuis des capteurs analogiques ou des réseaux de communication, traitement du signal, analyse et prise de décisions, émission de données ou rétroaction sur le monde extérieur. Une telle situation laisse entrevoir le développement de produits électroniques de faible coût, destinés au grand public, et dotés de fonctions intelligentes, grâce à l'emploi de tels systèmes embarqués ou systems on-chip. Le concept de system-on-chip implique une complexité croissante concernant la spécification, la vérification fonctionnelle et temporelle ainsi que le test de circuits monolithiques de plus de 200 millions de transistors. Ces aspects étaient encore gérables pour des circuits plus petits, et leur intégration en plus grands systèmes modulaires était plus aisée, précisément parce que les éléments restaient séparables.
À l'heure actuelle, toutes les méthodes de conception commencent par fragmenter le système en éléments plus petits (processeurs, DSP, mémoires, interfaces, ...), si possible puisés dans des bibliothèques. Le problème incontournable qui se pose est le suivant :

- Par quel moyen implémenter la communication entre ces éléments ?

De nombreux travaux de recherche sont désormais présents en littérature, tous abordant l'étude d'une nouvelle méthodologie pour l'implémentation

de l'interconnexion pour circuits intégrés : les ainsi-dits *Networks-on-chip* [40]. Le rôle des réseaux sur puce consiste à résoudre les divers problèmes qui affligent la technologie d'interconnexion couramment utilisée pour les systèmes monopuce, le bus partagé. En particulier, ces systèmes contiennent de plus en plus d'éléments, ce qui implique des besoins en bande passante et latence maximum de plus en plus strictes, concept qui contraste avec la nature partagée d'un bus, qui pose donc des problèmes non négligeables en terme de mise à l'échelle.

1.2 Contribution

Les travaux de cette thèse consistaient à étudier, modéliser et implémenter, un réseau de communication flexible pour les futurs systèmes monopuce. Une nouvelle méthodologie de modélisation permettant la simulation au cycle près de réseaux complexes a été développée, qui a permis une étude approfondie de plusieurs system-on-chip au sein de STMicroelectronics. En particulier, les travaux de cette thèse ont contribué d'une façon significative au flot de conception de STNoC™, la nouvelle technologie d'interconnexion développée par STMicroelectronics. Les modèles développés dans le cadre de cette thèse constituent une partie fondamentale de la méthodologie de conception de réseaux au niveau système qui a été appliquée à deux systèmes monopuce réels développés par STMicroelectronics, orientés vers la télévision numérique à haute définition.

1.3 Présentation de la structure de la thèse

Ce manuscrit se compose de deux parties fondamentales :

- Une partie introductive en français, qui présente les principaux concepts et contributions.
- Une partie en anglais, dont le contenu décrit d'une façon détaillée les travaux développés.

La thèse est constituée des chapitres suivant :

- ce chapitre présente la structure de la thèse.

- le deuxième chapitre résume le contenu de la thèse en français, en décrivant les principaux points abordés, détaillés en anglais dans les chapitres qui suivent.

- le troisième chapitre réalise un état de l'art qui définit les composants d'interconnexion pour les systèmes monopuce existants ainsi que de nouvelles technologies qui viennent d'être déployées.

- le quatrième chapitre définit les couches dont un réseau se compose et les éléments architecturaux du réseau STNoC™ développé par STMicroeletronics.

- dans le cinquième chapitre, la méthodologie de modélisation de réseaux sur puce développée dans le cadre de cette thèse et appliquée au flot de conception de STNoC™ est expliquée en détail, ainsi qu'une méthode qui permet des simulations SystemC distribuées.

- le sixième chapitre complète la description des composants utilisés dans l'étude approfondie de systèmes monopuce réels développés par STMicroelectronics, où le bus d'interconnexion a été partiellement remplacé par une architecture de réseau STNoC™.

- le septième chapitre montre les résultats obtenus lors de tests effectués sur le réseau STNoC™ dans des conditions de trafic aléatoire et de trafic engendré par des IPs de STMicroelectronics.

- enfin le huitième chapitre tire les conclusions et décrit les évolutions futures concernant STNoC™ qui sont en cours chez STMicroelectronics.

Chapitre 2

Modélisation de réseau de communication flexible pour les systèmes monopuce

2.1 Introduction

Ce chapitre constitue un résumé de la thèse en français ; la thèse complète est développée en détail dans la deuxième partie, en anglais, qui commence au chapitre 3. Tout au long de ce chapitre, les concepts de base de réseaux sur puce et le flot de conception du STNoC™, la nouvelle technologie d'interconnexion au sein de STMicroelectronics, sont décrits, avec une attention particulière accordée à la modélisation du réseau et son implémentation.

2.2 État de l'art

Cette section vise à présenter les principales technologies qui ont été déployées jusqu'à présent et les principaux axes de recherche concernant les systèmes de communication sur puce.
La technologie la plus utilisée pour ce qui concerne l'interconnexion des systèmes monopuce est celle du bus partagé (voir figure 2.1). Elle a déjà été utilisée dans un très grand nombre de produits, de sorte que tous les systèmes intégrés utilisent à peu près des technologies équivalentes.

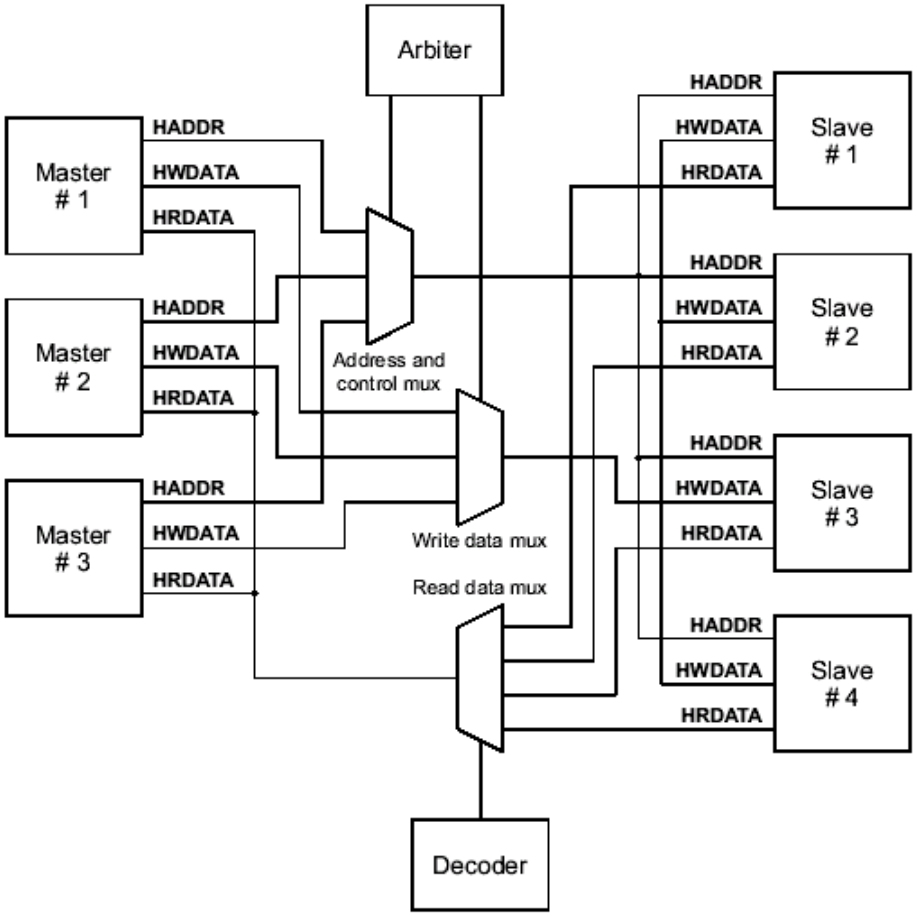

Fig. 2.1: Un exemple d'architecture de bus [2] (ARM AHB).

Un bus partagé se compose notamment :

- D'un bus de données partagé (unique et connecté à tous les éléments du système).
- D'un bus d'adresses ayant les mêmes caractéristiques.
- D'un élément particulier du système, appelé arbitre du bus, connecté par des liaisons point à point à tous les autres éléments.

Tous les éléments connectés au bus sont à même d'échantillonner ou piloter les lignes du bus de données ; par contre le bus d'adresses ne peut être piloté que par certains d'entre eux, les maîtres.
Dans une architecture de bus ces éléments appelés maîtres initient les transactions sur le bus en pilotant les bus d'adresses, transactions adressées à des esclaves qui sont identifiés par des plages d'adressage. La nature partagée d'un bus implique une étape d'arbitrage qui sert à choisir le maître qui peut disposer du bus. Cette centralisation est une caractéristique spécifique qui distingue les bus partagés des réseaux de communication pour lesquels l'arbitrage est distribué. Les transactions sont soit des écritures (le maître requiert une écriture dans une zone de mémoire appartenant à l'esclave), soit des lectures (le maître demande des données à l'esclave qui les lui retourne). Il existe plusieurs extensions qui peuvent être apportées aux protocoles des bus. Par exemple l'atomicité des primitives de synchronisation entre processeurs, pour la gestion de la cohérence des mémoires caches ainsi que pour la politique d'arbitrage.
Un bus partagé, et en particulier sa topologie, possède plusieurs avantages. Premièrement, il peut supporter directement le modèle de communication par adressage mémoires des CPUs, qui représente actuellement le modèle le plus répandu pour les IPs (« Intellectual Properties »). En y intégrant de légères modifications les interfaces du bus peuvent supporter des opérations complexes (communication par flux) et surtout réutilisables (« transaction-centric-design »). Deuxièmement, le mode de fonctionnement de l'arbitre est simple et on peut facilement l'adapter aux différentes applications requises. Finalement, le concept de bus est parfaitement maîtrisé par les concepteurs de matériel, ce qui favorise son usage et sa large diffusion.
Malheureusement un bus partagé a aussi des limites, impliquées dans la plupart des cas par sa nature partagée. Ce sont des limites qui ont stimulé sensiblement des travaux de recherche pour essayer de déterminer de vraies alternatives au concept de bus. Un bus partagé implique de nombreux problèmes de mise à l'échelle. Ceci s'avère être inacceptable, surtout pour le

débit global. Un arbitrage plus sophistiqué et des mémoires cache ne peuvent qu'alléger cette contrainte assez forte. Le problème peut être résolu de deux façons : soit en augmentant la largeur du bus, soit en augmentant la fréquence d'horloge. Les deux solutions ne sont pas satisfaisantes car elles impliquent des problèmes électriques auxquels il est coûteux de remédier convenablement.
Même si le débit global est suffisant, le bus n'est plus utilisable pour des grands systèmes avec un nombre élevé d'éléments à connecter. Dans ce cas, le goulot d'étranglement devient l'arbitre central qui doit choisir un maître auquel concéder le bus parmi des dizaines. Un arbitre central peut difficilement accomplir cette tâche efficacement, sans introduire un accroissement de la latence diminuant ainsi le débit global, à cause des cycles où le bus ne peut pas être utilisé. Cet effet va empirer de plus en plus au fur et à mesure que les technologies évoluent et que l'échelle d'intégration augmente.
Un bus partagé, ne possédant aucun concept de couches de communication, empêche un découplage efficace des deux parties de calcul et de communication, ce qui représente une entrave majeure au développement de systèmes hiérarchiques.
En résumant, on peut ramener toutes les limites d'un bus partagé à un problème de mise à l'échelle. Une solution assez répandue pour résoudre ce problème consiste à multiplier les bus dans les systèmes (voir figure 2.2). Ainsi, les concepteurs sont à même de découper le système en parties plus ou moins indépendantes formant une hiérarchie de bus ou un bus multi-couche. Ces bus peuvent être connectés entre eux par des ponts qui permettent de mieux gérer le débit requis et créer des liens entre les différentes parties. Une infinité de solutions est possible, ce qui favorise l'exploration d'architecture. Ceci permet de mieux choisir les arbitrages dans les divers bus grâce à une politique d'arbitrage qui devient de plus en plus distribuée par rapport au concept d'arbitre central d'un bus partagé. Beaucoup d'exemples de bus multi-couche sont déjà présents dans des produits commerciaux ; le STBus [67] de STMicroelectronics et l'architecture AMBA [2] développée par ARM en représentent une partie. Toutefois ces solutions doivent être conçues ad-hoc pour chaque système, ce qui rend la conception coûteuse et surtout non durable. De plus, comme le nombre des composants sur puce augmente de plus en plus, la solution multi-couche ne pourra pas résoudre indéfiniment le problème fondamental de mise à l'échelle.
Le concept de réseau sur puce s'est imposé comme solution possible au problème de l'interconnexion (voir figure 2.3). Il existe deux raisons principales qui le soutiennent. Premièrement, les réseaux sur puce aident à résoudre les problèmes électriques que les nouvelles technologies (submicroniques) imposent, parce qu'ils structurent d'une façon convenable les fils globaux (voir

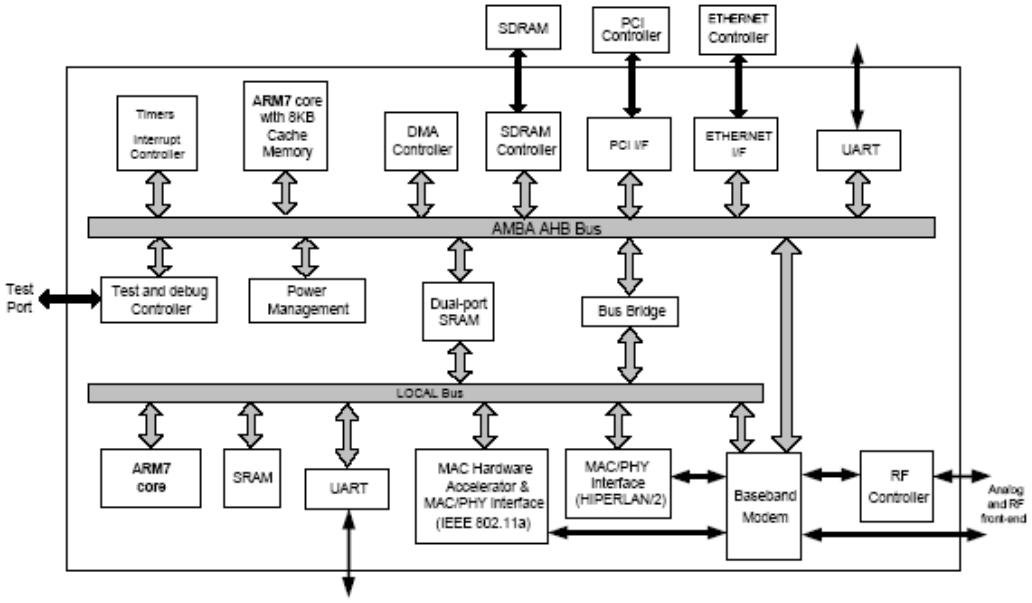

Fig. 2.2: Un exemple d'architecture de bus multi-couche

figure 2.4). En même temps ils permettent de les partager en réduisant leur nombre et en améliorant leur usage. Les réseaux sur puce peuvent être efficaces et fiables, et surtout ils facilitent la mise à l'échelle par rapport aux bus partagés et multi-couche. Deuxièmement, les réseaux sur puce aident à découpler les parties de calcul et de communication (NI - Network Interface, voir figure 2.4), ce qui est essentiel lorsqu'il faut gérer des systèmes contenant un grand nombre d'éléments. Les réseaux sur puce constituent la future technologie d'interconnexion pour les systèmes intégrés et permettent à la fois de fournir une interface complète et flexible aux couches logicielles et un niveau d'abstraction de l'interconnexion qui convienne au développement et à la conception au niveau système.
L'emploi de réseaux comme interconnexion pour systèmes sur puce implique un certain nombre de questions qui doivent être prises en compte. Contrairement aux interconnexions telles que bus ou fils point-à-point, où les modules communicants sont connectés directement, dans les réseaux ces mêmes modules communiquent par l'intermédiaire de nœuds. Ainsi, l'arbitrage d'un réseau change de centralisé à distribué et les problématiques telles que transactions désordonnées, latence élevée et contrôle de flux « end-to-end » doivent être prises en compte par les IPs ou le réseau même.

Il existe un grand nombre de propositions en littérature concernant les

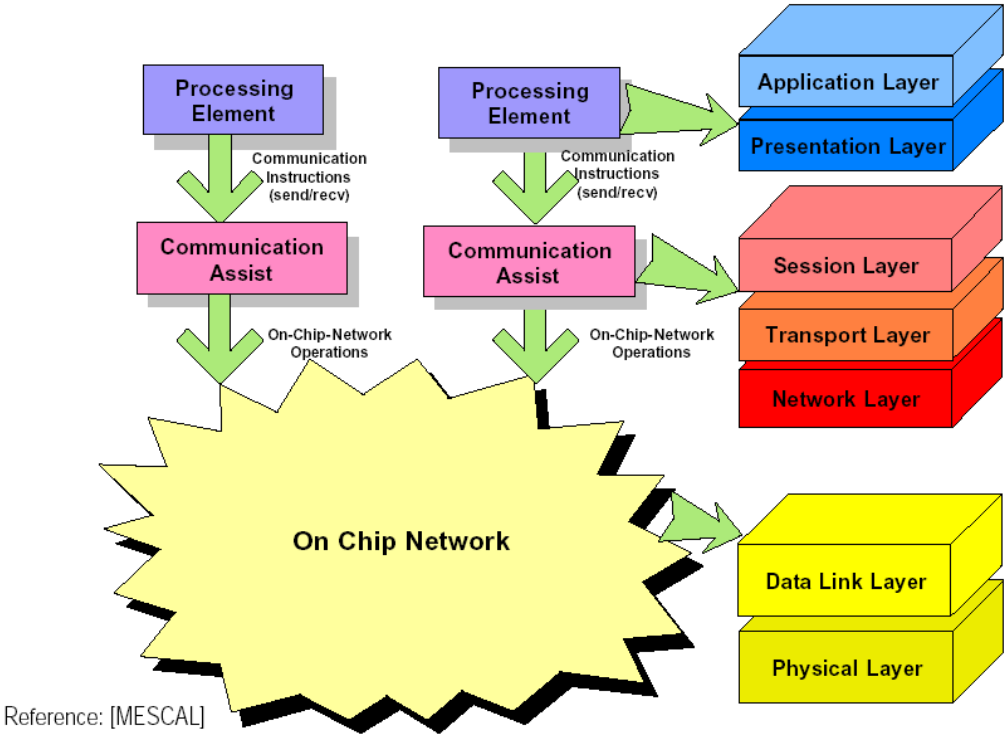

Fig. 2.3: Structuration de la communication pour les futures systèmes sur puce

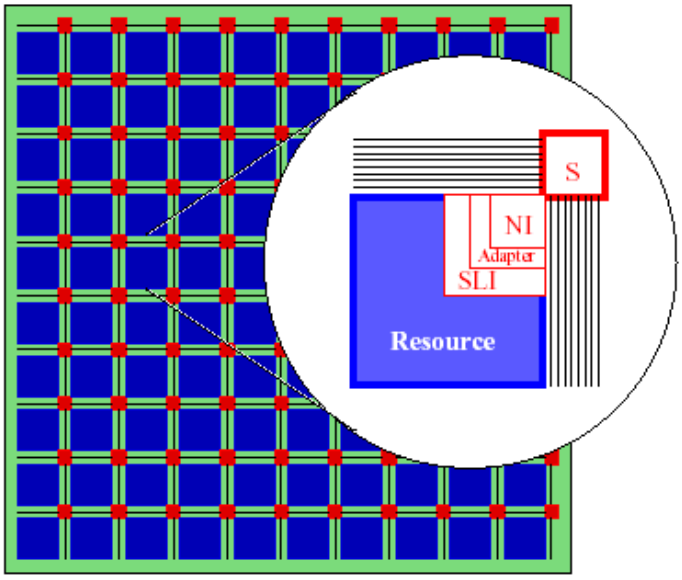

Fig. 2.4: Réseau vu comme un ensemble de commutateurs (S), interfaces (NI), et ressources. Le réseau permet une effective structuration de l'interconnexion.

réseaux sur puce, chacune avec ses atouts et ses points faibles. Il est tout à fait utile de les approfondir toutes, tout en sachant que, considérant les grands axes de recherche, deux points clés s'avèrent être fondamentaux et discriminatoires :

1. Topologie fixe ou paramétrable
2. Infrastructure mise en place pour garantir une certaine qualité de service du réseau

En ce qui concerne le premier point, une topologie fixe implique beaucoup d'avantages. La topologie étant fixe, les concepteurs peuvent mieux régler les paramètres des nœuds du réseau (mémoire tampon, canaux virtuels, algorithme de routage). Par contre, l'exploration d'architecture de l'interconnexion qui en découle s'en voit être limitée, car du moment que la topologie d'interconnexion a été fixée à priori, les concepteurs ont moins de degrés de liberté. Une topologie paramétrable possède des avantages et des points faibles spéculaires par rapport à une topologie fixe. Bien sûr, elle permet une exploration d'architecture de réseau optimale, aidant à adapter la meilleure topologie et le meilleur algorithme de routage à une application donnée. Par contre la possibilité d'avoir un réseau paramétrable implique un nombre élevé de degrés de liberté pour la conception des éléments fondamentaux du réseau (nœuds et interfaces), concept qui contraste bien souvent avec un réglage optimal et surtout à bas coût en terme de surface, détail très important pour la conception de circuits intégrés.
Le deuxième point représente une question cruciale dont la solution ne peut pas être définie incontestablement. La théorie basique des réseaux recommande deux approches pour garantir une certaine qualité de service aux IPs qui les utilisent :

Approche « Best-Effort » : Une application est structurée sur un ensemble d'IPs communiquant par un réseau avec une certaine topologie. L'application a des besoins tels que débit et latence. Le réseau est dimensionné de façon à pouvoir fournir le débit et la latence requis. L'atout principal de cette solution est la simplicité d'implémentation et le bas coût en termes de ressources occupées qui en découle pour le système. Le point faible réside dans le fait que cette solution peut causer un surdimensionnement du réseau et, comme pour un bus multicouche, doit être appliquée cas par cas selon les conditions requises par l'application ; Cette approche souffre donc d'un défaut de programmabilité.

Approche « Guaranteed services » : Le réseau de communication peut être considéré comme un fournisseur de services intégrés : les IPs requièrent certaines conditions de fonctionnement qui doivent être négociées avec le réseau. Le réseau en tant que fournisseur de service peut accepter ou refuser la connexion selon l'état actuel où le réseau se trouve. La transmission et la réception des données peuvent avoir lieu seulement après la mise en place d'une connexion entre les IPs conforme aux conditions requises. Évidemment cette approche possède un degré de programmabilité bien plus important que l'approche «best-effort» ; le système a donc la qualité d'être prévisible et flexible. Le principal point faible est dû au coût en surface qui découle d'une architecture de réseau capable de fournir des services intégrés[1], considérant les tenants et les aboutissants que cette caractéristique implique savoir. Laquelle de ces deux approches est la meilleure reste une question ouverte, ce seront les réalisations qui éliront la plus convenable pour chaque type d'application (étant donné que toute considération correcte devra sans doute être rapportée à chaque cas particulier).

L'évolution de ces deux approches représente la solution aux futurs systèmes de communication pour les systèmes monopuce. Dans le cadre de cette thèse, une nouvelle technologie d'interconnexion de systèmes monopuce ainsi que les outils pour sa modélisation et son implémentation ont été développés. STNoC™, le réseau sur puce développé par STMicroelectronics, est l'aboutissement d'un grand effort de conception, dont cette thèse constitue tout de même une partie importante.

[1] Les services garantis requièrent des composants matériels spécifiques dans les nœuds, par exemple des tables de routage ; une qualité de service construite sur un réseau « best-effort » ne requiert pas de composants supplémentaires, ce qui diminue les besoins en surface.

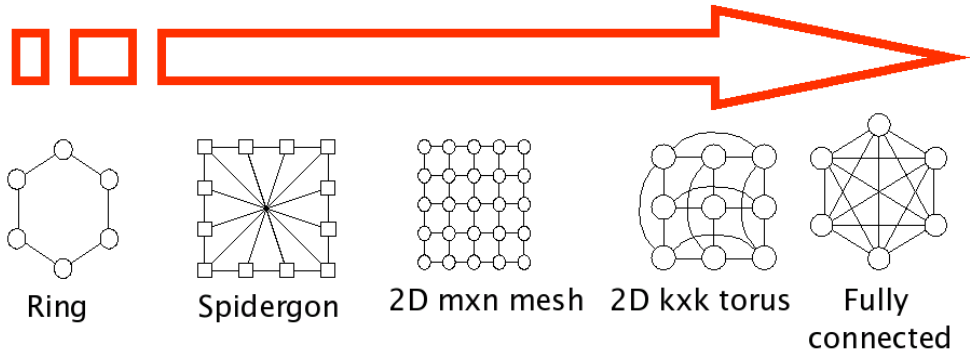

Fig. 2.5: Exemples de topologies

Le réseau sur puce STNoC™ est basé sur une topologie fixe surnommée « Spidergon » (voir figure 2.5), dont l'algorithme de routage qui détermine le chemin le plus court entre deux nœuds est le suivant (où N correspond au nombre de nœuds) :

if $(dest = curr)$ **then**
  $output \leftarrow NI$
else
  if $(|dest - curr| \leq \frac{N}{4})$ || $(|dest - curr| \geq (N - \frac{N}{4}))$ **then**
    if $(dest \in [curr + 1, .., curr + \frac{N}{4}])$ **then**
      $output \leftarrow RIGHT$
    else
      $output \leftarrow LEFT$
    else
      $output \leftarrow ACROSS$
    end if
  end if
end if

STNoC™ a choisi comme implémentation de services une modalité « best-effort », sur laquelle une infrastructure de qualité de service très puissante et flexible a été construite ; les détails de cette implémentation sont confidentiels STMicroelectronics et donc ne peuvent pas être dévoilés dans cette thèse. Un exemple d'application de cette technologie sera décrit dans la section 2.4 à la page 20. Les sections suivantes décrivent et résument le développement de l'environnement de simulation de STNoC™, un élément qui a fait preuve de maturité et d'efficacité en systèmes réels ciblant la télévision numérique à

haute définition, produits par STMicroelectronics.

2.3 Méthodologie de modélisation de réseaux sur puce

La conception des systèmes monopuce représente une tâche très complexe dont les différents stades deviennent de plus en plus découplés et indépendants. La définition d'un flot de conception qui fournit un niveau d'abstraction adéquat pour la conception au niveau système représente un élément fondamental pour la conception de circuits à complexité croissante.

Le projet On-Chip Communication network (OCCN) [17], développé sous « sourceforge », fournit un environnement efficace et performant pour la spécification, modélisation et simulation de réseaux sur puce basé sur une méthodologie orientée objet conçue sur le noyau SystemC. OCCN augmente la productivité du développement d'adaptateurs de communication grâce à une interface de programmation (API) universelle [17]. Cette interface de communication fournit une nouvelle méthode de modélisation qui prône la création et la réutilisation de modèles exécutables au niveau transactionnel entre différentes plate-formes de simulation. La méthodologie OCCN se focalise sur la modélisation de réseaux sur puce complexes en proposant une approche par couches pour la modélisation (voir figure 2.6). OCCN définit

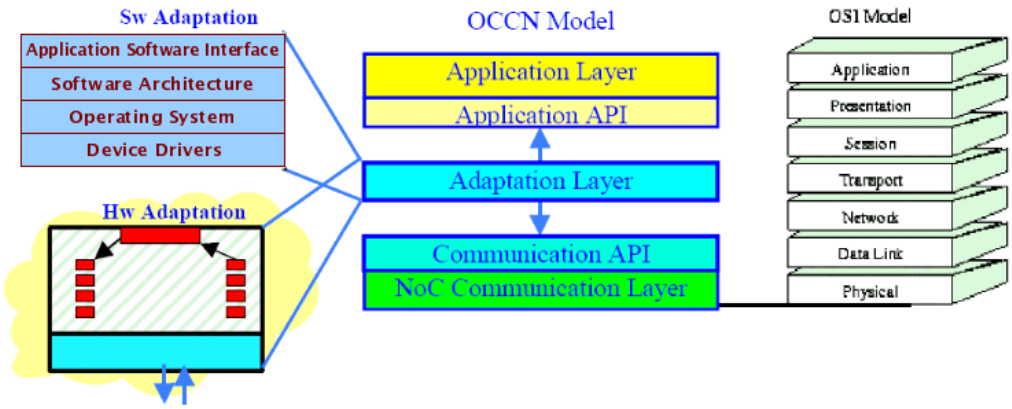

Fig. 2.6: Couches de modélisation du protocole OCCN

trois couches différentes. Le plus bas niveau implémenté par OCCN, intitulé *NoC Communication Layer* intègre une ou plusieurs couches OSI, en commençant par abstraire le niveau physique. Au plus haut niveau de la pile du

protocole d'OCCN, la couche applicative se calque sur la couche applicative du protocole OSI [63]. Positionnée entre la couche applicative et la couche de communication, la couche d'adaptation implémente une ou plusieurs couches du protocole, en incluant composants logiciels et matériels.
OCCN implémente la communication entre modules en utilisant l'approche générique SystemC [33], où un port peut être vu comme un point d'accès à un service définit par l'API d'OCCN. L'application du modèle conceptuel d'OCCN à SystemC est définie comme suit :

- La couche de communication est implémentée comme un ensemble de classes C++ dérivées de la classe `sc_channel`. Le canal de communication établit le transfert de messages entre les différents ports selon la pile du protocole supportée par un réseau spécifique.
- L'interface de communication (API) est implémentée comme une spécialisation de la classe `sc_port` de SystemC. Cette interface fournit les mémoires tampon requises pour la communication entre modules et la synchronisation.
- La couche d'adaptation convertit les transactions de requête entre modules engendrées par l'interface de l'application (API) en primitives appartenant à la couche de communication.

Les composants fondamentaux de l'API d'OCCN sont le « protocol data unit » (ou PDU, d'après la terminologie OSI) et les interfaces MasterPort et SlavePort. Dans OCCN la communication entre modules est basée sur des canaux qui implémentent des protocoles en définissant des règles et des types pour les PDUs. En général, les PDUs peuvent représenter des bits, des jetons, des messages d'un réseau d'ordinateurs, ou des signaux sur un réseau sur puce. Chaque PDU est composé de deux entités :

- L'en-tête intègre l'adresse de destination et il inclut l'adresse de la source. De plus, l'en-tête contient des codes d'opération qui servent à distinguer (a) requêtes / réponses (b) lectures/écritures (c) instructions de synchronisation (d) instructions bloquantes/non-bloquantes et (e) instructions de système.
- Le champ de donnée (appelé *payload*), ou *service data unit* est une séquence de bits qui n'ont pas de signification pour le canal.

Le paradigme utilisé par les interfaces de communication OCCN (objets *MsgBox*) afin d'envoyer et de recevoir des données est le passage de messages,

qui définit des primitives *send* et *receive* pour communiquer par l'envoi et la réception de PDUs.

Les modèles d'interconnexion de STNoC™ ont été développés en utilisant les briques de base fournies par l'environnement OCCN. La figure 2.7 définit le squelette d'un modèle de nœud STNoC™ en OCCN. La partie maître du

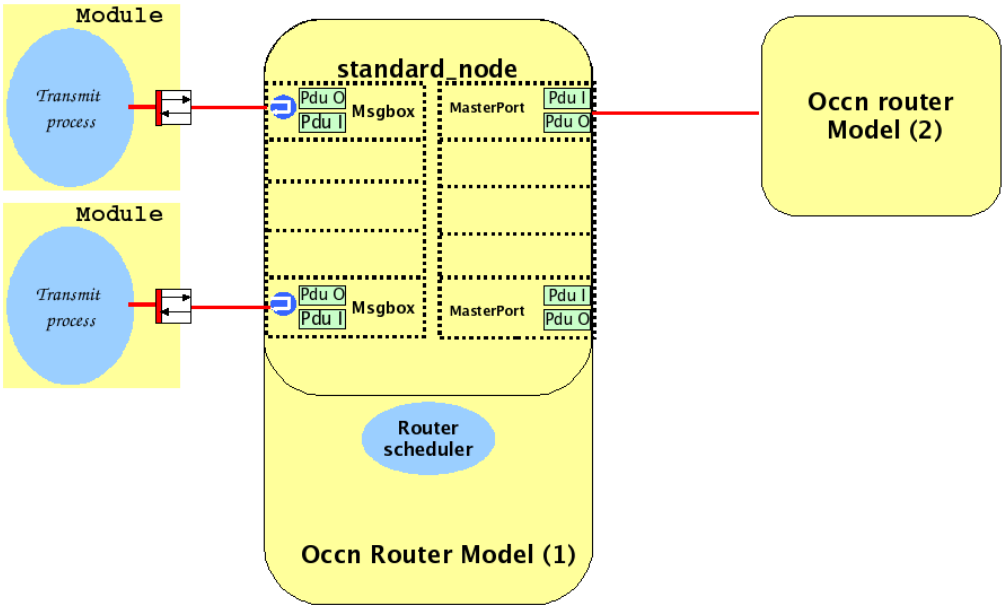

Fig. 2.7: Infrastructure d'un modèle de nœud

nœud est contrôlée par un objet OCCN *MasterPort* (un port par lien d'un maître et une *Msgbox* par lien d'un esclave), qui consiste en un port SystemC hiérarchique, construit pour s'adapter aux interfaces implémentées dans les objets *MsgBox*. En OCCN un nœud peut être vu comme une machine à états hiérarchique, où les étapes de pipeline sont décomposées en entrée, commutation et sortie. Chaque étape de pipeline est composée de détails microarchitecturaux.

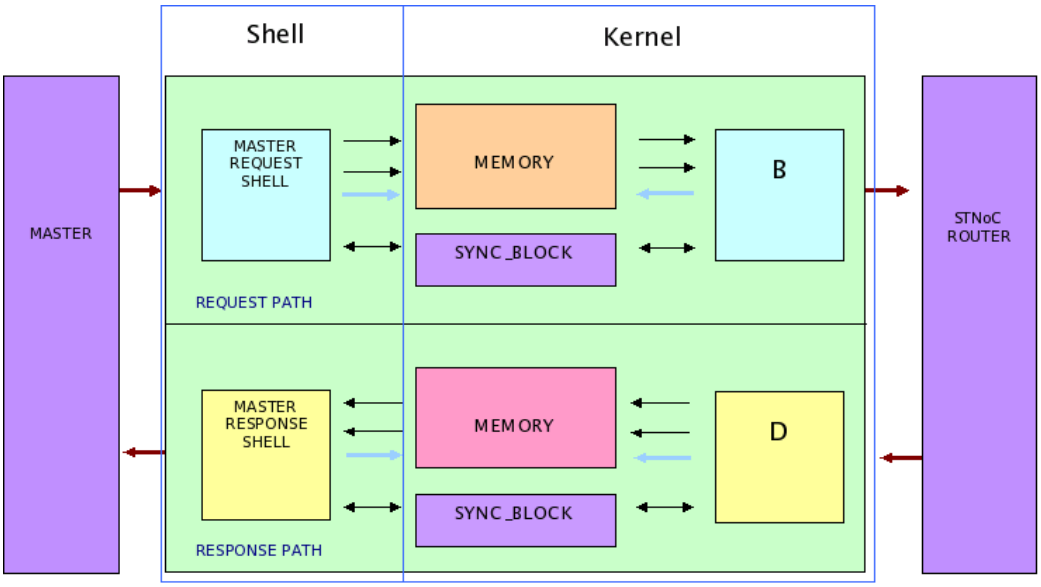

Fig. 2.8: Modèle de microarchitecture d'une interface réseau

Le modèle d'une interface réseau suit la configuration de la microarchitecture (voir figure 2.8), où le composant est divisé en deux parties principales :

- Noyau.

- « Shell ».

Le rôle du « shell » consiste à gérer la couche transport de l'interface réseau tandis que le noyau est chargé de contrôler le flux des données vers le nœuds et mettre en place la réalisation des paquets. La figure 2.9 définit un schéma basique en UML du modèle de l'interface réseau qui éclaircit les relations entre les différents objets utilisés. L'environnement de modélisation développé pour STNoC™ autour d'OCCN contient aussi des adaptateurs qui permettent la co-simulation de modèles transactionnels tels qu'OCCN avec des composants RTL décrits soit en SystemC soit en Verilog. Le développement de ces couches d'adaptation a été une partie fondamentale de ma thèse car il a permis de réutiliser et d'adapter le nouveau flot de conception aux éléments déjà existants au sein de STMicroelectronics.

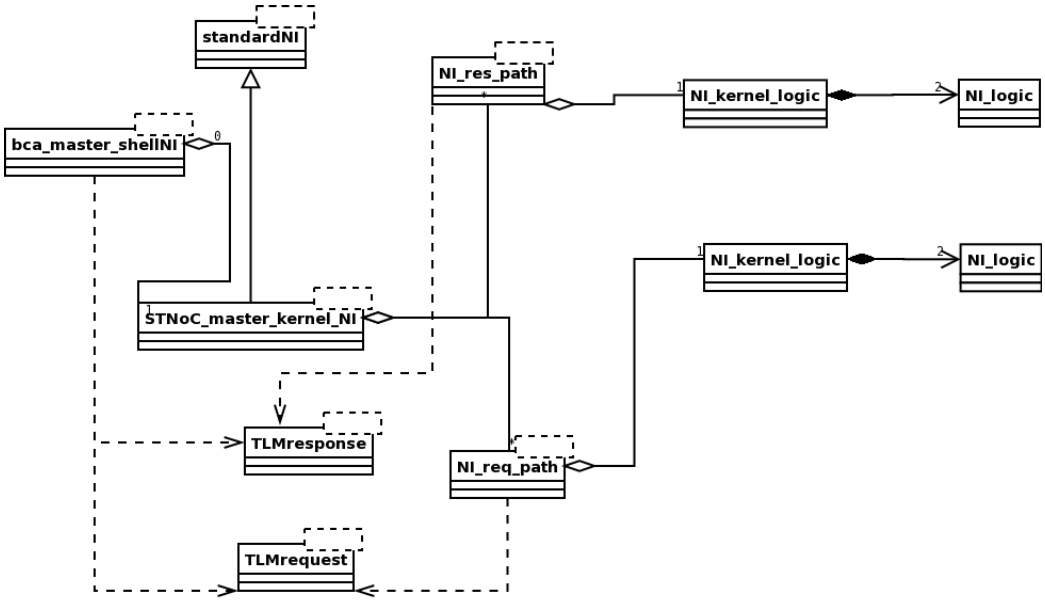

Fig. 2.9: Modèle UML d'une interface réseau

2.3.1 Simulations distribuées pour les systèmes monopuce

Une partie des travaux concernant la modélisation a été consacrée à la réécriture du noyau SystemC afin de permettre des simulations distribuées de systèmes monopuce. La solution mise en place est basée sur le concept de *helper threads*, à savoir des tâches d'exécution qui permettent la mise en parallèle et la concurrence lors de l'exécution de simulations en SystemC. Une première partie du travail a été nécessaire pour définir avec quelle granularité la mise en parallèle du noyau SystemC était possible.
Une fois convenu que la partie de code pouvant être mise en parallèle correspond au *delta cycle* [33], le développement du code a été assez facile. En effet, un *delta cycle* correspond au plus petit grain de synchronisation possible entre « processes » SystemC. Il restait tout de même des problèmes non négligeables concernant la protection des *sections critiques*, zones de code partagées en mémoire qui doivent être protégées afin d'éviter une concurrence indéterministe (« race conditions ») entre tâches (voir l'exemple de code suivant, qui représente une section critique dans la gestion de « processes » SystemC).

```
inline void sc_runnable::push_back_method( sc_method_handle method_h )
{
    // assert( method_h->next_runnable() == 0 ); // Can't queue twice.
    method_h->set_next_runnable(SC_NO_METHODS);
#ifdef MP_ST
    pthread_spin_lock(&m_lock);
#endif
    m_methods_push_tail->set_next_runnable(method_h);
    m_methods_push_tail = method_h;
#ifdef MP_ST
    pthread_spin_unlock(&m_lock);
#endif
}
```

Section critique, accès aux listes chaînées

Dans le noyau SystemC il y a beaucoup de sections critiques, en particulier en ce qui concerne la gestion des méthodes et « processes ». La solution au problème des sections critiques a été une des étapes les plus compliquées dans le cadre de cette thèse.
Des améliorations sont à l'étude pour parachever le nouveau noyau de simulation SystemC. Les premiers résultats de simulation ont démontré l'efficacité du nouveau noyau pour des modèles contenant un nombre moyen (10 - 20) de nœuds. Les simulations ont été exécutées sur une machine Intel Xeon (biprocesseur), Linux SMP 2.4, fonctionnant à la fréquence de 2.8 Ghz.

2.4 Exploration d'architecture du réseau ST-NoC™

L'environnement de simulation développé dans le courant de cette thèse a été utilisé à plusieurs reprises pour l'exploration d'architecture d'interconnexions de systèmes monopuce réels de STMicroelectronics. Afin de protéger les propriétés intellectuelles de STMicroelectronics, il est impossible de décrire les études menées desdits systèmes. C'est pourquoi, dans le cadre de cette thèse, deux types de tests ont été créés expressément afin de démontrer à la fois les propriétés de STNoC™ et du simulateur. La figure 2.10 décrit le flot de conception de STNoC™ en détail. Le flot de conception fait usage de différents composants puisés dans des librairies de modèles propres à STMicroelectronics (par exemple le Genkit pour le STBus et les IP Traffic Generator pour la génération de trafic). Grâce à une forte reconfigurabilité cet instrument permet de tester et de simuler un grand nombre de configurations de réseau en un temps restreint. La configuration est faite par des fichiers XML dont la grammaire permet une analyse facile et standard pour la création et la connexion de composants.

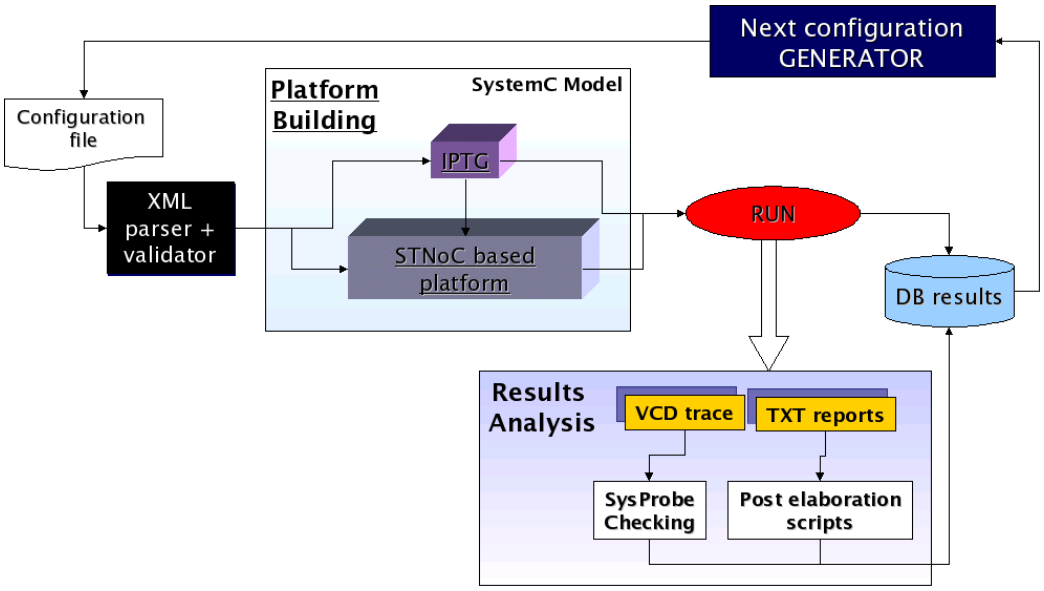

Fig. 2.10: Environnement d'exploration d'architecture pour STNoC™

Les résultats qui suivent sont décrits dans le seul et unique but de démontrer comment on peut caractériser un vrai réseau sur puce d'un point de vue du flot de conception. Ces résultats partiels ne sont pas optimisés et ils ne correspondent surtout pas à la caractérisation officielle du réseau STNoC™. La caractérisation officielle de STNoC™ est disponible comme document interne et confidentiel de STMicroelectronics et ne peut pas être dévoilée dans le cadre de cette thèse pour des raisons de confidentialité.

Un premier test a été écrit pour caractériser les propriétés de STNoC™ à travers un trafic purement aléatoire. La configuration choisie est celle d'un réseau STNoC™ à huit nœuds où tous les nœuds envoient et reçoivent des paquets de tous les autres. Le graphe en figure 2.11 compare le débit offert au débit accepté par le réseau. L'axe X correspond au nombre de « flits » (FLow control digITS [21]) injectés par cycle d'horloge. L'axe Y correspond aux « flits » acceptés. La courbe se compose de deux parties :

- Linéaire.
- Saturation.

En zone linéaire le réseau est à même d'accepter le trafic injecté ; l'efficacité de l'algorithme de routage, du contrôle de flux et de la mise en mémoire

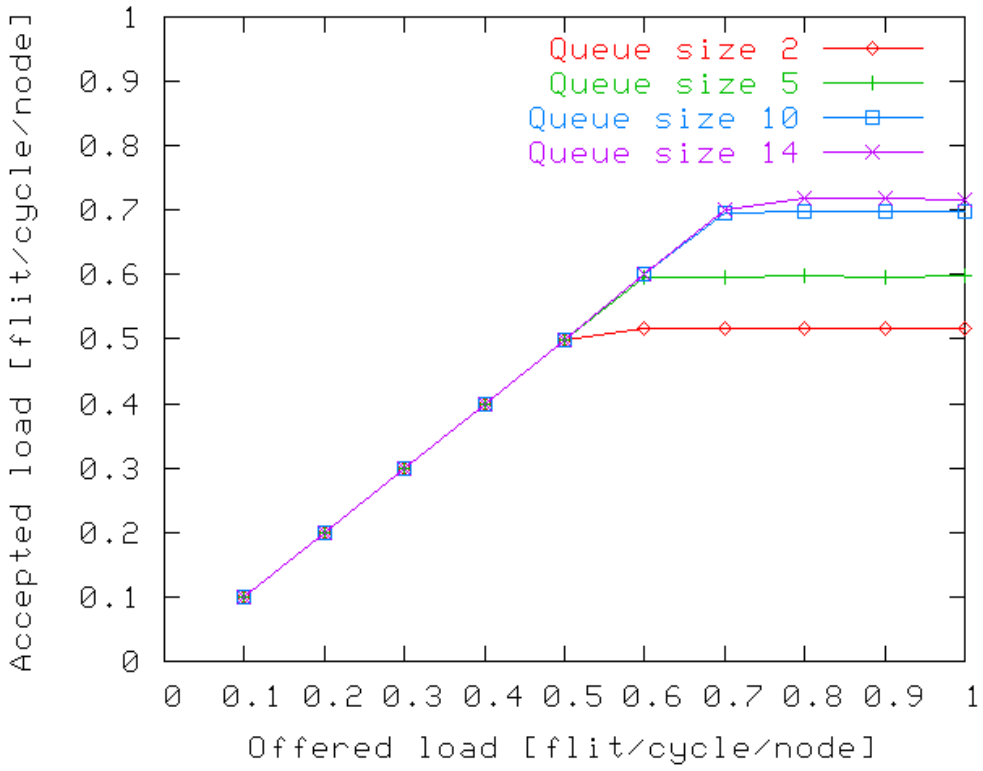

Fig. 2.11: Débit atteignable par un réseau STNoC™ $8x8$

tampon peut être mesurée grâce aux latences des paquets. Dans ce but, le graphe en figure 2.12 montre les latences des paquets sur le réseau.

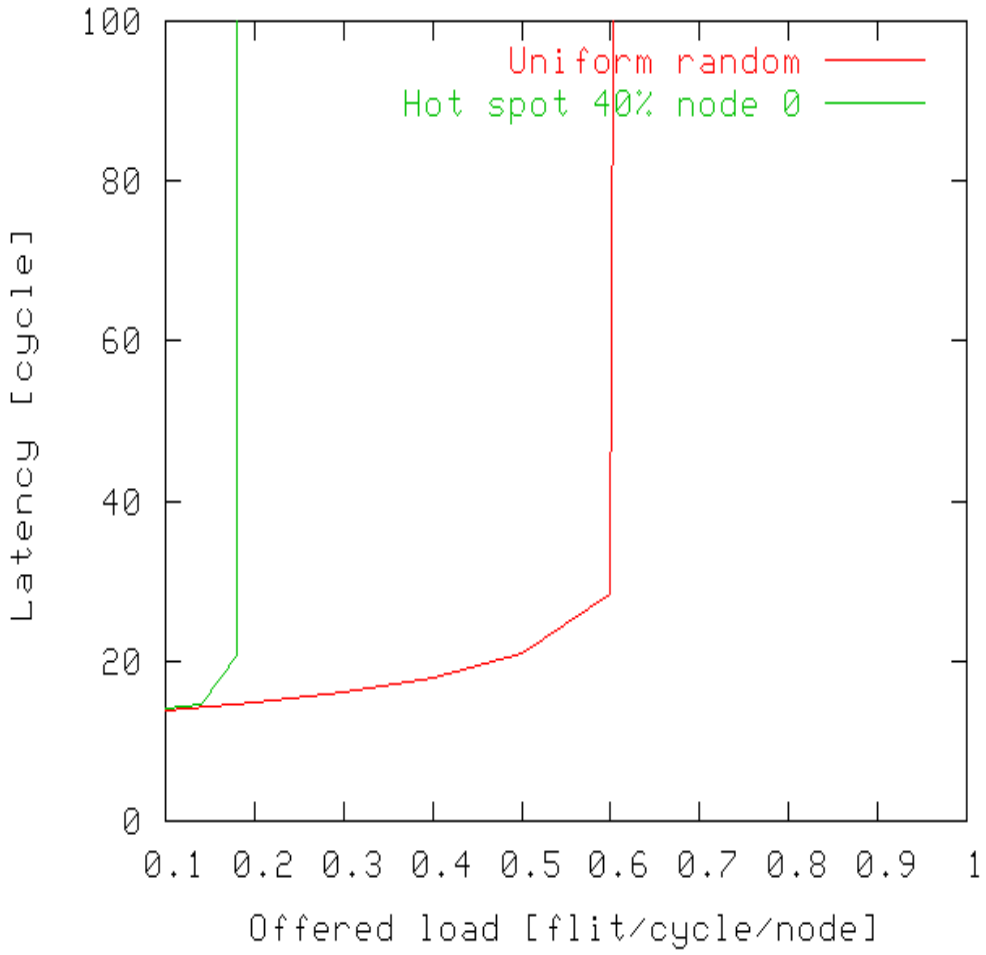

Fig. 2.12: Latence d'un réseau STNoC™ $8x8$, avec trafic « hot-spot » et aléatoire

Plus particulièrement, le trafic injecté est de deux types :

Trafic aléatoire qui consiste à envoyer des paquets uniformément distribués à tous le nœuds du réseau qui fonctionnent à la fois comme maîtres et esclaves.

Trafic « hot-spot » qui consiste à envoyer un pourcentage p du trafic au nœud « hot-spot » selon la formule :

$$P(destination == hotspot) = (p) * (N-1) + (1-p) * \frac{N-1}{N} \quad (2.1)$$

où N est le nombre de nœuds et p la probabilité selon une distribution de Bernoulli d'une destination de type « hot-spot ».

Lorsque 40% du trafic est dirigé vers une seule cible (« hot-spot »), le point de saturation décroît d'une façon très importante, ce qui implique une saturation à très bas seuil.

Des contraintes de confidentialité obligent à ne pas décrire les résultats, fondamentaux pour STNoC™, obtenus dans l'étude approfondie de deux systèmes réels produits par STMicroelectronics orientés vers la télévision numérique à très haute définition. Afin de remplacer ces expériences, un test simplifié a été mis en place, dont le schéma est montré en figure 2.13. Deux

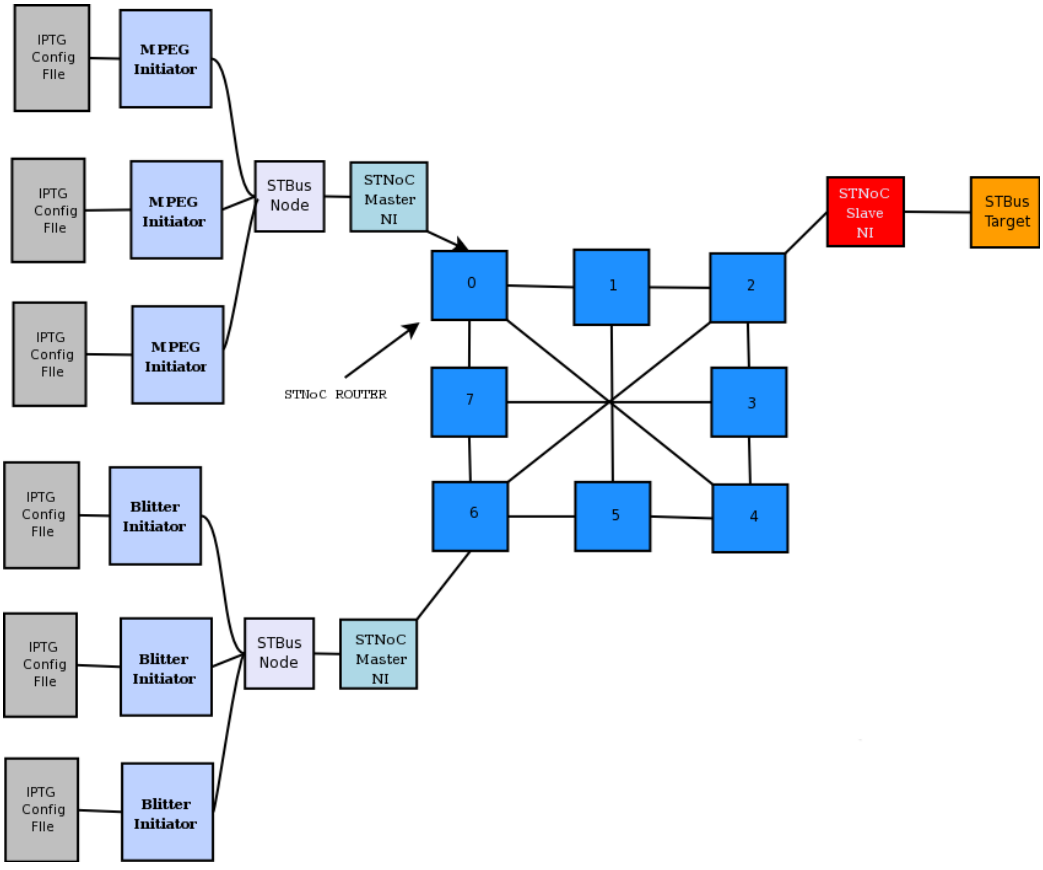

Fig. 2.13: Architecture de l'expérimentation menée sur STNoC™

sous-systèmes (MPEG et Blitter) ont été modélisés en utilisant les générateurs de trafic IPTG développés par STMicroelectronics. Une première mesure a été effectuée sur l'occupation des FIFOs du maître MPEG qui exécute des lectures.

En figure 2.14 une trace vcd de l'occupation des FIFO du maître MPEG en lecture est montrée, pour deux cas :

- Arbitrage « Least Recently Used » (LRU), qualité de service non intégrée.

- Qualité de service intégrée.

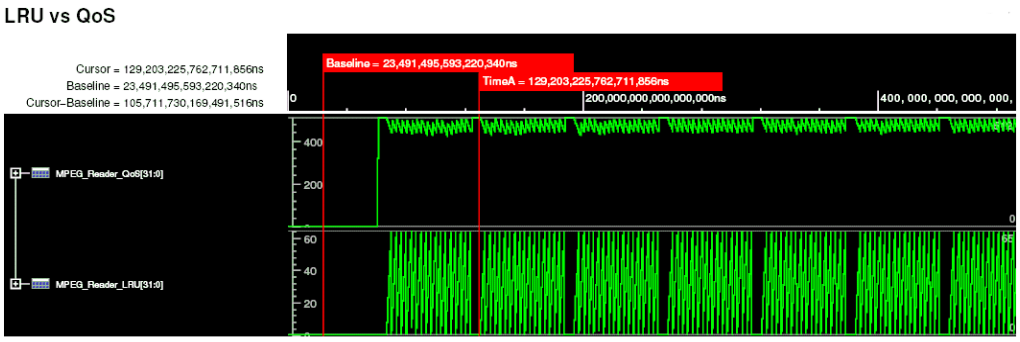

Fig. 2.14: Effet de l'arbitrage sur l'occupation des FIFOs des maîtres MPEG

Un but impératif dans la conception d'une interconnexion consiste à éviter une situation de « famine » d'une IP pour ce qui concerne les transactions à traiter. C'est le cas d'une FIFO dont l'occupation croise l'axe X (confronter figure 2.14). Si la FIFO est vide, ce qui arrive régulièrement avec un arbitrage LRU qui ne garantit pas un débit correct, l'IP ne peut avancer dans le traitement des données. Ceci s'avère être un problème grave, signifiant que l'interconnexion ne procure pas le débit nécessaire.
Il existe une réponse à ce problème ; cette réponse, implémentée en STNoC™, est dénommée qualité de service. Les détails d'implémentation ne peuvent être dévoilés pour des raisons de confidentialité. Par contre, la figure 2.14 montre très clairement que lorsque STNoC™ intègre des arbitres capables de garantir une qualité de service nécessaire, le comportement de la FIFO, et du coup de l'application, est bien meilleur. Non seulement l'occupation ne descend jamais à zero, mais elle reste à des valeurs toujours élevées, ce qui garantit un très bon fonctionnement en toutes conditions. En ce qui concerne la comparaison des latences sur le réseau, STNoC™ utilise un outil dénommé SysProbe. Les latences peuvent être mesurées de différentes façons (de la requête à l'acquittement, de la requête à la réponse), et les résultats sont disponibles en format texte.

```
STbus type : 3
Data size: 64 bits
Simulation Start Time: 0 ps
Simulation End Time: 1499998000 ps
Simulation duration: 1499998000 ps
Clock period: 4000 ps
STbus Frequency: 250.00 Mhz
STbus Clock cycles 374998
Window Time Frame 10
LATENCY :
      Name          Min     Max
      Req2R_Req      23      67
      Req2R_Eop      26      67
THROUGHPUT (MB/s):
      Data Throughput :            32.4 MB/s
      Full Data Throughput :       32.4 MB/s
BANDWIDTH :
      Max Available Bandwidth       2000.0 MB/s
      Max Available Bandwidth Req   2000.0 MB/s
      Max Available Bandwidth Resp  1600.00 MB/s
      Real_Bandwidth               32.4 MB/s
```

« Process » de lecture MPEG, arbitrage de type LRU

Ce fichier de texte produit par SysProbe montre les champs les plus intéressants des statistiques mesurées sur STNoC™. Des valeurs sont obtenues statiquement, d'autres dynamiquement. Dans cet exemple le débit demandé par le maître MPEG est d'environ 30 Moctets par seconde. Le débit disponible en réponse est limité par la sérialisation des transactions en « flits » (FLow control digITS).
Les « flits » de header (en-tête) ne sont pas comptés comme débit disponible parce qu'ils transportent des informations de contrôle. Les opérations sont toutes des lectures de 32 octets, sur un chemin de données de 64 bits, ce qui fait qu'un cinquième de la bande disponible est utilisé pour envoyer des contrôles (un paquet de réponse est composé de cinq « flits » : 1 de *header* + 4 de *payload* 4x8=32 octets). En intégrant le mécanisme de qualité de service, le réseau permet de limiter les latences (voir les statistiques qui suivent).

```
STbus type : 3
Data size: 64 bits
Simulation Start Time: 0 ps
Simulation End Time: 1499998000 ps
Simulation duration: 1499998000 ps
Clock period: 4000 ps
STbus Frequency: 250.00 Mhz
STbus Clock cycles 374998
Window Time Frame       10
LATENCY :
      Name          Min       Max
      Req2R_Req      23        42
      Req2R_Eop      24        42
THROUGHPUT (MB/s):
      Data Throughput :            32.4 MB/s
      Full Data Throughput :             32.4 MB/s
BANDWIDTH :
      Max Available Bandwidth       2000.0 MB/s
      Max Available Bandwidth Req       2000.0 MB/s
      Max Available Bandwidth Resp       1600.00 MB/s
      Real_Bandwidth                    32.4 MB/s
```

« Process » de lecture MPEG, qualité de service intégrée

La latence maximum est un paramètre fondamental pour les IPs, car les FIFOs internes sont dimensionnées en fonction de la latence maximum de l'interconnexion afin d'éviter une situation de « famine » pour le traitement des données. La qualité de service, en diminuant la latence maximum (de 67 à 42 cycles), améliore d'une façon très importante le comportement des IPs, y compris les besoins en surface.

2.5 Conclusion

Ce chapitre a permis d'introduire les principaux concepts et contributions de cette thèse. La partie restante du document (en anglais) détaille les travaux, en approfondissant tous les concepts mentionnés dans ce chapitre concernant la modélisation, la conception et l'étude de STNoC, la nouvelle technologie d'interconnexion développée au sein de STMicroelectronics.

Chapter 3

SoCs interconnections

3.1 Introduction

This chapter provides an in-depth outlook over existing interconnection architectures. First, networks-on-chip motivations are highlighted, with a clear focus on both technology and system level design issues. Secondly and finally, a state of the art section describes interconnection mediums deployed in current system-on-chip solutions, with a fleeting wink to first motivating networks on-chip examples just appearing in real designs.

3.2 Networks on-chip motivations

IC manufacturing technology will empower system designers with the capability to integrate a few billion transistors on a single chip within a few years. If these predictions are correct and the market will continue to demand ever higher volumes of ICs, the key question shifts on how the future chips will be designed. A few trends are radically complicating the architectures and the design of integrated circuits [40]:

Scalability concerns. Technology scaling works better for transistors than for wires (see figure 3.1). Hence, wires [76] dominate performance figure, power consumption and area, so that transistors constraints on design methodologies are relaxed. This implies a profound shift in system level design, the focus changes from number crunching and computation to data transport and communication.

Deep submicron effects (DSM). Deep submicron effects have been proposed as potential showsteppers to the continuing advancements in integrated circuit performance [69]. Examples of DSM include the rising resistance-capacitance (RC) delay of on-chip wiring, noise issues such as crosstalk and delay degradation [15], and increasing power dissipation. A digital or system designer with an expected design productivity of millions of transistors per day is not able to deal with these effects properly. This is the reason behind the tremendous need for IP reuse, designed by skilled experts. However, it is of the utmost importance that DSM effects do not show up again when reused blocks are combined. Consequently, blocks must be built with composability properties since the beginning of the design flow.

Clock distribution methods. One of the toughest problem in today's ASICs design, concerns the distribution of a skew-free synchronous clock over the whole chip. Many methods for distributing a clock have been treated thoroughly in the research literature over the last few years, from more obvious solutions, such as using asynchronous communication between locally clocked regions (Globally Asynchronous Locally Synchronous - GALS) to more elegant methods like distributing a standing wave on the clock-wires across the whole chip. Most of today's research is targeted towards reducing the clock-skew and jitter by enhancing current clock distribution methods together with new and better de-skew circuits and improved noise filtering in order to reduce the jitter. However, it is very unlikely that future chips will be synchronous designs whose clock is supplied by a single root clock tree.

Intellectual Properties (IPs) reuse. Synthesis technology development does not keep the pace with IC manufacturing technology development. As an aftermath, semiconductor companies are pushing the reuse of complex design units which represent the basic components of the design flow. These primitives made great strides from individual transistors to ALU till processor cores. Reuse is the main driving factor to boost productivity and is unlikely to change in the years to come.

Networks on-chip (NoCs) represent the best candidate to address the aforementioned issues [40]. NoCs leverage two main mechanisms to overcome the limits of current interconnection solutions: *reuse* and *predictability*.
Reuse represents the ultimate mean to bridge the technology gap. More and more complex components, from transistors to gates to functional blocks such as ALUs to microprocessors and DSP cores have become the primitive building blocks. In this way, the designer is empowered with a level of abstraction

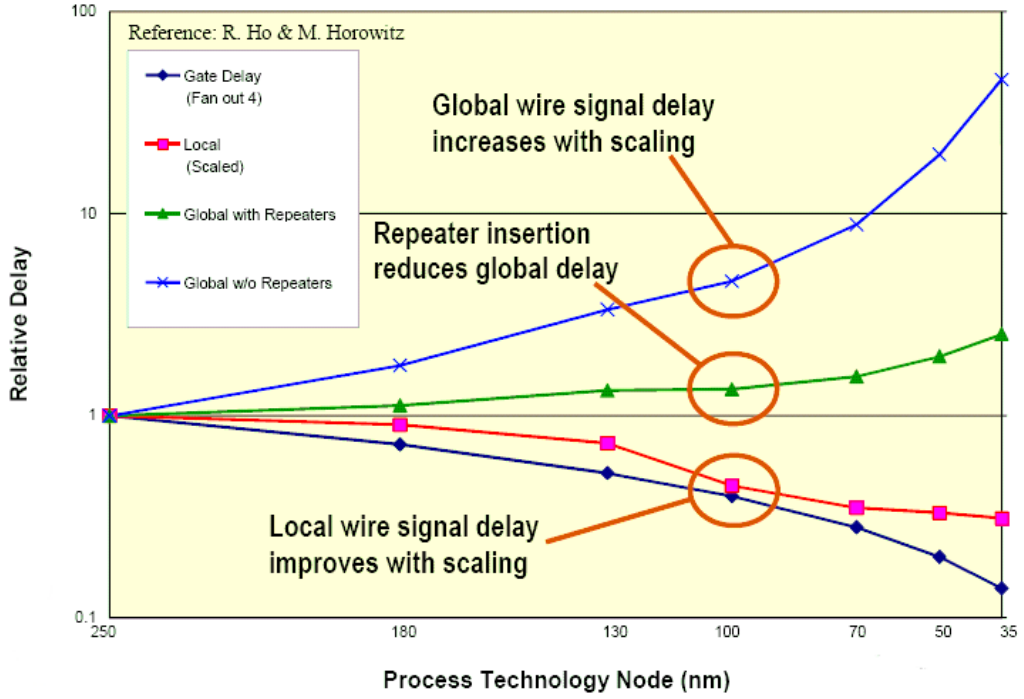

Figure 3.1: Local vs. Global wires delay

that allows to focus more on system functionality rather than low level design details. However, as a difference to the past, communication "components" also have to become primitive design elements. This is exactly where NoCs come into play.
The main immediate benefit of a NoC based approach is clearly due to the possibility to reuse the communication networks throughout different products. Furthermore, reuse is not limited to components. New products can be made up of existing complete systems. For instance, a traditional mobile phone can be enhanced by a video streaming subsystem and a digital still camera subsystem. The same modules must be reused again and again.
Obviously, defining a standard for high level interfaces between these features is a subject of the utmost importance to allow an efficient communication. NoC provides an excellent groundwork for this kind of features.
From a predictability perspective, the regularity of NoC layout provides well--characterized electrical and physical properties. The router-to-router wires, most likely the longest on the chip save clock, power and ground wires, have all exactly the same fixed length. Due to its regular geometry and communication network, the interconnection becomes much more predictable. With the sharing of communication medium among many resources, an active component can affect the available resources of other components.
Thus, the network has not only to guarantee bandwidth requirements but also access policies for bandwidth scheduling. A reliable communication resource allocation policy providing a predictable communication performance for all applications in the NoC is a crux for improving the composability of the design (see [45]). If a component can request, obtain and use communication bandwidth independently of all others in the NoC, modularity is improved because the basic components do not change throughout different designs.

From *users'* point of view, a certain behavior is expected of applications; in other words, they must guarantee certain predictable behaviors.
While those expectations may be low, as is often the case for personal computers, a certain robustness is always assumed. Consumer electronics is subject to higher demand: a television must provide a solid user interface; crashes or weird behaviors would be definitely frowned upon by consumers [31]. Real-time applications (e.g. involving audio and video, or control systems) demand even stricter requirements; a television must display at least 50 pictures of a constant quality per second; hence, the essence of Quality of service consists in providing a predictable system behavior to the user. To limit the exponential complexity of global methods and solutions, there is an increasing interest in subdividing global problems into *local*, *decoupled* problems [30]

and then composing the local solutions.
All the approaches that advocate local solutions have a common reliance on a scalable and compositional communication medium to efficiently combine the large number of (hardware and software) IPs or subsystems in a working system.

Predictability and reuse are the challenges addressed by NoCs, which therefore play a pivotal role in future SoCs. The interconnection becomes the real added value; to keep abreast of current technologies, systems on-chip need flexible, predictable and *scalable* solutions for on-chip communication.
In the remainder of this section, a state of the art of current interconnection technologies is provided, starting from typical bus based solution to more sophisticated switching networks.

3.3 State of the art

Buses have successfully been implemented in virtually all complex System On-Chip Silicon designs. Buses have typically been handcrafted around either a specific set of features relevant to a narrow target market, or support for a specific processor.
Several motivations (see section 3.2) forced evolutions of systems architectures, in turn driving evolutions of required buses. Buses have made great strides since the shared buses era. New features include split and retry techniques, pipelining and various attempts to define standard communication sockets.
In the remainder of this section a state of the art is unfolded trying to sum up the main interconnection architectures for systems on-chip.

3.3.1 Shared multi-layer buses and crossbars

In this subsection several bus architectures are reported with a key focus on advanced features and enhancements brought about by technologies improvements and evolutions.

3.3.1.1 AMBA Bus

The *Advanced Microcontroller Bus Architecure (AMBA)* specification is by now a renowned on-chip bus architecture. It has been deployed in a number

of chips coming from e.g. STMicroelectronics, for a variety of applications. Three distinct busses are defined within the AMBA specification:

- the *Advanced High performance Bus* (AHB)
- the *Advanced System Bus* (ASB)
- the *Advanced Peripheral Bus* (APB)

The most interesting component of AMBA bus is the AHB bus, hence it is worth a brief description.
Within AMBA architecture AHB acts as the high-performance backbone system bus. AHB supports an efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power macrocell functions. In figure 3.2 an hypothetical bus cut is shown, including an AHB bus as system bus. The backbone AHB bus is able to sustain the external

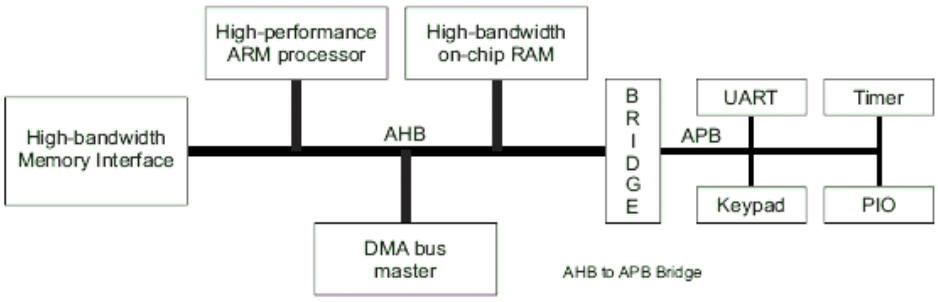

Figure 3.2: An example of AMBA bus Architecture

memory bandwidth, on which the CPU and other *Direct Memory Access (DMA)* devices reside, plus a bridge to a narrower APB bus on which the lower bandwidth peripheral devices are located (UART, PIO).
The AHB architecture integrates the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single cycle bus master handover
- single clock edge operation

- non-tristate implementation

The AHB bus protocol is designed to be used with a central multiplexor interconnection scheme (see figure 3.3). Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder selects the appropriate signals from the slave that is involved in the transfer.

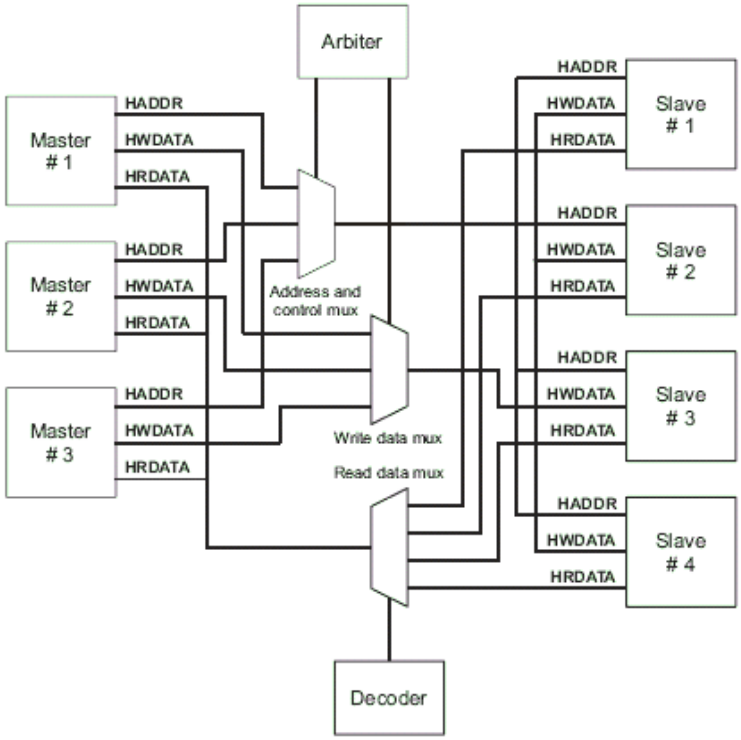

Figure 3.3: AHB bus microarchitecture

3.3.1.2 IBM™ Core Connect

The IBM Core Connect architecture provides three buses for interconnecting cores, library macros and custom logic:

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)

- Device Control Register(DCR) Bus

The PLB and OPB buses provide the primary means of data flow among macro elements. Because these two buses have different structures and control signals, individual macros are designed to interface to either the PLB or OPB.
Usually the PLB interconnects high-bandwidth devices such as processor cores, external memory interfaces and DMA controllers.

In particular, the PLB addresses the high performance, low latency and design flexibility issues needed in a highly integrated SOC through:

- Decoupled address, read data and write data buses with split transaction capability
- Concurrent read and write transfers yielding a maximum bus utilization of two data transfers per clock
- Address pipelining that reduces bus latency by overlapping a new write request with an ongoing write transfer and up to three read requests with an ongoing read transfer
- Ability to overlap the bus request/grant protocol with an ongoing transfer

Figure 3.4 illustrates the connection of multiple masters and slaves through the PLB macro. Each PLB master is attached to the PLB macro via separate address, read data and write data buses and a plurality of transfer qualifier signals. PLB slaves are attached to the PLB macro via shared, but decoupled, address, read data and write data buses along with transfer control and status signals for each data bus. Figure 3.5 illustrates how the CoreConnect architecture can be used to interconnect macros in a Power PC 440 based SoC.
High performance, high bandwidth blocks such as the PowerPC 440 CPU core, PCI-X bridge and PC133/DDR133 SDRAM Controller reside on the PLB, while the OPB hosts lower data rate peripherals. The daisy-chained DCR bus provides a relatively low-speed data path for passing configuration and status information between the PowerPC 440 CPU core and other on-chip macros.

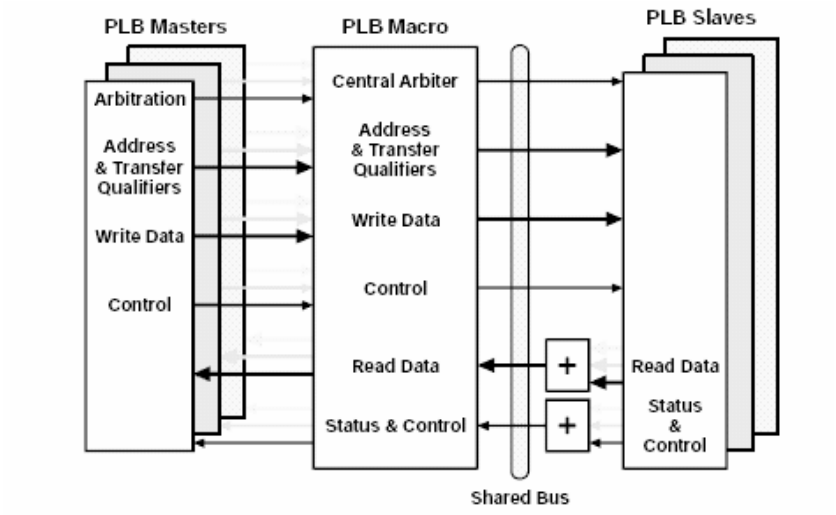

Figure 3.4: PLB interconnection

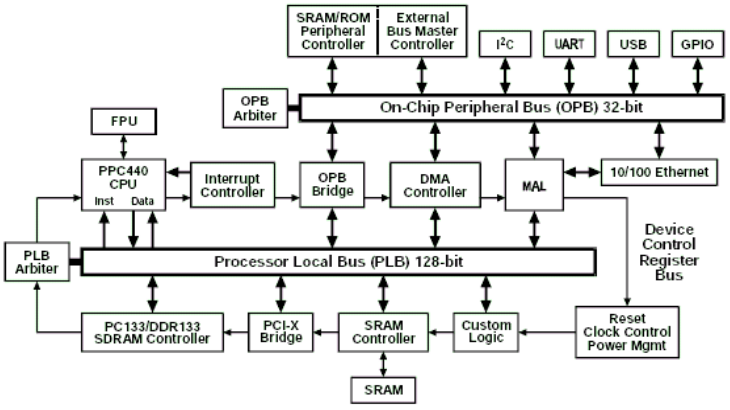

Figure 3.5: CoreConnect Based System-on-a-chip

3.3.1.3 STMicroelectronics STBus

The STBus [67] is a set of protocols, interfaces, primitives and architectures specifying an interconnection system, versatile in terms of performance, architecture and implementation. The STBus is the result of the evolution of the interconnection subsystem developed for microcontroller dedicated to consumer applications, such as set top boxes, ATM networks, digital still cameras and others.

Three different types of the STBus protocols exist, each having a different level of complexity in terms of both performance and implementation:

Type 1 is the simplest protocol, and is intended to be used for peripherals registers access. No pipeline is implemented in type 1 protocol. It acts as a RG protocol.

Type 2 adds pipelines features. It supports operation code for ordered transactions. The number of request cells mirrors the number of responses.

Type 3 is an advanced protocol implementing split transactions for high bandwidth requirements (high performance systems). It supports out of order executions. This interface is asymmetrical in that response size might differ from request size.

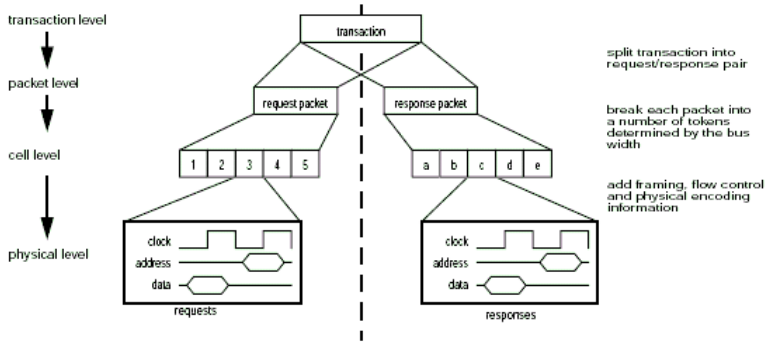

Figure 3.6: STBus protocol hierarchy

The STBus protocol can be seen as organized in layers, as shown in figure 3.6. Each operation is broken into one or more request/response pairs. Primitive operations have a single request/response pair. Depending on the STBus type, compound operations may contain multiple pairs of packets. These packets are then mapped to cell in physical interface.

Depending on the interface type, the amount of information to be transferred in the request phase is the same or may differ from response phase. This asymmetricity is pivotal for performance and bandwidth allocation.
To conclude this brief description of STBus architecture a list of building blocks is in order.

An STBus architecture is made up of:

- node
- registers
- size/type/frequency converters
- buffers

The node is responsible for the arbitration and the routing of the transactions. The arbitration is performed by one or more highly tunable arbiters. The type/size/frequency converters are self-explicative components to perform whatever conversion is required for mismatched interfaces.
A buffer is just a retiming stage usually designed to break critical paths for back-end synthesis. The register file is used to program the interconnect, including arbitration policies and dynamic priorities.

3.3.1.4 AMBA AXI

The *Advanced eXtensible Interface* AXI [3] is the latest generation AMBA interface. It is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed submicron interconnect. The key features of AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes
- burst-based transactions with only start address issued
- separate read and write data channels to enable low-cost *Direct memory Access*
- multiple outstanding accesses
- out-of-order transactions management

AXI protocol is burst-based. Every transaction has address and controls information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write channel to the slave or a read channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI has an additional write response channel to allow the slave to signal to the master the completion of write transaction.
One of the key feature of AXI protocol resides in its possibility to issue address information ahead of the actual data transfer and to enable support for multiple outstanding transactions as well as out-of-order completion of transactions. In figure 3.7 the architecture of read channels is reported.

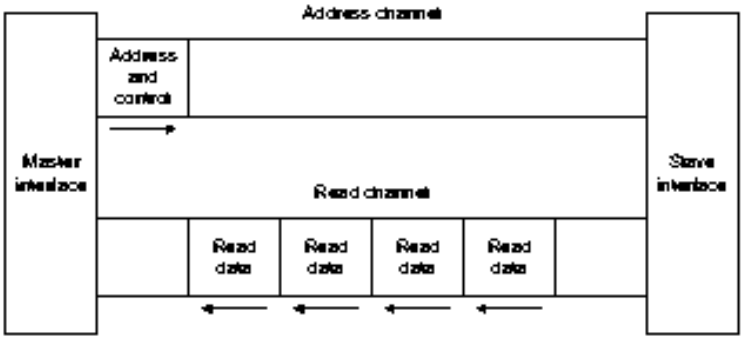

Figure 3.7: AXI read channel overview

The AXI protocol provides a single interface definition for describing interfaces:

- between a master and the interconnect
- between a slave and the interconnect
- between a master and a slave

This flexible interface enables a variety of different interconnect implementations. Most systems use one of three interconnect approaches:

- shared address and data buses
- shared address buses and multiple data buses

- multilayer, with multiple address and data buses

In most systems, the address channel bandwidth is significantly less than data channel bandwidth. Such systems can achieve good balance between system performance and interconnect complexity by using a shared address bus with multiple data buses to enable concurrent data transfer.

It is worth mentioning some advanced AXI features that represented a stepping stone in current buses architecture. In particular the AXI protocol sports:

Burst types The AXI supports three different burst types that are suitable for:

- normal memory access
- wrapping cache line bursts
- streaming data to peripheral FIFO locations

System cache support The cache-support signal of the AXI enables a master to provide to a system-level cache the bufferable, cacheable, and allocate attributes of a transaction.

Protection unit support To enable both privileged and secure access, the AXI provides three levels of protection unit support.

Atomic operations The AXI defines mechanisms for both exclusive and locked accesses.

The AXI architecture features also an additional low-power interface as an optional extension to the data transfer protocol that targets two different classes of peripherals:

- Peripherals that require a power down sequence, and that can have their clocks turned off only after they enter a low-power state.
- Peripherals that have no power-down sequence, and that can independently indicate when it is acceptable to turn off their clocks.

3.3.1.5 SONICS™ Silicon backplane

Silicon backplane III [66] is the first product of Sonics' SMART interconnect series whose goal consists in providing an innovative, highly configurable, scalable SOC inter-block communication (see figure 3.8) system that integrally

manages data, control, debug and test flows. SMART interconnect-based design addresses SOC integration both at the IP core level and at the system architecture level.
The interface to the IPs is based on the industry standard Open Core Protocol (OCP) (see [54]), which provides a standardized, configurable point-to-point core socket interface.
The SiliconBackplane interconnect is comprised of agents, each of which is mated to an IP core in order to decouple core functionality from inter-core communications [65], a crucial feature in nowadays systems. The block di-

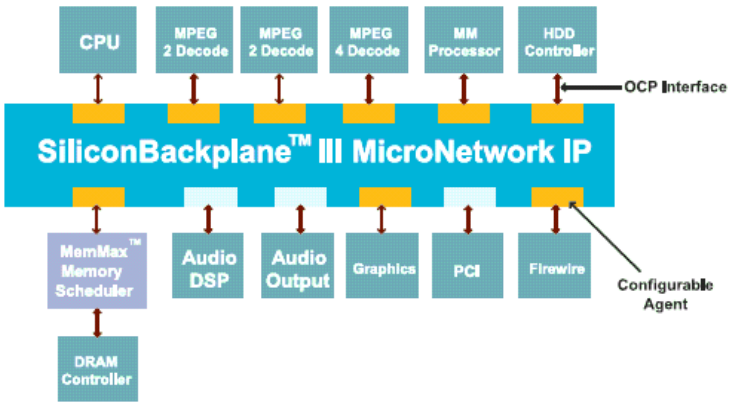

Figure 3.8: Example of Silicon backplane III integration in a multimedia SoC

agram in figure 3.9 shows the fundamental components of the silicon backplane. Each IP core communicates with an attached agent through standardized µNetwork interface ports using the Open Core Protocol (OCP). The agents communicate with each other using a network that implements the silicon backplane protocol. The agents provided by the Silicon-Backplane collectively manage all SOC communication, which occurs in the µnetwork. The focus of silicon backplane design flow is more on configurability than design itself. Socket design allows designer to concentrate on OCP configuration on a per IP basis. Through the Fast Forward Development Environment siliconbackplane empowers the designer with a tool to configure agents that can match configuration requirements of the cores and the communication requirements of the application.
The real added value that silicon backplane brought about is the possibility of greater reuse of SoC larger portions, in that it enables and fosters the development of chips as hierarchical ”tiles”.
A tile is a collection of function requiring minimal assistance from the rest

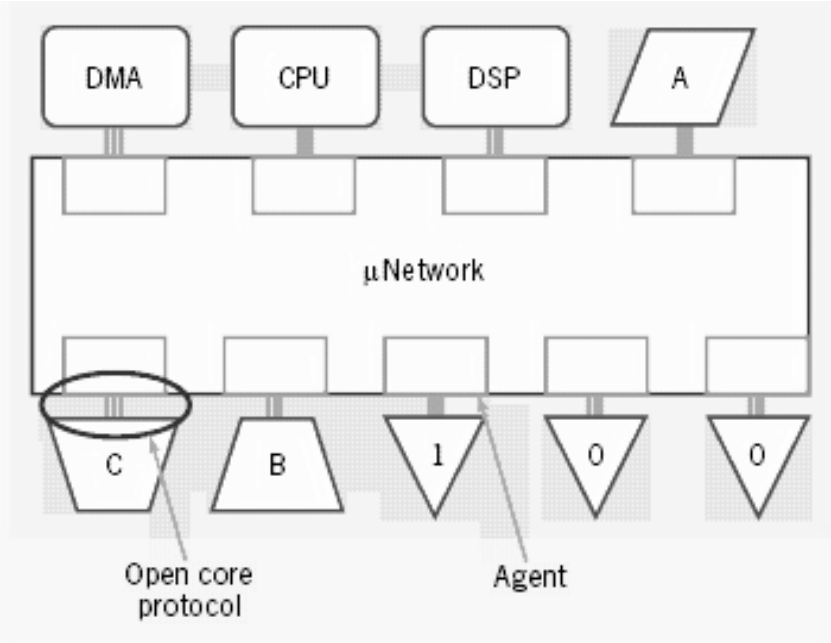

Figure 3.9: IP blocks integration in a network architecture

of the die and frequently includes an embedded processing unit, local memory, and relevant I/O resources. Through agents and reconfigurability, tile based architecture facilitates reuse across a SoC product family improving time-to-market.

3.3.2 On-chip switching networks

Innovative NoC architectures include the LIP6 SPIN, the MIT Raw, Philips' Æthereal NoC and Arteris NoC. In this section additional insights will be provided about these interesting technologies for on-chip communication based on switching networks.

3.3.2.1 LIP6 SPIN

The *Scalable Programmable Integrated Network* (SPIN) developed at LIP6 laboratory of "Pierre and Marie Curie" University in Paris represents a landmark in networks on-chip literature [34], and constitutes one of the first concrete implementation of switching networks brought on silicon [5]. The SPIN project sports an adaptive and distributed routing strategy with wormhole

flow control.
The SPIN point-to-point, full-duplex physical links are 36 bits wide in each direction and use a credit-based flow control. A SPIN network consists of three VLSI macrocells, a router and two wrappers Virtual Component Interfacing (VCI) standard compliant.
The RSPIN router (see figure 3.11) routes packets to their final destination whilst the two wrappers (i.e.VCI/SPIN SPIN/VCI) represent the glue useful for interfacing the SPIN network with the subscribers (e.g. processors, memories, DSP, ...).
For layout reasons SPIN network is organized as a Fat-Tree topology (see figure 3.10); a Fat-Tree is a tree structure with routers on the nodes and terminals on the leaves, except that every node has replicated fathers. The size of this network grows with a factor $\frac{(n)\log(n)}{8}$ with n equal to the number of terminals.

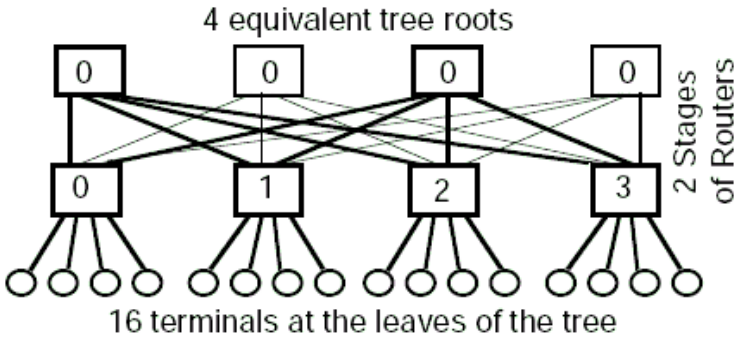

Figure 3.10: SPIN topology:A Fat-Tree Network

The SPIN router macrocell (see figure 3.11) was endowed with small (4 words) input buffers designed to hide link latency and control logic delay.
In case of output contention the SPIN router contains two output buffers of 18 words each. This solution is cleverer and more hardware efficient than choosing longer input buffers, because case studies showed that in average just two inputs are subject to contention. The SPIN network proved the feasibility of networks on-chip as interconnection medium for embedded systems.
Interested readers are encouraged to peruse [35] where additional insights about SPIN design and testbenches can be found.

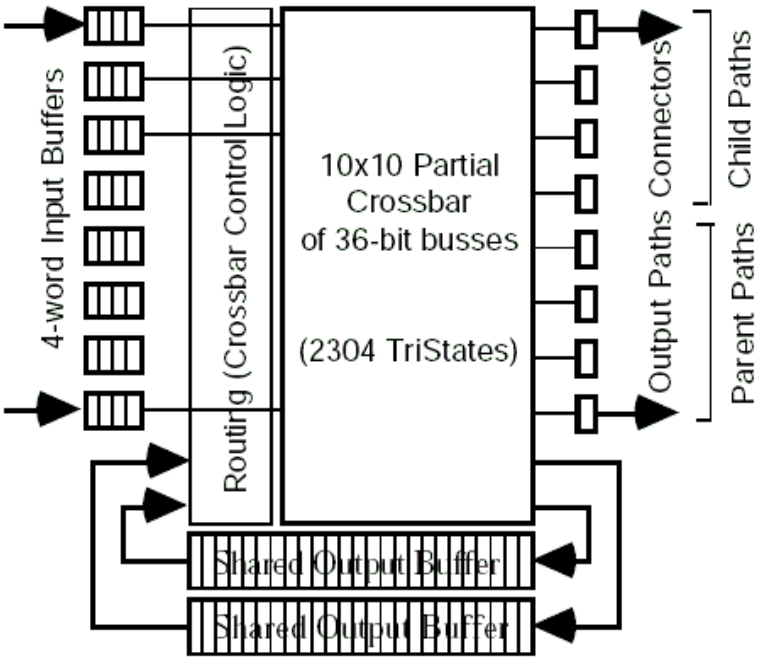

Figure 3.11: SPIN router macrocell microarchitecture

3.3.2.2 Philips' Æthereal Network on-chip

The Æthereal Network on-chip is a full-fledged on-chip interconnection network developed by Philips. It offers guaranteed services to obtain advantages in composability and robustness of QoS-based design. These services are mainly used for real-time and critical functions. The Æthereal NoC also provides best-effort services, to take advantage of their lower resource requirements and potentially better average performance. The Æthereal network services are efficiently implemented through a mix of time-division-multiplexed circuit switching and packet switching. To give time-related guarantees on a connection, such as throughput guarantees (on a finite time scale) or latency bounds, the interference of other traffic in the NoC in Æthereal is limited and characterized.
The Æthereal NoC uses *contention-free routing*, which is based on a time--division-multiplexed circuit switching approach where one or more circuits are set up for a connection, which is assumed to be relatively long-lived.
Guaranteed throughput (GT) packets never use the same link at the same time, namely all contention is avoided. In Æthereal this can be achieved by controlling both the time GT packets enter the network, and their speed in

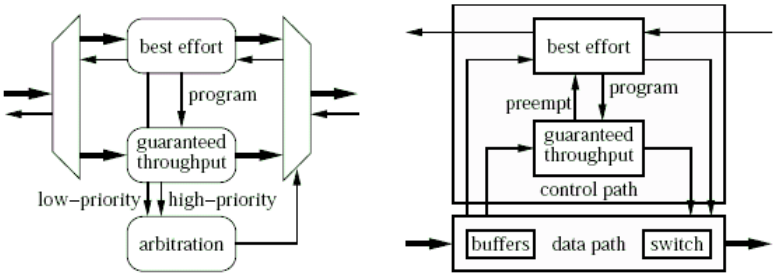

Figure 3.12: Overview of an Æthereal router

the network. All routers have internally a common notion of time, embedded in a slot counter. GT packets propagate at the rate of one router per slot counter increment.
The Best-Effort part of the Æthereal router (see figure 3.12) uses packet switching and has a more conventional structure, with different possibilities at different costs (wormhole routing, input vs. output buffering). The logically separated guaranteed (GT) and best-effort (BE) routers are combined (see figure 3.12) to share the router resources and to obtain the advantages of both. The GT offers a fixed end-to-end latency for its traffic, which has the highest priority enforced by an arbiter. The BE router uses all bandwidth (slots) that has not been reserved or is not used by the GT traffic. This allows the sharing of links and data path. The Æthereal network interface converts the OSI network layer of the routers to transport layer services for the IP. All end-to-end connection properties are implemented by network interfaces (e.g. reordering, transaction completion and flow control). IPs negotiate with network interfaces to obtain connections with certain properties. For this Æthereal network interfaces may reserve resources, such as network interface buffers and credit counters, and slots in router tables.

To sum up, an Æthereal router has been prototyped, with an area of $0.26mm^2$ (CMOS12) and offers a 80 Gbyte/sec as peak throughput. The Æthereal network interface has an area of $0.25mm^2$ in 0.13 μm, running at 500Mhz.

3.3.2.3 MIT Raw

The MIT Raw microprocessor[75] research prototype uses a scalable instruction set architecture to attack the emerging wire-delay problem by providing a parallel software interface to the gate, wire and pin resources of the chip.

The Raw processor design divides the usable silicon area into 16 identical, programmable tiles. Each tile contains:

- one static communication router;
- two dynamic communication routers;
- an eight-stage, in order, single issue, MIPS style processor;
- a four-stage, pipelined floating point unit;
- a 32-Kbyte data cache; and
- 96 Kbytes of software managed instruction cache.

Future raw processors will integrate hundreds of these tiles. The tiles interconnect using 32-bit full-duplex *on-chip networks*, consisting of over $12,500$ wires, as figure 3.13 shows. Two networks are static (routes specified at com-

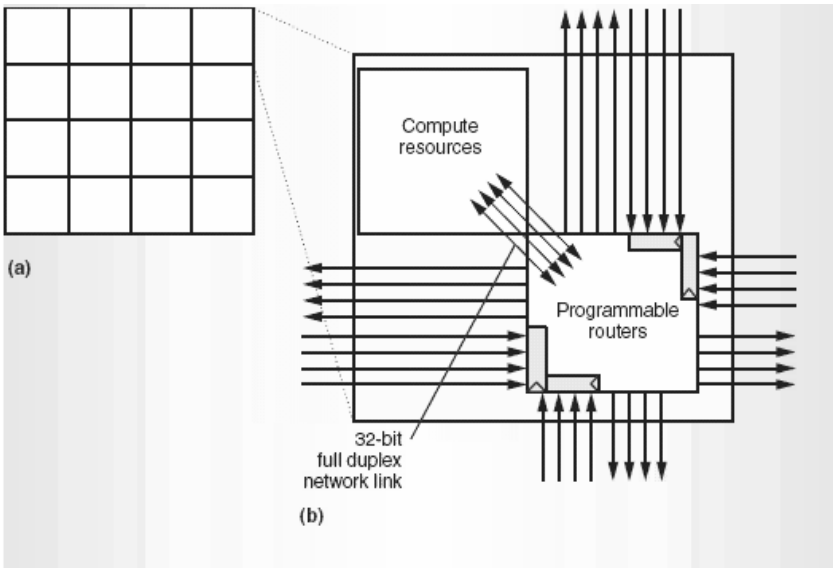

Figure 3.13: RAW on-chip interconnect a) RAW 16 tiles architecture b) Each tile has computational resources and four networks, each with eight point-to-point 32-bit buses

pile time) and two are dynamic (routes specified at runtime).
Each tile connects to its four neighbours, implying that the length of the longest wire is no greater than the length or width of a tile. This somehow

ensures *scalability* of the architecture.
The Raw ISA exposes these on-chip networks to the software, enabling the programmer or compiler to directly program the wiring resources of the processor and to orchestrate the transfer of data values between the computational portions of the tiles – much like routing in a full-custom application specific integrated circuit (ASIC). Effectively the wire delay manifests itself to the user as networks hops.
The static router is a five-stage pipeline that controls two routing crossbars and thus two physical networks. Each crossbar routes values between seven entities (see figure 3.14), – the static router pipeline; the north, east, south, and west neighbour tiles; and the other crossbar. The RAW microprocessor

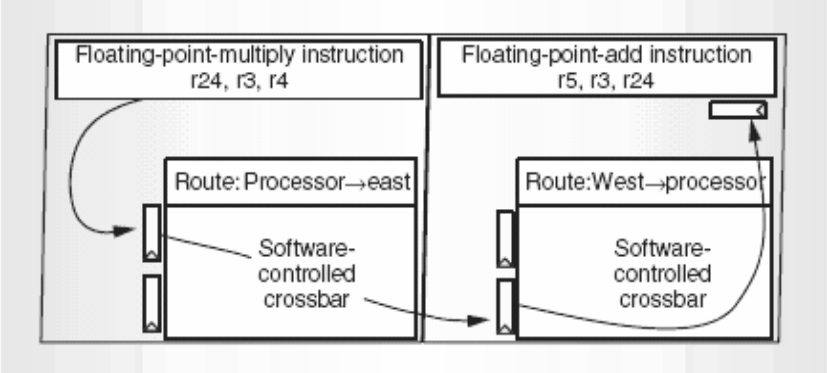

Figure 3.14: Two RAW tiles communicating over a static network

also sports a pair of dimension-ordered, wormhole-routed dynamic networks to the architecture. To send a message on one of these networks, the user injects a single header word that specifies the destination tile (or I/O port) a user field, and the length of the message.
The MIT Raw processor represents a stepping stone in advanced computing and it covers a subject of the utmost importance, namely the need to mitigate the considerable wire delays that looms on the horizon for nowadays systems (e.g. Pentium 4 architects had to allocate two pipeline stages solely for the traversal of long wires), through a scalable yet very impressive design made up of processing units connected through full-blown static and programmable on-chip interconnection networks.

3.3.2.4 Arteris NoC

The Arteris Network on-chip [8] splits the functionality of the interconnection into three layers (see figure 3.15). Using the transaction layer, the Network

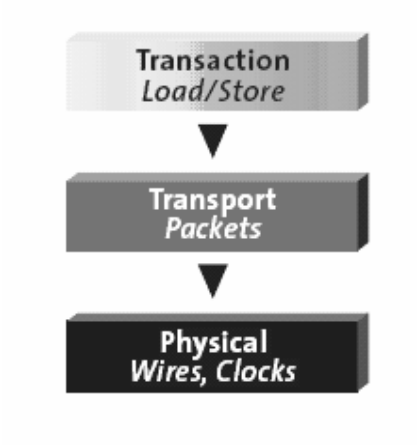

Figure 3.15: NoC layers in Arteris solution

interface units manage the communication with a connected IP core, providing the network services to that core.
The Network Interface Unit converts the conventional load/store transactions into the proper packets for transport across the network. On the other hand, transport layer deals only with packets, and their routing and switching across the network. Finally, physical layer defines how packets are physically transmitted between NoC units.
The Arteris network on-chip solution is available through an IP library called "Danube". The basic units comprise, Network interface units, packet transport units and physical links (see figure 3.16). The Arteris network on-chip IP has the following characteristics:

- Support for OCP-IP, AMBA AXI and AMBA AHB
- Globally asynchronous, locally synchronous links technology
- Claimed clock frequency up to 750 Mhz in 90 nm process
- Flexible pipelining and FIFO management
- Flexible topology

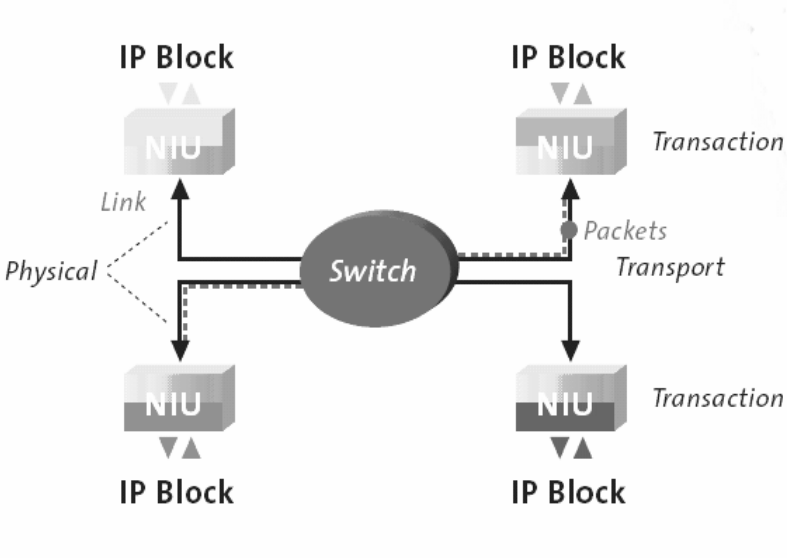

Figure 3.16: Arteris "Danube" design flow

In order to create a NoC instance, Arteris developed a tool named *NoCExplorer* to capture the dataflow requirements of the IP blocks to be serviced by the NoC.
Strictly speaking the Arteris NoC solution can be described as a pipelined crossbar architecture, with asynchronous solutions for on-chip links to tackle the hurdles provided by global wires poor performance.
The Arteris design flow allows a top down methodology to build on-chip interconnection with a tool suite useful to compile and generate SystemC as well as VHDL models of the aforementioned network on-chip.
The Arteris solution differs from other described solutions in that there is no notion of router as a basic building block; hence a network can also be treated as single crossbar instead of a composition of routers blocks connected and laid out in a more advanced distributed scheme. Fine-grain details of the interconnection are not provided with Arteris documentation so that additional details cannot be provided in this short overview.

3.4 Summary of existing interconnections

The different technologies presented in this chapter share some common characteristics that affect performance and cost. AMBA bus is the simplest (and

oldest) architecture; transactions are not pipelined, and request/response pairs are not split (request/response pairs are atomic; this means that just one outstanding request is allowed per master at a given time). AMBA is widely used but it is becoming obsolete as more sophisticated architectures are deployed. It is still useful in legacy subsystems that are not likely to be changed in new designs in order to re-use existing components.
IBM™ Core Connect, AMBA AXI and STBus are all high-performance, pipelined, split request/response busses, developed for high-end systems--on-chip interconnections. They share common features, such as multiple outstanding requests, multiple interfaces (for instance APB bus in AMBA, OPB in Core Connect and T1 interface in STBus are just used to connect peripherals with low-traffic requirements). An interesting feature that lacks in other buses and is implemented in AMBA AXI is exclusive access, namely the granting of a transaction to a given master just if the transaction turns out to be atomic from a slave perspective. This is useful to avoid locking the bus, that is how atomicity is guaranteed in Core Connect and STBus.
Silicon backplane shares some features of buses (multiplexing of control and data signals) but integrates also typical techniques used to guarantee services in networks such as Time Division Multiplexing (TDM). Detailed area figures are not available for all busses so that silicon area assessments are not possible at this stage.
Concerning switching networks SPIN and RAW interconnections are typical networks developed to build parallel embedded processors. SPIN fosters a fat-tree topology whilst the RAW micro-processor sports a more general interface where routers can be connected in a configurable topology. A comparison of topology is not possible due to lack of test-benches that could help gauge different design choices (for example flow-control techniques and related effectiveness). Æthereal network on-chip boasts both guaranteed services and best-effort networks. It is entirely programmable, even though silicon requirements in terms of area are not clear. An application of Æthereal to a real Philips system-on-chip would help assess performance and costs as a whole. In particular, in author's opinion, time-division-multiplexing in the guaranteed services network turns out to be a hard constraint on network design and effectiveness.
Arteris network on-chip represents the simplest form of switching network brought on-silicon. It would be more appropriate to describe it as a "serialized" bus rather than a network because the only network concept present in Arteris NoC is through the notion of packet. The network is split in two disjointed networks for requests and responses. Arteris network interfaces were developed to be compliant with many buses protocols, some of which mentioned in this chapter (e.g. AHB) . Arteris network effectiveness must

be verified in multi-processor design where data-path parallelism is of the utmost importance.

The design of STNoC™ drew the lessons derived from the aforementioned technologies in order to build a brand-new switching network with all nuts and bolts but with pragmatism and simplicity in mind as design tenets. Section 4.6 on page 77 reports STNoC™ architecture features.

3.5 Conclusion

All along this chapter, a detailed overview of interconnection technologies has been unfolded. Motivations behind the introduction of on-chip networks as main substitute for shared buses were described in details, focusing on different facets of system-on-chip design. In particular, technology issues such as global wire length and the increasing need for computation/communication decoupling were explained thoroughly.
A state of the art section summed up existing technologies for on-chip communication, in order to lay down a solid background very useful for the chapters to come. Both shared-buses and networks on-chip examples were reported, in order to point out strengths and weaknesses of the two design methodologies. The following chapter describes the development process of STNoC™ network on-chip, the brand-new interconnection medium developed by STMicroelectronics, starting first by describing the general network protocol stack fairly adapted to fit the on-chip world.

Chapter 4

Networks On-chip: A layered approach for On-chip communication

4.1 Introduction

As discussed in chapter 3, one of the major drawbacks of shared busses was and currently is its lack of modularity. Albeit efforts were made to improve computation and communication decoupling, interconnection media such as busses are not endowed with layering concepts, which causes a complete blurring of the communication layers. Networks on-chip, through the notion of packet, provide a suitable solution to the decoupling issues that are hindering system on-chip development. Furthermore, through arbitration, networks bring on-chip pivotal design features such as quality of service (QoS). This chapter describes the whole new concept of layering on which networks on-chip are based, as well as advanced arbitration schemes to achieve good quality of service properties that turn out to be so important in current system on-chip design methodologies. After a short introduction highlighting new frontiers in communication design, network layers adapted to on-chip world are analyzed step-by-step, with detailed insights on arbitration schemes and advanced protocol issues. A detailed description of STNoC™, the new inter-

connection technology developed by STMicroelectronics, ends the chapter, to describe the architecture internals and to provide the required background information essential to understand the STNoC™ modeling environment, the ultimate achievement of this thesis.

4.2 Networks on-chip: a micronetwork of components

Future generation Systems on-chip will be integrated as a micronetwork of components[9][20]. The network is the abstraction of the communication among components and must satisfy quality of service requirements such as – reliability, performance, and energy bounds – under the limitation of intrinsically unreliable signal transmission and significant communication delays on wires. A micronetwork stack paradigm – such as the one in figure 4.1 can

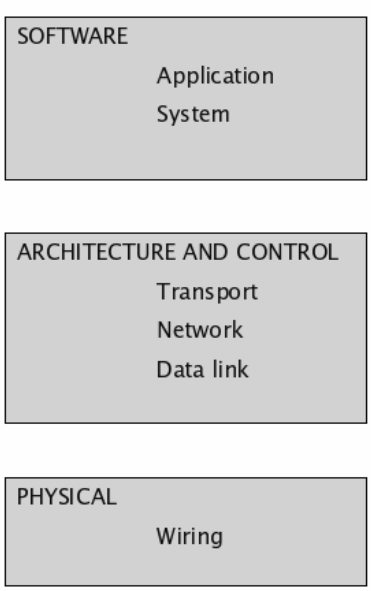

Figure 4.1: Adaptation to networks on-chip paradigm of OSI [63] stack

be used to abstract the electrical, logic, and functional properties [78] of the interconnection scheme. SoCs networks differ from wide area networks in their local proximity and because they exhibit less nondeterminism.
Local, high-performance networks – such as those for large-scale multiprocessors – have similar requirements and constraints. Some distinctive characte-

ristics, such as energy constraints and design-time specialization, are unique to SoC networks though. Whereas computation and storage energy greatly benefits from device scaling, which provides smaller gates and memory cells, the energy for global communication does not scale down.
On the contrary, as the "wiring delays" sidebar indicates, projections based on current delay optimization techniques for global wires show that global on-chip communication will require increasingly higher energy consumption. Thus, minimizing the energy used for communications will be a growing concern in forthcoming technologies. Furthermore, network traffic control and monitoring can help manage the power that networked computational resources consume.
Another facet of the SoC network design problem, design-time specialization, raises even tougher challenges. Communication network [24][59] design has traditionally been decoupled from specific end applications and is strongly influenced by standardization and compatibility constraints in legacy network infrastructures. In SoC networks, these constraints are less restrictive because developers design the communication network fabric from scratch. Hence, only the abstract network interface for the end nodes requires standardization.
Developers can tailor the network architectures to specific applications. These considerations lead to an envisionable vertical design flow in which every layer of the micronetwork stack is specialized and optimized for the target application.
From a design standpoint, network reconfigurability will be pivotal in providing plug-and-play component use because the components will interact with one another through reconfigurable protocols. In figure 4.2 the basic components of a network on-chip are highlighted with embedded layers information.
The transport layer defines the communication primitives available to IP blocks. Special components (network interfaces), located at NoC periphery [8] provide transport layer services to IP block with which they are paired.
The Network layer defines rules that describe how packets are routed through switches (routers) to reach final destination. Very little information contained within the packet is needed to actually transport the packet. A packet is very flexible, e.g. it can include byte enables or user information without altering network layer information (aka encapsulation).
The data link layer defines how packets flow through the network (link protocol, buffers management). Using micronetwork architectures effectively requires relying on protocols-network control algorithms that are often distributed. Network control dynamically manages network resources during system operation, striving to provide the required quality of service.

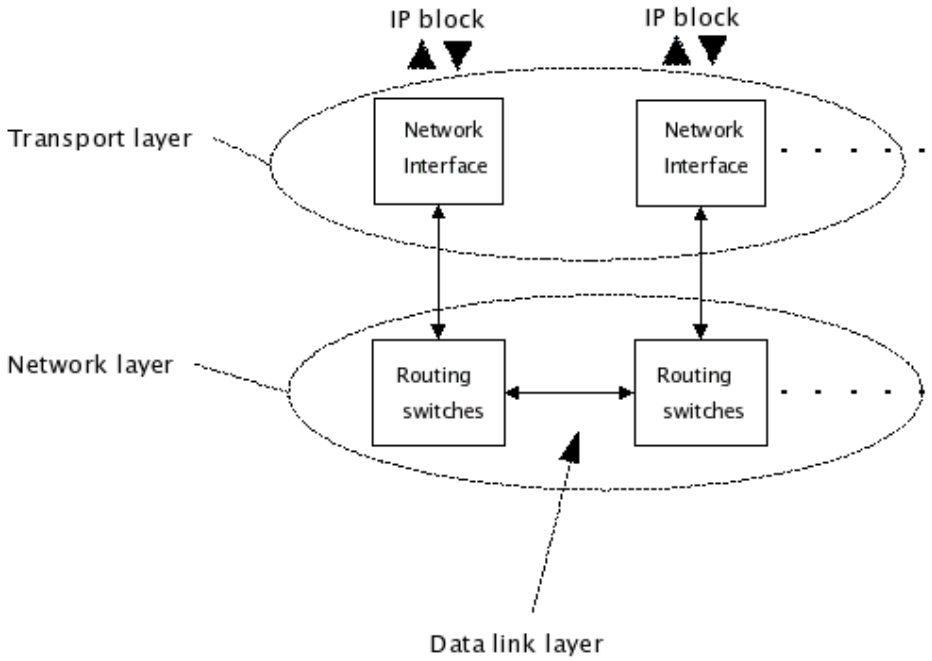

Figure 4.2: Layers clustering

Following the micronetwork stack layout shown in figure 4.1, the remainder of this chapter unfolds the three architecture and control layers – data link, network, and transport – from the ground up.

4.3 Data link layer

The data link layer defines how to manage the allocation of resources required by packets as they advance along their route to final destination [21]. The key resources in most interconnection networks are the *channels* and *buffers*. Buffers are storage implemented in the routers, such as registers or memories, and allow packets to be held temporarily at the routers ingress and egress stages.
A good flow-control strategy at data link layer must avoid conflicts that could end up idling useful resources (e.g. buffers). For instance, it should not stop a packet craving for an unused channel because it is waiting on a buffer locked by a packet that is blocked on a busy channel. The solution consists in *decoupling* resources, allowing the blocked packet to advance without waiting.
A well-designed flow control, hence an effective data link layer, should be *fair*. An unfair control flow can cause a packet to wait endlessly. Wormhole flow--control is a well-known technique to achieve high throughput. Each packet in wormhole flow control is divided into *flits* [46] (FLow control digITS). A flit is the smallest piece of information that is schedulable by the flow control method.
In the remainder of this section flow control methods will be discussed with a clear focus on their applications to on-chip networks.

4.3.1 Flit-level flow control

Resources coupling is a major hurdle in any interconnection network, and it represents a very limiting factor in achieving good throughput performance. Interconnection networks are made up of two types of resources: buffers and channels. Once a packet A gets a buffer b_i, no other packet B can use the associated channel c_i until A releases b_i. In networks that implement flit--level flow control, packet A may be blocked due to contention elsewhere in the network while it is still holding b_i. In this case channel c_i bandwidth is wasted because there may be other packets in the network, e.g. packet B, that can make use of the channel resource.
This problem of bandwidth waste due to resource coupling is unique to interconnection networks endowed with flow control managed at the flit-level.

4.3.1.1 Wormhole flow control

In Wormhole routing the head of a packet advances directly from incoming to outgoing channels of the routing chip; this contrasts with cut-through switching[80], where packets are atomic and they have to be stored as an atomic piece of information.
A packet is divided into a number of *flits* for transmission. The size of a flit depends on system parameters, in particular the channel datapath. The header flit (or flits) opens the route. A router examines the header flit(s) of a message, and it selects the next channel on the route and begins sending flits on that channel.
As the header advances along the specified route, the remaining flits follow in the network pipeline. The head flit has to reserve three resources before it can be forwarded to the next node along a route: a virtual channel for the packet, one flit buffer, and one flit cycle of the link bandwidth to advance.
The tail flit of a packet is handled like a body flit, but also releases the virtual channel at which it passes through. A virtual channel maintains the state needed to allocate the flits of a packet over a channel.
Compared to cut-through flow control [80], wormhole flow control makes far more efficient use of buffer space, as only a small number of flit buffers are required per virtual channel. In contrast, cut-through flow control allocates buffer space on a per packet basis, which implies much more storage than wormhole flow control. Buffers in wormhole flow control are allocated on a per flit basis, hence a flit can block because the channel it is requesting has already been granted to another packet. If a flit cannot make progress because of a buffer allocation possible failure, the channel goes idle. Even if there is another packet that could potentially exploit the unused bandwidth, it cannot, because the blocked packet owns the single virtual channel associated with this link. Even though wormhole flow control allocates channel bandwidth on a per flit basis, only the flits of one packet can exploit this bandwidth.

4.3.1.2 Virtual channel flow control

Virtual-channel flow control enhances wormhole flow-control by allocating several virtual channels (channel state and flit buffers) to a single physical channel; it helps overcome the blocking problems of wormhole flow control. It permits to other packets to use the channel bandwidth that would be left unused when a packet blocks. As in wormhole flow control, a header flit reserves a virtual channel, a downstream flit buffer and an unit of link band-

width to advance. Afterwards, intermediate flits from the packet use the virtual channel allocated by the header and still must allocate a flit buffer and channel bandwidth.
However, unlike wormhole flow control, these flits have not a guaranteed access to channel bandwidth because other virtual channels compete to send flits of their packets along the same link.
Virtual channels allow packets to pass blocked packets, making use of otherwise idle channel bandwidth, as shown in figure 4.3. Figure 4.3 (b) shows

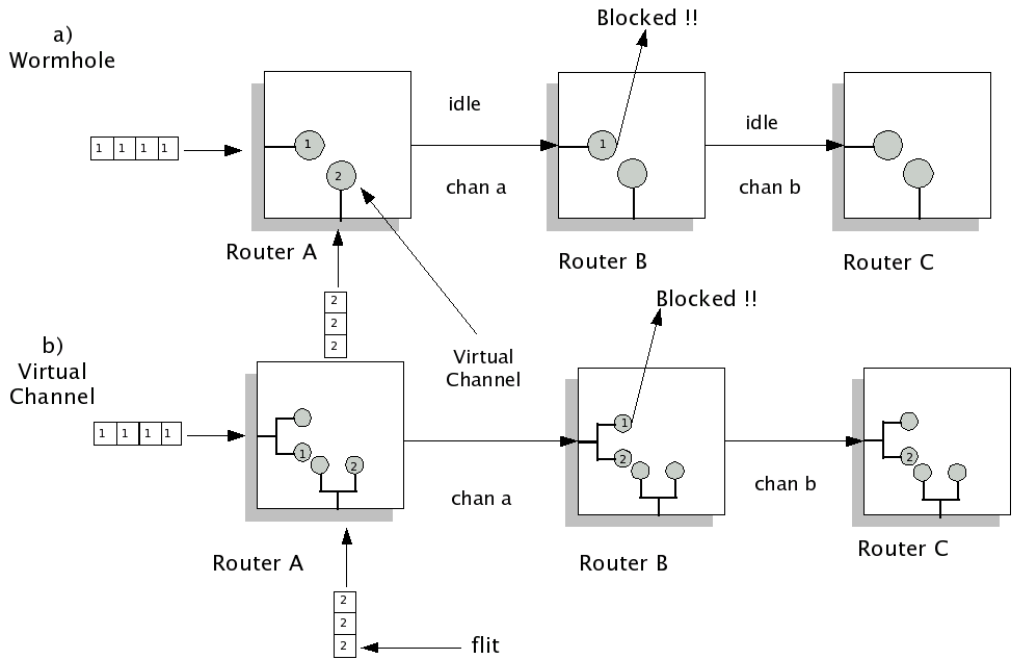

Figure 4.3: In wormhole flow control (a) Packet 1 blocks; channel a and channel b are idle even though packet 2 could exploit their bandwidth. In virtual channel flow control (b) 2 can advance by using a second virtual channel.

a router configuration with two virtual channels per physical channel. Each virtual channel encompasses a buffer and a state. In this case packet 2 can allocate the second virtual channel on router B and thus advance to router C, using otherwise wasted bandwidth of channel a and b.

4.4 Network Layer

The network layer role consists in delivering packets from the source to the destination. This function differs from that of the data link layer, which has the goal of managing flits between node end-points. To achieve its goals, the network layer must know about the topology of the communication network (the interconnection lay-out of all routers) and choose appropriate paths through it.
The remainder of this section will highlight different network layer issues closely related to networks on-chip.

4.4.1 Network topologies

Scalability is an important issue in designing SoC [57]. The *direct network* is a network architecture that scales well to a large number of processing elements [24]. A direct network consists of a set of *nodes*, each node being directly connected to a subset of other network nodes in the network. A common component of this node is a *router*, which implements network layer communication among nodes.
Direct networks are defined by a graph $G(C, N)$, where the vertices of the graph N represent the set of computation elements and the edges of the graph C represent the set of the communication channels. Several network properties can be defined considering the graph representation:

- *Node degree*: Number of channels used to connect a node to its neighbors
- *Diameter*: The maximum distance between two nodes in the network
- *Regularity*: A network is *regular* when all the nodes have the same degree
- *Symmetry*: A network is *symmetric* when it looks alike from every node

The *topology* defines how the nodes are interconnected by channels and is usually modeled by a graph as indicated above. Many network topologies have been proposed in terms of their graph-theoretical properties (see figure 4.4).

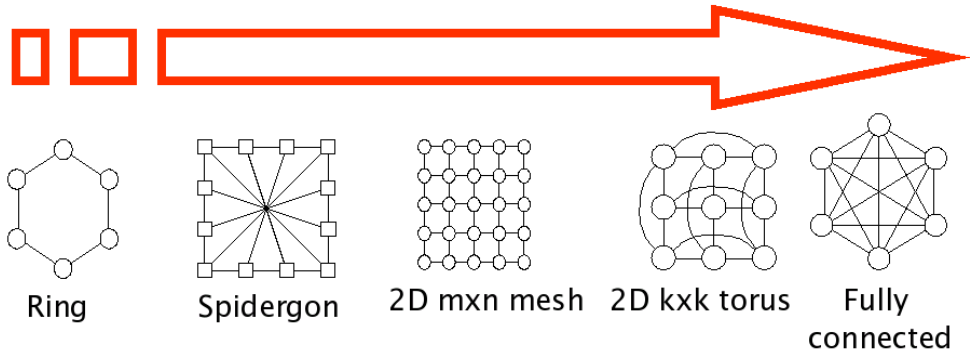

Figure 4.4: Example of networks topologies

The most popular direct networks are the *n-dimensional mesh*, the *k--ary n-cube* or *torus* and the *hypercube*. All of them are strictly orthogonal, meaning:

- Nodes can be laid out in an orthogonal n-dimension space end every link can be arranged in such a way that it produces a displacement in a single dimension.
- Every node has at least one link crossing each dimension.

In addition to strictly orthogonal topologies, many other topologies have been proposed in literature. A popular topology is the tree. This topology defines a connection scheme in which a root node connected to a given number of descendant nodes. Each of these nodes is in turn connected to a subset of descendants.
A node with no descendants is a *leaf node*. In trees the root node and the nodes in its proximity become a bottleneck to the global interconnection. A practical way to design trees with higher channel bandwidth in the proximity of the root node is a fat tree (see figure 3.10 on page 43) implementation (see [6] for further details).

STMicroelectronics chose to develop an in-house topology dubbed "Spidergon". Spidergon uses a regular, constant degree topology that belongs to the family of *undirected k-circulant graphs*, i.e. it can be represented as $G(N; s_1; s_2; \ldots; s_k), 0 < s_I < N$, where s_i is an undirected edge between any network node l and node $(l + s_i) mod N$. *Chordal rings* are circulant graphs,

with $s_1 = 1$. These families of graphs have been theoretically studied as competitors to meshes and tori in respect to graph optimality, i.e. minimum diameter graphs for a given number of nodes and constant degree for a given number of nodes.
The Spidergon topology has an extensibility of 2 nodes, and for N=8 it reduces to STM *Octagon* [26]. The total number of edges is $\frac{3N}{2}$, while the diameter is $[\frac{N}{4}]$.
For current realistic NoC configurations with up to 60 nodes, the proposed graph has a smaller number of edges, and a smaller diameter than fat-tree or mesh topologies, leading to latency reduction for small packets, even when wormhole routing is employed.

4.4.2 Routing algorithms

Routing algorithms define the path followed by each message or packet. The list of routing algorithms proposed in the literature is almost endless.
The routing algorithm used influences many properties of interconnection networks. Hereinafter the most important ones are reported:

Connectivity. It deals with the capability to route packets from any source node to any destination node.

Adaptivity. Ability to find alternative paths for packets in the presence of contention.

Deadlock and livelock freedom. Ability to guarantee that packets will not block or wander across a network forever.

The next subsections present the two most important classes of routing algorithms.

4.4.2.1 Deterministic routing

The simplest routing algorithms are deterministic – they send every packet from source x to destination y always using the same path. The choice of routing algorithm can significantly affect network load. For instance, *deterministic* routing algorithms always choose the same path between x and y, even though there can be multiple paths (not minimal, though). These algorithms do not consider path diversity of the underlying topology which worsens load balance properties.

Despite this shortcoming, they are very common in on-chip networks because they are easy to implement and require little silicon area. For networks on-chip they are the most used routing scheme because of their simplicity.
For Multiprocessors System on-chip in which ordering of packets is a crux between particular source-destination pairs, deterministic routing is often a simple way to provide this ordering. This is important for instance, for certain cache coherence protocols.

4.4.2.2 Adaptive routing

An adaptive routing exploits the network state, typically queue occupancies, to select among all the possible paths to deliver a packet. Because routing algorithm depends on the network state, the flow-control method plays a role in adaptive routing.
This contrasts with deterministic routing in which the routing algorithm and the flow control mechanisms are definitely decoupled. Many of the issues involved with adaptive routing can be explained by considering an example of a 6-node ring topology (see figure 4.5). Node 4 is sending a continuous burst

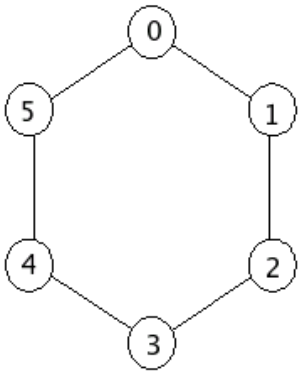

Figure 4.5: 6-nodes ring topology to demonstrate adaptive routing

of packets to node 5, using all available bandwidth [4,5]. In the meantime, node 2 wants to send a packet to node 0.
It can choose either clockwise or counterclockwise route. Readers could easily state that the router at node 2 should choose the counterclockwise route

to avoid contention. However, a typical router does not keep a global state of the network.
The way in which a router collects the state of the network is a crucial question for adaptive routing. Algorithms can use local or global information, and also current or old information. Several kinds of routing can be implemented ranging from minimal adaptive routing to fully adapting routing. Minimal adaptive routing always chooses the shortest path towards destination. This assumption is relaxed in fully adaptive routing.
Whether or not adaptive routing algorithms will be integrated in networks on-chip is due mainly to complexity and most importantly to transaction ordering. Paths adaptation can lead to out-of-order management of packets that could easily end up wreaking havoc in memory coherent multiprocessor systems.

4.4.3 Guaranteed services

System-on-chip era forces designers to deal with the increasing design complexity, promoting a heavy reuse of intellectual properties (IPs). This means that applications become dynamic compositions of IP blocks which require a network which is scalable, programmable and behaves predictably under the traffic offered by those blocks.
So far, methods to achieve *efficiency* of resources allocation (e.g. flow control) were described. However, even the best routing and flow control methods cannot overcome situations in which a request for a particular resource cannot be served, because the resource is already overcommitted.
Quality of Service (aka QoS) represents a methodology to provide a *fair* allocation of resources according to some service policies. On-chip traffic can be classified in two broad categories:

- Guaranteed traffic
- Best Effort traffic

This distinction between traffic mirrors traffic requirements. In particular, the notion of best effort traffic is somewhat related to connectionless service in that the network ”does its best” to deliver packets with low latency and providing enough throughput, but network does not ensure any guarantees.
On the other hand guaranteed service classes are guaranteed a certain level of performance as long as the traffic they generate complies with a set of constraints. Somehow, there is a network ”contract” between the IP and the

network.
The remainder of this subsection explores two possible ways of providing guaranteed services.

4.4.3.1 Traffic shaping

IP Traffic can either be generated regularly or irregularly, causing spots of non-uniform behavior that possibly leads to congestion. The simplest way to guarantee a service is to require that the compound requests for a given traffic class are less than a bound. Traffic conforming to this bound then is guaranteed not to saturate the network.
Traffic shaping is about regulating the average *rate* (and burstiness) of IP data transmission. When a connection is set-up, the IP and the network agree on a certain traffic pattern. This could be seen as a *service level agreement*. As long as the IP sends packet according to the agreed-on contract, everything goes as expected. Traffic burstiness can exceed bandwidth of some ports within the network wreaking havoc in performance. Hence, in situations where stronger guarantees are strictly necessary, it might be compulsory to reserve specific resources over the network at of course much higher hardware cost.

4.4.3.2 Resource reservation

In situations where hard constraints are required, it is mandatory to reserve resources rather than rely on traffic shaping. The major drawback is that resource reservation comes at greater hardware overhead, because these reservations must be stored in the network, implying silicon area to be allocated for this purpose.
Two mainstream and well-known resource reservation methods seem feasible for on-chip networks:

- Virtual circuits

- Time-division multiplexing

In virtual circuits, each flow is assigned a specific route through the network. This approach, which is inherently *connection-oriented*, requires first of all a connection set-up from source IP to destination IP before any data is sent. In the connection set-up phase, it is necessary to establish a ”connection state” in each of the switches between the source and the destination hosts. The

connection state for a single connection consists of an entry in a "VC table" in each switch through which the connection passes. One entry in the VC table on a single switch contains:

- a *virtual circuit identifier* (VCI) that uniquely identifies the connection at this switch and that will be carried inside the header of the packets that belong to this connection
- an incoming interface on which packets for this VC arrive at the switch
- an outgoing interface in which packets for this VC leave the switch
- a potentially different VCI that will be used for outgoing packets

The semantics of one such entry is as follows. If a packet arrives on the designated incoming interface and that packet contains the designated VCI value in its header, then the packet should be sent out the specified outgoing interface with the specified outgoing VCI value first having been placed in its header.
Note that the combination of the VCI of packets as they are received at the switch and the interface on which they are received uniquely identifies the virtual connection. There may of course be many virtual connections established in the switch at one time.

Time-division multiplexing (TDM) provides the best control in flits scheduling. In order to avoid fluctuations due to resources sharing, TDM reserves all the resources needed by a particular flow in both time and space.
Because flows have a scheduled access to resources, guarantees are easier to design.
A TDM implementation splits time into a fixed-number of small slots. The size and number of slots then define the granularity at which a resource can be allocated. For instance, the flow of time might be partitioned into 10 slots, with each slot equal to the transmission time of a single flit. If the channel bandwidth is 2Gbytes/s, each traffic flow could allocate bandwidth in multiples of 200 Mbytes/s.
If a flow required 400 Mbytes/s it has to allocate 2 of the 10 slots for each resource it needed. As shown in figure 4.6, to store a timetable for each resource in the network a *timewheel* can be employed.
A pointer into the timewheel table indicates the current time slot and the resource owner during that time slot. For this example, unused slots are marked as "BE" because they can be scheduled for best-effort traffic. The pointer is incremented to the next table entry when time ticks, wrapping to the top once it reaches the bottom of the table.

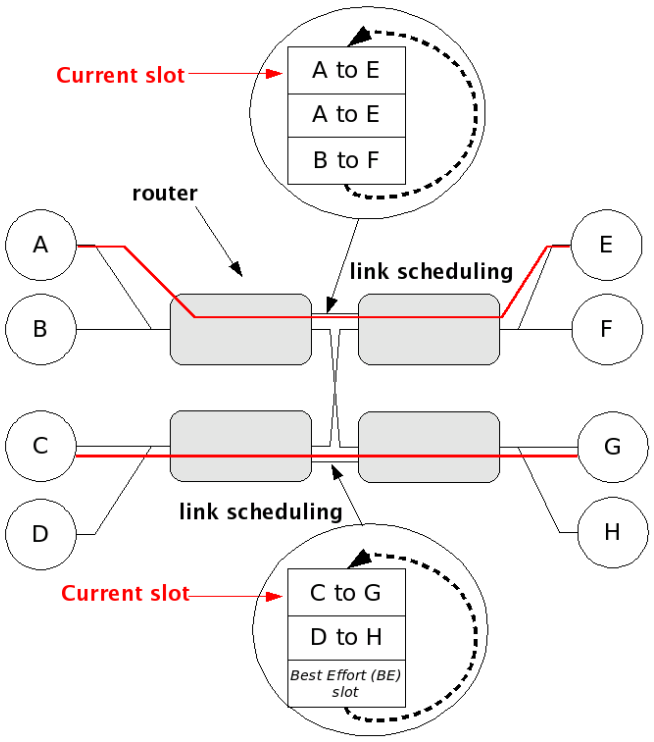

Figure 4.6: TDM network policy example

4.4.4 Best effort services

In the *best effort model* the network just makes its "best effort" to deliver the packets injected into it without agreeing to any quantitative performance (quality of service QoS) bounds.
IPs do not request permission before transmitting, and therefore performance is determined not only by the network itself, but also from IPs traffic combination, resulting in a complete lack of isolation. Best-effort model is closely related to the notion of *fairness*. The notion of fairness is of major importance in the best effort on-chip networks, due to the lack of explicit admission control and quantitative service assurances. Fairness is conceptually related to congestion control; under conditions of low load IPs requirements are satisfied, there is no need for trade-offs and no considerations for decisions that lead to fair allocation of resources. Fairness becomes an issue only when there are traffic cases which lead to network overcommitments and IPs have to compete for their share. Although several definitions of fairness arise from various disciplines, in the networking world the most popular notion actually is that of *max-min fairness* (also referred to as classical notion of fairness). Formally, let I be a set of requesters and $x = (x_i; x_i > 0; i \in I)$ the vector of the allocations to each requester. The vector is called feasible if the sum of the allocations does not exceed the capacity of the resource. Max-min fairness operates as follows:

- Resources are allocated in increasing order of demand.
- A requester is never allocated a share higher than its demand.
- All requesters with unsatisfied demands are allocated equal shares.

Initially all requesters get at least as much as the "small" requester demands, and the remaining resources are *evenly* distributed among the requesters with unsatisfied demands. This means that from those requesters with unsatisfied demands no one can increase its share without reducing the share of a requester with an already small one. This can be formally stated as follows: a vector of allocations x is max-min fair if for any other feasible vector y there exists a requester j such that $y_j > x_j$ implies that there exists requester i such that $y_i < x_i < x_j$. Fairness [29] should not necessary imply equal distribution of resources to all those users with unsatisfied demands. A fair allocation of resources is usually defined with respect to a given policy. A complete discussion of fairness policies is out of scope of this thesis; however, max-min fairness represents the basic concept on which fairness for

best-effort networks can be developed with a mechanism called arbitration. In the next subsection a number of arbitration schemes will be addressed to explain how to achieve fairness for different purposes and at different costs.

4.4.5 Arbitration policy and algorithms

Arbitration is the method through which a certain fairness can be achieved in networks on-chip. Different arbitration algorithms provide a broad range of performance at variable costs in terms of hardware required.
A crucial property of an arbiter is its capability to serve different requests in an equal manner. The meaning of "equal" here is somewhat misused, because its meaning depends strictly on the context and varies from application to application.
A typical example of a completely unfair arbiter is a fixed priority one. Each request is assigned a priority which is hardwired and does not change. The implementation of such a hardware circuit is very light in terms of hardware, but the arbitration method is utterly unfair in that if the request coming from the highest priority is always asserted, none of the other requests will *ever* be served.
Round-robin arbitration scheme partially solves the drawbacks of fixed priority arbiter in terms of fairness. A round-robin arbiter is based on the principle that a request that was just served should get the lowest priority on the next round of arbitration. The round-robin arbiter ensures that after a request has been served it becomes lowest priority. If no requests are asserted priorities do not change.
In order to introduce a controlled degree of unfairness, a *weighted round--robin* arbiter can be designed. When this arbitration scheme is deployed a requester is given a number of grants proportional to a *weight*. Each request is coupled with a weight w_i that indicates the maximum fraction f_i of grants that requester i can receive according to $f_i = \frac{w_i}{W}$ where $W = \sum_{j=0}^{n-1} w_j$. A requester with a large weight will receive a large fraction of the grants while a requester with a small weight receives a smaller fraction. For example, a weighted round-robin arbiter with five inputs with weights of 2, 3, 3, 6 and 8 will receive $\frac{2}{22}$, $\frac{3}{22}$, $\frac{3}{22}$, $\frac{6}{22}$ and $\frac{8}{22}$ of the grants respectively. Choosing the number of weight bits allows weights to be specified with very high precision.
The last arbitration algorithm considered in this subsection is Least Recently Used (LRU). It slightly differs from a round-robin policy.
The logic behind round-robin is simple but in on-chip networks could lead to unfairness. The main drawback of round-robin comes up when requests of a given round-robin index are not driven, causing the priority "turn" to be

lost (the index just takes into account the currently granted requester and it increments). LRU helps overcome this issue. Whenever a requester that has not been served from the longest time finally comes up with a request, it is automatically granted. Normally LRU is implemented as a simple ordered list in hardware, with little silicon area. Its usage is widespread in most of the current implementations of on-chip networks.

4.5 Transport Layer

The transport layer [74] is the core of the whole protocol hierarchy. Its task is to provide a data transport channel from the source IP (or entity, see figure 4.7) to the destination IP, independently of the physical network currently in use. Without the transport layer, the whole concept of layered protocols would make little sense. The transport layer defines the communication primitives available to interconnected IP blocks.
Special NoC network interfaces [32] (NI), located at NoC boundary, provide transport-layer services to IP blocks with which they are paired. This is analogous in data communication network, to Network Interface cards that source/sink information to the LAN/WAN media. The transport layer defines how information is exchanged between Network Interfaces to implement a particular transaction. For instance, a NoC transaction is typically made up of a request from a master Network Interface to a slave Network Interface, and a response from the slave to the master.
However, the transport layer hides the implementation details of the exchange to the network and physical layer. Network interfaces that bridge the NoC to an external protocol (such as AMBA AHB) translate transactions between the two protocols (bus to NoC and viceversa), tracking transaction state on both sides. For compatibility with existing bus protocols, NoCs implement traditional address based Load/Store transactions, with their usual variants including incrementing, streaming, wrapping bursts, and so forth.
In the remainder of this section an example of bus-bridging will be described as well as advanced operations that might possibly impact transport layer features of networks in the years to come.

4.5.1 Bus bridging

One of the major challenges for networks on-chip takeover is the capability to provide an homogeneous interconnection network which can behave as an open socket where components can be plugged at will. This challenge

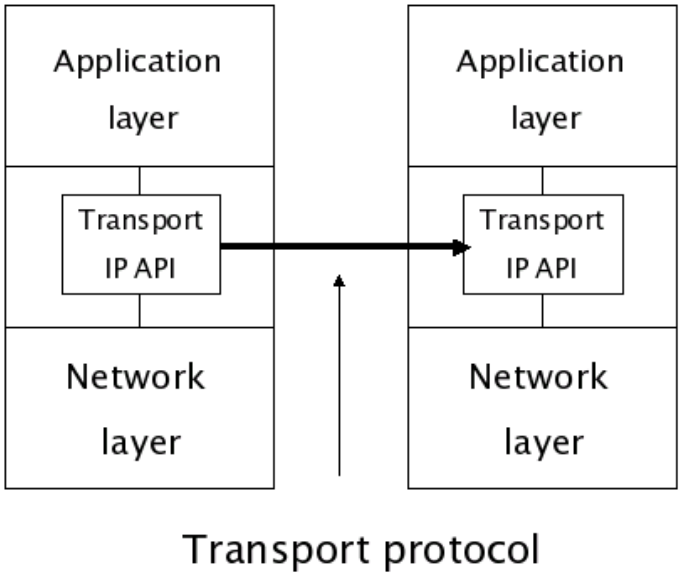

Figure 4.7: Transport layer

represents also a major need in that the tight deadlines for time-to-market force designers to leverage IPs reuse; IP reuse often implies interface reuse, so that typical bus based load/store transaction must be adapted to network protocol.
This is what transport layer is all about. Taking the example of an AHB bus [2], the transport layer protocol, namely the transactions between IP end-points is fixed by bus protocol rules. The role of transport layer is to serialize traditional busses control and data paths into a bunch of wires which correspond to flits, in a way that makes sense for both ends of the network, to wit, master and slave. In figure 4.8, a typical bus-transaction is shown. It is made up of typical controls such as address (HADDR), opcode (HTRANS, HWRITE) and data path (HWDATA, HRDATA), with related slave acknowledge mechanism (HREADY).
In order to establish an effective communication between networks end-points which actually are plain masters and slaves, these pieces of information must be packed in some way by master/slave components in order to be rebuilt at the other network end. In figure 4.9 an end-to-end communication is shown where a master wants to connect to a slave through an interconnection network.

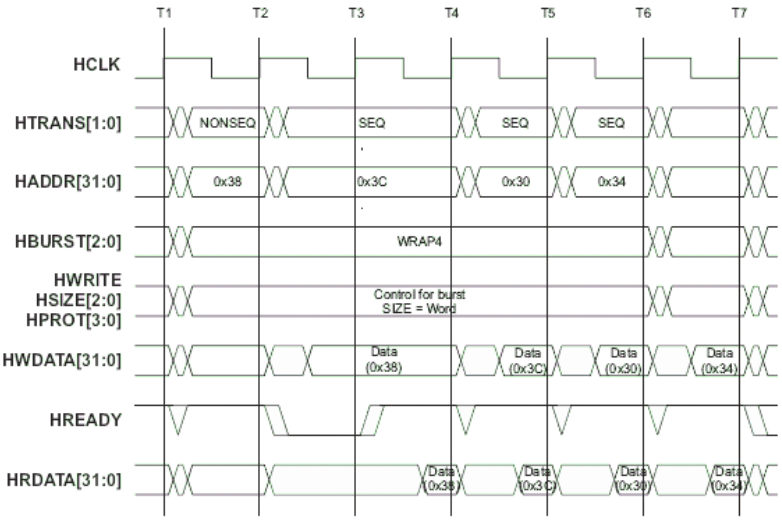

Figure 4.8: AHB wrap4 burst transaction

The transport layer represents the glue that ”packs” control and data into header flits suitable to be unpacked at the slave end point. Figure 4.9 shows also a possible bitmap where AHB controls are mapped in a hypothetical header flit to be send over a network. This header flit will drive the AHB slave control logic in order to rebuild the precise controls for the AHB slave, at the slave end-point. It is up to the Network Interface to fill up the header flit with proper bits, and also to manage the communication with the given upstream/downstream bus protocol.
Network Interface has also to fiddle with network layer bits, but this topic is out of scope for confidentiality reasons. Suffice it to say that the network interface is loaded with look-up tables providing network topology layout and proper routing information.

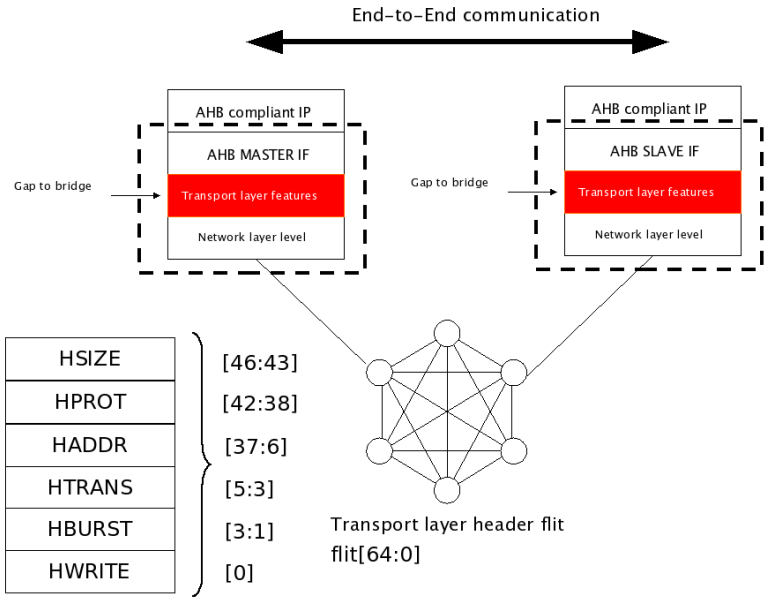

Figure 4.9: AHB end-to-end connection

To conclude this subsection, it is important to underline two crucial points:

- Bus bridging can be considered just a pain from the past (protocol conversion is a hindrance to bandwidth and latency just because it wastes bus cycles for nothing). A brand new transport layer protocol able to unify all the existent bus interfaces would be a godsend from different perspectives. That is why major semiconductor companies are striving to exploit NoC development to end, somehow, bus bridging madness.

- Different systems require different protocols, hence bus opcodes. Whilst Multi-processor systems force the usage of complex end-to-end opcodes (as the next sections describe) for memory coherence, other just require opcode to bring big chunks of data to memory back and forth. An open protocol (this is how it has been defined) must take into account all possible opcode combinations. There are not many systems designers who are wringing their hands to accomplish this very convoluted task.

4.5.2 Advanced protocol issues

Multiprocessors are by now common in server environments, and several desktop multiprocessors are available from vendors such as Sun, Intel, Compaq and Apple. In the embedded space, a number of special purpose designs have used customized multiprocessors, including the Sony Playstation.
Many special purpose embedded designs consist of a general-purpose programmable processor with special purpose finite-state machines that are used for stream oriented I/O. In applications ranging from computer graphics and media processing to telecommunications, this style of special purpose multiprocessor is becoming common.
Centralized shared memory architecture multiprocessors are becoming common even in embedded world. Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are defined as *symmetric shared-memory multiprocessors* (SMPs) and this style of architecture is called UMA for uniform memory access (see figure 4.10). A second type of multiprocessors consist of multiprocessors with physically distributed memory. To support a large number of processors, memory must be distributed among the processors rather than centralized, this mostly for bandwidth concerns.
Of course, the larger number of processors raises the need for a high bandwidth interconnect (see figure 4.11). This is where networks come into play.

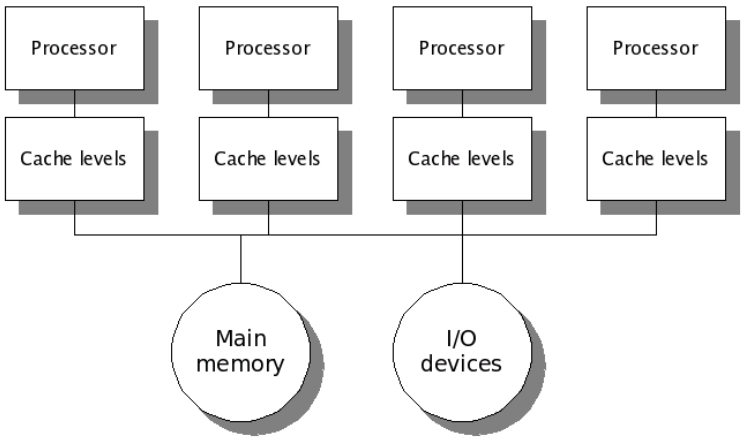

Figure 4.10: UMA multiprocessors architecture

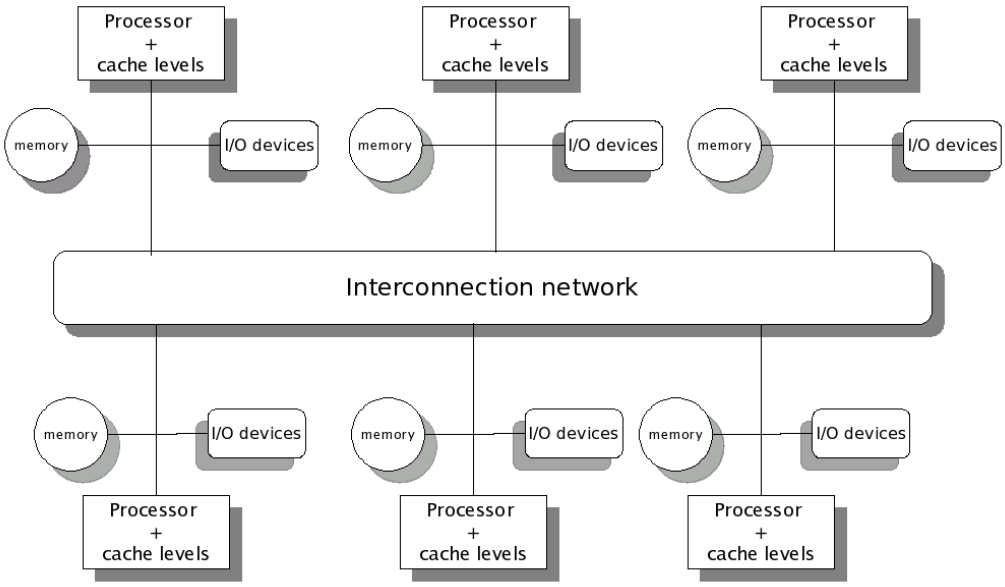

Figure 4.11: NUMA multiprocessors architecture

A number of programming models have been proposed for parallel machines (e.g. MPI) and memory architecture (e.g DSM). The interested reader is invited to refer to [58] and [23].
Multiprocessors systems not only affect bandwidth but also how memory access should be performed. In particular *synchronization* is pivotal to avoid multiple threads of execution to race on shared variables (locking) and to provide *atomic operations* such as test and set.
Possibly concurrent execution paths can wreak havoc if some degree of caution is not exercised when accessing shared variables between threads. A typical primitive to protect *critical regions* is a *spinlock*, that means a lock that a processor continuously tries to acquire, spinning around a loop until it succeeds.
While programming details of synchronization primitives are beyond the scope of this thesis (see [47][10][37][11] for additional insights), the aftermath of this synchronization primitives is of major importance for interconnection; namely, the network must support primitives to guarantee atomicity of operations that in principle are not atomic (e.g. read-modify-write). The remainder of this section deals with advanced protocol opcodes in networks that can be useful to guarantee multiprocessor consistent programming models.

4.5.2.1 Atomic transactions and compound operations

In figure 4.12 two processors execute code that pokes a shared variable `i`. As the variable is shared some form of synchronization is needed between the threads that make up the application.
In this case a *spinlock* has been used. If the two processors compete for the spinlock, the one that locks it can directly access the shared variable whilst the other keep spinning on the lock till it is released.
Anyhow, even the spinlock itself, being a shared variable, suffers from racing problems between concurrent access. So, there must exist some way in which the system can guarantee that spinlock locking is *atomic* from a thread of execution perspective. In shared busses this requires a feature such as *lock* to merge interconnection transactions together in order to be indivisible from a memory access point of view (acquiring a spinlock implies at least two transactions on the network a read and a write).
For a bus solution is not too difficult to guarantee atomicity, but what about a network where links are distributed ? Should the network lock all the links to memory ? or some cleverer solutions must be adopted ? There is no easy answer. In a system containing more than three or four processors the

bandwidth required by spinlocks contention would be unbearable. Moreover, all this bandwidth is completely and definitely wasted because it just carries synchronization variables, not useful data for further processing.
Several solutions have been devised, all applicable to networks on-chip designs. The most interesting resides in hardware locking primitives. The major problem with original spinlock implementation is that it introduces a large amount of unneeded contention. For instance, when a lock is released, all processors generate read and write cache misses, although at most one processor can get the lock in an unlocked state. This situation can be improved by explicitly handling the lock from one waiting processor to the next.
In hardware, a list of waiting processors competing for the lock is held, and it grants the lock to one explicitly, when the locking processor stops spinning because it exits the critical region. This concept is called *queuing lock*. This greatly enhances synchronization operations in multiprocessors programming model at little hardware cost.

Atomic operations are just one part of the whole story. Current multimedia systems on chip contain a load of IPs with strict throughput requirements to access memory. Double Data Rate (DDR) memories are made up of different banks, and within each bank memory is split into pages. The way in which transactions are ordered towards memory definitely change how pages banks are accessed, affecting memory effectiveness.
In order to group transactions directed towards the same memory page (accessing consecutive locations in a memory page avoids costly page misses that imply memory stall cycles), compound transactions are built up to *lock* transactions together. In this way the interconnect is not allowed to split traffic coming from different flow because the path is *locked*. This issue is strictly related to atomicity of transfers, but for different purposes compared to synchronization. Atomicity for compound operations is a good feature because it boosts memory effectiveness. Hence, it is an *improvement* (see [67]) of existing interconnection protocol. Atomicity for synchronization is just an hurdle to deal with; it does not bring anything useful on system performance. When spinlocks are not based on hardware queuing locks, a simple lock signal on network protocol suffices to guarantee both compound operations and synchronization primitives.
However, bandwidth waste due to synchronization in this case is hard to sustain, so that synchronization in future systems is likely to be implemented through hardware mechanisms. Advanced bus protocols such as AMBA AXI™ feature also a different form of synchronization opcodes: exclusive access. In this case path locking is avoided, but this synchronization comes at a price.

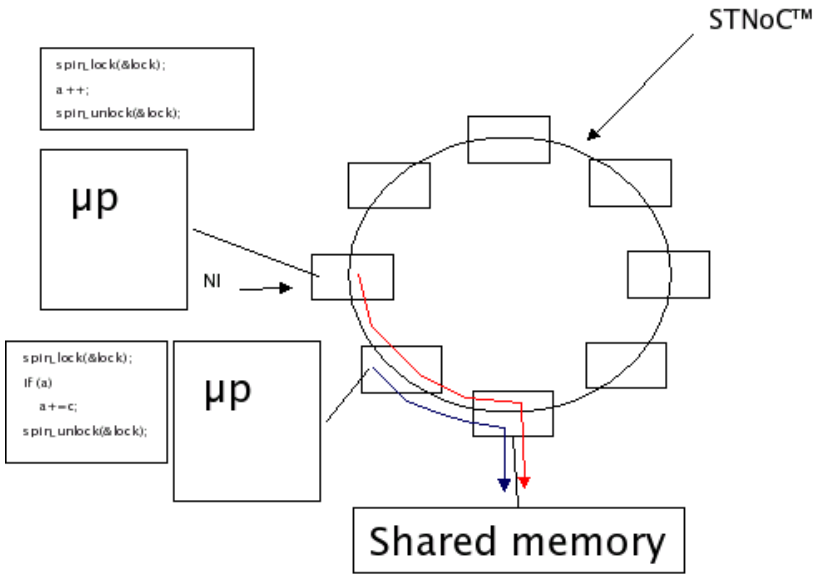

Figure 4.12: Example of race prevention

Atomicity is guaranteed by the slave component; if it decodes an exclusive access violation, an error opcode is returned to the master, meaning that the request was not successful. While this approach avoids path locking, it can unleash an unsustainable bandwidth on the communication channel (masters getting error responses retry transactions again and again), ending up with an unfeasible solution for most systems.

To sum up, multiprocessor systems atomicity and bandwidth requirements put tough issues on interconnection design. While for shared busses atomicity guarantees are easy to provide (locking the bus), a distributed network require thoughtful design choices to be made in order to avoid bandwidth waste and design overcommitments.

4.6 STNoC™ network on-chip

The STMicroelectronics STNoC™ project can be seen as a seamless evolution of the STM proprietary STBus protocol, by now deployed in a number of SoC designs. The main objective and idea behind STNoC is to bring an innovative solution to on-chip communication, based on a packet switching µnetwork of components.
STBus based solutions started suffering from shortcomings such as wires

congestion and difficult timing-closure (see section 3.2 page 28), so that some technological improvements are in order to empower ST designers with an interconnection scheme suitable for next generations systems. STNoC™ has been built with two basic guidelines:

- Simplicity
- Pragmatism

An innovative solution to the on-chip interconnection problem has to be somehow "simple" in that silicon area is a major cost and competitive factor; interconnection networks must prove effective with as less buffering as possible, saving area for advanced computing features such as cache-memories, processor register files, etc.
Even if the interconnection network may by now be considered as the real added-value for SoC, its role is "just" to shuttle code and data within the system. Computation, so applications, is executed elsewhere (Processing elements); thus, computation part of the chip represents a major cost factor and its improvement demands area (e.g. memory hierarchy enhancements, hardware acceleration). A SoC where the interconnection scheme occupies a big slice of the whole chip area would be definitely frowned upon by system designers, implying its quick death.

Pragmatism is a pivotal concept in SoC design; the introduction of networks on-chip in semiconductor industry arena might be seen as the integration of convoluted protocol stacks to provide on-chip services such as connection-oriented schemes, namely the translation on-chip of well-known concepts coming from the fascinating networking world. The design of on--chip network must be "pragmatic" from this point of view; for instance a hardware/software network stack such as TCP/IP is well worth in computer networks, but it is just a harebrained concept for on-chip interconnection where routers have to be plain hardware elements used to dispatch packets with no software stack built on top of them.
STNoC™ was designed with simplicity and pragmatism in mind, in order to design a network with all nuts and bolts, but without introducing exotic concepts and implementation features on already burdened on-chip interconnects. STNoC™ topology is Spidergon based (see subsection 4.4.1 page 59); despite its simplicity Spidergon topology provides a good trade-off between performance (e.g. bisection bandwidth and diameter) and cost (e.g. router arity); STNoC™ implements virtual channel flow control, as described in section 4.3, with decoupled virtual networks for requests and responses to guarantee deadlock-free routing and operations.

The remainder of this section describes, as long as possible due to confidentiality reasons, the different components of STNoC™ in order to provide the groundwork to describe STNoC™ modeling and benchmarking in the following chapters.

4.6.1 STNoC™ router

The router is one key component of the STNoC™ interconnection network. It represents the core communication medium and it covers the protocol layers described in this chapter up to network layer. Thus, the router is responsible for forwarding and routing packets throughout the network, from source all the way down to destination.

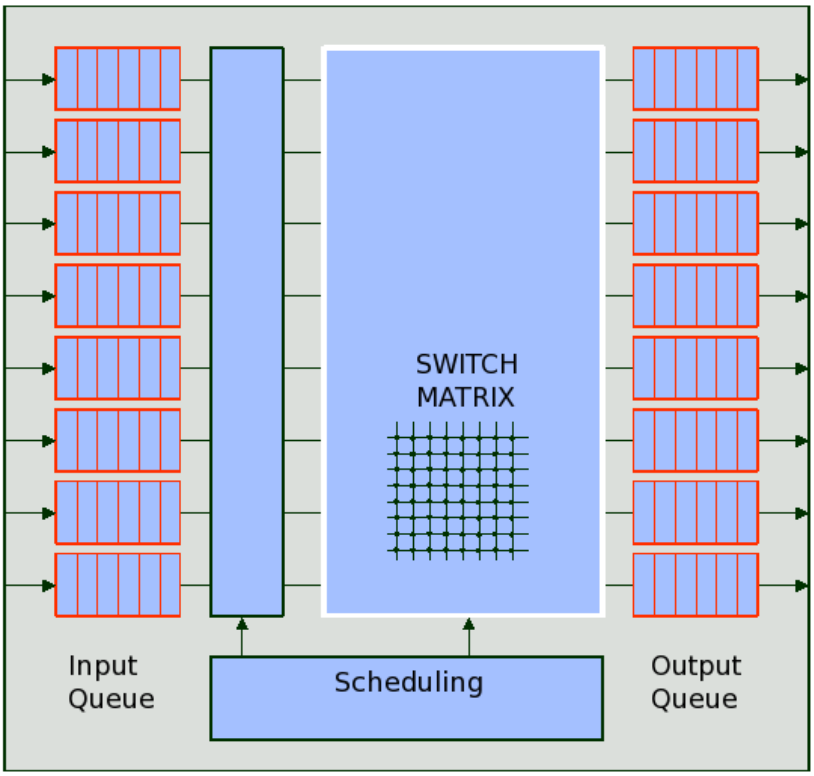

Figure 4.13: Generic NoC router microarchitecture

In figure 4.13 a generic router microarchitecture is shown. Generally speaking, a router is made up of three blocks:

- Input

- Switch
- Output

This is by no means a thorough description. A detailed router description implies the definition of a number of finite state machines, including flow control logic, arbitration and multiplexors. Besides, the inherent concurrency in network communication implies the presence of buffering in routers to guarantee a certain pipelining even under heavy traffic congestion situation. Different solutions apply to different cases. Normally the choice is between input/output/mixed queuing architecture. Each solution has its strengths, weaknesses and cost factors.
In networks on-chip routers, the routing algorithm is either embedded in routing tables or laid out in hardware circuits within the router. STNoC™ resorts to combinatorial logic within the router to decode the path that has to be followed by a packet traversing the network. The STNoC™ routing scheme is deterministic (see subsection 4.4.2) and it is based on Spidergon topology (see figure 4.14); given the high symmetricity and regular properties the packet decoding scheme is light in terms of gates count. In STNoC™ each

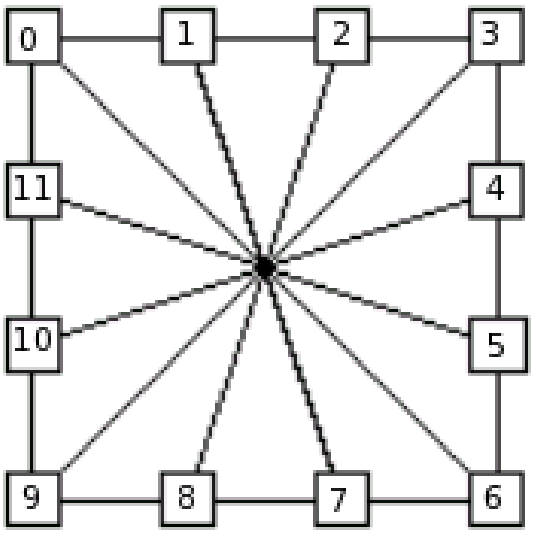

Figure 4.14: Spidergon topology

router has a unique address I in the network, $0 \leq I \leq N$, where N is the network size in nodes (see figure 4.14, where $N = 12$, numbering is clockwise).

Since the STNoC™ routing algorithm is local, i.e. identical (or symmetric) for all router nodes, and the topology is – vertex and edge transitive, we may describe the routing algorithm at any node. When a router receives a header flit, the algorithm compares the address of the current router (*curr*) to the address of the target router (*dest*) stored in the header flit; the Spidergon routing algorithm chooses the proper output port according to hardwired rules.

Hereinafter the basic deterministic shortest-path routing algorithm is unfolded:

if $(dest = curr)$ **then**
  $output \leftarrow NI$
else
  if $(|dest - curr| \leq \frac{N}{4}) \parallel (|dest - curr| \geq (N - \frac{N}{4}))$ **then**
    if $(dest \in [curr + 1, .., curr + \frac{N}{4}])$ **then**
      $output \leftarrow RIGHT$
    else
      $output \leftarrow LEFT$
    else
      $output \leftarrow ACROSS$
    end if
  end if
end if

The basic algorithm implies that the cross communication port is selected at most once, always at the beginning of each packet's route. Thus, only packets arriving from an IP resource or cross output need to be considered for routing. All other packets use the same port, i.e. clockwise or counter-clockwise, as the one they have used in the previous routing step. This might be exploited with an extra directional bit in the packet header.
The basic algorithm is appropriate not only for one-to-one (point-to-point), but also for one-to-many (broadcast, scatter), and many-to-many total exchange traffic configurations.
The configuration of the routing algorithm, together with advanced protocol features, allows STNoC ™ to support and usher a brand new quality of service mechanism built on-top of best-effort services. In particular, fairness concepts described in section 4.4 are integrated in STNoC™ router through arbitration mechanisms of router FIFOs. STNoC™ router provides combinatorial logic that allows to lock paths in order to support atomic and compound operations, as discussed in subsection 4.5.2 on page 73. This logic is light in terms of gates, providing mechanism to support advanced multi-processors

architecture and programming models.
For confidentiality reasons, the details of STNoC™ router microarchitecture and arbitration mechanism cannot be unveiled here.
STNoC™ router proved to be a really cost effective solution, allowing clock frequency up to $1Ghz$ in 90nm ST technology, with a per link bandwidth of $8G_{bytes}/sec$ (64 bits data path/flit width).

4.6.2 STNoC™ network interface

Whereas the router component manages the on-chip protocol stack up to network layer, the network interface (see also subsection 5.6.5 page 131 for NI modeling) is a key component of transport layer management and bus-bridging.
One of the toughest issues in STNoC™ deployment consisted in re-use existing IP interfaces. Within STMicroelectronics, two bus based interfaces are used. One can simply argue that two interfaces are not that many to deal with; however, we must take into account that the problem resides in their number but also in their *usage count*.
Both STBus and AMBA interfaces are *massively* deployed in STMicroelectronics systems; these legacy interfaces are "golden handcuffs" that somehow forced STNoC™ developers[1] to rethink the role of a NI in an on-chip environment. The role of the network interface (NI) [61] is to provide the conversion of the packet based communication of the NoC to the higher-level protocol than IP modules use. The design of a network interface (either master or slave) is split in two subcomponents (see figure 5.15 page 132):

- The Shell NI, which manages IP flow control, transaction ordering and other high-level protocol issues specific to the protocol offered to the IP.

- The kernel NI, which implements the channels, packetize messages and schedule them to the routers, implement kernel/router flow control and clock domain crossing.

The NI kernel has the uphill task of receiving and providing packets, which contain the data driven by the IP modules via their protocol after sequentialization. The packet structure may vary depending on the protocol used

[1] Legacy interfaces are a pain for any semiconductor company building networks on-chip. Every NoC designer would like tossing these interfaces of old out of the way, but time-to-market constraints and IP re-use definitely work against brand new network interfaces, encouraging legacy hardware.

by the IP module. However, the packet structure is irrelevant to the kernel NI, as it just sees packets as pieces of data to be transported over the NoC. The NI kernel communicates with the NI shell through memory buffers. A synchronization block ensures proper operations when frequency conversion is performed. The shell NI acts as a slave/master for master/slave IPs.
A key feature of the NI consists in driving all of the IP controls, according to the IP protocol specification (e.g. STBus). As the end-point of the communication, the shell NI has to manage advanced protocol issues such as write posting mechanisms, quality of service and out-of-order management.
To be noted that the shell NI is the only component whose behavior can bias somehow routers behaviors in terms of quality of service requirements. The NI is the entry point of the network, hence in some broad sense "the service provider" (see subsection 4.4.4 on page 67 in this chapter). According to the interconnection specification, the designers of STNoC™ can tune the NI through specific "knobs" in order to let it meet the IP requirements. This simple programming can be done either statically or dynamically through registers address space. In figure 4.15 is reported the structure of STNoC™ packet omitting low level details.

Segmentation and reassembly are very common operations in on-chip networks. Let us consider an example. An AHB bus protocol defines an end-

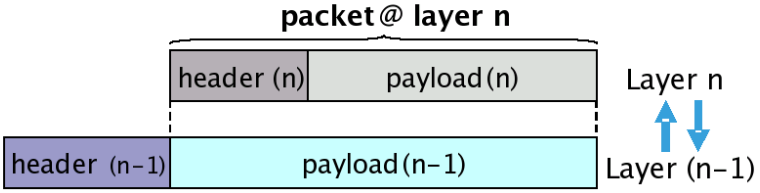

Figure 4.15: STNoC™ packet layering

-to-end protocol specification through which IPs communicate. Referring to figure 4.9 on page 72, the header part of a transport layer packet is defined as a bitmap of AHB control signals crammed (so sequentialized) in a header flit. This header field at *transport* layer becomes a payload at network layer. The kernel NI fills the network layer header in order to build up a proper packet suitable to be forwarded to the NoC.
This layering within the network interface fosters modularity of the design. The kernel NI can be easily reused throughout different interconnection designs, with nary a change. Only the transport layer protocol changes (of

course just if the IP interface changes), so does the shell NI in order to adapt the two parties (network and IP) no longer matched.
The STNoC™ NI has been designed for both STBus and AMBA AXI end--to-end on-chip protocols, with little area and avoiding CAM (Content Addressable Memory) to manage transaction ordering and addresses alignment issues.

4.7 Conclusion

This chapter went through several design concepts applied to on-chip interconnection networks. Firstly, communication layers were described as well as their implications on interconnection design patterns. Networks on-chip foster a layered protocol stack that ultimately allows decoupling of communication and computation parts of the chip, improving the whole design methodology. Moreover, advanced arbitration schemes and protocol issues were presented in order to underline real challenges that are looming large for system on-chip designers. As long as possible due to confidentiality reasons, STNoC™ network on-chip architecture was described, highlighting how theoretical concepts such as layering apply to real network design.
Given the general background provided so far, the following chapter presents STNoC™ network on-chip methodology and design flow, with a clear focus on our modeling methodology, OCCN, developed on-purpose for system level design of the new STMicroelectronics network on-chip. To the best of our knowledge, transaction level models (TLMs) of STNoC™, developed within OCCN, represent the first existing example of clock-accurate TLMs of on-chip networks.

Chapter 5

Networks On-Chip Modeling: Application to STNoC™

5.1 Introduction

Modeling, especially at system level, has become a fundamental step on building systems on-chip. This chapter, the core of the whole thesis work, deals with system and platform modeling of on-chip networks by providing an in-depth overview of simulation and modeling techniques integrated in OCCN, a freeware, networks on-chip simulator available under sourceforge, developed internally by STMicroelectronics. Firstly, a comprehensive description of the SystemC kernel is reported in order to provide basic simulator semantics. Afterwards, OCCN details of STNoC™ models are unfolded, together with a wealth of code snippets useful to grasp modeling principles. Finally, an in-depth overview of advanced simulation techniques such as distributed simulations are reported in order to show our interesting achievements in this field. In particular, the SystemC kernel has been thoroughly ported to an SMP aware configuration, which allows concurrent simulation execution on multiple processors at a given time. Section 5.8 reports all the development stages as well as benefits for networks on-chip simulations, which is the

subject this chapter is all about. STNoC™ SystemC models are described step-by-step all along this chapter, including interesting titbits very useful in modeling context.

5.2 System level design

The emergence of the System-On-Chip era is bringing about many new challenges at all stages of the design process. Designers have to face complex designs; hence a complete revision of systems specification, verification and partitioning is in order. The ITRS roadmap for design technology correctly identifies design productivity as the key hurdle in keeping up with the technology advances. In order to keep abreast with technologies advances, a "small" team of engineers must continue to be able to design a leading-edge chip in a restricted short time. There are two categories of productivity issues:

System complexity issues that rise from handling the sheer size of the SoC. Moore's law governs scalability properties of nowadays systems. Over the past decade, Electronic Design Automation tools such as RTL synthesis improved designer's productivity from 4Kgates per manyear to over 100Kgates per engineer year. Although significant, this rate is insufficient to keep up with Moore's law. Design re-use and new hierarchical technologies are also required to maintain the productivity pace.

Silicon complexity issues are related to the manufacturing technology. This includes device and interconnect parasitics, physical and electrical design rules, device reliability and process variability. In this way silicon complexity is the outcome of the underlying physics that enlarges system complexity. In sum, silicon complexity increases the number of steps in the design flow.

The rapid evolution of *Electronic System Level* (ESL) methodology challenges productivity issues through a new design paradigm defined as *system level design* [43]. System level design focuses on the functionality and the relationships of the primary systems components, separating system design from implementation. Low-level implementation issues greatly increase the number of parameters and constraints in the design space, thus extremely complicating optimal design selection and verification efforts. Similar to near--optimal combinatorial algorithms, ESL models effectively prune away poor

design choices by identifying bottlenecks and focus on closely examining feasible options.
Abstraction is a powerful approach for design and implementation of complex systems. In order to deal with less and less details, system level design fosters a brand-new methodology which relaxes models constraints.

Network-on-chip paradigm brought about many new challenges for SoC simulations, ranging from simulation speed to models distributeness. Networks made up of several routers and network interfaces further load already burdened simulations, leading to poor CPU load efficiency. From this perspective, on building a Network-On-Chip simulator [77], a good balance must be found in order to endow models with an adequate level of abstraction to meet simulation targets, getting rid of pointless details that can be considered furtherly in successive design steps. To specify, design and implement such complex Networks on-chip, simulators developers are compelled to move on from the Hardware Description Languages (HDLs) of old to simulator able to cope with system level design requirements in a standardized manner. This was one of the main driving factor that pushed the confluence of many streams ideas into Open SystemC Initiative (OSCI) SystemC language [72][71], a full-fledged C++ simulator for system level design.

SystemC (see section 5.3) has become the *de-facto* mainstream design environment for hardware and software constructs. The reason is clear: the design complexity steep increase driven by Moore's law [52] demands fast and accurate executable specifications to validate systems concepts and only C/C++ languages can provide adequate levels of abstraction, hardware-software integration and performance.

5.3 SystemC environment

One of the primary goals of SystemC is to enable system level modeling – that is, modeling of systems above the RTL level of abstraction, – including systems which might be implemented in software or hardware or some combination of the two. One of the challenges in providing a system level design language is that there is a wide range of design models of computation, design abstraction levels and design methodologies used in system level design. To address this challenge in SystemC, a small but very general purpose modeling foundation has been added to the language. On top of this language foundation more specific models of computation, design libraries and modeling guidelines can be added, leaving room for further extensions

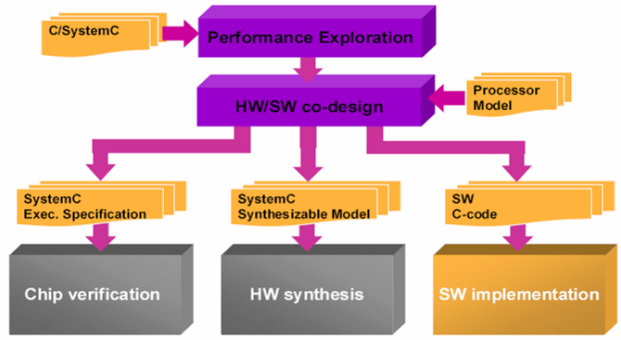

Figure 5.1: SystemC flow

(see figure 5.1). SystemC empowers designers with an environment suitable to build a mixture of models of computation.
The notion of model of computation (MOC) is fundamental to system level design. A *model of computation* could be defined as follows [33]:

1. The model of time employed (real-valued, integer-valued, untimed) and the event ordering constraints within the system (globally ordered, partially ordered, etc.)

2. The supported method(s) of communication between concurrent processes.

3. The rules for process activation.

Most "traditional" languages such as VHDL, Verilog, and SDL can be seen as having a single fixed model of computation, and provide little or no way for users to customize the given model. In this sense, SystemC also has a single fixed model of computation, but it is different from other traditional design languages in several key points:

- The base model of computation is designed to be extremely general.

- The SystemC language as a whole is designed so that customized models of computation can be efficiently layered on top of the base capabilities provided by the SystemC core language.

Some well-known models of computation which can be naturally implemented in SystemC include:

- Static multirate dataflow
- Dynamic multirate dataflow
- Kahn process networks
- Discrete event as used for:
  - RTL hardware modeling
  - network modeling (e. g. , stochastic or ”waiting” room models)
  - transaction-based SoC platform modeling

The Register Transfer Level (RTL) MOC is an acronym for a modeling style that corresponds to digital hardware synchronized by clock signals. This modeling style is deployed within languages such as Verilog and VHDL, and it is by now widely integrated in commercial hardware synthesis tools. In the RTL style, all the models interfaces are connected through *signals*. Processes can either model sequential logic, in which case they are fired off by a clock edge (positive or negative), or they can model combinatorial logic, which means that a change of an input to which the process is sensitive to causes process execution. RTL models are *pin-accurate* and *cycle-accurate*, meaning that the ports of an RTL modules correspond to wires in the hardware implementation of the module. SystemC signals and VHDL signals

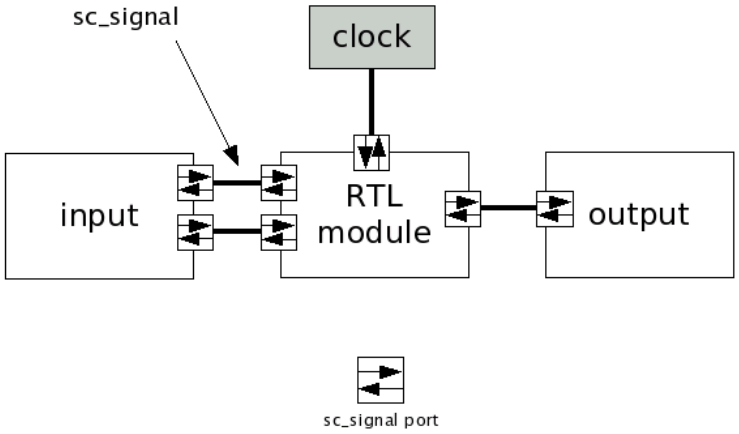

Figure 5.2: SystemC sample RTL model

behaviors are identical. At Register Transfer Level the precise connections

of registers are explicit at the structural boundaries (see figure 5.2). These explicit connections impose hard constraints on the methodology used to describe modules interfaces. The restrictions placed on interfaces represent somewhat limiting factors in terms of modeling methodology; in particular, as the interfaces level of abstraction plays a pivotal role in models ease-of-use, adaptation and simulation speed, RTL level of abstraction can be definitely considered *verboten* for system level design. Due mainly to complexity reasons, Networks on-chip modeling requires a high-level approach for models interfaces, implying the need of raising abstraction level.

Transaction Level Modeling (TLM) represents one specific type of the discrete-event MOC. In TLM, communication between modules is abstracted through function calls that execute transactions defined in a given target platform. TLM has become a de-facto standard for system level modeling; a thorough explanation of its concepts requires a brief introduction of the SystemC kernel which starting from version 2.0, integrates the basic components to create full-fledged TLM of systems.

5.3.1 SystemC Kernel

The description of a simulator such as SystemC does not differ much from whatsoever large program, application or operating systems. Indeed, SystemC can be described with well-known concepts in computer science, the so-called user level and kernel level. In this section kernel level is described. Unraveling the SystemC kernel represents a head start for SystemC platform developers; simulation speed, models testing and correctness, all depend on a thorough understanding of the underlying simulator core.
The primary purpose of the SystemC kernel is to trigger or resume the execution of the processes that are supplied by the user as part of the application. The scheduler is event-driven, meaning that the processes are executed in response to the occurrence of events. In SystemC, simulation time is an integer quantity. It is initialized to zero at the start of simulation, and increases monotonically. Similar to VHDL and Verilog, the SystemC scheduler supports delta cycles. A delta cycle is comprised of separate evaluate and update phases (see figure 5.3), and multiple delta cycles may occur at a particular simulated time. Delta cycles are useful or modeling fully-distributed, time-synchronized computation as found for instance in RTL hardware.

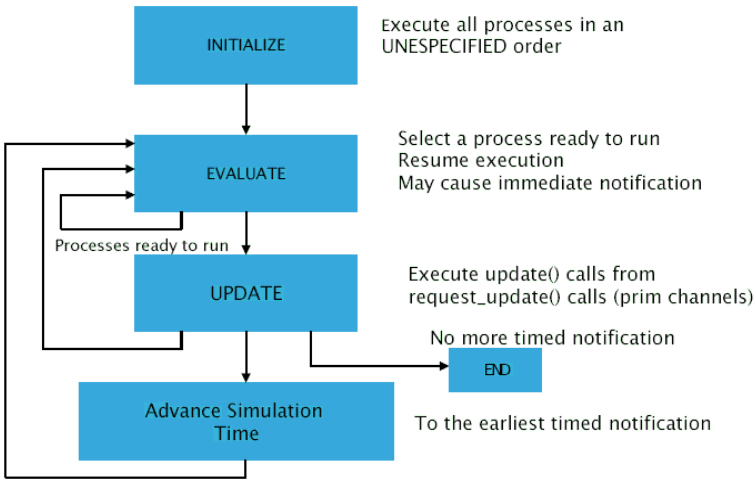

Figure 5.3: SystemC scheduler scheme

The entities that are visible and scheduled by SystemC kernel can be listed as follows:

- The set of runnable processes
- The set of update requests
- The set of delta notifications and time-outs
- The set of timed notifications and time-outs

Within SystemC environment channels can be modeled through two structures: *primitive channels* and *hierarchical channels*. A *primitive channel* is one that supports the *request-update* method to model its behavior. The request-update method of access is designed for simulating concurrency. For instance, when simultaneous actions are performed (e. g. changing the value of a signal) it delays any changes of the channel internal state to avoid indeterminacy. Primitive channel expressiveness is somehow limited by its internal construct, that is, the request-update scheme represents what the channel is all about; no SystemC structures (e.g. SystemC *SC_THREAD*) are allowed in it.
To model networks routers and network interfaces (NI), a more expressive form of modeling is required. Through concepts such as ports and interfaces, SystemC provides a modeling construct dubbed *hierarchical channel*; it is a

far more powerful description than a primitive channel, as it allows to define and use constructs such as threads and methods to model channel behavior (e.g. arbitration).
Given this brief introduction on channels, the semantics of SystemC scheduler execution can be sketched as follows:

1. *Initialization phase* - Execute all processes in an unspecified order.
2. *Evaluation phase* - Select a process that is ready to run and resume its execution. This may cause immediate event notifications to occur, which may result in additional processes being made ready to run in this same phase.
3. If there are still processes ready to run, go to step 2.
4. *Update phase* - Execute any pending calls to update() resulting from request_update() calls made in step 2.
5. If there are pending delayed notifications, determine which processes are ready to run due to the delayed notifications and go to step 2.
6. If there are no more timed notification simulation is finished.
7. Advance the current simulation time to the earliest pending timed notification.
8. Determine which processes are ready to run due to the events that have pending notifications at the current time. Go to step 2.

An in-depth introduction of SystemC kernel was in order before starting to describe the basic features of Transaction Level Modeling, OCCN, an open source environment for modeling on-chip networks and distributed simulations for Networks On-Chip.

5.3.2 SystemC groundwork for transaction-level modeling

The primary objective[1] of SystemC 2.0 was to enable system level design through a wealth of primitives aimed at enhancing the three main constructs of any SystemC design, to wit:

[1]SystemC 1.0 was mainly focused on pure hardware constructs such as those of HDLs.

- Modules
- Channels
- Interfaces

In particular, the generalization of channels to hierarchical channel greatly simplified the development of Transaction Level Model of interconnection systems, through the capability of a neat decoupling between communication and computation design partitions [65]. In SystemC `modules` (a.k.a. SC_MODULE) are the basic building blocks to partition a design. Modules allow designers to break complex systems into small manageable pieces. A modularized design aims at hiding internal data representation and algorithms for other modules.
Through modules, designers succeed in developing *hierarchical* design. Modules help outline the *structure* of the system, whilst `processes` (a.k.a. SC_THREADS or SC_METHODS) provide the *functionality*. Processes and functions identify to the SystemC kernel and get called whenever signals these processes are "sensitive to" change value.
Electronics systems are inherently parallel with lots of parallel activities constantly taking place. SystemC has the concept of methods, threads and clocked threads to model parallel activities of a system.
Main difference between threads and methods is about the "context" of the process. Methods are plain functions, so that they have no context, whilst threads (i.e. user level co-routines such as quickthreads [42] or fibers (Windows)) do have a context, meaning that they are able to save system state.
The last basic brick of SystemC environment is the `interface`, that is, the access mechanism of a module. Modules, processes and interfaces are the basic building blocks through which designers can develop Transaction Level Models of SoC platforms.

5.4 Transaction Level Modeling (TLM)

Transaction-level modeling is a high level methodology used to model digital systems where details of communication among modules are separated from computation architecture subsystems details. Communication mechanism such as on-chip networks are modeled as channels (primitive or hierarchical), and they export to modules function calls through SystemC interface classes. Transaction requests are executed calling interface functions defined and implemented in channels, which encompass low-level details of the data

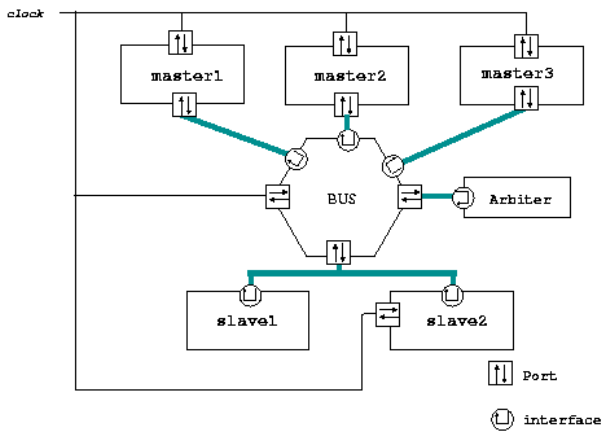

Figure 5.4: SystemC Simple Bus TLM structure

exchange.
The *simple bus* example (see figure 5.4) shipped with OSCI SystemC library is a stripped down but still complete Transaction Level Model of a Bus. The `simple_bus` represents a stepping stone[2] for TLM developers, because it defines the three basic interfaces that turn out to be useful when describing systems at this level of abstraction:

- *Blocking interface*

- *Non-Blocking interface*

- *Direct Access interface*

A blocking master interface such as `simple_bus_blocking_if` which follows is used by high-level software models that generates `burst_read` and `burst_write` on the bus during their execution. Normally, these software models run "natively" on the host processor and they are mostly employed for software debugging early in the design cycle. They are not cross-compiled, because this would imply the usage of an Instruction Set Simulator (ISS) to execute the code, which causes a substantial computational overhead.

[2]The up-and-coming TLM OSCI standard corresponds to its natural evolution.

```
class simple_bus_locking_if
 : public virtual sc_interface
{
public:
virtual simple_bus_status burst_read(
                    unsigned int unique_priority,
                    int *data,
                    unsigned int start_address,
                    unsigned int length=1,
                    bool lock=false)=0;

virtual simple_bus_status burst_write(
                    unsigned int unique_priority,
                    int *data,
                    unsigned int start_address,
                    unsigned int length=1,
                    bool lock=false)=0;

};
```

simple_bus_blocking_IF

The non-blocking master interface is used by masters which are not allowed to block. If several masters drive requests concurrently, an arbitration takes place and the transaction completion time is not bounded. After calling the method, a non-blocking master can call a given method (`get_status()` in `simple_bus`) on subsequent clock cycles to poll the bus in order to check for transaction completion. The non-blocking interface is usually used by processor models (i.e. ISSes). These models are not allowed to block because they must be activated on each clock cycle to simulate proper processor behavior (e.g. pipeline).
The direct interface provides instantaneous access to slaves. The accesses are routed through the bus just because a routing scheme (i.e. address decoding) can be used to select and enable the slave module. When a direct access is performed, the SystemC scheduler does not play any role, so that simulated time does not advance and no arbitration occurs.
A key use of direct interface is for running ISS debuggers, which should get instantaneous snapshots of windows in memory (i.e. slave modules). The `simple_bus`, as already mentioned, defined the guidelines for transaction--level modeling. The next subsections will provide additional insights about TLM state of the art, with a particular focus on TLM OSCI standard.

5.4.1 TLM State of the art

The TLM modeling effort undertaken by Open SystemC Initiative fostered a new way for modeling systems at a higher level of abstraction than RTL. TLM concepts and features are by now well-established and integrated in a number of design environments worldwide both from CAD vendor (e.g. Synopsys (SystemStudio)[70] , ARM(Realview / MaxSim)[7], Coware(ConvergenceSC

)[19], Cadence(NCSIM)[12], semiconductor companies (STM (OCCN, TLM-_INFRA)[17], Sonics (OCP-IP Channels))[36] and academical works (S.Malik et al. (OCCA)[81], A.A.Jerraya et al.(ROSES)[16], D.D.Gajski (SpecC))[27] just to mention a few.
To define a proper TLM state of the art, a tight-knit group of engineers from Nokia, Texax Instruments, Synopsys and Sonics defined, through a white paper (see [36]), the by now renowned TLM communication layers, with the objective of standardizing TLM interfaces. TLM Layering is a subject of the utmost importance in system level design. TLM layering is supported through interface based design [64]. The interoperability of models from different abstraction layers can be implemented with adapter components such as layer-wrappers. Figure 5.5 has the worth of summing up the very sparse definitions of TLM defined in literature. *Message Layer* could be defined as a

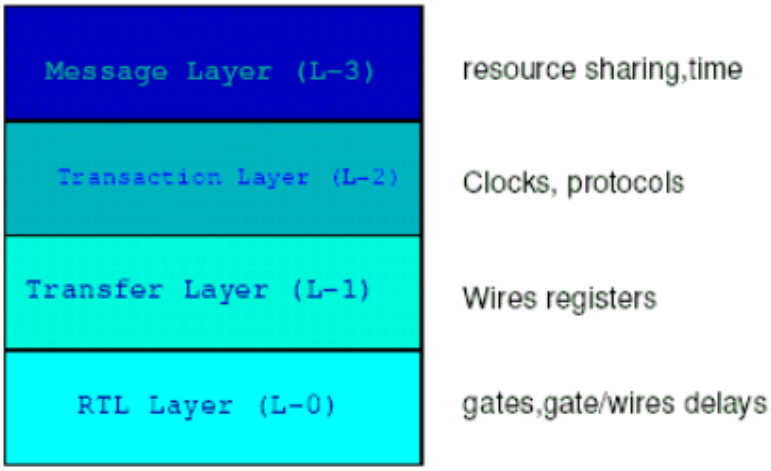

Figure 5.5: TLM stack of communication layers

completely functional level of modeling. Models at this level are untimed and simulation event-driven. A sample transaction between initiator and target implies the transfer of several data, which may be of very abstract data type. RTL has been already mentioned; its purpose corresponds to SoC hardware design. It is not handy when it comes down to modeling software components and behaviours.
L-1 and L-2, that is, *Transfer Layer* and *Transaction Layer*, strictly draw a line between models whose simulations are event-driven and those whose

behaviour is clock-accurate[3].
Event driven models are mainly oriented towards software debugging where *just event ordering is meaningful*. Clock-accurate models, still not useful for purposes such as synthesis, allow detailed architecture exploration and in-depth performance analysis. Strictly speaking, this simple distinction represents the essence of TLM layering and suffices to clearly identify a State-of-the Art for TLM. Models cited in literature belong to one of the two TLM layers[4], depending upon the context and modeling objectives.

5.4.2 TLM OSCI standard

The main shortcoming of TLM methodology introduced by OSCI was the lack of fixed guidelines for TLM models definition. While an RTL model has well-defined and fixed set of rules by construction (due to synthesis constraints), TLM models basic constructs leave room for different programming styles and most importantly different *interfaces*. Hence, a TLM standard was in order, in particular to boost Intellectual Property (IP) models exchange, a key factor for IP-reuse and an effective system level design.
OSCI TLM standard [62] takes into account different facets of system level design:

- Providing an early platform for software development
- System level design exploration and verification
- The need to use system level models in block level verification

TLM standard defines three key concepts:

- Interfaces
- Blocking and Non-Blocking
- Bidirectional vs Unidirectional

The emphasis on interface is due to the object oriented nature of C++ [68][22][25] language and to the need for decoupling communication and computation parts of design. As already stated, in SystemC there are two types of processes (SC_THREADS and SC_METHODS). Threads have a context, so

[3]It is worth noting that the SystemC MOC enables both types of simulations.
[4]CAD vendors usually provide both models in their tools.

OSCI Terminology	Contains wait()	Can be called from
Blocking	Possibly	SC_THREAD only
Non Blocking	No	SC_METHOD or SC_THREAD

Table 5.1: OSCI terminology

that they can call `wait()` macro to suspend and save their state (see [42] for additional details about SystemC threads implementation in Unix). Methods are plain functions, so more efficient but more convoluted to use. Thus, OSCI defines interfaces as in table 5.1. The third definition concerns transactions direction. An example will clarify this concept. Common transactions over a shared bus are bi-directional (e.g. load/store). Other are unidirectional, such as packet flow through a network.
The remainder of this subsection describes unidirectional interfaces defined by OSCI standard, because they have been used for modeling of STNoC™ networks On-chip. The following code snippet reports the signature of blocking unidirectional interfaces defined by OSCI standard:

```
template < typename T >
class tlm_blocking_get_if :
public virtual sc_interface
{
public:
  virtual T get( tlm_tag<T> *t = 0) = 0;
  virtual void get(T &t) { t = get();}
};

template < typename T >
class tlm_blocking_put_if :
public virtual sc_interface
{
public:
  virtual void put(const T &t) = 0;
};
```

Unidirectional Blocking Interface

Since TLM standard allows modules to call wait in the blocking function, they never fail. For convenience (and with a simple C++ trick) several `get()` signatures are provided, using a dummy `tlm_tag<T>` template object in order to endow the slave interface with multiple `get()` function calls. The choice of `put()` and `get()` as function names is due to the overloaded nature of instructions such as *read* or *write* that do not ensure any generalization of semantics for control and data movement. These functions must be called by threads (they are supposed to block through `wait` calls) otherwise the function call would simply crash the system.
The non-blocking interface (code snippet embedded which follows) may fail, since they are not allowed to wait for the correct conditions for these calls to succeed. Hence `nb_put` and `nb_get` must return a bool to indicate whether

the non-blocking access succeeded.

```
template < typename T >
class tlm_nonblocking_get_if :
public virtual sc_interface
{
public:
  virtual bool nb_can_get( tlm_tag<T> *t = 0) const  = 0;
  virtual bool nb_get(T &t) { } = 0;
  virtual const sc_event &ok_to_get( tlm_tag<T> *t = 0) const = 0;
};

template < typename T >
class tlm_nonblocking_put_if :
public virtual sc_interface
{
public:
  virtual bool nb_put(const T &t) = 0;
  virtual bool nb_can_put( tlm_tag<T> *t = 0) const = 0;
  virtual const sc_event &ok_to_put( tlm_tag<T> *t = 0) const = 0;
};
```

Unidirectional non-blocking Interface

OSCI standard defines also `nb_can_put` and `nb_can_get` to poke the channel without moving any data. These methods are sufficient to do polling puts and gets.
Models of STNoC™ Network Interfaces sport successfully unidirectional interfaces, providing the end users with simple and standard function calls to access the communication medium. STNoC™ Network interface models provide also wrappers to standard RTL interfaces, proving in this way a nice flexibility of the approach that TLM OSCI standard enabled.

5.5 OCCN: On-Chip Communication Network

The On-Chip Communication Network (OCCN) provides an efficient, open--source, GNU-GPL licensed framework, developed within Sourceforge for the specification, modeling, simulation, and design exploration of Networks--On-Chip (NoC) based on an object oriented C++ library built on top of SystemC. OCCN was shaped by Advanced System Technology (AST) laboratory experience in developing communication architectures for different Systems on chip. OCCN increases the productivity of developing communication adapter models through the definition of an universal communication Application Programming Interface (API)[17]. This flexible API provides a new design pattern that enables creation and reuse of executable transaction level models across a large variety of SystemC based environments and simulation platforms. It also addresses model portability, simulation platform independence and high-level performance modeling issues.

5.5.1 OCCN methodology overview

As all systems development methodologies, any SoC object oriented modeling would consist of a modeling language, modeling heuristics and a methodology. Modeling heuristics are informal guidelines specifying how the language constructs are used in the modeling process. Thus, the OCCN methodology focuses on modeling complex networks on-chip communication by providing a flexible, open-source, object-oriented C++ based library built on top of SystemC. Alike OSI layering, OCCN methodology for NoC establishes a conceptual model for inter-module communication based on layering, with each layer translating transaction requests to a lower level communication protocol. As shown in figure 5.6 OCCN methodology defines three distinct

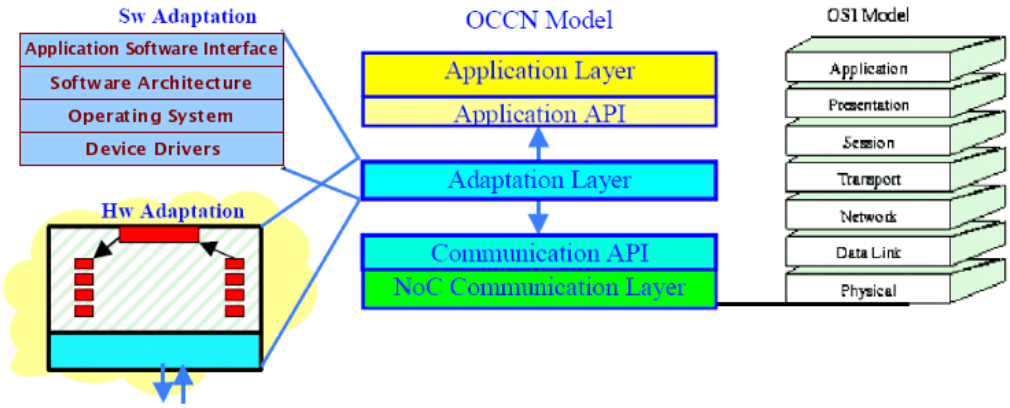

Figure 5.6: OSI like OCCN layering model

OCCN layers. The lowest layer provided by OCCN called *NoC Communication Layer*, implements one or more consecutive OSI layers starting first by abstracting the physical layer.
For instance, the communication layer of STNoC™ router model abstracts physical, link and network layers. On top of the OCCN protocol stack, the user-defined *application layer* maps directly to the application layer of the OSI protocol stack. Sandwiched between the application and NoC communication layers lies the *adaptation layer* that maps to one or more middle layers of the OSI protocol stack, including software and hardware adaptation components. The aim of this layer is to provide, through efficient, interdependent entities called communication drivers, the necessary computation, communication and synchronization library functions and services that allow the application to run. Although adaptation layer is usually user defined, it

utilizes functions defined within the OCCN communication API.

5.5.2 OCCN API and library components

The OCCN implementation for inter-module communication layering uses generic SystemC methodology, e.g. a SystemC port is seen as a service access point (SAP), with the OCCN API defining its service. Applying the OCCN conceptual model to SystemC, we have the following mapping:

- The *NoC communication layer*, is implemented as a set of C++ classes derived from the SystemC `sc_channel` class. The communication channel establishes the transfer of messages among different ports according to the protocol stack supported by a specific NoC.
- The *communication API* is implemented as a specialization of the `sc_port` SystemC object. This API provides the required buffers for inter-module communication and synchronization.
- The *adaptation layer* translates inter-module transaction requests coming from the application API to the communication API.

The fundamental components of the OCCN API are the Protocol Data Unit (PDU), the MasterPort/SlavePort and Master/Slave Interface.
OCCN ports and interfaces are integrated as a superset of basic SystemC ports and interfaces (`MasterPort, SlavePort, MasterIf, SlaveIf`) in order to define environment specific access primitives. A great effort has been made in order to define these primitives as a reduced subset of functions (see figure 5.7), to improve models portability.
The `BusBaseChannel` class constitutes the main building block from which complex models of busses can be built. It contains functions useful at port binding to create dynamically Master and Slave Interfaces.
The paradigm used for sending and receiving PDUs through ports is message passing with send and receive primitives for point-to-point and multi-point communication. In figure 5.8 a simple schematic behavior of how send and receive primitives are implemented is reported, in order to give an idea about data flowing. `Send()` like primitives are used to send Pdus (a.k.a. flits) on the link channel end, and can be seen as active agents from a protocol standpoint. `Receive()` and `Reply()` functions act as slave agents in that they respond to send transactions. All of these primitives make up the OCCN protocol suite. Next chapter will provide more detailed information on send/receive primitives inner workings. It is worth noting that primitives used in ports

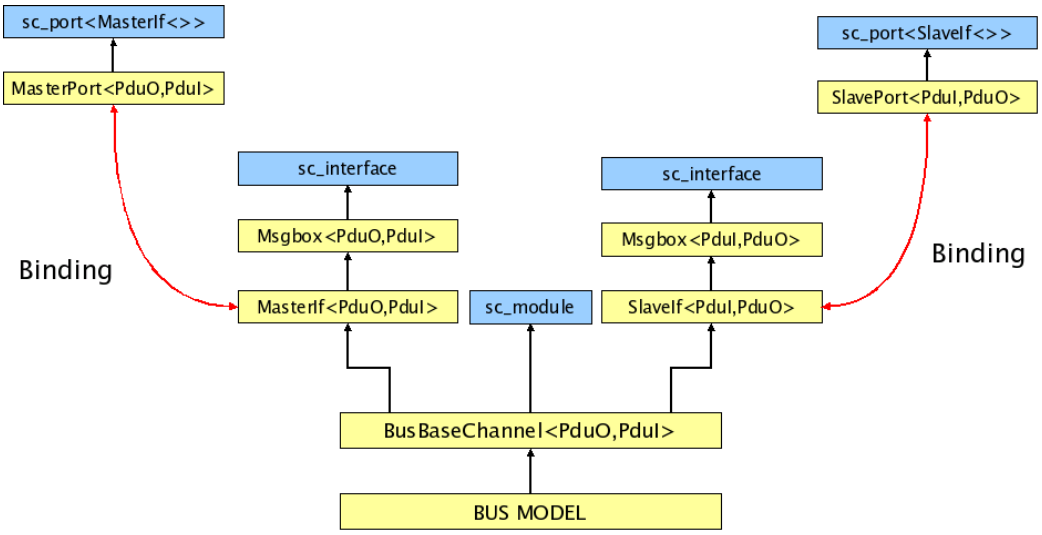

Figure 5.7: OCCN API class hierarchy

and busses (such as `notify_sending_completion()`) are exclusively managed by bus model developer. They are not, or better, they must not be visible to external modules because they represent OCCN internal mechanisms that provide transactions scheduling and release. It is up to models writers to implement them properly in order to achieve expected goals. In particular, they are used to fire off events and flags of blocking (`send()`) and non-blocking interfaces (`asend()`).

In the next subsections the different components of the OCCN API will be explained along with code snippets useful to describe code functionalities.

5.5.2.1 PDU

In OCCN, inter-module communication is based on channels implementing well-specified protocols by defining rules (semantics) and types (syntax) for sending and receiving protocol data units (or PDUs according to OSI terminology). In general, PDUs may represent bits, tokens, cells, frames, or messages in a computer network, signals in an on-chip network or jobs in queuing network. Thus, PDUs are a fundamental ingredient for implementing inter-module (or inter processing element) communication using arbitrarily complex data structures.
Each PDU is usually made up of two different entities:

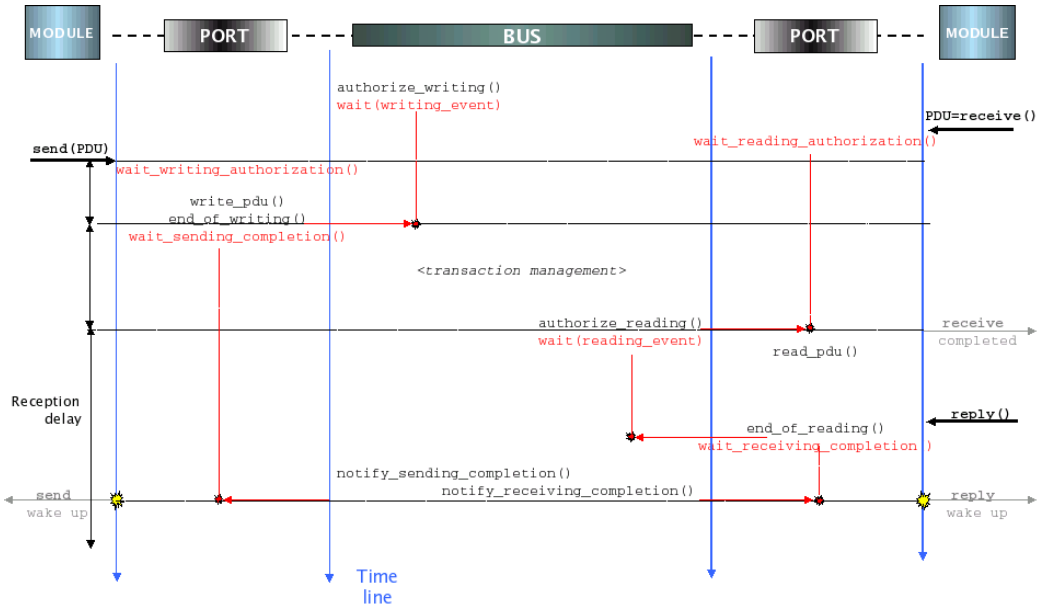

Figure 5.8: Send/Receive protocol example

- The *control* field (template H, also called *protocol control information*) provides destination address(es) and sometimes it includes source address. Moreover control integrates operation code that distinguishes (a) requests/responses, (b) load/store instructions, (c) synchronization instructions, (d) blocking/non-blocking instructions, and (e) system instructions. Sometimes performance related information is included, such as a transaction identify/type and epoch numbers.

- The *data* field (array of BU template type, called *payload*, or *service data unit*) is a sequence of bits that are usually meaningless for the channel.

The PDU Class (see code snippet) provides modeling support for the control and data field:

```
template <typename H, typename BU=H, int size=1>
class Pdu
{
 public:

   Pdu();

   // Assignments modify & return lvalue.
   Pdu& operator=(const BU& right);
   Pdu& operator=(const BU* right);
   BU& operator[](unsigned int x);
```

```
    operator const BU();

    // Conditional operators return true/false:
    int operator==(const Pdu& right) const;
    int operator!=(const Pdu& right) const;

    // std streams
    friend ostream& operator<< <>(ostream& os, const Pdu& ia);
    friend istream& operator>> <>(istream& is, Pdu& right);

    // Pdu streams
    friend Pdu<H,BU,size>& operator<< <> (Pdu& left,const Pdu& right);
    friend Pdu<H,BU,size>& operator>> <> (Pdu& left, Pdu& right);


    template <typename H2, typename BU2, int size2>
    void* operator new(unsigned int sz, MasterPort<Pdu,Pdu<H2,BU2,size2> > *port);
    template <typename H2, typename BU2, int size2>
    void* operator new(unsigned int sz, SlavePort<Pdu,Pdu<H2,BU2,size2> > *port);
    void* operator new(unsigned int sz);

    // for channels implementation like AHB (need for BU OR Hdr transfer only)
    void copy_sdu(Pdu& src);
    void copy_pci(Pdu& src);

  public:
    enum {pci_size = sizeof(H)};
    enum {sdu_size = size * sizeof(BU)};
    enum {pdu_size = sizeof(H) + size * sizeof(BU)};

    union
    {
        struct
        {
            H hdr;
            BU body[size];
        }
        pdu;
        char stream[pdu_size];
    } view_as;

    unsigned int stream_tail;
    unsigned int stream_head;

};
```

PDU Class

The `view_as` union, the actual Pdu class data (control plus data), is a hack implemented to pack data [38] in a way suitable for subsequent segmentation and re-assembly operations on a per-byte basis. In particular, the union can be easily unpacked using the `stream` array which provides a per-byte view of the Pdu class data, as a stream of serial bytes. The Pdu class provides also proxy calls such as `copy_sdu` and `copy_pci` which are useful to copy independently the two instances (hdr, body a.k.a. *pci*/*sdu*) of the Pdu class. They are implemented using normal `memcpy` operations, in order to copy in an effective way a simple stream of bytes. Pdu class also features a memory allocator (*new*) whose role is to allocate a memory arena at simulation start [68][51] to improve heap based allocation of PDUs. Getting a hold of memory becomes a matter of allocating object from the arena without resorting to libc malloc calls (unless arena size is exhausted).
A layered approach like OCCN one allows also packed data encapsulation; in particular PDUs may belong to different layers that make up the network protocol stack. The PDU plays a very important role in OCCN API, with its

structure being determined by the corresponding on-chip communication architecture. In a network on-chip context normally the control field (template H, `hdr` field in the `view_as` union) implements the point-to-point link controls whilst the body field (template BU, array `body` in the `view_as` union) mirrors the data path. It is very important to note that a data path can actually encompass header flits; this notation could result a bit misleading, hence it deserves an example.
In section 5.6 our modeling methodology for a network on-chip router will be detailed. Within that context the control signals crammed on the Pdu class (the H template field) are better described through the `cntrl_flit` struct:

```
typedef struct {
  bool req;
  bool ack;
  bool eop;
  bool sop;
} cntrl_flit;
```

OCCN PDU Controls struct

Throughout the different OCCN models, control variables are always modeled using plain C structs. Modeling control signals through plain variables, the OCCN methodology, as I am going to report, differs from standard RTL modeling techniques where signals are modeled using the `sc_signal` class. SystemC signals are primitive channels, scheduled by the SystemC kernel and triggered by event notifications. These constructs burden the kernel worsening simulation speed, which is why OCCN does not rely on signals, using variables instead. In the router example available within sourceforge http://occn.sourceforge.net (`occn::std_router` in occn_2.0.1beta), a Pdu, namely the class exchanged by routers instances, is declared as follows:

```
typedef Pdu<cntrl_flit,data_flit<unsigned int> > NoC_Pdu;
```

PDU definition example

where the `data_flit` template represents the data sent along the data path wires. The data flit class (see next snippet) just cloaks abstract data types (`meta_data`) used to model header and data flits in an abstract manner. The pointer to the data structure allows to avoid many copies in flit transmission (the pointer is set up for each packet, so that transmitting a packet is a matter of copying a pointer). The controls class (template H of PDU, `cntrl_flit` described early, instantiated in the previous snippet) defines signals which govern the link-to-link protocol.

```
template <typename T>
struct data_flit{
  meta_data<T> *data;
  unsigned int flit_number;
};
template<typename T>
class meta_data
{
public:
    meta_data();
    meta_data(int hdr_size);

    ~meta_data();

    //Overloaded output stream
     friend ostream &operator<< <> (ostream& output,const meta_data<T>& value);

    void set_source(int source_address);

    void set_destination(int destination_address);

    void set_id(unsigned int id);

    void set_data(T data_in);

    unsigned int get_source();

    unsigned int get_size();

    unsigned int get_destination();

    unsigned int get_id();

    T get_data();

    void clean();

private:

    /* Source address of packet*/
    unsigned int source;

    /* Destination address of the packet */
    unsigned int destination;

    /* Number of flit */
    unsigned int size;

    /* This is the packet */
    QueueObject< T > data;
    unsigned int pkt_id;

};
```

data_flit class

The `meta_data` class, along with the actual flits to send in the queue `data`, declares and defines utility functions (inline) that set/poke/poll sensible packet fields such as `source` and `destination`, with function such as for instance `get_destination()`. Additional fields not strictly present in the header flit but useful for debugging can be added at will such as `pkt_id`.

5.5.2.2 MasterPort/SlavePort

The second pillar of OCCN API is represented by *MasterPort* and *SlavePort* (figure 5.7).

```
template<class WPdu, class RPdu=WPdu>
class MasterPort :public sc_port<MasterIf<WPdu,RPdu>,MAX_IF >
{

public:
   MasterPort();

   void bind(MasterIf<WPdu,RPdu>& interface);
   void operator () (MasterIf<WPdu,RPdu>& interface);
   void operator () (sc_port<MasterIf<WPdu,RPdu>, MAX_IF> &port);

   // communication API
   void send(WPdu* pk);          // synchronous blocking call (emission + propagation + reception delays)
   void asend(WPdu* pk);       // asynchronous blocking call (emission delay)
   RPdu* receive();
   void reply();
   void reply(N_uint nb_cycles);
   void reply(sc_time& delay);

   // same with time-out feature
   void send(WPdu* pk, sc_time& time_out, bool& sent);
   void asend(WPdu* pk, sc_time& time_out, bool& sent);
   RPdu* receive(sc_time& time_out, bool& received);

protected:

private:

};
```

MasterPort class

An OCCN Master/Slave Port inherits a plain SystemC port with an interface declared as a `MasterIf`. The Master/Slave Port class defines the primitive visible to the end-user, namely send, asend/ receive and reply functions. These functions represent the upper layer of the OCCN protocol suite, and qualify the Message Passing paradigm. To be noted that the blocking/non-blocking behaviour of these primitives is managed internally in the MasterIf/SlaveIf, and it is up to the channel developer. Thus, put it in a general context, a send primitive should be blocking, the normal OCCN behaviour, but it *has not* to be blocking. As I am going to report in the next subsections, blocking is not always the wished behavior, sometimes polling produces better results, avoiding the usage of multiple threads to manage concurrent hardware behaviours in communication channels. To provide the reader with a code example the next code snippet reports the behaviour of the send primitive :

```
template<class WPdu, class RPdu>
void MasterPort<WPdu,RPdu>::send(WPdu* pk)
{
   //pseudo-code
   ..
   (*this)->wait_write_authorization();
   (*this)->write_pdu(*pk);
   (*this)->end_of_writing();
   (*this)->wait_sending_completion();
}
```

Send primitive

Through the -> C++ overloaded operator, the port class gains access of the related interface, where methods such as `wait_write_authorization()` are actually defined. The Master/Slave Port class is a template on Pdus that characterizes the full-duplex point-to-point link. A point-to-point link is created by the bound of Port and Interface carried out before simulation start. As the reader may easily notice, the next code snippet reports the OCCN port bind function which is actually syntactic sugar to conceal the real binding function of the primitive SystemC port, called using the usual C++ namespace determiner.

```
void MasterPort<WPdu,RPdu>::bind(MasterIf<WPdu,RPdu>& interface)
{
   sc_port_b<MasterIf<WPdu,RPdu> >::bind(*(interface.get_master_if_pointer()));
}
```

Master/Slave Port bind function

The purpose of having standard Ports relies on the increasing need for standard interfaces, in order to speed up the development of common wrappers to the modeling environment. This kind of modeling, where a Pdu is sent in one shot through a function call (i.e. send), defines a modeling style that might be classified as Transaction Level clock-accurate modeling. The reason is twofold: firstly, protocol management is carried out through interface method calls such as send, which is the essence of TLM. Secondly and eventually, the modeling of control and data signals at the bit level (through variables not SystemC signals) permits to control the models dynamic at the bit level, as in RTL simulations. Clock accuracy remains a choice of the model developer, as it relates to the instant of sampling. The bit-accuracy of the models interfaces does not automatically guarantee the clock accuracy of models. In section 5.6, where I describes thoroughly the internals of a clock-accurate model of a router, the network is governed by a thread which samples incoming signals at the positive edge of the clock. This means that the model behaviour follows the clock tick, and OCCN models developed in this way are unquestionably clock accurate even though they do not rely on SystemC signals.

5.5.2.3 Master/Slave Interfaces

Master/Slave Interfaces description wraps up our tour over OCCN basic components. As in all interface method calls based models, the interfaces either directly (through function definitions) or indirectly (through virtual pointer dereferencing) represents the place where the protocol is defined. OCCN

developers made a design decision where the interface clearly draws a line between functions available to the models users and those available to models developers. In the code snippet below the Master/Slave interface declaration is reported:

```
template<class WPdu, class RPdu>
class MasterIf :public Msgbox<WPdu,RPdu>
{
public:
    MasterIf(MasterIf<WPdu,RPdu>* ptr_if=0);
    virtual MasterIf* get_master_if_pointer();
    virtual ~MasterIf();

    // re-routed functions
    virtual void register_port( sc_port_base&, const char*);
    virtual void back_door_write(unsigned int size,unsigned int address,unsigned char *buffer);
    virtual void back_door_read(unsigned int size,unsigned int address,unsigned char *buffer);
    virtual void set_index(N_uint new_index, MasterIf *me=0);
    virtual const sc_time& get_clock_period();

protected:

private:
    MasterIf<WPdu,RPdu> *ref_interface;

};
```

Master/Slave Interface definition

The Master/Slave Interface is mostly a container for the Msgbox object (see 5.7), plus some utility functions for debugging and direct access (i.e. backdoor access which skips channel arbitration for instance, see simple bus direct interface). The MsgBox object (see the stripped down header file) is a sort of hodge-podge where all kind of useful functions are defined and declared.

```
template <class WPdu, class RPdu>
class Msgbox  : public sc_interface
{

public:
    // constructors
    Msgbox();
    ~Msgbox();

    // Access methods typically if Msgbox is used as master
    // module side
    N_int  wait_write_authorization();
    N_int  wait_write_authorization(sc_time& time_out);
    N_int  ask_write_authorization();
    N_uint write_pdu(WPdu& _ref);
    void   end_of_writing();
    N_int  wait_sending_completion();
    N_int  wait_sending_completion(sc_time& time_out);
    void   cancel_sending();

    // channel side
    void authorize_writing();
    bool is_writing_completed();
    void assign_writing_event(sc_event*);
    void enable_writing_event();
    void disable_writing_event();
    void notify_sending_completion();
    void assign_sending_cancel_event(sc_event*);
    bool is_sending_cancelled();
    void reset_sending_cancel();
    void reset_writing_access();
```

```
    // Access methods typically if Msgbox is used as slave
    // module side
    N_int  wait_read_authorization();
    N_int  wait_read_authorization(sc_time& time_out);
    N_int  ask_read_authorization();
    N_uint read_pdu(RPdu* _ref);
    void   end_of_reading();
    N_int  wait_receiving_completion();
    void   cancel_receiving();

    // channel side
    void authorize_reading();
    bool is_reading_completed();
    void assign_reading_event(sc_event*);
    void enable_reading_event();
    void disable_reading_event();
    void notify_receiving_completion();
    void enable_receiving_completion_event();
    void disable_receiving_completion_event();
    void assign_receiving_cancel_event(sc_event*);
    bool is_receiving_cancelled();
    void reset_receiving_cancel();
    void reset_reading_access();

    // data/ctrl members access
    WPdu*   get_write_pdu_ptr();
    RPdu*   get_read_pdu_ptr();
    void    set_write_pdu_ptr(WPdu*);  // shouldn't be used by user
    void    set_read_pdu_ptr(RPdu*);   // shouldn't be used by user
};
```

Master/Slave Interface definition

The core of the interface functionality is implemented in the Msgbox object which contains cycle callable functions to manage communication. The code comments explicitly defines the functions available to access methods such as send or receive and those that should in principle be visible only to the channel side, namely to the model developer. Access functions such as `wait_write_authorization()` are used by primitive such as send to ask channel control and to transfer data. Control functions such as `authorize_writing()` are used explicitly by model developers to control flow of data and protocol timing. Let us suppose that a master module, possibly a Network interface (NIC) which contains a MasterPort has just called a send primitive to send a Pdu to the downstream router (which contains and implements a Master Interface, so a Msgbox) as depicted in figure 5.9. The send primitive orderly calls `wait_write_authorization()`, `write_pdu()`, `end_of_writing()` and `wait_sending_completion()`. To describe OCCN inner-workings I am going to describe these functions step by step.

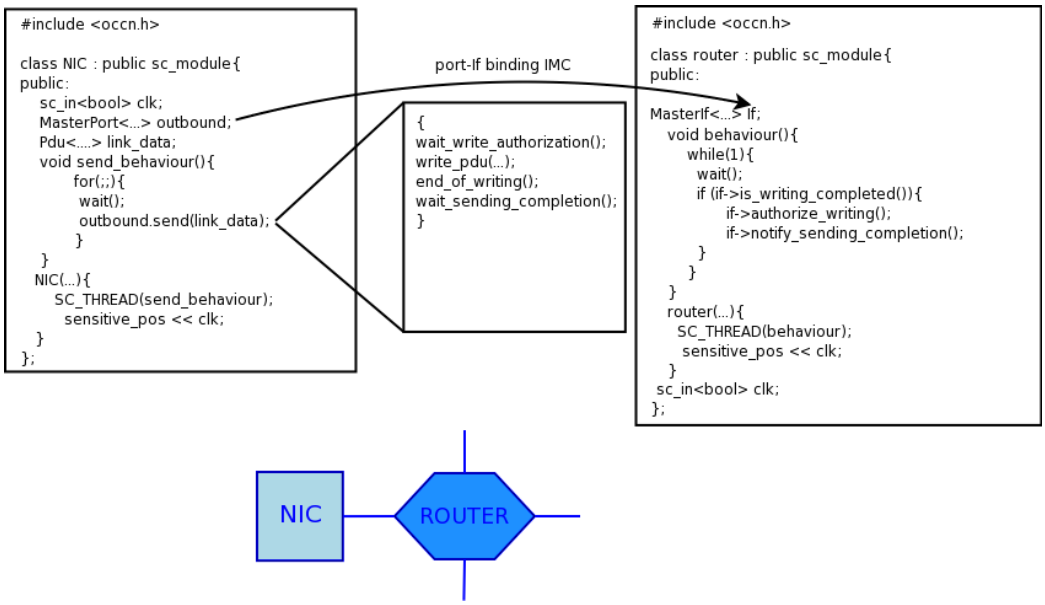

Figure 5.9: Example of protocol implementation through MsgBox functions

```
template<class WPdu, class RPdu>
inline
N_int Msgbox<WPdu,RPdu>::wait_write_authorization()
{
    return (writing_access->lock()==0 ? 1 : 0);
}
```

wait_write_authorization()

```
N_int Mutex::lock()
{
    if (locked)
    {
        // mutex is already locked : we wait for being notified
        sc_event* ev=free_ev.remove();
        if (ev == 0)
        {
            ev = new sc_event();
        }
        lock_candidates_ev.add(ev);
        wait(*ev);
        free_ev.add(ev);
    }
    else
    {
        // mutex free : we take it but maybe with a delay of one delta cycle
        locked=true;
        if (unlocked_during_current_delta_cycle)
        {
            notify_lock_spys(SC_ZERO_TIME);
            wait(SC_ZERO_TIME);
        }
        else
        {
            notify_lock_spys();
        }
    }
```

```
    return 0;
}
```

Mutex lock() function

The `wait_write_authorization()` function try to lock an object called a mutex (`writing_access`). This function may block, when the mutex is already locked, allowing just one thread to call it and to access the channel at a given time. The mutex state is controlled by the router `behaviour()` thread (through interfaces utility functions such as `authorize_writing()`, see figure 5.9) that can unlock the mutex when the protocol allows it. To be noted that the wait calls in the locking function do not imply a blocking behaviour of the mutex; it is up to the model developer to unlock the mutex at given times in order to make the lock call pass with no blocking. In particular, calling an `authorize_writing()` (following code snippet) before a user try to lock the mutex always end up providing a non-blocking behaviour.

```
template<class WPdu, class RPdu>
inline
void  Msgbox<WPdu,RPdu>::authorize_writing()
{
    if (*writing_completed) // current writing was completed by user
    {
        *writing_completed = false; // reset writing status
        writing_access->unlock();
    }
}
```

authorize_writing()

As the reader may have understood by now, it is the scheduling of function calls that determines the models behaviour; a proper scheduling of function calls, as I am going to describe in section 5.6, may provide a pipelined architecture, modeled with these MsgBoxes function calls to be completely non-blocking, still providing a cycle-true behaviour.
The `write_pdu` function is an inline function that just copies the Pdu in MsgBox internal storage. The calling of `end_of_writing()` (code snippet which follows) sets a flag used by the router to check the send primitive completion (through `is_writing_completed()` function).

```
template<class WPdu, class RPdu>
inline
void Msgbox<WPdu,RPdu>::end_of_writing()
{
    *writing_completed = true;
    if (*writing_event_enabled)
    {
        writing_event->notify();
    }
}
```

end_of_writing()

The function `wait_sending_completion()` is a further blocking function that can be used to manage multiple blocking conditions on a single send call. The `notify_sending_completion()` function releases the event on which the send call is blocked.
All of the functions described so far provide a simple but still powerful set of primitives to manage a given point-to-point link protocol in a timely or event driven manner (the event granularity is defined by the threads that calls these functions not by the functions themselves). Through these simple primitives (which end up providing blocking and releasing conditions, and flags to avoid polling) any kind of on-chip protocol can be modeled. To be noted that in this example the router does not check the Pdu fields in order to make decisions. In a real world router such as the one I describe in section 5.6, the router checks control fields (such as *req* in struct `cntrl_flit`, see subsection 5.5.2.1 above) to guarantee the point to point protocol correctness. From this standpoint, the router FSMs can be thought as RTL finite state machines, because the precision in terms of bits and timing may be the same (in case a clocked implementation of primitives is chosen). The only difference consists in choosing in a proper way the instant of sampling of Pdu fields (which models link controls and data signals), which is up to model developers, because plain variables do not guarantee threads order consistency in that variables changes take effect immediately, not after a delta cycle. As SystemC threads or methods ordering is unspecified (and this is correct because it is the way hardware works) this modeling methodology requires some degree of caution to schedule execution of functions that implement e.g. router behaviour (`behaviour()` thread in figure 5.9), as I am going to describe more thoroughly in section 5.6.

5.6 Networks on-chip modeling methodology

Interconnection networks have been becoming the main added-value of current semiconductor technologies. Their complexity in terms of gates, protocols and structure implies an increasing need for modeling abstraction. Networks may easily contain tens of routers and Network Interfaces, impacting in a significant way simulations performance. OCCN and its networks modeling methodology provide a solution to handle this steep increase in interconnection complexity. Unfortunately, due to confidentiality issues, models of STNoC™ router cannot be explained within this thesis. Hence, as a proof of concepts, this section describes in details the models of a simple clock-accurate router, available under sourceforge, endowed with virtual channels

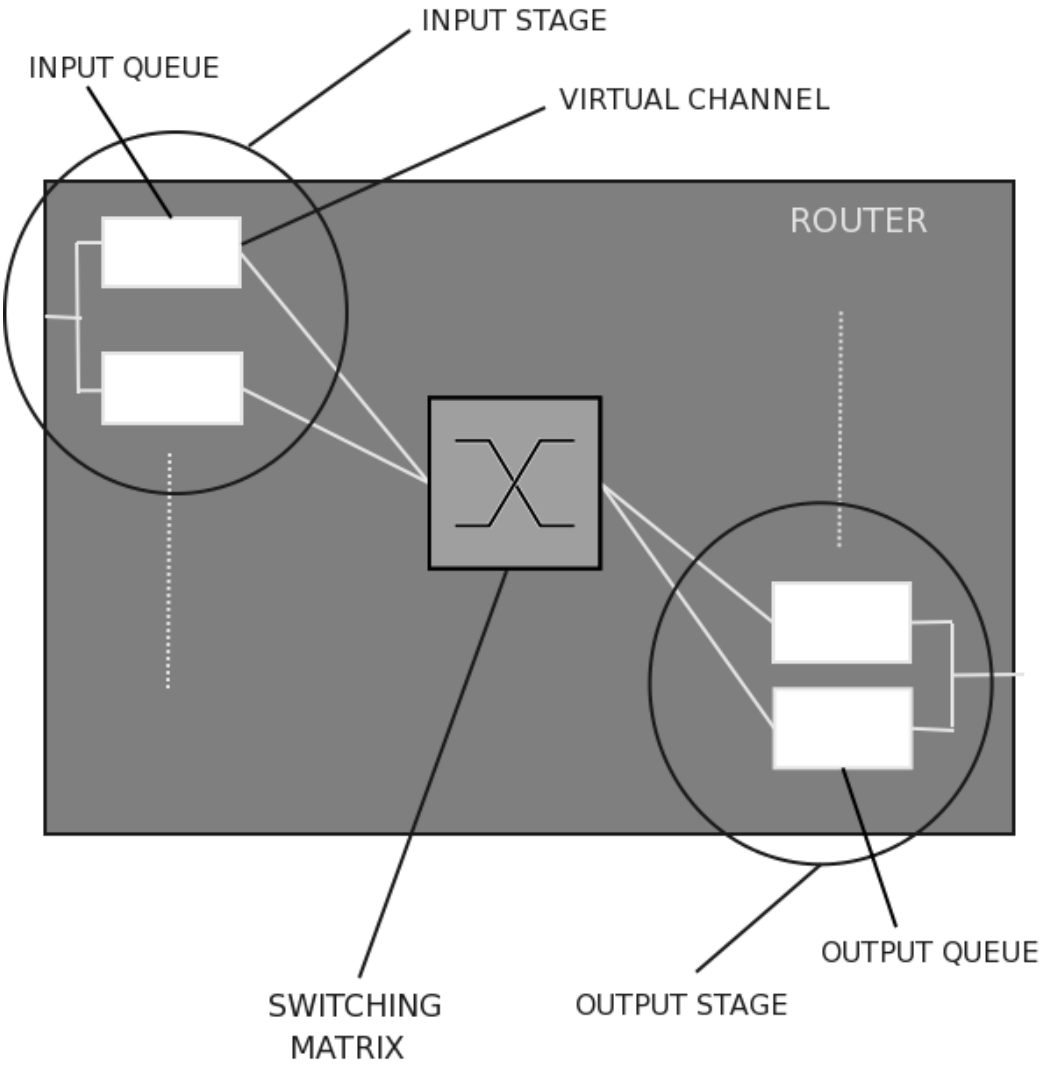

Figure 5.10: Standard router sketched micro-architecture

and two pipelined stages (see pseudo-architecture, figure 5.10), developed through the modeling methodology also deployed in STNoC™ models. The router microarchicture has been kept simple on purpose in order to demonstrate modeling methodologies, not hardware design principles. The input and output queues make up the two pipeline stages. Arbitration of output stages is executed in a round-robin fashion, virtual channel inclusive. Output queues are locked on a per-flit basis, in order to avoid intermingling flits belonging to different packets, which is the essence of wormhole flow control. Link arbitration between virtual channels is performed once again in a round-robin fashion. The reported tests are carried out on a mesh topology, even though other topologies are built in the OCCN network executable. The routing algorithm chosen is X-Y [79], a widespread deadlock-free routing algorithm for 2-D meshes.
The router-to-router and router-to-NI links protocol is shown in figure 5.11. It is based on a simple request acknowledge handshake where the *sop eop* signal are raised respectively when the first flit and the last flit of a packet are driven on the data path. In particular, figure 5.11 shows a hypothetic

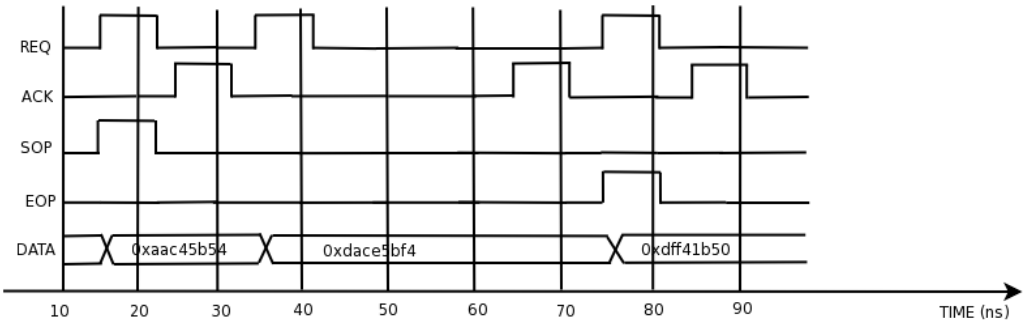

Figure 5.11: Standard router link protocol handshake

protocol handshake for a three flits packet, with a bubble cycle where the acknowledge is not raised within a clock cycle from request by the downstream channel end.
The router model, developed with all OCCN building blocks allows to unfold two advanced simulation techniques. Firstly, as already described, link protocol control and data are modeled using plain variables through PDUs; this allows us to get rid of SystemC signals which somehow burden the SystemC kernel. Secondly and eventually, the router pipeline is scheduled statically; this means that the order in which the different pipeline stages executes is determined by model developer, not by the SystemC kernel through methods and threads sensitivity lists. This second technique requires a thorough un-

derstanding of the router microarchitecture, because, as I am going to prove, the network scheduler (just a SC_THREAD per network, which executes on the positive clock edge) has to call the functions that model the pipeline FSMs in an order that provides a cycle-accurate behaviour of the router, still guaranteeing consistency through different simulations (modeling signals through variables is error prone because changes to signals are visible instantaneously not after a delta cycle; the functions scheduling becomes vital to guarantee the model consistency). A profiling sections will help highlight the main differences between a model which exploits signals and OCCN, along with a simulation speed comparison to gauge the improvements brought by our modeling techniques.

5.6.1 Routers modeling principles

Models of routers must satisfy two basic requirements:

Simulation speed is a major modeling issue when it comes down to router modeling. A router is a basic components of a network so it is replicated a number of times. Hence, its simulation burden must be kept as low as possible.

Parametric models represent a key factor in interconnection exploration; recompilation for huge platforms is definitely frowned upon by system designers.

As usual, these requirements conflict. Dynamic reconfiguration strictly implies models complexity, which in turn causes simulation slow down. A trade-off must be found to achieve both goals. The following code snippet reports the OCCN standard router class declaration.

```
template <typename MPDU,typename SPDU>
class std_router : public BusBaseChannel<MPDU,SPDU>
{
 public:
    //constructor
    std_router(sc_module_name name,...);

    //destructor
    ~std_router();

    MasterPort<MPDU> **out; // communication port
    sc_in<bool> clk;
    SC_HAS_PROCESS(std_router);

 private:

    MPDU ** from_in_to_queue;
    MPDU ** msg_from_in;
    MPDU ** from_input_queue_to_output_queue;
    MPDU ** from_queue_to_out;
```

```
    MPDU ** msg_to_out;
    MPDU ** pdu_for_bw;

    CQueueObject<MPDU *>  **output_queue;
    CQueueObject<MPDU *>  **input_queue;

    N_uint32 router_ports;
    N_uint32 router_in_buffer_max_size;
    N_uint32 router_out_buffer_max_size;
    bool router_VC, ni_wait_for_eop;

    N_uint32 * routing_table;
    N_uint32 * RR_port;

    bool * wait_for_receive;
    bool * output_VC;
    bool * wait_for_eop;
    bool * sending_response;

  public:
    void clock_th();
    void input_th();
    void output_th();
    void input_to_output_th();


    void get_Msgbox_ptr(Msgbox<MPDU,SPDU> **);
    N_uint32 get_bandwidth();
    void reset_bandwidth();
    void create_routing_table(N_uint32);
    void set_routing_table_element(N_uint32, N_uint32);
    N_uint32 get_routing_table_element(N_uint32);
};
```

Standard router class declaration

In figure 5.12, the standard router model scheme is reported. A standard router contains a parametric number of interfaces that are instantiated dynamically at binding time, and an array of master ports which depends on the router arity. The black magic used to create instances of `MasterIf` suitable to be connected to master ports is hidden in the BusBaseChannel class which is the router base class. Each time a master port is bound to the standard router (see subsection 5.5.2.2), a function of the BusBaseChannel is called (`get_master_if_pointer()`, see code snippet), which returns a new interface pointer which is stored in the master port to carry out primitive calls such as send.

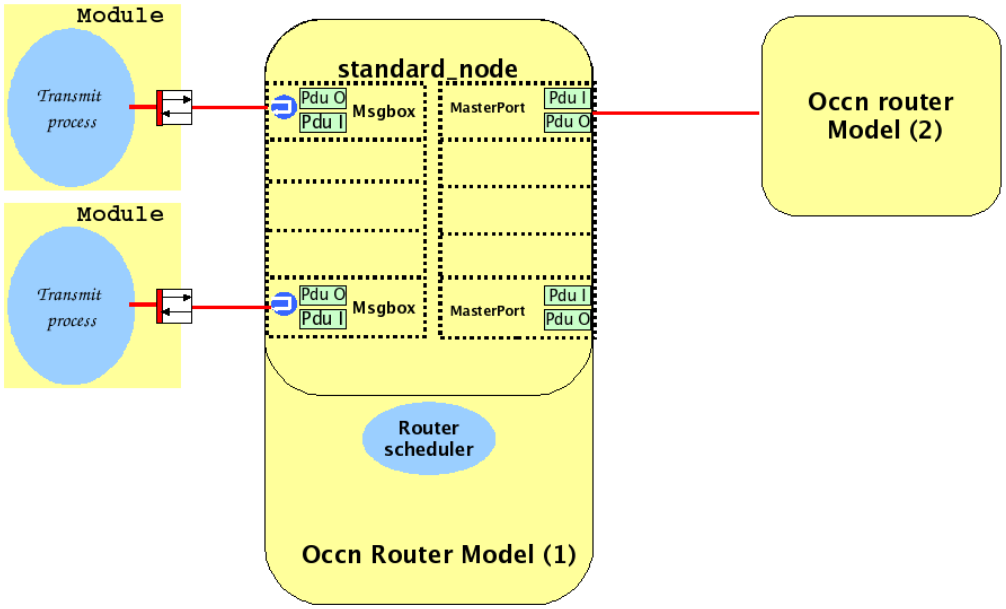

Figure 5.12: Router model infrastructure

```
template<class MPdu, class SPdu>
MasterIf<MPdu,SPdu>* BusBaseChannel<MPdu,SPdu>::get_master_if_pointer()
{
    MasterIf<MPdu,SPdu> *if_ptr = new MasterIf<MPdu,SPdu> (this);

    if(if_ptr == 0)
    {
        OCCN_error_exit("Interface allocation in Router %s failed",name());
    }

    masters.add(if_ptr);

    char tmp[256];
    sprintf(tmp,"%s_master_%d",name(),masters.get_length());
    if_ptr->set_name(tmp);

    return if_ptr;
}
```

BusBaseChannel get_master_if_pointer() function

Once the binding is completed, the routers instantiated a number of interfaces equal to its inbound links a number of `MasterPort` objects equal to its outbound links. As reported in the router class declaration, the router microarchitecture contains input and output queues which represents the two claimed pipeline stages. These queues are OCCN objects (`CQueueObject`) available in the library. Simple routing tables completes the router microarchitecture. Input and output FSMs have been modeled using C++ sequential constructs, as the next section describes.

5.6.2 Pipeline modeling and scheduling

OCCN modeling methodology promotes modeling contructs which are the outcome of lessons learned from years of interconnection modeling experience [13][14][18]. The big challenge in interconnection modeling consists in describing an inherently parallel systems (router have many links all possibly *active* at any given time stamp), with the least number of processes (either threads or methods). Processes cause context switches and function calls; both have strengths and weaknesses. No one-fits-all solution exists.
One of the most important achievement of OCCN router modeling is the complete avoidance of SystemC processes in routers. In OCCN a network (see following snippet) is a class that instantiates the given number of routers and contains a scheduler (`clock_tick()`) which statically calls the routers functions that model their microarchitecture. From this standpoint, SystemC does not play any role in the network scheduling; the order of function calls is determined beforehand and is static throughout the simulation. The `clock_tick()` thread is the only thread in the whole network. This means that through our methodology we can model a synchronous network using just one thread whatever the size of the network is, which is a result of the utmost importance because it is a solution whose complexity is *constant* or O(1) in *Big O* notation (see [47]) in terms of SystemC process control.

```
template <typename MPDU,typename SPDU>
class std_network : public sc_module
{
  public:
     sc_in<bool> clk;

     std_router<MPDU,SPDU>  **my_std_router;

     SC_HAS_PROCESS(std_network);
     void clock_tick();
     std_network(sc_module_name name_,...);

    ~std_network();

   private:
     N_uint32 std_network_size;
     std::string std_network_topology;
     N_uint32 std_network_max_in_buffer_size;
     N_uint32 std_network_max_out_buffer_size;

};

template <typename MPDU,typename SPDU>
std_network<MPDU,SPDU>::std_network(sc_module_name name_,... )
{
  std_network_size = size;
  std_network_topology = noc_topology;
  std_network_max_in_buffer_size = in_buffer_size;
  std_network_max_out_buffer_size = out_buffer_size;

  SC_THREAD(clock_tick);
     sensitive_pos << clk;
  ..
  //pseudo-code
}
```

Standard network class and constructor

The network scheduler function (`clock_tick()`) is not more than a plain loop (see snippet which follows) that scans all of the routers and orderly fires off the different pipeline stages, namely `output_th()`,`input_to_output_th()` and `input_th()`.

```
template <typename MPDU,typename SPDU>
void std_network<MPDU,SPDU>::clock_tick(){
   unsigned int router_index;

   while(1){
      wait();
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->output_th();
      }
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->input_to_output_th();
      }
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->input_th();
      }
   }
}
```

Standard network clock_tick thread

A router could be seen as a hierarchical finite-state-machine. This state machine contains the different pipelines stages (input, switching, output), each modeled with a sequential function (e.g. `input_th()`) which does not block. Blocking is not allowed otherwise the whole technique is broken (if any of the functions blocks, the `clock_tick` scheduler thread is stuck, and it cannot schedule routers which may have something useful to do in the given clock cycle). Each pipeline stage is furtherly decomposed in micro--architecture details [56]; in the standard router, as already stated, these microarchitecture FSMs are modeled using simple state variables. STNoC™ implemented more complex C++ structures, but this does not add anything to the techniques and achievements. From now onwards the modeling of the different pipeline stages of the router will be explained.

```
template <typename MPDU,typename SPDU>
void std_router<MPDU,SPDU>::input_th()
{
  N_uint32 input_channel;
  for(N_uint32 i=0;i<router_ports;i++){
    input_channel=i;
    if (!sending_response[i]){
      if (masters[i]->is_writing_completed()){
        msg_from_in[i]=(masters[i]->get_write_pdu_ptr());
        if (router_VC){
            if (((int)((*(*(msg_from_in[i])).view_as.pdu.body[0].data).get_destination())-(int)((*(*(msg_from_in[i])).
            view_as.pdu.body[0].data).get_source()))<0){
              input_channel=i+router_ports;
            }
        }
        if (input_queue[input_channel]->get_length()<router_in_buffer_max_size){
          from_in_to_queue[i] = new (MPDU);
          *(from_in_to_queue[i]) = *(msg_from_in[i]);
          input_queue[input_channel]->add(from_in_to_queue[i]);
          occn_hdr(*msg_from_in[i],ack)=true;
        }
```

```
        else occn_hdr(*msg_from_in[i],ack)=false;
        masters[i]->authorize_writing();
        sending_response[i]=true;
      }
    }
    else{
      masters[i]->set_read_pdu_ptr(msg_from_in[i]);
      masters[i]->authorize_reading();
      sending_response[i]=false;
    }
  }
}
```

Standard router input_th() function

The `input_th()` function (see previous code snippet) just models the input FSMs of the standard router. Its behaviour is straightforward. The `sending_response` array is a state variable that determines whether the router is sending a response for the given input port or not. If not, it checks the MsgBox flag through `is_writing_completed()` function. A positive test means that the upstream link sent something so the related PDU is grabbed and analysed. In particular, through utility functions such as `get_source()` and `get_destination()` the routing related fields of the PDU data (which mirrors the link data path) are obtained. If there is enough space in the input buffer, the flit is stored in it, and an acknowledge (`ack` field accessed through `occn_hdr` macro, see subsection 5.5.2.1 for control details, struct `cntrl_flit`) is sent back. From this example the reader can understand the cycle-true behaviour of the model. The input pipeline stage is called on a per clock basis, and it manages the link signals in a bit accurate way (e.g. the `ack` signal). On the other hand, if the input stage for the given port is already sending a response (`sending_response[i] == true`), the input stage just copy the acknowledge back (through `set_read_pdu_ptr()`) and unlock the MsgBox mutex through `authorize_reading()` to let the master port read the related PDU. This implies a clock cycle to send an acknowledge back, and it is completely arbitrary as it is an example. The ack could have been sent in the same cycle the request was driven, if microarchitecture worked this way.

The `input_to_output_th()` router function (see following code snippet) models sequentially the router input/output switching. Its behaviour is easy to understand. The state array `wait_for_eop` is used to lock an output queue. Just to recall that this simple router implements wormhole flow control but queues are locked on a per packet basis, namely (a router is barred to intermingle flits belonging to different packets). So, from a queue perspective, a packet must be seen as an atomic data, that is why output queue are locked this way. If the output port is locked (`wait_for_eop == true`), the router just check if the corresponding input queue (through the `RR_port` array) has something to send. If it is the case, it copies the flit from input

to output like any switch. The function also checks if the flit is the last of a packet (through the PDU field `eop` see subsection 5.5.2.1, `cntrl_flit struct`); if it is, it unlocks the output queue `wait_for_eop = false`.
When an output queue is unlocked (`wait_for_eop == false`) the router behaviour is even simpler. It just arbitrates in a round robin fashion (using the `RR_port` array) and if there is any winner (input queue `is_not_empty() == true`) it locks the related output queue (`wait_for_eop = true`) and update the `RR_port` index for the next arbitration step. The `break` C statement allows to stop arbitration when a winner is found (the code bails out of the `for` loop).

```
template <typename MPDU,typename SPDU>
void std_router<MPDU,SPDU>::input_to_output_th()
{

  for(N_uint32 o=0;o<(router_ports*(1+router_VC));o++){
     if (output_queue[o]->get_length()<router_out_buffer_max_size){
       if (wait_for_eop[o]==true){
           if (input_queue[RR_port[o]]->is_not_empty()){
              from_input_queue_to_output_queue[o]=input_queue[RR_port[o]]->check();
              output_queue[o]->add(from_input_queue_to_output_queue[o]);
              input_queue[RR_port[o]]->remove();
              if (occn_hdr(*from_input_queue_to_output_queue[o],eop)==true) wait_for_eop[o]=false;
              else wait_for_eop[o]=true;
              //}
           }
        }
        else{
          if (o<router_ports){
            for(N_uint32 i=0;i<router_ports;i++){
              RR_port[o]=(RR_port[o]+1)%router_ports;
                if (input_queue[RR_port[o]]->is_not_empty()){
                  from_input_queue_to_output_queue[o]=input_queue[RR_port[o]]->check();
                  if (get_routing_table_element((N_uint32)((*(*(from_input_queue_to_output_queue[o])).view_as.pdu.
                  body[0].data).get_destination()))==o){
                     output_queue[o]->add(from_input_queue_to_output_queue[o]);
                     input_queue[RR_port[o]]->remove();
                     if ((*(from_input_queue_to_output_queue[o])).view_as.pdu.hdr.eop==true) wait_for_eop[o]=false;
                     else wait_for_eop[o]=true;
                     break;
                  }
                }
              }
            }
            else{
              for(N_uint32 i=router_ports;i<(router_ports*(1+router_VC));i++){
                RR_port[o]=(((RR_port[o]+1)%router_ports)+router_ports);
                  if (input_queue[RR_port[o]]->is_not_empty()){
                    from_input_queue_to_output_queue[o]=input_queue[RR_port[o]]->check();
                    if (get_routing_table_element((N_uint32)((*(*(from_input_queue_to_output_queue[o])).view_as.pdu.
                    body[0].data).get_destination()))==(o%router_ports)){
                       output_queue[o]->add(from_input_queue_to_output_queue[o]);
                       input_queue[RR_port[o]]->remove();
                       if ((*(from_input_queue_to_output_queue[o])).view_as.pdu.hdr.eop==true) wait_for_eop[o]=false;
                       else wait_for_eop[o]=true;
                       break;
                    }
                  }
              }
            }
        }
     }
  }
}
```

Standard router input_to_output_th() function

Finally, the third and last pipeline stage of the OCCN standard router corresponds to its output behaviour (`output_th()`, see following code snippet). The output behaviour just mirrors the input behaviour of the router. The state array `wait_for_receive` governs the output scheduling. If the router is not waiting any acknowledge (`wait_for_receive == false`) this means it is in a send state. If an output queue is not empty, the flit can be sent. To be noted that the `out` array is an array of OCCN MasterPort, used to govern the link behaviour.

```
template <typename MPDU,typename SPDU>
void std_router<MPDU,SPDU>::output_th()
{
 for(N_uint32 i=0;i<router_ports;i++){
   if (!wait_for_receive[i]){
     if (output_queue[i+(router_ports*output_VC[i])]->is_not_empty()){
       if ((*(out[i]))->ask_write_authorization()){
           (*(out[i]))->set_write_pdu_ptr(output_queue[i+(router_ports*output_VC[i])]->check());
           (*(out[i]))->end_of_writing();
           wait_for_receive[i]=true;
       }
     }
     else{
       if (i!=0) output_VC[i]=(output_VC[i]+1)%(1+router_VC);
       else if (ni_wait_for_eop==false){
         output_VC[i]=(output_VC[i]+1)%(1+router_VC);
       }
     }

   }
   else{
       if ((*(out[i]))->ask_read_authorization()){
           msg_to_out[i]=(*(out[i]))->get_read_pdu_ptr();
           if (occn_hdr(*msg_to_out[i],ack)==true){
             delete output_queue[i+(router_ports*output_VC[i])]->remove();
           }
           out[i]->reply();
           wait_for_receive[i]=false;
           if (i!=0) output_VC[i]=(output_VC[i]+1)%(1+router_VC);
           else if (occn_hdr(*msg_to_out[i],eop)==1){
             output_VC[i]=(output_VC[i]+1)%(1+router_VC);
             ni_wait_for_eop=false;
           }
           else{
             ni_wait_for_eop=true;
           }
       }
   }
 }
}
```

Standard router output_th() function

To guarantee a non-blocking behaviour of the output sequential functions, instead of using possibly blocking function to access the link such as `wait_write_authorization` in send primitive (see subsection 5.5.2.2), the standard router calls `ask_write_authorization()` (see following code snippet). It differs from the former in that it executes a `trylock` on the mutex object, which does not block if mutex locking fails. To be noted that, as already explained, the sequential functions that model the router microarchitecture are not allowed to block otherwise the whole modeling technique is broken. If the output stage is on the receiving path (`wait_for_receive == true`),

the router checks the acknowledge `ack` (using `occn_hdr` macro) and, if positive, it deletes the acknowledged flit from the output queue. The rest of code is about arbitrating between virtual channels for a given output port and does not add anything to the modeling technique.

```
template<class WPdu, class RPdu>
inline
N_int Msgbox<WPdu,RPdu>::ask_write_authorization()
{
   return (writing_access->trylock()==0 ? 1 : 0);
}
```

Msgbox ask_write_authorization() function

The description of the output stage ends the different pipeline stages. One additional remark is in order. The number of clock cycles needed to cross a router from input to output through this modeling and with the router scheduler `clock_tick()` written as previously reported is equal to two. This to let the reader note that the exact timing of the flits is completely determined by the scheduler and stated before simulation start through the static scheduling policy. As an example, if the function calls order in the network is swapped, as in the hypothetic `clock_tick()` thread which follows, it takes just a clock cycle to cross a router.

```
template <typename MPDU,typename SPDU>
void std_network<MPDU,SPDU>::clock_tick(){
   unsigned int router_index;
   //Calling input first causes the router to propagates flit from in to out in
   //a clock cycle !!!!
   while(1){
      wait();
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->input_th();
      }
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->input_to_output_th();
      }
      for(router_index=0;router_index<std_network_size;router_index++){
            my_std_router[router_index]->output_th();
      }
   }
}
```

Swapped standard network clock_tick thread

Hence, the description of pipeline stages through sequential functions and the router scheduling are distinct problems and must be treated separately. The decomposition of routers in decoupled pipeline stages allows to describe a router a sequence of function calls. The *scheduling* of this functions determines the exact timing of the microarchitecture. The proposed simulation technique works well for both combinatorial and sequential logics inside the routers. It is up to the scheduler policy to determine the exact order in which

the function modeling combinatorial and sequential logic must be called to guarantee a proper microarchitecture behaviour.
The next subsection will show through profiling results and simulation speed comparisons the benefits of our modeling methodology.

5.6.3 Models profiling and simulation speed

Through tools such as *gprof* developed by GNU, it is possible to profile code execution in order to understand and compare the models developed at different levels of abstraction. Our golden model is represented by OCCN standard network just described in the previous subsection, where network static scheduling, SystemC signal avoidance and non-blocking behaviour are described. To compare our methodology, we profiled a model identical to the OCCN one (a simple router with virtual channels and two pipeline stages), scheduled statically as OCCN does, but using SystemC signals instead of variables to model link protocol. This comparison allows us to isolate the simulation speed up due to the SystemC signals avoidance. To measure the simulation speed up brought by the two modeling techniques altogether, OCCN models simulations speed is compared to a synthesizable VHDL model of the router, where microarchitecure FSMs were developed through VHDL combinatorial and sequential process to model the respective logics.
To show the skeleton of the signal level interfaces router model used in our comparisons, the following code snippet reports the router class, where signal ports are easily recognizable. The pipeline functions which model the router combinatorial and sequential logic, scheduled by the network global scheduler, as for OCCN, are provided in the class declaration (e.g. `xbar2out()`). As for OCCN, a single SC_THREAD schedules the routers pipeline on the positive edge of the clock. The only difference between this model and OCCN one is represented by the routers interfaces: TLM for OCCN and SystemC signals for signal level interface model.

```
template <unsigned int INPUT_PORTS, unsigned int OUTPUT_PORTS,...> class prou : public sc_module {
 //pseudo-code
  //PORTS
 public:
  sc_in<int> data_in[INPUT_PORTS];
  sc_in<bool> req_in[INPUT_PORTS];
  sc_in<bool> eop_sop_in[INPUT_PORTS];
  sc_out<bool> ack_out[INPUT_PORTS];

  sc_out<int> data_out[OUTPUT_PORTS];
  sc_out<bool> req_out[OUTPUT_PORTS];
  sc_out<bool> eop_sop_out[OUTPUT_PORTS];
  sc_in<bool> ack_in[OUTPUT_PORTS];

  sc_in<bool> clk;
  sc_in<bool> rst;
```

```
    //INTERNAL SIGNAL

    //VARIABLES
    bool eop_sop_flag[OUTPUT_PORTS];
    bool direction_eop_sop[INPUT_PORTS];
    int round_robin[OUTPUT_PORTS];
    int ROUTER_ID;
    int prev_direction[INPUT_PORTS];

    queue<int> data_out_queue[OUTPUT_PORTS];
    queue<bool> eop_sop_out_queue[OUTPUT_PORTS];
    queue<int> data_in_queue[INPUT_PORTS];
    queue<bool> eop_sop_in_queue[INPUT_PORTS];
    queue<int> direction_in_queue[INPUT_PORTS];

    state_port input_state[INPUT_PORTS];
    state_port output_state[OUTPUT_PORTS];
    routing_class<mesh> routing_table[INPUT_PORTS];

    //pipeline functions declarations
    void in2out();
    void xbar2out(int);
    void read_in();
    void read_in_port(int);
    void write_out_port(int);
    void write_out();
    void reset_method();
    void clk_method();

    //CONSTRUCTOR
    SC_HAS_PROCESS(prou);

};
```

Signal level interface router

The profiling results shows unquestionably some benefits brought by our OCCN modeling technique. In the following profiling call graph, the syntax is the gprof one. *index* is the function ranking, *time* the percentage of simulation time spent in current function, *self* the time spent in the function itself, *children* the time spent in functions called by the current function, *called* the number of times function get called and finally *name* field is self explicative, the demangled function name. In a SystemC signal level simulation, ten percent of simulation time is spent on updating SystemC signals (`perform_update()` call), with no strings attached. OCCN simulations, as they do not exploit SystemC signals at interfaces (signals modeled through plain C++ variables), spend a percentage of time close to 0 in the function (`perform_update()`) which is called by the SystemC kernel in any case, but in OCCN does nothing at all. As these SystemC models are just used for benchmarking and never for synthesis, modeling SystemC signals at the interface can be considered with no doubt as a waste of CPU. OCCN simulations are still clock accurate and any kind of measures obtained through SystemC signal interfaces are possible within OCCN (trace file included). The only benefits coming from SystemC signal level interfaces is about the models writing: SystemC signals guarantee consistency to threads ordering by construction (delta cycle update aware) whilst plain variables do not. But what OCCN showed is that, with cautious coding, this ten percent of CPU time wasted in updating signals away can be reused for loftier purposes.

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    0.00       1/400018      sc_simcontext::initialize(bool) [116]
                0.02    0.46  400017/400018      sc_simcontext::crunch() [6]
[18]    10.0    0.02    0.46  400018        sc_prim_channel_registry::perform_update() [18]
                0.02    0.44 7691723/7691723     sc_prim_channel::perform_update() [19]
-----------------------------------------------
```

Signal level interface router, perform_update()

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    0.00       1/300001      sc_simcontext::initialize(bool) [129]
                0.00    0.00  300000/300001      sc_simcontext::crunch() [5]
[209]    0.0    0.00    0.00  300001        sc_prim_channel_registry::perform_update() [209]
-----------------------------------------------
```

OCCN interface router, perform_update()

To furtherly bolster the claims set out so far, the following code snippet reports the profiling sections related to the `notify_delayed()` function which is called by signal channels anytime a signal value changed value. Signal level interface router spends nearly six percent of simulation time in notifying events in virtue of signal values changes. In the report, the functions above the profiled function (`notify_delayed()` in this case) represent the callers; the column value tells the following: `notify_delayed()` was called 13244405 times. In particular, it was called 1093826 times by `sc_signal<int>::update()`, 1445239 times by `sc_signal<state_port>::update()` and so on. As the reader can countercheck the clock update plays a minor role in event notification (see `sc_clock::posedge_action()`) and most importantly they are *inevitable* in a clock accurate simulation.
From the OCCN profiling report, it is easy to see that only the clock causes event notifications, because OCCN simulations do not use SystemC signals at the interfaces. From this standpoint, on the upshot, these profiling outcomes state that OCCN simulations trigger the bare minimum of event notifications in a SystemC *clock accurate* simulation, which is a major goal to get better and better simulation speed, proving the methodology effectiveness.

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    0.00  200012/13244405     sc_clock::posedge_action() [67]
                0.00    0.00  200012/13244405     sc_clock::negedge_action() [75]
                0.00    0.02 1093826/13244405     sc_signal<int>::update() [58]
                0.00    0.03 1445239/13244405     sc_signal<state_port>::update() [47]
                0.03    0.20 10305316/13244405     sc_signal<bool>::update() [22]
[24]    6.1     0.04    0.25 13244405         sc_event::notify_delayed() [24]
                0.04    0.21 13244405/13244406     sc_simcontext::add_delta_event(sc_event*) [25]
-----------------------------------------------
```

Signal level interface router, notify_delayed()

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    0.00  200000/400000      sc_clock::posedge_action() [39]
                0.00    0.00  200000/400000      sc_clock::negedge_action() [38]
[66]     0.6    0.00    0.01  400000         sc_event::notify_delayed() [66]
                0.01    0.00  400000/400001      sc_simcontext::add_delta_event(sc_event*) [63]
-----------------------------------------------
```

OCCN interface router, notify_delayed()

For what concerns simulation speeds, the two models developed with different modeling techniques at the interfaces (SystemC signals and TLM) were compared using a testbench where all network interfaces inject flits at full rate in the network in order to make the code crunch as much as possible. The topology tested was a 10 nodes mesh with a X-Y routing algorithm. Input and Output buffer lengths were set to 3 flits. Simulation speed comparisons were carried out on an Intel Xeon machine, running Linux kernel 2.4.21 compiled for SMP, CPU frequency 2.80 Ghz with cache size 512 Kbyte. All of the models and SystemC library were compiled with O3 gcc compiler option, no debug mode. To be noted that, as already stated, the two models deploy the same methodology concerning network scheduling, the only difference resides in models interfaces and this is important to gauge the simulation speed up brought by SystemC signal riddance. The simulation speed outcomes in figure 5.13 clearly states that OCCN models run at nearly twice the speed of an equivalent SystemC signal level interface routers, and this speed-up is mostly due to the SystemC signals avoidance (hence, event notifications).
Figure 5.13 reports the simulation speed comparison even for respectively 20 nodes and 40 nodes meshes networks. No surprise the simulation speed up (see figure 5.14) increases (75/41=1.82 (10 nodes), 35/18=1.94 (20 nodes) and 14/6=2.33 (40 nodes)) bolstering anew that the simulation speed up is brought mainly by SystemC signals avoidance. The more the nodes the worse the simulation speed of the signal level interface routers, because the signals update burden becomes heavier for bigger networks. The time spent in network control (scheduling) is the same for both models and obviously worsens as the network grows in size (the scheduler has to manage and schedule more functions so that to simulate one clock cycle it takes more time).
In order to end and achieve this section, I am going to report, in table 5.2 simulation speed of a router developed within OCCN and a synthesizable router written in VHDL. This comparison was chosen in order to demonstrate how much the combined modeling techniques implemented in OCCN (static pipeline scheduling plus signals avoidance) to achieve clock accuracy can gain over a pure RTL model. A synthesizable RTL model defines processes for combinatorial and sequential hardware blocks, and internally the router

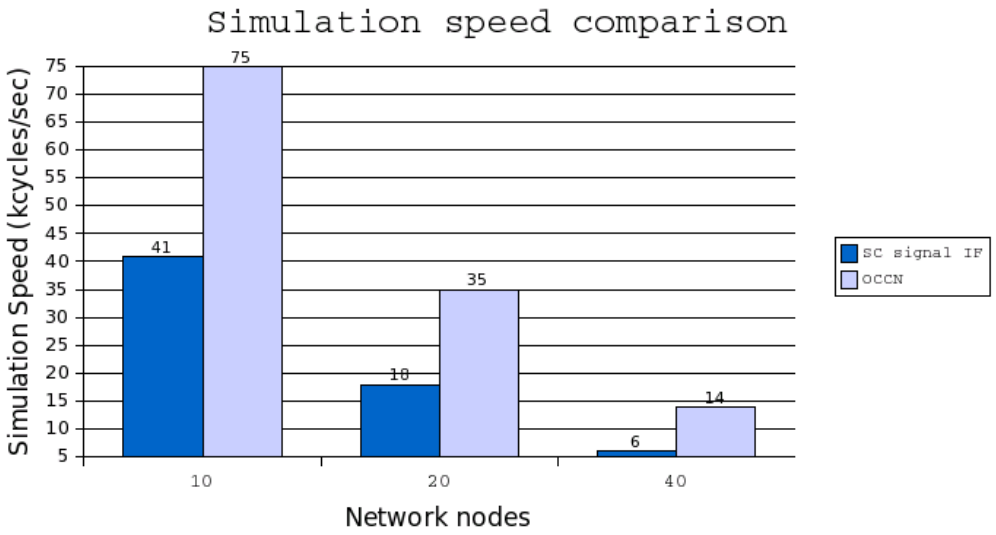

Figure 5.13: Simulation speed comparison

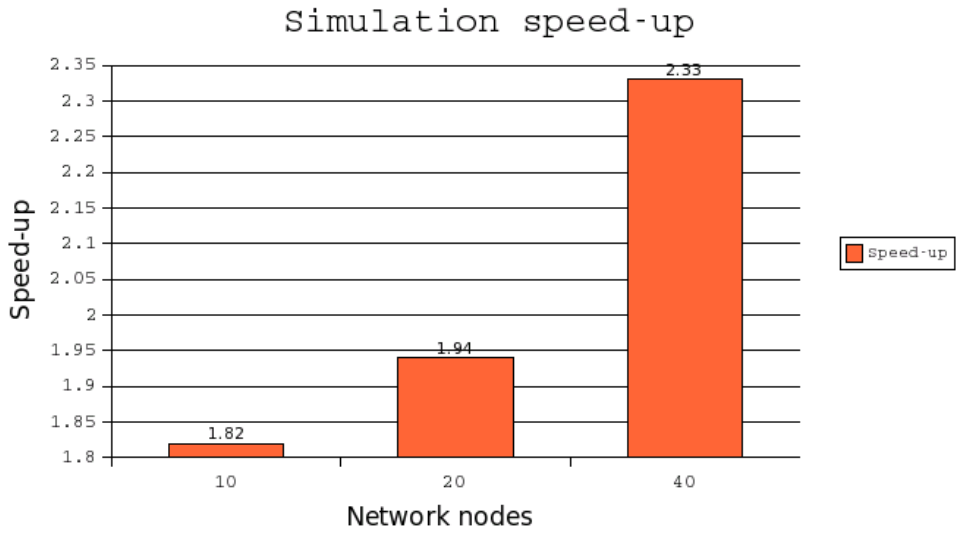

Figure 5.14: Simulation speed-up

building blocks (e.g. arbiters) communicate through signals. VHDL language was chosen as RTL model because, at least at STMicroelectronics, SystemC is not well-accepted for synthesis, hence pure hardware modeling. Anyhow, it is fair to state that a pure, synthesizable SystemC model will never run an order of magnitude faster than a VHDL compiled one (i.e. see Interleaved Compiled Code Architecture (INCA) by Cadence [12]), this because process control of the scheduler is significant in both kernels and simulation speed is kernel-bounded. VHDL simulations were run in batch mode using ncsim simulator [12]. In contrast to the previous comparison, the VHDL versus OCCN

VHDL	OCCN
3.3 (Kcycles/sec)	206 (kcycles/sec)

Table 5.2: VHDL vs. OCCN

router test aimed at comparing the simulation speed of just *one* router, in order to get the real speed-up brought by the combined simulation techniques (static pipeline scheduling plus signals avoidance). Strictly speaking, the simulation speedup factor ($206/3.3 \approx 60$), compared to the one in figure 5.14 is mainly brought by the router static scheduling; put it in another words, static scheduling allows to tease as little as possible the respective simulation kernel (SystemC), gaining a definitely significant factor in simulation performance. Once again, apart from synthesis purposes, I want to remind that benchmarking results obtained through OCCN and VHDL models are the same, which means that benchmarking tests using OCCN can be carried out with the same precision in much shorter time.
With an eye to the future, the next milestone in OCCN development consists in making it independent of the SystemC kernel, possibly through a synchronous (and light) simulation micro-kernel whose role is about to fire off router pipeline stages as clock ticks come. Even deploying static scheduling, the SystemC kernel plays a heavy role in simulation performance which is of no interest for simulations whose scheduling has been determined beforehand. It is difficult to predict the further simulation speed improvement that can be obtained through this approach, but in author's opinion it is really worth the effort for next generation systems on-chip simulations.

5.6.4 Modularized arbitration

Arbitration plays a paramount role in on-chip interconnections. Arbitration schemes are hierarchical (link, queue) and require a nested modularization. A key element for networks on-chip modeling consists in the capability to

explore different solutions early in the design cycle. Arbitration is not an exception.
The solution to this issue applied to STNoC™ routers and network interfaces models is templatization. This means that queues and link arbiters are template classes with well specified APIs (e.g. function arbitrate). Changing an arbiter is just a matter of redeclaring a given component and recompiling it. *Policy* and *mechanism* are fundamental concepts in computer science, and widely applied in C++ design patterns. When the model is recompiled the arbitration mechanism does not change (inputs request to the arbiter the link or the queues possibly with a priority), but the policy does change (the method to determine the channel or queue owner).
Embedded a simple code snippet of a router queue arbiter.

```
template <typename MANAGER_T>
inline
void outa<MANAGER_T>::arbitrate()

{
   switch(fsm_status){
      case FREE:
         arbitrate_priority();
         break;
      case LOCK:
         break;
      default:
         break;
   }
}
```

router queue arbiter

As stated early, the arbiter is templatized. The mechanism provided by the arbiter object is embedded in `arbitrate_priority()` function. Different arbiters implement this function in a different way. Policy changes but the mechanism stays the same. C++ design patterns proved efficient for networks on-chip modeling.

5.6.5 Network Interface models structure and hierarchy

The Network Interface (NI) is a key component of a network on-chip. It represents the glue defining the interface between transport and network layer, so that, somehow it has to provide mechanisms to protocol translation. Moreover, NI performs size and frequency conversions to adapt flows coming from ”remote” parts of the systems to the backbone frequency and data path. The modeling of the Network Interface component mirrors the network interface

microarchitecture.
At microarchitectural level the network interface is split in two parts: kernel and shell (see figure 5.15). The shell manages transport layer protocol

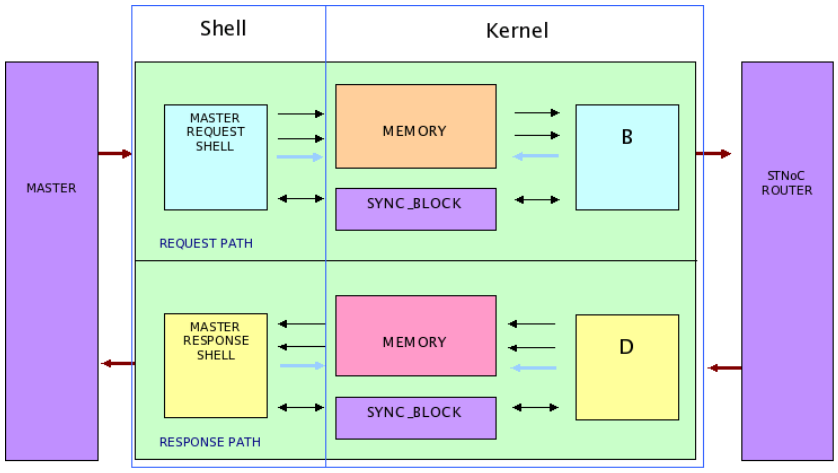

Figure 5.15: Mock-up of a NI microarchitecture

from/to IPs whilst kernel network interface packetizes transactions into flits and controls the flow from/to routers.
Therefore, an STNoC™ model of NI is a hierarchical composition of objects (see UML notation in figure 5.16). The main class is a templatized class whose template types are the incoming/outgoing transactions corresponding to transport layer protocol.

```
template <typename REQ,typename RSP,typename DATATYPE,typename MPDU,typename SPDU,
unsigned int TRANSPORT_DP_SIZE>
class bca_master_shellNI :
    virtual public tlm_blocking_get_if<RSP>,
    virtual public tlm_blocking_put_if<REQ>,
    public sc_module
    {};
```

master network interface

The remainder of template types is mainly used by the kernel network interface. In particular, MPDU and SPDU represent flits data and control sent over the network. The SystemC interface to/from the IPs is modeled after the TLM OSCI standard using blocking put/get interfaces.
A key development feature of network interface is represented by its class hierarchy. The design pattern used is composition. This means that the network interface is built from two objects modeling shell and kernel level.

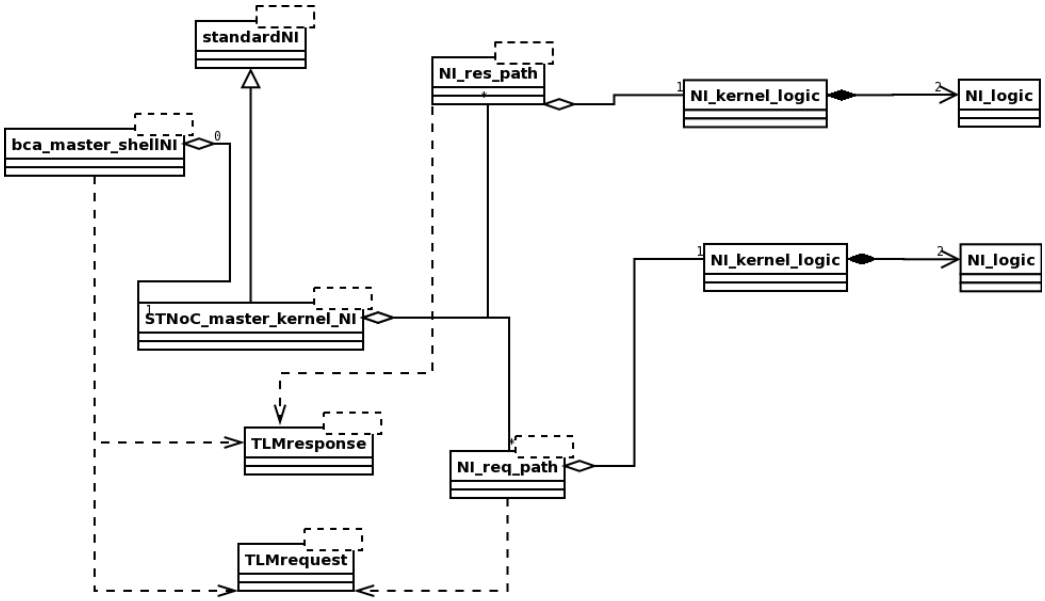

Figure 5.16: NI UML basic representation

Inheritance was not used because a shell network interface *is not* a superset of a kernel network interface, the two objects have distinct features that must be kept apart. The following code snippet highlights this important distinction.

```
template <typename REQ,typename RSP,typename DATATYPE,...>
class bca_master_shellNI :
    virtual public tlm_blocking_get_if<RSP>,
    virtual public tlm_blocking_put_if<REQ>,
    public sc_module
{
  STNoC_master_kernel_NI<DATATYPE,TRANSPORT_DP_SIZE,MPDU,SPDU>* get_ni(){
      return ni;
  }
 STNoC_master_kernel_NI<DATATYPE,TRANSPORT_DP_SIZE,MPDU,SPDU>* ni;
};
```

network interface composition

This apparently useless distinction turns out to be very important when it comes to changing kernel part of the network interface. Composition allows changes to be local in kernel network interface, whilst inheritance would have changed the methods available to network interface users (shell) (changing a hierarchical object means changing the methods available to that object, this is not true for composition [25]).

5.6.6 Network interface size and frequency conversion

As described early the network interface plays a crucial role in interfacing the ”external world” (IPs) to the network on-chip backbone. Size and frequency conversion are tough details to deal with even in simulations. Whilst frequency conversion in SystemC is rather simple to implement, the size conversion is a fiendish puzzle from different perspectives. It must be parametric, flexible and reconfigurable. IP data path sizes range from 32 bits to 256 bits and all the possible combinations must be taken into account to provide good tools for architectural exploration. In STNoC™ size conversion is handled through templates for a simple reason: avoiding code bloat. Template allows to define IPs sizes at compilation time so that the compiler can remove useless code whether size conversion is upsize (from 32 bits to 128 bits), downsize or there is no size conversion at all. STNoC™ models of Network Interfaces implement size conversion in a very efficient way from a code size perspective (code snippet embedded, see figure 5.16 on page 133).

```
template <..>
void NI_req_path<..>::data_put(const sc_bv<TRANSPORT_DP_SIZE>& data,
const sc_bv<TRANSPORT_BE_SIZE>& be,bool data_flag)
{
   if (TRANSPORT_DP_SIZE > FLIT_DATA_SIZE){
     \\ downsize code
   }
else if (TRANSPORT_DP_SIZE == FLIT_DATA_SIZE) {
   \\nosize conversion code
}
else {
  \\ upsize code
}
}
```

size conversion

The rest of the duty is done by the respective code that actually converts data path sizes in an efficient manner (data packing [38]).
Frequency conversion is of no interest because is handled automatically by hardware logic modeled in NI so that it just requires the possibility to declare multiple clock sources in network interfaces models.

5.6.7 Network interface C++ objects inheritance and composition patterns

Since the network interface is a highly modularized component, the C++ code used to describe it is highly modularized as well. A simple example was given in subsection 5.6.5 with the composition of kernel and shell network interfaces. The kernel network interface is made up of a number of

subcomponents, each one covering a particular task. Two points are worth mentioning here:

Simulation speed is strictly influenced by function calls. The composition of objects imposes function calls to communicate between them, somehow causing a simulation slow down. *Functions inlining* is highly used in STNoC™ models to define functions as fast as macros, namely inline small functions.

Code modularity is important to achieve, especially for code re-use. Of course, a monolithic module without a function call (code "slush") would simulate faster than a highly modular simulation platform but debug time and classes re-use must be taken into account in modeling time. Debug time is becoming the longest spell in modeling deadlines and things are getting worse for software programmers because of the code bloat due to platforms complexity.

An example is in order here to describe how modularity is boosted in STNoC™ models.

```
template <typename DATATYPE,unsigned int TRANSPORT_DP_SIZE>
class NI_kernel_logic : public sc_module
{
public:

    NI_logic combinatorial_forward_logic;
    NI_logic combinatorial_backward_logic;

};
```

NI logic

The class `NI_kernel_logic` encloses (see UML in figure 5.16) combinatorial logic paths useful to model hardware logic behaviors. In this way, through function calls (often inlined) combinatorial logic provides mechanism to NI kernel logic. A library of different logic components can be developed and maintained independently of kernel logic class. Once again, the decoupling of mechanism and policy helps improve code modularity.

5.7 Co-simulation wrappers

Communication refinement can be defined as the mapping of an abstract communication protocol to an actual implementation through co-simulation wrappers normally dubbed as *adapter* and *converter*. All along this process

the abstract communication scheme is either mapped onto whatsoever is provided by a given target architecture, or, refined into a more detailed custom implementation. Before describing communication refinement examples of STNoC™, the process of communication refinement must be addressed in a general way in order to introduce some key terms and ideas. Communication refinement is easily described considering a simple example. Formally, two high-level processing elements (PE1 and PE2 in figure 5.17) connected through an abstracted channel CH. A possible procedure to refine this model

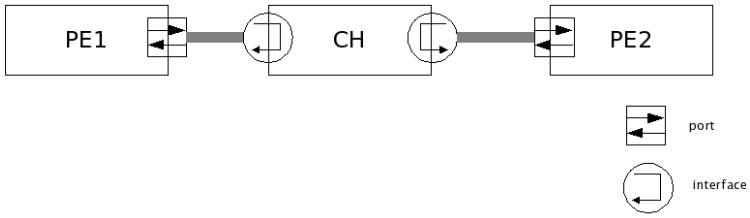

Figure 5.17: Simple architecture example

in a more detailed architecture can be carried out in several steps.

1. Choose a level of abstraction of the refined channel
2. Replace the communication channel CH with a refined protocol which describes the chosen channel
3. Implement the communication of the modules PE1 and PE2 over $CH_{refined}$ by two possible approaches:
   - wrapping $CH_{refined}$ in a way that the resulting channel $CH_{wrapped}$ matches the interface needs of PE1 and PE2, or by
   - refining PE1 and PE2 into $PE1_{refined}$ and $PE2_{refined}$ such that their respective interfaces match $CH_{refined}$ (adaptation)

After step 2 the interfaces of the modules and the refined communication are no longer compatible. In step 3, this mismatch is fixed through the modification of one of the connection points; either the channel is enclosed in a *wrapper module* to endow it with the interface of the original channel, or the modules are refined so that they comply with the respective interfaces again (see figure 5.18). Both procedures "refine" the channel in a given im-

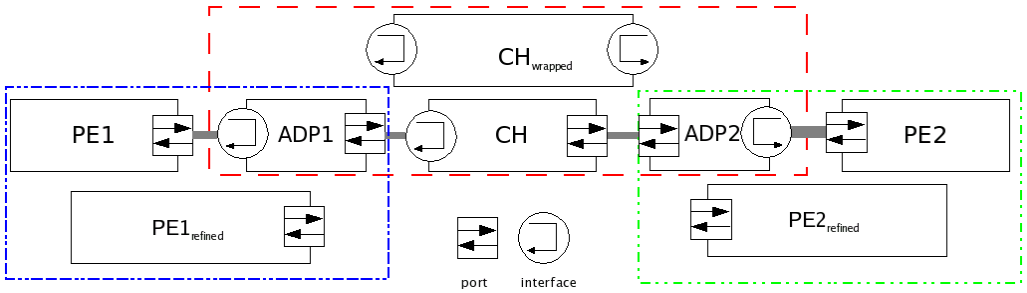

Figure 5.18: Wrapper example

plementation. The designer can either choose a wrapper approach, so that the mapping is carried out in the wrapper containing the refined channel or implement the mapping in the modules themselves.

5.7.1 SystemC/Verilog wrappers

Backward compatibility and legacy code are issues to take into account for modeling of complex systems. The definition ”legacy” code here unfolds to different meanings. Verilog components are still in use within ST in system simulation, verification and tuning, in particular for memory subsystems where the effort of modeling memory component in a cycle accurate way is tough. For STNoC™benchmarking, memory controller was simulated through Verilog code. Moreover, Verilog model of ST memory controllers are endowed with RTL gate delays (signals can be treated as early, middle or late) [5] depending on timing of combinatorial critical paths. SystemC wrappers must be designed with this delay in mind for two reasons:

- To guarantee clock accuracy

- To avoid bugs very hard to track down

To simulate Verilog modules in SystemC a simple ”stub” developed in C++ sufficed to wrap the two environments. Simulations were performed within NCSIM[12] which allows co-simulation of Verilog and SystemC in the same run. Hereinafter a code snippet of the stub is reported.

[5] This refers to the instant of sampling.

```
class MEMORY_WRAPPER_CLASSNAME : public sc_module
{
      MEMORY_WRAPPER_CLASSNAME(sc_module_name name_, const char *hdl_name,...):
            sc_module( name_ ), m_mem_base_addr(t_mem_base_addr)
      {

            MEMORY_inst = new MEMORY_CLASSNAME("MEMORY_inst", hdl_name);
            SC_METHOD(update_signals);
            \\sensitivity list omitted for confidentiality
      }

      void update_signals()
      {
        \\update of signals between systemc and verilog worlds
      }
};
```

SystemC/Verilog wrapper

Actually, the `update_signals()` class just acts as a signals converter and write them in variables readable by a Verilog module. The object `MEMORY_inst` represents the wrapped Verilog object instantiated through a pointer to a compiled verilog code defining the memory controller architecture (`hdl_name`). There are some more hurdles to tackle anyway. In particular, the `update_signals()` function has a very complex sensitivity list.
Leaving out implementation details for confidential reasons, SystemC / Verilog wrapping lasts one of the most complex problems in SoC benchmarking. Guaranteeing clock accuracy with mixed level simulation due to legacy code still represents a major challenge and a lot of work has to be done about this topic in the years to come.

5.7.2 TLM to signals wrappers

As STNoC™ routers and network interfaces sport a TLM interface and generic SystemC RTL models have signal interfaces, wrappers were needed to adapt interconnection models sporting mismatched interfaces. These wrappers were developed as adapters (see A1 figure 5.18). To provide further details, a TLM to RTL adapters implements an interface and contains a port suitable for the refined channel. The role of the adapter consists in getting/putting transactions and translating them into signals. In the following code snippet a TLM transaction of STNoC™ is reported.

```
template <typename CONTROLS,...>
class basic_bca_request{
  public:

  data_path<TRANSPORT_DP_SIZE_INBYTES> data;

  CONTROLS controls;

};

struct control_signals{
 unsigned char opcode;
 unsigned char t_id;
 unsigned char source_id;
};
```

TLM request transaction

These transactions must be translated by the adapter in `sc_signals` suitable to be driven to RTL modules. Hereinafter an adapter module is reported.

```
template <class ADDRESS_TYPE,...>
class RTL_to_TLM_NI : public sc_module {
   /** STNoC Side */
   sc_port<tlm_blocking_get_if<TLM_response<TRANSPORT_DP_SIZE> > > get_port;
   sc_port<tlm_blocking_put_if<TLM_request<TRANSPORT_DP_SIZE> > > put_port;

   RTL_port<DATA_TYPE, ADDRESS_TYPE, BE_TYPE> ln;
};

template <class ADDRESS_TYPE,... >
void  RTL_to_TLM_NI< ADDRESS_TYPE,...>::receive_request()
{
   while (1) {// while loop
         ln.gnt.write(true);
         if (ln.req.read()){
            TLM_request<TRANSPORT_DP_SIZE> req;
            req.controls.address=(unsigned int) (ln.add.read()); // ADDRESS_TYPE
            req.controls.be= (unsigned int) (ln.be.read()); //BE_TYPE
            req.controls.eop=ln.eop.read(); //bool
            req.controls.opcode= (unsigned int) (ln.opc.read());

            put_port->put(req);
         }
         wait();
   }
}
```

TLM to RTL adapter

The `receive_request()` process picks up signals from the input port, it prepares the TLM request and it sends it to TLM channel through put interface. To be precise the wrapper in this case functions as a *converter* (see figure 5.19), because on both sides (upstream and downstream) ports are used. In SystemC terms, an adapter differs from a converter in that an adapter is a channel because it has a set of *interface methods* that can be accessed through the ports of the connected modules. On the other hand, to replace a channel a module with ports instead of interfaces may be used. Hence, the main difference between *adapter* and *converter* is that the adapter is a hierarchical channel (because it defines *interfaces*) whilst a converter is a normal module that implements the protocol through SystemC plain ports. Converters

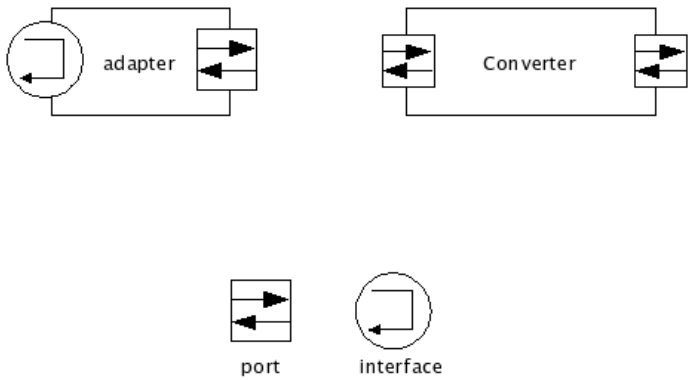

Figure 5.19: Adapter/Converter

were successfully and heavily utilized throughout STNoC™ benchmarking for different purposes, mainly for re-use of legacy simulators code.

5.8 Distributed simulations for Networks On--chip

Although computers keep getting faster, the demand placed on them by SoC simulation platforms is increasing at least as fast. Hardware designers want to simulate chips containing lots of concurrent Intellectual Properties (IPs). Software developers would like testing and debugging their programs early in the design cycle. Verification engineers would like bumping RTL designs with complex testbenches and nested constraints. To cut a long story short, computing power for many users is never enough. In order to handle larger and larger problem, computer architects have started turning more to parallel computers.
Parallelism has been introduced at several levels. At the ISA level, for instance, pipelining, superscalar and VLIW designs can be exploited to gain about a factor ten in performance over pure sequential simulations. However, in order to improve significantly SoC simulations speed, it is necessary to replicate entire CPUs or parts of them (hyper-threading technology), and to make them all work together efficiently. Herein lies the challenge for distributed simulations of SoCs, and in particular simulations run on SMP

machines, as described in the remainder of this section.

5.8.1 POSIX primitives for concurrent simulations

On a multi-processor system, threading (i.e. programming using threads lightweight parallelism in processes, see [11][73][4]) partitions a process in multiple streams of execution capable of performing more than one computation at the same time. The threaded programming model isolates independent or loosely coupled functional code units (threads) in a clever way that is made explicit in the program's source code (through threading primitives). If activities are designed as threads, *synchronization* must be somehow ensured to guarantee dependencies correctness. Threading has implicit advantages such as exploitation of programs parallelism and modular programming model.
Of course, these advantages come at a price. The time it takes to synchronize threads has to be included in the unwanted side-effects. It is very easy to lose performance by crippling the threaded code with too many synchronization points. The threaded programming model paradigm is simple to grasp, but writing proper threaded code is never trivial. Writing code that works well in multiple threads implies a thorough knowledge of threaded code possible issues such as race conditions, deadlocks and priority inversion. Writing multi-threaded code implies almost certainly to use library code such as POSIX *pthreads* library.

Most threaded programs need to share data between threads [11]. The most common and general way to synchronize between threads is to ensure that all memory accesses to the same (or related data) are *mutually exclusive*. For SystemC SMP kernel development two types of synchronization primitives have been used:

- `pthread mutexes`

- `pthread spinlocks`

The main (but still crucial) difference between mutexes (MUT mutual EX exclusion) and spinlocks concerns thread blocking. The critical distinction is that a thread trying to lock a spinlock does not block when the spinlock is already locked down by another thread. Rather the thread keeps "spinning", trying to grab the lock quickly again and again until it gets granted access to protected data. Spinlocks are great for fine-grain parallelism [47] [10], when

the code has been designed to execute only on a multi-processor, optimized and balanced to grab the spinlock for only a bunch of instructions, and performance is more important than yielding the processor resources to other processes (possibly processor hogs, craving for CPU time). Threads context switch time is a good measure of spinlocks effectiveness. If locking a spinlock takes more than the time a context switch from one thread to another takes, performance can be improved using mutexes and letting the operating system scheduler switch context for a while.
Mutexes and spinlocks are useful synchronization tools. But to build a proper SystemC parallel kernel, *barriers* are also needed to ensure that all threads cooperating in SystemC kernel execution reach a specific point before any can pass (see subsection 5.8.2). Pthreads library provides an opaque type for barrier (`barrier_t`) built on top of mutexes, meaning that threads waiting on this barrier are allowed to sleep. For fine-grain parallelism such as it is in SystemC, those barriers could end up providing dreadful performance[6]. Spinlocks based barrier will be described shortly as an alternative.

5.8.2 Delta cycle parallelism through kernel helper threads

The former sections provided an in-depth overview of SystemC kernel and methodology to achieve high simulation speeds (TLM). Strictly speaking, SystemC scheduler runs sequentially processes that are supposed to be simultaneously executed, ensuring that the execution time and causality remain consistent for the whole system. In other words, the kernel is a single thread of execution that exploit resources (Hardware) to represent the semantics of a parallel processing system (see figure 5.20).

[6]STNoC™ models profiling has been carried out using *Oprofile*, see [55] for further details.

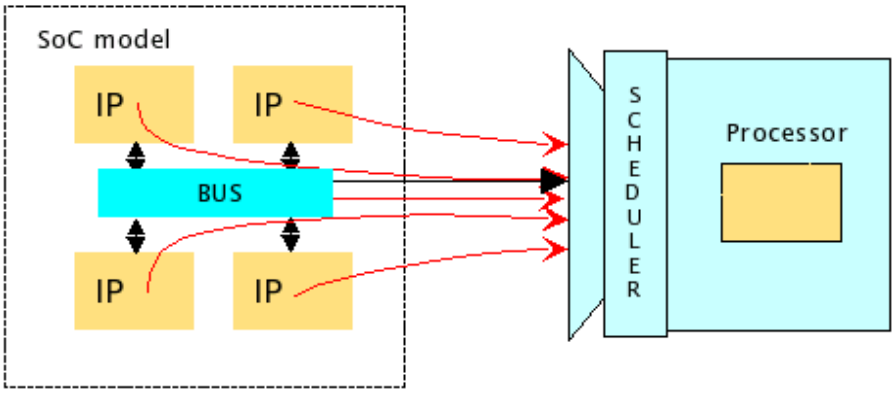

Figure 5.20: Parallel resources

The figure 5.21 hereinafter shows the classical time consumption of the

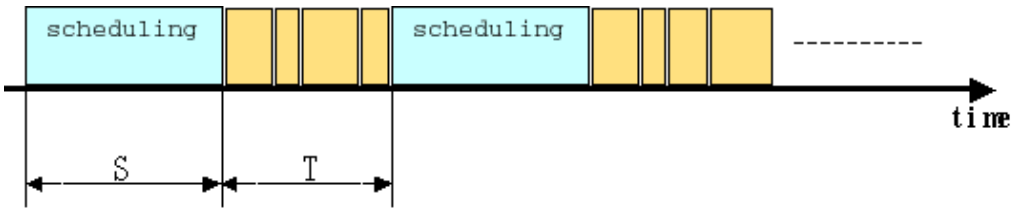

Figure 5.21: SystemC scheduler flow of time

unique processor usable at any given time by the current SystemC kernel. It may be noticed that the time consumed by the scheduler (S) is not negligible compared to the time consumed by the processes themselves (T). This is mainly due to the large number of synchronization needed between the processes (see the notion of delta cycle in subsection 5.3.1).
Strictly speaking, the core functions of the SystemC kernel are `crunch` and `simulate` functions within the `sc_simcontext` class; their role is to manage simulation time, event notification and the notion of delta cycles. Furthermore, the `crunch` function is in charge of firing off threads and methods that happened to be activated in a given simulation context (delta cycles, clock edges or instantaneous notification). The following code snippet reports the `crunch` function core loop.

```
//execute method processes

sc_method_handle method_h = pop_runnable_method();
while( method_h != 0 ){
    try{
        method_h->execute();
    }
    catch( const sc_exception& ex ){
        cout << "" << ex.what() << endl;
        m_error = true;
        return;
    }
    method_h = pop_runnable_method();
}

// execute (c)thread processes

sc_thread_handle thread_h = pop_runnable_thread();
while( thread_h != 0 && ! thread_h->ready_to_run() )
    thread_h = pop_runnable_thread();

if( thread_h != 0 )
    m_cor_pkg->yield( thread_h->m_cor );
```

Crunch function loop

The code just represents a sequence of statements that pops off each method or thread that is ready to run (hence it is "runnable") and fires it off in order to "crunch" its instructions. To manage clusters and then to distribute processes according to the processor's load, the crunch loop may be replicated in a number of kernel *helper* threads [7] equal to the number of processors minus 1 (the main co-routine is already a thread). Thus, within the `sc_context` class some pthreads get created whose role is to execute the code of the main crunch loop in an infinite tight-loop (see following code snippet).

```
/* NP normally should match number of processors (hardware or logical)
Note the main_t array of pthread_t identifiers (PID),
stored to implement searching by PID keys in
sc_simcontext associative arrays (arrays of main co-routines
and current
processes) */

for(int i=1;i< NP;++i){
        if (pthread_create(&main_t[i],NULL,&sc_simcontext::pthread_helper_method,NULL)){
            printf("ERROR Could not create thread\n");
            exit(1);
        }
}

// pthread helper (static) function simply replicates crunch main loop

void* sc_simcontext::pthread_helper_method(void* arg)
{
    while(1){
```

[7] In this context a thread is an operating system control path; e.g. in Linux, threads are a specialized form of the general `task` structure. Interested readers are encouraged to peruse the Linux `clone` system call. Linux (STNoC™ development has been carried out exploiting this great operating system) schedules threads like any process in the system, so that "kernel threads" means threads schedulable by the Operating System.

```
        sc_method_handle method_h = sc_curr_simcontext->pop_runnable_method();
        while( method_h != 0 ) {
              try {
                 method_h->execute();
              }
              catch( const sc_exception& ex ) {
                 cout << "\n" << ex.what() << endl;
                 m_error = true;

              }
              method_h = sc_curr_simcontext->pop_runnable_method();

        }

        sc_thread_handle thread_h = sc_curr_simcontext->pop_runnable_thread();
        while( thread_h != 0 && ! thread_h->ready_to_run() ) {
           thread_h = sc_curr_simcontext->pop_runnable_thread();
        }

        if( thread_h != 0 ) {
              sc_curr_simcontext->cor_pkg()->yield( thread_h->m_cor );
        }

     }
  }
```

pthread helper

The main thread, namely the one that executes the whole SystemC kernel code, must trigger the execution of these helper threads in order to inform them that the main crunch loop is about to start and they all have to start crunching code as a thread *working crew* (see [11]). In a work crew, data is processed independently by a set of threads (see figure 5.22). The members

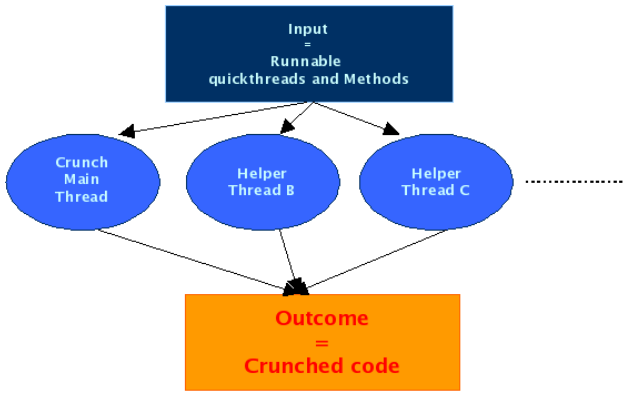

Figure 5.22: Working crew paradigm

of SystemC work crew remove runnable "processes"[8] from SystemC kernel shared linked lists (of runnable methods and threads) and execute whatever is required by theses processes instructions. Each queued SystemC process

[8]The term process is somehow misleading in this section. SystemC kernel processes are either SC_THREADS or SC_METHODS, not the pthreads used to allow concurrent execution.

could describe a variety of operations (hardware or software behaviors), but the common queue and mission (to execute the linked lists of processes) make the operating systems helper threads a "crew" rather than independent worker threads. To synchronize the helper threads with the main kernel co--routine (the one that executes the whole SystemC simulator) at the end of the tight-loop, a sort of barrier is needed[9]. Barrier synchronization provided by pthreads library is based on mutexes, namely it is a blocking barrier of type `barrier_t` (see following code snippet); its performance easily ends up being unaffordable for SystemC simulations, because each threads arriving at the barrier (unless it is the last) blocks and a context switch occurs.

```
//Helper thread function
void* sc_simcontext::pthread_helper_method(void* arg)
{
   while(1){

   //Posix barrier
   pthread_barrier_wait(&s_barrier);

       sc_method_handle method_h = sc_curr_simcontext->pop_runnable_method();
       while( method_h != 0 ) {
             try {
                method_h->execute();
             }
             catch( const sc_exception& ex ) {
                cout << "\n" << ex.what() << endl;
                m_error = true;
             }
             method_h = sc_curr_simcontext->pop_runnable_method();

       }

      sc_thread_handle thread_h = sc_curr_simcontext->pop_runnable_thread();
      while( thread_h != 0 && ! thread_h->ready_to_run() ) {
         thread_h = sc_curr_simcontext->pop_runnable_thread();
      }
             if( thread_h != 0 ) {
                      sc_curr_simcontext->cor_pkg()->yield( thread_h->m_cor );
      }
   //Posix barrier
   pthread_barrier_wait(&s_barrier);
   }

}
```

helper threads barrier example

A spinlock based barrier might represent a solution. This represents a very tricky point: an helper thread spinning on a barrier consumes its timeslice pointlessly. It might happen that an helper thread ready to crunch code (after having spun for ages) is preempted by the operating systems because its timeslice is exhausted, a complete and utter non-sense but still real due to spinning barrier implementation. It is not clear which barrier (mutex or spinlock based) provides a better solution to this issue, it heavily depends on simulation patterns.

[9]Note that the main co-routine can easily run out of process to execute whilst an helper thread is executing one. If no barrier is used, havoc occurs.

5.8.3 SystemC Kernel critical regions

Helper kernel threads allow for parallel execution of `crunch` main loop; this implies the possibility to fire off more than one System process (methods or threads) at a given time. Parallel execution is not for free anyway. SystemC kernel shared data structure must be protected by concurrent accesses in order to avoid pesky critical regions to deal with. In particular, events queues and runnable process linked lists are notorious candidates for bugs.
Parallel execution can be serialized in two ways: mutexes or spinlocks. SystemC parallelism is very fine-grained (meaning that just all the activities within a delta cycle can be executed in parallel without losing models semantics); mutex are not a suitable solution to critical regions protection because they are blocking and operating system context switches are not affordable in SystemC simulations. The only feasible way to proceed consists in using spinlocks for critical regions. Spinlocks have to be placed carefully in order to boost concurrency in System kernel [48], getting good overall performance. An example is in order here:

```
inline void sc_runnable::push_back_method( sc_method_handle method_h )
{
   // assert( method_h->next_runnable() == 0 ); // Can't queue twice.
   method_h->set_next_runnable(SC_NO_METHODS);
#ifdef MP_ST
   pthread_spin_lock(&m_lock);
#endif
   m_methods_push_tail->set_next_runnable(method_h);
   m_methods_push_tail = method_h;
#ifdef MP_ST
   pthread_spin_unlock(&m_lock);
#endif
}
```

Linked list critical region

A spinlock in this case is declared and used to protect the global pointer of runnable methods `m_methods_push_tail`. A cleverer solution would consist in declaring per cpu data and use processor affinity in order to avoid contention in data access. This solution is under thorough study and shows very promising.
Events management too, contains some code that unfortunately must be serialized. Delta and time events insertion and removal from scheduler event vectors must be protected through synchronization primitives (see following code snippet, function `remove_delta_event(sc_event*)`). Perhaps, this represents the sneakiest hurdle to parallel execution; delta events and time notifications execute very often in SystemC simulations, reducing in a significant way the parallelizable code, therefore the achievable speed-up.

```
void
sc_simcontext::remove_delta_event( sc_event* e )
{
    pthread_spin_lock(&e_lock);
    int i = e->m_delta;
    int j = m_delta_events.size() - 1;
    assert( i >= 0 && i <= j );
    if( i != j ) {
        sc_event** l_delta_events = m_delta_events.raw_data();
        l_delta_events[i] = l_delta_events[j];
        l_delta_events[i]->m_delta = i;
    }
    m_delta_events.decr_count();
    e->m_delta = -1;
    pthread_spin_unlock(&e_lock);
}
```

remove_delta_event

This section reported some of the major SystemC critical regions, not all, just because the concepts are the same for them all and the others do not add anything to the underlying issues.

5.8.4 Wrap-up and on-going work

The current main issues regarding SystemC porting to SMP systems concern a careful placement of spinlocks and an in-depth profiling of the time spent in the synchronization barriers.
Furthermore, the brand-new kernel must be studied using real world SystemC platforms in order to understand which simulation patterns may take advantage of parallel simulations, pursuing the path towards scalable systems. Early tests, performed on STMicroelectronics platforms containing a significant number of routers and Network Interfaces [10] (hence a high level of hardware parallelism) showed 20%-30% simulation speed increase compared to sequential execution (simulation performed on a 2 way Intel-Xeon SMP box, with Linux kernel 2.4, running at 2 Ghz of clock frequency). Compiling a SystemC kernel for SMP simulations is just a matter of switching a compilation macro. Further tests will be undertaken in order to simulate real designs.
Before concluding, a different approach is worth mentioning. In literature [50] some works pointed out the possibility to simulate different SystemC schedulers in parallel, each representing a "tile" of a very complex system. It is a really attractive concept, mostly within distributed systems, but in author's opinion it is still far from the reality of current platforms.

SystemC 3.0 release (software modeling constructs) will be bringing about

[10] Ranging from 10 to 20, considered as average complexity networks.

new software models to execute on top of current SystemC scheduler. Therefore, software models will still end up scheduling in lower layers of the SystemC kernel, leaving room for further enhancements of SMP scheduling. In sum, all the exposed facets will play a role in SystemC on SMP effectiveness, a role whose importance relies on SystemC kernel developers.
Herein lies the challenge for future Systems on Chip simulations.

5.9 Conclusion

This chapter, the very core of the thesis, explained a number of modeling features that are by now integrated in STNoC™ design flow. The description covered SystemC kernel and its evolution developed by STMicroelectronics for distributed simulations. Details were provided about STNoC™ models and advanced C++ modeling techniques, such as objects inheritance and composition. Profiling and simulation speed comparisons of different modeling techniques were carried out. To complete the description of STNoC™ design flow, the next chapter describes STNoC™-related modeling environments and general concepts which proved useful for STNoC™ benchmarking, such as traffic generators, traffic characterization and STBus design flow.

Chapter 6

STNoC™ benchmarking

6.1 Introduction

The former chapters provided an in-depth description of networks-on-chip state of the art, layering and STNoC™, the STMicroelectronics network on-chip modeling environment. In order to discuss interesting benchmark results of STNoC™, further concepts and modeling components descriptions are still lacking and must be unraveled to grasp simulation outcomes with some degree of detail. This chapter fills the gap, with a brief description of the components used to benchmark some applications on top of STNoC™ backbone.

6.2 STMicroelectronics STBus based SoC

STNoC™ is the natural evolution of the proprietary STBus protocol developed by STMicroelectronics. SoC are calling for parallelism and this is where an on-chip interconnection network comes into play. In figure 6.1 an extremely complex SoC by STMicrolectronics is shown. It is a full-blown integrated system for High-definition Digital TeleVision (HDTV) which comprises a number of IPs with different traffic characteristics both in terms of bandwidth, latency and real-time requirements. The interconnection subsystem represents a key component of the architecture and has been designed as a hierarchical STBus interconnect. Strictly speaking, the entire interconnection is not shown in all its glory; in particular subsystems such as video pipes actually contain local interconnects to let the IPs making up the subsystem communicate.

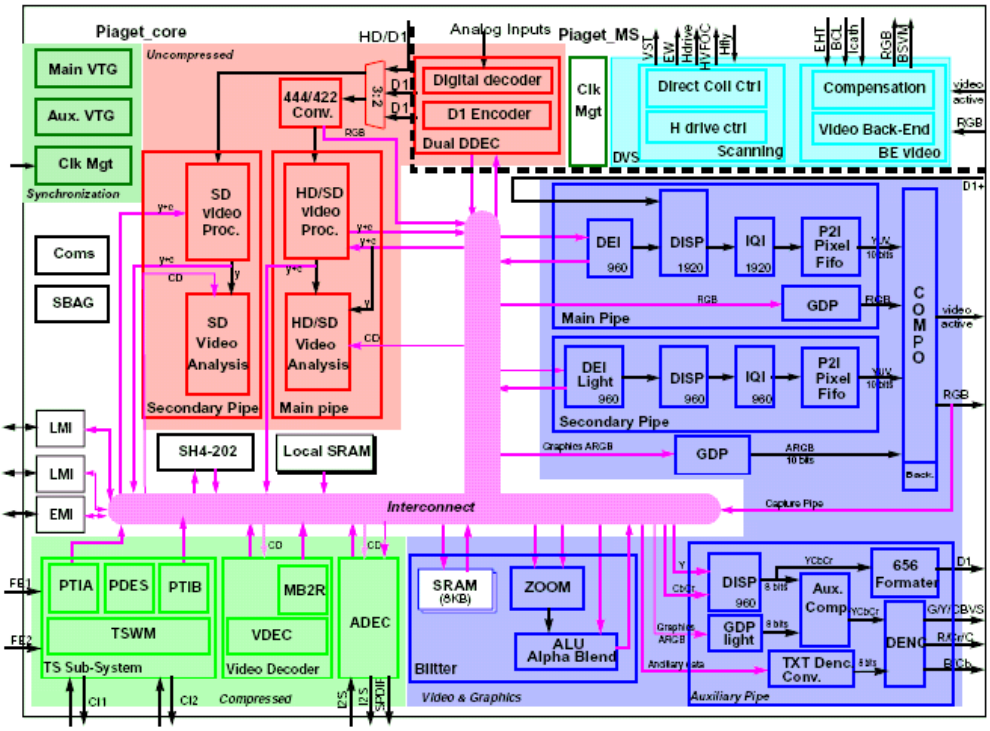

Figure 6.1: An example of a STMicrolectronics SoC

It is quite simple to understand how things get really complicated when design of interconnection systems such as this one must be developed. First of all, a major hurdle consists in defining the bus hierarchy, namely how to structure the bus layers into different components. Secondly, at top level, traffic is mainly directed towards external memory. The way in which traffic flows are grouped is of a major importance to the system performance. Finally, arbitration clearly becomes a distributed concept. Whereas arbitration in a shared bus is local and concentrated in a single point, in a distributed bus such as the STBus, global arbitration corresponds to the aggregate actions of different single arbiters. An arbitration which is effective locally could be a drag for the entire system when checked from a global perspective.

STBus provided a really smart solution to this kind of systems, but, as in all nice pictures, there are some shadows looming large at the horizon. One main hindrance to the evolution of buses is wires congestion. In a top-level (leaving out for now the low-level buses) interconnect such as the one shown in figure 6.1, the number of wires is tremendous. Wires congestion has not been a problem to the design of system yet, but for future systems it might cause severe performance degradation.
On-chip networks through serialization of wires into flits, allows the mitigation of wires congestion problems.
Even though it is clear that through serialization a NoC could provide great benefits to the wire congestion issue, it is not always that easy to understand at which level of the bus hierarchy a network on-chip can lead to better or at least same performance reducing wiring issues. In such complex systems, the only way to have an in-depth knowledge of system properties is through benchmarking.
In the remainder of this chapter the different components used to benchmark such complex SoC are explained as long as basic descriptions of typical applications. The next chapter will cover a given test-case that, unfortunately, is just a reduced subset of an entire system, this due to confidentiality reasons.

6.2.1 STBus Genkit

The STBus Genkit is a development platform for building any STBus interconnect from high-level SystemC model down to gate level implementation. It is a key component of the STBus and is developed internally by STMicrolectronics OCCS group which is in charge of STBus deployment. The STBus Genkit, through a user friendly graphical user's interface (GUI) developed using QT library, allows the designer to fully exploit the STBus

reconfigurability. In short, the Genkit is a way to reduce STbus based system time to design.

In figure 6.2 the Genkit design flow is sketched. Following a top-down

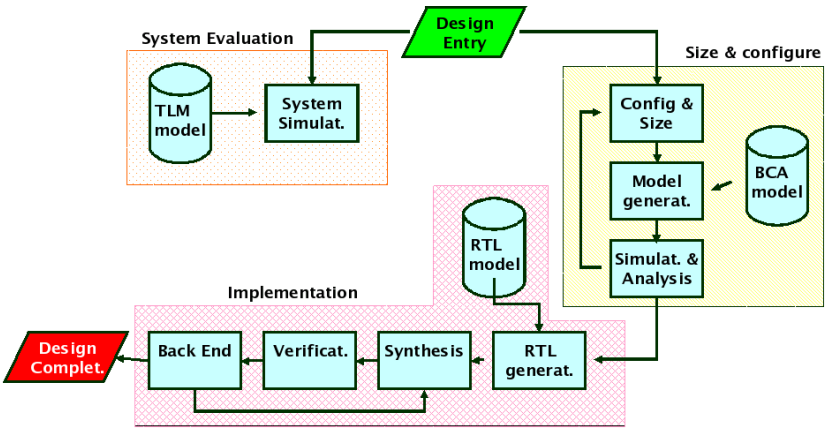

Figure 6.2: STBus genkit design flow

methodology, the design starts at transaction level where first estimations are made, using high-level simulations (mostly for software debugging early in the design cycle). Clock-accurate simulations (BCA stands for Bus Clock Accurate) are used to size up STBus components in a more detailed way. At this level of accuracy bandwidth and latency measures are accurate; the designers can choose to reconfigure the components if the designed system does not meet the requirements.

To be noted that the previously described OCCN simulator of STNoC™, even though it is a transaction level model, it can be described as a BCA simulator, in that it allows clock-accurate measures of bandwidth and latency. The real difference between STBus BCA models and STNoC™ models is the way in which interfaces are abstracted. OCCN sports a TLM clock-accurate interface, where communication takes place using function calls. STBus models sport a signal level RTL interface, which causes a simulation slow-down. Work is in progress to convert STBus interfaces to transaction level. BCA models do not contain gate level delays and synthesis constraints.

RTL libraries included in the Genkit can be linked to create a synthesized version of the STBus interconnection, ready for verification and back-end. To be noted that not all STBus components are available as SystemC BCA models; in particular the memory controller is available just as a Verilog component.

Additional insights will be provided briefly in subsection 6.2.2.

The Genkit design flow is well-supported by a powerful graphical user interface based on the QT library [60]. In figure 6.3, the extension of graphical

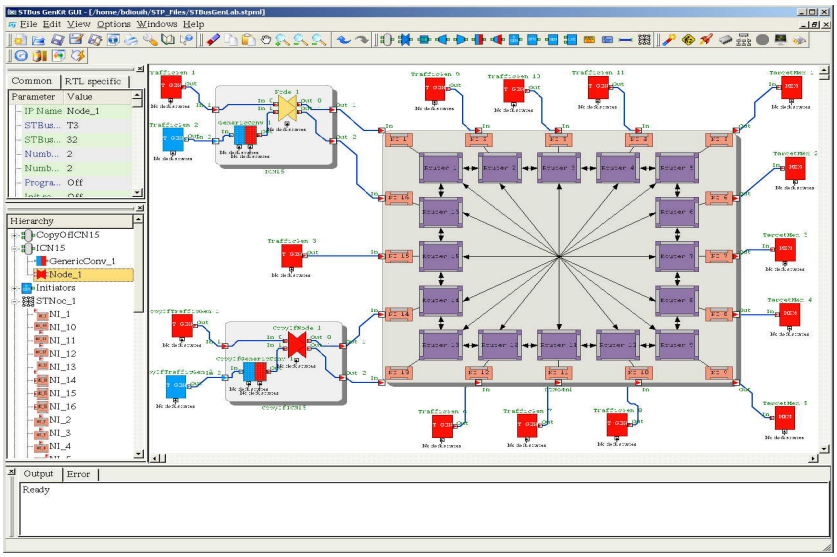

Figure 6.3: NoC QT graphical user interface

user interface to NoC components is reported, providing an idea of the environment. On the left-hand side of the snapshot, model hierarchy and textual description of elements are present. Models are picked-up from a library and connected as graphic objects. In this way, the end user does not have to know the underlying models details, binding rules and so on. Anything is automatically managed through the graphical user interface. Command buttons on upper toolbars hide scripts that generate model, makefiles and top-level instantiation.

Due to the ever increasing complexity in interconnection design, STNoC™ integrated an extension of the Genkit design flow to promote architectural exploration of different network architectures early in the design cycle. In figure 6.4 the procedure to build STNoC™ platforms using the NoC architectural exploration framework is shown. Starting from an XML configuration file, a NoC platform is built, together with appropriate traffic generators (see section 6.4 on page 159). Simulation runs are checked after simulation completion, saved in a data base repository, and analyzed through perl scripts. VCD files are analyzed through Sysprobe tool developed by STMicroelectronics to extract latencies, request to grant statistics and possible protocol violation for high-level verification.

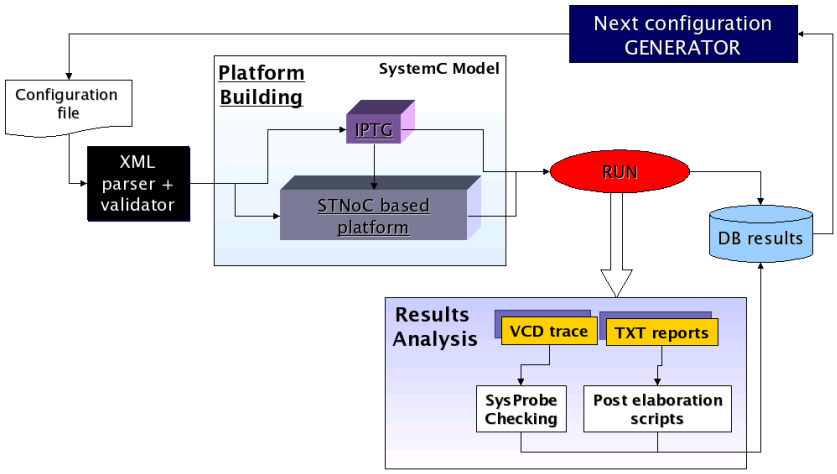

Figure 6.4: STNoC design flow extension

Hereinafter a code snippet of a STNoC XML configuration file is reported:

```
<platform>
<interconnect>
   <topology>spidergon</topology>
   <node id="0">
   <type> Master </type>
   <initiator>
      <config_r>CUSTOM_R</config_r>
      <config_w>CUSTOM_W</config_w>
   </initiator>
   </node>
   <node id="1">
   <type> Slave </type>
   <target>
      <start_address>0x100</start_address>
      <end_address>0x1FF</end_address>
   </target>
   </node>
```

STNoC XML grammar of a platform

The given platform is instantiated as a Spidergon topology, with two nodes configured as a master and a slave. It is possible to declare the slave address space and initiator configuration files. This "STNoC™ explorer" turned out to be very useful to explore different NoC architectures and some results will be reported and highlighted in chapter 7.

6.2.2 Mixed-level interconnection simulations

One of the major concern of STNoC™ benchmarking consisted in mixed level simulations. As already stated bandwidth and latency accurate measures require bus cycle accurate models. To build a hierarchical interconnect made

up of a hybrid solution (bus for local subsystem, NoC for system backbone), models with different interfaces had to be connected. Moreover, to benchmark memory access co-simulation of SystemC and Verilog models had to be carried out, implying the use of a proprietary (NCSIM[12]) simulator that allows to execute concurrently different kernels (SystemC and Verilog) at a time.
Adapters and converters have been thoroughly described in section 5.7 on page 135. Even if it may seem the easiest part of the work, mixed-level simulations turned out to be one of the most complex task for platform developers. In particular, wrappers must guarantee clock-accuracy, meaning that the interface translation has to be somehow "transparent" from a timing point of view; the wrappers are just the glue that merges the mismatched interfaces, they must not affect the flow of time. This was not an easy task at all.
Besides, Verilog models of the memory controller contain gate level delay to simulate instant of sampling (for instance, a signal declared *early* is allowed to change in the first 25% slice of the clock cycle). Wrappers must take delays into account to guarantee clock accuracy; this delay is parametric, implying configurable wrappers to be instantiated depending on the compiled Verilog architecture.
Mixed-level simulations allowed the benchmarking of STNoC™ in real SoC designs such as the one in figure 6.1. Even if the real expected outcome of such platforms is architectural exploration, the possibility to run simulations of real SoC can be already considered a factor of success, given the tremendous complexity of nowadays systems.
Together, STBus Genkit and STNoC™ OCCN models have been used to benchmark two real SoCs developed by STMicroelectronics with high simulation speed and clock-accuracy.

6.3 Applications high-level descriptions

Multimedia is considered as a godsend in the networking domain. This because providing multimedia, which implies high bandwidth, represents an immense challenge from different perspectives. Multimedia is an important market share even for semiconductor companies such as STMicroelectronics whose revenue is mostly brought and made up of chips capable of running high-end multimedia applications. STMicroelectronics is on the leading edge concerning set top boxes chips, high definition television and mobile platforms (that are integrating multimedia components). No surprises the application

explained in this thesis is part of the multimedia world.
If providing multimedia through computer networks (e.g. video on-demand) represents a critical task, on-chip designers (mostly *interconnection* designers) are not having that much fun in designing interconnection systems able to sustain these impressive bandwidth requirements either. The design of an interconnection for system on-chip capable of guaranteeing bandwidth and latency constraints for application such as MPEG (Motion Picture Experts Group) standard is more an art than a science; this mainly because, as the next section highlights, the application generated traffic is somewhat statistical, depending on algorithm combination of images whose bandwidth is difficult to predict statically.
Networks on-chip, STNoC™ in this case, can change the methodology applied to design interconnection systems through a well-known concept named Quality of Service. Achieving quality of service represents a fundamental step towards predictable systems. An efficient Quality of Service mechanism clearly empowers designers with a methodology to *control* how traffic is routed through the interconnect, changing the design paradigm in a significant way.
The remainder of this section reports a basic introduction of complex video applications, in order to highlight how their traffic can impact interconnection design for systems on-chip.

6.3.1 Video streams

Video streams traffic is normally generated through images scanning [74]. A typical 2-D image is obtained through a camera scan; an electron beam is used to get camera signals across the image and down it, recording the light intensity. At the end of the scan, called a *frame*, the loop restarts. This intensity of the beam as a function of time is broadcast, and receivers reconstruct the image.
The precise scanning parameters vary from country to country. North and South America and Japan has 525 scan lines, a horizontal-to-vertical aspect ratio of 4:3, and 30 frames/sec. The European System has 625 scan lines, the same aspect ratio of 4:3 and 25 frames/sec.
Color video uses the same scanning method as monochrome (black and white), except that instead of scanning the image with one moving beam, it uses three synchronous beams, one for each primary colors: red, green and blue (RGB). Standards of TV (NTSC, PAL, SECAM) combine RGB signals into a *luminance* (brightness) and two *chrominance* (color).
As already stated, semiconductor companies are investing a lot in High Def-

inition Television (HDTV) systems in that they show promising and ensure a huge market share. They improve the granularity of images by doubling the number of scan lines (aspect ratio of 16:9). Digital Video is a sequence of frames, each made up of a rectangular grid of picture elements, or *pixels*. Color video uses 8 bits per RGB color. Typical resolutions of monitors are in the ranges $1024x768$, $1280x960$ and $1600x1200$. Even the smallest of these with 24 bits per pixel and 25 frames/sec unleashes a bandwidth of 472 Mbps. As the reader can understand, not a light workload for on-chip interconnections.

Encoding and decoding algorithms are widely deployed to compress video streams. The MPEG standard defined many algorithms over the the last few years to compress videos. Because movies contain both images and sound, MPEG can compress both audio and video.
Let us consider video compression: two types of redundancies exist, namely spatial and temporal. Spatial redundancy can be solved by working out each frame separately with an on-purpose algorithm (e.g. JPEG). Additional compression can be achieved by exploiting the similarity between consecutive frames. MPEG takes into account different operating modes of cameras by defining four kinds of frames:

1. I (Intracoded) frames: Self-contained image frames, coded without the need for reference to other frames. Use JPEG for I-frame encoding (spatial redundancy).

2. P (Predictive) frames: Encoding of motion vector and small macroblocks differences. Used preceding frames for temporal redundancy.

3. B (Bidirectional) frames: Forward/Backward interpolated prediction. Predicted by both previous and next frames.

4. DC (DC-Coded) frames: Encoded through block averages, they include only discrete cosine value of each block; used for rapid searching.

I frames are just still pictures that can be encoded/decoded through a form of JPEG. To understand the statistical behavior of bandwidth in MPEG stream, P frames are much more interesting. They are based on the concept of *macroblocks*, whose *typical* resolution corresponds to $16x16$ pixels in luminance space and $8x8$ pixels in chrominance space. A macroblock is encoded by searching the previous frame for it or a picture block that slightly differs from it.
An example is in order here, shown in figure 6.5. The two frames backgrounds

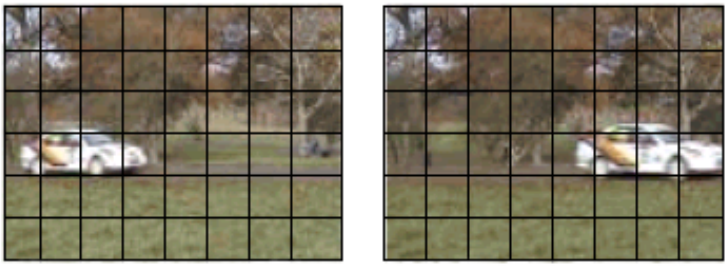

Figure 6.5: Example of MPEG P-frame

are the same, but they differ in the position of the car. The macroblocks which encompass the background scene will provide a perfect match, but the macroblocks which include the car differ in position by some details and have to be searched through some algorithm.

Normally in STMicroelectronics chips MPEG decoding pass through memory access. Whilst memory accesses caused by uncompressed flows are predictable (a given number of pixels accessed sequentially, aka raster mode), memory accesses unleashed by MPEG decoding are inherently dependent on the type of images (still picture, camera panning or zooming) so less predictable. This type of traffic can cause network overcommitment (network designed for the worst case). Quality of services "knobs" can turn out to be very helpful to avoid network overcommitment. In particular network interfaces dynamic programming can help overcome the fluctuation in bandwidth demands coming from irregular traffic flow such as MPEG video. The memory access footprint has also severe consequences on memory efficiency, a subject which is out of scope of this thesis.

6.4 Traffic Modeling

The benchmarking of a complex object such as a hierarchical interconnection requires models capable of running at very high simulation speed. Platforms containing 50-60 bus masters connected through a hybrid bus/NoC solution are common for STMicroelectronics designers. Whilst the interconnection modeling features have been thoroughly described in chapter 5 and in subsection 6.2.1, how traffic is generated has not been described yet.
First and foremost, it is important to underline that VHDL code of STMi-

croelectronics IPs is available for simulations; so it may seem natural to use VHDL IP models to generate real traffic on the network. But there is a catch: simulations of many IPs using VHDL code takes *days* to be executed, something that conflicts with fast architectural exploration needs. Fine tuning of interconnection requires many simulation runs so that VHDL simulation of IP traffic is not affordable to benchmark current systems on-chip.
To solve, somehow, computational burden issues related to VHDL simulations, system platform group of STMicroelectronics developed high-level traffic generators, in SystemC environment, whose name is IP Traffic Generator (IPTG). Due to confidentiality reasons, IPTG cannot be described. In the next subsection a brief description of a hypothetical configuration file is reported.

6.4.1 IPTG overview and sample file

The role of an IPTG consists in generating IP traffic according to some configuration file loaded at run-time. Strictly speaking, an IPTG is a SystemC element whose processes behave as an IP with a given bus interface. Combination of opcodes, operating modes and timing of transactions allows to describe an IP in an accurate manner, taking advantage of the compiled nature of SystemC code which in turn provides high-simulation speed. In figure 6.6 an IPTG pseudo-config file is reported. Through simple variable assign-

```
$IPTG_NAME          MPEG
$IPTG_COM           STBUS

$IPTG_DATA_SIZE                 64


$BEHAVIOUR_NAME         MAIN
$BEHAVIOUR_SEQ          START
                        NOP 200
                        STORE  0x08000008  0xff 0xe9C4015f 0x00000000
                        STORE  0x08000018  0xff 0x22b47e8a 0x00000000
                        STORE  0x08000030  0xff 0x18001503 0x00000000
                        STORE  0x08000038  0xff 0x18001503 0x00000000
                        NOP 7000
                        END
```

Figure 6.6: Pseudo IPTG config file

ments, IP protocol, data size and a number of others parameters can be set to describe an IP configuration. Then, a section named ”behavior” contains the dynamic execution of opcode according to complex internal mechanisms,

hidden to the users. In figure 6.6, a behavior that executes store transactions mixed with no-operations (NOP) at different addresses is shown.
IPTG allows to define the configuration of complex operations by means of behavior composition; this means that different behavior components could be integrated together to simulate complex threads of execution with fast C++ models. Platforms containing more than 50 IPTGs have been executed for STNoC™ benchmarking with satisfactory simulation speeds.

6.5 Conclusion

The aim of this chapter consisted in introducing different components and concepts useful to understand one of the most important activities developed in this thesis, namely networks on-chip benchmarking. Firstly, high level traffic descriptions were thoroughly explained, in particular regarding MPEG traffic behavior. STBus Genkit basic features were considered as well as its powerful design flow, which completed STNoC™ design environment in order to build hierarchical interconnections. The explication of the inner-workings of IP Traffic Generators (IPTGs), very useful to generate traffic behaviors over the network, ended the chapter. The next chapter shows the outcomes of STNoC™ benchmarking executed on synthetic case-studies, as the completion of a lot of work, including modeling, architectural design, traffic study and mapping.

Chapter 7

Outcomes analysis of STNoC™ benchmarking

7.1 Introduction

STNoC™ benchmarking was a compelling work due to the huge amount of parameters involved in architectural exploration and system complexity. STNoC™ models made great strides to tackle systems complexity through a reconfigurability pushed to the limit, allowing benchmarking of extremely complex systems on-chip. As described those chips models are made-up of heterogeneous[1] components, connected together through some adapters and converters.
Due to confidentiality reasons, unfortunately, these complex benchmarks cannot be described within this thesis. To partially replace such complex benchmarks, this chapter reports two kinds of measures for STNoC™, chosen on-purpose as proof-of-concept. Firstly, generic topology characterization through random traffic is reported, in order to better describe and highlight Spidergon topology properties. Secondly and finally, outcomes of throughput and latency of real applications whose traffic passes through STNoC™ are described. In particular, a comparison of arbitration is reported in order to show how quality of service can help build chips with predictable performance, achieving low-latency and fair bandwidth share on top of a best-effort network.

[1] Heterogeneous here is used in modeling context.

7.2 STNoC™ general characterization

STNoC™ general characterization has been carried out through random traffic generation as provided with SystemC Verification Library, which is an open-source collection of objects developed on top of SystemC to somehow enhance the SystemC capabilities in terms of verification properties. Seeds randomization was taken into account in our NoC simulator (see figure 6.4 on page 155), in order to generate random traffic with good statistical properties. IPTGs were configured in this case as a traffic instance that drives burst of read/write transaction over the network according to some IPTG internal commands.
Random traffic characterization is the first step after mathematical study of the network, in order to check whether or not the behavior of queue architecture, flow control and routing fits with expected results. In some sense, it represents the first "practical" feedback obtained through system simulation, and its usage is typical of early stages of design when just high-level parameters are significant to the whole system performance (e.g. which routing algorithm performs better in a given context).
In the following subsection throughput and latency measures of various network configurations are presented with embedded comments.

The objective of the results described in the next subsection (7.2.1) is to underline how network characterization can be carried out in a real world network on-chip. They must not be taken as official results and bounds of STNoC™ network on-chip because they have not been optimized. Official STNoC™ characterization is a confidential document of STMicroelectronics and cannot be unveiled due to confidentiality reasons.

7.2.1 Throughput and latency measures

Generic measures for STNoC™ characterization and benchmarking were carried out in order to definitely state topology properties and find out the best approach concerning buffering model and allocation. In figure 7.1 a typical measure of networks on-chip (accepted vs. offered load) is reported for STNoC™, for plain random traffic (each node sends traffic randomly to other ones, 8 nodes STNoC™ configuration). The X-axis unit corresponds to the number of flits injected into the network per each node per clock cycle. The Y-axis corresponds to the accepted flits.

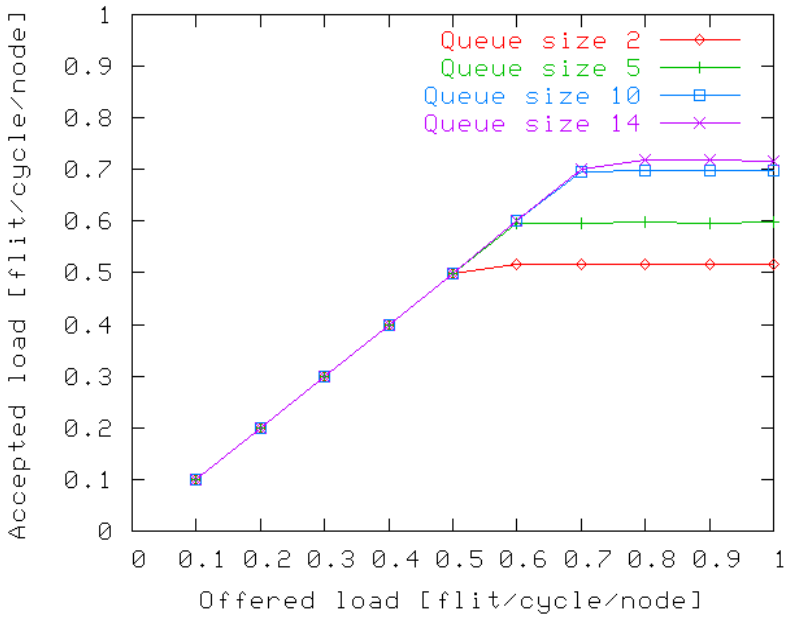

Figure 7.1: Accepted throughput

The shape of the curve is a typical one for interconnection networks. In the first part of the curve, as the offered traffic increases, the accepted traffic increases as well; the network in this configuration is *linear*. Whatever is injected gets accepted by the network, meaning that the network under these traffic conditions is able to sustain the required traffic. A more interesting point is about the threshold at which the network starts saturating, meaning the throughput curve becomes flat. At this point the network is no longer able to behave linearly, and it becomes somehow "outgunned".
Injected flits, as highlighted later, suffer from unbounded latencies, making the network completely unusable (who is willing to fiddle with unbounded latencies after all ?). It is worth noting that the saturation threshold strictly depends on FIFO sizes inside the routers[2]. As the attentive reader might be noticing, the saturation point increases as FIFO sizes increase, this because FIFOs improve the network pipeline in terms of accepted load, at the cost of larger router silicon area. The routing algorithm used in this experiment is dubbed *Afirst* in that packets choose the across link as the first hop or never, due to Spidergon topology properties. Many other algorithms are possible each with its strengths and weaknesses.
In figure 7.2 an interesting latency plotting is reported.

[2]Queuing strategy cannot be unveiled due to confidentiality reasons.

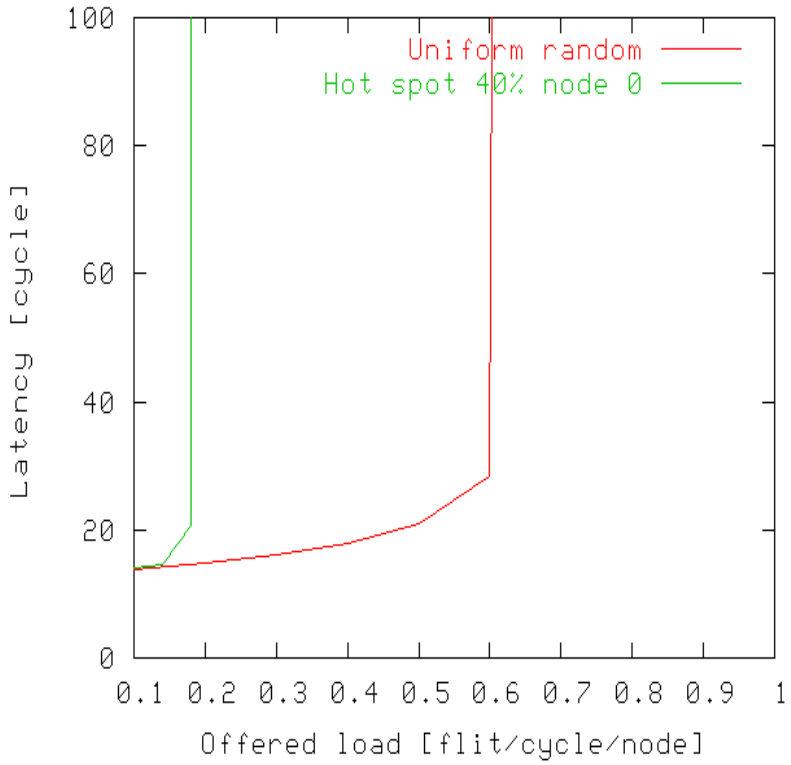

Figure 7.2: Latency: random and hot-spot traffic

In figure 7.2 packets' latency is measured under two different traffic conditions:

Random Traffic consists in sending uniform randomly distributed packets to all networks nodes that act as masters and slaves in the same time.

Hotspot traffic consists in choosing a percentage p of the traffic to be *hotspot*, namely directed towards a given target, according to the hot-spot model:

$$P(dest == hotspot) = (p) * (N-1) + (1-p) * \frac{N-1}{N} \tag{7.1}$$

with N number of nodes and p Bernoulli probability of hot-spot destination.

It is worth emphasizing that when just 40% of traffic is directed towards a hot-spot destination, the saturation point decreases in a staggering way, implying early saturation. This straightforward example demonstrates how useless is the usage of a network on-chip when most of the traffic is directed towards a single target. No network could improve this kind of gridlock, which falls back to a shared bus solution[3].
Even if a network cannot be considered the holy grail when traffic is just-one-target based, networks on-chip turned out to exhibit interesting physical properties with respect to busses (see section 3.2 on page 28).
Clearly, the exponential steep slope of the latency curve at saturation point describes an unusable service.

The saturation point states which workloads can be sustained by the networks, and the goodness of a network can be judged by two major factors:

- Latency behavior in linear sector.
- Saturation point.

In the linear portion of the curve, the network ensured linear throughput with latency dependent on routing algorithm, flow control, arbitration and buffer architecture. All of these parameters influence the network behavior in the linear operation range.
The saturation point as well is determined by the aforementioned parameters. For current systems the network range in which STNoC™ provides linear

[3]As traffic becomes more and more hot-spot, the concurrency in the network in terms of routing path is lost.

behavior with low latency proved to be fairly suitable[4], throughout different configurations of real world systems on-chip.

Random traffic characterization was also very useful to compare different routing algorithms[5] and mapping of STNoC™ configurations. In figure 7.3 several mappings of initiators and targets on a $6x6$ STNoC™ network is shown. As each Processing Element (PE) is supposed to be mapped on a NI, mapping becomes a very important step of the interconnection design flow, as topology properties and routing algorithm ones can influence network performance. In a $6x6$ network (network arity 12 with 6 initiators and

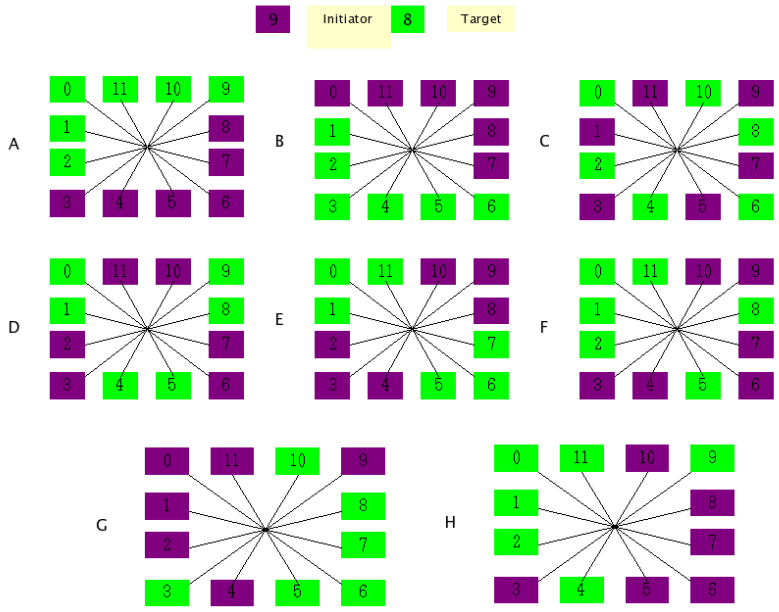

Figure 7.3: STNoC™ possible mapping in $6x6$ configurations

6 targets) initiators and targets can be mapped according to different permutations. However, this placement inherently affects performance because of network topology and routing algorithm properties. Through straightforward maths, it is easy to verify that mapping D provides the least average

[4]It might be better described as "overkill".

[5]They are not described here due to confidentiality reasons.

distance in terms of hops (1.83 hops) between initiators and targets, whilst layout E provides the worst case (2.44 hops). No surprise the results mirror these theoretical averages. In figure 7.4, outcomes of throughput for all mapping configurations in figure 7.3 are reported (*afirst* routing algorithm); it is worth noting the difference in throughput performance between the D and E cases (see D and E configurations), where D case study neatly outperforms E case. Latency results are more self-evident in order to gauge the saturation

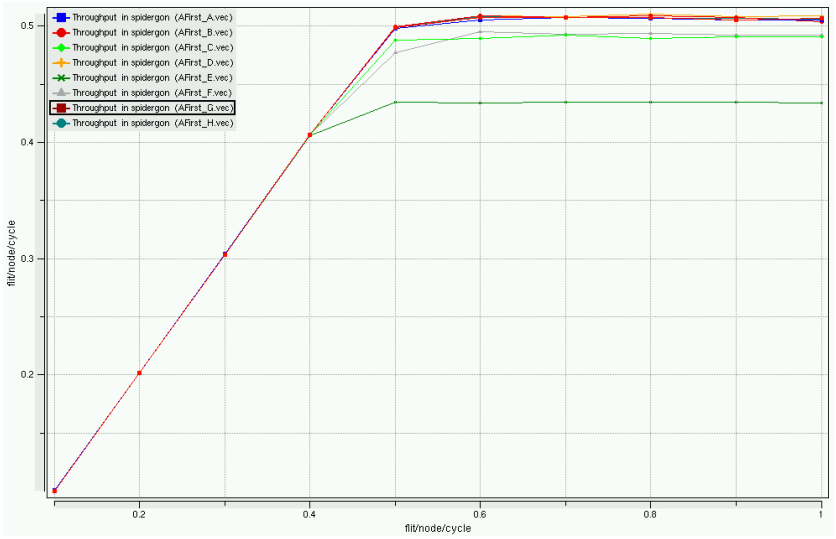

Figure 7.4: Throughput measure for mappings in figure 7.3

point of the network (that is what these estimations are all about). In fig 7.5 the gap in performance between case D and E is clearer; network mapping D saturation point is about 0.5 flit/cycle/node whilst for case E is 0.4; a significant factor, given that its value resorts just to how IPs are logically placed on the on-chip network.

Several mapping tools such as [53][39] are available in literature, with different characteristics and capabilities. While mapping subject is out of scope of this thesis, it is worth pointing out that, as in multi-layer busses, mapping is a compelling step of the design flow that strictly affects interconnection performance. Bad mapping of IPs on a distributed interconnection can cause severe performance degradation, so that it must be considered early in the design process as a vital factor of success to the system performance. Mapping of IPs on Network Interfaces is a hot research topic that deserves further study in order to become really mature to be deployed in real world systems on a chip.

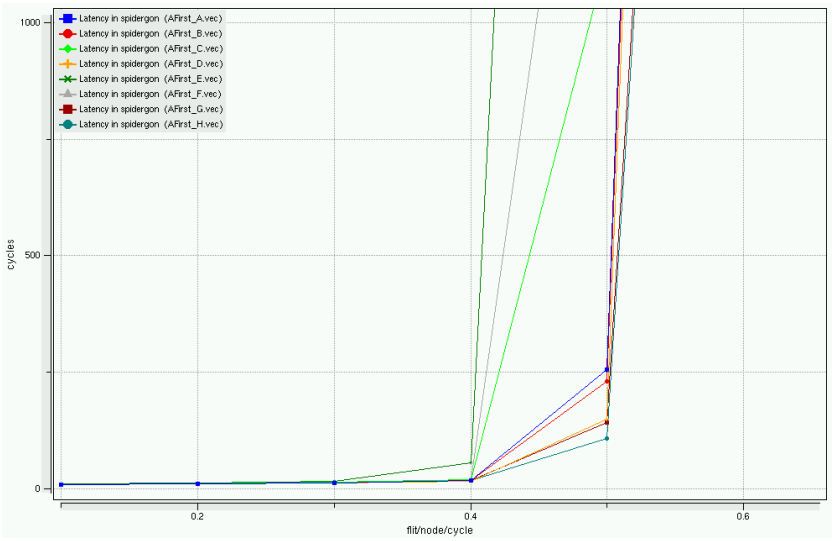

Figure 7.5: Latency measure for mappings in figure 7.3

7.3 STNoC™ benchmarking of real applications

The simulation environment described throughout this document has been successfully used to simulate and benchmark real SoC products by STMicroelectronics. Two chips for High-Definition Digital Television (HDTV)[6] have been successfully and thoroughly benchmarked using our NoC simulation environment. Just to provide additional insights, these chips integrate more than 50 bus initiators, with an interconnection hierarchy made up of several layers. In order to talk numbers, the interconnection was comprised of tenths of routers and STBus nodes, including a memory controller slave co-simulated in Verilog. A run of this kind of platform normally takes minutes to run [7], with all probes on (vcd files and quality of services statistics). Writing on probes implies system calls, which in turn means that simulation speed is system calls dominated. The C++ simulator by itself runs at very high-speed ($\sim$ 100 Kcycles/sec) compared to RTL simulations that take hours if not days to execute, and there is still room for code improvements. The effectiveness of TLM modeling proved to be quite staggering after all. In order to highlight what kind of benchmarks are possible using STNoC™

[6]It is worth noting that the simulator is general purpose, namely any kind of chip could be simulated with clock-accuracy; it is not limited to subclasses of applications.

[7]On an Intel Xeon box running at 2.8 Ghz, Linux kernel 2.4.

simulator apart from random traffic characterization, a case study is explained in this section.
In figure 7.6 an STNoC™ layout is shown. In figure 7.6, two clusters of ini-

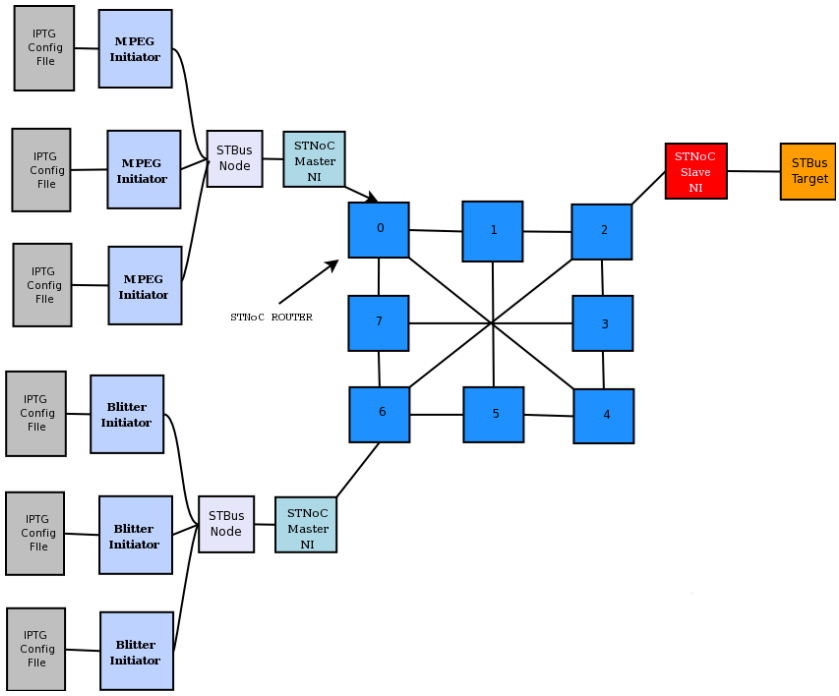

Figure 7.6: An STNoC™ case study

tiators are connected to STNoC™ backbone through two STBus nodes working as traffic concentrators, namely they coalesce different somewhat related traffic behaviors into one stream directed towards an STBus target. The interconnect must be tailored in a way that guarantees MPEG (see section 6.3, on page 156) the required Quality of Service support from the interconnection (bandwidth met plus bounded acceptable latencies).
The Blitter application is a graphic co-processor, whose role is to offload processors for tasks such as graphic motion and scrolling and whose requirements can be easily satisfied using the leftover bandwidth of other applications.
The simple platform was built using our NoC explorer, based on XML configuration (see figure 6.4 on page 155). In figure 7.7 a QoS proof of concept is shown. The two VCD waveforms display fifo occupancy of the IP generating traffic. In this specific case an MPEG reader was chosen. A FIFO level of zero means an IP in a starvation state, namely read responses corresponding to requests sent over the network have not come back on time to avoid IP idling, implying a lack of bandwidth on the interconnection. In particular,

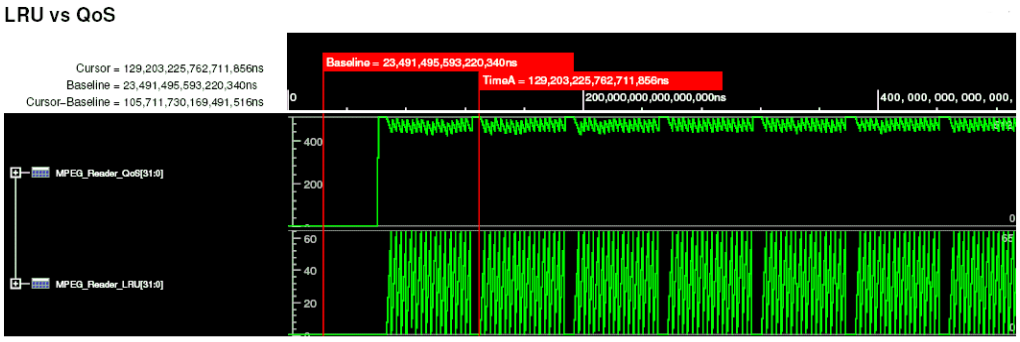

Figure 7.7: MPEG QoS proof of concept, FIFO starvation avoidance

the two waveforms compare fifo occupancy in the cases where LRU arbitration was used in STNoC™ arbiters (QoS off) to a case in which QoS knobs of STNoC™ were turned on to regulate traffic smoothness.
The simulations outcomes are overly clear: LRU arbitration is not effective, leading to periodical starvation of IP (points in which FIFO occupancy crosses zero). QoS knobs of STNoC™ proved effective, in that FIFO occupancy is high, near full, all along the simulation time corresponding to steady state.
As described in section 6.3 on page 156, MPEG is a particular kind of traffic, whose behaviour strictly depends on the compressed images. In order to get a complete proof of concept, in figure 7.8 a comparison similar to the MPEG one is shown, in a network layout identical to the one in figure 7.6, replacing MPEG initiators with high-definition, uncompressed streams, with a more predictable behavior in terms of traffic. In this case, due to the regular nature of uncompressed streams, the outcomes are even clearer. The LRU arbitration mechanism turned out to provide really bad performance, with steep slope towards zero crossing of FIFO occupancy (steep slope means more zero crossings per time unit). Turning on the QoS knobs[8] the FIFO occupancy is in a stable, still zero slope state, all along the simulation. This result turns out to be pivotal to the interconnection, and therefore to the the chip, performance. The interconnection network is able to ”feed” the IP regularly with the data it needs, meaning a perfect behavior.

So far, we concentrated on FIFO occupancy properties. To highlight more features of NoC explorer, it is time to discuss latency measures. Latency

[8]These features cannot be described due to confidentiality reasons. Additional insights might be got within STNoC™ architecture documentation.

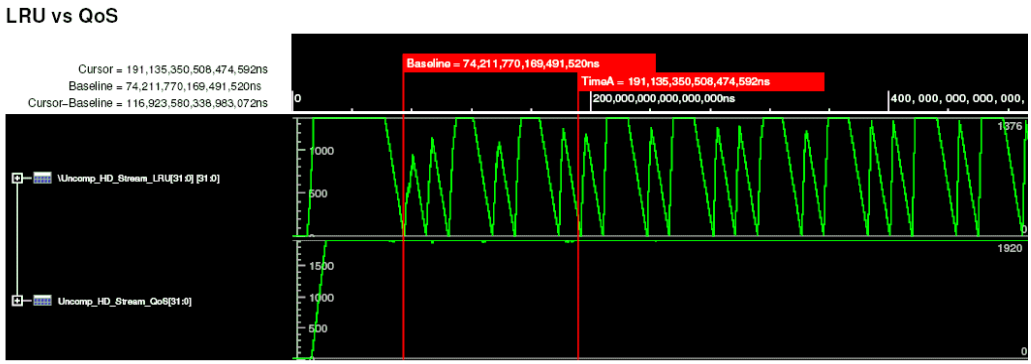

Figure 7.8: High-Definition (HD) QoS proof of concept, FIFO starvation avoidance

measures are obtained using SysProbe tool developed by STMicroelectronics. Latency can be measured per initiator in a number of flavors (request to grant, request to response, etc.) and results exported as plain text files.

```
STbus type : 3
Data size: 64 bits
Simulation Start Time: 0 ps
Simulation End Time: 1499998000 ps
Simulation duration: 1499998000 ps
Clock period: 4000 ps
STbus Frequency: 250.00 Mhz
STbus Clock cycles 374998
Window Time Frame 10
LATENCY :
      Name          Min     Max
      Req2R_Req      23      67
      Req2R_Eop      26      67
THROUGHPUT (MB/s):
      Data Throughput :            32.4 MB/s
      Full Data Throughput :       32.4 MB/s
BANDWIDTH :
      Max Available Bandwidth       2000.0 MB/s
      Max Available Bandwidth Req    2000.0 MB/s
      Max Available Bandwidth Resp   1600.00 MB/s
      Real_Bandwidth               32.4 MB/s
```

MPEG reader statistics LRU arbiters

In the embedded snapshot the most interesting fields of Sysprobe concerning NoC benchmarking are reported. Some values are computed statically (e.g. max available bandwidth) whilst others are sampled dynamically. The snapshot above reports outcomes of an MPEG read process initiator whose bandwidth is around 30 Mbytes/sec. As it is an STBus initiator, latency is measured as delay between assertion of request and response request, in number of cycles.
To be noted, that the maximum available bandwidth of responses is limited by the serialization of transactions into flits. Header flits are not considered

as available bandwidth. As most of the opcodes are *load 32 bytes* in a 64 bits data path, one fifth of the bandwidth is wasted to send headers (a response packet is made up of five flits: a header plus four payloads 4x8=32 bytes). Other fields are self-explanatory.
It is worth noting the low maximum latency (67 clock cycles)[9]. This is due to the "synthetic"nature of the testbench, where just few initiators are involved for demonstration purposes.
In real STMicrolectronics chips, where initiators easily exceed 50 units, latency increases solid when all traffic is directed towards a single target (see figure 7.2 on page 165), with hundreds of cycles to take into account.

```
STbus type : 3
Data size: 64 bits
Simulation Start Time: 0 ps
Simulation End Time: 1499998000 ps
Simulation duration: 1499998000 ps
Clock period: 4000 ps
STbus Frequency: 250.00 Mhz
STbus Clock cycles 374998
Window Time Frame       10
LATENCY :
        Name            Min         Max
        Req2R_Req       23          42
        Req2R_Eop       24          42
THROUGHPUT (MB/s):
        Data Throughput :               32.4 MB/s
        Full Data Throughput :          32.4 MB/s
BANDWIDTH :
        Max Available Bandwidth         2000.0 MB/s
        Max Available Bandwidth Req     2000.0 MB/s
        Max Available Bandwidth Resp    1600.00 MB/s
        Real_Bandwidth                  32.4 MB/s
```

MPEG reader statistics QoS arbiters

The preceding snapshot is just there to report the benefits of QoS in interconnection latencies. There are two points worth mentioning:

Maximum latency is the real critical parameter. Initiators FIFOs are designed to cope with worst cases; the lowest the maximum latency the least buffering is needed. This is vital to interconnection design.

40% gain is not a neglectable factor, even in these simple simulations. This is the essence of the whole thesis. Quality of service is a key component of future systems on-chip design. For sure, Quality of service is not a magic wand, but it still empowers designers with a design feature to implement better and better interconnection network with less design overcommitments and more efficient area utilization.

[9]"Low" here is misused because it must be always compared to a relative measure. In real systems on-chip initiators could well end up suffering from hundreds of clock cycles of latency. Author's point of view is highly biased by these bullying latency measures, so that 67 cycles sound as a relief from pain.

The transaction level models of STNoC™, together with STMicroelectronics tools such as IPTG, Genkit and Sysprobe, allowed to create a flexible yet powerful environment, which can be used to design and test complex, hybrid (bus plus NoC) interconnection systems.

7.4 Conclusion

This chapter, through simple but still effective measures and comments proved our methodology effectiveness for the design of ***real*** networks on-chip. The measures reported were of two types:

- Random traffic characterization.
- Bandwidth and latency outcomes of real traffic streams.

Random traffic characterization was executed to determine and show generic topological properties of the STNoC™ network on-chip. It leveraged statistical properties of SystemC Verification Library through IPTG configuration. Real traffic streams mirror traffic behaviors used in STMicroelectronics divisions to shape real interconnections and provided a simple but still complete idea on how to design real interconnection networks through our methodology. All of the results reported in this chapter represent the end product of the thesis work. Modeling, architecture design and traffic mapping, all take part in the whole design flow which is currently in use at STMicroelectronics to design interconnection networks.
The next chapter draws the thesis' conclusion, with an eye ahead towards the future, introducing exciting challenges that SoC manufacturers have been brewing for interconnection designers.

Chapter 8

Conclusions and future work

8.1 Introduction

Networks on-chip underwent years of debates, discussions and consequently improvements. The main objective of this thesis was to highlight some details, facts, research perspectives about this fascinating technology for on-chip interconnections. By now, networks on-chip are ready for the fray.
So far, networks on-chip researchers, and STMicroelectronics ones are no exception, strove to convince that networks provide on-chip designers a suitable solution to the on-chip interconnection problem. Benchmarking was a pivotal instrument to consolidate network concept, to prove performance effectiveness and high-level estimations of parameters. Transaction-Level modeling proved to be a very mature simulation technique, through which a number of simulations with a score of variable parameters have been carried out. Networks on-chip have come a long way since their early days.
It is time to move ahead towards future challenges and issues to solve that are just appearing at the horizon.
Security is already a major concern for on-chip designers and networks on--chip must integrate some degree of flexibility to support security enhancements. Furthermore, systems on-chip have been integrating functionality through flexible components such as programmable CPUs. Flexibility calls for software development, and networks on-chip must be able to support this shift in the design paradigm through an efficient interconnection network. The NoC must somehow support this flexibility through features allowing a powerful programming model and QoS support for programming services.

The last chapter of this thesis is dedicated to these two emerging issues

(opportunities) that ultimately influence the way in which code and data are shuttled forth and back in a system on-chip design.

8.2 Security monitoring

The growing number of instances of breaches in information security in the last few years has brought about a compelling case for efforts towards secure electronic systems. Embedded systems that are by now ubiquitously in use to capture, store, manipulate and access data of sensitive nature, put several interesting security challenges. Security has been the subject of intensive research in the area of cryptography, computing and networking. In embedded world, it represents an entirely new metric that designers have to consider throughout the design process, along with other metrics such as cost, power and performance. Various attacks on electronic and computing systems have shown that crackers rarely target flaws of well-designed cryptographic algorithms. Instead, they rely on exploiting security vulnerabilities in the software and hardware components of the implementation. As shown

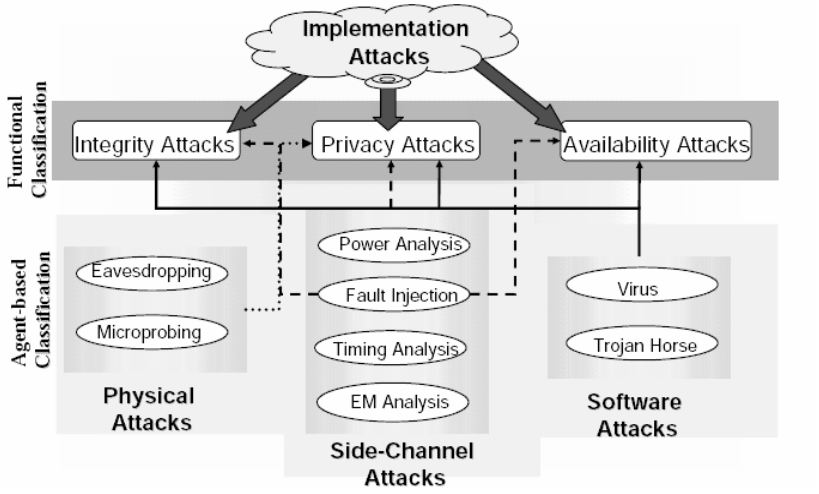

Figure 8.1: Security attacks classes

in figure 8.1 three kinds of security attacks have been classified [44]:

Software Attacks such as those against operating system kernels, represent a harmful example security breach. A kernel has full access to the system and can communicate with any part of the address space.

This means that an attacker can read or write to the BIOS memory on the motherboard or on a peripheral hardware. New BIOSes are implemented through Flash ROMs that can be re-written from software; these ROMs are rarely fully-utilized, unfortunately leaving room to store backdoor information and viruses.

Physical Attacks can be launched through probes to eavesdrop on inter-components communication. However, for systems on-chip, sophisticated micro-probing techniques become necessary, implying mandatory de-packaging. Physical attack at chip level are hard to use because their expensive infrastructure requirement. However they can be deployed once and then reused for subsequent non-invasive attacks.

Side Channel Attacks such as timing analysis and power analysis exploit information leakage on the chip. For instance, the operating current drawn by a hardware device is correlated to computations it is performing. Power consumption increases if more state transitions occurs. Power analysis attacks rely on the observation that in some systems, the power profile of cryptographic computations can be directly used to interpret the cryptographic key.

The evolution of on-chip macro architecture models have been proposed to provide adequate security functions. Basic security functions, like confidentiality, integrity and authentication, can be implemented with appropriate security protocols and cryptographic algorithms. A hardware-only approach uses ASICs to implement a given cryptographic algorithm in hardware. This approach is effective for few ciphers, but less effective in terms of cost and flexibility when a variety of ciphers are desired.
A software-only approach using a typical embedded general-purpose processor (EP) core for performing security protocol and cryptography processing. A number of hybrid hardware-software approaches have been proposed to efficiently implement security functions. In particular, interconnection schemes must be enhanced to support security attributes of transactions to allow security level monitoring in the interconnection core.
Through *observability* a system on-chip operating system is capable of tracking down the behavior of hardware tasks. As proposed in [49], the network interface plays a role of the utmost importance in monitoring transactions in that it is the glue that interfaces IPs to the communication medium. The role of the network interface is about checking transactions and possibly notify ”exceptions” to the software layer in order to take adequate countermeasures in case of security faults or violation.

Interested readers are directed to [28] to get additional insights about security monitoring in networks on-chip, a comprehensive overview of current security technologies applied to networks on-chip solutions.

8.3 Towards clustered NoC-based platforms

The demand for even more computing power keeps increasing in a staggering way. Since 1980, the complexity of the software as well as the scale and solution quality of applications have continuously driven the development of even faster processors. A number of important problems have been identified in the areas of defense, aerospace, automotive applications and science, whose solutions require a tremendous amount of computational power.
In order to provide a solution to these tough challenge problems, attention is turning more and more to computer systems capable of computing at teraflops (10^{12} floating-point operations per second) level. Even the simplest of these problems requires gigaflops (10^{9} floating-point operations per second) of performance for hours at a time. Parallel computers with multiple processors are opening the door to teraflops computing performance to meet the increasing demand of computational power. The demand comprehends more computing power, higher network and input/output (I/O) bandwidths, and more storage capacity.
Processors are becoming very complex. As an aftermath, processor design cost is growing so fast that only a few companies all over the world can afford to design a new processor. A possible and feasible alternative choice consists in designing parallel computers from legacy components (processors, memories, interconnects). Distributed-memory multiprocessors or multicomputers clusters can be built using this approach. This cluster consists of a set of processors, each one endowed with its own local memory (see figure 4.11 on page 74). Processors communicate between themselves by forwarding messages through an interconnection network.
Programming multicomputers turned out to be an exacting task. The programmer has to take care of distributing code and data among the processors in an efficient way, calling message passing code (e.g. MPI [41] API) whenever some data are needed by other processors.
On the other hand, *shared-memory multiprocessors* provide a single memory space to all the processors; in this way, the memory sharing decreases the latency penalties due to the exchange of data among them. Access to shared memory has been traditionally implemented by using an interconnection network between processors and memory (see figure 4.10 on page 74).

This architecture is referred to as *uniform memory access (UMA)* architecture. It is not scalable in that memory access time increases as system grows in size.
Recently, shared-memory multiprocessors followed some trends previously established for multicomputers. In particular, memory has been physically split among processors, therefore reducing the memory access time for local accesses and improving scalability. These parallel architectures are reported in literature as *distributed shared-memory multiprocessors (DSMs)*. Access to remote memory are performed through an interconnection network. The main difference between DSMs and multicomputers is that in DSM machines messages are initiated by memory accesses (i.e. page faults); in multicomputers messages are typical networks packets (e.g. MPI data).
Caching allows to reduce memory latency; in particular, cache memories are organized in a hierarchical layout with several levels. This architecture provides *non-uniform memory access (NUMA)* time. The main problem in DSM consists in maintaining caches coherent.
Although there are many similarities between interconnection networks for multicomputers and DSMs, it is important to bear in mind that performance requirements may be very different. Messages are usually very short when DSMs are used. Moreover, network latency is important because the time it takes to map and unmap pages of memory depends on that latency. Messages (e.g. send receive MPI packets) are typically longer and less frequent when using multicomputers and it is up to the programmer to adjust the granularity of messages.

Interconnection networks, and networks on-chip in this context, play a major role in the performance of modern parallel computers. STNoC™ is going to be deployed as interconnection network within the SHAPES project (Scalable Software Hardware Architecture Platform for Embedded Systems http://shapes.atmelroma.it), a scalable software and hardware architecture for current and forthcoming embedded applications. In figure 8.2, a realistic mock-up of a multi-tile parallel embedded architecture is shown. The heterogeneous tile in figure 8.2 is composed of a VLIW floating-point DSP[1], a RISC machine, on-chip memory, and a network interface. It includes a few million gates, for optimal balance among parallelism, local memory and IP reuse for future technologies.
So far, the challenges posed on STNoC™ were mostly ”hardware” pushed; software engineering was not taken into account throughout STNoC™ benchmarking just because software components in benchmarked STMicroelectronics chips were hidden in low-level subsystems. At top level, the goal of the interconnection was ”just” to provide enough bandwidth with limited latency

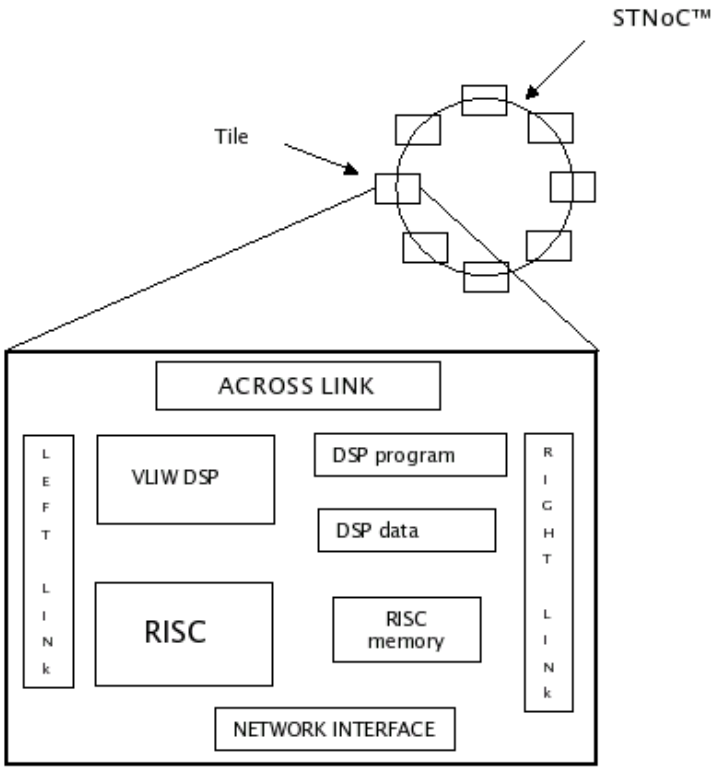

Figure 8.2: STNoC™ in a multi-tile parallel architecture

to hardwired IP, with no strings attached.
Given the shift of chips towards parallel computer architectures, software development is going to strictly influence the interconnection design.
Programming model goodness (e.g. MPI) depends on how effective the interconnection is in providing required QoS and fairness, forcing the NoC to supply some kind of on-purpose services to software APIs. Through dedicated resources STNoC™ supports classes of services to build stacked quality of service features on top of the bare interconnection medium. The real challenge looming large at the horizon consists in taking into account interconnection medium for software development performance early in the design cycle, not as an afterthought. Scalable programming models effectiveness for parallel machines such as SHAPES [1] rely on a scalable, programmable and flexible interconnection network such as STNoC™ to provide connection schemes able to support the tremendous pressure exerted on software development performance.

8.4 Conclusion

This chapter wrapped up and achieved the thesis work. Two important issues that have to be dealt with were stated, in order to underline current and future activities in the STNoC™ development process.
Security is a subject of the utmost importance in all steps of design flow; interconnections play an important role in guaranteeing secure transactions over the chip, by adding special opcodes recognizable by network interfaces and so by upper layers in the chip stack. The first section of this chapter described briefly security implications on on-chip networks.
The massive need for parallelism is setting new challenges for on-chip designers that have to deal with multi-processors architectures and related programming models. The network, as it is the medium that allows sending and receiving code and data, represents a paramount component to guarantee programming model efficiency and effectiveness. Section 8.3 highlighted STNoC™ development on this precise direction. Future STNoC™ enhancements will be strongly biased by software requirements in order to focus the research on features that may provide valuable support to this complex shift in the design paradigm.

Bibliographie

[1] F. Aglietti et al. *"The APEmille supercomputer: Linux in the Theoretical High Energy Physics"*. In *Proceeedings of the 5th annual Linux expo, Rayleigh,North Carolina*, pages 71–84, March 1999. 179, 181

[2] *"AMBA specification rev. 2.0"*. available at http://www.arm.com. 6, 8, 70

[3] *"AMBA AXI protocol"*. specification available at http://www.arm.com. 38

[4] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. *"Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism"*. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. 141

[5] A. Andriahantenaina. *"Implémentation matérielle d'un micro-réseau SPIN à 32 ports"*. PhD thesis, Université Paris VI, Janvier 2006. 42

[6] A. Andriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino. *"SPIN: A Scalable, Packet switched, On-Chip Micronetwork"*. In *Proc. Design Automation and Test in Europe Conf.*, pages 70–73, 2003. 60

[7] *"ARM Realview ™ ESL interfaces"*. available at http://www.arm.com/products/DevTools/ESLmodelinterfaces.html. 95

[8] *"A Comparison of Network-on-Chip and Busses"*. white paper available at http://www.arteris.net. 48, 54

[9] L. Benini and G. De Micheli. *"Networks On-Chip: A New SOC Paradigm"*. *IEEE Transactions on Computers*, 35:70–78, 2002. 53

[10] D. Bovet and M. Cesati. *"Understanding the LINUX kernel"*. O'Reilly, third edition, 2005. 75, 141

[11] D. Butenhof. *"Programming with POSIX threads"*. Addison-Wesley professional, 1997. 75, 141, 145

[12] *"NC-Sim simulator"*. documentation available at http://www.cadence.com. 96, 130, 137, 156

[13] M. Caldari, M. Conti, M. Coppola, P. Crippa, S. Orcioni, L. Pieralisi, and C. Turchetti. *"System-Level Power Analysis Methodology Applied to AMBA AHB Bus"*. In *Proc. Design, Automation and Test Conference, Munich*, 2003. 119

[14] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. *"Transaction-Level Model of AMBA Bus architecture using SystemC 2.0"*. In *Proc. Design, Automation and Test Conference, Munich*, 2003. 119

[15] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. *"Theory of Latency-Insensitive Design"*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(9), September 2001. 29

[16] W. Cesario et al. *"Multiprocessor SoC Platforms: A Component Based Design Approach"*. *IEEE Design and test of Computers*, 19(6), November 2002. 96

[17] M. Coppola, S. Curaba, M. Grammatikakis, G. Maruccia, and F. Papariello. *"The OCCN user manual"*. Available at http://occn.sourceforge.net. 15, 96, 99

[18] M. Coppola, Curaba.S., M. Grammatikakis, and G. Maruccia. *"IPSIM: SystemC 3.0 Enhancements for Refinement"*. In *Proc. Design Automation and test in Europe Conf.*, pages 106–111, 2003. 119

[19] *"Coware Platform Architect"*. documentation available at http://www.coware.com/products/platformarchitect.php. 96

[20] W. Dally and B. Towles. *"Route Packets, Not Wires: On-Chip Interconnection Networks"*. In *DAC 2001, Las Vegas, Nevada, USA*, June 2001. 53

[21] W. Dally and B. Towles. *"Principles and Practices of Interconnection Networks"*. Morgan Kaufmann, 2003. 21, 56

[22] H. Deitel and P. Deitel. *"C++ : How to Program"*. Prentice Hall, 1998. 97

[23] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *"The Source Book of Parallel Computing"*. Morgan Kaufmann, 2003. 75

[24] J. Duato, S. Yalamanchili, and L. NI. *"Interconnection Networks"*. Morgan Kaufmann, 2003. 54, 59

[25] B. Eckel. *"Thinking in C++ Volume 1: Introduction to Standard C++"*. Eckel,B., 2000. 97, 133

[26] K. Faraydon, A. Nguyen, S. Dey, and R. Rao. *"On-Chip Communication Architecture for OC-768 Network Processors"*. In *Proceedings Design Automation Conference Las Vegas, Nevada, USA*, June 2001. 61

[27] D. Gajsky, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *"SpecC: Specification Language and Methodology"*. Kluwer Academic, 2000. 96

[28] C. Gebotys and R. Gebotys. *"A Framework for Security on NoC Technologies"*. In *Proc. IEEE Society Annual Symposium on VLSI*, 2003. 178

[29] P. Gevros, J. Crowcroft, P. Kirstein, and S. Bhatti. *"Congestion Control Mechanism and the Best Effort Service Model"*. *IEEE Network*, May 2001. 67

[30] K. Goossens, O. Gangwal Prakash, J. Roever, and A. Niranjan. *"Interconnect and Memory Organization in SoCs for Advanced Set-Top Boxes and TV – Evolution, Analysis and Trends"*. In J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, editors, *"Interconnect-Centric Design for SOC and NOC"*, pages 399–423. Kluwer, 2004. 31

[31] K. Goossens and A. Rădulescu. *"Communication Services for Networks On Silicon"*. In S. Bhattacharya, E. Deprettere, and J. Teich, editors, *"Domain-Specific Processors: Systems, Architectures, Modeling and Simulation"*, pages 275–299. Marcel Dekker, 2003. 31

[32] A. Grasset, F. Rousseau, and A. Jerraya. *"Network Interface Generation for MPSoC: from Communication Service Requirements to RTL Implementation"*. In *15th IEEE International Workshop on Rapid System Prototyping (RSP), Geneva, Switzerland*, June 2004. 69

[33] T. Grötker, S. Liao, G. Martin, and S. Swan. *"System Design with SystemC"*. Kluwer Academic, 2002. 16, 19, 88

[34] P. Guerrier. *"Un réseau d'interconnexion pour systèmes intégrés"*. PhD thesis, Université Paris VI - Pierre et Marie Curie, UFR d'Informatique, 2000. 2, 42

[35] P. Guerrier and A. Greiner. *"A Generic Architecture for On-Chip Packet Switched Interconnections"*. In *Proc. Design,Automation and Test in Europe*, pages 250–256, 2000. 43

[36] H. Haverinen, M. Leclercq, N. Weyrich, and D. Wingard. *"SystemC based SoC Communication Modeling for the OCP protocol"*, October 2002. white paper available at http://www.ocpip.org. 96

[37] R. Hyde. *"The Art of Assembly Language"*. No Starch Press, 2003. 75

[38] R. Hyde. *"Write Great Code"*, volume 1. No Starch Press, 2004. 104, 134

[39] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. *"XpipesCompiler: A Tool for Instantiating Application Specific Networks On Chip"*. In *Proc. Design, Automation and Test in Europe Conf.*, 2004. 168

[40] A. Jantsch and H. Tenhunen. *"Networks on Chip"*. Kluwer Academic Publisher, 2003. 3, 28, 29

[41] G. Karniadakis and R. Kirby. *"Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation"*. Cambridge Universisty Press, 2003. 178

[42] D. Keppel. *"Tools and Techniques for Building Fast Portable Thread Packages"*. Technical report, University of Washington, UWCSE 93-05-06, 1993. 93, 98

[43] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. *"System Level Design: Orthogonalization of Concerns and Platform-Based Design"*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 19(12):1523–1543, December 2000. 86

[44] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. *"Security as a New Dimension in Embedded System Design"*. In *Proc. Design and Automation Conference, San Diego, California, USA*, June 2004. 176

[45] S. Kumar, Jantsch.A., J. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. *"A Network On-Chip Architecture and Design Methodology"*. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, April 2002. 31

[46] P. Li-Shuan and W. Dally. *"Flit-Reservation Flow Control"*. In *Proceedings of the 6th international symposium on high-performance computer architecture, Toulouse, France*, pages 73–84, January 2000. 56

[47] R. Love. *"Linux Kernel Development"*. Novell Press, 2005. 75, 119, 141

[48] "Linux Scalability Effort", http://lse.sourceforge.net. 147

[49] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, and R. Lauwereins. *"Networks on-chip as hardware components of an OS for Reconfigurable Systems"*. In *Proc. FPL*, 2003. 177

[50] S. Meftali et al. *"SOAP Based Distributed Simulation Environment for System-on-Chip (SoC) design"*. In *Forum on Specification and Design Languages*, 2005. 148

[51] S. Meyers. *"Effective C++: 50 Specific Ways to Improve Your Program and Design"*. Addison-Wesley, Reading, MA, 1997. 104

[52] G. Moore. *"Cramming more Components onto Integrated Circuits"*. *Electronics*, 38(8), April 1965. 87

[53] S. Murali and G. De Micheli. *"SUNMAP: A Tool for Automatic Topology Selection and Generation NoCs"*. In *Proceedings of Design and Automation Conference, San Diego, California, USA*, June 2004. 168

[54] *"OCP IP Data Sheet"*. available at http://www.ocpip.org. 41

[55] "Oprofile: A system wide profiler for Linux Systems", http://oprofile.sourceforge.net. 142

[56] G. Palermo and C. Silvano. *"PIRATE: A Framework for Power/Performance Exploration of Network-On-Chip Architectures"*. In *Proc. Int. Workshop on Power and Timing modeling (PATMOS)*, pages 521–531, 2004. 120

[57] S. Pasricha, N. Dutt, and M. Ben-Romdhane. *"Constraint-Driven Bus Matrix Synthesis for MPSoC"*. In *Proceedings of ASP-DAC, Yokohama, Japan*, January 2006. 59

[58] D. Patterson and J. Hennessy. *"Computer Architecture: A Quantitative Approach"*. Morgan Kaufmann, 2003. 75

[59] L. Peterson and B. Davie. *"Computer Networks"*. Morgan Kaufmann, 2003. 54

[60] *"QT 4.0 white paper"*. Available at http://www.trolltech.com. 154

[61] A. Rădulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage. *"An Efficient On-Chip Network Interface offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming"*. In *Proc. of Design, Automation and Test Conference in Europe*, February 2004. 82

[62] A. Rose, S. Swan, J. Pierce, and J. Fernandez. *"Transaction-Level Modeling in SystemC"*, 2004. white paper available at http://www.systemc.org. 97

[63] T. Rose. *"The OpenBook: A Practical Perspective on OSI"*. Prentice Hall, 1990. 16, 53

[64] J. Rowson and A. Sangiovanni-Vincentelli. *"Interface Based Design"*. In *Proceedings of the Design Automation Conference*, pages 178–183, 1997. 96

[65] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. *"Addressing the System-on-a-chip Interconnect Woes Through Communication Based Design"*. In *Proceedings of the 38th Design Automation Conference*, June 2001. 41, 93

[66] *"SiliconBackplane III Micronetwork IP"*. available at http://www.sonicsinc.com. 40

[67] *"STBUS Communication System: Concepts and Definition"*. STMicroelectronics internal document. 8, 37, 76

[68] B. Stroustrup. *"The C++ Programming Language"*. Addison-Wesley, 2000. 97, 104

[69] D. Sylvester. *"A Global Wiring Paradigm for Deep Submicron Design"*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 19(2), February 2000. 29

[70] *"System Studio"*. documentation available at http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html. 95

[71] *"Functional Specification for SystemC 2.0"*, 2001. Available at http://www.systemc.org. 87

[72] *"SystemC User's Guide"*, 2001. Available at http://www.systemc.org. 87

[73] A. Tanenbaum. *"Modern Operating Systems"*. Prentice Hall, 2001. 141

[74] A. Tanenbaum. *"Computer Networks"*. Prentice Hall, 2003. 69, 157

[75] M. Taylor et al. *"The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs"*. *IEEE Micro*, March 2002. 45

[76] T. Theis. *"The Future of Interconnection Technology"*. *IBM Journal of Research and Development*, 44(3), 2000. 28

[77] R. Thid, M. Millberg, and A. Jantsch. *"Evaluating NoC Communication Backbones with Simulation"*. In *Proceedings of the IEEE NorChip*, November 2003. 87

[78] A. Valentian and A. Amara. *"On-Chip Signaling for Ultra Low-Voltage 0.13µm CMOS SOI Technology"*. In *Proc. NEWCAS, Montreal, Canada*, June 2004. 53

[79] J. Wu. *"Fault-Tolerant Adaptive and Minimal Routing in Mesh-Connected Multicomputers Using Extended Safety Levels"*. *IEEE Transactions on Parallel and Distributed Systems*, 11(2), February 2000. 115

[80] M. Yang and L. M. Ni. *"Design of Scalable and Multicast Capable Cut-Through Switches for High-Speed LANs"*. In *Proceedings of the International Conference on Parallel Processing*, pages 324–332, August 1997. 57

[81] X. Zhu and S. Malik. *"A Hierarchical Modeling Framework for On-Chip Communication Architectures"*. In *Proceedings of International Conference on Computer-Aided Design 2002*, November 2002. 96