Design, Parallel Simulation and Implementation of High-Performance Fault-Tolerant Network-on-Chip Architectures

Mohamed El Amir Charif

▶ **To cite this version:**

Mohamed El Amir Charif. Design, Parallel Simulation and Implementation of High-Performance Fault-Tolerant Network-on-Chip Architectures. Micro and nanotechnologies/Microelectronics. Université Grenoble Alpes, 2017. English. NNT : 2017GREAT075 . tel-01743726

HAL Id: tel-01743726

https://theses.hal.science/tel-01743726v1

Submitted on 26 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES

Arrêté ministériel : 25 mai 2016

Présentée par

Mohamed El Amir CHARIF

Thèse dirigée par **Michel NICOLAIDIS**, Directeur de Recherche , , et
codirigée par **Nacer-Eddine ZEIRGAINOH**

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

Conception, Simulation Parallèle et Implémentation de réseaux sur puce hautes performances tolérants aux fautes

Design, Parallel Simulation and Implementation of High-Performance Fault-Tolerant Network-on-Chip Architectures

Thèse soutenue publiquement le **17 novembre 2017**,
devant le jury composé de :

Monsieur Nacer-Eddine ZERGAINOH
Maître de Conférences, Université Grenoble Alpes, CoDirecteur de these
Monsieur Frédéric PETROT
Professeur, TIMA, Président
Monsieur Ian O'CONNOR
Professeur, Ecole Centrale de Lyon, Rapporteur
Monsieur Bruno ROUZEYRE
Professeur, Université Montpellier 2, Rapporteur

Abstract

Networks-on-Chip (NoCs) have proven to be a fast and scalable replacement for buses in current and emerging many-core systems. They are today an actively researched topic and various solutions are being explored to meet the needs of emerging applications in terms of performance, quality of service, power consumption, and fault-tolerance. This thesis presents contributions in two important areas of Network-on-Chip research:

- The design of ultra-flexible high-performance deadlock-free routing algorithms for any topology.
- The design and implementation of parallel cycle-accurate Network-on-Chip simulators for a fast evaluation of new NoC architectures.

While aggressive technology scaling has its benefits in terms of delay, area and power, it is also known to increase the vulnerability of circuits, suggesting the need for fault-tolerant designs. Fault-tolerance in NoCs is directly tied to the degree of flexibility of the routing algorithm. High routing flexibility is also required in some irregular topologies, as is the case for TSV-based 3D Network-on-Chips, wherein only a subset of the routers are connected using vertical connections. Unfortunately, routing freedom is often limited by the deadlock-avoidance method, which statically restricts the set of virtual channels that can be acquired by each packet.

The first part of this thesis tackles this issue at the source and introduces a new topology-agnostic methodology for designing ultra-flexible routing algorithms for Networks-on-Chips. The theory relies on a novel low-restrictive sufficient condition of deadlock-freedom that is expressed using the local information available at each router during runtime, making it possible to verify the condition dynamically in a distributed manner. A significant gain in both performance and fault-tolerance when using our methodology compared to the existing static channel partitioning methods is reported. Moreover, hardware synthesis results show that the newly introduced mechanisms have a negligible impact on the overall router area.

In the second part, a novel routing algorithm for vertically-partially-connected 3D Networks-on-Chips called First-Last is constructed using the previously presented methodology. Thanks to a unique distribution of virtual channels, our algorithm is the only one capable of guaranteeing full connectivity in the presence of one TSV pillar in an arbitrary position, while requiring a low number of extra buffers (1 extra VC in the East and North directions). This makes First-Last a highly appealing cost-effective alternative to the state-of-the-art Elevator-First algorithm.

Finally, in an aim to speed up the evaluation of new Network-on-Chip designs, the third and last part of this work presents the first detailed and modular parallel NoC simulator design to run fully on Graphics Processing Units (GPUs). First, a flexible task decomposition approach, specifically geared towards high parallelization is proposed. Our approach makes it easy to adapt the granularity of parallelism to match the capabilities of the host GPU. Second, all the GPU-specific implementation issues are addressed and several optimizations are proposed. Our design is evaluated through a reference implementation, which is tested on an NVidia GTX980Ti graphics card and shown to speed up 4K-node NoC simulations by almost 280x.

Résumé

Introduction

Grâce à une réduction considérable dans les dimensions des transistors, les systèmes informatiques sont aujourd'hui capables d'intégrer un très grand nombre de coeurs de calcul en une seule puce (System-on-Chip, SoC). Faire communiquer les composants au sein d'une puce est aujourd'hui assuré par un réseau de commutation de paquets intégré, communément appelé Network-on-Chip (NoC). Cependant, le passage à des technologies de plus en plus réduites rend les circuits plus vulnérables aux fautes et aux défauts de fabrication. Le réseau sur puce peut donc se retrouver avec des routeurs ou des liens non-opérationnels, qui ne peuvent plus être utilisés pour le routage de paquets. Par conséquent, le niveau de flexibilité offert par l'algorithme de routage n'a jamais été aussi important.

La première partie de cette thèse consiste à proposer une méthodologie généralisée, permettant de concevoir des algorithmes de routage hautement flexibles, combinant tolérance aux fautes et hautes performances, et ce pour n'importe quelle topologie réseau. Cette méthodologie est basée sur une nouvelle condition suffisante pour l'absence d'interblocages (deadlocks) qui, contrairement aux méthodes existantes qui imposent des restrictions importantes sur l'utilisation des buffers, s'évalue de manière dynamique en fonction de chaque paquet et ne requiert pas un partitionnement stricte des canaux virtuels (virtual channels). Il est montré que ce degré élevé de liberté dans l'utilisation des buffers a un impact positif à la fois sur les performances et sur la robustesse du NoC, sans pour autant augmenter la complexité en termes d'implémentation matérielle.

La seconde partie de la thèse s'intéresse à une problématique plus spécifique, qui est celle du routage dans des topologies tri-dimensionnelles partiellement connectées, qui vont vrais-

semblablement être en vigueur à cause du coût important des connexions verticales, réalisées en utilisant la technologie TSV (Through-Silicon Via). Cette thèse introduit un nouvel algorithme de routage pour ce type d'architectures nommé "First-Last". Grâce à un placement original des canaux virtuels, cet algorithme est le seul capable de garantir la connectivité totale du réseau en présence d'un seul pilier de TSVs de coordonnées arbitraires, tout en ne requérant de canaux virtuels que sur deux des ports du routeur. Contrairement à d'autres algorithmes qui utilisent le même nombre total de canaux virtuels, First-Last n'impose aucune règle sur la position des piliers, ni sur les piliers à sélectionner durant l'exécution. De plus, l'algorithme proposé ayant été construit en utilisant la méthode décrite dans la première partie de la thèse, il offre une utilisation optimisée des canaux virtuels ajoutés.

L'implémentation d'un nouvel algorithme de routage implique souvent des changements considérables au niveau de la microarchitecture des routeurs. L'évaluation de ces nouvelles solutions requiert donc une plateforme capable de simuler précisément l'architecture matérielle du réseau au cycle près. De plus, il est essentiel de tester les nouvelles architectures sur des tailles de réseau significativement grandes, pour s'assurer de leur scalabilité et leur applicabilité aux technologies émergentes (e.g. intégration 3D). Malheureusement, les simulateurs de réseaux sur puce existants ne sont pas capables d'effectuer des simulations sur de grands réseaux (milliers de coeurs) assez vite, et souvent, la précision des simulations doit être sacrifiée afin d'obtenir des temps de simulation raisonnables. En réponse à ce problème, la troisième et dernière partie de cette thèse est consacrée à la conception et au développement d'un modèle de simulation générique, extensible et parallélisable, exploitant la puissance des processeurs graphiques modernes (GPU). L'outil développé modélise l'architecture d'un routeur de manière très précise et peut simuler de très grands réseaux en des temps records.

Dans ce qui suit on décrit plus en détail chacune des solutions proposées.

Contribution I: Une nouvelle théorie pour la prévention d'interblocages dans les réseaux

Certaines parties de cette contribution ont fait l'objet de publications dans des conférences internationales Charif et al. [2016], Charif et al. [2017b].

Une approche très souvent adoptée pour augmenter le degré d'adaptabilité d'un algorithme de routage sans provoquer d'interblocages (deadlocks) consiste à introduire plusieures files de paquets par canal physique. Ces files d'entrée sont appelées canaux virtuels, ou Virtual Channels (VC). Une majeure partie des algorithmes adoptés dans les NoCs aujourd'hui Bahmani et al. [2012], Salamat et al. [2016b] sont conçus suivant d'anciennes théories telles que celle de Dally Dally and Seitz [1988], ou des méthodologies basées sur ces théories Ebrahimi and Daneshtalab [2017]. Dans ces approches, les canaux virtuels servent à diviser le réseau en plusieurs sous-réseaux (Sub-Networks), de manière à ce que les paquets présents dans chaque sous-réseau utilisent un algorithme de routage sans dépendances cycliques (cycle-free).

On identifie plusieurs problèmes avec cette approche. Premièrement, puisque la méthode même est basée sur la séparation stricte des différents sous-réseaux, les canaux qu'un paquet peut occuper à une étape donnée sont définis de manière statique. Cela signifie qu'un paquet ne peut utiliser un canal virtuel qui n'appartient pas à son sous-réseau actuel, même si ce canal est disponible. Sans cette restriction, les canaux virtuels pourraient être utilisés pour réduire le blocage des paquets et augmenter la bande passante du réseau. Deuxièmement, pour garantir l'absence de dépendances cycliques, cette méthode impose un ordre strict sur la traversée des différents sous-réseaux. Cela implique qu'un paquet ne peut jamais visiter le même sous-réseau deux fois, ce qui réduit la flexibilité du routage, et donc la tolérance aux fautes.

Ces limitations sont principalement dues au fait que dans toutes ces théories, des "sous-fonctions" de routage sont associées à un ensemble précis de canaux virtuels. Un paquet ayant besoin d'une sous fonction particulière pour atteindre sa destination est donc obligé d'occuper le bon canal virtuel.

On propose une nouvelle vision du routage sans interblocages qui permet de résoudre toutes ces limitations à la fois. Au lieu d'associer une fonction de routage aux canaux, on les associe à des "classes" ou "groupes" de paquets. Chaque paquet du réseau transporte l'identifiant de son

groupe et est routé en fonction de cet identifiant. Aucun partitionnement des canaux virtuels n'est effectué, et les paquets peuvent occuper, à priori, n'importe quel canal virtuel sur le canal physique sélectionné par leur fonction de routage.

On définit alors la condition suffisante à l'absence d'interblocages comme suit: Un paquet doit sélectionner un canal physique qui comporte au moins un canal virtuel occupé par un paquet d'une classe égale ou supérieure à sa classe. Intuitivement, cela peut être expliqué de la manière suivante: si chaque paquet dans le réseau suit des paquets de son groupe, et sachant que les paquets d'un même groupe sont routés suivant la même fonction de routage qui ne contient pas de cycles, au moins un chemin de dépendances ne contenant aucun cycle existe pour chaque paquet du réseau, ce qui suffit pour prévenir les interblocages.

On présente alors une méthodologie généralisée, comprenant étapes de construction et règles de routage à différents niveaux de complexité, permettant la construction systématique d'algorithmes qui satisfont la condition minimum d'absence d'interblocages, et ce pour n'importe quelle topologie réseau. En effet, on s'appuyera sur cette méthodologie pour construire les algorithmes présentés dans la seconde partie de la thèse pour les réseaux 3D.

En plus d'être très peu restrictive par rapport aux conditions d'absence d'interblocages utilisées jusque là, notre condition permet de déduire des propriétés très puissantes. Par exemple, étant donné que la classe de routage de chaque paquet est connue, il est possible de savoir, au cas par cas, les paquets qui peuvent se suivre au sein d'un même canal virtuel. Des canaux virtuels non-vides peuvent alors être réalloués du moment que la condition d'absence d'interblocages n'est pas violée. Ceci n'est pas possible dans les autres théories qui permettent les dépendances cycliques telles que Duato [1995], où il faut attendre qu'un canal virtuel soit complètement vide avant de le réallouer à un nouveau paquet. Un gain significatif en performances a été démontré par les simulations effectuées. De plus, contrairement aux approches classiques qui ne permettent de traverser les sous-réseaux que dans l'ordre croissant, notre condition, étant formulée suivant les données dynamiques sur les classes des paquets, prévoit les changements de classe dans les deux sens, sans risque d'interblocages. Les résultats de simulation montrent que cela se traduit par un niveau accru de fiabilité qui n'est pas atteignable suivant les méthodes classiques qui ne permettent le changement de sous-réseau que dans un seul sens.

On montre que notre approche, en plus d'être utile pour la construction de nouveaux al-

gorithmes, permet également d'expliquer et de prouver les algrorithmes existants de manière plus intuitive. La reconstruction de ces algorithmes en utilisant notre méthodologie introduit une amélioration systématique au niveau de l'utilisation des canaux virtuels, et donc des performances.

Tous les mécanismes nécessaires à l'implémentation matérielle de notre méthode sont expliqués en détail, et on montre de plus que leur impact sur la surface totale du routeur est négligeable.

Contribution II: Une nouvelle solution de routage économique et robuste pour les NoCs tri-dimensionnels partiellement connectés

Une version préliminaire de ce travail a été publiée dans Charif et al. [2017c].

Le TSV (Through-Silicon-Via) est une technologie prometteuse permettant de faire communiquer des composants en passant par la troisième dimension. Pour ce qui est du NoC, cela permet d'envisager des topologies tri-dimensionnelles à faible latence pouvant inclure un très grand nombre de noeuds.

Cependant, le coût élevé des TSV ne permet pas d'assurer des connexions verticales pour tous les noeuds, et les topologies 3D partielles, où seuls certains noeuds sont verticalement connectés, ont été regardés de très près ces dernières années. En effet, ce type de topologies irrégulières qui serait à priori inévitable en pratique, pose un certain nombre de challenges en matière de routage sans interblocages.

On constate que parmi les solutions proposées jusqu'à présent, certaines requièrent l'ajout d'un canal virtuel dans toutes les directions du plan Bahmani et al. [2012], ce qui implique un surcout considérable, alors que d'autres nécessitent moins de canaux additionnels Salamat et al. [2016b], voire même pas de canaux additionnels du tout Lee et al. [2015], mais posent des restrictions importantes soit sur le placement des TSVs, c'est à dire que les TSVs doivent être placés à des endroits spécifiques du réseau afin d'en garantir la connectivité, soit sur la sélection des TSVs à utiliser durant le routage, ce qui peut forcer l'emprunt de chemins trop longs et avoir

un impact négatif sur la performance.

En réponse à ces différents compromis, on introduit une nouvelle méthode de routage pour ce type de topologies possédant les caractéristiques suivantes:

- La connectivité est garantie en présence d'un seul pilier de TSVs placé n'importe où dans le réseau. Il n'y donc pas de restrictions sur le placement des piliers.
- N'importe quel pilier du réseau peut être sélectionné par l'algorithme de routage. Aucune restriction sur l'emprunt des TSVs.
- Un canal virtuel est requis dans seulement deux directions du plan.

On utilise donc le même nombre de canaux virtuels que Salamat et al. [2016b] mais notre méthode ne souffre pas des restrictions sur le placement des piliers et leur sélection.

De manière générale, l'approche proposée peut être décrite comme suit:

1. Diviser le plan (Est, Ouest, Sud, Nord) en deux ensembles de directions, de sorte à ce que le second ensemble + les deux directions Haut et Bas (la dimension Z) ne permettent pas la formation de cycles.
2. Lors du routage, si un mouvement vertical est nécessaire, utiliser les directions du premier ensemble, puis celles du second ensemble pour atteindre n'importe quel TSV. Puisque toutes les directions sont incluses, il est toujours possible d'atteindre n'importe quel TSV.
3. Utiliser les direcitons du second ensemble + haut et bas pour atteindre l'étage (le niveau) où se trouve la destination.
4. A l'étage de destination, utiliser les directions du second ensemble puis ceux du premier ensemble pour atteindre la destination. Là encore, toutes les directions sont incluses, et par conséquent, la destination peut être atteinte quelle que soit sa position.
5. Les directions du premier ensemble étant utilisées au niveau source comme au niveau destination, un canal virtuel est requis uniquement dans ces directions pour prévenir les interblocages.

On montre que cette méthode, appelée "First-Last" du fait que les premières directions utilisées sont aussi les dernières, est réalisable en utilisant les directions positives (Est, Nord)

comme premier ensemble et les directions négatives (Ouest, Sud) comme second ensemble. Cela signifie que des canaux virtuels sont nécessaires uniquement dans les directions Est et Nord.

Notre algorithme est le premier à permettre une telle distribution de canaux virtuels. En effet, les propositions existantes ne considèrent que les configurations où le nombre de canaux virtuels est le même le long d'une même dimension. Par exemple, l'algorithme East-then-West Salamat et al. [2016b] rajoute un canal virtuel le long de la dimension Y (Nord et Sud), conformément à ce qui a historiquement été fait dans le cas du routage 2D Schwiebert and Jayasimha [1993]. On a démontré que cette règle n'était pas nécessaire et ne faisait que limiter les solutions possibles.

Les résultats de simulations et de synthèse matérielle montrent que nous arrivons non seulement à réduire considérablement le coût d'implémentation (d'environ 15%) par rapport à l'algorithme de référence Elevator-First Bahmani et al. [2012], mais aussi à offrir des performances bien supérieures grâce à une bonne exploitation des ressources disponibles.

Notre contribution au routage 3D ne se limite pas à l'algorithme First-Last. Afin de proposer une solution de routage complète et scalable, nous explorons également plusieurs méthodes de sélection de TSV. Chacune des méthodes proposées comporte un algorithme hors-ligne permettant d'assigner à chaque routeur un TSV et de configurer des registres de taille fixe pour pointer vers le TSV sélectionné, et un algorithme de routage capable d'exploiter ces données pour trouver un TSV lors de l'exécution. La recherche de TSV étant effectuée de manière distribuée, il est nécessaire de garantir que chaque paquet finit forcément par atteindre un TSV. Nous élaborons une preuve formelle qui stipule que si les TSVs sont assignés en fonction de leur distance par rapport aux noeuds, et quel que soit le critère utilisé pour départager des TSVs à équi-distance, chaque paquet finit toujours par atteindre un TSV, et ce en parcourant l'un des chemins de distance minimale à partir de la source. Cette preuve est très importante car elle implique que l'algorithme de routage (implémenté en hardware) n'a pas besoin d'être modifié pour éviter qu'un paquet ne cherche un TSV indéfiniment, comme d'autres solutions proposent de faire Niazmand et al. [2016].

Contribution III: Simulation parallèle et précise de NoCs sur GPU

Cette contribution a fait l'objet d'une publication à ASP'DAC 2017 Charif et al. [2017a].

Les simulateurs de NoCs précis au cycle près (cycle-accurate) sont très souvent utilisés pour la validation de nouvelles architectures. Cependant, comme les puces émergentes sont capables d'accueillir de plus en plus de routeurs, les nouvelles solutions doivent être validées sur des réseaux de plus en plus grands (centaines voire milliers de routeurs). Les simulateurs populaires comme Garnet Agarwal et al. [2009] ou Booksim Jiang et al. [2013] ne parviennent pas à effectuer ce genre de simulations en des temps raisonnables.

Une solution évidente à ce problème consiste à paralléliser les simulations sur des machines multicoeurs Eggenberger and Radetzki [2013], Eggenberger et al. [2016], Ren et al. [2012]. Cependant, à cause des temps de synchronisation non-négligeables dans le cas des CPUs, et étant donné que la précision des simulations requiert une synchronisation globale après chaque cycle simulé, des compromis doivent être faits et souvent, la précision des résultats doit être sacrifiée pour obtenir des accélérations décentes Ren et al. [2012].

Les GPUs (Graphics Processing Units) sont par ailleurs devenus l'un des supports de calcul parallèle les plus prisés. Bien qu'ils n'offrent pas une solution miracle au problème de synchronisation, ils contiennent un grand nombre de ressources de calcul et permettent la création d'un grand nombre de threads, pouvant compenser le temps passé dans les synchronisations. On se propose donc de concevoir et d'implémenter un simulateur cycle-accurate de NoCs capable d'exploiter au mieux les ressources du GPU sans pour autant demander à l'utilisateur des efforts d'optimisation.

Le design proposé se base sur une décomposition des tâches qui n'est pas définie en terme de routeurs Eggenberger and Radetzki [2013] ou de ports Zolghadr et al. [2011], mais de groupements personnalisés de modules. Un modèle de programmation où tous les modules définis peuvent s'exécuter en parallèle est proposé. Les modules communiquent à travers des registres, où chaque registre se compose de deux cases (odd, even). Selon la parité du cycle simulé, les modules lisent et écrivent dans des cases différentes du registre. Cela assure que les modules ne lisent pas les valeurs trop tôt et préserve la fidélité au matériel simulé.

On peut définir autant de modules que l'on veut, pouvant effectuer des tâches complètement distinctes et de complexités différentes (code divergent) et les organiser dans des groupes. La composition des groupes se fait par l'utilisateur sans aucune contrainte sur le nombre de modules ou leur type. Ce sont alors les groupes de modules qui sont considérés comme des tâches parallélisables. Cette indirection est très importante car on peut ajuster la granularité de la parallélisation comme on le souhaite. Le même design (mêmes modules, même code) peut être simulé sur un GPU à faibles ressources en changeant simplement le groupage pour contenir moins de groupes (et donc moins de threads). Sur un bon GPU, cela permet de simuler des NoCs de tailles très grandes (plus de 4000 routeurs ont été simulés sur un GTX980Ti).

Au niveau de l'implémentation, on propose plusieurs optimisations sur l'utilisation de la mémoire ainsi que la synchronisation globale. Le point le plus important est le mapping des tâches, qui doit être réalisé de sorte à ce que la divergence de code soit minimale. Bien que les groupes de modules contiennent des codes complètement hétérogènes, on observe que tous les routeurs comportent les mêmes groupes. La stratégie de mapping doit donc exploiter ce parallélisme inhérent à l'architecture du NoC afin de minimiser, voire supprimer la divergence de code. On arrive à accomplir cela en identifiant chaque thread par deux coordonnées g, r (Groupe et Routeur). Sachant que tous les W threads (typiquement 32) doivent exécuter le même code (modèle SIMT Single Instruction Multiple Theeads), on propose de définir l'identifiant de thread comme suit $ID = g * R + r$, où R est le nombre total de routeurs. Ainsi, au sein de W threads (appelés Warp), on change de routeur, avant de changer le groupe de modules, ce qui permet de garantir, en s'assurant que R est multiple de W, que les threads d'un warp exécutent le même code (même groupe).

En s'appuyant sur ce design, on a développé un outil de simulation ultra-rapide appelé GNoCS, qui a servi entre autres à la réalisation de toutes les simulations présentées le long de cette thèse. Des détails sur son implémentation (API, modèle de programmation, etc.) sont présentés dans l'Annexe B.

Contents

Contents xiv

List of Figures xxi

List of Tables xxiv

I INTRODUCTION 1

1 Introduction 3

1.1 Introduction . . . 3

1.1.1 Contribution I: A New Methodology for the Design of Highly Flexible Deadlock-Free Routing Algorithms . . . 4

1.1.2 Contribution II: A Cost-Effective Routing Solution for TSV-Based 3D Networks-on-Chips . . . 5

1.1.2.1 Scalable TSV assignment strategies . . . 6

1.1.2.2 The First-Last routing algorithm . . . 6

1.1.3 Contribution III: Ultra-fast GPU-Based Parallel Simulation of NoCs . . . 6

1.2 Experimental setup . . . 7

1.2.1 Hardware implementation . . . 7

1.2.2 Simulation . . . 7

II ON THE DESIGN OF DEADLOCK-FREE ROUTING ALGORITHMS

FOR WORMHOLE NETWORKS 9

2 A Dynamic Sufficient Condition of Deadlock-Freedom for Highly Flexible Routing in Wormhole Networks 11

2.1 Introduction 11
2.2 State-of-the-art 13
2.2.1 VC-based deadlock avoidance 13
2.2.2 Necessary and sufficient conditions of deadlock-freedom 13
2.2.3 Routing algorithm design frameworks 14
2.3 Routing algorithm design methodology 15
2.3.1 Switch model 15
2.3.2 Fault model 17
2.3.3 Example design: Fault-tolerant routing in 2D mesh NoCs 17
2.3.4 Upgrades and Downgrades 21
2.4 Implementation details 22
2.4.1 Without downgrades 23
2.4.2 With downgrades 25
2.4.3 Repeating route computation 26
2.4.4 Negative tag back-propagation 26
2.5 Formal generalization 27
2.5.1 Preliminary definitions 27
2.5.2 Generic routing algorithm 28
2.5.3 Proof of deadlock-freedom 28
2.5.4 Livelock-freedom and termination 30
2.6 Flow control 31
2.7 A fresh look at fully adaptive routing algorithms 31
2.8 Experimental results 33
2.8.1 Simulation setup 33
2.8.2 Simulation methodology 33
2.8.3 Evaluating the fault-tolerant algorithm 34

2.8.4 The impact of flow control and dynamic masking 37
2.8.5 Hardware synthesis 37
2.9 Conclusions 39

III ROUTING IN TSV-BASED THREE-DIMENSIONAL NETWORKS-ON-CHIPS 41

3 A framework for scalable distributed TSV assignment 43
3.1 Introduction 43
3.2 State-of-the-art 44
3.3 Target architecture 46
3.3.1 NoC architecture 46
3.3.2 Routing 47
3.4 Manhattan-distance-based elevator selection 50
3.4.1 Elevator location bits 50
3.4.2 Safe Selection Algorithm (md-safe) 50
3.4.3 Randomized Selection Algorithm (md-random) 51
3.5 Proof of reachability for Manhattan-Distance-Based approaches 55
3.6 Optimistic elevator selection 60
3.6.1 Elevator location bits 61
3.6.2 Routing algorithm (optimistic) 61
3.7 Experimental results 62
3.7.1 Hardware synthesis results 62
3.7.2 Performance evaluation 65
3.8 Conclusions 69

4 The First-Last routing algorithm: A cost-effective alternative to Elevator-first 70
4.1 Introduction 70
4.2 State-of-the-art in 3D Routing 71
4.3 The First-Last routing algorithm 73
4.3.1 General approach 73

4.3.1.1 Identifying cycle-free routing classes 73
4.3.1.2 Step 2: Assigning virtual channels 74
4.3.2 Enhanced-First-Last: Boosting network performance and resilience with vertical VCs 75
4.3.3 Flow control 76
4.3.3.1 First-Last 77
4.3.3.2 Enhanced-First-Last 77
4.4 Deadlock-freedom, livelock-freedom, and connectivity 78
4.4.1 Deadlock-freedom 78
4.4.2 Livelock-freedom 78
4.4.3 Condition of connectivity 78
4.5 Hardware implementation details 80
4.5.1 Scalable TSV assignment 81
4.5.2 Route computation logic 82
4.6 Experimental results 85
4.6.1 Hardware synthesis 85
4.6.2 Performance evaluation 86
4.7 Conclusion 87

IV ULTRA-FAST GPU-BASED PARALLEL CYCLE-ACCURATE SIMULATION OF NETWORKS-ON-CHIPS 91

5 Highly Parallelizable Cycle-Accurate Network-on-Chip Simulation 93
5.1 Introduction 93
5.2 State-of-the-art 94
5.3 Generic Task Decomposition 95
5.3.1 Modules 96
5.3.2 Module groups 99
5.3.3 Tasks 99
5.4 GPU-based implementation 100

5.4.1 Overview on the GPU architecture 101
5.4.2 Warp-friendly task mapping 101
5.4.3 Compact flit queue implementation 102
5.4.4 Simulating large networks 104
5.4.5 Simulation kernel and final notes 105
5.5 Experimental results 106
5.5.1 Speedup 106
5.5.2 Hardware fidelity 108
5.6 Conclusions 108

V CONCLUSIONS 112

6 Conclusion and Future works 114
6.1 Conclusions 114
6.2 Future directions 116

VI APPENDICES 119

A Congestion metrics for adaptive routing 122
A.1 Introduction 122
A.2 Generic congestion management unit description 123
A.2.1 Router events 123
A.2.2 Updating the congestion value 124
A.2.3 Congestion metric description 124
A.2.3.1 The free_buf metric 124
A.2.3.2 The free_vc metric 125
A.2.3.3 The x_bar metric 125
A.2.3.4 The fr metric 126
A.3 Introducing new congestion metrics 126
A.3.1 The FRN congestion metric 126

A.3.2 The NAFAE congestion metric 127
A.4 Experimental results and conclusions 127

B GNoCS: A highly-extensible GPU-based parallel network-on-chip simulator 130
B.1 Introduction 130
B.2 Basic data types and API 130
B.2.1 Integer API 130
B.2.2 Registers 131
B.2.3 Flit queue 131
B.3 Programming model 133
B.3.1 Example module: Credit manager 134
B.4 Simulation automation 136
B.4.1 Structure of a simulation description file 136
B.4.2 The *_except* directive 138
B.4.3 SSH Support: The *_nodes* directive 138

Bibliography 141

List of Figures

2.1 Router architecture. 16
2.2 Physical channels used by the two classes of the example algorithm. 18
2.3 An example packet configuration. 20
2.4 Static vs. Dynamic masking. (a) Static: class 0 packets always mask the same channel during VC allocation. (b) Dynamic: The mask changes every time a class 1 packet acquires a VC. 24
2.5 Physical channels used by the two classes of FT-CAR. 32
2.6 Throughput results. 35
2.7 Delivery success rate. 36
2.8 Critical packet delivery success rate. 36
2.9 Impact of flow control and dynamic masking on average latency. 38

3.1 Overview of the partially connected 3D-NoC Topology. 48
3.2 Illustration of a potential deadlock scenario. 53
3.3 Example of an inefficient route when using MD-based algorithms. 60
3.4 Average packet latency for an 8x8x2 NoC. 66
3.5 Average packet latency for an 8x8x4 NoC. 67

4.1 The 3 routing classes of First-Last. 75
4.2 The routing classes of Enhanced-First-Last. 76
4.3 An example in which Enhanced-First-Last improves the connectivity of the network. 77
4.4 Examples of connected networks using First-Last. 80

4.5 Average packet latency. 88

5.1 Module groups. 100
5.2 Mapping of tasks onto threads. 103
5.3 Compact flit queue representation. 104
5.4 Average Network Latency (RTL vs. GPU). 109

A.1 Average packet latency obtained with different congestion metrics. 128

List of Tables

2.1 Hardware synthesis results 39

3.1 Hardware synthesis results 65

3.2 Simulation parameters 68

4.1 Hardware synthesis results 86

4.2 Simulation parameters 86

5.1 Simulation parameters 107

5.2 Speedup results (1) 107

5.3 Speedup results (2). (*) Injectors evicted from shared memory. 108

Part I

INTRODUCTION

Chapter 1

Introduction

1.1 Introduction

The ever-growing need for processing power in modern digital systems has led to a significant increase in the number of Intellectual Properties (IPs) integrated in a single chip. This was partly enabled by the aggressive scaling of transistor feature sizes, which, along with the many benefits it brings in terms of area, delay and power consumption, is known to pose some serious concerns about reliability. In this context, Networks-on-Chips (NoCs) Dally and Towles [2001] have emerged as the new paradigm of choice for on-chip communication, and are today widely used in many-core systems, as well as Graphics Processing Units (GPUs) Bakhoda et al. [2010], Wentzlaff et al. [2007]. In addition to being a power-efficient and scalable replacement for traditional buses, they contribute greatly to the chip's fault-tolerance and performance thanks to the path diversity that is inherent to the widely adopted NoC topologies. They are today an actively researched topic and various solutions are being explored to meet the needs of emerging applications in terms of performance, quality of service, energy, and fault-tolerance. This thesis presents contributions in two important areas of Network-on-Chip research:

- The design of flexible deadlock-free fault-tolerant routing algorithms for different topologies.
- The design and implementation of parallel cycle-accurate Network-on-Chip simulators for a fast evaluation of new NoC architectures.

These contributions are summarized in the remainder of this section.

1.1.1 Contribution I: A New Methodology for the Design of Highly Flexible Deadlock-Free Routing Algorithms

While aggressive technology scaling has its benefits in terms of delay, area and power, it is also known to increase the vulnerability of circuits, suggesting the need for fault-tolerant designs. Fault-tolerance in NoCs is directly tied to the degree of flexibility of the routing algorithm. High routing flexibility is also required in some irregular topologies, as is the case for TSV-based 3D Network-on-Chips, wherein only a subset of the routers are connected using vertical connections. One challenging aspect in the design such flexible algorithms is the risk of deadlock formation, which can occur if cyclic dependencies are formed between packets.

Many existing routing solutions make use of virtual channels (VCs), which consist in several disjoint input queues, to avoid deadlocks while offering enough routing flexibility to avoid faulty and congested areas in a NoC. However, most of the current solutions rely on an overly restrictive, static partitioning of VCs, which results in an under-utilization of their throughput enhancement capabilities Ebrahimi et al. [2013a], Chaix et al. [2011]. In effect, VCs can also be used to reduce Head-of-Line (HOL) blocking and greatly reduce the transmission delays Dally [1992].

To overcome the limitations of such approaches, Chapter 2 introduces a new sufficient condition of deadlock-freedom that greatly relaxes the restrictions imposed by the classic VC-based deadlock-avoidance methods. The strength of our condition lies in the fact that it is imposed on packets at runtime and does not require any partitioning of virtual channels, which makes it possible to fully exploit them to reduce packet blocking and boost performance. Based on this condition, we present a generic, topology-agnostic routing algorithm design methodology that can be used to construct highly flexible routing algorithms in only a few steps. Several examples are presented to showcase the usefulness of our approach for the construction of fault-tolerant routing algorithms, as well as the enhancement and the proof of existing routing algorithms. The implementation of all the required mechanisms in hardware is also described in detail, thereby demonstrating its feasibility in an on-chip environment.

A significant gain in both performance and fault-tolerance when using our methodology compared to the existing static channel partitioning methods is reported. Moreover, hardware synthesis results show that the newly introduced mechanisms have a negligible impact on the overall router area. This methodology is subsequently used in the rest of the thesis to construct new routing algorithms for 3D NoCs.

We have published some parts of this work in Charif et al. [2016] and Charif et al. [2017b].

1.1.2 Contribution II: A Cost-Effective Routing Solution for TSV-Based 3D Networks-on-Chips

The emergence of 3D integration can greatly benefit future many-cores by enabling low-latency three-dimensional network-on-chip (3D-NoC) topologies Feero and Pande [2009], Pavlidis and Friedman [2007]. However, due to the high cost, low yield, and frequent failures of Through-Silicon Via (TSV) Benini [2008], 3D-NoCs are most likely to include only a few vertical connections Bartzas et al. [2007], resulting in partially connected topologies that pose new challenges in terms of deadlock-free routing and TSV assignment.

With a limited number of vertical connections, the routers of such networks require a way to locate the nodes that have vertical connections, commonly known as elevators, and select one of them in order to be able to reach other layers when necessary. Both the strategy used to select which elevator to take, and the routing algorithm used to reach the destination, have a critical impact on the cost and performance. However, most existing solutions either require too many VCs, which significantly impacts the cost Dubois et al. [2013], or poses too many restrictions on the placement and selection of TSVs Salamat et al. [2016b], Lee et al. [2015]. The goal of our second contribution is to provide a full routing solution for partially connected 3D-NoCs that not only reduces the number of required virtual channels, but offers increased flexibility compared to the state-of-the-art algorithms. This contribution consists of two parts presented in two different chapters:

1.1.2.1 Scalable TSV assignment strategies

In Chapter 3, we explore, for the first time, various strategies for assigning TSVs to routers both offline and during runtime. All the solutions that we propose use a fixed number of reconfigurable bits per router (scalability), and every assignment algorithm is formally proven to guarantee reachability and deadlock-freedom.

Many of the proposed strategies are generic and do not depend on a specific routing algorithm, resulting in a reusable framework for scalable TSV assignment in 3D-NoCs.

1.1.2.2 The First-Last routing algorithm

Chapter 4 presents a novel routing algorithm targeting 3D-NoCs. Thanks to a unique distribution of virtual channels, our algorithm is the only one capable of guaranteeing full connectivity in the presence of one TSV pillar in an arbitrary position, while requiring a low number of extra buffers (only 1 extra VC in the East and North directions). Moreover, because it is based on the deadlock-avoidance approach presented in Chapter 2, it attains a high level of performance with respects to the state-of-the-art Elevator-First algorithm, in spite of using less virtual channels. We further exploit the efficient TSV assignment framework from Chapter 3 to implement a full, highly cost-effective routing solution for partially-connected 3D-NoCs.

A preliminary version of the First-Last routing algorithm was published in Charif et al. [2017c].

1.1.3 Contribution III: Ultra-fast GPU-Based Parallel Simulation of NoCs

In order to speed up the evaluation of new Network-on-Chip designs, the last chapter of this thesis (Chapter 5) presents the first detailed and modular parallel NoC simulator design to run fully on Graphics Processing Units (GPUs). First, a flexible task decomposition approach, specifically geared towards high parallelization is proposed. Our approach makes it easy to adapt the granularity of parallelism to match the capabilities of the host GPU. Second, all the GPU-specific implementation issues are addressed and several optimizations are proposed. Our design is evaluated through a reference implementation, which is tested on an NVidia GTX980Ti graphics card and shown to speed up 4K-node NoC simulations by almost 280x.

Part of this work was previously published in Charif et al. [2017a].

1.2 Experimental setup

1.2.1 Hardware implementation

Although our contributions introduce novel approaches that are interesting from the theoretical standpoint, it is important to make sure that the presented solutions are feasible in terms of hardware.

For this reason, all the new algorithms and mechanisms introduced throughout this manuscript are implemented and synthesized. We present both details about the implementation and synthesis results for every proposal. SystemVerilog was used for all hardware implementations. Full router implementations were based on the Netmaker on-chip router library Mullins [2009], whereas other implementations were written from scratch.

The syntheses were performed using Synopsys Design Compiler and the nangate 45nm library Nangate [2017].

1.2.2 Simulation

For performance and reliability assessment, we use an in-house cycle-accurate simulator based on the design presented in Chapter 5. In addition to being very fast, it models the simulated router very accurately and in compliance with the hardware implementation. Moreover, the outputs of every single module are thoroughly checked throughout the simulations, giving us great confidence in the correctness of the presented results. It is worth mentioning that the simulation process is fully automated (see Appendix B) and that the figures were generated by the simulator and presented with no alterations.

Part II

ON THE DESIGN OF DEADLOCK-FREE ROUTING ALGORITHMS FOR WORMHOLE NETWORKS

Chapter 2

A Dynamic Sufficient Condition of Deadlock-Freedom for Highly Flexible Routing in Wormhole Networks

2.1 Introduction

A convenient and widely adopted method for avoiding routing deadlocks is to make use of virtual channels Dally [1990], which consist in independent input queues added at each input port, allowing many packets to be multiplexed on the same physical channel. VCs are partitioned into several virtual networks (VNs), such that no cycles can form with-in each VN. Typically, packets of one VN cannot acquire virtual channels from another virtual network, the idea being the suppression of cyclic dependencies between virtual networks. Although this approach has proven to be a simple and effective deadlock-avoidance solution for fault-tolerant systems as well as emerging topologies Chaix et al. [2010], Dubois et al. [2013], its main drawback is that it underutilizes the available virtual channels. In fact, virtual channels are an expensive resource, and if we were able to use them for their performance boosting potential instead of reserving them for the sole purpose of deadlock-avoidance, it would make the extra cost worthwhile.

From a theoretical point of view, while the strict partitioning of virtual channels is sufficient to avoid deadlocks, it is not necessary. In effect, necessary and sufficient conditions

of deadlock-freedom Duato [1995], Schwiebert and Jayasimha [1995], Fleury and Fraigniaud [1998] suggest that the absence of cyclic dependencies between channels is not necessary for deadlock avoidance. Moreover, by adding virtual channels the chances of deadlock formation are significantly decreased as explained in Pinkston and Warnakulasuriya [1999]. The currently adopted approaches are therefore overly restrictive, and the resulting resource waste can definitely be avoided.

In this first chapter, we propose a general solution to this issue by introducing a very low-restrictive sufficient condition of deadlock-freedom that has the major advantage of being formulated using the locally available runtime information in each router. This means that the condition can be verified by the hardware online, and no static reservation of virtual channels for deadlock-avoidance is necessary, allowing for ultra-flexible high-performance routing algorithms to be designed.

Based on this condition, an intuitive, topology-agnostic routing algorithm design framework comprising the minimum set of rules that routing algorithms should fulfill to satisfy our deadlock-freedom condition is presented.

We provide details on the implementation of our approach in hardware, and highlight its wide range of applicability through several case studies. In addition to fault-tolerant routing, which we showcase in the first part of this chapter, we also demonstrate that the presented condition of deadlock-freedom can be used to enhance the performance of existing routing algorithms by relaxing the restrictions on their virtual channel usage. Finally, we show how our condition can provide a more intuitive understanding of already existing routing algorithms, as we easily reconstruct a maximally adaptive routing algorithm for the 2D topology, which was previously proven deadlock-free using channel dependency graphs and classic necessary and sufficient conditions of deadlock-freedom.

2.2 State-of-the-art

2.2.1 VC-based deadlock avoidance

The vast majority of routing algorithms that require virtual channels for deadlock avoidance have been adopting a conservative approach, which consists in a strict partitioning of virtual channels at design time such that each virtual channel has its own routing rules that packets must follow. Usually, blocked packets can only request one of the two virtual channels at a time. When fault-tolerance is required, moving from one virtual channel to the other is permitted only in one direction (e.g. in increasing order).

Example of recent fault-tolerant routing algorithms using this same approach include Ebrahimi et al. [2012], Chaix et al. [2010]. This method has also been employed in the context of 3D-NoCs Dubois et al. [2013], Salamat et al. [2016b], where there is always a strict separation between Upward and Downward packets Dubois et al. [2013], Eastward and Westward packets Salamat et al. [2016b], etc.

The issue with this approach is that the virtual channels are underutilized because packets have to wait for one specific VC at all times, even when other VCs are idle. In context of fault-tolerance, in order to provide all packets with enough flexibility to route around faults, some algorithms require that all packets start routing in the first VC, so that they get a chance to move to a higher VC when necessary Ebrahimi et al. [2012]. This means that the second VC remains entirely idle in the absence of faults, which is a waste of resources.

However, because the classic method simply consists in splitting the network into two disjoint deadlock-free networks that use familiar routing algorithms Glass and Ni [1992], it makes it easy to build new routing solutions by combining several previously known deadlock-free turn models. In this thesis, we aim at improving VC utilization without jeopardizing this important property.

2.2.2 Necessary and sufficient conditions of deadlock-freedom

It has long been known that the suppression of cyclic dependencies is not necessary for deadlock-freeness Pinkston and Warnakulasuriya [1999]. Several necessary and sufficient conditions

of deadlock-freedom have been proposed in the context of wormhole routing Verbeek and Schmaltz [2011], Duato [1995], Duato [1997], Fleury and Fraigniaud [1998]. These are all excellent for proving the deadlock-freedom of an existing algorithm and even for building formal verification tools Taktak et al. [2008].

However, they can hardly be used to construct new algorithms. For instance, the authors in Kumar et al. [2014] have recently proposed a routing algorithm for NoCs that aims at reducing the restrictions on the VCs that can be used. The algorithm uses Duato's necessary and sufficient condition Duato [1995] to prove the network deadlock-free despite the presence of cyclic dependencies. The resulting algorithm is highly flexible, but it can be difficult to have an intuitive understanding of how it works without thinking in terms of dependency graphs. There is therefore a clear need for a design methodology that makes it possible to systematically build such algorithms.

Using our design methodology, we will reconstruct the routing algorithm in Kumar et al. [2014] from scratch, and show how our approach can help explain it in a more intuitive way.

2.2.3 Routing algorithm design frameworks

Our methodology is based on a sufficient condition of deadlock-freedom that is expressed using local information, which is made available through a virtual channel tagging mechanism during runtime. This makes our condition verifiable by the hardware dynamically, allowing for a high degree of flexibility.

The idea of using runtime information for deadlock avoidance is not new. Early works on adaptive routing, such as Dally and Aoki's algorithm Dally and Aoki [1993], also make use of a similar labeling mechanism. More notably, in Boppana and Chalasani [1996], the authors propose a framework for designing adaptive routing algorithms. The terminology as well as many of the concepts they propose are quite similar to our proposal. However, there are fundamental differences between our methodology and these works. First, both Dally and Aoki [1993] and Boppana and Chalasani [1996] avoid deadlocks by suppressing cyclic dependencies from the packet wait-for graphs. Our condition is less restrictive and allows the presence of cyclic dependencies both in the channel dependency graph and the packet wait-for graph. In

this regard, our deadlock-freedom condition is closer to the necessary and sufficient condition as formulated in Schwiebert and Jayasimha [1995]. Like Boppana and Chalasani [1996], our methodology supports class upgrades to further increase routing flexibility and enhance virtual channel utilization. However, our framework also supports class downgrades, pushing the level routing freedom even further. More importantly, our formulation is specifically tailored for on-chip networks, and can use the already available deadlock-free routing algorithms as building blocks. We also provide several details on the implementation of our deadlock-avoidance mechanisms in hardware.

More recently, in Ebrahimi and Daneshtalab [2017], the authors have introduced a new theory for designing deadlock-free routing algorithms. It is a formal generalization of the method used to construct several routing algorithms such as Ebrahimi et al. [2013a]. Because it is based on the partitioning of network channels as in Dally and Seitz [1988], it requires the absence of cyclic dependencies between channels for deadlock-freedom, limiting the utilization of VCs compared to the approach that we propose.

2.3 Routing algorithm design methodology

In this section, we introduce the proposed routing methodology by example. After defining the network architecture, we design a new fault-tolerant routing algorithm for 2D Mesh NoCs and highlight the differences with the traditional static approaches.

2.3.1 Switch model

We consider a typical input-buffered wormhole router such as the one shown in fig. 2.1. Several Virtual Channels (VCs) may be available per input port. Routing a packet is performed in 5 steps. When the first (head) flit of a packet is read from an input port, it is buffered in the input VC that was allocated to it by the upstream router and, in the same cycle, the Route Computation Unit selects the next output port to which the packet should be forwarded. The input VC, which was in “Idle” state, moves to “Waiting” state and waits until the VC Allocator grants it an Idle VC in the downstream router. When a downstream VC is acquired, the input VC enters the “Active” state. Each flit of the packet then waits for the Switch Allocator to grant it permission

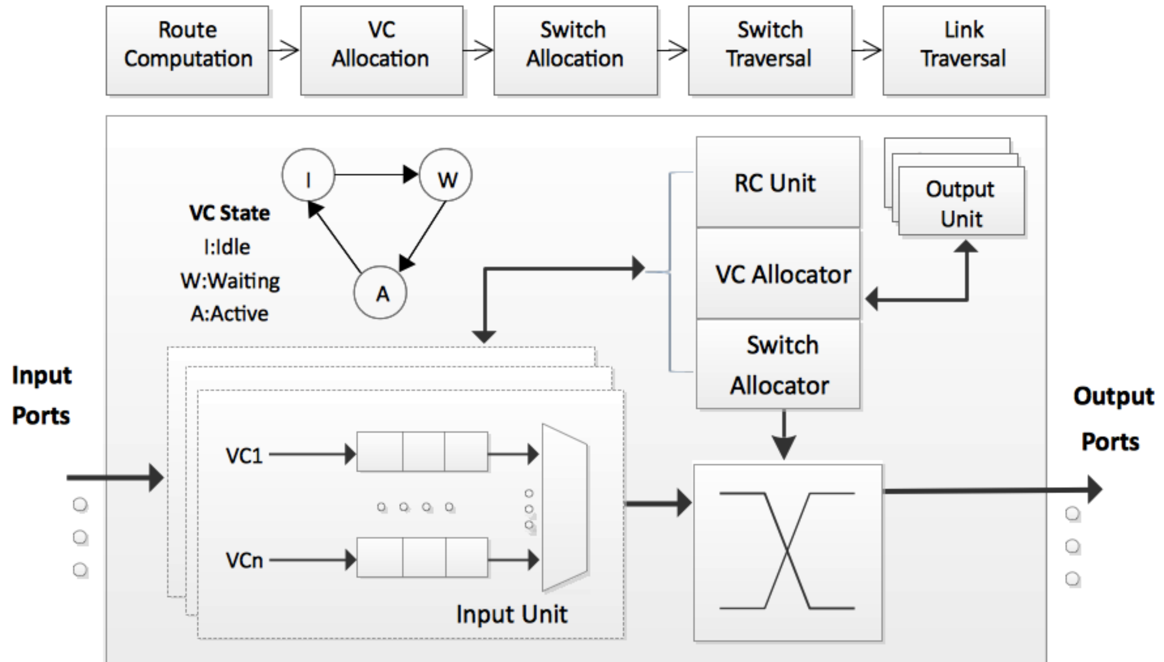

Figure 2.1: Router architecture.

to traverse the Crossbar and reach the output port, provided that there are enough free buffers to store the flit in the downstream VC. When a flit acquires the switch, a credit signal is sent to the upstream router, indicating that there is a newly available buffer in this VC. Finally, the flit traverses the link to reach the next router. When the last (tail) flit of a packet leaves the router, the input VC goes back to the “Idle” state and becomes ready to receive new packets.

The output unit (fig. 2.1) maintains a structure representing the status of each downstream VC. The structure representing one downstream VC has the following fields:

- State: Idle or Active.
- Credits: Number of free buffers.
- Tag: A signed integer used by the algorithm. Its usage will be explained later.

It is important to note that the only strict assumption that we make on the router’s architecture is that the output port is selected before VC allocation is performed. All the other aspects, such as the allocator types, pipelining, speculative switch allocation, have no affect on the validity of our routing approach.

2.3.2 Fault model

A faulty network is modeled by a set of permanently faulty unidirectional links. This model is often used to evaluate fault-tolerant routing algorithms as it demonstrates their ability to route packets in a maze-like irregular topology.

In practice, this can correspond to either a link actually failing, or input and output ports being disabled due to faults in their control logic, e.g. stuck-at faults. We consider that in the presence of faults in the central logic (allocators, crossbar, etc.) the router simply disables all of its input and output ports, so the same model can be employed to model node failure as well.

2.3.3 Example design: Fault-tolerant routing in 2D mesh NoCs

We consider, as an example, that the routers are interconnected in a 2D Mesh topology. Therefore, each router has four input/output ports connecting it to at most four neighboring routers (North, South, East, West), and one input/output ports (Local) connecting it to a network interface (NI).

The first step in designing a routing algorithm using our methodology is to identify a set of **routing classes**, such that each routing class is associated with one cycle-free routing function. The routing function supplies a set of physical channels (output ports) that the packet can occupy. It can achieve cycle-freedom in two different ways: Either the set of physical channels supplied by the routing function cannot form a cycle (e.g. take only the physical channels of one dimension) as in Ebrahimi and Daneshtalab [2017], or the physical channels can form a cycle but they are supplied in a specific order (e.g. the turn models Glass and Ni [1992]).

In Section 2.7, we will present an algorithm that uses the former approach. In this example, however, because we want to maximize the routing options within each class, we use the turn model Glass and Ni [1992] as a routing function.

Here, we chose to use the non-minimal South-last and North-last turn models to be assigned to routing classes 0 and 1 respectively. Fig. 2.2 shows the physical channels used within each class, along wih the numbering that indicates the order in which the physical channels must be traversed to suppress cycles, as per Glass and Ni [1992]. We can see that all four physical channels are used by both classes.

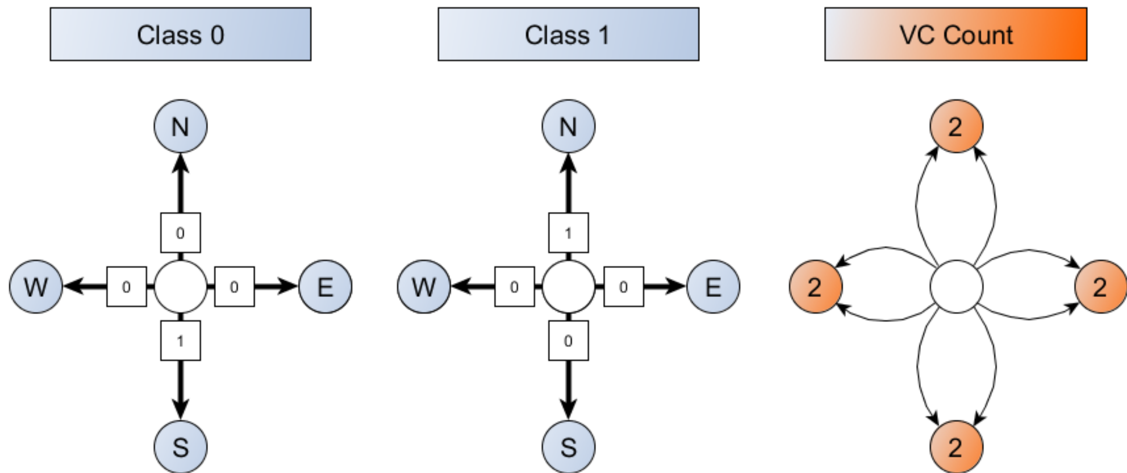

Figure 2.2: Physical channels used by the two classes of the example algorithm.

From the set of physical channels, we can deduce the minimum number of virtual channels that are required in each direction. The VC count in one physical channel, as shown in Fig. 2.2, simply corresponds to the number of classes that use the same physical channel. Our algorithm will require two VCs in all directions.

Thus far the setup is analogous to that uses by many related works, such as Chaix et al. [2010]. We intentionally chose similar routing functions to make it easier to highlight the differences. The first key difference is that unlike the classic approach, the routing functions are not assigned to specific virtual channels. Instead, each packet carries its class number, and this class number is used by the route computation unit to select the appropriate routing function, regardless of which virtual channel the packet is currently occupying. This also means that the virtual channels that packets can occupy are never known a priori, which is the main factor in achieving a higher throughput than the classic approaches. The VC acquisition rules are determined dynamically following the rules described in the rest of this section.

When a packet acquires a downstream VC, it must set the corresponding *Tag* field in the output unit to its class number. The following rule, which corresponds to our sufficient condition of deadlock-freedom, is enough to guarantee deadlock-free operation.

Rule 2.1. *“A packet may not request an output port unless at least one of the virtual channels associated with that port is either free, or tagged with a number higher than or equal to its class*

number".

For our example case, this means that packets of class 1 cannot wait for an output port in which all the channel tags are less than 1. Consider the class 1 packet coming from the west port of router 3 in Fig. 2.3. It is able to request the east port because there is one VC tagged with "1" in that port. Similarly, the packet coming from the south port of router 0 can legally request the east port because it is occupied by packets of class 1, which is higher than class 0. The intuition behind this rule is simple. By ensuring that packets of class 1 always have a dependency in class 1, and knowing that all the packets in class 1 are routed using an acyclic turn model, we guarantee that packets of class 1 dynamically form an acyclic dependency chain. Packets of class 0 either have all their dependencies in class 0, in which case they cannot form cycles because the South-last turn model is cycle-free, or they may have class 1 dependencies, leading forcibly to the dynamic escape path formed by class 1 packets.

Therefore, the first rule guarantees at least one escape for every packet in the network.

It is very important to understand that while only one of the virtual channels of the requested port needs to have a higher or equal tag, nothing is done to force the packet to wait for that one VC specifically. Going back to our example in Fig. 2.3, the class 1 packet coming from the west port of router 3 may very well acquire the top-most VC, which is held by a class 0 packet about to leave router 4, even though the port was selected because of the second VC from the top, which is tagged with "1". This means that the packet was not waiting for only one VC, but for both VCs, including the one tagged with a lower class number. This is a crucial point in achieving high performance.

While rule 2.1 ensures deadlock-freedom, there are a few issues that need to be addressed. In fig. 2.3, there is a packet coming from the west port of router 4 and requesting the east port, which has one free VC. If this free VC is granted to this packet, the port will be full of class 0 tags. Consequently, if a class 1 packet arrived at router 4 and wanted to go east, it would not be able to request the east port, because otherwise it would violate rule 2.1. Rule 2.2 imposes one restriction on the VC requests to alleviate this issue.

Rule 2.2. *"In every output port, there must be at least one free virtual channel for every class higher than the greatest tag in that port".*

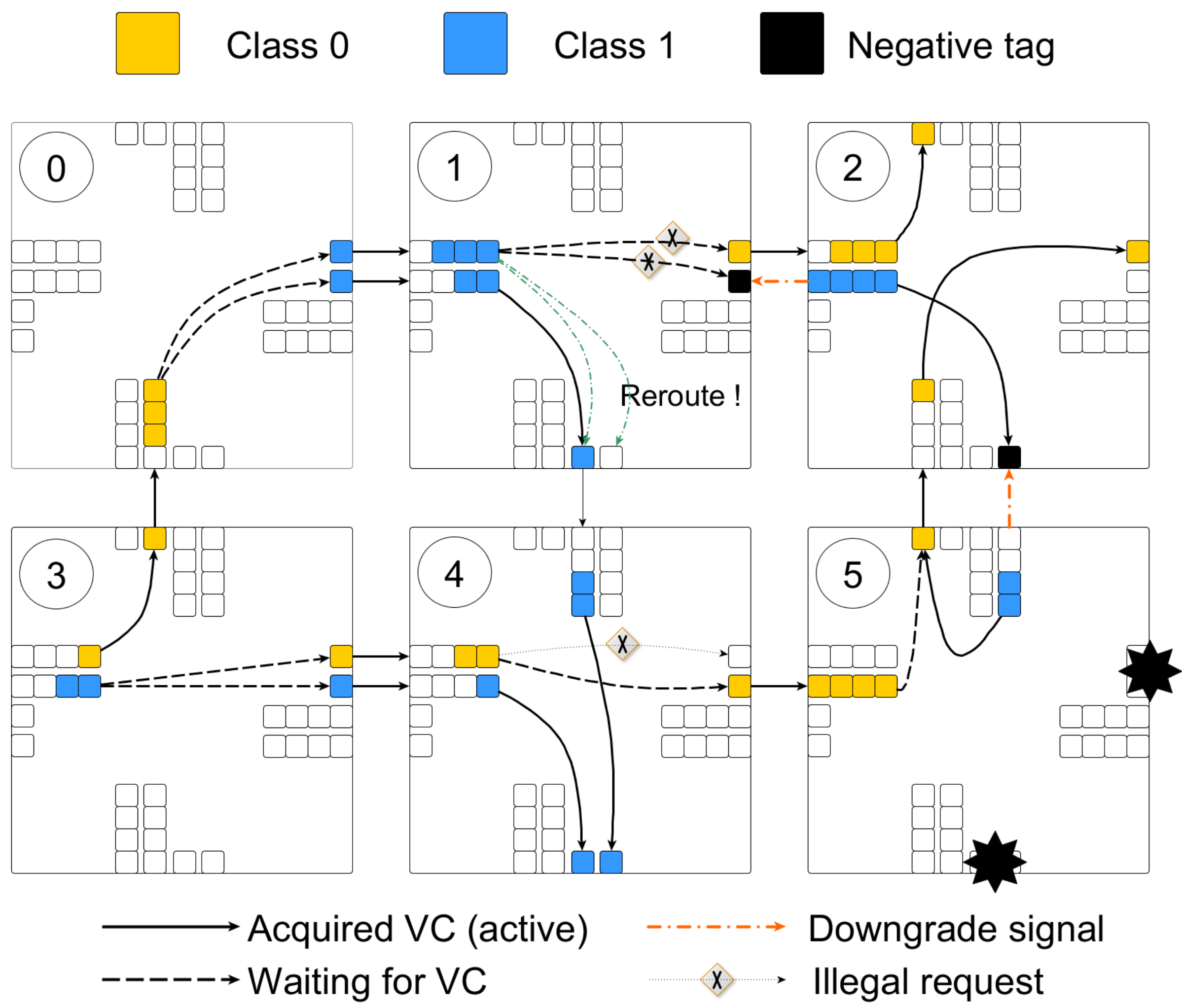

Figure 2.3: An example packet configuration.

In other words, the VC allocator should not grant all the virtual channels of a port to class 0 packets. Therefore, the 2-flit class 0 packet in the west port of router 4 (fig. 2.3) cannot request the free VC at the east port and has to wait for the class 0 packet to leave the router. However, note that router 0 has allocated all the virtual channels of its east port to class 1 packets. This does not violate the rule since there are 0 classes higher than 1, and therefore 0 VCs need to be kept free. We will show how this rule can be implemented in the next section.

2.3.4 Upgrades and Downgrades

To provide packets with even more routing freedom, our framework also supports class changing. Packets may prefer to change their class for various reasons. The most obvious use case is when their current class can no longer provide the necessary output channels to make progress towards the desitnation, either because of faults, or because they are not supplied by the routing function.

This feature is therefore a great contributor to fault-tolerance, as packet can either upgrade or downgrade to possibly get access to more physical channels and route around faults. A second case where packets may decide to move to another class is when all the physical channels they need to reach their destination are provided by a higher class. In this case, they may as well upgrade, as they will gain access to more virtual channels, as per rule 2.2.

We observe that if a packet of class 0 upgrades to class 1, it does not violate any of the previous rules. In effect, packets allowed to wait for class 0 are also allowed to wait for class 1, and therefore, the escape guarantee is maintained.

On the other hand, if a class 1 packet downgrades to class 0, then packets depending on it would lose their escape guarantee. This gives rise to the third and last rule.

Rule 2.3. *"When a packet downgrades, it must tag all of its containing virtual channels with a negative number".*

For instance, consider the packet arriving from the north port of router 5 in fig. 2.3. It is a long packet with flits in the west port of router 2, north port of router 5 and south port of router 2. Note that due to faults in router 5, the packet, initially of class 1, was not able to go east or south, therefore, it had to make a U-turn to go north then east, which is not allowed by the

North-last algorithm. In router 5, the class of the packet had to be changed to 0. Since it is a downgrade, according to rule 3, all the virtual channels the packet is occupying must be tagged with a negative number.

To make this possible, the channel holding the head of the downgrading packet sends a signal upstream to its corresponding output unit (as in the case of credits), which will tag the corresponding VC with a negative number at the reception of the signal, to indicate that the packet is violating the routing rules. In the cycle following the tag, if there is an active input VC in the upstream router which is associated with the newly negatively tagged VC, then it must in turn signal it to its upstream router, and so on until the VC holding the tail of the packet is tagged.

A negative number is smaller than any existing packet class, and therefore, packets will only wait for a negatively tagged VC if there is another VC in the same port with a higher or equal tag than their class. In our example, there is a class 1 packet waiting for the east port at router 1. At the time the port was selected for this packet, there was one VC tagged with “1”, so the port was valid. However, after the downgrade, the port is no longer valid, and therefore the packet waiting for it needs to be rerouted to a different port. The same port is however still legal for class 0 packets. In summary, no packets can depend on downgrading packets so deadlock-freedom is preserved.

The idea behind this method is the following: A packet only downgrades when it is about to be dropped and has no other options but to route in the other class. Packets waiting for the downgrading packet may, on the other hand, have other options available, and can adapt and take an alternative deadlock-free path.

2.4 Implementation details

After introducing our routing approach, it is essential to demonstrate its feasibility in hardware. In this section, we discuss possible implementations of the previously introduced concepts. Because the exact implementation depends on the target application, we consider various design possibilities having different levels of complexity and routing flexibility.

2.4.1 Without downgrades

The ability to switch back and forth between packet classes is a powerful feature of our approach. However, in many cases, disabling class downgrades to reduce the design's complexity can be a sensible trade-off. As a first step, we present a very simple implementation under the assumption that packets never need to downgrade.

In this case, instead of using channel tags and continuously checking the deadlock-freedom condition, we can instead control the virtual channel requests in such a way that the condition is always verified. For the sake of illustration, let us assume that 2 packet classes are in use, with an arbitrary number of virtual channels. According to our condition, class 0 packets can request any port in the absence of downgrades, as no negative tags exist. However, class 1 packets cannot request a port in which all virtual channels are occupied by class 0 packets. Therefore, in order to satisfy the deadlock-freedom condition for all packets in the network, it is sufficient to make sure that class 0 packets never occupy all virtual channels simultaneously.

This can be achieved using a simple masking mechanism. During the virtual channel request phase, class 0 packets need to mask their VC requests such that all but one VCs are requested at all times. Class 1 packets always request all VCs and do not apply any masking. Here, we propose two different ways to select which VC to mask:

- **Constant mask:** A first solution is to select one of the two VCs at design time and forbid all class 0 packets from requesting it. The constant mask solution is cheap to implement but somewhat defies the purpose of what we are trying to achieve in terms of VC utilization. Consider the case shown in fig. 2.4 (a), where only one VC is acquired by a packet of class 1 and packets of class 0 are unable to request the free VC because it is masked, even though the request would be legal according to our deadlock-freedom condition.
- **Dynamic mask:** The alternative that we propose is to maintain in each router one V-bit (1 bit per VC) register per output port, except for the Local port. This register is used as a mask, only by class 0 packets, during the virtual channel request phase. The mask update policy is as follows: Whenever a packet of class 1 acquires a VC from a given output port, update the mask on that output port to mask the newly acquired VC (see Fig.

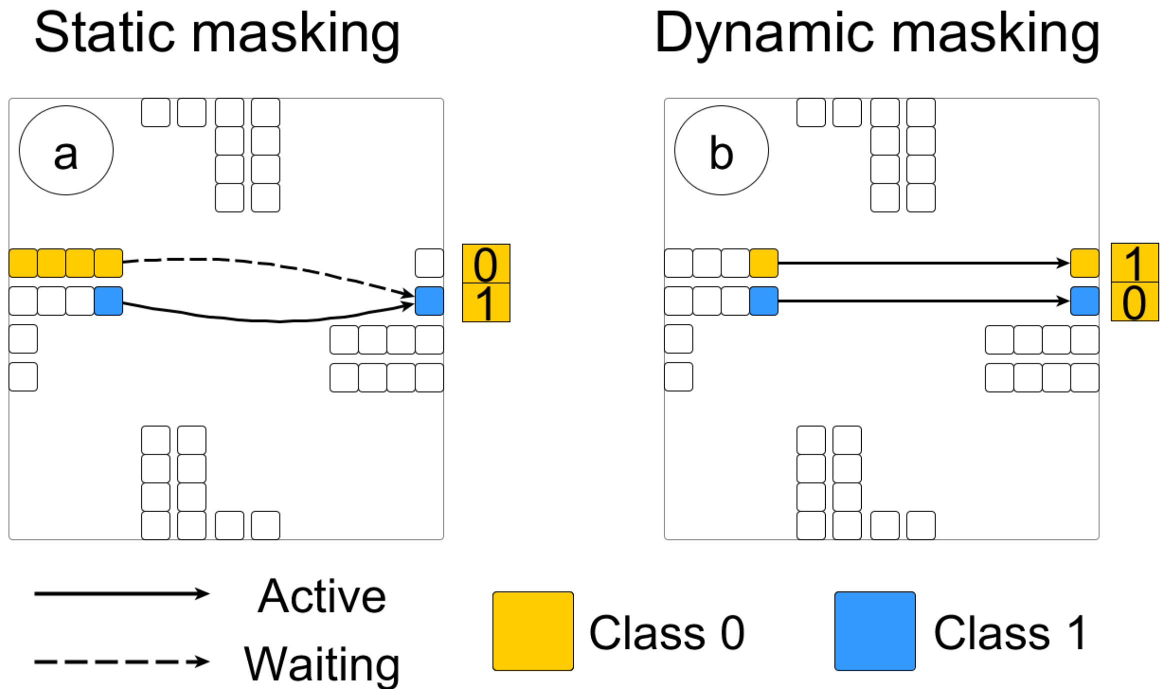

Figure 2.4: Static vs. Dynamic masking. (a) Static: class 0 packets always mask the same channel during VC allocation. (b) Dynamic: The mask changes every time a class 1 packet acquires a VC.

2.4 (b)). That is, as soon as we have a class 1 packet in one VC, make all the other VCs available to class 0 packets. This is a cost-effective method for implementing a dynamic VC acquisition policy.

Because some router architectures can allocate several VCs belonging to the same output port in the same cycle, it can be difficult to test all the possible combinations of classes to decide on the way the mask should be updated. More importantly, making such a decision would require the classes of requesting packets, which are stored in the input FIFOs to be made available at the output of VC allocation, which would imply the use of more complex VC allocation logic. To alleviate this, we propose to take advantage of the fact that the first flit of a packet leaves the router shortly following VC allocation, to perform the mask update as soon as the first packet traverses the crossbar. This is convenient for several reasons. First, because we intercept the flit at the output of the crossbar, there is no need to modify the VC allocator to propagate the packet classes to the output. Instead, we can get this information directly from

the packet header itself. Second, because only one single flit can leave from an output port at a given cycle, the mask update logic becomes much simpler, as all we need to do is mask the output VC of the packet and unmask the other ones. This may only incur a short delay between updates, which has no incidence on correctness, and usually very little impact on performance, as all it means is that some legal VCs may become available to class 0 packets a few cycles later than they could have.

2.4.2 With downgrades

To enable downgrading, we can no longer rely simply on the VC allocation policy to satisfy deadlock-freedom. This is because the condition depends on the behavior of the packets in the downstream routers. In this case, in addition to the masking mechanism, which prevents class 0 packets from fully occupying the same port, each router needs V bits per output unit that we note $T_0, T_1, \ldots T_{v-1}$ to keep track of the tags (0 or 1) of all VCs. In addition, one extra bit per VC is needed to indicate whether a packet is downgrading. We note these bits $D_0, D_1, \ldots D_{v-1}$. During the routing process, two signals are computed for each output port. The first signal E_0, indicates whether the port can be selected by class 0 packets. The second signal E_1 indicates the eligibility of class 1 packets to acquire this port. These eligibility signals can be computed as follows, assuming the current value of the VC mask is $M_0, M_1, \ldots M_{v-1}$:

$$E_0 = \overline{D_0} * M_0 + \overline{D_1} * M_1 + ... + \overline{D_{v-1}} * M_{v-1}$$

$$E_1 = T_0 * \overline{D_0} + T_1 * \overline{D_1} + ... + T_{v-1} * \overline{D_{v-1}}$$

That is, a port is legal for class 0 if at least one VC is has not been negatively tagged and can be requested according to the current value of the mask. Class 1 packets require that one VC has a tag equal to 1. Free VCs must therefore be tagged with "1".

All input VCs that are in "Waiting" state must continuously check these signals as they may change while the packets are waiting to acquire an output VC. When a packet is waiting on an output port that has become illegal for its class, route computation should be repeated and a different port should be selected for that packet. Here again, different options are available.

2.4.3 Repeating route computation

The most reliable to repeat route computation would be to replicate the route computation logic at each input VC, whereas traditionally, only one route computation block is needed at each input port. While this would not be a major issue for simple routing algorithms, because we are in the context of fault-tolerance, non-minimal routing algorithms are typically used, which usually means that the route computation logic is can be quite complex.

Other mechanisms can also be adopted. For instance, during the initial route computation, instead of saving the final selected route, a bit vector containing the list of candidate routes can be saved in a register, and a different candidate can be selected whenever necessary directly from this register, without repeating route computation. This still means that the selection logic needs to be replicated for each VC, however, the comparison of the current and destination addresses, as well as the computation of candidate routes according to the routing class, are only performed once. The only disadvantage of this approach is that it only saves the candidates for one class, which means that it does not support switching to other classes to try more routes upon rerouting.

2.4.4 Negative tag back-propagation

Finally, we consider the implementation of the propagation of negative tags to upstream routers. As shown in Section 2.3, this mechanism is the key to avoiding deadlocks in the presence of downgrades. To implement this feature, it is necessary to place additional signal wires between routers, in addition to the already existing credit signals. One wire is required per VC. Each VC computes the value of the negative tag signal every cycle as follows: if a route computation took place in the current cycle, set a combinational signal R (for route computation) to indicate whether a downgrade was performed by the routing unit. If the channel is in the Active state, i.e. it has acquired an output VC v, then set a combination signal P (for propagation) to the value of the D_v bit defined previously. The final value of the signal will simply be set to $R + P$. That is, the signal is activated if the packet is either downgrading here, or in the downstream routers. At the reception of this signal, the output unit sets the values of D_i accordingly.

2.5 Formal generalization

We have introduced, through an example, the principles of our routing method. In this section, we formalize the proposed approach to provide the necessary proofs, as well as the general form of the routing algorithm to be used by any network topology.

2.5.1 Preliminary definitions

Definition 2.1 (Channel)**.** *Let C be the set of all (virtual) channels and N the set of all nodes in the network. The injection channel associated with a node n is noted* $local_n$*. The node at the end of some channel c is noted* $\rightarrow(c)$*. The state of a channel (Idle or Busy) will be denoted state(c). Every channel can be tagged with a signed integer and the current tag associated with the channel is denoted tag(c).*

Definition 2.2 (Packet)**.** . *Every packet* p *carries a class identifier, denoted* $class(p)$*, and a destination node* $dst(p)$*. The class identifier is an integer* $0 \leq t < T$ *where* T *is the total number of packet classes. The set of channels currently occupied by a packet is noted chan(p), and the last channel visited by a packet, i.e. the one containing the head of the packet, is denoted* $head(p)$.

Definition 2.3 (Routing function)**.** . *A routing function* $R_t : C * N \rightarrow P(P(C))$ *associated with the packet class* t*, where* $P(C)$ *is the power set of* C*, takes the last channel occupied by a packet, its destination, and returns a set of output ports that the packet can traverse to move to the next node. An output port is simply a set of channels. We assume the routing function only supplies healthy ports, i.e. the supplied set does not include faulty ports. The returned set may be empty, if no possible route exists. We say that a routing function is cycle-free if its channel dependency graph, as defined in Duato [1995], is acyclic.*

Definition 2.4 (Selection function)**.** . *A selection function S:* $P(P(C)) \rightarrow P(C)$ *takes a set of ports returned by a routing function and selects one output port, based on congestion, minimal distance, or other criteria. Given an empty set, the function returns an empty set.*

2.5.2 Generic routing algorithm

We are now able to formulate the first rule of the proposed algorithm (Section 2.3), as a condition $I : N * P(C) \rightarrow Bool$ between the class of a packet t and its selected output port o, such that:

$$I(t, o) \Leftrightarrow \exists c \in o, state(c) = Idle \vee tag(c) \geq t \tag{2.1}$$

Using the above definitions, we can now define the route computation function, which given an incoming packet, returns an output port and a (possibly) new packet class. This function, named $rc(p)$ is presented in Algorithm 1. If routing the packet is not possible in its current class, other classes are tried, starting with the higher classes. It is important to note that if another class is used, routing is done as if the packet was just injected (using the injection channel of the current node). This is to bypass any possible restrictions of the routing function.

After the next output port is computed for a packet, it waits until the VC allocator grants it one of the VCs of that port. We denote the output VC granted to a packet p as $next(p)$. While the packet is waiting, assume that $next(p) = Nil$. The complete routing process, up until the packet is granted an output VC, is presented in Algorithm 2. If routing the packet is not possible, it is dropped immediately. Otherwise, if it can be routed in a lower class, the channels holding the packet are tagged with a negative number. The packet then waits for the VC allocator to grant it an output VC. While the packet is waiting, its selected port may no longer satisfy (1). In that case, the whole routing process is repeated. If a packet is granted an output VC, the class of the packet is updated and the granted VC is tagged with the packet's class identifier. It is worth mentioning that the class of the packet is not updated immediately following route computation, so that in case it is rerouted, its original class is tried first.

2.5.3 Proof of deadlock-freedom

For deadlock analysis, we need a few more definitions. Our proof is quite similar to the one used in Dally and Aoki [1993].

Definition 2.5 (Packet dependency)**.** *Let $out(p)$ denote the selected output port for some packet*

Algorithm 1 Route computation

Input:

p : The packet to route

Output:

The selected output port and the new packet class

1: **function** RC(p)
2: Let $r \leftarrow \emptyset$
3: Let $t \leftarrow class(p)$
4: Let $i \leftarrow head(p)$
5: **repeat**
6: $r \leftarrow \{o \in R_t(i, dst(p))/I(t, o)\}$
7: $t \leftarrow (t + 1) \mod T$
8: $i \leftarrow local_{\rightarrow(head(p))}$
9: **until** $r \neq \emptyset$ or $t = class(p)$
10: **return** $S(r)$,$[(t - 1) \mod T]$
11: **end function**

Algorithm 2 The routing algorithm

Input:

p : The packet to route

1: Let r,$t \leftarrow rc(p)$
2: **if** $r = \emptyset$ **then**
3: drop p
4: **else**
5: **if** $t < class(p)$ **then**
6: **for all** $c \in chan(p)$ **do**
7: $tag(c) \leftarrow -1$
8: **end for**
9: **end if**
10: wait until $next(p) \neq Nil$ or $I(t, r) = false$
11: **if** $next(p) = Nil$ **then**
12: route(p) (repeat procedure from top)
13: **else**
14: $class(p) \leftarrow t$
15: $tag(next(p)) \leftarrow t$
16: **end if**
17: **end if**

p. *Packet* p *is said to be depending on packet* q *iff*

$out(p) \cap chan(q) \neq \emptyset$

We will note the set of all packets that a packet p *depends on as* $dep(p)$.

Definition 2.6 (Packet wait-for graph)**.** *Given a set of packets P present in the network at a given moment in time, the packet wait-for graph is a directed graph H(P, E), where E is the set of arcs* (p_i, p_j) *such that* p_i *depends on* p_j.

Theorem 2.1. *If for every packet class* $0 \leq t < T$, R_t *is cycle-free, then the network is deadlock-free.*

Proof. If the network is deadlocked, then there is a packet p_0, such that every possible path $(p_1, p_2 \dots)$ in the packet wait-for graph, where $p_n \in dep(p_{n-1})$, reaches p_0. Given (1), at least one of these paths verifies:

$$class(p_n) \geq class(p_{n-1}) \tag{2.2}$$

Let π be one of the paths verifying (2). Now let us consider a packet p_k such that $\forall p \in \pi, class(p_k) \geq class(p)$. That is, p_k is a packet of the highest class along π.

If $class(p_k) = class(p_0)$ then it means all the packets along π are routed using $R_{class(p_0)}$. However, since $R_{class(p_0)}$ is cycle-free, p_0 cannot reach itself from π. If $class(p_k) > class(p_0)$, then according to (2), every packet that follows p_k in π must be of class $class(p_k)$, and therefore the path never reaches p_0, as it is of class class (p_0).

Consequently, the supposed deadlock configuration can never occur. This proves the algorithm deadlock-free. □

2.5.4 Livelock-freedom and termination

The number of times a packet can be rerouted should be limited to ensure the termination of Algorithm 2. This limit was not explicitly included in the algorithm for readability. For the algorithm to be livelock-free, the number of times a packet can change its class must also be finite.

2.6 Flow control

An important part of every routing solution is the flow control mechanism. In particular, the way virtual channels are reallocated to new packets must not violate the deadlock-freedom condition. Usually, the algorithms that prohibit cycles allow non-empty VCs to be reallocated immediately after the tail flit of the last packet has been transmitted. On the other hand, algorithms that allow cycles but rely on escape channels for deadlock-freedom Duato [1997] require that packets always be at the head of a VC, i.e. only empty VCs can be reallocated. Our methodology offers the best of both worlds. From our condition, it follows naturally that a packet may safely follow another packet of an equal or higher class within the same VC, provided downgrades are not in effect. This means that algorithms based on our methodology are not only able to use more virtual channels through an escape mechanism, but packets need not always be at the head of their VC, allowing for faster flow control. This is one of the strongest features of our methodology.

2.7 A fresh look at fully adaptive routing algorithms

Thus far, we have shown how the proposed design methodology can be used for constructing new highly flexible routing algorithms, as well as enhancing existing ones.

Another interesting aspect of our condition is that it can also provide a more intuitive understanding of older routing algorithms as well. In this section, we showcase this property by reconstructing a fully adaptive algorithm for 2D meshes. More specifically, we will construct, step by step, the fault-tolerant FT-CAR turn model Kumar et al. [2014]. It is an improved variant of the older opt-y algorithm Schwiebert and Jayasimha [1993], which has been proven to provide the highest level of VC utilization and highest level of adaptiveness for 2D meshes with the minimum number of VCs. The FT-CAR algorithm was proven deadlock-free using Duato's theory Duato [1995], relying on channel dependency graphs and extended channel dependency graphs. In our case, we will express the same routing algorithm in terms of simple routing functions and class upgrades.

To enable full adaptiveness, we can define two routing functions corresponding to two rout-

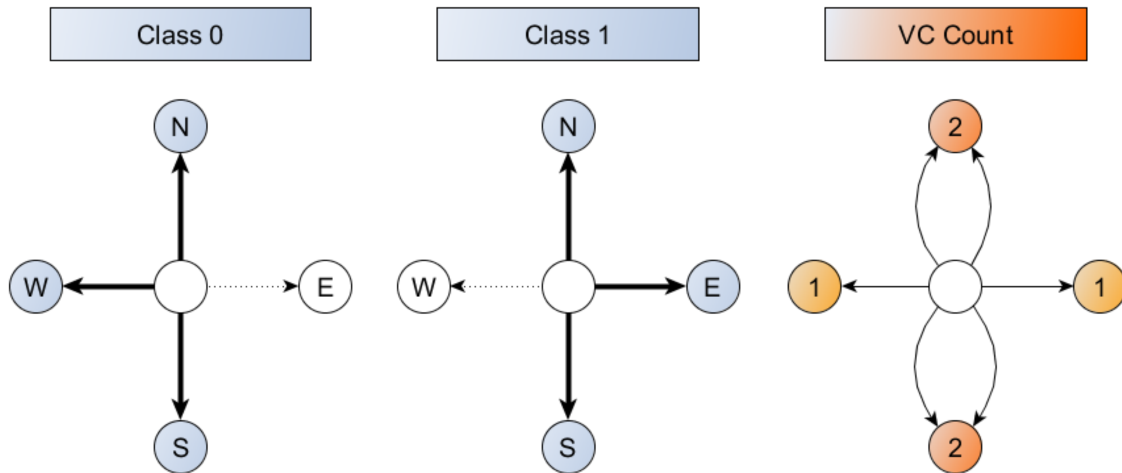

Figure 2.5: Physical channels used by the two classes of FT-CAR.

ing classes. Class 0 provides the North, South and West physical channels. Class 1 provides the North, South and East physical channels, as shown in fig. 2.5. Notice that both functions are cycle-free, as none of them spans both dimensions simultaneously. Routing in each class does not have to be minimal, i.e. packets can be routed away from their destination. Because the North and South physical channels are the only ones to be shared between both classes, we know that 2 VCs are required only in the North and South directions. Assuming the constant masking method introduced in Section 2.4, we also know that packets of class 1 can use both VCs, whereas class 0 packets can take only one VC.

Following our condition, packets can start in class 0, then, if necessary, upgrade to class 1. Packets needing to go east must always upgrade. Also, packets having reached the same column as their destination should always upgrade to class 1, giving them access to both VCs.

Now, if we examine what happens in each virtual channel individually, we can notice that VC0 can be occupied by both classes, and therefore, will include all the possible turns. When we look at VC1, however, we can see that it can only be used by the highest class and cannot include any turns leading to the west direction. Therefore, the resulting algorithm includes the exact same permitted turns and VC acquisitions allowed by Kumar et al. [2014]. However, because it is expressed in terms of very simple routing functions (Fig. 2.5) to which we simply applied our routing rules, it is much easier to make sense of and to prove deadlock-free.

In addition to being a more elegant way for designing fully adaptive routing algorithms, our approach makes it possible to get much more performance out of the available virtual channels thanks to the flexible flow control mechanism described in the previous section, as well as the dynamic masking mechanism, resulting in higher throughput than the existing algorithms. This will be demonstrated in the next section.

2.8 Experimental results

In this section, we evaluate the effectiveness of the proposed approach by simulating and implementing the algorithm presented in sections 2.3.

2.8.1 Simulation setup

We implement the proposed algorithms in an in-house cycle-accurate network-on-chip simulator that runs on GPU. The design of this simulator will be described in detail in Chapter 5. The router architecture comprising the mechanisms described in section 2.4 was accurately modeled. The microarchitecture uses separable input-first switch and VC allocators, as described in Becker and Dally [2009], and credit-based flow control. In all simulations, each VC comprises 4 flit buffers and packets have a fixed size of 5 flits.

2.8.2 Simulation methodology

Regardless of the tested network, simulations are run for 100000 cycles at different injection rates. At the end of the 100000 cycles, statistics are collected and the simulation proceeds until are the packets in the network are drained. During this drain phase, no new packets are injected. The network contains no packets at the end of the simulation, which helps us confirm the absence of deadlocks. A hundred iterations are run for each simulation and the average results are presented. To inject faults, the selected number of links is permanently disabled at the beginning the simulation. At each iteration, the links to disable are selected randomly. However, we always make sure that the network is connected, i.e. a path exists for each source-destination pair. When several routes are available, selection is done based on congestion as

described in Appendix A.

2.8.3 Evaluating the fault-tolerant algorithm

In section 2.3, we have built a fault-tolerant routing algorithm for mesh NoCs using the proposed design methodology. Our goal now is to evaluate two aspects of the resulting design:

1) Network throughput, which is affected by our VC acquisition rules as well as the masking methods described in section 2.4. We consider the normalized throughput, which we compute as the proportion of the packet ejection rate over the injection rate.

2) Reliability, which is affected by the class switching policy (upgrades and downgrades). This is measured as the proportion of packets that were successfully delivered to their destination (including the drain phase) over the total number of injected packets.

To conduct this evaluation, we compare our algorithm to two traditional approaches found in the literature:

- Baseline-0: Packets in the first virtual channel (VC0) are routed using the South-last turn model, whereas packets in the second virtual channel (VC1) are routed following the North-last turn model. Packets can move from VC0 to VC1, but not from VC1 to VC0. As in Ebrahimi et al. [2012], all packets are injected in the first VC to give all of them a chance to upgrade once.
- Baseline-1: The rules are the same as Baseline-0. However, instead of injecting all packets in the first VC, packets are injected in the VC that offers the highest level of adaptiveness. Therefore, northward packets are injected in VC0, whereas southward packets are injected in VC1 Chaix et al. [2010].

We compare these methods to the algorithm described in section 2.3, assuming the dynamic masking technique from section 2.4. As in Baseline-1, packets are directly assigned to their most adaptive class. Packets in class 0 are allowed to upgrade once. To demonstrate how downgrades should be used in practice, we designate a subset of the packets (10%) and mark them as critical packets. These critical packets are injected in class 0 and are allowed to upgrade, downgrade, then upgrade once again. Therefore, they benefit from a higher routing freedom than regular packets. Regular packets in class 1 are not allowed to downgrade.

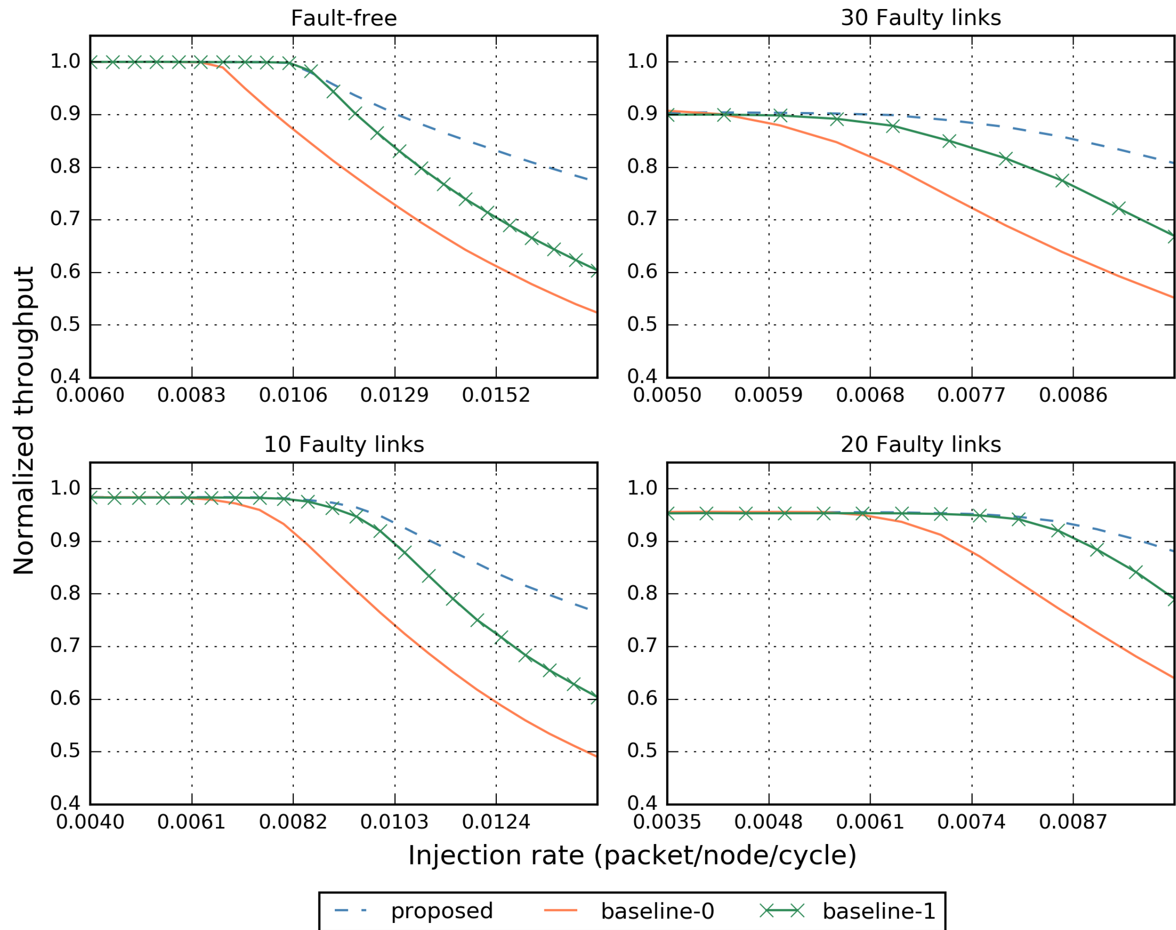

Figure 2.6: Throughput results.

We present the results for an 8x8 mesh network. The metrics are collected both for a fault-free network as well as networks with different numbers of failed links. Throughput results are presented in Fig. 2.6 and fault-tolerance results are presented in Fig. 2.7 and Fig. 2.8. Fig. 2.7 shows the overall proportion of successfully delivered packets, whereas Fig. 2.8 shows the proportion of successfully delivered critical packets.

The first observation that can be made is that although baseline-0 does offer a slightly higher level of fault-tolerance, it does so to the detriment of performance. Conversely, we can see that baseline-1 yields a much higher throughput in a fault-free network, but is less likely to successfully deliver packets in the presence of faults. The proposed algorithm consistently delivers a higher throughput than both approaches even in the presence of a high number of faults. In terms of reliability, we can see that the proposed approach offers a level of reliability

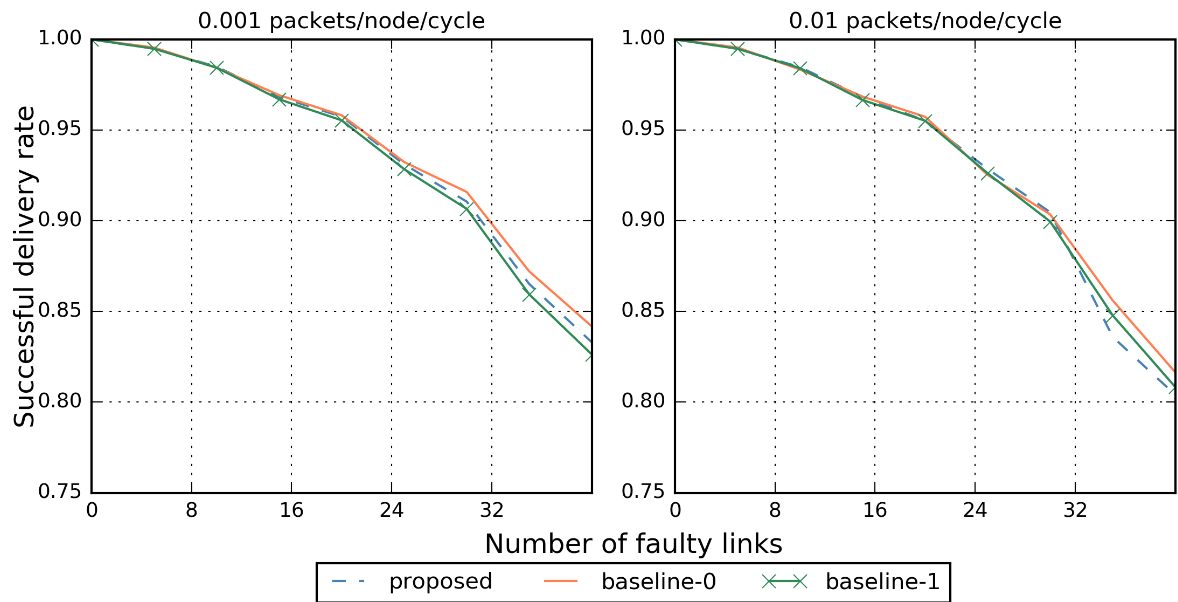

Figure 2.7: Delivery success rate.

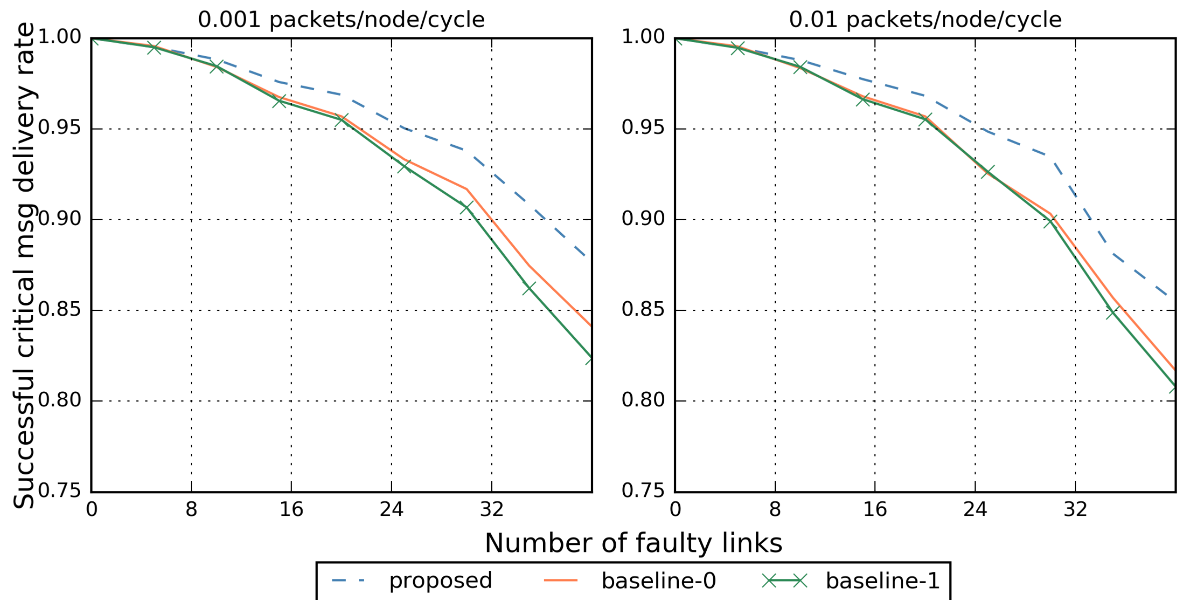

Figure 2.8: Critical packet delivery success rate.

that is in between the two baselines. However, the advantages offered by our class switching policy are clearly demonstrated by a higher delivery rate of critical messages. In summary, our approach has the potential to offer very high levels of reliability with much higher throughput than the existing methods.

2.8.4 The impact of flow control and dynamic masking

Although the fully adaptive routing algorithm described in section 2.7 can be reconstructed as is using only a subset of our theory, it can further benefit from the extra features that we have proposed. In particular, dynamic masking, which was introduced in section 2.4, and the fine-grained flow control scheme introduced in Section 2.6, can greatly enhance the performance of an existing routing algorithm. This is demonstrated in this section.

On the same platform described previously, we implement the original fully adaptive routing algorithm, as well as two variants that make use of our methods. The first variant introduces the flow control scheme, and the second variant adds dynamic masking to the solution. Performance results are presented in fig. 2.9. Here, performance is estimated as the average packet latency. The advantage of both the VC reallocation policy, and the dynamic masking method, are clearly demonstrated.

2.8.5 Hardware synthesis

Finally, we evaluate the overhead of the different mechanisms introduced in section 4. Starting with a baseline router architecture not including any of the mechanisms that are specific to our methodology, we incrementally implement the following features and observe their associated overheads: channel tags, negative tags, dynamic masking. The designs were synthesized using Synopsys Design Compiler and setup to work with an operating frequency of 1GHz, a power supply of 1V, and a commercial ST FD-SOI 28nm Library. Results are presented in table 2.1. As expected, the mechanisms required by our routing methodology incur a negligible area overhead. It is important to note that the negative tags and dynamic masking mechanisms are correct regardless of the number of cycles they take, meaning that they can be implemented without affecting the critical path of the router.

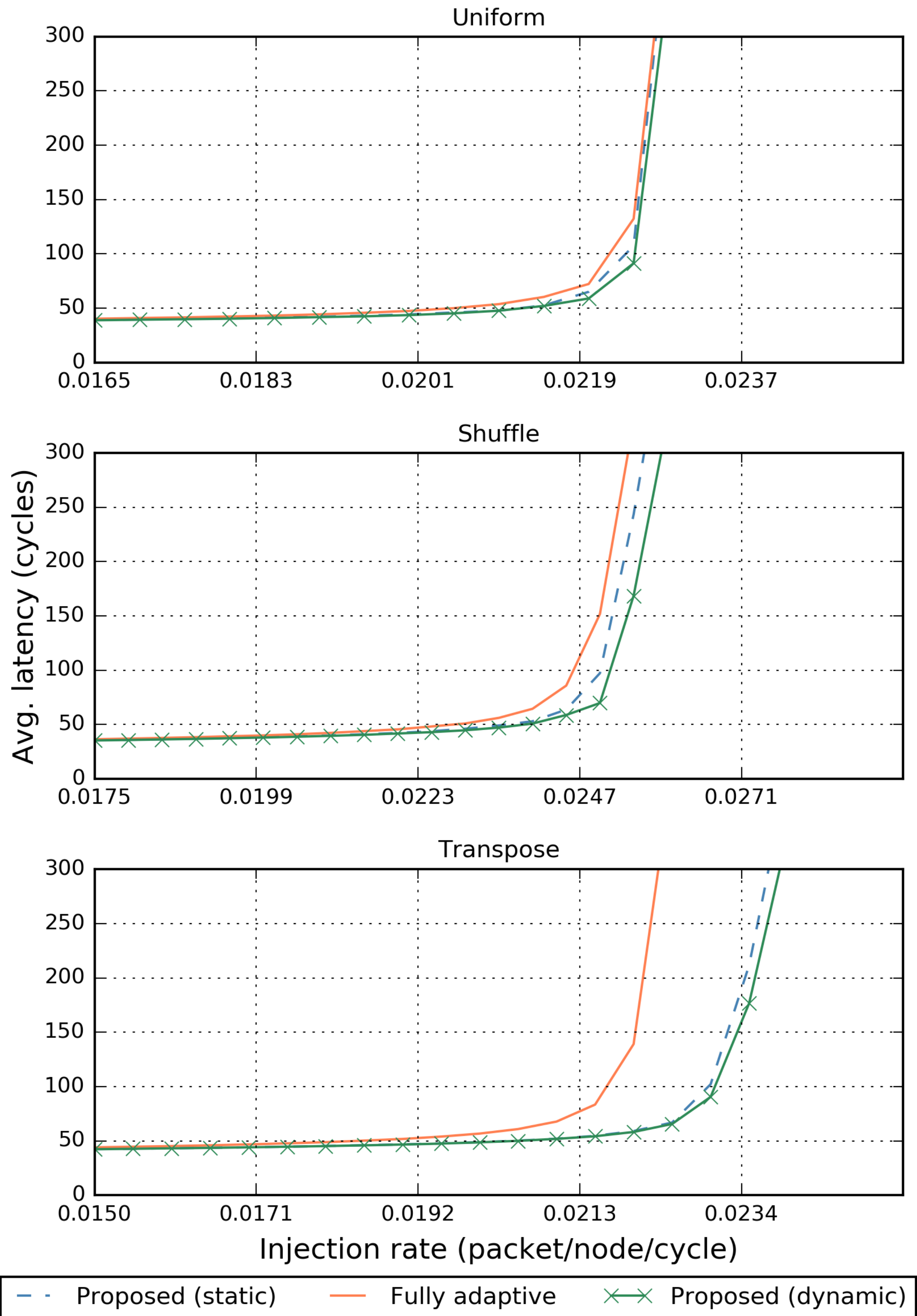

Figure 2.9: Impact of flow control and dynamic masking on average latency.

Table 2.1: Hardware synthesis results

Feature	**Area** (μm^2)	**Power** (mW)
Baseline	33137	25.8442
Channel tags	33198	25.8788
Negative tags	33355	25.9598
Dynamic masking	33362	25.9612

2.9 Conclusions

We have presented a generic, topology-agnostic routing algorithm design methodology that can be used to construct highly flexible routing algorithms in only a few steps: identify a set of routing classes and their respective routing functions, assign a class to every packet in the network, and perform routing following our three rules that guarantee deadlock-free operation. The ability to switch freely between classes is an extremely powerful feature of our method, as it provides packets with a very high degree of routing freedom. In particular, the downgrading mechanism that we have proposed can greatly benefit fault-tolerant systems, as we have demonstrated through our example routing algorithm for faulty 2D mesh NoCs. Our routing rules can also be used to enhance any existing VC-based routing algorithm, as we have shown through our improved version of a fully adaptive routing algorithm, where a simple application of one of our routing rules combined with class upgrades was able to yield a dramatic performance improvement over the original algorithm. Virtual channels are very often used for high-coverage fault-tolerant routing, and are even required for the correct operation of some NoC topologies. Because they are a costly and valuable resource, being able to leverage their throughput enhancement capabilities is absolutely crucial and our contribution serves exactly this purpose.

Part III

ROUTING IN TSV-BASED THREE-DIMENSIONAL NETWORKS-ON-CHIPS

Chapter 3

A framework for scalable distributed TSV assignment

3.1 Introduction

The recent emergence of 3D integration can further increase the viability of Networks-on-Chips as a communication paradigm by enabling the stacking of several silicon layers and allowing for inherently low-latency three-dimensional NoC topologies (3D-NoCs) to be considered Feero and Pande [2009], Pavlidis and Friedman [2007].

Through-Silicon-Via (TSV) is one of the most promising technologies that enable vertical communication between different NoC layers Davis et al. [2005]. However, due to the high cost and low yield of TSVs Benini [2008], Vertically-Partially-Connected NoCs, in which only a subset of the nodes are vertically connected, appear to be a reasonable compromise Bartzas et al. [2007].

Because such topologies require adequate routing algorithms to ensure correct operation and deadlock-freedom, several deadlock-free routing algorithms targeting partially connected NoCs have already been proposed Dubois et al. [2013], Salamat et al. [2015], Ying et al. [2014]. Regardless of which routing rules are applied, since only a subset of nodes are connected to TSVs, routers need a reliable way to locate the nodes that are vertically connected, commonly referred to as Elevators. Both the information regarding the elevators, which is set at configuration time (offline), and the way this information is exploited by the routers during runtime, plays a

decisive role in the chip's performance.

Due to its critical importance to both performance and implementation cost, we dedicate this chapter to the exploration of various algorithms for elevator assignment. Using Elevator-First Dubois et al. [2013] as a baseline routing algorithm, we propose a set of scalable, easy to implement elevator selection strategies, each featuring both the information to be stored in each router, and the algorithms used offline and during runtime to find an elevator. We formally prove all of our methods to be deadlock-free and reachable, meaning that each proposed combination of offline and online selection algorithms is guaranteed to eventually lead to an elevator. We test and compare all of the proposed solutions in terms of performance through cycle-accurate simulation and demonstrate the scalability of our methods through hardware synthesis.

3.2 State-of-the-art

A variety of routing algorithms targeting vertically-partially-connected 3D-NoCs have been proposed in the literature. Many of these algorithms need to follow specific rules that require TSVs to be placed in a specific manner, and often have further constraints as to which TSVs can be selected during runtime. In Ying et al. [2013], the authors propose two routing algorithms named SBSM and DBSM. SBSM selects the vertical link that is the closest to the source node, whereas DBSM selects the vertical link that is the closest to the destination. To make this possible, each router has to know the addresses of all the vertically connected nodes (elevators), which implies a significant hardware overhead. In their more recent works, the authors have introduced the Dynamic-Quadrant Partitioning algorithm Ying et al. [2014], which uses a constant number of bits per router to select an elevator. This makes the solution more interesting in terms of implementation cost. However, the algorithm can only take an elevator located in the north-east quadrant.

The East-then-West (ETW) algorithm Salamat et al. [2015], Salamat et al. [2016b] is a routing algorithm that requires that at least one TSV be placed in the east-most column in order to guarantee reachability. Due to the routing rules, the set of elevators that can be selected is constrained by the position of the destination. Each routers needs to know the location of 3 different elevators: 2 nearest elevators in the east and west directions, and 1 elevator in the

east-most column, in order to select the correct elevator based on the destination's position. Consequently, each router stores 3 node addresses, which limits the scalability of the routing logic.

By contrast with the aforementioned algorithms, Elevator-first Dubois et al. [2013], does not impose any constraints on the placement or the selection of elevators. By using Elevator-First as a baseline algorithm, we are therefore able to develop generic selection strategies that are not limited by the TSV placement strategy or any algorithm-specific constraints.

We identify two approaches to elevator selection for Elevator-First in the literature.

The first approach was introduced as part of the original Elevator-First proposal in Bahmani et al. [2012]. The authors propose to select an elevator for each router at configuration time (offline), and to store its address in a register. When a packet reaches a new layer, the current router prepends a new header to the packet, containing the address of its selected elevator. This mechanism has the major advantage of being generic and compatible with many offline selection algorithms. However, since complete node addresses need to be stored in the routers, the size of configurable data grows with the network size. Also, because the elevator to take is decided at the source router and never changed, there is no runtime adaptivity in the selection of the elevators.

Similarly to our method, the second approach aims at addressing the scalability issues of the original Elevator-First and is part of the LBDR3D framework Niazmand et al. [2016]. The authors use a limited amount of configurable bits in each router, named Vertical Bits, to point to the nearest elevator. The nearest elevator is selected offline based on the Manhattan Distance, and when several elevators with an equal distance from a given router exist, ties are broken randomly. Unfortunately, this specification suffers from a few issues that have not been addressed. First, because different routers may point to different elevators, there can be cases where one router forwards a packet in the direction of its own selected elevator, and where the next router forwards it to another direction towards its own elevator, in such a way that the two directions form an illegal turn, leading to potential deadlocks. In this work, when we consider the Manhattan Distance for elevator selection in section 3.4, we provide offline and online solutions to this critical problem. Second, in Niazmand et al. [2016], no proof of reachability was provided, and additional input signals were introduced to prevent packets from entering livelocks. In this

chapter, we provide a universal formal proof of reachability for all Manhattan-Distance-Based selection approaches, removing the need of any additional signals to ensure reachability.

Neither Elevator-first nor LBDR3D can take the packets' destination into account when selecting an elevator, as the elevators are selected offline in both approaches. This can heavily limit the level of adaptability of the routing solution in some cases. In addition to the Manhattan-Distance –Based algorithms, we also propose a method for selecting an elevator online based on the destination, while still using the exact same amount of information as the distance-driven approaches.

3.3 Target architecture

3.3.1 NoC architecture

We consider a network comprised of several 2D Mesh layers (or tiers) connected vertically using TSV, as shown in Fig. 3.1. Each layer consists of a mixture of classic 2D routers including only 5 ports (East, West, South, North, Local), and 3D routers having either an Up port, a Down port, or both. 3D routers will also be referred to as "Elevators" Dubois et al. [2013]. An upward (downward) elevator is one that connects to the upper (lower) tier.

While we do assume that the channels connecting routers of the same tier are bidirectional, we make no such assumption on the vertical connections. In effect, with a limited number of TSVs available, the system designer may prefer to place an upward and a downward TSV in two different routers instead of placing them in the same router, so as to better balance the load. For instance, in Fig. 3.1, router A and router B are connected using a bidirectional channel (2 TSVs), whereas routers C and D are connected using only an upward TSV.

The problem of finding the best placement strategy for TSVs is beyond the scope of this thesis, and has already been addressed in previous works Foroutan et al. [2014]. The solution provided in this chapter is compatible with any placement strategy.

Due to the limited number of vertically connected nodes, routers need to locate the elevators of their tier in order to be able to communicate with other tiers. We also consider the TSVs to be prone to defects, as well as permanent failure. The information stored in each router regarding

the location of TSVs must be updated to reflect the new state of the network upon failure.

3.3.2 Routing

To provide a deadlock-free routing solution, we rely on the routing rules of Elevator-First Dubois et al. [2013]. That is, the network is virtually portioned into two virtual networks using separate input FIFOs (a.k.a. Virtual Channels). Packets heading to an upper layer are injected in the first virtual channel, whereas packets heading down are routed in the second virtual channel. As per Elevator-First, routing within each layer is performed using a deadlock-free 2D routing algorithm. For the sake of illustration, the XY algorithm is assumed throughout this chapter.

Despite using the same deadlock avoidance technique, our routing methodology is different from the one described in Dubois et al. [2013]. In Dubois et al. [2013], each router stores the address of the nearest elevator and every time the packet reaches a new layer, a new header containing the elevator's address is prepended to the packet. Our goal is to find an elevator in a distributed manner while keeping the routing logic as simple as possible. Threfore, our approach does not involve storing node addresses. Instead, each router includes a fixed number of configurable bits named Elevator Location Bits, which contain information about the location of elevators. These bits can be reconfigured at any time to reflect the new state of the network upon the occurrence of TSV failures Eghbal et al. [2015]. The number of these bits is independent of the size of the topology. In addition, these bits are never inserted in the packet's header, but are used directly by the route computation logic to guide packets towards an elevator. The route computation logic can be generically described as in Algorithm 3. As is the case for all conventional router architectures, route computation starts by comparing the router's address to that of the destination. The result is a vector of bits that we call Compare Bits. If the destination is in the same layer, the simple logic of XY is used to determine the next output port. If the destination is in a different layer and the current router is an elevator, then route towards the destination layer (Line 7). If the destination is in a different layer and the current router is not an elevator, then use the Elevator Location Bits and the Compare Bits to route towards an elevator (Line 9).

Our focus in the rest of the chapter is to answer the following questions: - What to put in

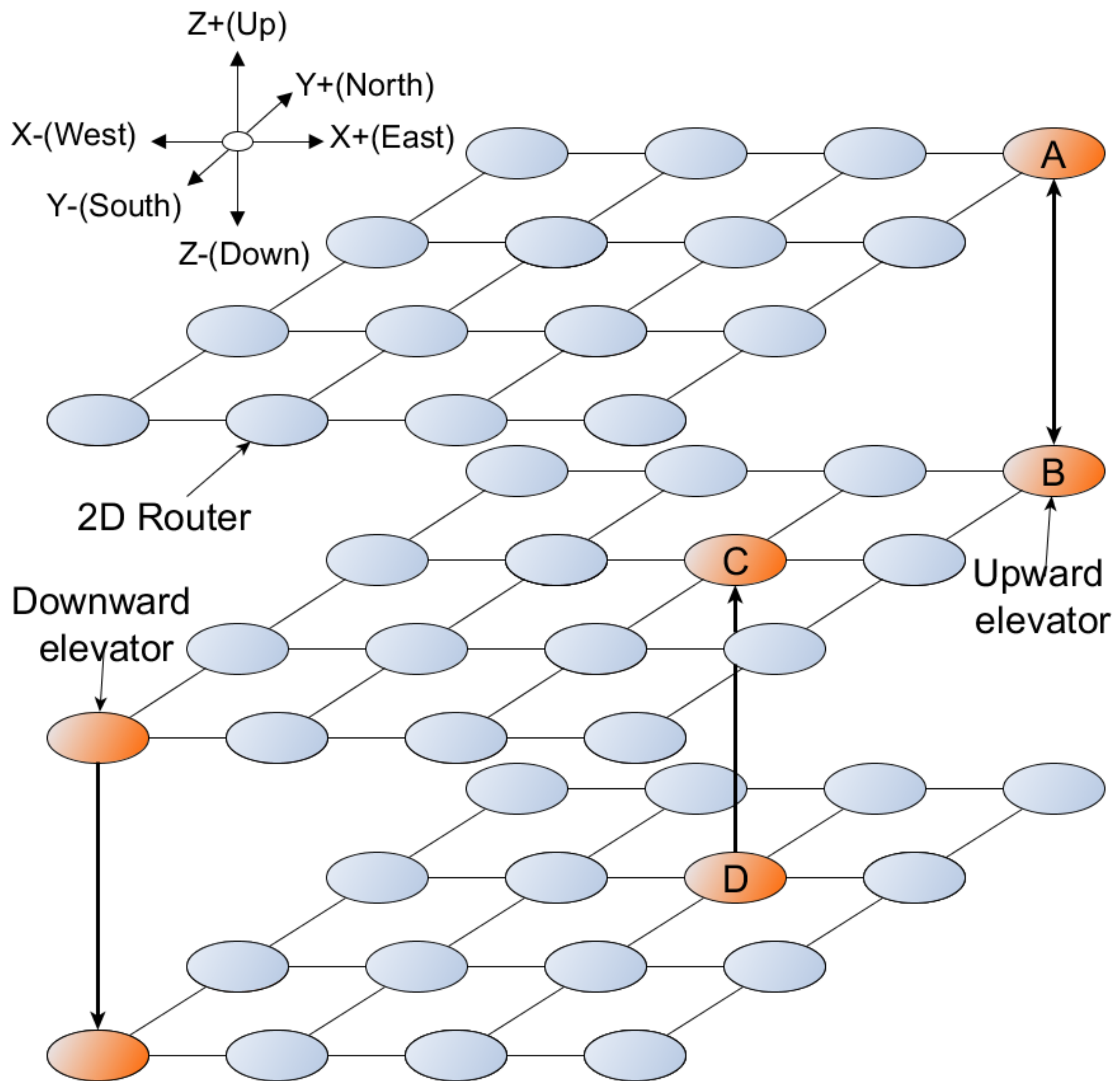

Figure 3.1: Overview of the partially connected 3D-NoC Topology.

Algorithm 3 Route computation logic (generic)

Input:
Elevator (Elevator location bits)
Output:
Direction (Output port)
Variable: Compare (Comparison bits)

```
 1: Compare.East = current.X < dest.X
 2: Compare.West = current.X > dest.X
 3: Compare.North = current.Y < dest.Y
 4: Compare.South = current.Y > dest.Y
 5: if current.Z ≠ dest.Z then                              ▷ different layer
 6:     if current.isElevator then
 7:         Direction = (current.Z < dest.Z?Up : Down)
 8:     else
 9:         Use Elevator and Compare to find an elevator
10:     end if
11: else                                                    ▷ XY logic
12:     if Compare.East then
13:         Direction = East
14:     else if Compare.West then
15:         Direction = West
16:     else if Compare.North then
17:         Direction = North
18:     else if Compare.South then
19:         Direction = South
20:     else
21:         Direction = Local
22:     end if
23: end if
```

the Elevator Location Bits ? - How to use these bits online (Algorithm 3 – Line 9). Several approaches offering different levels of complexity and performance are proposed.

3.4 Manhattan-distance-based elevator selection

One possible solution to our problem simply consists in choosing an elevator that is located as close as possible to the current router. The idea behind this approach is to minimize the time spent searching for an elevator and to quickly reach the destination layer. This is the criterion of selection that many other works have been adopting. In this section, we present three efficient algorithms that exploit the properties of Manhattan Distance to minimize the information required for locating the nearest elevators, while still guaranteeing reachability.

3.4.1 Elevator location bits

To reach one of the nearest TSV pillars, we will demonstrate that only 8 bits of information per router are sufficient. Let Elevator be a 4-bit vector stored within a router and (Elevator.North, Elevator.East, Elevator.South, Elevator.West) its four configurable bits. This vector uses the same encoding as the compare bits described previously. That is, each bit is set so as to indicate whether the selected elevator is in the given direction. For instance, if the offline configuration algorithm selects an elevator located North-East to the current router, Elevator will be set to (1,1,0,0). Each router needs to store two such bit vectors, one for the Upward elevator, and one for the Downward elevator. This encoding allows for the efficient routing algorithm implementation presented in Algorithm 4. Here, Elevator is set to either the upward or downward elevator according to the destination.

3.4.2 Safe Selection Algorithm (md-safe)

Given this encoding, all that the configuration algorithm has to do is select one elevator for each router. Here again, several approaches are possible. One thing to take into consideration is the fact that due to the distributed nature of this routing algorithm, it is possible for different routers to point to different nearest elevators, and consequently, one router that forwards a packet in

Algorithm 4 Route computation with MD-based elevator selection

Input:
Elevator (Elevator location bits)
Output:
Direction (Output port)
Variable: Compare (Comparison bits)

1: $Compare.East = current.X < dest.X$
2: $Compare.West = current.X > dest.X$
3: $Compare.North = current.Y < dest.Y$
4: $Compare.South = current.Y > dest.Y$
5: **if** $current.Z \neq dest.Z$ **then** ▷ seek elevator
6: **if** $current.isElevator$ **then**
7: $Direction = (current.Z < dest.Z?Up : Down)$
8: **else**
9: $Direction = XY(Elevator)$
10: **end if**
11: **else**
12: $Direction = XY(Compare)$
13: **end if**

the direction of its nearest elevator cannot guarantee that it will reach that same elevator after traversing the next hops. The first approach that we propose is to set these bits in such a way that all routers along one path point to the same elevator. This can be achieved using Algorithm 5 for each layer. Here, we iterate through each elevator in turn and check if it is the nearest elevator to every node in the layer. Even if the distance from some node to the new elevator is the same as its previously assigned nearest elevator, the new elevator is still preferred. This ensures that starting from any initial node A, who is pointing to elevator node E, following the routing function in the direction of E reaches a node B that points to the same nearest elevator E. If B had a nearest elevator node F different from E, then according to Algorithm 5, F would also be the nearest elevator to A. Therefore, our algorithm inherently guarantees that packets always reach their intended elevator. This approach also has the advantage of being independent of the planar routing algorithm, i.e. the offline algorithm is compatible with any online routing function, including adaptive routing algorithms.

3.4.3 Randomized Selection Algorithm (md-random)

While the safe selection algorithm has the interesting property of achieving consensus among several routers about where the nearest elevator is, it may not offer the best load balancing and

Algorithm 5 Setting the Elevator bits (safe)

Output:
Elevator[LayerNodes] (Elevator location bits)
Variable: Dist[LayerNodes] (Distance to closest elevator)

1: **for all** node i **do**
2: Initialize Dist[i] to infinity
3: Initialize Elevator[i] to all zeros
4: **end for**
5: **for all** elevator (xE, yE) **do**
6: **for all** node i of coord (x, y) **do**
7: **if** $|xE - x| + |yE - y| \leq Dist[i]$ **then**
8: $Dist[i] = |xE - x| + |yE - y|$
9: $Elevator[i].North = (yE > y)$
10: $Elevator[i].South = (yE < y)$
11: $Elevator[i].West = (xE < x)$
12: $Elevator[i].East = (xE > x)$
13: **end if**
14: **end for**
15: **end for**

elevator utilization, as many nodes will attempt to reach the exact same elevator at once. This can be problematic for performance-critical applications. Intuitively, better performance can be achieved by selecting a random elevator among several nearest elevators, as the load would be more uniformly distributed among TSVs. One challenging aspect of such a randomized approach is that it may cause packets to violate the routing rules, resulting in deadlocks.

Consider the example shown in Fig. 3.2, where a packet originates at node A and needs to take an elevator. In this example, two elevators, E1 and E2 are available. They are assigned to nodes A and B respectively. At router A, the packet takes the North direction to reach E1. However, at node B, it will take the West turn to reach E2 following the XY algorithm. By taking the West turn after the North turn, the algorithm has already violated the rules of XY. We propose two methods to alleviate this issue.

The first approach consists in rewriting Algorithm 4 in such a way that Y to X turns cannot be made. The alternative routing algorithm is presented in Algorithm 6, where input_direction indicates the direction from which the packet has arrived. The idea behind this method is straightforward: if a packet in search for an elevator is received at the north (or south) port, then it forcibly has an elevator in the south (or north) direction, otherwise the previous router would not have forwarded it following the Y dimension. This means that it is enough to make sure that packets traveling along the Y axis keep going in the same direction until an elevator is

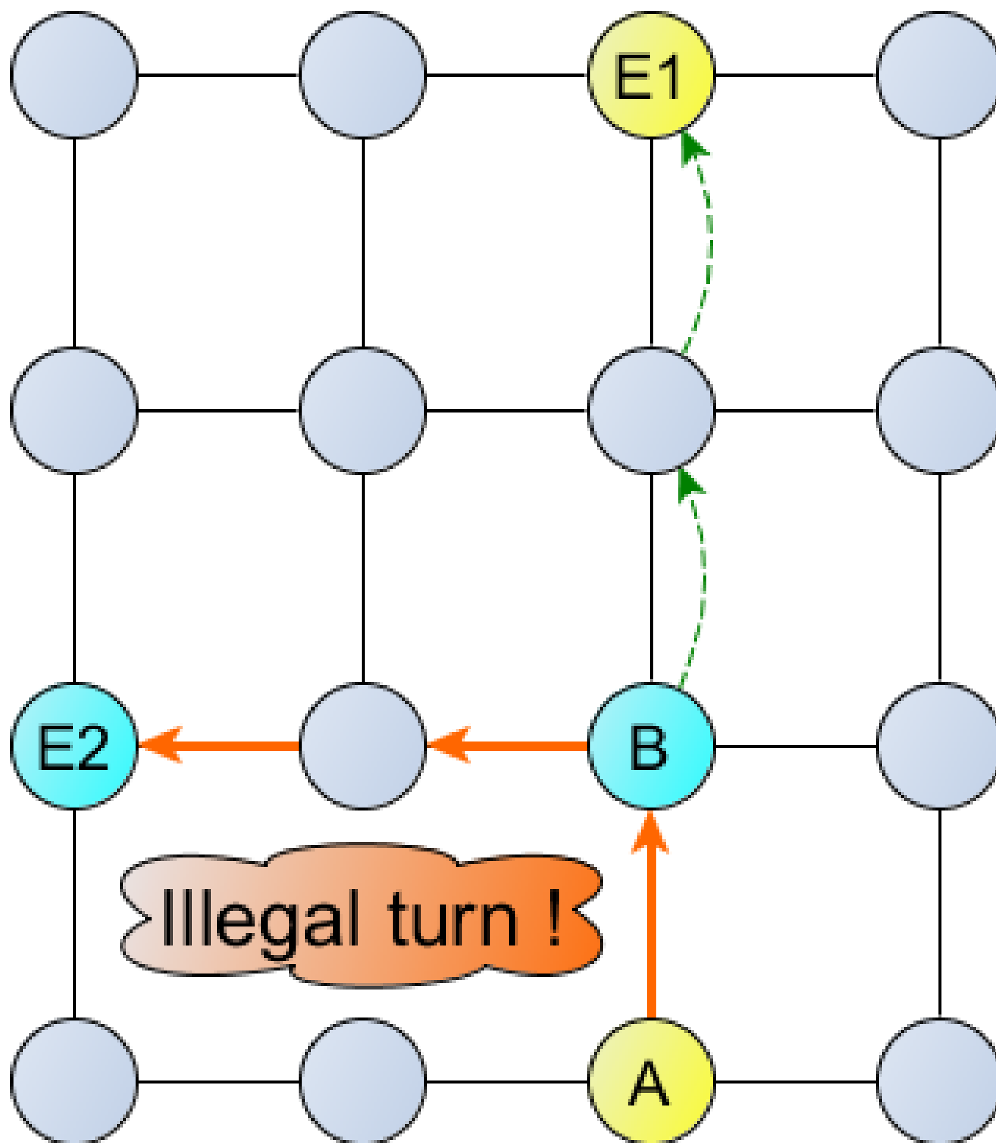

Figure 3.2: Illustration of a potential deadlock scenario.

eventually reached. While simple, the main drawback of this approach is that it heavily depends on the XY algorithm and is very hard to adapt to other algorithms. In fact, in the case of an adaptive routing algorithm, it is not possible for the current router to infer which elevator was intended for the packet simply from the direction it has taken last, as it may have been only one of the possible directions available at the previous router.

Algorithm 6 Route computation for online deadlock-freedom (random-online)

Input:
Elevator (Elevator location bits)
Output:
Direction (Output port)
Variables: Compare (Comparison bits)

1: $Compare.East = current.X < dest.X$
2: $Compare.West = current.X > dest.X$
3: $Compare.North = current.Y < dest.Y$
4: $Compare.South = current.Y > dest.Y$
5: **if** $current.Z \neq dest.Z$ **then**
6:     **if** $current.isElevator$ **then**
7:         $Direction = (current.Z < dest.Z?Up : Down)$
8:     **else if** $input_direction == North$ **then**
9:         $Direction = South$
10:     **else if** $input_direction == South$ **then**
11:         $Direction = North$
12:     **else**
13:         $Direction = XY(Elevator)$
14:     **end if**
15: **else**
16:     $Direction = XY(Compare)$
17: **end if**

A less rigid approach consists in maintaining the original online routing algorithm, and preventing deadlock scenarios at the offline selection stage. We propose a method that is compatible with XY as well as the three deadlock-free adaptive turn models Glass and Ni [1992]. One property of all of these algorithms is that they impose an order on the traversal of physical channels. For instance, in the west-first turn model, the east, north and south directions are taken last. In the north-last turn model, the north direction is taken last. In the XY algorithm, north and south are taken last. The idea is to exploit this property during the elevator selection process, by giving precedence to the elevators that can be reached using only the last directions of the given routing algorithm.

Since we are working with the XY algorithm, the selection algorithm can be written as in

7. By prioritizing the nearest elevators that are on the same column, we ensure that a packet only takes East or West when there are no closest elevators in the same column. Once the Y dimension is taken, all subsequent routers will agree that there is a nearest elevator on the same column as per Algorithm 7, and there will therefore not be a need to take the X directions again. This effectively removes any risk of deadlocks.

A more general form of the algorithm, which is not tied to a specific routing algorithm, is also presented in Algorithm 8. This generic algorithm takes the set of the last directions of the planar algorithm $LastDirSet$ as an input. The last direction set is defined as follows:

Definition 3.1 (Last direction set)**.** *Let D be the set of all planar directions in a mesh network, such that $D = \{West, South, East, North\}$. A deadlock-free planar routing algorithm can be defined as a list A of subsets of D. Let $A = \{D_0, D_1, ..., D_n\}$. The last direction set L of algorithm A is simply the last element (D_n) of A.*

For instance, the west-first routing algorithm can be written as
$A = [\{West\}, \{North, South, East\}]$.
The last direction set of the west-first algorithm is therefore
$L = \{North, South, East\}$.

Another challenging aspect of randomized elevator assignment is to ensure that packets eventually reach an elevator. In what follows, we provide an elaborate proof of reachability and livelock-freedom by using the properties of Manhattan Distance.

3.5 Proof of reachability for Manhattan-Distance-Based approaches

Because packets needing to reach a different layer may traverse routers that point to different elevators as per our selection algorithms, it is necessary to make sure that packets are always able to reach an elevator, i.e. that they are never lead to a dead end, and are never able to fluctuate between different nodes indefinitely.

While related works introduce extra signals to prevent packet looping at runtime Niazmand et al. [2016], we provide a formal proof of reachability showing that packets are bound to reach

Algorithm 7 Setting the Elevator Bits for offline deadlock-freedom (random-offline)

Output:
Elevator[LayerNodes] (Elevator location bits)
1: **for all** node i **do**
2: Initialize Elevator[i] to all zeros
3: **end for**
4: **for all** node i of coord (x, y) **do**
5: E = List of all elevators
6: Sort E By distance from i
7: $Min = e \in E / distance(e, i) == distance(E[0], i)$
8: $SameCol = (x', y') \in Min / x' == x$
9: **if** SameCol is empty **then**
10: $(xE, yE) =$ random elevator from Min
11: **else**
12: $(xE, yE) =$ random elevator from SameCol
13: **end if**
14: $Elevator[i].North = (yE > y)$
15: $Elevator[i].South = (yE < y)$
16: $Elevator[i].West = (xE < x)$
17: $Elevator[i].East = (xE > x)$
18: **end for**

an elevator regardless of the criteria used to select one elevator among the nearest ones. We further show that routing from any node to the final elevator is always done following minimal distance.

This proof allows for very efficient routing logic implementations, as it completely removes the need to perform any runtime checks to make sure packets do not reach a dead-end while seeking an elevator.

Theorem 3.1 (Elevator reachability)**.** *If each router forwards a packet one hop closer to one of its nearest elevators, then the packet will eventually reach an elevator.*

Proof. Let (X_c, Y_c) be the coordinates of the current router C in a given routing scenario. Let (X_{ec}, Y_{ec}) be the coordinates of the elevator E_c selected by the offline algorithm for router C. The Manhattan distance between node C and its elevator E_c is defined as: $MD(C, E_c) = |X_c - X_{ec}| + |Y_c - Y_{ec}|$.

We know, from Algorithm 11, that the current router will forward packets to a next node N (X_n, Y_n) so as to get closer to E_c.

Algorithm 8 Generic Algorithm for Setting the Elevator Bits for offline deadlock-freedom

Input:
LastDirSet (The last directions set of the routing algorithm)
Output:
Elevator[LayerNodes] (Elevator location bits)

1: **for all** node i **do**
2: Initialize Elevator[i] to all zeros
3: **end for**
4: **for all** node i of coord (x, y) **do**
5: E = List of all elevators
6: Sort E By distance from i
7: $Min = e \in E/distance(e, i) == distance(E[0], i)$
8: $Last$ = $\emptyset$
9: **for all** $e(x', y') \in Min$ **do**
10: Dir = $\emptyset$
11: **if** $x' > x$ **then**
12: add East to Dir
13: **end if**
14: **if** $x' < x$ **then**
15: add West to Dir
16: **end if**
17: **if** $y' > y$ **then**
18: add North to Dir
19: **end if**
20: **if** $y' < y$ **then**
21: add South to Dir
22: **end if**
23: **if** $Dir \subset LastDirSet$ **then**
24: add e to $Last$
25: **end if**
26: **end for**
27: **if** $Last$ is empty **then**
28: $(xE, yE) =$ random elevator from Min
29: **else**
30: $(xE, yE) =$ random elevator from $Last$
31: **end if**
32: $Elevator[i].North = (yE > y)$
33: $Elevator[i].South = (yE < y)$
34: $Elevator[i].West = (xE < x)$
35: $Elevator[i].East = (xE > x)$
36: **end for**

By definition, we have:

$$MD(N, E_c) = MD(C, E_c) - 1 \tag{3.1}$$

That is, router N is closer to E_c than C. Now let E_n denote the elevator selected by the offline algorithm for N, and let (X_{en}, Y_{en}) be its coordinates. Because E_c was selected by the offline algorithm as the elevator of C, we know that E_n cannot be closer to C than E_c, as otherwise E_n would have been selected as the nearest elevator instead. This means that:

$$MD(C, E_c) \leq MD(C, E_n) \tag{3.2}$$

The same applies to the selection of E_n for N:

$$MD(N, E_n) \leq MD(N, E_c) \tag{3.3}$$

By combining (1) and (3), we obtain:

$$MD(N, E_n) \leq MD(C, E_c) - 1 \tag{3.4}$$

This is an important property, as it shows that the distance between a node and its own selected elevator decreases at every hop. By recurrence, this implies that the distance will eventually reach 0, thereby proving that packets always reach an elevator. □

Theorem 3.2 (Minimality)**.** *When seeking an elevator, the path a packet takes from any node to the final elevator is a minimal path.*

Proof. First, we show the distance between a node and its own elevator decreases by exactly 1 at every traversed hop.

Let us assume that there is a node C, with elevator E_c, that forwards a packet to a next hop N, with elevator E_n, such that:

$$MD(N, E_n) < MD(C, E_c) - 1 \tag{3.5}$$

We know that node C is able to reach elevator E_n in $MD(N, E_n)$ hops, plus 1 hop from

C to N. And from (5), we know that the distance from C to its own elevator E_c is greater than $MD(N, E_n) + 1$. In other words, E_n is closer to C than E_c, which contradicts with (2).

Consequently, we obtain the following equation from (4):

$$MD(N, E_n) = MD(C, E_c) - 1 \tag{3.6}$$

Let $F(X_f, Y_f)$ be the finally reached elevator in a given routing scenario. Assuming non-minimal routing, the list of visited nodes from the source to F must include two nodes P and Q, such that P was visited before Q and routing from Q to F was done following minimal distance and:

$$MD(P, F) = MD(Q, F) \tag{3.7}$$

Assuming H hops were visited between P and Q, we have from (6) that:

$$MD(P, E_p) = MD(Q, E_q) + H \tag{3.8}$$

Because routing from Q to F was done following minimal distance, the number of hops from Q to F is $MD(Q, F)$, also, from (6) we know that this also corresponds to $MD(Q, E_q)$. That is:

$$MD(Q, F) = MD(Q, E_q) \tag{3.9}$$

From (7), (8) and (9), we can write:

$$MD(P, E_p) = MD(P, F) + H \tag{3.10}$$

This means that F is closer to P than E_p which again contradicts with our initial assumption that E_p is the closest elevator of P. Therefore, routing from a source node to the final elevator is always done following the minimum distance.

□

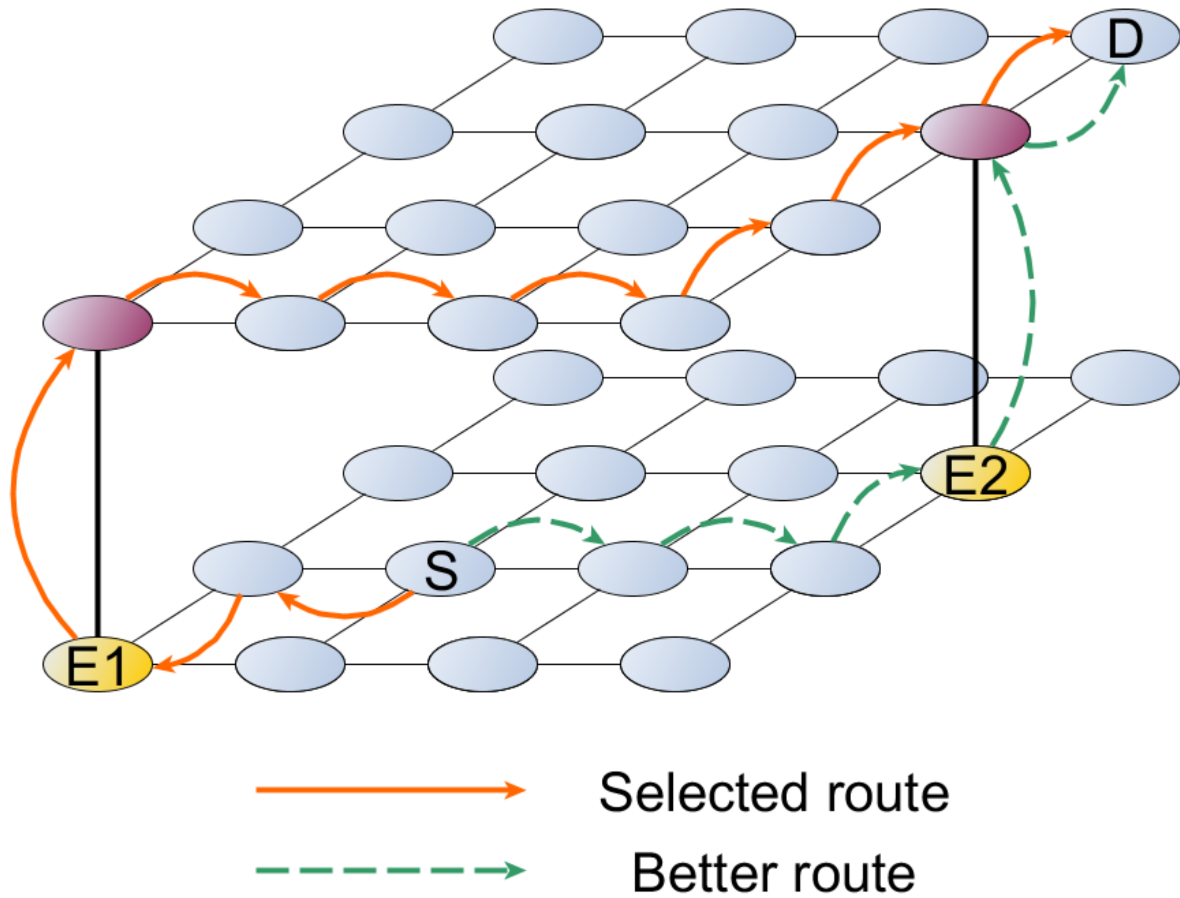

Figure 3.3: Example of an inefficient route when using MD-based algorithms.

3.6 Optimistic elevator selection

The goal of the MD-based selection algorithms presented in the previous section was to minimize the distance between a source node and the selected elevator. While this is a reasonable option most of the time, it can perform poorly in various scenarios. The main reason being that the position of the final destination of the packets is never taken into account while routing a packet towards its elevator. As an example, let us consider the example shown in Fig. 3.3. Here, the packet has originated at node S and is destined for node D located in a different layer. Using the previously defined algorithms, the packet is routed to the nearest elevator E1, drifting away from the destination, before reaching the destination layer. The total hop count from source to destination could have been greatly reduced had the packet taken elevator E2.

In this section, we introduce another type of selection called Optimistic Elevator Selection. In this approach, routers attempt to reduce the distance to an elevator AND to the final destination simultaneously.

3.6.1 Elevator location bits

The exact same amount of configuration bits is required for the optimistic selection approach as the MD-based approach, thereby maintaining scalability. Here again, each router stores two 4-bit vectors (North,East,South,West). However, the meaning of these bits differs from the previous specification. Instead of pointing to a specific elevator location, these bits act as a compass that vaguely indicates the presence of any elevators in the given directions. The North and South bits are set if there is at least one elevator in the same column to the North or to the South, respectively. The East and West, on the other hand, are used to indicate the existence of any elevator in the east or west directions, not necessarily on the same row as the current router. That is, East is set if at least one elevator exists in the east, north-east or south-east directions. The algorithm used to set these bits is described in Algorithm 9.

Algorithm 9 Setting the Elevator bits (optimistic)

Output:
  Elevator[LayerNodes] (Elevator location bits)
1: **for all** node i **do**
2:   Initialize Elevator[i] to all zeros
3: **end for**
4: **for all** elevator (xE, yE) **do**
5:   **for all** node i of coord (x, y) **do**
6:     $Elevator[i].North| = (yE > y\&\&xE == x)$
7:     $Elevator[i].South| = (yE < y\&\&xE == x)$
8:     $Elevator[i].West| = (xE < x)$
9:     $Elevator[i].East| = (xE > x)$
10:   **end for**
11: **end for**

3.6.2 Routing algorithm (optimistic)

Because the selection of an elevator now accounts for the destination's position, most of the selection complexity must be transferred to the online route computation algorithm, i.e. the hardware. Of course, Algorithm 4 can no longer be used. Instead we replace it by Algorithm

10. It should be noted that input_direction is assumed to be a generation variable, therefore, the test on the input direction is not performed at runtime but is processed at generation time. This means that in hardware, each input port will include a different combinational logic for this algorithm. As can be seen, the logic is still quite simple.

The routing process can be described as follows. First route along the X dimension while trying to get closer to the destination as long as possible. If the column of the destination was reached, route along the Y dimension while trying to get closer to the destination. If the destination was exceeded following the X dimension, keep going in the same direction until a column having an elevator is reached. If the destination is exceeded in the Y dimension, then keep going in the same direction until an elevator is reached.

Because routing is still performed following the XY rules, our algorithm has no impact on deadlock-freedom. Moreover, because the routing algorithm never selects a direction unless the configuration bits indicate the presence of an elevator, we also guarantee that the algorithm is always capable of finding an elevator. Since the routing rules are enforced by the routing logic, no proof of reachability is necessary.

3.7 Experimental results

After exploring various strategies for elevator selection, we are now going to evaluate and compare them to have a clear idea of the scenarios under which each solution is the most appropriate. Two aspects of our algorithms are evaluated: hardware implementation cost and network performance.

3.7.1 Hardware synthesis results

To estimate the cost of the proposed solutions, we have implemented a 3D router's Route Computation Unit (RCU) in SystemVerilog. The RCU takes as an input the flits coming from all the router's input channels and outputs the route computation result, i.e. next output port, for each input channel. It is worth reminding that a 3D router includes one local port (packets coming from the tile), two vertical ports (Up and down) and 4 planar ports (east, west, south, north), such that each planar port includes 2 virtual channels.

Algorithm 10 Route computation (optimistic)

```
Input:
    Elevator (Elevator location bits)
Output:
    Direction (Output port)
    Variables: Compare (Comparison bits)
 1: Compare.East = current.X < dest.X
 2: Compare.West = current.X > dest.X
 3: Compare.North = current.Y < dest.Y
 4: Compare.South = current.Y > dest.Y
 5: if current.Z ≠ dest.Z then
 6:     if current.isElevator then
 7:         Direction = (current.Z < dest.Z?Up : Down)
 8:     else
 9:         if input_direction == East then
10:             if Elevator.West&Compare.West then
11:                 Direction = West
12:             else if Elevator.North&Compare.North then
13:                 Direction = North
14:             else if Elevator.South&Compare.South then
15:                 Direction = South
16:             else if Elevator.North then
17:                 Direction = North
18:             else if Elevator.South then
19:                 Direction = South
20:             else                                  ▷ the elevator is West
21:                 Direction = West
22:             end if
23:         else if input_direction == West then
24:             Same as East (replace West by East)
25:         else if input_direction == North then
26:             Direction = South
27:         else if input_direction == South then
28:             Direction = North
29:         else                                      ▷ have not engaged in direction yet
30:             if Elevator.West&Compare.West then
31:                 Direction = West
32:             else if Elevator.East&Compare.East then
33:                 Direction = East
34:             else if Elevator.North&Compare.North then
35:                 Direction = North
36:             else if Elevator.South&Compare.South then
37:                 Direction = South
38:             else if Elevator.North then
39:                 Direction = North
40:             else if Elevator.South then
41:                 Direction = South
42:             else if Direction.West then
43:                 Direction = West
44:             else
45:                 Direction = East
46:             end if
47:         end if
48:     end if
49: else
50:     Direction = XY(Compare)
51: end if
```

For new packets, i.e. when a head flit is read from an input channel at a given cycle, a new route is computed and stored in a register. The body and tail flits of the same packets will simply be routed to the same port.

We implement both the MD-based (Algorithm 6) and optimistic (Algorithm 10) routing algorithms described in this chapter. As a reference, we also implement the RCU of the original Elevator-First algorithm, as described in Bahmani et al. [2012]. It differs from our architecture in that the local and vertical ports generate a temporary header that includes the address of the selected elevator. The logic used to create and output this temporary header followed by the original flits is implemented in the RCU, however, the logic used to remove this header is performed on the output side, so the RCU simply computes a signal that indicates whether this header should be written or not.

The RCU also stores the configurable information required for locating the elevator nodes, i.e. the elevator bits for our algorithms and the full coordinates of the selected elevator for Elevator-First.

The implemented RCU is synthesized using Synopsis Design Vision. We have set the optimization parameters to achieve the minimum area overall, like area effort for high and set the max area to zero. Furthermore, for the synthesis process, the NanGate Open Cell 45 nm Library Nangate [2017] is used, and the designs were setup to work with the operating frequency of 1GHz, and a power supply of 1V.

The synthesis results are presented in Table 4.1 for different network sizes. First, it is interesting to compare the three algorithms for a layer size of 4x4, because these dimensions require 4 bits to address each elevator, which means that the size of configurable data in both Elevator-first and the proposed algorithms is identical. Here, it can be seen that the area of Elevator-First is slightly larger than the MD-based proposed algorithm, mainly due to the temporary header logic. However, note that the optimistic algorithm is more complex than Elevator-First. This is because although Elevator-First requires extra logic in the local and vertical ports, the presence of a temporary header actually simplifies the routing logic in the planar input channels. By contrast, the optimistic algorithm uses a more complex routing logic at the East and West ports.

When the network size increases, all the algorithms grow in size, as the destination and current addresses are getting larger. However, what is interesting to see is that the size of Elevator-

Table 3.1: Hardware synthesis results

Size (x,y,z)	Elevator-First			MD-based			Optimistic		
	Area (um^2)	Power (uW)	Max. Freq (MHz)	Area (um^2)	Power (uW)	Max. Freq (MHz)	Area (um^2)	Power (uW)	Max. Freq (MHz)
4x4x4	975	299	3709	955	277	3709	1115	338	3684
8x8x4	1173	346	3709	1091	312	3709	1230	367	3684
16x16x4	1350	397	3708	1170	335	3701	1324	394	3684
24x24x4	1562	445	3668	1302	363	3701	1450	424	3684

First grows much faster than the proposed approaches, because in addition to the compare logic, the amount of configurable data, which consists of full elevator addresses, also increases. We can see that the size of Elevator-First increases by 60% when going from a 4x4 to a 24x24 layer size, whereas the area increase for the MD-based and optimistic algorithms is of 36.33% and 30% respectively. This shows a better scalability of the proposed approaches to large layer sizes.

3.7.2 Performance evaluation

We are now going to test the proposed algorithms by simulation to understand how they perform with respects to each other. To this end, we use the cycle-accurate parallel network-on-chip simulator presented in Chapter 5. The router microarchitecture, including all the proposed routing algorithms, are modeled in great detail and the network is consistently tested for incorrect behavior or potential deadlocks. The network parameters used for simulations are summarized in Table 3.2. TSV density corresponds to the proportion of elevators in each layer. All layers are supposed to have the same density, however, the placement of TSVs is different form one layer to the other. TSVs are placed randomly at each iteration.

The performance metric we consider is the average packet latency, which is the average time elapsed between the queuing of a packet in the network interface, and the reception of its tail flit at the destination network interface. Simulations are performed on two network sizes: a 128-node network with two 8x8 layers, and a 256-node network with four 8x8 layers. The goal is to evaluate the impact of the layer count on various algorithms. We present the results in Fig. 3.4 and Fig. 3.5.

We first examine the latency in a network with two layers (Fig. 3.4). The first observation

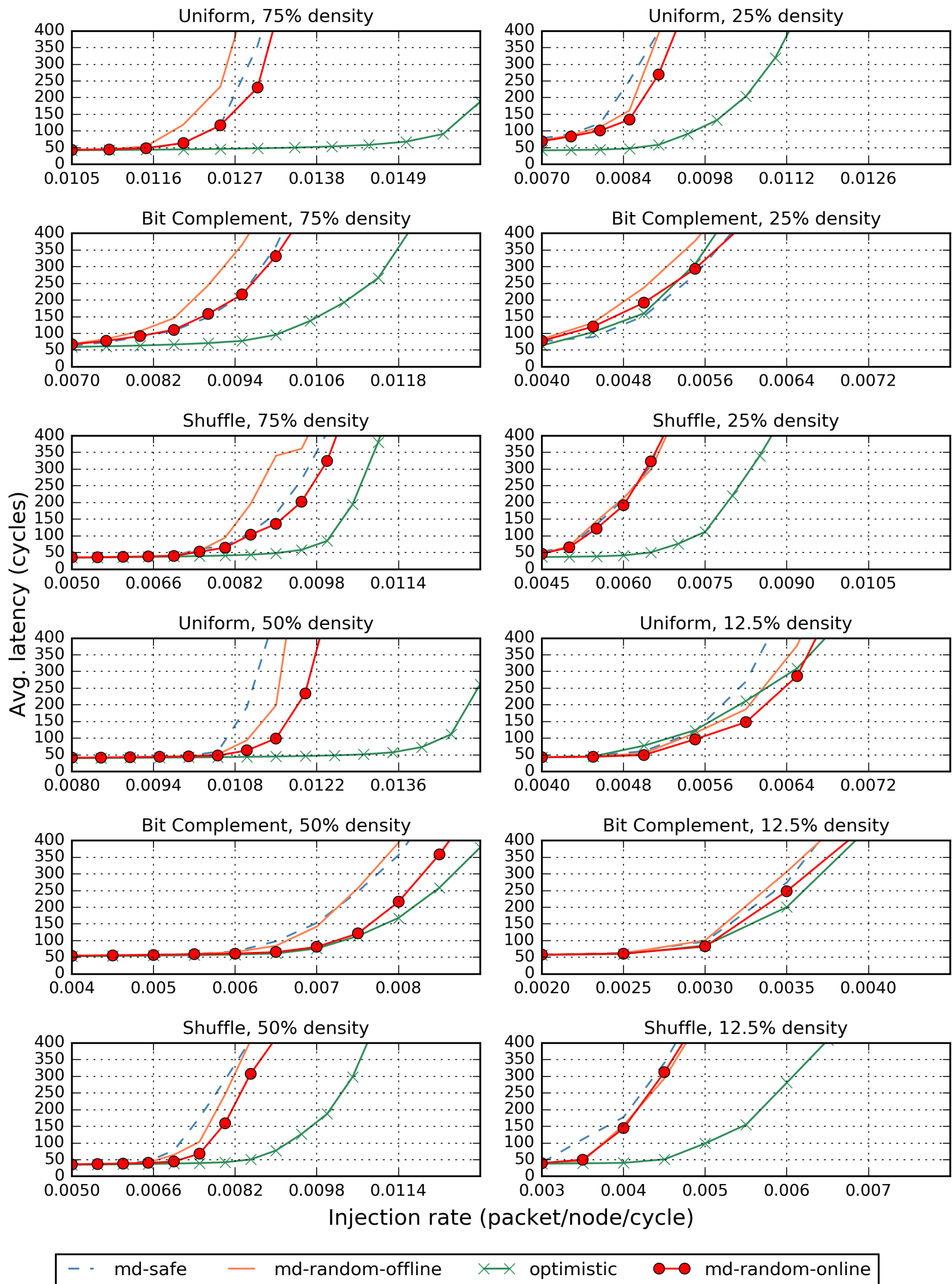

Figure 3.4: Average packet latency for an 8x8x2 NoC.

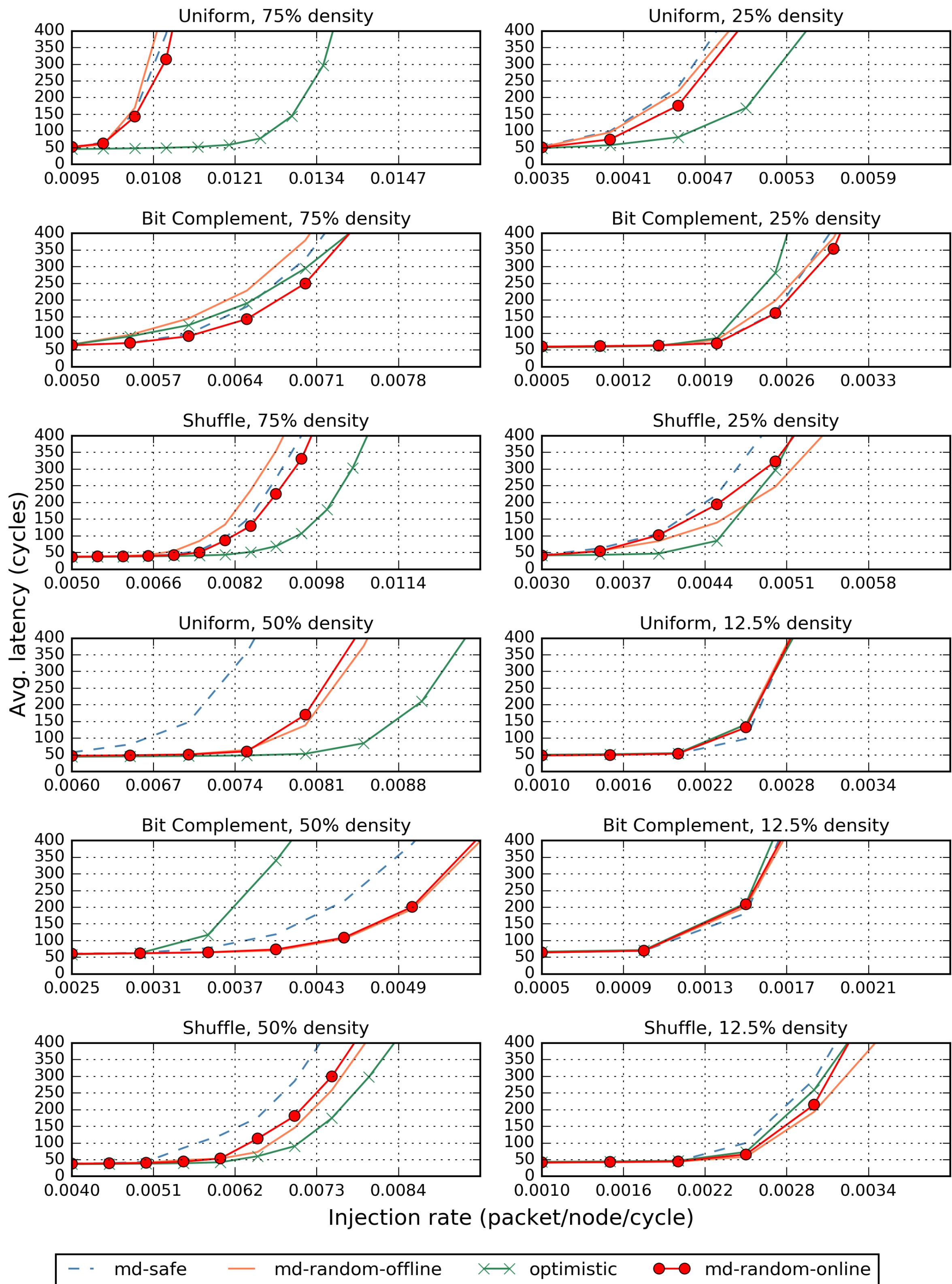

Figure 3.5: Average packet latency for an 8x8x4 NoC.

Table 3.2: Simulation parameters

Parameters	Value(s)
Buffers/VC	4
Flits/Packet	5
Traffic pattern	Uniform, Complement, Shuffle
TSV density	75%, 50%, 25%, 12.5%
Sim. duration	100000 cycles
Iterations	50
2D Routing	XY

that can be made is that the optimistic algorithm greatly outperforms the other approaches in most scenarios. The only cases where it underperforms is when the number of available elevators is very small, which reduces the chances of finding an elevator on the way to the destination. The effect of low TSV density on the optimistic algorithm is even more severe under complement traffic, in which all the nodes send the packets to the other layer. In shuffle traffic, where many nodes communicate within the same layer, we see that the optimistic algorithm maintains a better performance than other approaches even if only 12.5% of the nodes are vertically connected. If the system is designed in such a way that vertical communications are minimal, the optimistic routing algorithm is clearly the best option to adopt. Among the MD-based approaches, we can see that md-random-online consistently delivers the best performance. This was expected because selection between several nearest elevators offline is performed in a fully randomized manner, whereas md-safe and md-random-offline both pose some deterministic constraints on the selection.

We now consider the performance under a 4 layer network (Fig. 3.5). While the optimistic algorithm still performs better under uniform and shuffle traffics, it now yields very poor performance under complement traffic (heavy inter-layer communication), even under high TSV densities. This is due to the fact that when packets fail to find an optimal elevator in the first layer, they will also be penalized in subsequent layers as well. In other words, even if packets are likely to find an elevator on their path to the destination in their original layer, they are less likely to also find one in every layer along the path. This suggests that a hybrid solution, in which optimistic routing is performed only one layer away from the destination layer, can be a better compromise in scenarios where nodes often need to communicate with farther layers.

3.8 Conclusions

Targeting 3D-NoCs with partial vertical connections, we have presented two families of solutions for selecting among several available TSVs in the network.

The first type of selection is based on the Manhattan-Distance. Although already adopted in the literature, we have solved many concerns to which no satisfactory and general solutions were provided.

First, we have provided a universal proof that shows that selection algorithms based on the Manhattan distance can never lead to dead-ends. This is an important proof, as it removes the need to explicitly address livelocks and dead-ends at runtime, which would incur unnecessary hardware overhead.

Second, we have shown that using partial information about the location of elevators, which is necessary to guarantee scalability, is not inherently deadlock-free. We have subsequently presented three selection strategies that alleviate this issue. MD-safe sets the elevator location bits in such a way that routers along any given path have the same vision of the nearest elevator's location. The algorithm is simple, but fails to achieve any sort of load balancing among TSVs. MD-random-offline selects among the nearest elevators randomly so as to better balance the load. However, it gives priority to elevators in a given set of directions depending on the routing algorithm in use, such that deadlock situations cannot be reached. MD-safe and MD-random-offline solve deadlocks fully offline and require no specific modification to the hardware, making them compatible with any planar routing algorithm. To provide an even better load balancing, we have proposed MD-random-online. It selects an elevator offline in a fully randomized manner, but the routing algorithm is modified to ensure deadlock-freedom online. We have shown that this algorithm performs better than the two offline approaches, but its implementation is heavily dependent on the routing algorithm.

The second type of selection that we have introduced improves upon the existing solutions by taking the destination into account. Although it uses a slightly more complex routing logic than the MD-based algorithms, it dramatically improves performance in various situations. All the algorithms that we proposed require 8 bits per router, regardless of the network size, making them highly scalable, as we have shown through the hardware synthesis results.

Chapter 4

The First-Last routing algorithm: A cost-effective alternative to Elevator-first

4.1 Introduction

Perhaps the most challenging aspect in designing routing algorithms for partially connected topologies is ensuring correct operation (Deadlock-freedom, livelock-freedom, connectivity) at a reasonable cost, without heavily limiting the flexibility of the algorithm and the number of fault scenarios (random topologies) it can support. More specifically, as mentioned in the first chapter of this thesis, deadlock-avoidance often requires adding a certain number of Virtual Channels (VCs) in each router, consisting of disjoint flit FIFOs used to separate different flows. As these FIFOs occupy the largest part of a NoC router's area Bahmani et al. [2012], an algorithm that can operate using a small number of VCs is strongly desirable.

While several algorithms requiring no or few virtual channels have been recently proposed Lee and Choi [2013], Salamat et al. [2015], they often follow specific routing rules that pose restrictions on the location and the selection of vertical links, hindering both reliability and performance. A routing algorithm capable of relaxing these restrictions while keeping the implementation cost to a minimum is yet to be introduced.

In this thesis, we address this challenge by introducing a lightweight, adaptive and highly resilient routing algorithm targeting partially vertically connected 3D-NoCs named "First-Last". Our algorithm requires a very low number of virtual channels to achieve deadlock-freedom (2

VCs in the East and North directions and 1 VC in all other directions), and guarantees full connectivity as long as one TSV pillar is available in the network, regardless of its position.

At the time of this writing, ”First-Last” is the only algorithm capable of offering this guarantee using such a low number of virtual channels. Moreover, because it is constructed based on the methodology introduced in Chapter 2, our algorithm is capable of taking full advantage of the available virtual channel buffers for throughput enhancement.

4.2 State-of-the-art in 3D Routing

In the context of 3D-NoCs, several routing algorithms have been proposed. From simple deterministic algorithms such as ZXY, to fully adaptive algorithms such as 3D-FAR Ebrahimi et al. [2013b] and DyXYZ Bahmani et al. [2012], 3D-FT was introduced in Ebrahimi et al. [2013b], which is capable of tolerating the absence of vertical or horizontal links. However, like 3D-FAR, it requires a very large number of virtual channels (2, 2 and 4 along the Z, X and Y dimensions, respectively). In Pasricha and Zou [2011], the authors extend the turn model for 2D meshes Glass and Ni [1992] to the third dimension and propose an algorithm that tolerates faults by replicating each packet and sending it in two different virtual networks, one using the 3D negative-first algorithm and the other using the 3D positive-first algorithm. AFRA Akbari et al. [2012] is another algorithm that can tolerate a certain number of faulty vertical links in fully connected NoCs.

Only a few proposals have been made in the context of partially vertically connected 3D-NoCs. In Dubois et al. [2013], the authors propose to use any deterministic deadlock-free 2D mesh routing algorithm to deliver a packet to an elevator (vertical link), which will be used to deliver the packet to its destination layer, then to continue routing using the planar routing algorithm until the packet reaches its destination. It was proven to be deadlock-free using 2 virtual channels along the X and Y dimensions. This approach, named “Elevator-First”, is appealing because of its simplicity, its support for any layer topology, and because it does not impose any constraints on the position of healthy vertical links. Routing a packet towards an elevator requires the insertion of a temporary header containing the elevator’s address. Addresses of the up and down elevators are stored inside each router Bahmani et al. [2012], requiring an amount

of storage that increases with the network size. In order to reduce the requirements of Elevator-First in terms of virtual channels, authors in Lee and Choi [2013] add certain constraints on the usage of the elevators and show that routing is possible without the use of virtual channels. In Ying et al. [2014], another algorithm that does not require the use of virtual channels is presented, but it requires the presence of one vertical link at the north-east corner.

The ETW (East-then-West) routing algorithm Salamat et al. [2015] aims at reducing the required number of virtual channels of Elevator-First, while maintaining a certain routing flexibility and offering partial adaptiveness to mitigate congestion. ETW uses 1, 2 and 1 virtual channels along the X, Y and Z dimensions respectively. The authors have also proposed some solutions to tolerate runtime failures using the dynamic elevator assignment in Salamat et al. [2016b] or the propagation of TSV status in Salamat et al. [2016a]. Unfortunately, ETW poses some limiting constraints on both the location, and the selection of the elevators. It requires the existence of at least one pillar in the east-most column, and for packets heading south, an elevator located east to the destination must be taken, leading to inefficient routes in some cases. In addition, because the choice of the elevator depends on the destination, 3 elevator addresses need to be stored in each router.

Our algorithm uses the same total number of VCs as ETW Salamat et al. [2016b] but does not require the presence of TSV pillars, i.e. the position of TSVs may differ from one layer to the other. Moreover, unlike ETW, our algorithm makes it possible for packets to reach their destination node regardless of which TSV they decide to use, allowing for much shorter routes than those imposed by ETW.

Finally, the 3D variant of the LBDR (Logic Based Distributed Routing) Flich and Duato [2008] was recently presented in Niazmand et al. [2016]. As is the case of LBDR, LBDR3D supports a variety of partially adaptive routing algorithms and is fully reconfigurable to tolerate faulty horizontal and vertical links. It was proven deadlock- and livelock- free using the same method as Elevator-First and requires the same minimum number of virtual channels to separate between Upward and Downward flows. However, in LBDR3D, only a fixed number of bits are stored within each router to locate healthy elevators.

Like LBDR3D, to keep track of healthy elevators without having to store router addresses, in Section 4.5 we propose a scalable method that uses a fixed number of bits per router (12

bits) to guide packets to the nearest elevator. However, we will show that the method adopted in Niazmand et al. [2016] for assigning elevators to nodes is not inherently deadlock-free. We propose an offline selection algorithm that effectively guarantees deadlock-freedom without changing the hardware. Moreover, we provide an elaborate proof of reachability, that shows that packets always reach an elevator, without needing any extra signals during runtime to prevent packets from looping as in Niazmand et al. [2016].

4.3 The First-Last routing algorithm

4.3.1 General approach

Since the primary goal of this work is to reduce the VC requirements of Elevator-First Dubois et al. [2013], it can be expected that the resulting solution offers a lower throughput than elevator-first, due to the presence of less buffers.

However, we want to compensate for the missing buffers by making a better use of the available ones. In Chapter 2, we have introduced a design methodology that aims at optimizing the utilization of VC buffers. We rely on this methodology to construct our new routing algorithm for 3D NoCs.

4.3.1.1 Identifying cycle-free routing classes

As explained in Chapter 2, a routing class is a set of **physical** channels (output ports) that the packet can use to make progress towards its destination. The class number is stored in the packet header and is used by the route computation logic to determine the set of candidate directions that can be taken at every hop.

Moving from one routing class to the next is simply performed by updating the class number in the packet, as will be shown in Section 4.5.

The first-last routing algorithm defines 3 routing classes as shown in Fig. 4.1. The first and third classes (C_0, C_2) only include the X+ and Y+ physical channels. The second class (C_1) includes all the remaining directions (X-, Y-, Z+, Z-). Note that it is not possible to form a cycle within any of these classes, as none of them spans two full dimensions Ebrahimi and

Daneshtalab [2017].

Because the X+ and Y+ physical channels are shared between two classes, two virtual channels are required in these directions. The two virtual channels in the X+ direction are noted (X0+, X1+), whereas the virtual channels in the Y+ direction are noted (Y0+, Y1+). The other directions, which are only used by C_1, do not necessitate additional virtual channels.

Packets may traverse the classes only in increasing order ($C_0 \rightarrow C_1 \rightarrow C_2$). The algorithm is named First-Last, because the physical channels that are used first (Positive channels) are also used last.

The routing algorithm can be simply described as follows: If a packet has reached the destination tier, i.e. the tier where the destination node is located, then route it towards the destination node using the negative directions first (C_1 then C_2). Otherwise, route the packet towards an elevator using the positive directions first (C_0 then C_1) and use the channels of C_1 to elevate the packet to the destination layer. Whenever a packet is headed East-North or West-South, routing can be performed adaptively and the least congested route is selected. Selection can be performed using one of the techniques introduced in Appendix A.

Details about a possible implementation are provided in Section 4.5.

4.3.1.2 Step 2: Assigning virtual channels

To keep the solution as simple as possible, we only make use of a subset of the mechanisms presented in Chapter 2. Instead of using channel tags and dynamic masking (see Chapter 2), we opt here for a fixed assignment of virtual channels to packet classes, i.e. static masking.

The (X-, Y-, Z+, Z-) channels are only used by packets of C_1 and the corresponding single virtual channel in each of these directions is fully dedicated to packets of C_1.

Packets of C_0 are only allowed to use virtual channels (X0+, Y0+), whereas packets of C_2 are allowed to use all the virtual channels associated with the physical channels of C_2 (X0+, X1+, Y0+, Y1+). In other words, virtual channels (X1+, Y1+) are only used by C_2 packets, whereas virtual channels (X0+, Y0+) are shared between C_0 and C_2 packets. This is enough to satisfy the sufficient condition of deadlock-freedom presented in Chapter 2.

Note that unlike the traditional approaches that prohibit cycles by dedicating each virtual channel to a single class of packets, here we allow extra freedom on the acquisition of virtual

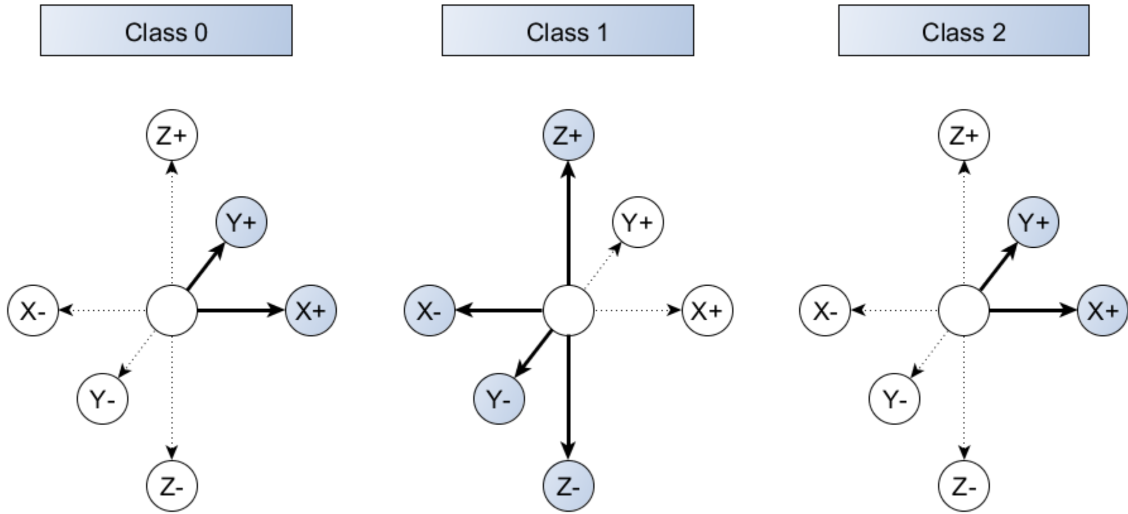

Figure 4.1: The 3 routing classes of First-Last.

channels for packets of C_2. This implies that packets that have reached their destination layers are allowed to occupy any VC in the network to reach their destination node.

4.3.2 Enhanced-First-Last: Boosting network performance and resilience with vertical VCs

With a reduced number of vertical channels, elevator nodes are very likely to turn into hotspots as several flows may need to share a single TSV.

A simple way to mitigate this issue is to add a virtual channel along the Z dimension to help reduce the pressure on vertically connected nodes. This VC can be used by all C_1 packets without any restrictions.

Because the VC is added in the vertical ports, only 3D routers need to be modified. This means that although 3D routers now include as many VCs as Elevator-First Bahmani et al. [2012], 2D routers still include fewer VCs, resulting in a lower overall cost than Elevator-First, especially for designs with a low TSV density. Under heavy inter-layer communication, VCs along the Z dimension will have a much higher impact on performance than VCs in the planar ports, making them well worth the extra cost.

Furthermore, in addition to throughput enhancement, this extra VC, if wisely used, can also

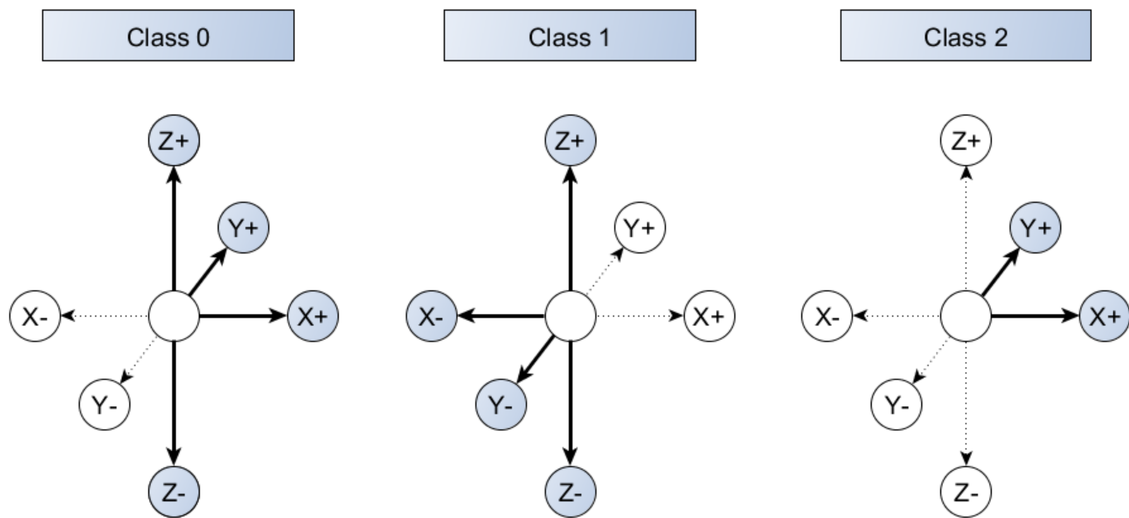

Figure 4.2: The routing classes of Enhanced-First-Last.

contribute to the resilience of the routing algorithm.

Consider the virtual network definition presented in Fig 4.2. Compared to the original First-Last algorithm, C_0 now also includes the vertical dimension. This means that packets of C_0 can now reach other layers without having to transit to C_1, provided that an elevator could be reached using only the positive directions. In this case, after reaching the next layer, any elevator can be taken as packets can still use C_0 and C_1.

Fig. 4.3 illustrates an example network in which layer 0 cannot reach layer 2 using the original First-Last algorithm. However, Enhanced-First-Last is capable of partially connecting layer 0 to layer 2.

Here again, we allow C_1 packets to use both vertical VCs (Z0+, Z1+, Z0-, Z1-) while ensuring they can always escape to (Z1+, Z1-), whereas packets of C_0 may only ever use one of the two VCs (Z0+, Z0-).

4.3.3 Flow control

In the absence of channel tags, we also present a simplified version of the flow control mechanism described in Section 2.6.

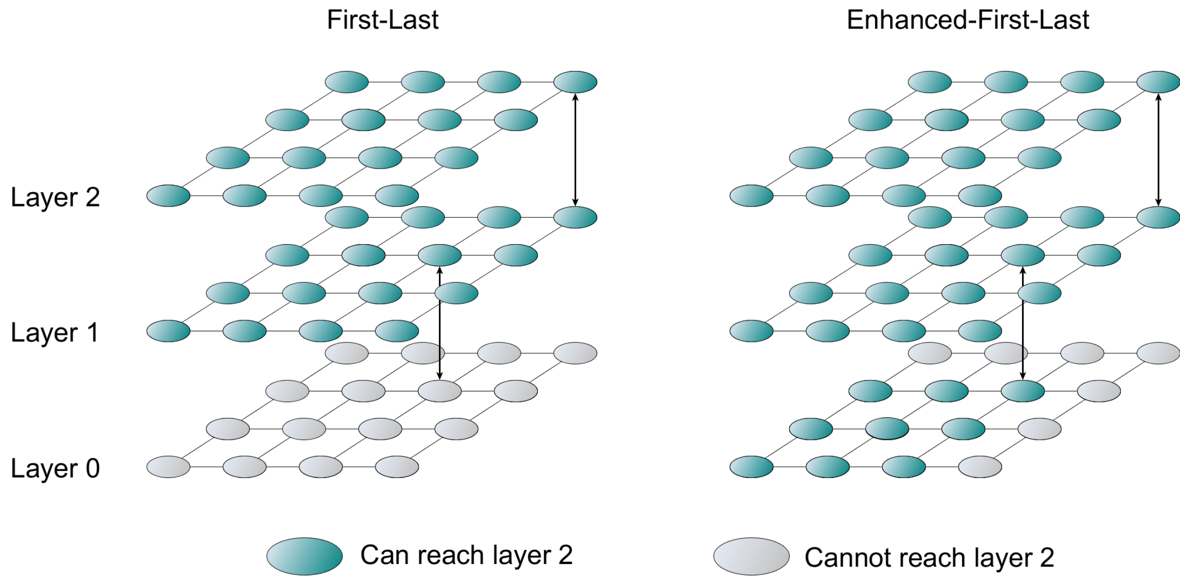

Figure 4.3: An example in which Enhanced-First-Last improves the connectivity of the network.

4.3.3.1 First-Last

Packets that can only request one VC at all times are allowed to acquire the VC of their selected port even if the VC is not empty. This applies to packets of class 0 and class 1.

Packets of class 2 need to be able to escape to their dedicated channels (X1+, Y1+) at all times when they are occupying channels (X0+, Y0+). Therefore, it is sufficient to ensure that packets of class 2 only acquire channels (X0+, Y0+) when they are empty, so that packets of class 2 never follow packets of class 0 within the same VC. However, packets of class 2 are allowed to acquire VCs (X1+, Y1+) even when these are not empty.

4.3.3.2 Enhanced-First-Last

The same reasoning applies in the case of the Enhanced-First-Last algorithm. In addition to the restriction on class 2 packets, packets of class 1 should not be allocated channels (Z0+, Z0-) unless they are empty, so that they do not follow packets of class 0 that may be occupying these channels.

4.4 Deadlock-freedom, livelock-freedom, and connectivity

4.4.1 Deadlock-freedom

The First-Last routing algorithm is deadlock-free by construction, as it is based on the methodology presented in Chapter 2. In what follows, we show that every packet class in the network satisfies our deadlock-freedom condition (see Chapter 2) at all times.

Class 0: Because it is the lowest class in the network, packets of C_0 are always waiting for other packets of C_0 or higher classes, regardless of the virtual channel. Therefore, packets of C_0 always satisfy the deadlock-freedom condition.

Class 1: The virtual channels used by C_1 (X-, Y-, Z+, Z-) are only used by packets of C_1, which means packets of C_1 can only wait for other packets of C_1. This satisfies the deadlock-freedom condition.

Class 2: Packets of C_2 can wait for virtual channels (X0+, X1+, Y0+, Y1+). Although (X0+, Y0+) may be occupied by packets of C_0, channels (X1+, Y1+) can only be occupied by C_2 packets as per the VC assignment described in section 4.3.1.2. Packets of C_2 are therefore always waiting for other packets of C_2 occupying channels (X1+, Y1+), which conforms to the deadlock-freedom condition.

4.4.2 Livelock-freedom

Since the individual packet classes do not allow packets to loop indefinitely, and because they are traversed in an increasing order, the algorithm is also livelock-free. In the worst case, a packet reaches the north-east corner in C_0 at the source layer (top or bottom layer in the case of Enhanced-First-Last), then the west-south corner of either the top or the bottom layer in C_1 then end up in the north-east corner of the same layer in C_2.

4.4.3 Condition of connectivity

The network is connected if the routing algorithm is able to provide a path for all source-destination pairs. We will identify the condition that must be met by the TSV placement strategy in order for the network to be connected using the First-Last algorithm. In other words, we will

identify the set of topologies that are supported by the First-Last routing algorithm.

First, we know that routing from any source to any elevator in the same layer, which is done using C_0 and C_1, is always possible. Similarly, routing from any elevator to any destination on the same layer is possible using C_1 and C_2. This means that for the network to be connected, it is enough to ensure that every layer in the network can reach every other layer.

Although Enhanced-First-Last makes it possible to traverse layers in C_0 for some packets, in the general case, packets may need to go to C_1. So to guarantee connectivity, we consider the worst case, wherein packets need to move to C_1 to reach other layers.

Let us consider a network consisting of L layers, with U_l and D_l being the set of upward elevators and the set of downward elevators of layer l, respectively. The condition can then be expressed as follows:

The network is connected if and only if

$\forall l \in]0, L-1[$,

$\exists u_-(x_-, y_-) \in U_{l-1}$, such that $\exists u(x, y) \in U_l$ with $x \leq x_-, y \leq y_-$

and

$\exists d_+(x_+, y_+) \in D_{l+1}$, such that $\exists d(x, y) \in D_l$ with $x \leq x_+, y \leq y_+$.

That is, each layer must be able to forward packets coming from a previous layer through a certain elevator E_{in}, to the next layer through an elevator that is reachable using only the negative channels (C_1), i.e. that is located south-west to the incoming elevator E_{in}.

Fig. 4.4 illustrates a few examples of connected networks. In the first example (Fig. 4.4 (a)), two TSVs are used to connect each consecutive layers. The network is connected because the bottom layer can reach the top layer through elevators $1 \rightarrow 2 \rightarrow 3$, and the top layer can reach the bottom layer through elevators $A \rightarrow B \rightarrow C$.

The second example (Fig. 4.4 (b)) shows that it is possible to connect the network using only one TSV "pillar". Note that the network is connected regardless of the position of the pillar.

For the remainder of this chapter, we will consider topologies consisting only of TSV pillars, i.e. an upward (downward) elevator can reach any upper (lower) layer without changing the (X,Y) coordinates. This architectural simplification has been adopted by other algorithms such as Salamat et al. [2016b] and Ying et al. [2014], although unlike these solutions,

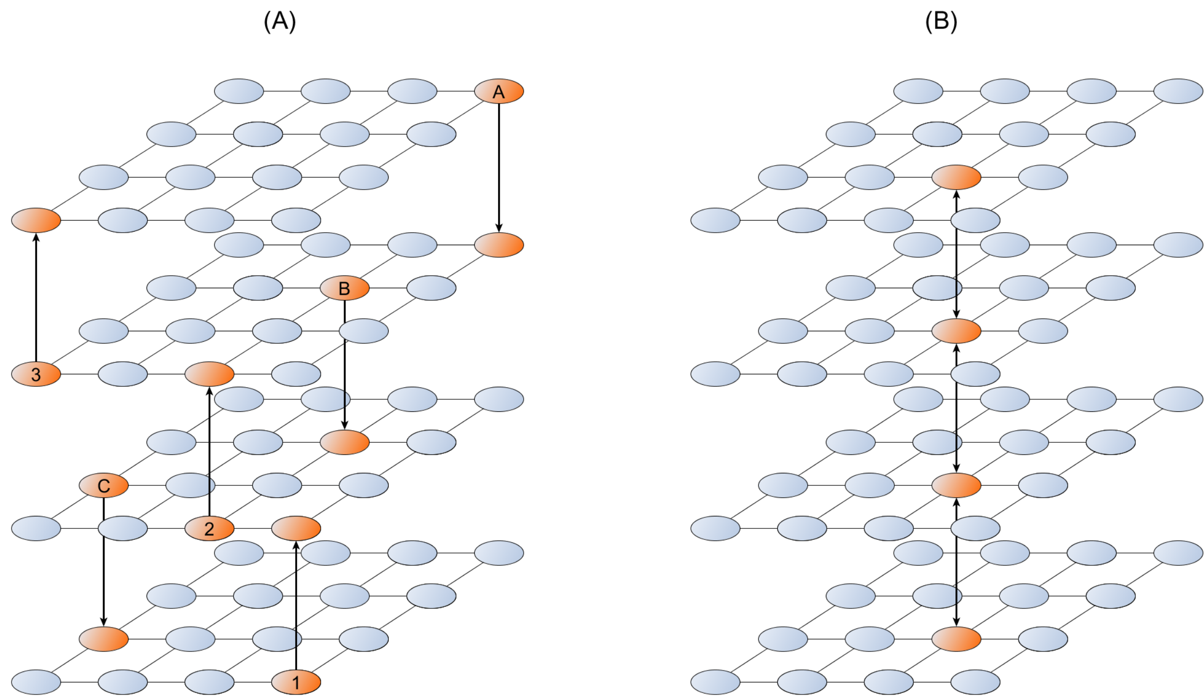

Figure 4.4: Examples of connected networks using First-Last.

the First-Last algorithm does not impose any restriction on the position of the pillars, which may be placed arbitrarily.

4.5 Hardware implementation details

In Chapter 3, we have presented a generic framework for TSV assignment and selection that can be applied to any routing solution targeting TSV-based 3D NoCs. Due to its high efficiency in terms of area and scalability, we propose an implementation of the First-Last routing algorithm based on this framework.

The implementation will consist of three parts:

- **Encoding the elevator location:** We will show that the encoding presented in Chapter 3 can be applied to First-Last with no modification.
- **Specializing a generic TSV assignment algorithm to the First-Last algorithm:** One of the elevator assignment algorithms presented in Section 3.4.3 will be used in conjunction with our routing algorithm.

- **Implementing the routing logic**

4.5.1 Scalable TSV assignment

Due to the presence of TSVs as pillars, packets only need to search for an elevator in their source layer. Because packets originate in routing class C_0 and are able to move to C_1, they can reach an elevator located at any position. Therefore, all a router has to store is the location of the upward and downward elevators as two 4-bit vectors named $Elevator_Up$ and $Elevator_Down$, respectively. The 4 bits $\{East, South, West, North\}$ indicate whether the selected elevator is located East, South, West or North, respectively. This encoding is identical to the one described in Chapter 3.

One sensible criterion of selection is the distance of the elevator from the current router. That is, each router forwards a packet towards an elevator located at minimum Manhattan distance (MD). The main idea behind this choice is to make sure packets spend as little time as possible at intermediate layers, and are able to reach their destination layer as quickly as possible. Moreover, in Section 3.4.3, we have presented several options for MD-based elevator assignment. We have established that the proposed MD-based approaches had the advantage of not requiring any involvement from the route computation logic (the hardware) to ensure correct deadlock-free operation.

In this section, we opt for the randomized elevator assignment approach, as it offers the best trade-off between simplicity and performance. In Chapter 3 - Algorithm 8, we have presented a generic randomized assignment algorithm that can be used for any planar routing algorithm. As an input, the algorithm takes a description of the routing algorithm as its Last Direction Set (see Definition 3.1) as its input and assigns an elevator to each node. That is, all we need to do is define the Last Direction Set of the First-Last routing algorithm in order to be able to use this TSV assignment algorithm.

When seeking an elevator, packets are routed following routing classes C_0 then C_1. That is, the planar routing algorithm can be described as the following list of sets:

$A = [\{East, North\}, \{West, South\}]$.

Which means that the Last Direction Set for the First-Last algorithm is

$L = \{West, South\}$.

It is worth reminding that the correctness of the assignment algorithm, as well as the reachability of the elevators assigned using this algorithm were formally proven in Section 3.5.

4.5.2 Route computation logic

The route computation logic is presented in Algorithm 11. As an input, the algorithm takes a bit vector $Dest$ describing the position of the destination (Up, Down, East, West, South, North), and the current class number c_{in}. The $Dest$ vector is obtained by comparing the position of the current router to that of the destination indicated in the packet header. The packet header also stores the class number c_{in}, as mentioned in section 4.3. The algorithm outputs the set of possible output ports, and possibly a new class number, if moving to the next class is necessary.

Algorithm 11 Route computation logic

Input:
  $Dest$: Destination position bits
  c_in: Current routing class
Output:
  R : Set of possible output channels
  c_out : Output routing class

```
1:  if Dest.Up or Dest.Down then
2:      if Dest.Up then
3:          Elevator ← Elevator_Up
4:      else
5:          Elevator ← Elevator_Down
6:      end if
7:      if Elevator = {0,0,0,0} then                    ▷ Self is elevator
8:          c_out ← c_in
9:          if not using Enhanced-First-Last then
10:             c_out ← 1                           ▷ Must move to C_1 to go vertical
11:         end if
12:         if Dest.Up then
13:             R ← {Z+}
14:         else
15:             R ← {Z−}
16:         end if
17:     else
18:         (R, c_out) ← Positive_First(Elevator, c_in)
19:     end if
20: else
21:     (R, c_out) ← Positive_Last(Dest, c_in)
22: end if
```

If the destination is on the same layer as the current router (Algorithm 11 - Line 20), routing

is performed following the $Positive_Last$ routing algorithm presented in Algorithm 12. Otherwise, the algorithm first computes the position of the appropriate elevator according to the destination layer (up or down) (Algorithm 11 - Lines 2 to 6).

Algorithm 12 The Positive_Last routing function

Input:
  $Dest$: Destination position bits
  c_in: Current routing class
Output:
  R : Set of possible output channels
  c_out : Output routing class

```
 1: c_out ← c_in
 2: if Dest.West and Dest.South then
 3:     R ← {X−, Y−}
 4: else if Dest.West then
 5:     R ← {X−}
 6: else if Dest.South then
 7:     R ← {Y−}
 8: else
 9:     c_out ← 2
10:     if Dest.East and Dest.North then
11:         R ← {X+, Y+}
12:     else if Dest.East then
13:         R ← {X+}
14:     else if Dest.North then
15:         R ← {Y+}
16:     else
17:         R ← {L}                                  ▷ Local port
18:     end if
19: end if
```

If the current router is an elevator (Algorithm 11 - Line 7), then the packet is forwarded appropriately either to the up or down port. With Enhanced-First-Last, moving up and down is possible in both C_0 and C_1, so changing the class is not necessary. However, in the First-Last algorithm (Algorithm 11 - Line 9), moving vertically can only be done in C_1, so the class number must be updated.

If the current router is not an elevator, the packet is routed towards the selected elevator following the $Positive_First$ routing algorithm presented in Algorithm 13.

Algorithm 13 The Positive_First routing function

Input:

$Dest$: Destination position bits
c_in : Current routing class

Output:

R : Set of possible output channels
c_out : Output routing class

```
1: c_out ← c_in
2: if Dest.East and Dest.North then
3:     R ← {X+, Y+}
4: else if Dest.East then
5:     R ← {X+}
6: else if Dest.North then
7:     R ← {Y+}
8: else
9:     c_out ← 1
10:    if Dest.West and Dest.South then
11:        R ← {X−, Y−}
12:    else if Dest.West then
13:        R ← {X−}
14:    else if Dest.South then
15:        R ← {Y−}
16:    else
17:        R ← ∅
18:    end if
19: end if
```

4.6 Experimental results

4.6.1 Hardware synthesis

To analyze the hardware cost of the proposed solutions, we have extended the Netmaker library Mullins [2009] to support 3D router architectures. Netmaker is a library of parameterizable and synthesizable NoC routers written in SystemVerilog. This library was used to implement the First-Last and Enhanced-Fist-Last routing algorithms described in this chapter, as well as the Elevator-First algorithm described in Bahmani et al. [2012].

All routers include 4-flit deep virtual channel FIFOs and perform virtual channel allocation followed by switch allocation (2 cycles). To evaluate the area overhead, we synthesize the Elevator-First, First-Last, and Enhanced-First-Last routers using Synopsys Design Compiler. The designs were setup to work with an operating frequency of 1GHz, a power supply of 1V, and a NanGate Open Cell $45nm$ Library Nangate [2017]. The resulting area and power estimates for each router are summarized in Table 4.1. Three types of routers were considered: 5 port 2D routers, 6 port 3D routers with one vertical connection, and 7 port 3D routers with both vertical connections.

First, we compare the area overhead for all 5 ports routers. The area for First-Last and Enhanced-First-Last is the same because in a 2D router, both algorithms require the same number of virtual channels and use the same routing logic. On the other hand, although Elevator-First uses a simple XY routing function for routing, it includes one extra virtual channel in the South and West directions, which increases the area overhead by 15.2%.

A similar comparison can be done for the 6 ports routers. In this case, Elevator-First has two more virtual channels than First-Last and one more virtual channel than Enhanced-First-Last. Those additional virtual channels can explain the area overhead observed for Elevator-First, which is of approximately 12% compared with First-Last, and 5.1% with respects to Enhanced-First-Last. However, when comparing Elevator-First and Enhanced-First-Last in the case of 7-port routers, where the total number of virtual channels is the same, we note a slightly larger area (1%) when using Enhanced-First-Last, due to its more complex routing logic when compared to the simple XY algorithm used by Elevator-First.

In sum, the Enhanced-First-Last algorithm can be considered an appealing alternative to

Elevator-First, as it not only reduces the area and power, especially for designs that use more 5-port and 6-port routers than 7-port routers, but also is capable of attaining higher levels of performance, as will be shown in the rest of this section.

Table 4.1: Hardware synthesis results

Type	**Elevator-First**		**First-Last**		**Enhanced-First-Last**	
# Ports	Area (μm^2)	Power (mW)	Area (μm^2)	Power (mW)	Area (μm^2)	Power (mW)
5 ports	36185	12.0	31407	10.3	31407	10.3
6 ports	44300	14.7	39526	12.9	42132	13.8
7 ports	54080	17.8	49197	15.9	54593	18.0

4.6.2 Performance evaluation

Finally, we use the NoC simulator described in Chapter 5 to compare the performance of our algorithm to that of Elevator-First. The simulation parameters are summarized in table 4.2. Here, TSVs were placed randomly as pillars, i.e., the upward and downward TSVs are placed at the same X,Y coordinates across all layers.

Table 4.2: Simulation parameters

Parameters	**Value(s)**
Buffers/VC	4
Flits/Packet	5
Traffic pattern	Uniform, Complement, Shuffle
TSV density	75%, 50%, 25%, 12.5%
Sim. duration	100000 cycles
Iterations	50

We present the average packet latency obtained for an 8x8x4 mesh network in fig. 4.5. The first observation that we can make about the results is that the Enhanced-First-Last routing algorithm outperforms Elevator-First significantly in all the tested scenarios, even though the total number of VCs in the network is lower. This is due to a combination of a higher availability of VCs for packets in their destination layer (class 2), and an extra VC to use for packets that are moving vertically to reach other layers. Under low TSV densities, this extra VC greatly reduces the pressure on vertical ports.

As can be expected, the First-Last algorithm has a higher latency than Elevator-First, as the number of planar virtual channels is lower. However, it can be noticed that the difference is less significant under shuffle traffic, in which many nodes communicate within the same tier. In the presence of a sufficient number of TSVs (75%), the First-Last algorithm is even able to outperform Elevator-First. This can be explained by the fact that packets that are in their destination layer use the routing classes 1 then 2. Once a packet moves to class 2, it is able to freely acquire both VCs in the East and North directions, allowing them to reach their destination much faster than in Elevator-First, where each packet can only acquire one same VC from source to destination. Note however that even with First-last, packets that are in a different layer than that of their destination can only use one VC.

In sum, although the main goal of First-Last is to save cost, it can still outperform Elevator-First if inter-layer communication is kept to a minimum.

4.7 Conclusion

We have presented a novel algorithm targeting partially vertically connected 3D-NoCs named "First-Last" that guarantees packet delivery as long as one TSV pillar is available anywhere in the network. Although our algorithm uses the same number of virtual channels as other low-cost algorithms, it is in the way these virtual channels are distributed that lies all of its strength. What we have demonstrated in this chapter is that by carefully placing these virtual channels and appropriately defining the routing rules, it was possible to eliminate all the restrictions that other solutions impose on both the supported fault patterns and the runtime elevator selection, which can negatively impact the resilience and performance of the NoC. Moreover, we have shown, through Enhanced-First-Last, that by adding one VC in the vertical dimension, it was possible to dramatically boost the NoC's performance in presence of a few vertical connections, while still ensuring a lower implementation cost than state-of-the-art algorithms.

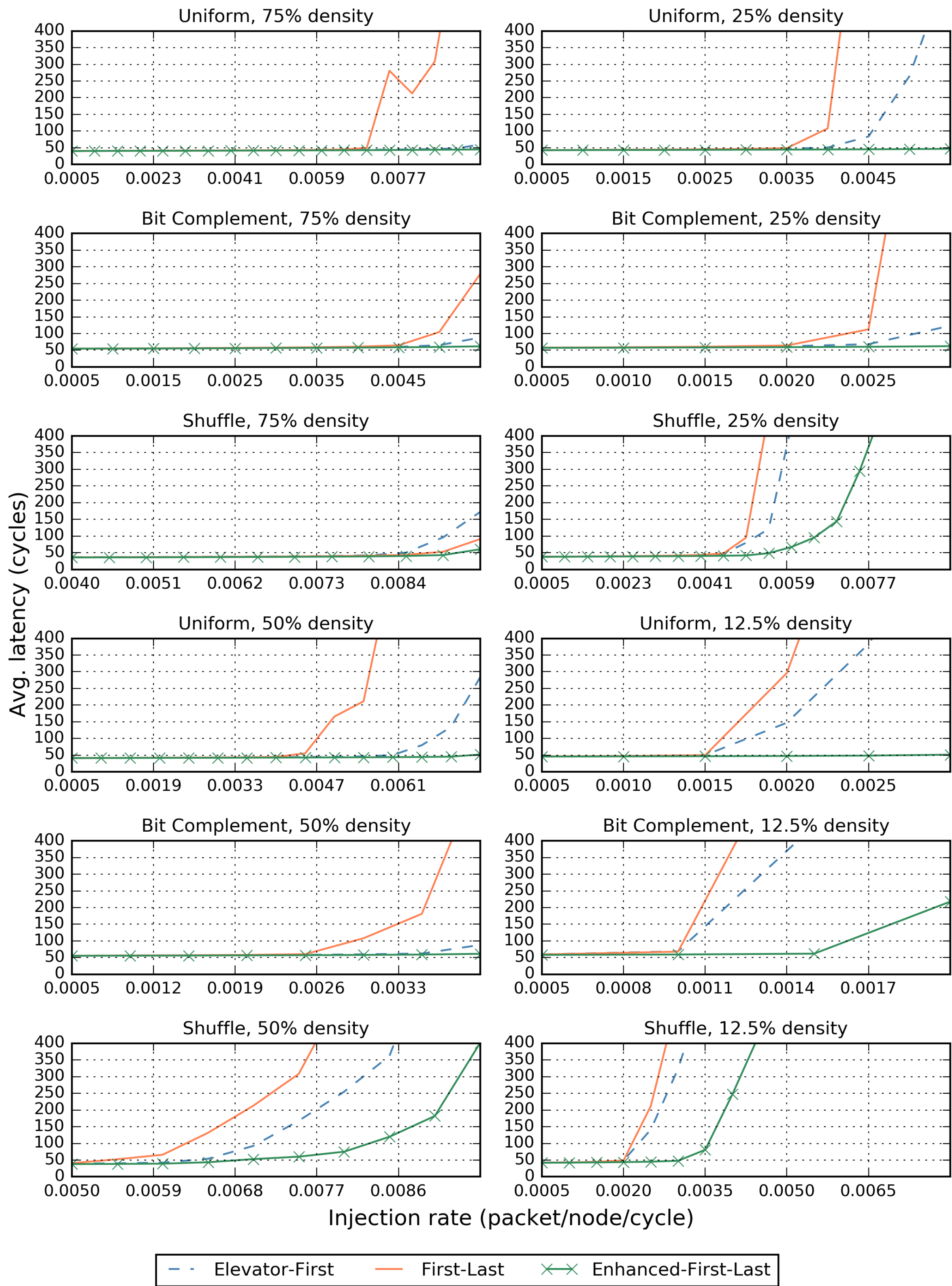

Figure 4.5: Average packet latency.

Part IV

ULTRA-FAST GPU-BASED PARALLEL CYCLE-ACCURATE SIMULATION OF NETWORKS-ON-CHIPS

Chapter 5

Highly Parallelizable Cycle-Accurate Network-on-Chip Simulation

5.1 Introduction

Throughout this thesis, we have presented several new Network-on-Chip designs that need to be tested and validated prior to hardware implementation.

At an early stage, novel NoC designs are usually evaluated and validated by means of cycle-accurate simulation. While simulating NoCs at the RTL (Register Transfer Level) can produce very accurate results, popular cycle-accurate simulators such as Booksim Jiang et al. [2013] are often preferred due to their shorter simulation run times and easier programmability. However, with the tremendous increase in the number of processing nodes in modern and emerging chips, new proposals will have to be validated against increasingly large NoCs, which can take impractical simulation times, even when simulating at a high level of abstraction.

Parallelization is one obvious solution to the issue and unsurprisingly, several works have attempted to take advantage of modern many-core processors to speed up NoC simulations. However, due to the significant synchronization overhead in CPU-based multithreading, such solutions often need to sacrifice cycle-accuracy to obtain decent simulation performance Ren et al. [2014]. More importantly, achieving high speedups usually requires CPUs with a very high core count, which can be rather costly.

By contrast, GPGPUs (General-Purpose Graphics Processing Units) have been widely adopted

in the HPC (High-Performance Computing) field as a more cost-effective parallel processing solution. GPUs have been gaining in popularity in various domains, including cycle-accurate simulation Zolghadr et al. [2011], Pinto et al. [2011]. Unfortunately, prior attempts at simulating NoCs on GPU Zolghadr et al. [2011] assume an overly simplified NoC architecture and propose a static parallelization method that is very hard to extend and generalize. To the best of our knowledge, no general and scalable method for performing realistic NoC simulations on GPU has yet been proposed.

In this final chapter of this thesis, we introduce the first detailed and modular NoC simulator design targeting GPU platforms. First, a flexible task decomposition approach, specifically geared towards high parallelization is proposed. Our approach makes it easy to adapt the granularity of parallelism to match the capabilities of the host GPU. Second, all the GPU-specific implementation issues are addressed and several optimizations are proposed. Our design is evaluated through a reference implementation which is tested on an NVidia GeForce GTX980Ti graphics card and shown to speed up simulations by over 250 times.

5.2 State-of-the-art

The vast majority of NoC simulators available today are single-threaded. General-purpose NoC simulators such as Booksim Jiang et al. [2013], Noxim Catania et al. [2015], and Garnet Agarwal et al. [2009], are very widely used in NoC research. In spite of being very good tools, all of these simulators are overly slow at performing long running simulations of large NoCs, comprising over 256 routers.

Several attempts have been made to build scalable multi-threaded NoC simulators that leverage modern multicore machines. A popular example is Hornet Ren et al. [2014], which distributes simulation tasks evenly among CPU threads to speed up simulations. To avoid per cycle thread synchronization, which is the major showstopper when it comes to CPU-based parallelism, Hornet performs synchronization periodically, which results in a certain loss in timing accuracy. Authors in Eggenberger and Radetzki [2013] propose a task distribution based on thread pools, in which idle threads execute the next available task in order to better balance the work load. However, because threads may access different locations in memory throughout

the simulation, it exhibits less cache affinity than the approach used in Hornet. More recently, in Eggenberger et al. [2016], the authors have proposed a new simulation method that combines uniform task distribution, and a GALS (Globally Asynchronous Locally Synchronous) approach to avoid frequent synchronization.

A few works have attempted to perform NoC simulations on GPU. Authors in Pinto et al. [2011] managed to simulate a thousand core system, including a NoC on an NVidia GeForce GTX 480 GPU. However, since the focus is not on the network level, this solution uses a simple NoC model, which cannot be used for detailed network-only simulation.

The closest related work that we are aware of is Zolghadr et al. [2011]. In this solution, every GPU thread executes the tasks of one bidirectional link in a router and speedups of up to 17x were reported. In practice, however, we need to be able to simulate a detailed router architecture, which consists of several modules, including potential extensions by the end user, all of which can be quite complex. Including all of these modules in the actions of input/output links severely limits the level of parallelism, which heavily impacts performance. At the same time, if we were to simulate a very large network, the link count of which cannot be accommodated by the GPU, then the simulation would simply be unfeasible and the core of the simulator would have to be reprogrammed.

We provide a generic task definition that does not suffer from these limitations and an implementation that yields much higher speedups than those reported in related works.

5.3 Generic Task Decomposition

The design of a parallel simulator requires an appropriate decomposition of the simulation into elementary tasks. These tasks are then mapped onto threads for parallel execution. The definition of simulation tasks is crucial as it determines the granularity, and consequently the performance of the parallel simulation. For instance, in Eggenberger and Radetzki [2013], a task is defined as the set of actions performed by a single router. Due to the significant synchronization overhead of CPU-based multithreading, this coarse-grained decomposition is appropriate.

By contrast, in GPU based parallelism it is usually preferable to run as many threads as possible. In Zolghadr et al. [2011], a task corresponds to the actions performed by one output

and one input port of a router. This finer-grained decomposition is more appropriate to the inherently parallel architecture of GPUs. However, in addition to the scalability issues mentioned in section 5.2, this static approach poses many issues in terms of programmability. In effect, in a real router architecture, not all the modules are distributed, and some tasks are centralized. This is the case, for instance, of the VC and Switch arbiters, which are not tied to a specific input/output direction, but need to operate in a centralized manner. Programming such modules can be quite difficult in the model proposed by Zolghadr et al. [2011], which makes the simulator hard to extend. Extensibility is a crucial feature of the design that we propose.

We introduce in this work a new task definition that makes it easy to configure the granularity of parallelism to match the capabilities of a given GPU and, more importantly, does not suffer form the aforementioned programmability and extensibility issues. This section will consist of three parts. First, we provide a definition of modules, which are the basic building blocks of our simulator design. Then, we introduce a new abstraction called module group. Finally, we give our definition of what a single simulation task is in terms of module groups.

5.3.1 Modules

In hardware, every NoC router comprises several ***modules***, each responsible of accomplishing a specific action every cycle. For the sake of illustration, we will assume, throughout the rest of this chapter, a canonical VC (Virtual Channel) router architecture, as it is the one that most simulators adopt as their baseline model Jiang et al. [2013], Agarwal et al. [2009]. However, note that our proposals are general and are not architecture-specific. A VC router typically includes the following modules:

- Input Unit (one per input port): This module reads a flit from the corresponding input port, writes it to the appropriate virtual channel FIFO, and, if it is a head flit, performs route computation to select the next output port to forward the packet to. It is also responsible of updating the states of its associated virtual channels and sending a credit upstream when a flit is read from one of its queues.
- VC Allocator: allocates one free output VC to each input VC that is waiting for VC allocation.

- Switch Allocator: decides which input VCs can transmit a flit through the crossbar in the next cycle.
- Output Unit (one per output port): This module reads a flit from the input VC that won the switch allocation in the previous cycle (corresponding to crossbar traversal), and writes it to the corresponding output port. It also updates credit counters and output VC states upon receiving credits from the downstream input unit.
- Injector: Although not part of a real router, this module is necessary for simulation. Its main task is to generate new packets and schedule them for transmission.
- Ejector: This module receives flits at their destination and updates simulation statistics such as packet latency, number of received packets, etc.

In this work, we refer to Input Unit, VC Allocator, Switch Allocator, etc. as module types or module classes. We distringuich between module types and module instances.

Definition 5.1 (Module Type)**.** *A module type or module class is a routine $f(c, \sigma)$ that describes the behavior of one hardware module at a given cycle c. Every module type is uniquely identified using a positive integer. For instance, module type 0 may refer to VC allocators, module type 1 to switch allocators, etc.*

Definition 5.2 (Module Instance)**.** *Every module instance in the network is identified by its router ID r, and its module type ID m. Both are unique positive integers. For instance, if module type 0 is a VC Allocator, then ID $(15, 0)$ would identify the VC allocator of router 15. The execution of module instance (r, m) at cycle c is performed using a global function called $execModule((r, m), c)$.*

In order to obtain a model that behaves exactly like hardware in terms of timing, every value written by a module during one cycle cannot be visible to other modules until the next cycle. We also need modules to be executable concurrently and in any order. Many simulators solve this issue by adopting an Evaluate/Update mechanism, in which data is first evaluated in a temporary data set before becoming visible to other modules.

We propose a method that is inspired by the odd/even scheme used in Eggenberger and Radetzki [2013], as it requires less copying. This method consists of defining two data sets.

During even cycles, all the modules read from the first data set and write to the second data set, while in the odd cycles accesses are done on the opposite data sets. This is also similar to the table swapping scheme employed in Pétrot et al. [1997].

In this work, every module is defined as a function, and a set of registers. We incorporate the odd/even idea into our definition of registers.

Definition 5.3 (Register). *A register is defined as a tuple* (o, e) *with two associated operations: Read and Write. Given* c *the cycle at which the register is accessed, the two operations can be defined as follows:*

$Read((o, e), c) = o$ *if* $c \equiv 0[2]$,e *otherwise*

$Write((o, e), x, c) = e \leftarrow x$ *if* $c \equiv 0[2]$, $o \leftarrow x$ *otherwise*

In C, this can be efficiently implemented by declaring every register internally as a double entry array. For example:

```
typedef unsigned short reg16[2];
reg16 vc_states;
```

The read and write operations can be defined as follows:

```
#define reg_read(reg,cycle) (reg)[(cycle) & 1]
#define reg_write(reg,v,cycle) (reg)[~(cycle) & 1]=v
```

Note that this very simple mechanism solves all timing inconsistencies at absolutely no cost in performance, as these are still simple memory accesses. A problem can still arise if two modules try to write a register at the same time. However, this issue can easily be solved by ensuring that every module is allowed to read any other module's registers, but can only write to its own registers. This restriction on the programming model removes the need to use expensive atomic operations or other synchronization primitives. More importantly, it makes it very easy to define modules that behave very similarly to the simulated hardware.

Concrete module implementation examples and code samples are presented in Appendix B.

5.3.2 Module groups

While it is possible to execute all modules in parallel with proper mapping, such fine granularity is not always desirable and feasible. Therefore, we provide a means to group modules to fine tune the granularity as necessary.

Definition 5.4 (Module Group)**.** *A module group is a unique collection of module types. No two module groups contain the same module type and every module type belongs to exactly one module group. Given a module group identifier g, assume that the set of module types contained in g can be accessed using a function called $getGroupModules(g)$.*

To help make the idea of module groups clearer, fig. 5.1 shows two possible groupings of modules. In fig. 5.1 (a), 8 groups were defined. Each of the first 5 groups contains one input unit and one output unit of the same direction (East, West, North, South, Local), assuming a mesh-like topology. The sixth group includes the injector and ejector and the final two groups contain the switch and VC allocators respectively. In fig. 5.1 (b), modules were reorganized to fit in 4 groups.

Module groups are meant to be run in parallel. Therefore, the number of module groups affects the number of threads used for simulation. The module grouping method allows us to adjust the number of threads to match the capabilities of a given GPU. As we will see in Section 5.5, this indirection is necessary for simulating very large networks. In practice, module groups can be represented as a 2-dimensional array, in which the row numbers correspond to group IDs and the values in each row to module types. Rows may be terminated by a negative integer.

5.3.3 Tasks

Given the above definitions, we are now able to provide our own definition of a simulation task.

Definition 5.5 (Task)**.** *A task is the actions of one module group associated with one router. Every task is identified by one router ID r and one module group ID g. The execution of task (r, g) at cycle c is done via the function $execTask((r, g), c)$, which is presented in Algorithm 14.*

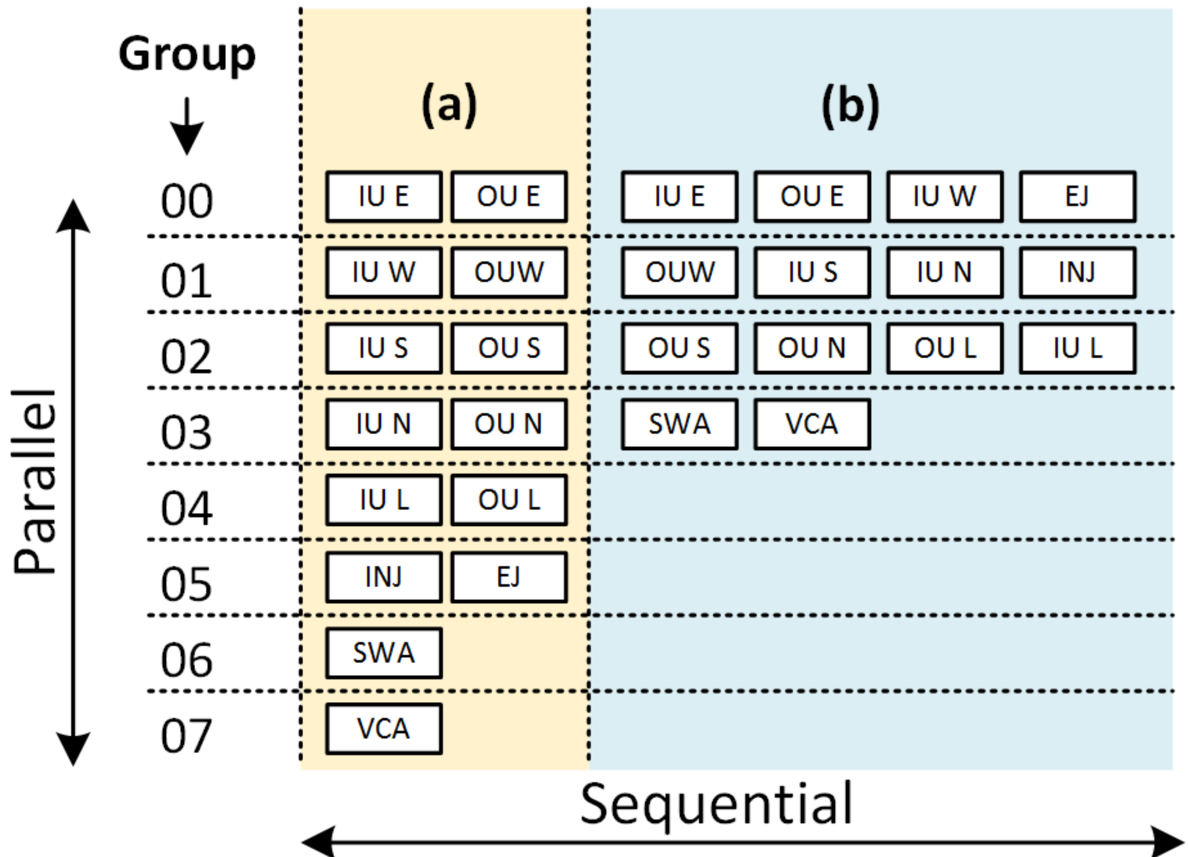

Figure 5.1: Module groups.

5.4 GPU-based implementation

We now show how we can leverage the previously defined task decomposition to develop a very fast parallel simulator on GPU using CUDA C NVIDIA [2017]. A GPU has an architecture that is fundamentally different from that of a CPU, and without proper understanding of the numerous challenges and performance pitfalls related to this architecture, it is very easy to produce a naïve implementation that yields suboptimal performance. In this section we start off by explaining the general mode of operation of a GPU. We will then point out the key implementation challenges and performance considerations and how they can be addressed. Note that even though we will be using CUDA-specific terminology, the same concepts apply to other GPU platforms as well.

Algorithm 14 Task execution routine

Input:
$taskID$: Task ID (router, module group)
c: Cycle

1: let $r, g \leftarrow taskID$
2: **for all** $m \in getGroupModules(g)$ **do**
3: $execModule((r, m), c)$
4: **end for**

5.4.1 Overview on the GPU architecture

At the software level, programming a GPU starts by defining a Kernel, which is simply a function to be executed by several GPU threads. These threads are organized in Blocks. When launching a kernel, the programmer needs to specify the number of blocks, as well as the number of threads within each block.

Threads can access several memory spaces in the GPU, the most important of which are Global Memory, which is a slow off-chip DRAM, and Shared Memory, which is a very fast on-chip cache. Each block can use a limited portion of Shared Memory (typically 48KB), which is visible only to threads within the same block. At the hardware level, GPU resources are organized in units called Stream Multiprocessors (SMs). Each SM contains several processing cores, a Shared Memory cache to be used by blocks, and registers to be used by threads. During execution, threads are divided into fixed sized groups called Warps. All threads within a warp execute the same instruction at once, following the SIMT (Single Instruction Multiple Threads) paradigm. More architectural details will be revealed as needed in the remainder of this section.

5.4.2 Warp-friendly task mapping

Since all threads of a warp execute the same instruction, they cannot branch to different locations simultaneously. If threads of the same warp happen to execute different branches of code, these executions are serialized. While this is not a major issue for simple conditional statements within modules, having threads of the same warp execute different module groups can be highly inefficient. Clearly, task mapping cannot be performed in an arbitrary manner.

Fortunately, the way thread IDs are assigned to warps is deterministic. For instance, assuming an architecture with W threads per warp, the first W thread IDs are guaranteed to be in the

same warp, the same applies to threads of IDs between $W$and $2W-1$, and so on.

We propose a mapping that minimizes the divergence between threads of the same warp. Assuming that W is the number of threads per warp in a given architecture (typically 32), and R the number of simulated routers, we identify the threads using two coordinates (x, y) such that the absolute thread ID is equal to $yR + x$. Every task (r, g) is then mapped to thread (r, g) as shown in fig. 5.2. Since the y coordinate is the slowest changing coordinate in the thread ID, the module group only changes every R threads, which means that there is a lower chance that different module groups, which correspond to different y coordinates, are contained within the same warp. Moreover, if the number of simulated routers is a multiple of W, then all the threads within one warp are guaranteed to be executing the same module group, i.e. the same code. For network sizes that are not multiples of W, padding the network using dummy nodes that do not receive or generate packets can offer better performance.

5.4.3 Compact flit queue implementation

The next issue that needs to be addressed is that of memory accesses. Accessing global memory is very costly. While some level of optimization is possible by adjusting alignment and coalescing, reorganizing the simulation data to meet all of these constraints can be tedious. It would be much more convenient if we could store all simulation data in the very fast, shared memory cache and not have to worry about the way data is accessed. However, the amount of shared memory that can be used by one block is usually very limited (typically 48KB), so fitting all the simulation data in this memory space can be rather challenging. Some obvious optimizations are possible nonetheless. For instance, the states of several virtual channels can all be stored in one integer. This has the double benefit of both saving memory space and allowing for constant time round-robin arbiter implementations using a series of bitwise operations. The most important optimization, however, lies in our implementation of flit FIFOs.

Our representation is based on two observations. First, assuming atomic VC allocation, one virtual channel can only be storing flits of the same packet. Second, apart from the head flit, which stores useful information used for routing, the flits of a packet are of little interest to the simulator, as we are usually only interested in knowing whether they are present or not. Thus,

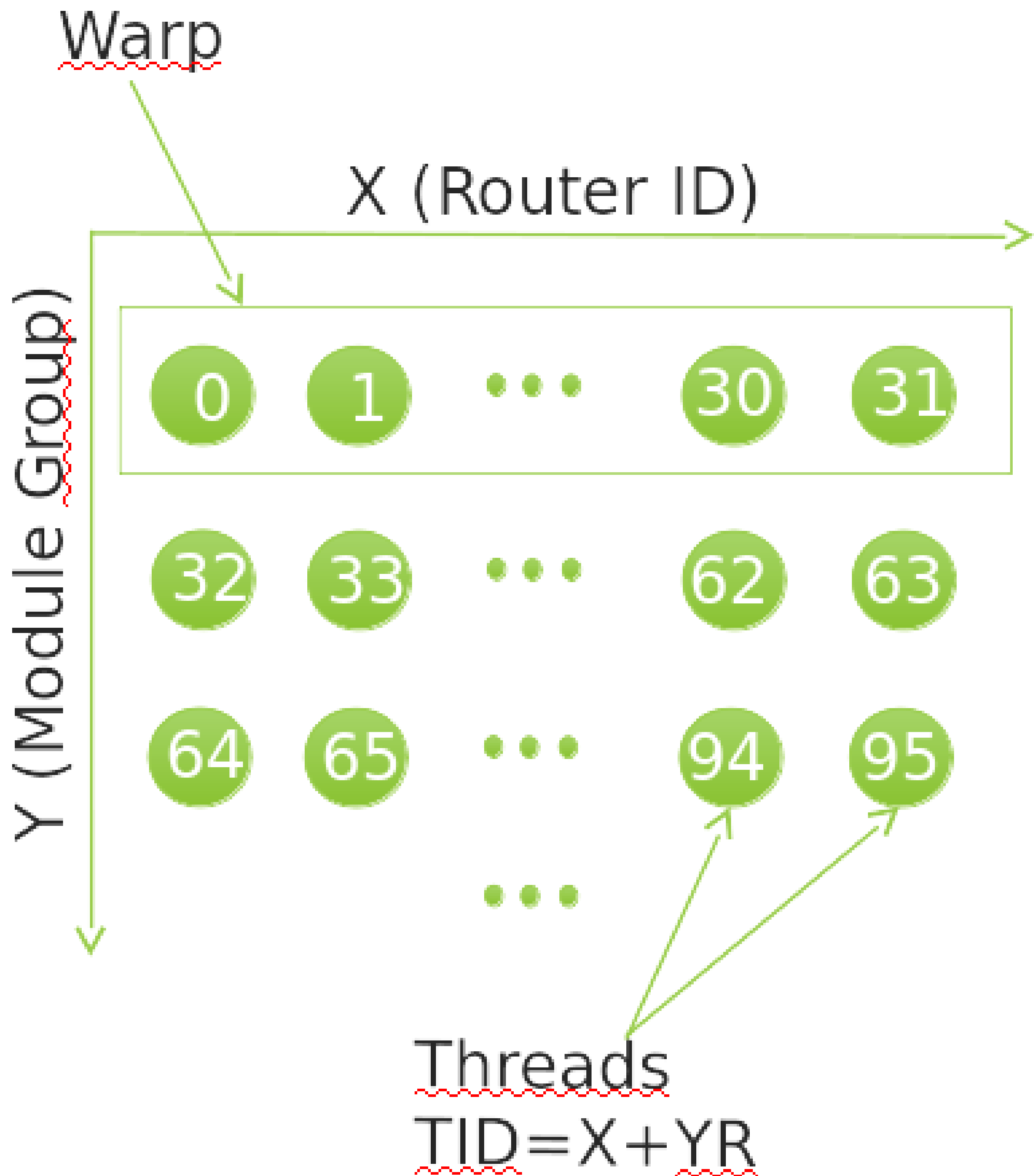

Figure 5.2: Mapping of tasks onto threads.

we propose the representation shown in fig. 5.3.

This structure, which can fit in one integer (32 or 64 bits depending on the simulation parameters), is used to represent a sequence of flits (from start to end) of the same packet of size

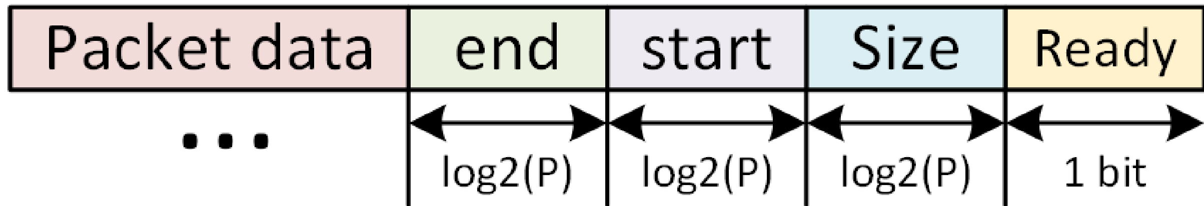

Figure 5.3: Compact flit queue representation.

$size + 1$. It can also represent an entire packet when $start = 0$ and $end = size$, a single flit when $start = end$, and an empty queue when $start > end$. The $enqueue$ operation is implemented by incrementing the end field, and $dequeue$ by incrementing the $start$ field. The type of an individual flit can be deduced from the values of $size$, end and $start$. The "packet data" field contains information such as the destination of the packet, its allocated VC at every hop, and latency. Thanks to this representation, we were able to fit enough data to simulate a 64 node network with 4 virtual channels in less than 40KB of shared memory.

5.4.4 Simulating large networks

Due to the limited amount of shared memory and number of threads per block, there is a maximum number of routers that can be simulated in one block using a given module grouping. Simulating larger networks may therefore necessitate the creation of multiple blocks. Since shared memory is only visible to threads within the same block, the structures used to transfer flits and credits between routers must be stored in global memory.

To ensure cycle-accuracy, all threads of all blocks must be synchronized at the end of each cycle. This is done using the lock-based synchronization method introduced in Xiao and Feng [2010], but with a slight but important modification. Our synchronization function is presented in Algorithm 15, where $__syncthreads()$ is a native barrier used to synchronize all threads within a block.

Compared to the solution in Xiao and Feng [2010], we have one extra call to $__syncthreads()$ before the inter-block synchronization. This is to prevent cases where thread $(0, 0)$ gets to the synchronization point and allows other blocks to proceed to the next cycle while threads of its

Algorithm 15 Synchronization routine

Input:
$threadID$: Thread ID (x, y)
$target$: Target value to reach by all threads

1: let $x, y \leftarrow threadID$
2: $__syncthreads()$
3: **if** $x, y = (0, 0)$ **then**
4: **atomically** $SyncVar \leftarrow SyncVar + 1$
5: **while** $SyncVar \neq target$ **do**
6: **end while**
7: **end if**
8: $__syncthreads()$

own block are still running in the current cycle.

This synchronization method only works if all blocks are executed concurrently. If block execution is serialized, then a deadlock will inevitably occur. Generally, if the number of executed blocks is at most equal to the number of SMs (Stream multiprocessors) of the GPU, then each block is scheduled in a separate SM and the deadlock situation can never occur. However, if the number of blocks exceeds the number of SMs, then several blocks have to be scheduled on the same SM. In this case, blocks can still be executed in parallel provided that SM resources (mainly the amount of shared memory and the number of thread registers) are sufficient to accommodate all the blocks simultaneously.

Luckily, module grouping already solves the cases where the number of threads needs to be reduced either due to an insufficient number of registers or to an excess in the number of threads per SM. However, we should in addition make it possible to evict some of the simulation data from shared memory. We have achieved this in our implementation by providing compile time options to disable shared memory caching for certain modules. Of course, this only affects the way some pointers are initialized and does not change the way modules are programmed. By carefully setting module grouping and shared memory caching, it is possible to fit very large networks into a low-end GPU with limited resources.

5.4.5 Simulation kernel and final notes

After addressing various design issues, we can now define the simulation kernel as in Algorithm 16. In our implementation, constant memory space was used to store the module grouping table.

For pseudo random number generation, which is essential for the injection process, we have used a CUDA library NVIDIA [2017] called cuRAND.

Algorithm 16 Simulation kernel

Input:
  $threadID$: Thread ID (x, y)
1: let $cycle \leftarrow 0$
2: let $router, group \leftarrow threadID$
3: **while** $cycle < SimulatedCycles$ **do**
4:   $execTask((router, group), cycle)$
5:   $cycle \leftarrow cycle + 1$
6:   $synchronize(threadID, cycle * NumBlocks)$
7: **end while**

5.5 Experimental results

In this final section, we evaluate our simulator implementation in terms of performance and accuracy.

5.5.1 Speedup

As a performance metric we consider the execution speedup, which is obtained by dividing the execution time of a sequential simulation, in which all modules are executed iteratively, by that of the GPU-based parallel simulation.

The sequential version was compiled with level 2 compiler optimizations enabled and run on an AMD A8-6500 @3.5GHz under Ubuntu Linux. For the parallel version, we use an NVidia GeForce GTX980Ti graphics card, which features 22 SMs (Stream Multiprocessors), each including 96KB of shared memory. Each simulation was run 10 times and the execution times were averaged. For all simulations, we have only counted the time spent in the actual network simulation, not including initialization and memory allocation. Several mesh sizes were tested (16x16, 24x24 and 32x32) with 2 or 4 VCs, using the network parameters presented in table 5.1. In all tested configurations, we were able to fit an 8x8 mesh network in a single block of threads. Therefore, a 16x16 mesh was simulated using 4 blocks of 8x8, a 24x24 mesh using 9 blocks, and a 32x32 mesh using 16 blocks.

Table 5.1: Simulation parameters

Parameters	Value(s)
Buffers/VC	4
Flits/Packet	5
Traffic pattern	Uniform Random
Routing Algorithm	XY
Simulated cycles	100000
Injection rate	0.001 packets/node/cycle

We have used 8 module groups organized exactly as shown in fig. 5.1 (a). It should be noted that we have not noticed any improvement in speedup when using more module groups, mainly because some modules have very different complexities and running them in parallel forces faster modules, such as output units, to spend more time waiting for more complex modules, such as the VC allocator, to finish executing. The results are presented in table 5.2. First, notice the very high speedups obtained in all simulations, especially for larger networks. Simulating a 32x32 network is 100 times faster on GPU. This reflects the high level of parallelism that is possible using our method. Assuming a perfectly scalable CPU-based parallel simulator, achieving such high speedups would require a machine comprising at least 100 hardware threads.

Table 5.2: Speedup results (1)

16x16x2vc	16x16x4vc	24x24x2vc	24x24x4vc	32x32x2vc	32x32x4vc
26.53	28.60	55.07	59.65	95.53	102.59

Thus far, we have tested networks that required a number of blocks that is smaller than the number of SMs, which is why we were able to run the simulations at the desired granularity with no compromises. In order to push the GPU to its limits, we now try to simulate larger networks (40x40 to 64x64).

Since the number of required blocks is higher than the number of SMs, several blocks need to share SM resources. As indicated in the results presented in table 5.3, it was necessary to reduce the number of module groups to 7 in order to be able to simulate 40x40 and 48x48 networks, and we could only use up to 4 module groups to simulate 56x56 and 64x64 networks. Moreover, since thread blocks also need to share Shared Memory, global memory had to be used to store some modules when simulating 56x56 and 64x64 networks with 4 VCs (see Section

5.4.4). In this case, only the injectors were removed from shared memory. We can see that in spite of these restrictions, the speedup continues to scale with the number of nodes, exceeding 250x when simulating a 4K node network. As a point of comparison, simulating a 64x64 network with 4 VCs using Booksim Jiang et al. [2013] took approximately 25 minutes and 20 seconds, whereas our GPU-based simulator produced identical simulation statistics in only 1.2 seconds, which is over a thousand times faster.

Table 5.3: Speedup results (2). (*) Injectors evicted from shared memory.

# groups	40x40x2vc	40x40x4vc	48x48x2vc	48x48x4vc
7	139,92	146,75	190,55	193,37
# groups	**56x56x2vc**	**56x56x4vc**	**64x64x2vc**	**64x64x4vc**
7	210,33	182,03*	277,27	252,90*

5.5.2 Hardware fidelity

Finally, we evaluate our simulator in terms of how precisely it can mimic hardware behavior, by comparing the results produced by our reference implementation to those obtained using an RTL model based on Netmaker Mullins [2009], which is a library of synthesizable NoC routers written in SystemVerilog. We have simulated an 8x8 mesh network with 2 VCs for 100000 cycles. Single flits were injected at varying rates and the average network latency results are presented in fig. 5.4. Note that only the network latency, i.e. the time flits spend in the network, was measured. The queuing delays were not included in the results.

We can see that our simulator and the RTL model exhibit identical behaviors and the difference in latency was less than 5% for all the tested injection rates. These results indicate that our simulator is not only ultra-fast, but also matches RTL accuracy.

5.6 Conclusions

In this final chapter, a novel GPU-based NoC simulation method was introduced. We have presented a module specification that offers high hardware fidelity, thread safety, and easy extensibility. These properties are suitable for parallel NoC simulation in general, not only on

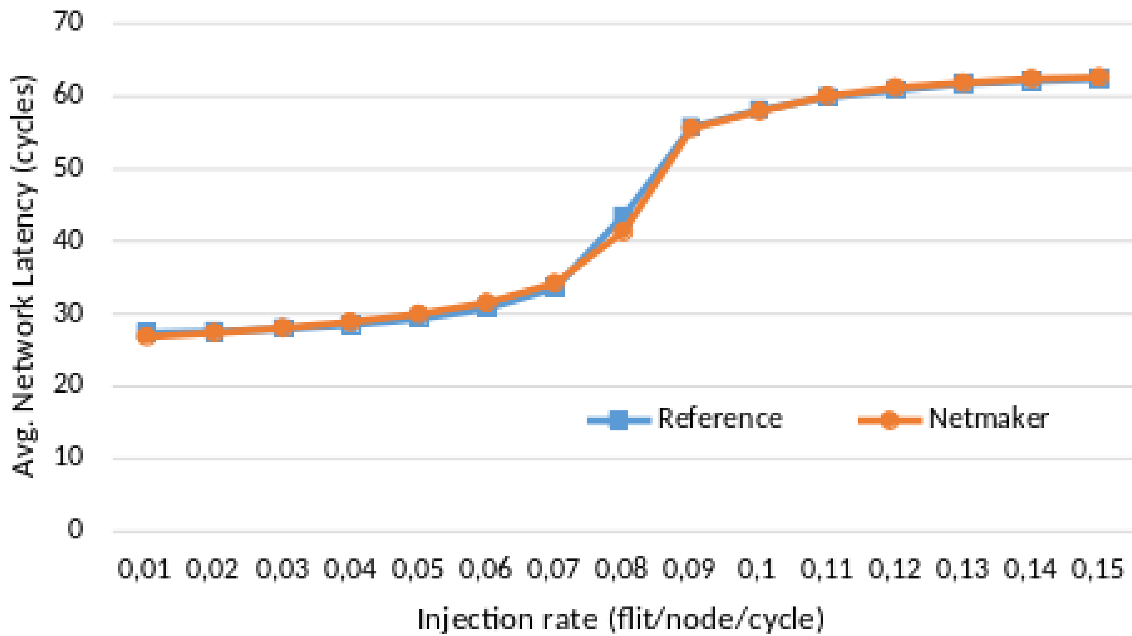

Figure 5.4: Average Network Latency (RTL vs. GPU).

GPU. Our module group abstraction, which decouples the definition of modules from their mapping on the platform, has proven extremely useful for simulating very large networks (4K nodes) on GPU and excellent speedups were obtained (280x). We have addressed GPU-specific implementation challenges such as thread divergence, memory usage and synchronization to implement a simulator capable of producing very accurate statistics at ultra-high speeds, making it an ideal tool for modeling and validating future NoC designs consisting of thousands of nodes.

Part V

CONCLUSIONS

Chapter 6

Conclusion and Future works

6.1 Conclusions

After establishing that the current methods used to construct deadlock-free routing algorithms were too restrictive in the way they reserve virtual channels, we have proposed a new theory and design methodology that offer more routing freedom and flexibility in terms of buffer utilization. This increased flexibility translates into higher throughput as well as better resilience.

Beyond the observed performance gains, because this new theory classifies individual packets according to the way they are routed, not the channels that they are occupying, we believe that it provides a more intuitive understanding of how routing algorithms operate. It also makes it trivial to define fine-grained flow control schemes that support non-atomic VC reallocation and deadlock escapes at the same time. To the best of our knowledge, existing theories do not offer this possibility. We have shown through several examples how this approach can be used to rethink the existing routing solutions as well as constructing new ones, regardless of the target topology. We hope that our proposal serves as a framework for the design of future routing algorithms for various applications.

In this thesis, we have also addressed a more specific challenge, which is that of deadlock-free routing in partially connected 3D topologies. Due to the emergence of TSV as a promising vertical communication technology, it is necessary to provide efficient routing algorithms for such topologies at a reasonable cost. After exploring the existing proposals, we have come to the conclusion that there existed a trade-off between the number of virtual channels (the cost),

and the freedom in terms of TSV placement and selection at runtime.

We have demonstrated that it was possible, through a clever attribution of VCs, to eliminate, to some extent, this trade-off. The First-Last routing algorithm that we presented requires the addition of only 1 VC in the East and North directions, resulting in a great cost reduction compared to Elevator-First, which requires a VC in all the directions. It guarantees full connectivity in the presence of at least one TSV pillar in the upward and downward directions. Like Elevator-First, the pillars can be placed anywhere in the network, and during runtime, any pillar can be selected.

This high level of freedom compared to algorithms that use the same number of virtual channels is due to the unique way in which we chose to place the extra VCs. In fact, our algorithm is the first one to allow different numbers of VCs along the same dimension (2 VCs East, but only 1 VC West). Traditionally, authors tend to only consider symmetrical placement of VCs (e.g. 1 VC along the Y dimension), even though this symmetry has no real advantage and is limiting in terms of possible solutions, as we hope to have demonstrated in this contribution. The First-Last algorithm is therefore an appealing routing solution in terms of cost, performance as well as resilience, making it a great candidate to adopt in future 3D designs.

The final contribution of this thesis pertained to the parallelization of cycle-accurate simulations of NoCs. This has long been a challenging problem mainly due to the high overhead incurred by thread synchronization, which has to be performed on a per-cycle basis in order to preserve timing accuracy. Although GPUs make it possible to create enough threads to compensate for the large synchronization overhead, they are often overlooked in the field of system simulation, mainly because they tend to require too much optimizations in terms of memory and thread divergence which limits the flexibility of the coding style as well as the extensibility. In response to these concerns, we have shown that in the case of NoCs, proper task decomposition and mapping could remove the need to think about optimization when programming the hardware modules. We have obtained excellent speedups even though the simulated system consists of completely heterogeneous modules. An implementation of the proposed design has served for the fast simulation and evaluation of all the architectures presented throughout this manuscript.

6.2 Future directions

In the field of parallel simulation, an interesting research direction would be the simulation of a full system (cache hierarchy, processing cores) on GPU platforms. Although Network-only simulation is useful and widely used for NoC research, being able to simulate a full system at high speed would offer more insight into the way the different components interact. Moreover, the mapping of the different controllers into a 3D topology would help better understand the traffic patterns, and subsequently help drive the placement of TSVs as well as the choice of the best routing configuration to adopt, opening up new research opportunities in the field of 3D-NoCs as well. Most current works in this area have been using random topologies, combined with either synthetic traffic, as we did in the context of this thesis, or traffic traces generated for other types of topologies that were not meant to support a large number of nodes.

We think that such simulation platforms are achievable today thanks to the increasing amount of resources included in modern GPUs, as well as faster multi-GPU communication.

Publication list

A. Charif, N. E. Zergainoh and M. Nicolaidis, ”A new approach to deadlock-free fully adaptive routing for high-performance fault-tolerant NoCs,” 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT’16), Storrs, CT, 2016, pp. 121-126.

A. Charif, N-E. Zergainoh, M. Nicolaidis, ”A Dynamic Sufficient Condition of Deadlock-Freedom for High-Performance Fault-Tolerant Routing in Networks-on-Chips.” IEEE Transactions on Emerging Topics in Computing, Accepted for publication.

A. Charif, A. Coelho, N-E. Zergainoh, M. Nicolaidis. ”MINI-ESPADA: A low-cost fully adaptive routing mechanism for Networks-on-Chips.” Test Symposium (LATS’17), 2017 18th IEEE Latin American. IEEE, 2017.

A. Charif, N-E. Zergainoh, M. Nicolaidis. ”Addressing transient routing errors in fault-tolerant Networks-on-Chips.” Test Symposium (ETS’16), 2016 21th IEEE European. IEEE, 2016.

A. Charif, N-E. Zergainoh, M. Nicolaidis. ”MUGEN: A high-performance fault-tolerant routing algorithm for unreliable Networks-on-Chip.” On-Line Testing Symposium (IOLTS’15), 2015 IEEE 21st International. IEEE, 2015.

A. Charif, A. Coelho, N-E. Zergainoh, M. Nicolaidis, ”A Framework for Scalable TSV Assignment and Selection in Three-dimensional Networks-on-Chips,” VLSI Design, Accepted for publication.

A. Charif, N-E. Zergainoh, A. Coelho, M. Nicolaidis. ”Rout3D: A Lightweight Adaptive Routing Algorithm for Tolerating Faulty Vertical Links in 3D-NoCs.” 22th IEEE European Test Symposium (ETS’17). ACM IEEE, 2017.

A. Charif, A. Coelho, N.-E. Zergainoh, M. Nicolaidis. ”Detailed and highly parallelizable cycle-accurate network-on-chip simulation on GPGPU.” ACM/IEEE Design Automation Conference (ASPDAC’17), Jan 2017, Chiba/Tokyo, Japan. ACM IEEE, pp.672-677.

A. Charif, A. Coelho, N-E. Zergainoh, M. Nicolaidis, ”GNoCS: An Ultra-fast, Highly extensible, Cycle-Accurate GPU-based Parallel NoC Simulator.” University Booth of the ACM/IEEE Design Automation and Test in Europe Conference (DATE’17), Lausanne, Switzerland, 2017.

Part VI

APPENDICES

Appendix A

Congestion metrics for adaptive routing

A.1 Introduction

Adaptivity is a highly desirable feature in modern routing solutions, as it makes it possible to route packets around hotspots and congested areas. This thesis has introduced several new adaptive routing algorithms. However, in order to properly benefit from their flexibility for congestion avoidance purposes, the routing unit needs a way to measure the level of congestion at each output port, so as to make the best possible choice at every step.

Keeping track of the level of congestion at every output is a difficult task, mainly because the routers have very limited knowledge about the global state of the network, but also because congestion depends on the interactions of too many flows. Although global and regional congestion awareness has been studied in previous works, the mechanisms required for the propagation and the bookkeeping of congestion data can be quite complex. Moreover, the delay due to the propagation of various events limits the precision of the congestion estimation, and consequently the overall performance gains.

By contrast, **local** congestion metrics, such as the number of free buffers Kim et al. [2005] (free_buf) or the number of free VCs Dally and Aoki [1993] (free_vc), which are computed based on the events that occur within the router itself are widely adopted in practice.

In the literature, several promising local congestion metrics were proposed. In Gratz et al. [2008], the authors estimate the congestion at one port as the number of crossbar requests made to that port (x_req). They also evaluate different combinations of the metrics (free_buf, free_vc,

x_req).

In Dimopoulos et al. [2014], a new metric called Flits Remain (fr) is introduced. The idea is to estimate the congestion at one port as the total number of flits that still need to leave from it. It was shown to offer considerable performance gains compared to the free_vc metric.

In this appendix, we present two new contributions:

- We present a new notation for precisely describing a congestion metric. Instead of describing it as its value at a given moment in time, we propose a description of the way it is updated over time in response to specific events.
- We introduce two new congestion metrics and compare their effectiveness to that of other local metrics.

A.2 Generic congestion management unit description

Our goal is to associate a congestion value to each output port in a router. In practice, this means that there is a fix-sized register associated with each output direction.

This register's value is updated in response to a series of events that occur within the router, so as to reflect the current congestion value. From a designer's point of view, we believe that the value of a congestion metric is better described using the way it is updated in response to each event, than using a mathematical formula that gives the congestion value at each instant in time Avresky et al. [2014]. This section presents a new notation for describing the dynamic value of a congestion value associated with an output port. We start by identifying the various events that can stimulate the congestion management unit to update its registers as well as the associated data, followed by the different ways the register can be updated, then we present the notation we use for describing a congestion metric.

A.2.1 Router events

We identify the following events, that can change the congestion state of one output port:

- $route(p)$ This output port was selected by the routing unit for packet p. The structure of

a packet is not defined here as it depends on the target design. For instance, if the size of the packet is of interest then the field $p.size$ may be referenced.

- $vc_alloc(p, v_{out})$ A virtual channel from this output port (v_{out}) was allocated to packet p.
- $flit(p, v_{out})$ A flit of packet p is leaving this port to enter VC v_{out} in the next router.
- $free(last, v_{out})$ A flit has left the downstream VC v_{out} so a buffer is free. $last$ is true if v_{out} is now empty, false otherwise.

For a typical router architecture, the above-mentioned events are sufficient to describe all the local congestion metrics that we are interested in. The list may be updated for other router architectures.

A.2.2 Updating the congestion value

A congestion value **update** is represented using one arithmetic operator and an integer value. For instance, $+1$ means that the register must be incremented by one. $*p.size$ means that the current value should be multiplied by the packet's size.

Conditional updates can be described using the C-like notation

$(p.size > 5? + 0 : +1)$

The above example means that the congestion value is only incremented if the packet's size is no higher than 5.

A.2.3 Congestion metric description

We now present the syntax for describing congestion metrics by providing the description of all the previously cited local metrics.

A.2.3.1 The free_buf metric

This is perhaps the simplest and most commonly adopted congestion metric. It simply consists of counting the number of free buffers in the downstream router. The less buffers are available, the more congested is the output port. Below is the description of the free_buf metric:

```
init=0
on flit(p, v_out): +1
on free(last, v_out): -1
```

The syntax is straightforward. We start by providing the initial congestion value. In this case, the congestion value is 0 for the output port. Then we provide the update to perform in response to each event. The events that do not affect the value need not be specified.

For this simple metric, we increment the congestion level by 1 when a flit leaves the output port (one less buffer available downstream). We decrement the congestion level when a buffer becomes free again.

A.2.3.2 The free_vc metric

This metric considers the output ports with the most available VCs as the least congested ones. It can be described as follows:

```
init=0
on vc_alloc(p, v_out): +1
on free(last, v_out): (last? -1: +0)
```

Here, we increment the congestion value as soon as an output VC is acquired (as it is no longer free). We only decrement the congestion value if the last flit of the packet has left the VC, i.e. the VC can be reallocated again.

A.2.3.3 The x_bar metric

This metric counts the total requests for each output port. It can be described as follows:

```
init=0
on route(p): +1
on free(last, v_out): (last? -1: +0)
```

Compared with free_vc, this metric not only counts the active requests (allocated VCs), but also the VCs waiting to be allocated later. From the description, it is easy to see why the x_bar

offers a better estimation of congestion than the other two metrics.

A.2.3.4 The fr metric

Finally, we present the description of the fr (Flits Remain) metric. The purpose of this metric is to accurately measure the amount of multiplexing taking place at a give output port by counting the number of flits that are going to leave from it. That is, every time a packet selects an output port, the congestion value is incremented by the number of flits contained in the packet. Every time a flit leaves from that output port, the congestion value is decremented. This can be written as follows:

```
init=0
on route(p): +p.size
on flit(p, v_out): -1
```

A.3 Introducing new congestion metrics

In this section, we present two new congestion metrics to overcome the limitations of the previous local metrics. The first metric is called FRN (Flits Remaining on Neighbor), which improves upon the original fr metric by taking the multiplexing at the next hop into account. The second metric (Not All Flits Are Equal) aims at offering an even finer measurement of the amount of multiplexing in the current and downstream routers.

A.3.1 The FRN congestion metric

The idea behind this metric is quite similar to fr, but the difference is that we consider that a flit keeps contributing to congestion in a given direction until it leaves not only the current router, but also the next one. Below is the detailed representation of the FRN metric.

```
init=0
on route(p): +p.size
on free(last, v_out): -1
```

Note that to perform this improvement, all we had to do was change the event that reduces the congestion value.

A.3.2 The NAFAE congestion metric

One shortcoming of the FRN metric is that it does not differentiate between the flits that are occupying the current router, and consequently slowing down the progress of all the packets heading in the same direction, and the flits occupying the downstream router, which should leave sooner. The purpose of the NAFAE metric is to give more weight to the flits that are waiting to leave the output port (weight=2) than to those flits that have already left (weight=1). This can be concisely written as follows:

```
init=0
on route(p): +(2*p.size)
on flit(p, v_out): -1
on free(last, v_out): -1
```

A.4 Experimental results and conclusions

We compare the effectiveness of the proposed congestion metrics to that of the existing local metrics by simulating an 8x8 mesh network with 4 buffers per VC. The packet size is fixed to 5 flits and a fully adaptive routing algorithm is assumed. Whenever several possible routes are possible (North and East, South and East, North and West, South and West), the least congested port, i.e. the output port with the lower congestion value is selected by the routing unit. We have used our cycle-accurate simulator described in Chapter 5 and each simulation was run for a duration of 100000 cycles.

The obtained results are presented in fig. A.1. We can see that as expected, the simple free_buf metric offers, in general, a less precise measurement of congestion than the other metrics. However, under some traffic modes, we can see that it can perform better than the fr and x_req metrics (Shuffle traffic). By contrast, the two proposed metrics (frn and nafae) consistently outperform the existing metrics in all the tested scenarios. Most notably, the frn metric

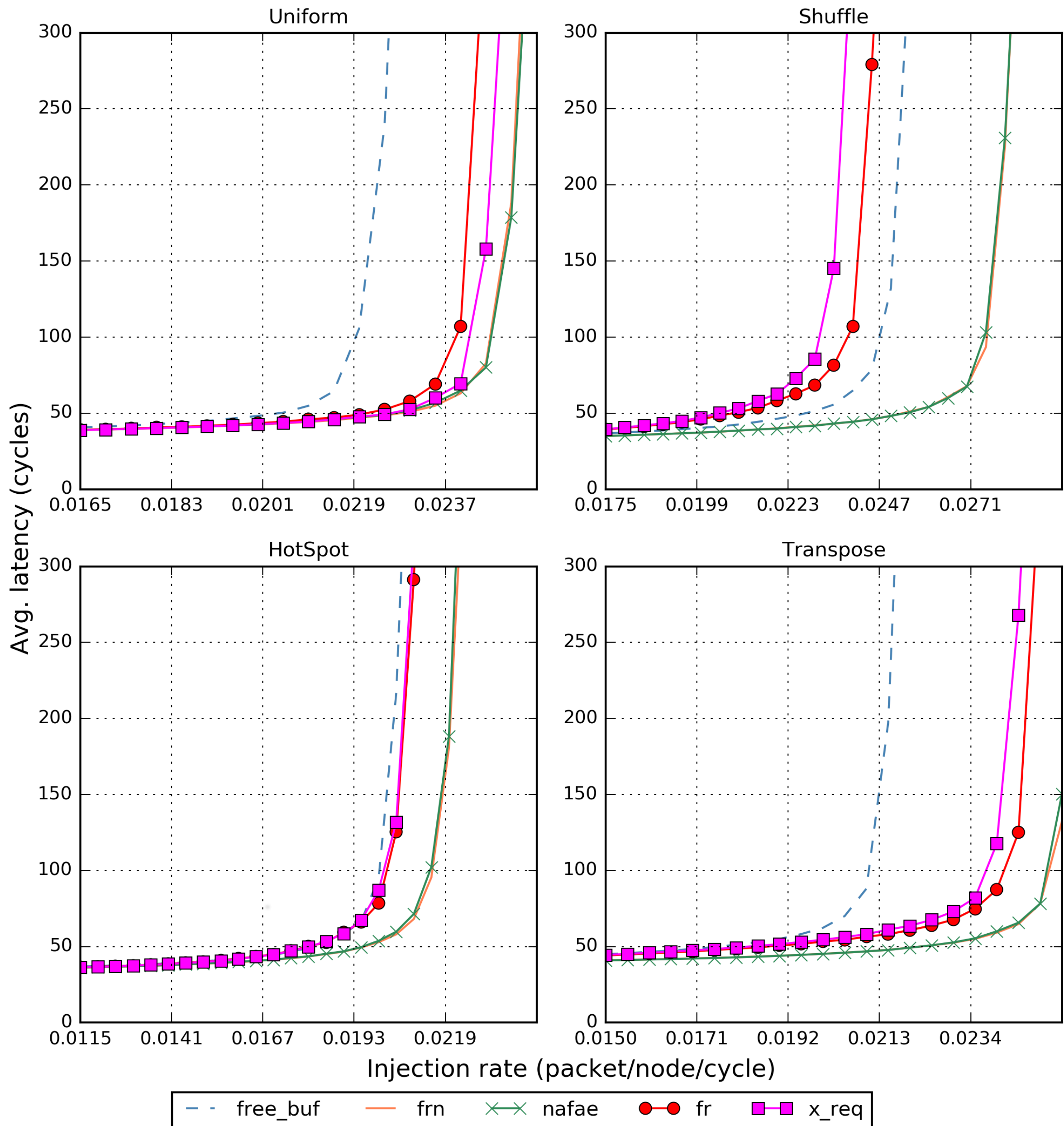

Figure A.1: Average packet latency obtained with different congestion metrics.

performs surprisingly well in spite of not incurring any extra overhead compared to the initial fr metric.

Generally speaking, the significant difference in latency that we observe in these experiments highlights the important role that congestion estimation plays in the overall system performance.

Appendix B

GNoCS: A highly-extensible GPU-based parallel network-on-chip simulator

B.1 Introduction

This Appendix provides details about the implementation of GNoCS, which is the tool that we have developed based on Chapter 5 and have used for all the experiments presented in this thesis.

B.2 Basic data types and API

B.2.1 Integer API

As it is meant to be a lightweight simulator, GNoCS is implemented using only native C integers. Nevertheless, we redefine these types for convenience and provide the following type names: $byte$, $int16$, $int32$ and $int64$, as well as their unsigned variants $ubyte$, $uint16$, $uint32$ and $uint64$.

Because all the bits and bit vectors contained within a router will be represented using these types, we provide the following macros for manipulating their contents:

```
// vector is of an integer type

// Treat vector as a succession of 'count' bit values
// and get the value at offset 'offset'
read_bits(vector, offset, count)
```

```
// Treat vector as a succession of 'count' bit values
// and set the value at offset 'offset' to data
write_bits(vector, offset, count, data)
```

As well as wrappers to access individual bits:

```
#define set_bit(vector,offset) write_bits(vector,offset,1,1)
#define clear_bit(vector,offset) write_bits(vector,offset,1,0)
#define get_bit(vector,offset) read_bits(vector,offset,1)
```

B.2.2 Registers

As explained in Chapter 5, communication between modules is done via `registers`. We provide register types $reg8$, $reg16$, $reg32$, $reg64$, $ureg8$, $ureg16$, $ureg32$, $ureg64$ to store the corresponding integer types. The value stored in a register can be retrieved or modified using the macros:

```
reg_write(reg, data, parity)
reg_read(reg, parity)
```

Where parity is calculated by each thread once every cycle as

```
parity=cycle & 1;
```

The way these types and macros are used will be demonstrated in Section B.3.1.

B.2.3 Flit queue

The following structure is used to model a set of successive flits, as described in Chapter 5.

```
typedef struct {
    uint64 size: _4_bits;
    uint64 destx: _3_bits;
    uint64 desty: _3_bits;
    uint64 destz: _2_bits;
    uint64 vc: _2_bits;
    uint64 start: _3_bits;
    uint64 end: _3_bits;
    uint64 ready: _1_bit;
    uint64 _class: _2_bits;
    uint64 delay: _rest_from(64);
} flit;
```

The $_x_bits$ and $_rest_from$ macros help keep track of how many bits were used for the previous fields and to know how many bits are left to use (out of the initial 64 bits) for the remaining fields.

Because flits are stored in registers withing the router, we also provide a register type for flits:

```
typedef flit flitreg[2];
```

The reg_read and reg_write macros can be used to read and write flit registers as well. Following is an example code that reads a flit from an input fifo and writes it to an output register. This corresponds to crossbar traversal in hardware.

```
struct InputUnit {
    flitreg fifo[NVC];
    ...};
struct OutputUnit {
    flitreg link;
    ...};
```

```
{   ...
    // read flit at head of input port i, virtual channel j
    flit f = reg_read(input_unit[i]->fifo[j], parity);

    f.ready = 1; // signal flit presence to next router

    // write flit to selected output port k
    reg_write(output_unit[k]->link, f, parity);
}
```

And below is the code executed by the `InputUnit` module to dequeue the flit when it traverses the switch.

```
{   ...
    // read flit at head of virtual channel vc
    flit f = reg_read(self->fifo[vc], parity);

    f.start++; // remove one flit from queue
    if (f.start == f.size) {
        // last flit, update channel states
    }

    // write back
    reg_write(self->fifo[vc], f, parity);
}
```

B.3 Programming model

In this section, we describe the model used to program the different modules of the simulator. This model should be followed by newly defined modules as well in order to preserve the

correctness and accuracy of the results.

Every module in the router must perform the following three steps:

- Read all the registers it needs, from any module including itself.
- Use the read values to do computation.
- Write a value in **each of its own registers**.

B.3.1 Example module: Credit manager

To illustrate the above steps, we implement an example module called `credit_manager`. This module keeps count of the number of free buffers (aka credits) in the downstream router's virtual channels. The switch allocator uses these values to determine whether there is room in the downstream router for a new flit.

This module operates as follows: Whenever the switch allocator authorizes a flit to leave from an output port, the number of credits corresponding to the output VC is decremented. The number of credits is incremented as soon as a credit signal is received from the downstream router, indicating that there is a newly available buffer in the corresponding VC.

It should be noted that the actions performed by this example module are too simple to be defined as a standalone module. In GNoCS, credit management is performed by the output unit. However, we define it in a separate module here for the sake of illustration.

First, we define the structure holding the registers, and other temporary data owned by the module:

```
typedef struct {
    reg8 credit_count[NVC];
} CreditManager;
```

The credit manager module only requires one register per VC to store the number of free buffers. The module requires the results of two other modules: the switch allocator (to check if a flit is leaving through this output port in the current cycle), and a credit signal link, to check if a credit was received by the downstream router.

```
struct SwitchAllocator {
  ...
  // bit n is active if output VC n is receiving a flit
  ureg8 alloc_output[NPORTS];
};
struct CreditLink {
  // bit n is active if credit received from VC n
  reg8 valid;
};
```

The module itself can now be implemented as follows:

```
__device__ void doCreditManager(
    CreditManager* self,
    SwitchAllocator* sa,
    CreditLink* cl,
    bool parity) {
      byte credit_valid = reg_read (cl->valid, parity);
      byte allocated_vcs=reg_read(sa->alloc_output[p],parity);
      for (int vc = 0; vc < NVC; vc++) {
        //read
        byte credits=reg_read(self->credit_count[vc],parity);

        //modify
        if (get_bit(credit_valid,vc)) credits++;
        if (get_bit(allocated_vcs,vc)) credits--;

        // write
        reg_write(self->credit_count[vc],credits,parity);
      }}
```

B.4 Simulation automation

Many simulators take a configuration file as an input to perform a single NoC simulation. For GNOCS, we have developed a very powerful Python program designed to run multiple simulations, process the results (average etc.), save the simulation output as well as generate the plots, all from a single configuration file. This section describes the features of this automation tool called `run.py`.

B.4.1 Structure of a simulation description file

The configuration file used by the `run.py` program follows a JSON-like syntax. Internally, it is read as a Python `dict` (dictionary) object. It consists of key-value pairs, where the keys are strings and the values can be of any type.

The keys starting with an underscore character (_) are considered as directives to the automation tool. For instance, compilation options, path to the simulator, the results to plot, size and other options for the generated figures, etc. The other keys are arguments to the simulator itself. As an example, we present a simplified version of the configuration file used to generate figure 3.4 from Chapter 3.

```
{
    "_netsize": (8, 8, 2),
    "_force_recompile": True,
    "_skip_existing": False,

    "cycles": [100000],
    "routing_algorithm": ['elevator-first',],
    "traffic": ['Random','BitComplement','Shuffle'],

    "elevators": [48, 32, 16, 8],
    "elevator_selection": ['safe','roff','opt','ron'],

    "_results": ["average_latency"],
```

```
"random_seed": randoms(50), # 50 iterations
"_average": ["random_seed"],

#packet injection rate
"pir": make_range(0.0005, 0.0165, 0.0005),

"_plot": [ {
    "x": "pir",
    "y": "*average_latency",
    "cmp": "elevator_selection",
    "ranges": { "y": (0,400) },
    "subplot": [3,4],
    "title": ['traffic','elevators'],},],
"_display_names": {
    "elevators": {
        48: "75% density",
        32: "50% density",
        16: "25% density",
        8: "12.5% density", },
    "elevator_selection": {
        'safe': "md-safe",
        'roff': "md-random-offline",
        'opt': "optimistic",
        'ron': "md-random-online",},
    "*": {
        "pir": "Injection rate (packet/node/cycle)",
        "*average_latency": "Avg. latency (cycles)",}}}
```

The directive names are for the most part self-explanatory. The *_force_recompile* direc-

tive is useful if the simulator code was modified and we want it to be recompiled before the simulations are performed. The $_skip_existing$ option makes it possible to avoid repeating simulations for which we have already stored the results.

Each simulation parameter (keys with no underscore), a list of values must be supplied (not a single value). The `run.py` program simulates all the combinations of parameters, unless otherwise specified through the $_except$ directive described below. As can be seen in the configuration file, we also provide some utility functions to generate lists of values, such as $randoms(n)$to generate a list of n random seeds, and $make_range(start, end, step)$ to generate a range of real values.

The $_plot$ directive, combined with $_display_names$ is very versatile, as it makes it possible to control the way the results are displayed (what to use as the x and y axes, what are we comparing, how many subplots in the figure, etc.) using only a few key-value pairs.

B.4.2 The $_except$ directive

Simulating all the possible combinations of parameters can be inconvenient in a number of ways. For instance, some ranges of injection rates may only make sense for some types of traffic. Note that in Chapter 3 - figure 3.4, we do not use the same range of injection rates in all the configurations.

The $_except$ directive makes it possible to exclude some combinations of parameters that are considered "incompatible" from the set of simulations to run. For instance, if, for some reason, we do not want to use 'opt' elevator selection method with bit complement and shuffle traffics, it is enough to add the following to the configuration file:

```
    "_except":[ {"elevator_selection": ["opt"],
                "traffic": ["Shuffle", "BitComplement"],}  ],
```

Note that the value to the $_except$ key is a list, so several exceptions may be defined.

B.4.3 SSH Support: The $_nodes$ directive

The automation tool also implements the distribution of simulations over several computers. To use this feature, all the user has to do is specify the list of nodes (ip addresses or hostnames)

to run the simulations on in the configuration file with the $_nodes$ directive. After running the `run.py` program, the user will be asked for their login information to establish a connection with each computer.

The total set of simulations to run is distributed evenly among the provided list of nodes. It is worth mentioning that if a problem occurs in one of the nodes all the nodes abort their simulations, as it may be caused by a fatal error in the simulator.

Bibliography

Agarwal, N., Krishna, T., Peh, L. S., and Jha, N. K. (2009). GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software*, pages 33–42. IEEE. xi, 94, 96

Akbari, S., Shafiee, A., Fathy, M., and Berangi, R. (2012). AFRA: A low cost high performance reliable routing for 3D mesh NoCs. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 332–337. IEEE. 71

Avresky, D., Chaix, F., and Nicolaidis, M. (2014). Congestion-aware adaptive routing in 2d-mesh multicores. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 50–58. IEEE. 123

Bahmani, M., Sheibanyrad, A., Pétrot, F., Dubois, F., and Durante, P. (2012). A 3D-NoC router implementation exploiting vertically-partially-connected topologies. *Proceedings - 2012 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2012*, pages 9–14. vi, viii, x, 45, 64, 70, 71, 75, 85

Bakhoda, A., Kim, J., and Aamodt, T. M. (2010). Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 421–432. IEEE. 3

Bartzas, A., Skalis, N., Siozios, K., and Soudris, D. (2007). Exploration of alternative topologies for application-specific 3d networks-on-chip. In *Proc. of WASP*. 5, 43

Becker, D. U. and Dally, W. J. (2009). Allocator implementations for network-on-chip routers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. 33

Benini, L. (2008). 3d-mpsocs: architectural and design technology outlook. In *Keynote presentation at 7th International Forum on ApplicationSpecific MultiProcessor SoC*. 5, 43

Boppana, R. V. and Chalasani, S. (1996). A framework for designing deadlock-free wormhole routing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):169–183. 14, 15

Catania, V., Mineo, A., Monteleone, S., Palesi, M., and Patti, D. (2015). Noxim: An open, extensible and cycle-accurate network on chip simulator. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, volume 2015-Septe, pages 162–163. IEEE. 94

Chaix, F., Avresky, D., Zergainoh, N. E., and Nicolaidis, M. (2010). Fault-tolerant deadlock-free adaptive routing for any set of link and node failures in multi-cores systems. In *Proceedings - 2010 9th IEEE International Symposium on Network Computing and Applications, NCA 2010*, pages 52–59. IEEE. 11, 13, 18, 34

Chaix, F., Avresky, D., Zergainoh, N.-E., and Nicolaidis, M. (2011). A fault-tolerant deadlock-free adaptive routing for on chip interconnects. *2011 Design, Automation & Test in Europe*, pages 1–4. 4

Charif, A., Coelho, A., Zergainoh, N. E., and Nicolaidis, M. (2017a). Detailed and highly parallelizable cycle-accurate network-on-chip simulation on GPGPU. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 672–677. IEEE. xi, 7

Charif, A., Coelho, A., Zergainoh, N.-E., and Nicolaidis, M. (2017b). Mini-espada: A low-cost fully adaptive routing mechanism for networks-on-chips. In *Test Symposium (LATS), 2017 18th IEEE Latin American*, pages 1–4. IEEE. vi, 5

Charif, A., Zergainoh, N.-E., Coelho, A., and Nicolaidis, M. (2017c). Rout3d: A lightweight adaptive routing algorithm for tolerating faulty vertical links in 3d-nocs. In *22th IEEE European Test Symposium (ETS'17)*, pages 1–6. ACM IEEE. viii, 6

Charif, A., Zergainoh, N.-E., and Nicolaidis, M. (2016). A new approach to deadlock-free fully adaptive routing for high-performance fault-tolerant NoCs. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 121–126, Storrs, CT, USA. IEEE. vi, 5

Dally, W. (1990). Virtual-channel flow control. In *Proceedings. The 17th Annual International Symposium on Computer Architecture*, volume 3, pages 60–68. IEEE Comput. Soc. Press. 11

Dally, W. J. (1992). Virtual-channel flow control. *IEEE Transactions on Parallel and Dis-*

tributed systems, 3(2):194–205. 4

Dally, W. J. and Aoki, H. (1993). Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475. 14, 28, 122

Dally, W. J. and Seitz, C. L. (1988). Deadlock-free message routing in multiprocessor interconnection networks. vi, 15

Dally, W. J. and Towles, B. (2001). Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE. 3

Davis, W. R., Wilson, J., Mick, S., Xu, J., Hua, H., Mineo, C., Sule, A. M., Steer, M., and Franzon, P. D. (2005). Demystifying 3d ics: The pros and cons of going vertical. *IEEE Design & Test of Computers*, 22(6):498–510. 43

Dimopoulos, M., Gang, Y., Anghel, L., Benabdenbi, M., Zergainoh, N.-E., and Nicolaidis, M. (2014). Fault-tolerant adaptive routing under an unconstrained set of node and link failures for many-core systems-on-chip. *Microprocessors and Microsystems*, 38(6):620–635. 123

Duato, J. (1995). A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067. vii, 12, 14, 27, 31

Duato, J. (1997). A theory of fault-tolerant routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):790–802. 14, 31

Dubois, F., Sheibanyrad, A., Petrot, F., and Bahmani, M. (2013). Elevator-First: A Deadlock-Free Distributed Routing Algorithm for Vertically Partially Connected 3D-NoCs. *IEEE Transactions on Computers*, 62(3):609–615. 5, 11, 13, 43, 44, 45, 46, 47, 71, 73

Ebrahimi, M., Chang, X., Daneshtalab, M., Plosila, J., Liljeberg, P., and Tenhunen, H. (2013a). DyXYZ: Fully adaptive routing algorithm for 3D NoCs. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013*, pages 499–503. IEEE. 4, 15

Ebrahimi, M. and Daneshtalab, M. (2017). Ebda: A new theory on design and verification of deadlock-free interconnection networks. In *ISCA*, pages 1–13. vi, 15, 17, 73

Ebrahimi, M., Daneshtalab, M., Liljeberg, P., and Tenhunen, H. (2013b). Fault-tolerant method with distributed monitoring and management technique for 3D stacked meshes. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)*,

pages 93–98. IEEE. 71

Ebrahimi, M., Daneshtalab, M., Plosila, J., and Tenhunen, H. (2012). MAFA: Adaptive fault-tolerant routing algorithm for networks-on-chip. In *Proceedings - 15th Euromicro Conference on Digital System Design, DSD 2012*, pages 201–207. IEEE. 13, 34

Eggenberger, M. and Radetzki, M. (2013). Scalable parallel simulation of networks on chip. *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, pages 1–8. xi, 94, 95, 97

Eggenberger, M., Strobel, M., and Radetzki, M. (2016). Globally Asynchronous Locally Synchronous Simulation of NoCs on Many-Core Architectures. In *Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016*, pages 763–770. IEEE. xi, 95

Eghbal, A., Yaghini, P. M., Bagherzadeh, N., and Khayambashi, M. (2015). Analytical fault tolerance assessment and metrics for tsv-based 3d network-on-chip. *IEEE Transactions on Computers*, 64(12):3591–3604. 47

Feero, B. S. and Pande, P. P. (2009). Networks-on-chip in a three-dimensional environment: A performance evaluation. *IEEE Transactions on computers*, 58(1):32–45. 5, 43

Fleury, E. and Fraigniaud, P. (1998). A general theory for deadlock avoidance in wormhole-routed networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):626–638. 12, 14

Flich, J. and Duato, J. (2008). Logic-based distributed routing for nocs. *IEEE Computer Architecture Letters*, 7(1):13–16. 72

Foroutan, S., Sheibanyrad, A., and Petrot, F. (2014). Assignment of vertical-links to routers in vertically-partially-connected 3-d-nocs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(8):1208–1218. 46

Glass, C. and Ni, L. (1992). The Turn Model for Adaptive Routing. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, volume 11, pages 278–287, New York, NY, USA. IEEE. 13, 17, 54, 71

Gratz, P., Grot, B., and Keckler, S. W. (2008). Regional congestion awareness for load balance in networks-on-chip. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 203–214. IEEE. 122

Jiang, N., Balfour, J., Becker, D. U., Towles, B., Shaw, D. E., Dally, W. J., Michelogiannakis,

G., and Kim, J. (2013). A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96. xi, 93, 94, 96, 108

Kim, J., Park, D., Theocharides, T., Vijaykrishnan, N., and Das, C. R. (2005). A low latency router supporting adaptivity for on-chip interconnects. In *Proceedings of the 42nd annual Design Automation Conference*, pages 559–564. ACM. 122

Kumar, M., Laxmi, V., Gaur, M. S., Daneshtalab, M., Ebrahimi, M., and Zwolinski, M. (2014). Fault tolerant and highly adaptive routing for 2D NoCs. In *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 104–109, San Francisco, CA. IEEE. 14, 31, 32

Lee, J. and Choi, K. (2013). A deadlock-free routing algorithm requiring no virtual channel on 3D-NoCs with partial vertical connections. *2013 7th IEEE/ACM International Symposium on Networks-on-Chip, NoCS 2013*, pages 0–1. 70, 72

Lee, J., Kang, K., and Choi, K. (2015). Redelf: An energy-efficient deadlock-free routing for 3d nocs with partial vertical connections. *J. Emerg. Technol. Comput. Syst.*, 12(3):26:1–26:22. viii, 5

Mullins, R. (2009). Netmaker. [Online; accessed 03-July-2017]. 7, 85, 108

Nangate (2017). Nangate open cell library 45nm. [Online; accessed 03-July-2017]. 7, 64, 85

Niazmand, B., Azad, S. P., Flich, J., Raik, J., Jervan, G., and Hollstein, T. (2016). Logic-based implementation of fault-tolerant routing in 3D network-on-chips. In *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE. x, 45, 55, 72, 73

NVIDIA (2017). Cuda toolkit. 100, 106

Pasricha, S. and Zou, Y. (2011). A low overhead fault tolerant routing scheme for 3D networks-on-chip. *Proceedings of the 12th International Symposium on Quality Electronic Design, ISQED 2011*, pages 204–211. 71

Pavlidis, V. F. and Friedman, E. G. (2007). 3-d topologies for networks-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1081–1090. 5, 43

Pétrot, F., Hommais, D., and Greiner, A. (1997). Cycle precise core based hardware/software system simulation with predictable event propagation. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 182–187. IEEE. 98

Pinkston, T. and Warnakulasuriya, S. (1999). Characterization of deadlocks in k-ary n-cube networks. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):904–921. 12, 13

Pinto, C., Raghav, S., Marongiu, A., Ruggiero, M., Atienza, D., and Benini, L. (2011). GPGPU-accelerated parallel and fast simulation of thousand-core platforms. In *Proceedings - 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, pages 53–62. IEEE. 94, 95

Ren, P., Lis, M., Cho, M. H., Shim, K. S., Fletcher, C. W., Khan, O., Zheng, N., and Devadas, S. (2012). HORNET: A cycle-level multicore simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):890–903. xi

Ren, P., Meng, Q., Ren, X., and Zheng, N. (2014). Fault-tolerant Routing for On-chip Network Without Using Virtual Channels. *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*, pages 1–6. 93, 94

Salamat, R., Ebrahimi, M., and Bagherzadeh, N. (2015). An Adaptive, Low Restrictive and Fault Resilient Routing Algorithm for 3D Network-on-Chip. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 392–395. IEEE. 43, 44, 70, 72

Salamat, R., Ebrahimi, M., Bagherzadeh, N., and Verbeek, F. (2016a). CoBRA: Low cost compensation of TSV failures in 3D-NoC. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 115–120. IEEE. 72

Salamat, R., Khayambashi, M., Ebrahimi, M., and Bagherzadeh, N. (2016b). A Resilient Routing Algorithm with Formal Reliability Analysis for Partially Connected 3D-NoCs. *IEEE Transactions on Computers*, 13(9):1–1. vi, viii, ix, x, 5, 13, 44, 72, 79

Schwiebert, L. and Jayasimha, D. (1995). A universal proof technique for deadlock-free routing in interconnection networks. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 175–184. ACM. 12, 15

Schwiebert, L. and Jayasimha, D. N. (1993). Optimal fully adaptive wormhole routing for meshes. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 782–791. ACM. x, 31

Taktak, S., Desbarbieux, J.-L., and Encrenaz, E. (2008). A tool for automatic detection of deadlock in wormhole networks on chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(1):6. 14

Verbeek, F. and Schmaltz, J. (2011). On Necessary and Sufficient Conditions for Deadlock-Free Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 22(12):2022–2032. 14

Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown III, J. F., and Agarwal, A. (2007). On-chip interconnection architecture of the tile processor. *IEEE micro*, 27(5):15–31. 3

Xiao, S. and Feng, W. C. (2010). Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1–12. IEEE. 104

Ying, H., Hofmann, K., and Hollstein, T. (2014). Dynamic quadrant partitioning adaptive routing algorithm for irregular reduced vertical link density topology 3-Dimensional Network-on-Chips. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 516–522. IEEE. 43, 44, 72, 79

Ying, H., Jaiswal, A., Hollstein, T., and Hofmann, K. (2013). Deadlock-free generic routing algorithms for 3-dimensional networks-on-chip with reduced vertical link density topologies. *Journal of Systems Architecture*, 59(7):528–542. 44

Zolghadr, M., Mirhosseini, K., Gorgin, S., and Nayebi, A. (2011). GPU-based NoC simulator. In *9th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011*, pages 83–88. IEEE. xi, 94, 95, 96