THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Amaury GRAILLAT

Thèse dirigée par **Pascal RAYMOND**
et codirigée par **Matthieu MOY**, Université Claude Bernard Lyon 1

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Génération de code pour un many-core avec des contraintes temps réel fortes

Code Generation for Multi-Core Processor with Hard Real-Time Constraints

Thèse soutenue publiquement le **16 novembre 2018**,
devant le jury composé de :

Monsieur PASCAL RAYMOND
CHARGE DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur MATTHIEU MOY
MAITRE DE CONFERENCES, UNIVERSITE LYON 1, Co-directeur de thèse
Monsieur JAN REINEKE
PROFESSEUR, UNIVERSITE DE LA SARRE - ALLEMAGNE, Rapporteur
Monsieur ROBERT DE SIMONE
DIRECTEUR DE RECHERCHE, INRIA CENTRE S. ANTIPOLIS - MEDITERRANEE, Rapporteur
Madame ANNE BOUILLARD
MAITRE DE CONFERENCES, ECOLE NORMALE SUPERIEURE DE PARIS, Examinateur
Monsieur BENOÎT DUPONT DE DINECHIN
INGENIEUR DE RECHERCHE, KALRAY S.A. - MONTBONNOT SAINT-MARTIN, Examinateur
Monsieur ALAIN GIRAULT
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Président
Madame CHRISTINE ROCHANGE
PROFESSEUR, UNIVERSITE TOULOUSE-IIIIII PAUL-SABATIER, Examinateur

Contents

Remerciements

Je voudrais remercier les personnes qui ont été à mes côtés durant ces 3 ans.

En premier lieu, je remercie mes directeurs de thèse, Matthieu et Pascal du laboratoire Verimag qui m'ont beaucoup appris et guidé ainsi que Claire pour ses conseils et son enthousiasme. Je remercie aussi vivement Benoît qui m'a proposé cette thèse chez Kalray et m'a transmis ses connaissances à la fois théoriques et industrielles. Je remercie le jury et les rapporteurs pour avoir relu ce manuscrit.

En second lieu, je remercie les collègues de Kalray, notamment Vincent et l'équipe hardware pour leur connaissance du processeur au cycle près. Merci à la BU Automotive, et aux thésards de Kalray anciens et nouveaux, Julien, grand druide du débug et de l'optimisation, et Cyril.

Je voudrais remercier les autres thésards/docteurs de l'écosystème CAPACITES avec qui il a été agréable de travailler: Hamza et Moustapha (que je n'oublie pas de citer cette fois). Merci aux amis du labo, de la coinche et du Kubb: Denis, Anaïs, Maxime et Thomas.

Enfin, je remercie tous ceux avec qui j'ai passé les soirs et les weekends. Merci à ma famille qui m'a soutenu: notamment mes parents, mon frère, mes grands-parents et Manon. Merci aux amis: Stéphane pour nos discussions jusqu'à tard le soir, Lou pour risquer fréquemment sa vie pour nous offrir du gâteau au caramel, et les pharmaciens.

Merci à l'association Théâtre à Grenoble INP pour le concentré de bonne humeur qu'elle a su générer et Florian de DAF. Merci à ceux que j'oublie.

Je n'oublie pas Skippy, Charade, Larzac, Béa, Billy, les mouchillons et les oiseaux de l'ORTF.

1 Résumé

1.1 Introduction

1.1.1 Applications temps-réel dur

Des millions de lignes de code [32] font tourner les objets que nous utilisons dans la vie de tous les jours. Ces logiciels améliorent le confort et la sûreté des avions, facilitent la conduite des voitures, réduisent la consommation énergétique ou augmentent la productivité des centrales énergétiques. Ces logiciels influencent directement notre sécurité, à la fois parce qu'ils améliorent la sécurité des systèmes et parce qu'une erreur dans leur code peut être dramatique.

Les systèmes réactifs [67] réagissent à leurs entrées à une vitesse définie par l'environnement. Par exemple, le système anti-collision d'un voiture freine lorsque la distance à l'obstacle est trop courte. En aéronautique, le système de commande de vol réagit en permanence à son environnement et en fonction des commandes du pilote. Ainsi, une réponse trop tardive à une commande ou à un changement de l'environnement est une erreur critique. On appelle temps-réel critique ou temps-réel dur un logiciel qui doit garantir des bornes sur son temps de réaction. Ces contraintes de temps sont spécifiées sous forme de latences, de délais ou de bandes passantes.

1.1.2 L'impact des évolutions matérielles sur les logiciels temps-réel

Mono-cœur. De nombreuses recherches on été menées dans le but de calculer une borne supérieure des temps de calcul sur des processeurs mono-cœurs. L'analyse de pire temps d'exécution (*worst-case execution time*, WCET) [8, 126] en est un exemple. La durée d'exécution d'un programme dépend du matériel et donc l'analyse requiert un modèle fidèle de celui-ci.

En général, l'analyse de WCET n'est pas exacte : pour être fiable, elle doit sur-estimer la durée réelle d'exécution. La précision du calcul du WCET dépend du matériel. Par exemple des processeurs très simples qui exécutent les instructions dans l'ordre et qui n'ont pas de prédiction de branchement autorisent un analyse très fine. Les comportements dynamiques sont moins prédictibles. Un cache mémoire est aussi un source de non déterminisme si sa politique d'éviction est non prédictible. En conclusion, l'analyse temporelle est plus facile sur du matériel qui autorise la composition des temps d'exécution (*time-compositional* [127] *hardware*).

La précision de l'analyse du matériel peut parfois être contrôlée par le logiciel : certains processeurs proposent des instructions pour limiter l'indéterminisme ; par exemple en désactivant les caches mémoires. Les modèles d'exécution spécifient la manière dont les programmes doivent être exécutés. Certains modèles d'exécution restreignent l'utilisation du processeur en désactivant tous les mécanismes qui ne sont pas prédictibles.

Des mono-cœurs aux multi-cœurs. Aujourd'hui, nous observons deux tendances : la production des mono-cœurs diminue et les logiciels embarqués sont de plus en plus complexes et requièrent de plus en plus de puissance de calcul [114].

Par conséquent, le multi-cœur est à la fois un besoin et une solution. Le multi-cœur sur puce (MPSoC) est une solution pour intégrer plus de fonctionnalités dans le même circuit réduisant ainsi le poids et la consommation des systèmes. Les pluri-cœurs (*many-core*) sur puce ont été inventés pour permettre l'intégration de plus de systèmes sur la même puce. Ils sont composés de dizaines, voire de centaines de cœurs. Il disposent d'un réseau sur puce (NoC) pour minimiser les coûts de communication.

L'isolation temporelle permet d'intégrer plusieurs applications dans le même processeur mono-cœur. Les applications sont isolées car elles ne s'exécutent pas en même temps. De plus, si l'état des registres est parfaitement sauvegardé et restauré lors du changement de contexte, alors l'isolation est parfaite et le WCET de chaque partie peut être calculé séparément.

Pour les multi- et les pluri-cœurs, l'isolation peut être basée sur le temps, l'espace ou les deux. L'isolation spatiale consiste à exécuter des processus sur des cœurs différents, utilisant des espaces mémoires différents. Néanmoins, un compromis doit être trouvé entre l'isolation et la communication. En d'autres termes, l'isolation n'est pas parfaite et les différents cœurs ont besoin de communiquer en utilisant des ressources partagées. Ainsi, le comportement d'un processus a un impact sur les autres.

Par exemple, l'accès mémoire d'un cœur peut être retardé par l'accès concurrent d'un autre cœur à la même ressource. Ces accès concurrents sont des interférences et ne sont pas limités à la mémoire : les bus, les réseaux sur puce (NoC), les ports d'entrées/sorties, les caches partagés soufrent aussi des interférences.

La cohérence de cache est un mécanisme assurant que, pour une adresse mémoire donnée, la donnée est la même dans le cache de tous les cœurs. Par exemple, un accès mémoire peut être provoqué même s'il n'y a pas de défaut de cache si la valeur présente dans le cache n'est pas à jour. Le principal inconvénient est que du point de vue d'un cœur, le temps d'accès n'est pas prévisible puisqu'il dépend des accès des autres cœurs à la même ligne de cache. Lorsque la cohérence de cache est implémentée par logiciel, la mise à jour du contenu du cache est explicite ce qui rend l'analyse temporelle plus facile.

Les multi-cœurs dans l'industrie critique. Dans l'industrie aéronautique, le CAST-32A est un document qui explique le point de vue des autorités de certification sur l'usage des multi-cœurs [48].

L'utilisation de multi-cœurs pour les systèmes aéronautiques requiert l'énumération exhaustive et la compréhension complète des sources d'interférences. Le CAST-32A décrit le partitionnement robuste comme une propriété du système où chaque cœur utilise uniquement les ressources qui lui ont été assignées. L'utilisation des ressources par un cœur doit se faire dans le respect des contraintes temporelles des autres cœurs. Les erreurs à l'exécution qui sont spécifiques aux fonctionnalités du multi-cœur doivent être détectées et prises en compte.

1.1.3 L'usage des multi- et pluri-cœurs pour des applications critiques

La recherche sur l'analyse du WCET sur multi-cœur est très active. Des travaux traitent de l'analyse d'interférences [43, 113, 120, 81], de la programmation de ces multi-cœurs pour minimiser les interférences [105, 82, 104, 10, 47] ou l'ordonnancement des tâches prenant en compte ces interférences. Des travaux proposent des modèles d'exécution isolant totalement les processus en supprimant les interférences [103, 96, 47]. D'autres, comme notre travail, minimisent les interférences et rendent leur analyse possible [113, 119]. Dans notre travail, nous cherchons un compromis entre les performances et l'isolation lors de l'analyse et de la configuration du matériel.

Développer des applications temps-réel dur sur les multi-cœurs pour tirer parti du parallélisme d'exécution tout en minimisant les interférences est un problème d'optimisation multi-critère. Cela nécessite de créer des tâches parallèles et de les déployer sur les cœurs et dans la mémoire. La configuration fine du processeur est aussi nécessaire pour minimiser l'impact des comportements dynamiques du matériel. Par exemple, certains processeurs disposent de cohérence de cache logicielle ce qui rend le processeur plus prédictible mais sa programmation plus complexe.

En conclusion, l'utilisation des multi-cœurs pour les applications temps-réel nécessite des logiciels complexes. Cette complexité rend les erreurs de programmation courantes. Une solution est la génération de code depuis un langage haut niveau afin de faciliter l'analyse temporelle [27, 26, 82, 74]. Notre travail en fait parti.

1.1.4 La programmation par modèle et les langages synchrones

Les langages synchrones ont été conçus pour faciliter le développement des systèmes réactifs critiques. Ils fournissent une notion de temps abstrait qui facilite la vérification des contraintes de temps. Lustre [68] est un langage synchrone flot de données et Scade [15] est sa version industrielle utilisée par l'industrie avionique.

Les langages synchrones sont des langages haut niveau qui abstraient l'implémentation système. Certains langages synchrones disposent d'une syntaxe graphique facilitant le développement des applications complexes. Il existes des langages similaires non-synchrones tels que Simulink qui permettent de décrire la fonctionnalité de l'application tout en cachant au développeur les problèmes liés à l'implémentation système.

Le développement par modèle sur multi- et pluri-cœurs est basé sur la génération automatique de code depuis la spécification de haut niveau. Les tâches parallèles doivent aussi être extraites automatiquement tout en assurant les critères de performance et de conservation de la sémantique.

1.1.5 Résumé des contributions : les programmes synchrones temps-réel dur sur pluri-cœur

Dans cette thèse, nous proposons une méthode complète de génération automatique de code temps-réel critique depuis les langages synchrones Scade ou Lustre vers une cible multi-/pluri-cœur.

Nous appliquons cette méthode au pluri-cœur MPPA2 de Kalray. Néanmoins, notre but est de tirer un enseignement plus large de ce travail pour permettre d'appliquer la méthode à toute une classe de multi-cœurs.

Premièrement, nous définissons une méthode pour extraire des tâches parallèles depuis un programme synchrone. Deuxièmement, nous générons une configuration automatique de la machine pour la rendre prédictible temporellement. La mémoire et le réseau sur puce (NoC) sont configurés afin de maîtriser les interférences. En particulier, nous implémentons un outil de configuration du NoC qui permet des communications sans interbloquage et le calcul de la latence de bout en bout.

Enfin, notre outil génère le code système nécessaire à l'implémentation des tâches de communication, la synchronisation et l'isolation spatiale. Un binaire est généré par assemblage du code fonctionnel, du code système et de la configuration du processeur.

La méthode de génération de code et les expériences on été publiées dans [63] et la méthode générale a été soumise pour publication dans [64]. La méthode de routage sur le NoC, sa configuration et les méthodes de calcul de latence on été publiées dans [21, 46, 45].

Nous présentons maintenant la méthode générale.

1.2 Méthode générale

Nous expliquons la méthode globale pour l'implémentation parallèle d'un programme synchrone flot de données sur processeurs pluri-cœurs.

La *Représentation Intermédiaire Parallèle* (PIR) décrit le programme comme un ensemble d'entités communiquant entre elles. Le PIR sépare le problème en deux : l'extraction du parallélisme depuis le programme d'entrée et l'implémentation du PIR sur un pluri-cœur.

1.2.1 Définition de la Représentation Intermédiaire Parallèle

Le PIR doit être assez précis pour rendre l'implémentation du programme possible mais suffisamment abstrait pour permettre le raisonnement.

Le PIR naturel d'un programme flot de données est basé sur la division en nœuds. Il ne décrit pas la fonctionnalité du programme mais contient un ensemble de nœuds communiquant ensemble.

Le PIR est composé d'un graphe acyclique décrivant les dépendances entre les tâches. Les langages synchrones permettent les communications directes ou les communications avec délais (opérateur `pre`). Seules les dépendances directes sont décrites par ce graphe. Le graphe est acyclique et décrit un ordre partiel entre les nœuds sous la forme de contraintes de précédence.

Le graphe de dépendances décrit les canaux de communications entre les nœuds. Il peut être cyclique si l'une des communications formant le cycle a un délai. Chaque canal décrit la taille de la communication ainsi que le nombre de délais.

Le graphe de dépendances est utilisé pour ordonnancer les tâches et générer les synchronisations tandis que le graphe des communications est nécessaire pour implémenter les canaux de communications.

1.2.2 Extraction des tâches parallèles

Le but de l'étape [**Extraction du parallélisme**] est d'extraire les tâches depuis le programme afin de générer le PIR. Notre méthode extrait les tâches depuis le nœud de plus haut niveau dans la hiérarchie du programme. Un tâche est créée pour chaque sous-nœud. Ces tâches sont candidates pour être exécutées en parallèle. Les tâches, les dépendances et les communications sont obtenues par analyse syntaxique du programme.

Cette méthode est similaire à celle employée par les langages de description d'architecture (ADL) tels que Prélude ou Giotto.

Le code fonctionnel de l'application est obtenu en compilant chaque nœud avec un compilateur Lustre ou Scade standard.

1.2.3 Choix d'implémentations pour tirer parti des bancs mémoire

Kalray MPPA2. Le MPPA2 de Kalray est un pluri-cœur composé de 16 clusters. Chaque clusters s'apparente à un multi-cœur de 16 cœurs et d'une mémoire partagée. Cette mémoire partagée est composée de 16 mémoires indépendantes appelées bancs. Les clusters sont reliés entre eux par un réseau sur puce (NoC) composé de liens et de routeurs. Chaque cluster est un nœud de ce réseau et peut écrire dans la mémoire d'un autre cluster. Chaque communication NoC peut se faire à travers un limiteur de bande passante matériel.

Dans cette section, nous présentons les problèmes et les choix techniques liés à la mémoire partagée.

Communication : lecture distante ou écriture distante. Nous considérons des plateformes avec une mémoire distribuée ou une mémoire composée de plusieurs bancs.

Un banc est associé à chaque cœur. Nous parlons *d'écriture distante* lorsqu'une donnée est copiée par un cœur vers la mémoire d'un autre cœur. À l'inverse, nous parlons de *lecture distante* lorsqu'une donnée est lue par un cœur dans une mémoire distante.

Dans notre travail, nous choisissons l'écriture distance puisqu'elle permet d'être cohérent avec le fonctionnement du NoC du MPPA2 qui donne un accès direct en écriture aux mémoires distantes.

Modèle d'exécution dirigé par le temps. Les nœuds des programmes flot de données sont exécutés lorsque leurs entrées sont disponibles. Pour reproduire le fonctionnement flot de données sur un multi-cœur, on peut utiliser des synchronisations. Dans ce cas, les tâches sont exécutées sur l'occurrence d'un évènement. Une autre manière est de programmer le démarrage de chaque tâche sur des dates fixées pour lesquelles on est certain que les entrées sont disponibles. Dans ce cas on parle d'exécution dirigée par le temps.

Nous analysons les interférences pour chaque tâche. Deux tâches peuvent interférer lorsqu'elles utilisent simultanément le même banc mémoire. Le pire temps d'exécution d'une tâche (WCET) est calculé sans prendre en compte les interférences. Ainsi, la durée d'exécution réelle peut être supérieure au WCET s'il y a des interférences. Pour cela on calcule le pire temps de réaction (WCRT) qui prend en compte les interférences.

Si les tâches d'un programme démarrent dès que possible, il est difficile de calculer un WCRT précis puisqu'un changement dans la durée d'exécution d'une tâche peut rendre concurrentes des tâches qui ne l'étaient pas. Pire, un temps d'exécution plus court pour une tâche peut mener à un temps d'exécution globalement plus long.

Ainsi, nous choisissons une exécution dirigée par le temps pour laquelle chaque tâche dispose d'une date de démarrage définie statiquement. Les tâches ne peuvent pas démarrer plus tôt que leur date de démarrage. Cela permet d'éviter les problèmes dus à la durée d'exécution des tâches.

Rihani *et al.* [113] introduisent un modèle d'exécution dirigé par le temps et un outil capable de calculer les dates de démarrage de chaque tâche en prenant en compte les interférences et les dépendances. Nous utilisons cet outil pour permettre l'exécution dirigée par le temps.

1.2.4 Implémentation du PIR sur un pluri-cœur

Une fois le PIR extrait, il peut être implémenté sur la plateforme. Son implémentation doit conserver la sémantique du programme d'entrée et satisfaire les contraintes de temps. La génération de code est découpée en différentes étapes que nous décrivons maintenant.

Déploiement et ordonnancement statique. L'étape [**Mapping+Scheduling**] (placement et ordonnancement) est basée sur un outil externe pour trouver un placement optimisé des tâches sur les cœurs et les clusters du processeur. Puis, l'outil calcule un ordonnancement compatible avec le graphe de dépendances.

Il peut choisir d'optimiser la durée du chemin critique du programme. Cela nécessite de connaitre la durée d'exécution de chaque tâche. Néanmoins, ces temps d'exécutions dépendent des interférences qui dépendent à leur tour du placement.

Cette interdépendance entre le placement et la durée d'exécution peut être vu comme un problème avec un point fixe. Nous choisissons de casser la boucle en utilisant le WCET comme durée d'exécution de chaque tâche. Par conséquent, l'outil de placement et d'ordonnancement requiert le graphe de dépendances et le WCET en isolation de chaque tâche.

L'ordonnancement est statique et non préemptif. Cela évite les problèmes dus à la préemption et simplifie le calcul du WCRT.

Un banc mémoire est associé à chaque cœur. Le code, les données et les tampons de communication sont placés sur le banc correspondant au cœur. Par conséquent, chaque cœur accède exclusivement à son propre banc, sauf quand il communique. Ce principe est réalisé par l'étape [**Code and Buffer Allocation**].

Date de démarrage des tâches et exécutable final. L'étape [**WCET Analysis**] a pour but de calculer le WCET de chaque tâche en isolation. Les outils d'analyse de WCET OTAWA [8] et AiT [50] sont compatibles avec le MPPA2.

L'étape [**MIA: Release date computation**] correspond à l'outil MIA [113] qui calcule les dates de démarrage des tâches en prenant en compte la précédence et les interférences.

L'étape [**Generation of system + communication code**] (génération du code système et du code de communication) génère le code pour démarrer les cœurs, le code pour implémenter l'ordonnancement statique et les communications. Pour l'implémentation dirigée par le temps, cette étape nécessite les dates de démarrage des tâches. Cependant, puisque la compilation du binaire a un impact sur la durée d'exécution du programme, le WCET des tâches est calculé sur le binaire final. Mais ce binaire final nécessite les dates de démarrage des tâches.

Ainsi, nous avons choisi d'intégrer les dates de démarrage en modifiant le binaire final.

Routage et pire temps de traversée NoC. Lorsque des tâches de différents clusters communiquent, le canal de communication passe par le NoC. Cette communication à travers le NoC est appelée *flux*.

Nous choisissons de configurer le NoC avec des routes statiques et un limiteur de bande passante pour chaque flux. Par conséquent, la configuration du NoC nécessite deux paramètres pour chaque flux : la route et la bande passante. Cette configuration facilite le calcul du pire temps de traversée du NoC appelé *WCTT*.

Le WCTT est nécessaire pour l'étape [**MIA: Release date computation**].

L'étape [**NoC Routing**] calcule l'ensemble des routes possibles pour chaque flux. Pour cela, il est nécessaire d'utiliser des algorithmes de routage garantissant l'absence d'interbloquage. Nous comparons différents algorithmes de routage point à point sur le MPPA2 et nous présentons un algorithme de routage 1-N (multicast) qui minimise le nombre de ressources NoC nécessaires à son implémentation.

L'étape [**Route selection & fair rate attribution**] sélectionne une seule route par flux. Pour que la bande passante de chaque lien du réseau soit distribuée équitablement entre les flux, nous utilisons le critère de *max-min fairness* [33]. Lorsque les bandes passantes des flux respectent ce critère, l'augmentation de la bande passante d'un flux ne peut pas se faire sans réduire la bande passante d'un flux déjà inférieur. Nous présentons un algorithme fournissant une solution optimale et une heuristique basée sur de la programmation linéaire pour implémenter ce critère.

L'étape [**NoC Configuration generator**] génère l'entête des paquets contenant la route sélectionnée. Il génère aussi la configuration du limiteur de bande passante.

L'étape [**WCTT Network Calculus**] utilise la théorie du Network Calculs pour calculer le WCTT de chaque flux.

1.3 Conclusion

Nous proposons un ensemble d'outils permettant la génération de code depuis un programme synchrone flot de données vers un processeur multi- ou pluri-cœur. Ces travaux ciblent des applications critiques et temps-réel dur qui nécessitent la correction du fonctionnement. En particulier, le code généré doit permettre la validation du logiciel et la traçabilité du code. L'objectif final est de garantir des contraintes de temps exprimées sous forme de bandes passantes minimales ou de latences. L'implémentation est réalisée sur le Kalray MPPA2. Les résultats des expériences sont prometteurs puisqu'ils montrent une accélération importante due à la parallélisation tout en offrant des garanties sur le temps d'exécution.

2 Introduction

2.1 Hard-Real-Time Applications

Millions of lines of code [32] are involved in our everyday life to improve comfort in planes, ease the driving of cars, lessen energy consumption and raise the energy production of plants. Most of these lines of code are critical and directly related to user safety: either because their purpose is to improve safety, or because an unexpected software behavior can endanger people.

Reactive systems [67] react to external events or values at a speed defined by the environment. For instance, the collision avoidance system in cars triggers an emergency brake if the obstacle distance is too short. In avionics, control systems are constantly reacting to changes in the environment to maintain the pilot command. A too late response to events or environment changes leads to unexpected behavior. Furthermore, we name time-critical or hard real-time, softwares that must guarantee bounds for the computation time. Timing constraints such as latency, delays or bandwidth of the systems are part of the specification of the system.

2.2 Impact of Hardware Evolution on Time-Critical Software

2.2.1 Single-Core

Extensive research has been carried out to formally assess bounds on the computation time of a program on single-core processors. The worst-case execution time (WCET) analysis [8, 126] is a general solution to compute an upper bound on the execution time of a program under the worst conditions. Indeed this duration depends on the hardware. Therefore an accurate model of the hardware behavior is required.

In general, the WCET analysis cannot be exact, therefore to be sound it must be overestimated. Some cores provide a tighter overestimation than others. This is the case of processors with simple pipeline compared to processors with an out-of-order pipeline or branch prediction. Dynamic behavior makes predictability harder. A cache is also a source of indeterminism if its eviction policy is not predictable. Finally, time-compositional [127] hardware makes the timing analysis easier.

The software has an impact also on the accuracy of the analysis. An execution model specifies the way programs are executed. Some execution models define a restricted usage of the processor ensuring time-predictability by disabling some unpredictable processor behaviors.

2.2.2 Single-Core vs. Multi-/Many-Core

Today, we observe a twofold trend. Single-core processors are less and less produced and the software is more and more complex in embedded systems leading to an increasing demand for computing power [114].

Consequently, multi-core is seen both as a solution and a necessity. Multi-core system-on-chip (SoC) can be used either as a way to increase computing power, or a way to integrate more functionality in the same piece of hardware; thus reducing weight, space and power consumption of systems. The many-core system-on-chips (SoC) has been invented in anticipation of this integration of several systems and functions in the same processors. Many-cores embed tens to hundreds of cores in the same SoC and provide optimized bus or network-on-chip (NoC) to minimize communication cost.

Integrating several functions in the same single-core processor is done using time isolation, meaning that the different isolated processes are not executed at the same time. Furthermore, if the state of the hardware is exactly saved and restored at each process switch, this isolation is perfect and the computation of the WCET of each piece of software can be done separately.

For multi- and many-core processors their isolation can be based on time, on space or both. The spatial isolation consists in executing different processes on different cores or making them use different memory spaces. Nevertheless, multi- and many-core processors offer a trade-off between the ability of the process to communicate and their isolation. In other words, since the spatial isolation is not perfect and since the inter-process communication is required, processes have an impact on each other.

Resource sharing has an impact on the processes execution time. For instance, memory access of one core can be delayed due to a concurrent access of another core. Such concurrent access is called interference. The interference is not limited to memory: bus, network-on-chip, input/output ports, shared caches, and other shared resources suffer from interference.

Coherent caches ensure that for the same address, the value is the same in the memory and in the caches of the other cores. For instance, a cache hit can lead to a memory access if the value is not up to date. The main drawback is that from the point of view of one core, the access time is unpredictable since it depends on the access of the other cores to the same cache line. For software-based coherent caches, an update of cache contents is explicitly triggered by software thus making the access time more predictable.

In the avionics industry, the CAST-32A document explains the position of certification authorities about multi-cores [48]. Using multi-cores for avionic systems requires understanding completely the sources of interference. The document gives a definition of robust partitioning where a core use only the resources assigned to him and the resource usage of the other cores has no impact on the deadline of this core. The runtime errors specific to the multi-cores features must be detected and handled.

2.2.3 Multi-/Many-Core for Time-Critical Applications

There is an active research on the analysis of WCET for multi-core including the interferences [43, 113, 120, 81], on programming multi-core in a way that minimizes interferences [105, 82, 104, 10, 47] or on tasks scheduling taking into account the interferences. Some work design execution models whose purpose is to isolate processes. Some execution models totally isolate processes by preventing interferences [103, 96, 47]. Others minimize interferences and make the impact of concurrency of execution time analyzable [113, 119]. Our work belongs to this category where a fine analysis and configurations of the hardware are performed to trade between performance and isolation.

Developing time-critical applications on multi-/many-cores that take advantage of the execution parallelism while minimizing interferences requires to solve a multi-criteria optimization problem. This requires creating parallel tasks and mapping them on the core and the memory. This also requires a fine configuration of the processor to minimize the impact of non-predictable mechanisms. For instance, some processors offer software-based cache coherency that is more predictable at the cost of more complex and error-prone software.

We can conclude that using a multi-core for time-critical applications leads to complex software and can be error-prone. One potential solution is the code generation from a high-level program. There is an active research in time-predictable software code generation for multi-core [27, 26, 82, 74] which enables timing analysis methods. Our work belongs to this category.

2.3 Model-Based Development and Synchronous Languages

Synchronous languages have been invented to ease the development of critical reactive systems. They provide an abstract notion of time designed to ease the verification of the timing constraints. Lustre [68] is a data-flow Synchronous language and Scade [15] the industrial version of Lustre used by the avionics industry.

Synchronous languages are high level languages that abstract the implementation from the developer view. Some Synchronous languages are provided with a graphical syntax easing the development of complex applications. There are similar languages such as Simulink which enable to describe the functionality of the application while abstracting the processor and system problems.

The requirements of model-based development is twofold. First, it requires code generation for complex processor from the functional code. Second, it needs an automatic parallelism extraction which ensures both performances and semantics preservation.

2.4 Contributions Overview: Time-Critical Synchronous Program on a Many-core

In this thesis, we propose a complete method to automatically generate time-critical parallel code from Lustre and Scade to a multi-/many-core processor. We apply our method on the Kalray MPPA2 many-core architecture, nevertheless, the purpose of our work is to learn a wider lesson that could be applied other multi-core processors.

First, we define a method to extract parallel tasks from the Synchronous program. Second, our work performs an automatic configuration of the machine in a mode enabling time-predictability. The memory and the network-on-chip are configured to minimize interferences. In particular, we implement a network-on-chip configuration tool to enable end-to-end latency computation and deadlock-free communication.

Then, our tool generates the system code used to implement task communications, synchronizations and spatial isolation. Finally, a binary is generated by assembling the functional code, the system code and the processor configuration.

2.5 Organization of This Document

The first chapters of the thesis provide the basis required to understand our approach. Chapter 3 describes the principles and motivations for the model-based design from data-flow Synchronous languages and we compare them to other parallel languages for reactive systems. Chapter 4 presents the main timing problems of multi-core and presents existing methods to cope with them. We present some existing architectures and in particular the Kalray MPPA2.

The general method of our work is exposed in Chapter 5. This chapter can be used as a table of contents pointing to the bricks composing this work. The method and some experiments have been published in [63] and the overall flow has been submitted for publication [64].

The next chapters present the implementation of the general method in more details. Chapter 6 explains with a high-level view different methods to extract parallel tasks from a data-flow Synchronous program and defines an intermediate representation format used along the toolchain. In Chapter 7, we discuss the complete implementation of our toolchain and the execution model for the MPPA2. This chapter also exposes the NoC routing and configuration and delay computation which have been published in [21, 46, 45].

Finally, in Chapter 8, we apply our method to some avionics use cases. In Chapter 9, we discuss the results and suggest future improvement to our work.

3 Background: Synchronous Programs & Parallelization

3.1 Reactive Systems and Synchronous Languages

In this section, we present the context and usage of the Synchronous languages. Automatic controllers are used in industry to control some values of a system by adjusting the value of some actuators. These systems are widespread in plants and transportation. Examples can be found in avionics with the altitude or speed control, in automotive to control the idle speed of the motor by controlling accelerator, in energy (see Example 1 for a wind turbine), etc. These systems have reaction latency constraints.

Example 1. Recent wind turbines are controlling the speed of the rotor to maximize the energy production. Rotor speed is controlled by acting on the blades angle. A too fast rotation of the rotor can damage the system; hence the control system has to ensure that the speed remains below the limit. The angle of the blades must be controlled fast enough to keep the rotor speed to the limit, maximize energy production and avoid damage [97].

Systems designed to implement control engineering problems have been formalized as Reactive Systems by David Herel and Amir Pnueli [70].

Reactive systems are systems constantly reacting to the environment at a speed defined by this environment [67]. The principle of a such system is shown in Figure 3.1. These systems have to be reliable because they are often safety critical and can possibly execute for years without been stopped. Figure 3.2 shows a reactive program executing in an infinite loop (`while(true)`). The `read` function generally consists in sampling on the inputs. The delay between the call to `read` and the call to `write` is named reaction time.

In this chapter, we only focus on the `compute` function. The reason is that reading the inputs and writing the outputs are orthogonal problems which rely on low-level platform-specific programming.

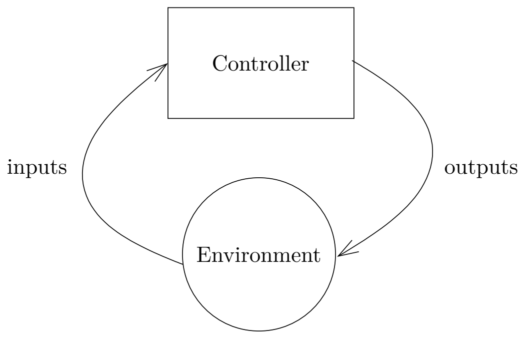


Figure 3.1: Reactive Program Principle

```
while(true) {
    inputs = read();
    output = compute(&memory, inputs);
    write(outputs);
}
```

Figure 3.2: Simple Reactive Program

Each call to the `compute` function is one reaction. The result of this reaction depends on the current inputs and previous computations stored in `memory`.

Synchronous languages have been invented to program reactive systems. We now define the main reasons why synchronous languages are well adapted for this purpose.

Determinism. A program is deterministic if for the same sequence of inputs, the same sequence of outputs is produced. An equivalent assertion is that for the same internal memory state and the same inputs, the program computes the same output. This is essential to guarantee the testability of the program. When high assurance is required mathematical proof is able to ensure functional correctness of the program. A requirement is that the language, and thus the program, has well-defined semantics. Synchronous languages have well defined semantics and a Synchronous program can be compared to an automaton.

To ease the development of complex systems, synchronous programs are modular and parallel. Synchronous languages are based on the notion of synchronous product of automata where at each transition of the resulting automaton, one transition of each automata is taken. A property is that a synchronous product of deterministic automata is deterministic [88]. On the other hand, the asynchronous product of deterministic automata does not necessarily lead to a deterministic automaton.

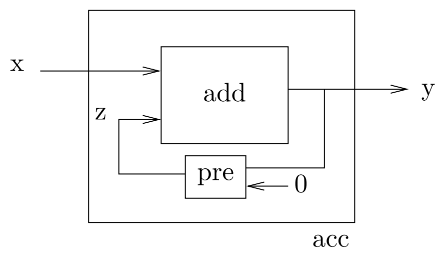
Synchrony Assumption. Implementing control software with classical languages such as C can be complex since there is no built-in notion of time. The Synchronous languages have been historically designed for this purpose. They provide a formal notion of time abstracting the host machine physical execution time.

Synchronous languages are based on the synchrony assumption where all inputs/outputs events occur on a logical clock. This clock is discrete and global. Each instant is defined on the logical time. As a result, the semantics of the program is independent from physicist time and execution time [67, 13]. The program reacts on this clock and thus does not depend on the system speed. The determinism is then enforced. This logical clock can represent physical time, kilometers or any measurement.

A good property is that the values of a variable are defined on this logical clock. In other words, the value of a variable at a previous instant stays accessible. For instance in automatic control, this allows a clear expression of the differential equations.

Bounded Memory and Execution Time. Engineers of a system must check that the reaction of the program (`compute` in Figure 3.2) between the moment the inputs are read and the moment the outputs are written is fast enough to comply with the physical requirements. In order to guarantee that reaction time is bounded, synchronous languages are voluntary limited to static memory allocation, non recursive function and bounded loop. This can be statically verified on the compiled code with the *Worst-Case Execution Time (WCET)* upper bound. In other words, the Synchronous Assumption guarantees the qualitative real-time. The quantitative real-time must be checked afterward by construction.

In this step, the synchrony assumption is checked, meaning that we verify that the logical instants can be executed on the required clock.



```
node acc(x: int) returns (y:int)
var
  z: int;
let
  y = add(x, z);
  z = 0->pre(y);
tel

node add(a,b:int) returns (c:int)
let
  c = a+b;
tel
```

Figure 3.3: Example of accumulator in counter in both textual and graphical Lustre computing the sum of the successive values of `x`.

Formal Semantics. Synchronous languages have formal semantics. The synchronous hypothesis makes the program behavior independent from the machine specificities. If the compiler preserves the semantics of the program, tests and verifications can be done at the program level instead of the generated code. This is the basis of the model-based design.

Model checking is a method that verifies properties on a model. With Synchronous languages, it is done in two steps: the properties that have to be checked and the constraints from the environment are encoded into an observer. An example of properties is *The two doors cannot be open at the same time*. Usually this observer produces a single Boolean indicating if the property is valid. A model checker verifies these properties by considering all the states of the program. Examples of model checker are Lesar [110], Kind2 [31], NP-TOOLS [92] or SCADE Design Verifier[1]. If the properties do not hold, the model checker exhibits trace leading to this problem.

Programming Style. Synchronous languages are either data-flow or control-flow (imperative). Lustre and Signal are data-flow whereas Esterel [16] and Statecharts are control flow. Some synchronous languages provide a graphical representation such as Argos [95] or Scade [15]. The data-flow describes relations between the data whereas the control-flow controls the execution of the program. In the graphical representation, data-flow programs are composed of component linked with arrows where one arrow is a data. Control-flow programs are represented with states and transitions between these states. This allows to modularly describe complex automata. For this reason Esterel has been used for circuit design in industry.

Details on Lustre are given in Section 3.1.1. Scade is an industrial language combining both Lustre and Esterel. The Scade automata are explained in Section 3.1.2.2.

3.1.1 The Core Language: Lustre

Lustre is a data-flow synchronous language. It is based on the synchronous hypothesis, as explained in Section 3.1.

A data-flow language. A data-flow program describes well-separated entities that communicate together allowing a clear extraction of the communication information. Entities of Lustre are called nodes and a program is a node which contains a network of sub-nodes. The communication between the nodes are expressed with a set of equations which are composed of node calls, data operations (arithmetic, logic) or timing operations (*e.g.*, atomic delay, called `pre`). Node is the compilation unit

[1] `http://www.esterel-technologies.com/products/scade-suite/verification-validation/scade-suite-design-verifier/`

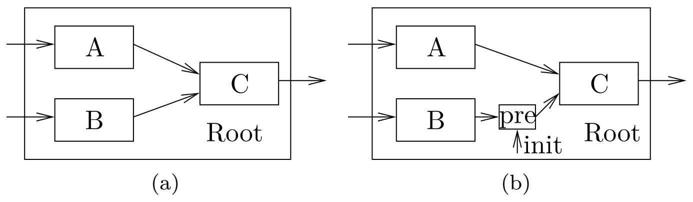


Figure 3.4: Lustre is a dataflow language. Nodes are executed in an order compliant with data-dependencies. In (b), the `pre` operator breaks the dependency.

and the modular abstraction. For instance, in Figures 3.3, `acc` is the top-level node containing a node `add` and the `pre` operator.

A synchronous language. Lustre has synchronous semantics: execution is an infinite sequence of atomic reactions. A variable in Lustre is also called flow as its value is function of time. The `pre` operator allows referencing the value of a flow at the previous logical instant. As the previous value is undefined at the first reaction, we need the arrow operator (`->`) to specify the initial value as shown in Figure 3.3. A node can depend on its own output and this construction is called feedback loop. In Lustre, a program that depends instantaneously on its outputs ($x = f(x)$) is illegal. This program would require a fixed-point computation to compile and this fixed-point often does not exist. To avoid this issue, feedback loops are accepted only if a `pre` operator breaks the loop.

Example 2. This table shows timing operations on a flow. At the first instant, `pre x` is not defined. Hence, the arrow operator is required to initialize the expression. Scade and some variants of Lustre adopt an alternative operator followed-by (`fby`) that takes both the initial value and the flow as parameter, *i.e.*, `1->pre x` ⇔ `1 fby x`.

Instant	1st	2nd	3rd	4th	5th
x	0	2	4	6	8
pre x	?	0	2	4	6
1->pre x	1	0	2	4	6

Classical Lustre Compilation for Single-Core. In Lustre, a node is expressed as a list of equations defining all outputs and local variables. Each equation is of the form `x=e;` where `e` is an expression made of constant, operations, node and external function calls. The compiler sorts the equations according to their dependencies (partial order) to obtain a sequential program. There can be several possible execution order for a program. In Figure 3.4a, the possible sequences are `A;B;C` or `B;A;C`. In Figure 3.4b, the dependency between B and C is broken, hence any permutation where `A` completed before `C` is possible, *i.e.*, `A;C;B`, `B;A;C` or `A;B;C`.

3.1.2 Conditional Computation in Synchronous Data-Flow Languages

Despite their data-flow aspect, the Lustre and Scade offer features to express conditional execution. In this section, we present the Lustre clock and the Scade automata.

3.1.2.1 Clocks in Lustre

Lustre provides a way to specify that parts of a data-flow compute "less often" than other. This is achieved via the notion of clocks.

Clocks in Lustre can be seen as a tree following the same hierarchy as the program. The top-level node is activated at each execution of the program (at each logical instant). Sub-nodes are activated according to the clock of their inputs. A sub-node of a node is activated at most as often as the node, *e.g.*, it can only be made slower.

In Lustre, a clock is a flow of Booleans. When the Boolean is true the clock is active; when it is false the clock is not. The expression `clk = true;` defines the always-true clock relatively to the parent node. This clock has the same speed as the parent node: it is active at the same time as the parent node. There are two operators to manipulate the clocks: `when` and `current`.

The `when` operator as in `expr when clk` creates a flow slower than `expr`. The result has a value only when clock `clk` is true. The `when` keyword acts as a sampling operator. In the expression `o = A(i when clk)`, node `A` is activated when clk is true, hence `o` is on the clock clk. Suppose that Node `A` multiplies its input by 10. The following table shows the behavior of the `when` operator.

Instant	1st	2nd	3rd	4th	5th	6th
`i`	1	2	3	4	5	6
`clk`	true	false	true	true	false	true
`i when clk`	1		3	4		6
`o=A(i when clk)`	10		30	40		60

The `current` operator as in `current(expr)` gives the clock of the parent node to the flow `expr`. In the expression `o = current(i when clk)`, the variable `o` has the clock of the parent node. The following table shows the behavior of the `current` operator on a small example.

Instant	1st	2nd	3rd	4th	5th	6th
`clk`	true	false	true	true	false	true
`a`	10		30	40		60
`current(a)`	10	10	30	40	40	60

Example 3 is complete program with two clocks.

Example 3. Root is a node with two sub-nodes. `A` computes $x * 10$ and `B` computes $x + 1$. Node `A` computes at every instant whereas node `B` computes only on a slow clock (once over two).

```
node root(i: int) returns (oA, oB:int)
var
  clk_slow: bool;
let
  clk_slow = true -> not pre(clk_slow);

  oA = A(i);
  oB = current( B(oA when clk_slow) );
tel
```

The following table gives the value of intermediate flows and outputs:

Instant	1st	2nd	3rd	4th	5th	6th	7th
`i`	0	1	2	3	4	5	6
`oA=A(i)`	0	10	20	30	40	50	60
`clk_slow`	true	false	true	false	true	false	true
`oA when clk_slow`	0		20		40		60
`B(oA when clk_slow)`	1		21		41		61
`current(B(oA when clk_slow)`	1	1	21	21	41	41	61

Flow `oA` is the output of node `A` which computes `i+1`. The clock of `oA` is the same as the one of root. Expression `oA when clk_slow` has a value only when `clk_slow` is true. Node `B` is activated only when its inputs have values, hence `B` is activated when `clk_slow` is true. The result of `B` is also on this clock.

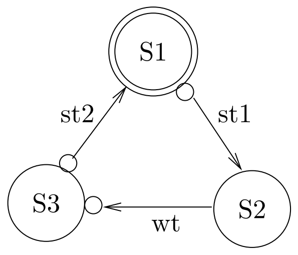


Figure 3.5: Example of Scade automaton with weak and strong transitions.

```
node root(i: int16) returns(o:int16)
let
  automaton FSM
  initial state S1
    unless if (st1) resume S2;
    let
      o = 1->pre(o) + i; -- body S1
    tel
  state S2
    let
      o = 4->(last'o) + i; --body S2
    tel
    until if (wt) resume S3;
  state S3
    unless if(s2) reset S1;
    let
      o = 8->pre(o) + i; --body S3
    tel
tel
```

Figure 3.6: Example textual Scade for the automaton of Figure 3.5.

The `current` operator makes the output of `B` faster: `oB` is defined on the root clock. Some values are duplicated to fill the “missing” values.

In the next section, we present automata which is a formalism to express conditional executions. This is another syntax to express execution clocks for the nodes.

3.1.2.2 Scade Automata

The Scade language inherits all the features of Lustre, including clock. However, it also offers an alternative feature to express conditional execution: automata.

An automaton is composed of states. For instance, the states can be initialization/nominal, degraded/performance or take off/flight/ landing. Each state has a body with some transitions to other states. In Figure 3.5, the initial state is represented with a double-circle, state with a circle and transitions with an arrow. Figure 3.6 is the same automaton in textual Scade. In Scade, only one state can be executed in the same instant. There are two kinds of transitions:

Strong transitions (`st1` and `st2` in the example) are represented with a circle at the start of the arrow. Conditions are evaluated before the execution of the state body. The first valid condition triggers the state change. In the textual format, the keyword `unless` introduces a strong transition.

Weak transitions (`wt` in the example) are represented with the circle on the arrow, at the same side as the point. They are evaluated after the execution of the state body. They select the next active state. The keyword `until` introduces a weak transition.

Example 4. In Figure 3.5, considering that the program starts in state S2 at the beginning of the cycle. The body of S2 is executed and weak transition `wt` is evaluated. If `wt` is triggered State S3 will be active at the next cycle. In the new cycle, if the condition of `st2` is false, the body of S3 is executed. If `st2` is triggered the state changes and the body of S1 is executed.

Only one transition can be triggered in the same instant, hence, if a strong transition has been triggered at the beginning of the cycle, the weak transitions are not evaluated.

Example 5. In Figure 3.5, if we are in S1 at the beginning of the cycle and strong transition `st1` is triggered, body of S2 is executed but weak transition `wt` is not evaluated.

Keywords `resume` and `reset` on transitions affect the memory behavior of the state nodes. When a state is accessed with `reset`, the sub-nodes start in initialization, *i.e.*, the initial value is used for all the variables. For instance, in the example, if `S1` is reset the value of `o` is 1. When a state is accessed with `resume`, the initial values of the variables are used only for the first execution.

All the variables defined outside the automaton are shared by default. For instance, `i` and `o` in Figure 3.6. There is only one state executed on the same cycle, hence there is no coherency problem. Nevertheless, it can be useful to choose between keeping them shared and using them as a local variable. To do this, there are two possibilities to access the previous value of a variable: `pre(v)` or `last'v`. With `pre`, the memory is local to the state, *i.e.*, the value is the one computed at the last instant when this state has been executed. With `last`, the memory is shared and the value is the one computed during the last instant, possibly in a different state.

Example 6. We compare the behavior of `pre` and `last` from the program in Figure 3.6. Variable `i` is 1. Executed state is given for each cycle.

Instant	1st	2nd	3rd	4th	5th	6th	7th	8th
Executed state	S1	S2	S2	S3	S1	S2	S2	S3
Value of `o`	1	4	5	8	1	2	3	9
`init->last'o`		1	4	5	8	1	2	3
`init->pre(o)`	1	4	4	8	1	5	2	8

For the first execution of each state (1st for S1, 2nd for S2, 4th for S3) the initialization value of `o` is used (1, 4 and 8). In the 1st instant, `last'o` is not defined. The second execution of S1 (5th instant) is triggered with a `reset` hence, the initialization value 1 is used for `o`. At the 8th instant, `last'o` is 3 but `pre(o)` is 8 (from the previous execution of S3).

We have presented some methods for representing the conditional execution of nodes: clocks in Lustre and automata in Scade . For both of them, the activation is defined by a Boolean and is not necessary regular. In the next section, we show the case of program where each node is activated regularly.

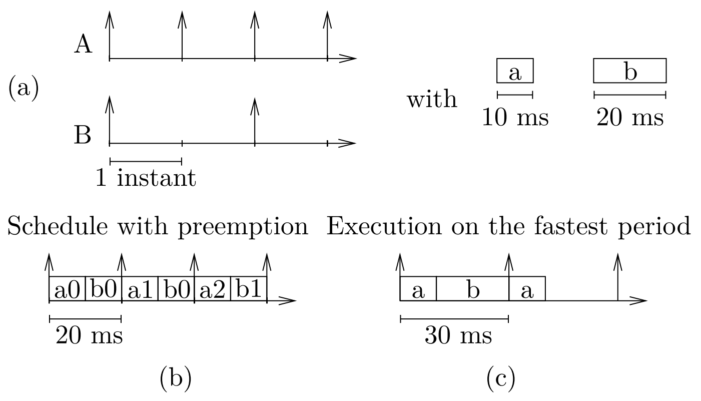
3.1.3 Multi-Periodic Programs

Multi-periodic programs are a particular case where the activation of the tasks is defined on a regular period. This is the common usage of clocks for embedded applications. Beside this activation problem, the communication has to be implemented in a way that semantics of the program are preserved.

3.1.3.1 Task Activation

There are several ways to handle multi-periodic programs. The first solution is to implement the entire program on the fastest period. We obtain a program with one fast task. The slow sub-nodes are executed conditionally according to the clock. Nevertheless, this compilation is not adapted to real-time systems since most of the time, slow (i.e. less frequent) nodes have a larger execution time than fast nodes. Another solution is to authorize the fast task to preempt the slow one. This solution is detailed in Example 7.

Example 7. Considering node `A` executed every period (or logical instant) and node `B` executed only one logical period over two (as in Example 3). As represented in Figure (a), the physical duration of `A` is 10 ms and the physical duration of `B` is 20 ms:



The real-time constraints of the program requires an execution period of 20 ms for `A` and 40 ms for `B`. The deadline of each task corresponds to its period. This example shows a schedule with preemption and an attempt of schedule on the fastest period.

Figure (c) shows that the execution on the fastest period 20 ms is not possible for this program since the execution time of one logical instant varies between 10 ms and 30 ms. Node `A` is executed every 30 ms instead of the 20 ms required in the worst case.

Intuitively, we can see that in (c), the execution time is not well balanced among the periods. As depicted in Figure (b), a solution is to split the slow node `B` and to execute in each period `A` plus half of `B`. Now, the duration of each period is 20 ms and `B` completes one period over two. As required, `A` completes every 20 ms and `B` completes every 40 ms.

This task-split can be done manually or rely on the dynamic scheduler of a real-time operating system (RTOS). The later requires the Lustre program to be compiled in a multi-task way.

Example 7 has shown a program which requires multi-tasking and preemption to meet real-time requirements. A drawback is that preemptions make the WCET estimation harder and possibly more pessimistic than the one of a plain sequential execution since it introduces context-switches and *Cache-Related Preemption Delays* (CRPD) [87].

Another solution that does not rely on a preemptive scheduler is to transform the multi-periodic program into a single-period program which behaves the same. The period of this program is called hyper-period and corresponds to the smallest period for which the pattern formed by execution of the periodic tasks is repeated. It corresponds to the least common multiple of all the periods.

Figure 3.7 shows an example of hyper-period expansion. Task `X` has a period of 2 and `X` a period of 3. As a consequence, the hype-period is 6. On a single-core processor, the schedule of one period is `x0;y0;x1;y1;x2`.

Our solution relies on hyper-period expansion. Even though this method is not always optimal since it can lead to a huge hyper-period, it allows easily avoiding usage of a scheduler. It also allows avoiding splitting the tasks. When the execution is physically parallel, the preemption can be avoided since tasks of different periods can execute on different cores.

Preserving the high-level semantics of the program and determinism in a multi-task compilation is possible but non-trivial [30, 38]. In some sense the compiler is not the only responsible for implementing

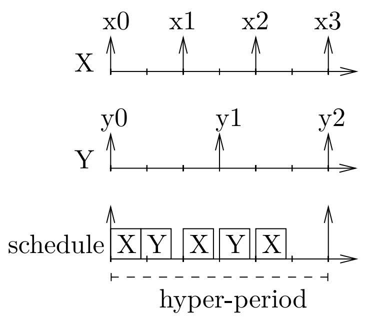


Figure 3.7: Hyper-period Expansion

the semantics: the scheduler is also responsible for this implementation. This will be discussed in the next section.

3.1.3.2 Determinism of Communication

Implementation of multi-periodic programs with several tasks usually involves communications between tasks running at different rates. The solution that consists in writing as soon a possible and reading when needed in a buffer is called "freshest value". Although simple, this solution is not deterministic since values that are read depend on the actual execution time of the producer and the actual release date of the reader [30]. To ensure testability and verification, communication has to be both deterministic and expressible in Lustre, hence "freshest value" is not expressible in Lustre.

Alras *et al.* [3] introduced an extension of Lustre called Lustre++ bringing a builtin notation for periodic clocks with phase to the Lustre language, e.g. `ck_20 = clock « 20,0 » (_)`.

Prelude [38] is a formal language especially designed to express the communication in a task-based multi-periodic program. It has been designed for system integration where tasks activation is regular and as a consequence, clocks are periodic. This is different from Lustre where clocks can be computed with any expression. In Prelude, each flow has a clock. Either their definition is absolute with initialization phase and a period, or relative to another flow. The "`expr /^ N`" operator divides the frequency by a constant and "`expr *^ N`" multiplies the frequency by a constant. The `fby` operator and the "`expr ~ > N`" are equivalent to `pre` in Lustre since the change the phase of the flow. More details on Prelude are given in Section 3.2.1.1.

Distinction is often made between two kinds of communications: fast task to slow task and slow task to fast task. In the former, some computed values are not needed and the communication is not always required. In the latter, the reader consumes more values that produced. Therefore, some values need to be repeated.

Example 8. We discuss the expression of the instantaneous communication between the two periodic tasks of Figure 3.7.

We represent the execution of both tasks on the common clock. `X` is activated on instants 1, 3, 5 and 7. `Y` is activated on instants 1, 4 and 7. For both tasks, the output at each active instant is given with the lower-case x or y.

Instant	1st	2nd	3rd	4th	5th	6th	7th
clk_X	true	false	true	false	true	false	true
clk_Y	true	false	false	true	false	false	true
Output of X	x0		x1		x2		x3
Output of Y	y0			y1			y2
X→Y	x0			x1			x3
Y→X	y0		y0		y1		y2

An example of fast to slow communication from task X to task Y is given in the table at the line X→Y. This communication is instantaneous since in the 1st and 7th instants, values x0 and x3 are both produced by X and read by Y.

In Lustre, X→Y can be expressed with `o = Y(current(X(i when clk_X)) when clk_Y)`.

In Prelude, the equivalent is `oX = X(i/^ 2); o = Y((oX*^ 2)/^ 3)`

Communication Pattern. If the period of the tasks are known and there exist a least one common multiple between them, a static pattern of communication can be computed [30].

For each activation, the pattern indicates whether the task has to write (or read). The periodic patterns corresponding to the X→Y communication of Example 8 are the following:

For `X`:

```
(x0 writes, x1 writes, x2 does not write, x3 writes)
```

For `Y`:

```
(y0 reads, y1 reads, y2 reads)
```

N-Synchronous Kahn Network. N-Synchronous Kahn networks [35, 107] are an extension of the Kahn networks with periodic clocks allowing bounded buffers between the tasks. Authors propose a relaxed clock calculus to compute the buffers size. The clocks are ultimately periodic and expressed with an initialization part and a periodic part, *e.g.*, `001(100) = 001100100100...`. Two clocks `clk1` and `clk2` are adaptable, denoted by `clk1 <: clk2`, if they never read an empty buffer and an upper-bound on this buffer's size can be computed.

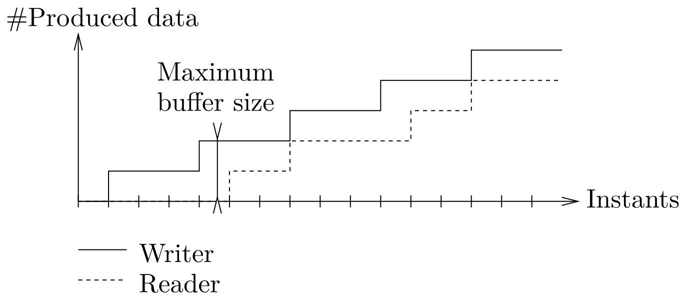


Figure 3.8: N-Synchrounous Kahn Network

Figure 3.8 is an example of two n-synchronous tasks: writer and reader with respective patterns `1001(001001)` and `0000(101000)` indicating the instants when they read or write. The maximum buffer size is given by the maximum difference between the number of written and the number of read values. In this case, a buffer of 2 is sufficient.

3.1.4 Conclusion

In this section, we described the Lustre language and the problems of compiling multi-periodic programs. Similar problems and solutions exist for Architecture Description Language whose aim is to describe the integration of several programs into the same systems. Compiling high-level program by taking advantage of parallel architectures leads to new problems.

3.2 High-Level Parallel Design

In the previous sections, we have presented some high-level formalism and their compilation for single-core processors. In this section, we present some solutions for exploiting multi-core parallelism. High-level formalisms allow writing the program as close as possible to the specifications. This reduces the potential of error. If the compilation processes and tools used are certified or proven to ensure semantics preservation, the need for test on the final binary is not longer required. We present some languages offering deterministic semantics and some that are not deterministic when executed on a multi-core. For the languages that are not deterministic when executed on a multi-core, some methods can make the parallel execution deterministic. For synchronous languages, we give some examples of methods preserving the deterministic behavior of the program when executed in a parallel way or when distributed on a network of computers.

3.2.1 Existing Approaches for Parallelism

This section presents some approaches targeting a parallel execution. These approaches are of different natures. We intentionally mix operating systems norm (such as ARINC 653), lower level languages (such as AUTOSAR), Architecture Description Languages (ADL) or coordination languages since they have the same role of assembling bricks to create a system.

3.2.1.1 Prelude

Prelude is a simple ADL software modeling that adopts the Synchronous semantics. We have briefly presented the functioning of the Prelude clocks in Section 3.1.3.2. Here, we focus on the general framework.

A Prelude program is a single node, with sub-nodes that are imported from other languages (Lustre, C). Input and outputs of the program are expressed with physical period and offset. This is a synchronous dataflow language, hence all the nodes are considered to be executed at the same time, and thus the physical execution time is abstracted away.

As in Lustre, the data-flow is expressed with “wires” (dependencies) between the nodes and the execution rate of each node is inferred from the rate of the input. As described in Section 3.1.3.2, the sampling rate of the output of the node can be multiplied or divided by a constant, this operation is carried on the wires (intermediate variables).

Prelude is strongly typed, and all the flows are clocked. The compiler does classical static verification: the program is type-checked and the clock calculus is performed. The clock for inputs and outputs of the program must be coherent. As Scade and Lustre, it rejects program with feedback loop that are not broken with a delay (`pre` in Lustre and `fby` in Prelude).

From this multi-periodic program, a set of periodic tasks is extracted together with their precedence constraints. SchedMCore [38] computes a schedule where preemption and migration of tasks are allowed. The real-time properties of this schedule are verified. Finally, the tool performs a schedulability analysis of the program using the WCET of each task.

Prelude has well-defined Synchronous semantics. A parallel execution which preserves the semantics of the program is deterministic. This is not the case of all the ADL. For instance, AUTOSAR has no well-defined semantics as shown in the next section.

3.2.1.2 AUTOSAR

AUTOSAR is a software architecture standard. It has been designed to abstract hardware. For this purpose, AUTOSAR is based on a Virtual Function Bus (VFB) which is implemented with a Runtime Environment (RTE) [99].

An AUTOSAR application is composed of several independent components connected together with the VFB. Each component is composed of runnables which implement the behavior of the program. Communication between the components is described with ports.

The component, runnables and the VFB is the abstract vision of the program. For the concrete implementation, an hardware is chosen and components are mapped on this hardware. The specific RTE or AUTOSAR OS implements the VFB and the runnables are mapped on the OS tasks. Ports are then implemented using the physical medium.

The AUTOSAR OS implements a scheduler. Runnables are triggered by events such as timer, data reception, mode switch, etc.

Communication ports are implemented with shared data. Sending and receiving from a port is equivalent to writing and reading to the shared data. Two interesting kinds of communication port of the AUTOSAR specification are the *explicit* communication and the *implicit* communication.

For the *explicit* communication, the runnable gets the last value when it reads it: this corresponds to a read in a shared variable. The problem is that the data can change during the execution and if the read of different inputs are not done at the same time it can lead to incoherent reads [69]. For the *implicit* communication, the runnable gets its own copy of the data at the beginning of the execution and writes its outputs at the end. As a consequence, data cannot change during the execution. Also, the cost of accessing remote data is paid only once. It ensures functional correctness of the application.

An activation period (or an event mechanism) is defined for each task. For instance, the runnables with the same period can be mapped on the same task or one task can be created for each runnable. When the tasks are created, at compile time, runnables are scheduled inside the tasks according to their dependencies. At runtime, the tasks are scheduled by the RTE.

In both explicit and implicit communication ports, determinism of the execution is not guaranteed since the moment when the data is read is not well-defined and depends on the execution time of the tasks [69]. See Section 3.1.3.2 about the “freshest” values.

Kehr *et al.* [80] introduce the Timed Implicit Communication (TIC) which adds predictability to implicit communication. This is a new port added to the AUTOSAR RTE. The order of execution between the producer and the consumer is enforced by embedding a timestamp with each data transfer. This timestamp corresponds to the end of the period; as a consequence the end-to-end latency is increased since communication is not possible within the same period. The advantage of this method is that the execution keeps a determined execution order on any platform and especially from single-core to multi-core processors.

TIC is specific for AUTOSAR, nevertheless, a more general solution to this lack of determinism is the LET paradigm. It allows defining the moments when the data are read and written and thus guaranteeing this determinism at the cost of increasing the end-to-end latency [69].

3.2.1.3 Giotto / Timing Definition Language (TDL)

The Logical Execution Time (LET) is an execution paradigm close to the one of the synchronous languages sometime called Zero Execution Time (ZET).

Synchronous languages are based on logical discrete time (see Section 3.1.1 for more details). Data and communications are dated on a discrete and logical time scale regardless of the execution time. After compilation, the timing of the program are compared to the specifications to make sure that the program can finish before the next release.

LET pushes the concept of timing constraints inside the program. It defines two release dates per task: a release date for the reading of the data by the task and a release date for the production of the output data by the task. This release dates are part of the behavior of the program. The compiler checks the schedulability of the program. The implementation of a LET program is not necessarily

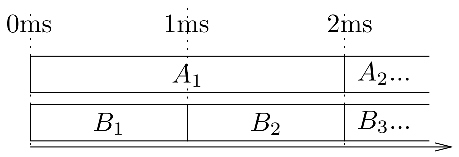


Figure 3.9: Example of Logical Execution Time program.

time-triggered. The tasks are started during the period. The release date when the input is read does not correspond to the beginning of the computation. The constraint is that a task has to finish before the release date of the output.

Figure 3.9 shows a periodic program with two tasks `A` and `B` respectively executed once and twice in the period, hence `A` produces a value every 2 ms and `B` every ms. Value of the 1st execution of A (A1) is available at 2 ms and can be read by B3. Value of the 1st execution of B (B1) is available at 1 ms. The main constraint on implementation is that tasks must be executed within their period.

Giotto [72] is the academic implementation of LET. Timinig Definition Language (TDL) [108] is a commercial language based on Giotto.

Mode, Task, Port and Driver. A mode is a set of periodic tasks. A Giotto program is composed of at least one mode with a fixed physical period. A task is introduced with the keyword `taskfreq`. It is characterized by a release date and a termination date. These release dates are computed with the period of the mode and the number of activation ω of the task in this period. *Drivers* are responsible for reading inputs and writing outputs at the beginning of the period and at the end. Drivers are considered to be executed in zero time at a precise release. Execution of a task is conditioned to the value of its inputs driver.

Example 9. This example is an extract of flight controller for helicopter written in Giotto [73]. The task `pilot` computes the path, the task `lieu` estimates the position and `control` computes the control of the helicopter.

```
mode hover () period 120ms {
    taskfreq 1 do pilot();
    taskfreq 2 do control();
    taskfreq 3 do lieu();
}
```

Task `pilot` executes 1 time ($\omega = 1$) in the period, `control` executes 2 times ($\omega = 2$) (every 60ms) and `lieu` executes every 40ms ($\omega = 3$).

There is no release date for the functional code of the tasks but there are for the drivers of the tasks. The input driver of `pilot` is executed at 0 ms; its output driver at 120 ms. Input drivers of `control` are executed at 0 ms and 60 ms; output drivers at 60 ms and 120 ms. Input drivers of `lieu` are executed at 0 ms, 40 ms and 80 ms; output drivers at 40 ms, 80 ms and 120 ms.

Modes Switches. Giotto offers modal execution. Depending on the mode, a different set of tasks is scheduled. Switch between modes is guarded by conditions evaluated periodically. The condition is introduced with the `exitfreq` keyword and is a driver producing a Boolean. Communication is allowed between the modes with the same principle as communication between tasks. Figure 3.10 is an example program with two modes.

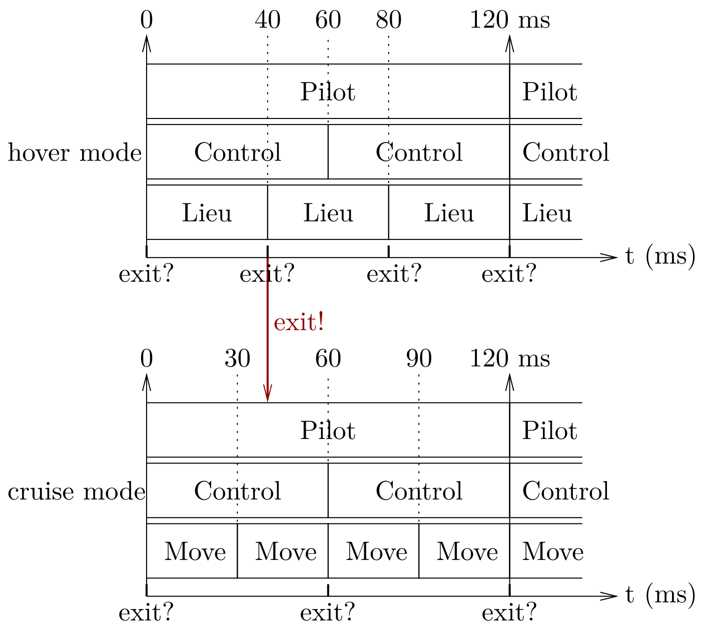
Mode switch is straightforward when the switch period and the task are harmonic. On the contrary, there are constraints on the program [72] since preemption is forbidden in Giotto. The constraint is that if the period of a task t is not harmonic with the period of the mode-switch driver s ($\frac{\omega_t}{\omega_d} \notin \mathbb{N}$),

```
start hover {
    mode hover () period 120ms {
        exitfreq 3 do cruise(switch);
        taskfreq 1 do pilot();
        taskfreq 2 do control();
        taskfreq 3 do lieu();
    }
    mode cruise() period 120ms {
        exitfreq 2 do hover (switch);
        taskfreq 1 do pilot();
        taskfreq 2 do control();
        taskfreq 4 do move();
    }
}
```

Figure 3.10: Example of Giotto program with two nodes from [73].

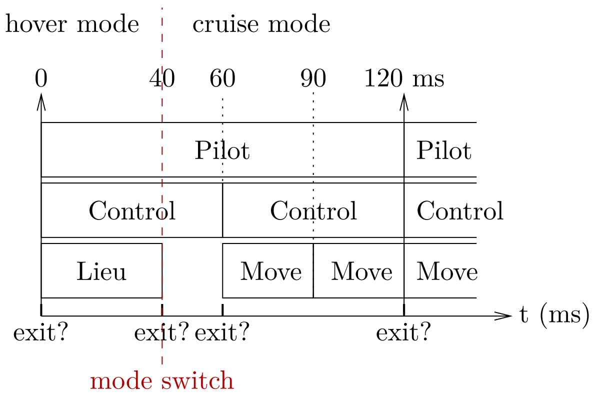
the task has to be in both modes and must keep the same period. This corner case is explained in Example 10.

Example 10. We explain how mode switches in Giotto work and what are the constraints. Considering the example in Figure 3.10 with a mode `hover` and a mode `cruise`. In `hover` mode, the helicopter keeps its position. In `cruise` mode, the helicopter follows a path. The `move` task computes the speed and the position of the rotorcraft whereas the `lieu` task compute only the current position.



As shown in the figure above, both modes execute `pilot` and `control` at the same frequency. Plus, mode `hover` executes `lieu` on a period of 40ms and `cruise` executes `move` every `30ms`.

Mode switch is evaluated every 40ms in `hover` and every 60ms in `cruise`. Now, let us analyze the case where the mode is changed from `hover` to `cruise` at 40 ms, as denoted with the red arrow.



The diagram above shows a mode switch from `hover` to `cruise` at 40ms. At this point, `pilot` and `control` keep executing since they are in both modes. Note that, Giotto forbids preempting or halting of tasks. If these tasks were not present in both modes, the program would be incorrect and rejected by the compiler.

On mode switch, execution of `lieu` is finished but it happens during an execution period of `move`, hence `move` cannot be scheduled instantaneously: it is scheduled on the new mode at its next release.

Kluge *et al.* [82], implemented the LET paradigm for a many-core processor. The implementation relies on NoC communications and a hardware global clock. Breaban *et al.* [25] provide a solution if the multi-core processor is not featured with a global clock. The solution consists of global time-triggered barriers to synchronize the tasks. The *time-aware processor* is responsible for unlocking the barrier according to its own clock.

The industrial language TDL [108] is an improvement of Giotto regarding several aspects. It adds an outer level: the modes are contained in components. These components are isolated and do not interfere together. Timing verification is performed on all the component together.

The drivers are made implicit and are automatically generated from the function parameters. It provides an extension to combine time-triggered and event-triggered activities. These asynchronous tasks are scheduled in the spare time.

As a conclusion, the LET paradigm and its implementations Giotto and TDL are ways to specify multi-period schedules. The inter-task communication is made deterministic since data transfers occur on fixed periods. This schedule is independent from the platform and the mapping of the tasks. Describing this schedule requires the specification of a period for each task. In Giotto, the duration of a task is considered to be the same as its period. For this reason, the communication between tasks of same period is always delayed which is neither necessary nor efficient.

3.2.1.4 ARINC 653

ARINC 653 is a standard of operating systems for avionics. This system is called Integrated Modular Avionic (IMA) and allows integrating and isolating software of different safety level in the same processor. This results in a gain of space and power consumption.

An ARINC 653 system is composed of *partitions* that execute software and are isolated from the other. This isolation is temporal or spatial.

The Application Executive (APEX) is an interface (API) to manage isolation, shared resources and communication between the partitions. This communication is done using *ports*. The applications are implemented using only APEX primitives making the software independent from the platform.

The purpose of isolation is to ensure that the execution of a partition does not affect the behavior of another. The different partitions are isolated by the mean of time or space. The time partitioning is ensured by allocating one time slice per partition. The spatial isolation is ensured by the Memory Protection Unit (MPU) and the semaphores for devices. There are two kinds of APEX communication ports: the sampling and the queuing [129]. Sampling makes only the last value available whereas queueing avoid losing values. There exist several implementations of ARINC 653 as PikeOS, TiCOS or VxWorks653.

The purpose of ARINC 653 is to isolate partitions, nevertheless, ensuring this total isolation between processes in a multi-core system is a challenge [114].

3.2.1.5 Architecture Analysis and Design Language (AADL)

AADL [49] is a formal language to describe both hardware and software in a system. It is different from Prelude (Section 3.2.1.1) which describes only the software. The model of the hardware is composed of processor, memory, external devices and bus. The software is described with processes (isolated) and threads. A thread executes a subprogram that is an external piece of software (that can be designed in C, Scade, Simulink, etc).

Communications between the processes is done through ports. Ports convey either a typed data or an event.

Hugues *et. al*[76] present an AADL specification for ARINC 653 systems. Both specification checking and automated code generation are presented.

3.2.1.6 Conclusion

In this section, we briefly described some methods that allow multi-core execution of time-critical systems. We saw that they offer a description of the program architecture allowing extracting tasks. They are also used to check timing properties. Nevertheless, the functional code (nodes in Prelude, runnables in AUTOSAR, etc) are imported and described in another language. Prelude and Giotto offer well defined-semantics for communications that abstract the communication and the execution time. Nevertheless, in AUTOSAR, communications are not deterministic since they depend on the execution time and the schedule of the tasks. Determinism has to be enforced using additional mechanisms [80].

3.2.2 Parallelization from Synchronous Languages

This section presents some work about parallelization of the synchronous languages. We focus on how to extract parallelism from data-flow programs. We also talk about task extraction but the word “task” has to be understood as *runnable*, *i.e.*, an atomic piece of code which has to be scheduled and mapped on a core to be executed.

3.2.2.1 Parallel Scade

An extension of the KCG Scade compiler allows compiling a legacy Scade program into parallel tasks was introduced in [101].

This extension allows describing parallel subsets which are composed of nodes that are candidate for parallel execution. Dependencies between the tasks of this set are forbidden.

At compile time, when the equations of the Scade program are ordered, the parallel subset is considered as a virtual node and is scheduled as any other node. Then, it is replaced by remote procedure call for each sub-node (SEND and RECV statements to send the inputs to the node and receive the outputs after the computation is done) making possible a parallel or a distributed execution of the node of the set. Implementation of the remote procedure call is not provided by KCG.

This compiler extracts parallel parts from the program and generates a file representing the communications and dependencies between these parallel parts of the program. Only the functional code (runnable) of these parts is generated. We will describe in Chapter 6 how these parts can be assembled and can communicate to form a parallel program for a many-core architecture.

3.2.2.2 Distributed Lustre

An extensive research has been done on the distribution of Lustre over a network of processors. Girault [57] presents a survey of these works.

Most of them are based on the Globally Asynchronous, Locally Synchronous (GALS) principle, where a Lustre program is partitioned into several programs communicating asynchronously through FIFOs. In other words, some synchronous programs are connected asynchronously. Read on FIFOs is blocking if the FIFO is empty while write is asynchronous and non-blocking. The motivation is that the clock synchronization can be expensive and inaccurate in a network.

The Ocrep [29] tool automatically distributes synchronous programs (Esterel or Lustre). The user must specify the location of each variable. Thanks to this information and the data dependencies, the partitioning algorithm distributes all computations while introducing all the necessary FIFOs.

Girault *et al.* [58] propose to use clocks for partitioning programs into locations. The main motivation is that in multi-periodic programs (see Section 3.1.3), low rate tasks are often compute-heavy. Their execution can be desynchronized [58], *i.e.*, they execute in parallel on another processor across several logical instants.

These methods are not specific to dataflow-synchronous languages and the dataflow structure (nodes) is not used to partitioning the program and extract parallel programs.

3.2.2.3 Other Languages

ForeC [128] extends C with synchronous semantics (global tick) and parallelism support (threads). A tick is finished when all threads execute the specific instruction `pause`. Operator `par` allows forking/joining a thread. Hence, the tasks are explicitly created by the programmer. Shared variables are duplicated at the beginning of the period and merged at the end with a deterministic operator (*e.g.* sum). The resulting program is statically scheduled. A method [128] is given to analyze the Worst-Case Response Time on a shared-memory multi-core architecture.

3.2.2.4 Parallelization of Dataflow

For most of the dataflow languages, the tasks are extracted from the structure of the program. In other words, tasks are created from the nodes as atomic entities. The advantage is that the traceability between the model and the parallel code is kept.

Distribution of Simulink programs by the mean of Lustre as an intermediate language presented by Caspi *et. al* [28]. In Giotto [74] and Prelude [38], periodic tasks are extracted from the nodes of the program.

Extracting parallelism from synchronous languages can be done either by following the structure of the data-flow if this language is data-flow or by splitting this program into sub-parts by data dependencies. Once the set of tasks is obtained, they have to be scheduled and mapped on core. The inter-task communications are inferred from the original program and implemented faithfully.

3.2.2.5 Conclusion

In this chapter, we presented the necessary background on language and compilation to understand our work. We can summarize this chapter as follow:

- The importance of computation time for time-critical software
- The advantages of Synchronous languages
- Presentation of Lustre and Scade
- Preservation of deterministic communication without OS
- The need of splitting the program into tasks and the expression of periodic tasks in ADLs

We present the hardware platform we target and the sources of non-determinism for the execution time analysis of an application.

4 Background: Many-Core

4.1 Multi- and Many-Cores System-on-Chip Architectures

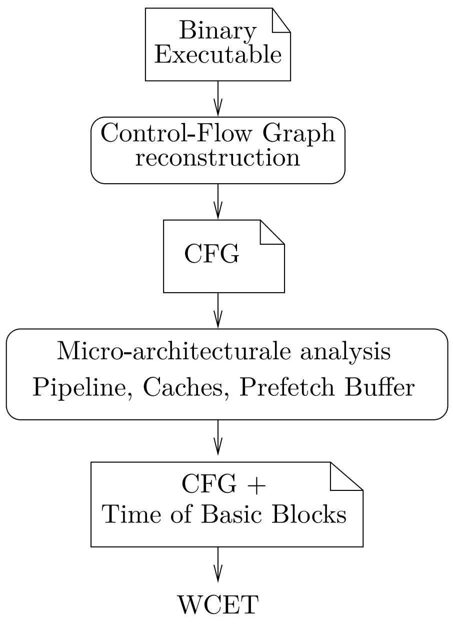
A Many-Core System-on-Chip (called simply many-core in the sequel of this thesis) is an architecture composed of a large number of cores designed for highly parallel execution. The consequence of this high number of cores is that the architecture and especially the on-chip communications must be scalable. In particular, the cores have to be simple and power efficient. A Network-on-Chip (NoC) offers both scalability and power efficiency of communications [12].

Multi-core and many-core processors are considered to be a best-fit solution to increase computing speed, optimize energy consumption and improve area efficiency in embedded systems.

4.1.1 Worst-Case Execution Time Analysis on Multi-Core

4.1.1.1 WCET in Isolation

The WCET analysis consists in finding the sequence of instructions in a program that leads to the longest execution time.



Binary Executable
Control-Flow Graph reconstruction
CFG
Micro-architecturale analysis
Pipeline, Caches, Prefetch Buffer
CFG + Time of Basic Blocks
WCET

Figure 4.1: WCET analysis for a single-core.

```
for(i=0; i<255; i++) {
  if(T[i] < 0) {
    a = f(b);
  } else {
    b = g(a);
  }
}
```

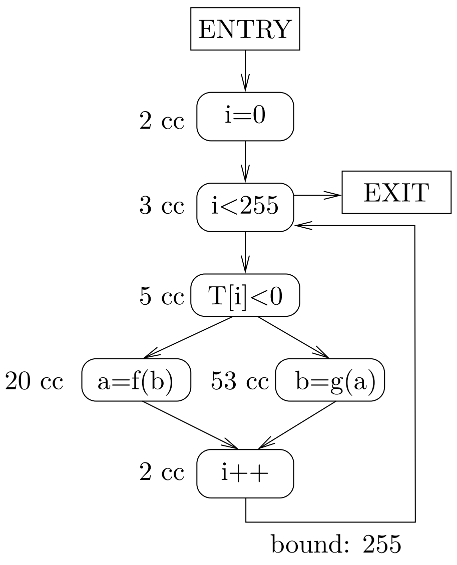


Figure 4.2: A C program with its CFG. Basic blocks are labeled with local WCET in cpu cycles. Back edge is bounded.

Figure 4.1 shows the main steps of a typical WCET analysis. The first step is the extraction of the control flow graph (CFG) from the binary. The vertex are sequences of instructions called *basic block* (BB) and the edges represent the control passing between two BB. A back-edge represents a control loop. Figure 4.2 shows a program and the corresponding CFG. The second step is the attribution of a duration to each BB and/or edge. These durations are computed thanks to a model of the micro-architecture of the processor including the pipeline timing, the cache policy, the prefetch buffer behavior, the memory latency, etc. For instance, program in Figure 4.2 is a simple model where the durations where only BB have durations. The pessimism of the durations is due to the hardware abstraction of the processor. The WCET corresponds to the most weighted execution path. Consequently, if the back edges are not bounded, the execution time is infinite. The program must be analysed to to find the back edge bounds which is the maximum number of iterations. The example of Figure 4.1 is simple since the bounds of the `for` loop are explicit. Finally, the longest path of this graph is computed. This example is timing compositional since the global WCET is the sum of the local WCETs of the BB.

Timing anomalies [112] are close to the concept of scheduling anomaly: a faster local execution can lead to a slower global execution. With timing anomalies, locally selecting the edges of the CFG leading to the highest execution time is made impossible: a global evaluation of the graph is required. This leads to complex and pessimistic WCET evaluation. Architecture is said fully timing compositional [127] if there is not timing anomaly in it. Others anomalies can be due to out-of-order pipeline since creating the CFG of the program is dynamic.

4.1.1.2 Interference-Aware WCET Analysis

The WCET analysis on multi-core processors is different from single-core since it has to take shared memory and more generally shared resources into account. For instance, a concurrent access of two cores to the same resource can lead to an extra delay for the access. Pellizzoni *et al.* [104] show a growth of nearly 3x between the execution time of a task in isolation and the execution time taking with concurrent access to the memory system. This interference can be caused by the shared cache, the external memory or the scratchpad memory.

Distinction is made between two notions: the WCET in isolation and the *Worst-Case Response Time* (WCRT). The WCET in isolation corresponds to the traditional WCET analysis that can be performed on a single-core processor. The WCRT is the execution time of a task taking into account the external phenomenon such as the interferences and the preemption delays. Nevertheless, this section considers non-preemptive schedule, hence WCRT means WCET with interference. Lv *et al.* [94] presented a framework for analyzing the interference due to shared cache.

There are several ways to compute the WCET with interference. One solution is to prevent all interferences by the mean of an execution model (congestion free). The opposite is to include the possible interference for each access in the WCET computation, *i.e.*, taking into account all the possible interfering accesses. An intermediate solution is to minimize the interference with an execution model and estimate a tight bound on the interference delay. With this solution, the set of interfering tasks is known and minimal.

Davis *et al.* [43] presented a WCRT analysis decoupling the computation of the WCET in isolation and the delays relative to the interference. The advantage is that a tool for WCET analysis designed for single-core is sufficient. Only the analysis of the cache and memory arbiter has to be tailored for multi-core.

Memory interferences have a consequence on the WCET of a task. Contrary to the analysis of the micro architecture, the interferences cannot be analyzed locally: the whole application has to be analyzed at the same time. The interference has consequences on the schedule since some tasks can be made slower. The schedule has also consequences on the interference analysis since it changes the set of tasks concurrently accessing the memory. Consequently, information about the release date and the completion date of each task is required to analyze the interferences. A time-triggered execution ensures these dates. Rihani *et al.* [113] adapted the approach of Davis *et al.* [43] where the WCRT analysis is decoupled from the isolated WCET analysis and adapted this method for the Kalray MPPA2. They introduced an algorithm computing the release dates of the tasks. A release date takes into account the data precedence and the interference. The execution of the tasks is time-triggered in such a way that a task cannot start earlier even if the previous task is faster than its WCET. The algorithm starts from the WCET in isolation of the tasks and iteratively adds the delays due to the interferences. This algorithm has been implementation in the MIA tool[1]. Our work relies on the tool from Rihani *et al.* [113] for the time-triggered execution. We generate code and meta-data that are used by MIA to compute tight WCRT bounds, and conversely we use the static information computed by MIA to staticly schedule the tasks.

Skalistis *et al.* [120] presented a similar method with a time-triggered execution model. Contrary to the previous method, this one starts with an initial duration for each task considering the maximal interference. Then, the WCET of the tasks is refined by removing some interference taking advantage of the banked memory. Release dates of the tasks are computed in a way that refinement of the WCET cannot add new interference. This leads to a loss of precision compared to the method of [113] that we are using in our work.

4.1.2 System-on-Chip For Time-Critical Software

In the previous section, we presented some methods for WCET analysis on a multi-core processor. In this section, we describe some multi- and many-core that have been designed for embedded predictable systems. We focus on the architecture properties giving good timing predictability to a processor.

Definitions. The Worst-Case Execution Time (WCET) is an upper bound on the execution time of a code in isolation. The Worst-Case Response Time (WCRT) is an upper bound on the execution time of a code in taking into account the shared component and the interferences due to the other concurrent code. The Worst-Case Traversal Time (WCTT) is an upper bound on the time required by a packet through a network to reach its destination.

[1]`http://www-verimag.imag.fr/Multi-core-interference-Analysis.html?lang=en`

Core. Most of the modern processors are optimized for average performance, neglecting the worst-case performance and leading to pessimistic WCET bounds. For time-critical applications, the cores ideally should be fully timing compositional in order to provide the least pessimistic WCET.

Caches. Cache replacement policy has also impact on the WCET analysis. Some cache replacement policy requires knowing all the execution history in order to compute the contents of the cache. If this complete history cannot be known, pessimistic hypothesis have to be done on the contents of the cache. Some caches replacement policies such as pseudo least recently used (PLRU) or first in first out (FIFO) has been shown to lead to very pessimistic WCET analysis [14]. The least recently used (LRU) policy offers both performance and predictability. If the data or instruction cache is shared by several cores, it leads to arbitration between the cores and makes the analysis pessimistic.

Memory. The access time of the memory has to be predictable. SDRAM memory is hardly time-predictable, nevertheless, active research in this field provides numerous software or hardware solution to bound the access time. Methods to make the SDRAM predictable will be given in Section 4.1.3. Static memories are expensive since they require more silicon area. Nevertheless, they are fast and they handle most of the request in constant time. It should be noticed that even for memory responding in nearly constant time it is a shared resource and requires an arbitration of the requests. Arbitration between the requesting cores leads to extra access delays and processor stall.

The memory can be distributed on the chip in such way that some small memories are close to the cores. A scratchpad [9] is a small and fast memory, private to a core whose purpose is to replace cache. If the memory is shared among several cores, it can be composed of several banks to decouple the accesses to different parts of the memory. An appropriate memory mapping allows minimizing the congestion by allocating the data relative to different cores in different banks.

NoC. The Network-on-Chip (NoC) has to provide bounded transmission time called *Worst-Case Traversal Time* (WCTT). A *route* defines the list of network elements to cross before reaching the destination. This route can be static, *i.e.*, computed before the transmission or dynamic if it can change during the transmission. Dynamic routing can be responsible for unpredictable WCTT (see Section 4.2). As the NoC is shared among several users, the link arbitration leads to delays. These delays can be avoided by preventing concurrent usage of the NoC with Time Division Multiple Access (TDMA). The WCTT can be computed using Network Calculus [39] or Real-time Calculus [124] theories.

Name	Cores	On-chip RAM	NoC Characteristics
Kalray MPPA2	16+1 per cluster, private I/DCache, software cache coherency	Banked SMEM private arbiter	Wormhole, hardware bandwidth limiter, remote memory write
T-CREST	1 per cluster, method, data and stack cache, software cache coherency	Private scratchpad memory (SPM)	TDMA, remote SPM write
PULP	Configurable (1-16), private ICache, no data cache,	Banked shared memory, tree arbitration	System bus, share-memory communications
CompSOC	1 per cluster no cache	Private Instruction + Data + DMA + communication memory	TDMA remote DMA read/write

Table 4.1: Comparison of the Kalray MPPA, T-CREST and PULP architectures.

Existing Architectures. The T-CREST project [116] proposes a predictable many-core processor designed for real-time critical systems. There is only one core per cluster and the communication between the cores is made through the NoC. Data are transferred from local memory to remote memory using a message passing interface. This NoC has a Time Division Multiple Access (TDMA) arbitration. Each core has a special instruction cache which is called *method cache*. A method cache stores entirely the procedure which is currently executed. A miss can only happen on procedure call or return and thus not during the execution of the procedure. The T-CREST has a cache for the data and a cache for the stack. The data cache is either a direct-mapped cache or two-way associative LRU cache. The stack cache [1] provides the `ensure(n)` instruction making the cache loading the n last elements of the stack and the `reserve(n)` instruction ensuring there are n free elements in the cache. For both of them, if there is not space enough in the cache, data is written back in the memory.

The PULP platform [37] is a many-core processor optimized for low energy consumption. There are several memory levels. In each cluster, there is one L1 shared-memory composed of several independent banks. A bus connects each cluster to a L2 memory common to all the clusters. For each core, there is a private instruction cache. There is no private data cache, as a consequence the memory system is coherent without the need of hardware coherency mechanisms. The shared-memory is accessible through a word-level [37] tree-based arbitration [109].

The CompSoC [61] processor has been especially designed for mixed-criticality. Consequently, it authorizes full isolation between different applications. It is composed of clusters (called tiles) connected with the synchronous dAElite [123] NoC.

dAElite is a time-division multiplexing (TDM) NoC. Packets do no have header. Slots are assigned to each connection and at each router the next direction to take is associated to each slot. In each router, a periodic table associates a direction to each time slot. A configuration network conveys the configuration packets required to set up the routes.

Each CompSoC cluster has one core with several dedicated memories. They are static-memory with a small and constant access time. Furthermore, there is no cache in order to make the WCRT computation simpler. There is one instruction memory and one data memory and there are two memories dedicated to NoC communications: one for DMA and one for communication. A NoC transaction is either a copy from a remote communication memory to the local DMA memory or from the local communication memory to the remote DMA memory.

The Kalray MPPA2 is a commercial off-the-shelf (COTS) many-core processor. It is a predictable architecture that particularly targets critical and embedded real-time applications. More details about this architecture are given in Section 4.3.

Table 4.1 summarizes some characteristics of the processors.

4.1.3 Synchronous Dynamic Random Access Memory (DDR SDRAM)

The SDRAM is widely developed due the high density of information stored in small silicon area compared to the static memory. Nevertheless, this density is obtained at the cost of a complex access protocol and an important latency.

In this section, we present the problem of making the memory access time-predictable and some existing research in this way. We consider a JEDEC [2] standard DDR3 SDRAM.

4.1.3.1 DRAM Overview

The purpose of the SDRAM is to provide a high density at low cost. It is composed of a physical part storing the data and a controller part managing the physical part. To save silicon area this controller is not duplicated even with a large amount of memory.

The purpose of the controller (Figure 4.3) is to convert the requests from the masters to lower-level commands that are linked to the physical implementation of the memory. This controller is configured to fit the memory physical constraints.

[2]`https://www.jedec.org/`

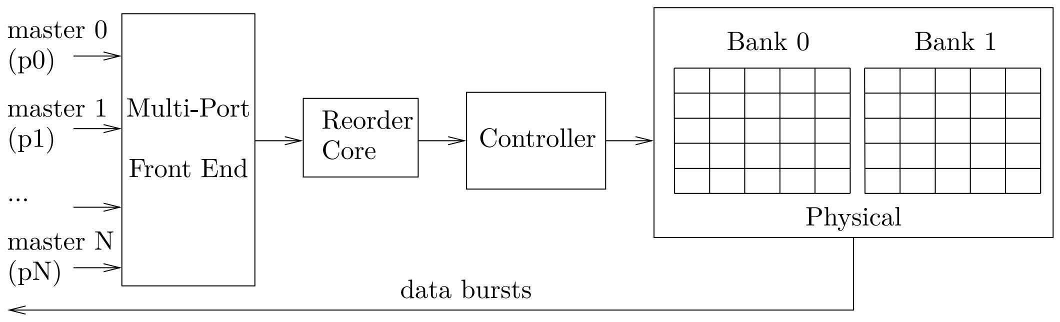


Figure 4.3: SDRAM controller overview.

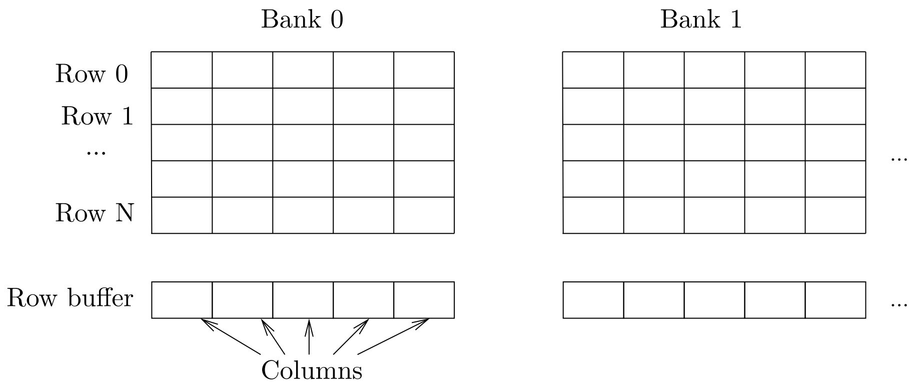


Figure 4.4: SDRAM physical view.

The aim of the reorder core (Figure 4.3) is to optimize the average throughput of the memory controller. For instance, physical SDRAM imposes some delays between series of reads and series of writes. Delays occur when accesses are non linear. In these cases, the reorder core changes the order of the requests to perform them globally faster. Nevertheless, this reorder makes the access time difficult to bound since the requests are not processed in First-Come First-Served (FCFS) order. The reorder core preserves the coherency of the data by enforcing the order of commands targeting the same address.

In multi-core systems, the SDRAM is shared among several masters, hence an arbiter is required. The Multi-Port Front End (MPFE) acts as an arbiter of requests with priority where a priority is assigned to each master. In the T-CREST processor, this MPFE is replaced with a memory tree which is considered more scalable [116].

4.1.3.2 Memory Organization: Rank / Bank / Row / Column

A SDRAM memory is composed of *banks* and each bank is composed of *rows*. Then, a row is divided into *columns*. Figure 4.4 shows an overview of this SDRAM organization. Several SDRAM memories can be assembled. In this case, each of them is one *rank*.

Banks are independent modules working in parallel. Due to the physical implementation, data in a bank cannot be modified directly. We call *activate* the action of copying a row from the memory to the row buffer. There is one row buffer per bank in which read and modification are performed. We detail the main commands of the SDRAM used to access the memory:

ACT	Activate: Open row and transfers it to the row buffer.
PRE	Precharge: Close the row and write back the contents of the row buffer to the memory.
RD	Read data from the row buffer.
WR	Write data to the row buffer.

A transaction always refers to a row in a bank, which may be opened or closed. When opened, only a single RD or WR command is issued. When closed, the command sequence PRE, ACT, RD or WR is issued. The Controller Core tracks the state of the banks in order to issue the correct commands. Refreshing a row inside a bank has the effect of closing any opened row in that bank.

Periodically, the columns of the SDRAM have to be refreshed to avoid data loss. This is performed periodically with the REF command. The consequence of this command is that the opened banks are closed; nevertheless it does not interrupt the current accesses.

RBC and BRC Modes. Most of the SDRAM controllers can be configured in two addressing modes: row/bank/column (RBC) and bank/row/column (BRC). These modes change the mapping between the address and the physical memory. In RBC the most significant bits of the address select the row, the middle bits select the bank and the least significant bits select the column. In BRC, the most significant bits select the bank while the middle bits select the row. In the RBC mode, consecutive addresses are related to different banks. This provides better average performances since linear accesses are distributed on the banks. The BRC mode eases the mapping of the data in specific banks since consecutive addresses are assigned to the same bank. This helps reducing interferences on memory accesses and makes timing analysis easier since the destination bank of each access is known.

4.1.3.3 Temporal Behavior

The DDR physical implementation imposes specific delays between the commands: PRE, ACT, RD and WR. For example, it is possible to issue several WR commands in a row but there is a minimum delay between an RW and a WR command. Some delays have effect on one bank and some are applied globally to the commands on all the banks. These physical constraints are known by the SDRAM controller which ensures the correct functioning of the memory. These constraints have to be taken into account in the computation of the worst case access time which makes it complex.

According to [81], the effect of SDRAM refresh is negligible and can be approximated to a global reduction of the bandwidth.

We describe the delays relative to the SDRAM commands for a standard JEDEC DDR3 SDRAM memory. The consequence of these timing are described in [81] and [105]. Some are global to the SDRAM, other are relative to one memory bank.

Timing constraints global to the SDRAM.

$\mathbf{t_{FAW}}$ Four-activate windows. No more than 4 activations can be performed during this period.

$\mathbf{t_{RRD}}$ Row to Row Delay. Minimum time between two row activations.

$\mathbf{t_{WTR}}$ Minimum delay between a write and a read command. This is global to the rank, *i.e.*, to one SDRAM memory.

Timing constraints relative to one bank.

$\mathbf{t_{RAS}}$ Minimum time between the activation and the precharge (close) of the page.

$\mathbf{t_{RCD}}$ Minimum delay between activation and read or write command.

$\mathbf{t_{WR}}$ Write Recovery time. Minimum of time between a write and the precharge of the row.

$\mathbf{t_{RP}}$ Minimum time between precharge and activate. The *Page Close Mode* (also called *auto-precharge*) makes the controller close the page right after the access. Due to the t_{RP} delay, it can be advantageous to configure the SDRAM controller in this mode to minimize the delay between the last access and the new activation.

4.1.3.4 Timing Predictability with SDRAM

We saw that a SDRAM can be configured in RBC mode to ease the mapping of the data on the banks and to facilitate the computation of access time. Nevertheless, as shown in Section 4.1.3.3, some SDRAM delays are global to the rank or the controller. As a consequence, access to one bank can interfere with access to another bank.

To be used in real time systems, the SDRAM accesses has to be made time-predictable in the sense that a tight bound of the access time can be computed. The solution is either hardware, relying on a time-predictable controller, or software by enforcing constraints on the requests.

The predictable controller schedule the SDRAM commands in the way that the accesses time is tightly bounded and bandwidth is guaranteed. Predator [2] is a controller with credit and static priority ensuring predictable latency and minimal bandwidth. Predator is based on access groups which are sequences of read or write performed on all the banks. These sequences are designed to have a constant time. For instance, the maximum size of an access is constant. A special group is reserved to the SDRAM refresh and scheduled when needed.

Reineke *et al.* [111] presented the PRET DRAM Controller which is time-predictable. The isolation between the masters is guaranteed by private bank allocation. Contrary to Predator it uses the property that banks and ranks can be considered as independent resources and performs the accesses to these resources in a pipelined manner. Usually, the memory controller rely on the refresh command which refreshes all the memory raws at once. The PRET DRAM Controller, instead, performs individual refreshes for each row. This minimize latency since refresh are scheduled alternatively with the requests.

We saw solutions relying on hardware modification. These solutions are not applicable to COTS hardware. Some works are able to provide an upper bound on access time for these COTS hardware under special configuration or constraints on access patterns.

Kim *at al.* [81] describe an analysis method to compute delays due to SDRAM memory interferences. The consequence of the reorder core is taken into account in the analysis. Only the maximum number of requests for a task is required. This paper does not provide a method to minimize these interferences.

Perret *at al.* [105] consider the computation of the delay of the maximum access time which corresponds to a read which follows a write that do not target the same bank. This delay makes no hypothesis on the concurrent access and is used as reference for the computation of the application delay. The authors state that this delay can be reduced by static scheduling of the requests.

In this section, we saw that even if the SDRAM is complex, there exist methods able to compute tight worst-case access latencies for COTS multi-core.

4.2 Network-on-Chip

Network-on-Chip is seen as the solution for the scalability of multi- and many-core communications [12]. It is power efficient and ensures a low latency. Nevertheless, like off-chip networks (Local Area Network or industrial networks), it suffers from the deadlock problem.

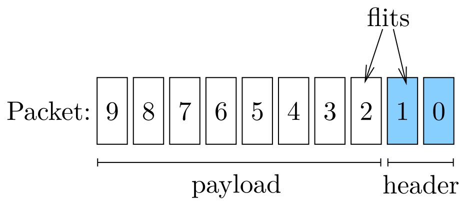
In this section, we concentrate on the *wormhole switching networks* recognized for their small latency and silicon area usage.

The classical topology is the 2D grid and most of the routing algorithms are designed for this topology. In this work, we mostly consider this topology since it is possible to form a 2D grid with a subset of the Kalray MPPA-256 Bostan topology (see Section 4.3.3).

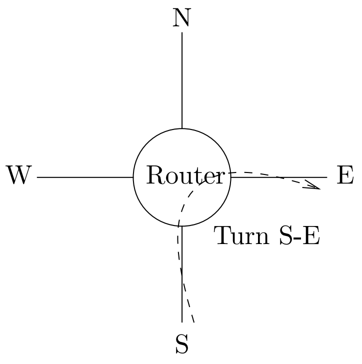
4.2.1 Definitions

In this section, we define the vocabulary required to understand the network-on-chips.

- **Packet/flits Header/payload**: A packet is the routing unit. It is composed of flits which corresponds to the width of the links. The first flits are the header carrying the route information and the destination, the others are the payload (see Figure 4.5a).



(a) Packets of wormhole networks are composed of flits. The first flits are the header, the next are the payload.



(b) Example of Turn.

Figure 4.5: Packet, Flits, Routers and Turn

- **Router**: The router reads the header of a packet and forwards the packet to the next router. The network is packet-switched meaning that the packets are forwarded entirely and cannot be split.
- **Link**: A link connects two routers. In our work, its bandwidth is 1 flit/cycle.
- **Grid**: A 2D-grid is a regular network where routers are organized in grids. A router is connected with 4 other routers at maximum: North, South, East, West. It is connected to the cluster with the Local link.
- **Turn**: A turn is defined with a router, a source and a destination direction. Figure 4.5b shows the South-Est turn for a router. There are 8 different turns: the 180-degree turn is forbidden and the straight directions (North-South, East-West) are not turns. In addition, there are 8 extra turns between between the Local link and the North, South, East and West directions.
- **Unicast/Multicast Flow**: A unicast flow is a tuple with one source and one destination. A multicast flow has several destinations. A flow has one bandwidth and one route even though several routes may be possible for a given flow.
- **Route**: A route is a list of direction taken by a packet to reach the destination.

4.2.2 Routing and Deadlock

A route is static if it is known when the packet is sent. This static route is often embedded in the packet header. If it is not embedded in the packet, it can be preset in each router. In this case, we talk about table-based routing (as in AFDX [85]), since, at each router, a static table is read to define the next link to take.

A route is dynamic if it is not known when the packet is sent. In this case, a piece of software computes the destination in each router. The main advantage of this solution is that route can be adapted in case of network congestion. It is widely used in best-effort networks.

In this thesis, our aim is to be able to bound network latency. This requires to know the route at compile time. Hence, we are only interested in static routing method.

We compare *store-and-forward* NoC method and wormhole switching NoC. The functioning of store-and-forward is simple: packets are sent to the next router after they are fully received. The advantage is that analysis can be done on the entire packet. For instance, the packet can be dropped if the error correction code (ECC) failed or the cycle redundancy check (CRC) is invalid. The drawback is double: the latency is high since packet has to be fully received before being sent and the routers' buffers have to be able to store the full packet.

Wormhole switching is usually chosen in NoC where the silicon area and the latency are important criteria. Once the first flits of the packet corresponding to the header are received, the package can be forwarded to the destination. The granularity of arbitration in the router is the packet since we cannot mix flits of several packets in the same link.

The router buffers can be smaller than the packet size. This implies that a packet can be spread out across the network and be at the same time in several routers (as a worm in an apple).

Wormhole Switching Deadlock Problem. Wormhole is best for latency and silicon area but suffers from deadlock. As mentioned previously, as flits of packets are forwarded as soon as the header is received, packets can be split along several routers and links. If a cycle is formed, this can lead to a deadlock.

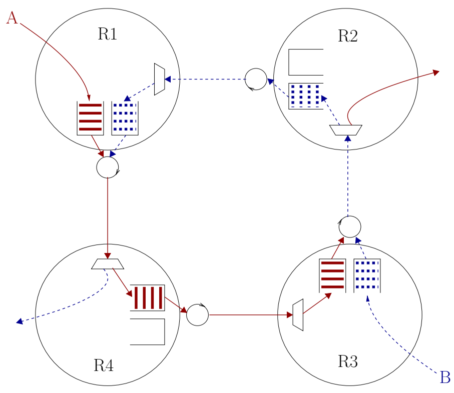


Figure 4.6: Example of wormhole switching network deadlock.

Figure 4.6 shows an example of deadlock in wormhole switching network. We recall that flows are arbitrated at packet level, hence the flits of the packets cannot be mixed. Flow A cannot use the link $R_3 \rightarrow R_2$ because flow B is using it. Also, flow B needs $R_1 \rightarrow R_4$, used by flow A.

Deadlock results from circuit of *agents* and *resources* connected by a *wait-for* relation [40]. In wormhole switching, agents are the packets, and resources are the router buffers and the links between the routers.

Figure 4.7 shows the dependency graph corresponding to the example of Figure 4.6. In this dependency graph, resources beginning with L are links and resources beginning with T are turn passing through a node, *e.g.* T1ES is “a Turn through router R1 from East to South” and T1LS is “the Turn through router R1 from Local to South”. It is clear that there is a cycle due to the sharing of L14 and L32 resources.

Due to this problem, absence of cycle must be ensured in the dependency graph. For set of flows and routes it is possible to compute whether the application can deadlock, nevertheless, a better approach is to guarantee with specific routing algorithm that no deadlock will occur.

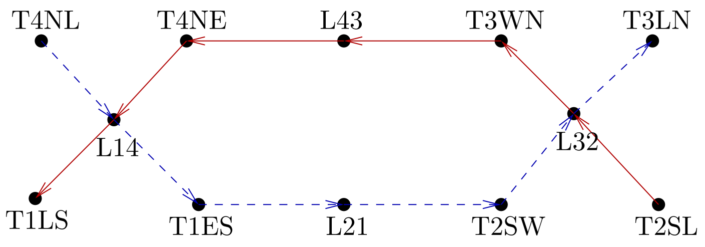


Figure 4.7: Resource dependency graph of example Figure 4.6. Arrows are *wait-for* relations.

Feed-forward Network. A network in which it is not possible for the flows to create cycles is called *feed-forward.* The deterministic network calculus (DNC) usually requires feed-forward networks[3]. The networks are in general non feed-forward. Algorithms such as spanning tree or turn-prohibition transform a network into a feed-forward network by preventing usage of some links or turns. Then, the DNC can be applied.

Most of the deadlock-free algorithms guarantee that the flows routed on a network do not form any cycle even though the network is non feed-forward. If for a non feed-forward network, the flows are routed with an algorithm preventing the flows to form cycles, the subset of the network used is feed-forward and the DNC can be applied on it.

Formally, a definition of a feed-forward network is that it is possible to find a labeling of its queues such that for any flows through the network, the queues are traversed in an increasing order [71]. A similar definition gives an order on the links instead of the queues [115]. A definition of deadlock-free algorithm is a routing algorithm preventing the directed cycles in the resource dependency graph. This is accomplished by imposing a partial order on the used resources (turns and links) and then, ensuring that flows traverse resources in increasing order [41]. Then, if we consider only the subset (links and turns, or queues) of a network used by a deadlock-free traffic. If this traffic has no cycle in the resource dependency graph, hence the subset of the network used by this traffic is feed-forward.

4.2.3 Adaptive Routing

Considering two different routing algorithms for a grid: XY which computes a route composed of a segment on the x-axis followed by a segment on the y-axis; and *shortest path* which computes the routes taking the smallest possible number of links. The XY has only one possible route to reach a destination while there can be several shortest path between a source and a destination.

A routing algorithm offering several possible routes to reach a destination has *path diversity.* An algorithm that exploits this path diversity is said to be is *adaptive.*

The term adaptive comes originally from dynamic routing where the route is evaluated at each router to deal with network congestion or faulty links. Nevertheless, we consider that this notion of dynamic routing is orthogonal to the adaptivity and a static routing also takes advantage of the adaptivity of an algorithm by offline selection of the best route.

Another orthogonal notion is the *path-splitting.* A routing algorithm is said to be path-splitting if when several routes are possibles, they can all be used at the same time. The advantage is double: balancing the charge among several paths and ensuring resilience in the case of network failure. Nevertheless, this method does not ensure ordering of the packets. In our work, we consider that a flow takes only one route. Criteria for selecting the routes can be bandwidth, link sharing or route length for instance. More details are given in Section 7.5.1.

4.2.4 2D Grid Deadlock-Free Routing Algorithms

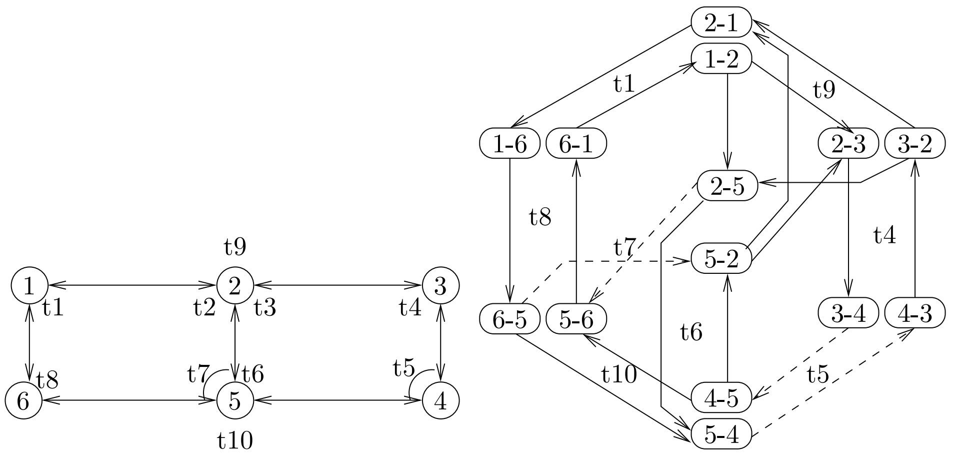
In this section, we present the main deadlock-free algorithms for wormhole switching for both unicast and multicast. Most are designed for 2D grid such as XY, HOE, OddEven and some are designed for arbitrary topologies such as Simple Cycle Breaking or TurnProhibition. These algorithms are explained later. We restrict this classification to static routing algorithm since they better correspond to the predictability requirements.

Dimension-Ordered. The simplest deadlock-free routing algorithms are the *dimension-ordered.* They are called dimension ordered since packets are routed along a first dimension, followed by a second dimension, etc. In a 2D topology there are only two of them: XY which routes the packet along the x-axis first and then along the y-axis; and the YX which routes the packet along the y-axis then along the y-axis.

[3]Methods such as stopped-time allow applying DNC on non feed-forward networks.

Intel Knight Lending architecture has a 2D-mesh with YX routing [121]. Tilera TILE64 is routed in a dimension-ordered way [11].

This kind of routing if often chosen for its simplicity, nevertheless, they are non-adaptive hence they do not allow route selection optimization. We now focus on *adaptive* routing that provides path diversity.



(a) Network with prohibed turns (b) Turnnet equivalent

Figure 4.8: Turnnet transformation

Cycle Breaking Algorithms. We now present another kind of algorithm able to transform an arbitrary network topology of bi-directional links and turns into a feed-forward network, *i.e.*, in which no cycle are present. The source of these algorithms is the spanning tree [117] which disables some links in the network to obtain a tree and hence remove cycles.

The Turn Prohibition algorithm [122] finds a set of prohibited turns ensuring the feed-forward property of a network. It works for an arbitrary network topology of bi-directional links and bi-directional turns. This algorithm guarantees the prohibition of at most a third of the turns. This feed-forward property is required by most of the methods to compute bounds on the network FIFOs.

Simple Cycle Breaking [90] is an improvement of Turn Prohibition that has been shown to prohibit less turns.

Once the turns have been prohibited, any routing algorithm can be implemented, in particular the shortest path. But, implementation of the shortest path for these topologies is not straightforward. In order to ease implementation of routing under Turn Prohibition, Fidler *et al.* [51] introduce the Turnnet, which is a graph whose vertices correspond to the external links, and edges to the internal turns of the network. Application of any cycle-breaking technique to ensure feed-forward network amounts to removing edges from the Turnnet so it becomes acyclic. It is then used to compute routes, for instance by applying Dijkstra's shortest path algorithm [51].

Example 11. In the 2D grid network of Figure 4.8a, turns t7 and t5 marked with a circular arc has been removed to prevent cycle. The turns have been removed in both ways *e.g.* from l4 to l3 and from l3 to l4.

Figure 4.8 is the Turnnet equivalent of the network in which turns become edges and links become vertices. Forbidden turns are removed from the graph. Computation of the shortest path on this graph can be done by applying Bellmand and Ford or Dijkstra.

Turn Model. The first attempt of providing diversity is by adding physical and virtual channels to the network. Nevertheless, it requires specific hardware. Glass and Ni [59] invented the turn model to bring adaptive routing algorithm while ensuring deadlock-freeness. This model allows expressing all the possible turns in the network (South to East, South to West, etc) and to restrict usage of some turns. Note that the 180-degree turn is useless hence always forbidden while straight lines (North-South, East-West) are always authorized.

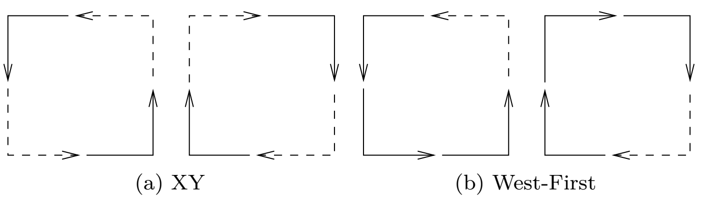


Figure 4.9: Turn Model for the XY and West-First routing algorithms. The square represents the four possible turns. The dashed turns are forbidden.

The authors show that only a quarter of the turns are needed to be forbidden to ensure deadlock-freeness. Note that all the combinations of forbidden turns do not lead to deadlock-freeness. They present several adaptive algorithms such as West-First, Negative First and North-Last. XY is not optimal since it forbids half of the turns where West-First forbids only a quarter. Figure 4.9 shows in dashed the prohibited turns and in plain the authorized turns. The squares show for a turn, all the possible turns that can be taken next.

With *West-First*, a packet has the right to go to the West direction at the beginning and then it can follow any other directions except West. Figure 4.9b shows the turn model of this algorithm where the South to West and North to West are forbidden. We explain how to understand the turn model of West-First. If the first turn is West to South from the left square, the packet can only go to the East, then to the North. Then, from the second square, the turn North to East can be taken and the packet can go to the East, then to the South.

Figure 4.9a shows the turn model of the XY routing algorithm and the dashed turns are forbidden whereas the plain turns are authorized. The figure has to read as follows: for a XY routing, the left or the right square can be selected. If the left is selected, the packet can go to the East then to the South or to the East then to the North.

We now explain the *Negative-First* algorithm. The x and y coordinates of the routers are numbered as follows: for a node of coordinate (x,y), the node on the right has the coordinate (x+1, y) and the node on the left has the coordinates (x-1, y). The Negative-First algorithm is done in two steps: first the packet can go to any node carrying smaller coordinates (x-1, y) or (x,y-1), then the packet can go to router with higher coordinates (x+1, y) or (x, y+1).

Odd-Even Turn Model. Chiu [34] shown that the adaptivity of the turn model is variable, *i.e.*, some flows has a high path diversity while some other has a small or no path diversity. As a consequence the author present the odd-even turn model which provides an homogeneous path diversity. The principle is that there are two sets of prohibited turns depending if the current column number is odd or even.

Multicasting. Multicast consists in sending the same packet from one source to several destinations. There are three kinds of multicast algorithms: The *unicast-based multicast* where the packet is sent independently to every destination. The main advantage of this solution is that it preserves the deadlock-freedom of the unicast algorithm. The main drawback of this solution is that the packet is duplicated for every destination. In the *tree-based multicast*, when a router forwards a packet, it can decide to duplicate this packet to several destinations. The third solution is the *path-based multicast* where the destination is split into several partitions. For each *partition*, a route passing through all

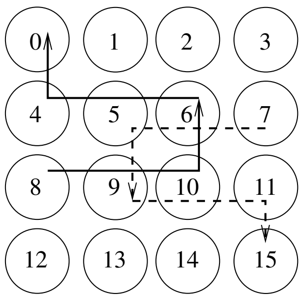


Figure 4.10: Example from [42] leading to deadlock by concatenating XY routes.

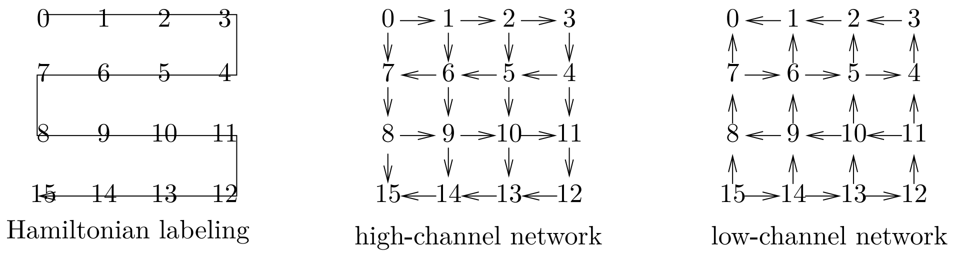


Figure 4.11: Hamiltonian Labeling and Subnetworks

the destinations of this partition is computed. A special field in the header indicates if the packet has to be forwarded by the router to the node.

Path-based multicast offers better performance than unicast- and tree-based multicast [7]. It is also widely used. For these reasons, we only detail this method in the rest of the section.

Path-based Multicast and Partitions. In each partition, a route passing through every destination has to be computed. The easiest solution would be to concatenate several routes. Nevertheless, concatenating several routes computed with a deadlock-free algorithm does not lead to a deadlock-free route, therefore specific algorithms have to be used. An example of deadlock due to the concatenation of routes from [42] is given in Figure 4.10. Both flows require links 6-5 and 9-10. If dashed flow blocks Link 6-5, plain flow blocks Link 9-10 and consequently dashed flow is blocked, etc. There is a cyclic dependency in the resource graph.

The first deadlock-free multicast algorithm has been proposed by Lin, McKinley and Ni [91]. It is a path-based multicast based on a Hamiltonian labeling of the network.

Hamiltonian Routing. As depicted in Figure 4.11, the Hamiltonian labeling passes through every node in the network exactly once. Several Hamiltonian paths exist for the same network.

In Hamiltonian routing there are two separated networks represented in Figure 4.11: the network following the label in increasing order called high-channel network and the network following the labels in decreasing order called low-channel network. For a source s and a destination d, the next router n is computed as follow:

If label(s) < label(d):

$$\max_{label(n)} \{label(n) < label(d) \text{ and } n \text{ is a neighbor of } s\}$$

If label(s) > label(d):

$$\min_{label(n)} \{label(n) > label(d) \text{ and } n \text{ is a neighbor of } s\}$$

The Hamiltonian routing algorithm gives the shortest path and is not adaptive.

Common Hamiltonian-based multicast are: *dual-path* and *multi-path*. They are presented in Figure 4.12 where node 9 sends a multicast packet to 2, 3, 5, 7 and 15.

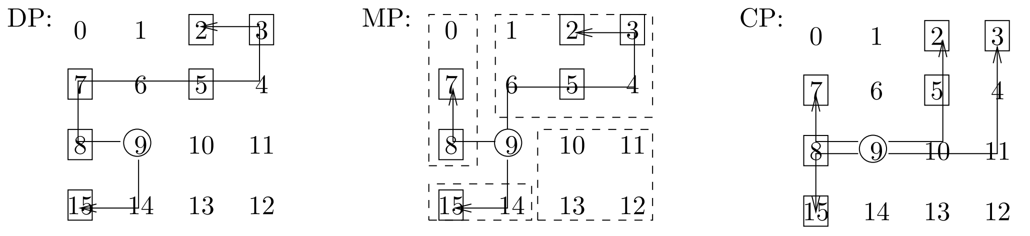


Figure 4.12: Column-Path, Dual-Path and Multi-Path routing algorithms.

Dual-path is composed of two partitions: one for the nodes higher in the Hamiltonian path and one for the nodes lower. In this example, the high partition contains 15 and the low partition contains 2, 3, 5, 7 and 8.

Multi-path is composed of two partitions: one for the nodes whose label is higher and one for the nodes whose label is lower. Then, these partitions are split in two to maximize usage of the output links of the routers: one for the nodes on the left of the source and on for the nodes on the right. The reason for this split in four partitions is that a 2D-grid router has 4 output ports. In this example, there are 4 partitions: (2, 3 and 5), (7 and 8) and (15).

Column-path routing algorithm has been present by Boppana *et al* [19] to authorize dimension ordered routing algorithms in multicast. Two partitions are computed for each column. Each column is accessed with two XY route: one for the nodes above the source and one for the node below. In the example of Figure 4.12, there are 4 partitions: (2,5), (3), (7, 8), (15).

Adaptive Path-based Multicast. Daneshtalab *et al.* [42] propose HAMUM, an improvement of the Hamiltonian routing algorithm to make it adaptive by selecting among the Hamiltonian shortest paths. They present Adaptive Multi-Path (AMP), later called HAMUM MP and Adaptive Column-Path (ACP) later called HAMUM CP. The partitioning is the same as in MP and CP but this routing algorithm brings path diversity.

Bahrebar *et al.* [7] shown that HAMUM can be improved for the first step in the high partitions and for the last step in the low partitions. They introduced the Hamiltonian Odd-Even (HOE) to increase the adaptiveness of HAMUM while authorizing both unicast and multicast routing. HOE CP and HOE MP are presented.

As mixing deadlock-free algorithms can lead to deadlock, a single algorithm must be chosen and has to handle both unicast and multicast if necessary. CP+XY, HAMUM or HOE are possibilities.

4.2.5 Bandwidth Limiter

It is necessary to know the bandwidth of the flows to bound their latency. This is also required to ensure that the router FIFOs do not overflow. Network architectures for safety-critical systems such as the AFDX network [85] and the Kalray MPPA2 provide a bandwidth limiter.

One model to limit bandwidth of a flow is the leaky bucket. The principle is that at each cycle, the budget of each node (or flow) increases. A node sends a packet only if it has enough budgets and the budget is reduced accordingly. An advantage is that a node benefits from more budgets if it has not sent packet for a long time. Obviously, there is a limit for the accumulated budgets which is called *burstiness*. This policy can be easily implemented in hardware.

Example 12. Figure 4.13 shows an example of two nodes with a limited bandwidth. The x-axis

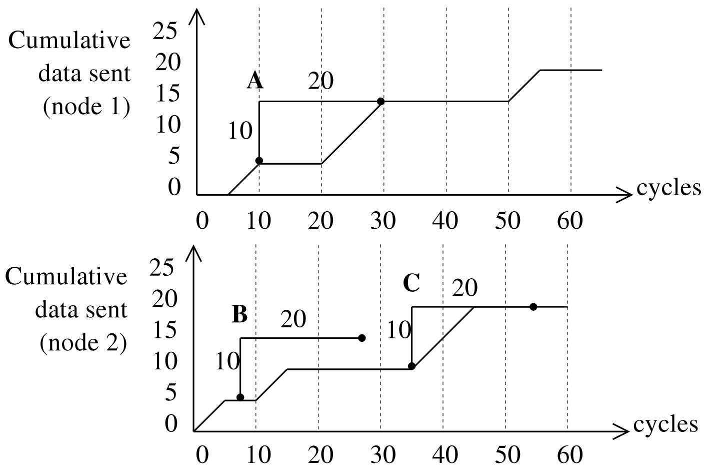


Figure 4.13: Leaky bucket bandwidth limiter.

is the number of cycles; the y-axis is the cumulative number of 32-bits flits sent. Both nodes have a budget b = 10 cycles in a sliding window w = 20 flits. Hence, they cannot send more than 10 flits in this sliding window. For Node 1, A represents the window from t=10 to t=30. At t=30, Node 1 emission is blocked since it has already sent 10 flits in the last 20 cycles. Node 2 is blocked before the end of the window C. The window B, shows that Node 2 does not use the entire budget: it could send 5 flits more during window B.

If we combine the bandwidth limiter with static routes, maximum bandwidth of each flow can be known. Then, bandwidth usage of each link can be computed as the sum of the bandwidth of the flows using this link. This bandwidth usage is compared to the capacity of the link. We talk about *bandwidth reservation* when the maximal bandwidth of the flows is known and the link capacity is statically shared among these flows.

A link is *saturated* if the sum of the rates of the flows taking this link equals the capacity of the link. If the link is *unsaturated*, the bandwidth of at least one flow could be increased.

We call *saturated flow*, a flow taking at least one saturated link. An *unsaturated flow* takes only unsaturated links, thus its bandwidth could be increased without reducing the others.

4.2.6 Max-Min Fairness and Lexicographic Vector

In the previous sections, we described some routing algorithms with path diversity and we presented the hardware bandwidth limiter. In this section, we discuss the fair flow rate attribution.

The maximum capacity of a link depends on the nature of the network. In wormhole switching networks, this capacity is given in flits. In this section, we consider that a link transmits one flit per cycle.

The fairness in a network is often reached at the cost of reducing the total bandwidth. A bandwidth allocation is said to be *max-min fair* iff an increase of any bandwidth must be at the cost of a decrease of some already smaller bandwidth [33].

A well know algorithm to solve max-min fair allocation for unsplittable path is the " *Water Filling*" algorithm [17, 4] also known as "*Progressive Filling*" algorithm. An instance of this algorithm and a comparison with rate-sum maximization has been proposed in Jafari *et al* [78]. The Rate-sum policy maximizes the sum of the rates without any guarantee of minimal rate for the flows. This policy can lead in the worst case to some flow with a bandwidth of 0 while in the worst case, max-min fairness ensures the same rate for all the flows.

Resolution of max-min fairness problem with splittable flow (MMFSP) can be done in polynomial time using series of linear programming [98].

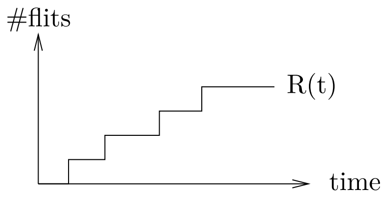


Figure 4.14: Cumulative input function $R(t)$ representing the data produced by a source in the network.

This linear program maximizes a `min` variable that is less than each sub-flow rate. As the flows are split into several sub-flows, the sum of these sub-flows is 1. After execution of the linear program, the saturated flows have the value `min`. The bandwidth of saturated flows are set to `min` and removed from the maximization function. These steps are repeated until all the flows are saturated.

We computed several set of routes. We need a mean to compare them. Comparing the average rate of the flows for a given set of routes does not guarantee to select a fair solution. For instance, the solution with the worst minimal rate can be selected. If the minimal rate is chosen as the criterion of comparison, it is possible that there are several solutions with the same minimal rate. In this case, the second one has to be compared, etc. In [98], the *lexicographical vector* is used to compare the solutions. This vector is composed of the sorted rates allowing a fair comparison.

Lexicographic vectors are sorted according to the rate ($\forall i < j, v_i < v_j$). Comparison of vector v and w is done in increasing order until one rate is greater than the other. Formally:

$$v > w \Leftrightarrow \exists n, \forall i < n, v_i = w_i \text{ and } v_n > w_n$$

Routing and bandwidth allocation are required to bound delays on the NoC. In the next section, we present one theory for real-time networks.

4.2.7 Network Calculus

In this section, we provide a brief introduction to the Deterministic Network Calculus (DNC). We do not claim to be complete but sufficiently to make this manuscript understandable. We advice the lecture of the reference book from Le Boudec and Thiran [86] for more detailed explanations.

Deterministic Network Calculus is a theory invented to compute end-to-end delays and buffer size in Asynchronous Transfer Mode (ATM) networks. More recently it has been used for Avionics Full Duplex Switched Ethernet (AFDX) networks certification [56, 65].

The principle of DNC is that, from a model of the network elements and from a model of the traffic or the exact traffic curve, properties can be deduced from this network. If maximum information is known, the bounds can be accurate.

A flow is represented by a cumulative input function $R(t)$ which represents the number of byte/flit produced by a source as represented in Figure 4.14. Normally, this function is discrete; nevertheless, this can be over-bounded with a continuous function. Input function in Figure 4.14 is a staircase function due to the functioning of a network. The output function of a network element is called $R^*(t)$ and corresponds to the transformation of $R(t)$ by this network element.

Notations.

- $[x]^+ := max(0, x)$
- $1_{\{y\}}$ is 1 when y is true, 0 otherwise.
- The infimum of a set S is the largest value x such that $\nexists v \in S, v < x$. If the minimum of this set exists, then $\min S = \inf S$.

Min-Plus Algebra. DNC is based on the min-plus algebra. Classic algebra is based on addition and multiplication. In min-plus algebra they have been replaced with infimum which is the greatest lower bound (or minimum if exists). This algebra is noted $(\mathbb{R} \cup \{+\infty\}, \wedge, +)$ where $\wedge$ is the minimum computation and $+$ is it addition.

Example 13. $(3 \wedge 7) + 2 = 3 + 2 = 5$

Now, we present the most useful operation of this algebra: the min-plus convolution. It describes operation on curves. This operator is associative, hence, we can reason on two curves without loss of generality.

Min-Plus Convolution

$$(f \otimes g)(t) = \inf_{0 \le s \le t} \{f(t-s) + g(s)\}$$

and 0 for $t < 0$.

Figuratively, the min-plus convolution of two curves a and b corresponds to sliding b on a over the time (the point (0,0) of b is a point on a) and keeping the minimum between them. This sliding of the curve b corresponds to the operation performed in Section 4.2.5 to check that the traffic complies with the leaky bucket. Better, the min-plus convolution describes exactly this process of traffic limitation. That is why the min-plus algebra has been chosen.

Min-Plus Deconvolution

$$(f \oslash g)(t) = \sup_{u \ge 0} \{f(t+u) - g(u)\}$$

Arrival Curve. To be able to guarantee end-to-end latency, traffic has to be bounded in the network. An arrival curve is a function α defined for $t \ge 0$. Input function R is bounded by a service curve if $R(t) - R(s) \le \alpha(t-s)$ [86].

Using this algebra, R is constrained with an arrival curve α iff:

$$R \le (R \otimes \alpha)$$

The arrival curve of a leaky bucket (see Section 4.2.5) is an affine curve expressed with:

$$\gamma_{r,b} = \begin{cases} rt + b & \text{if} \quad t > 0 \\ 0 & \text{otherwise} \end{cases}$$

The network element ensuring that the input curve is compliant with the arrival curve is called greedy shaper. A variant is the *packetized greedy shaper*, ensuring the transmission of packets of size $\le l_{\max}$ and these packets are delayed to ensure the arrival curve.

This maximal packet size constrains the arrival curve: if $\alpha(0) < l_{\max}$, the packets can stay in the buffer forever. Hence, as stated by Proposition 1.7.1, p59 of [86], $\forall t > 0, \alpha(t) \ge l_{\max}$. Hence, for a leaky bucket $\gamma_{r,b}$, $b \ge l_{\max}$.

Service Curve. We saw that an arrival curve provides an upper bound of the input curve. The minimum service curve is a lower bound of the traffic handled by a network element. The minimum service curve is often simply called service curve. Considering a system with an input R and an output R^*, if the guarantee offered by this system is the service curve β, then the output can be expressed as follows:

$$R^* \ge (R \otimes \beta)$$

An example of service curve is the peak rate function λ_R which ensures a fixed rate as output. This function cannot be implemented in reality since it does not introduce delay and is thus only theoretical.

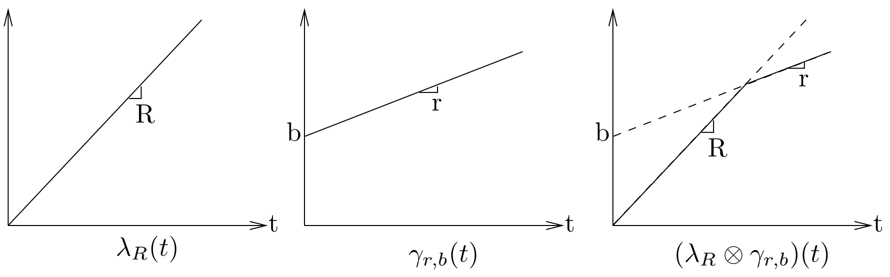


Figure 4.15: Min-plus convolution $(\lambda_R \otimes \gamma_{r,b})(t)$.

$$\lambda_R = \begin{cases} Rt & \text{if} \quad t > 0 \\ 0 & \text{otherwise} \end{cases}$$

Example 14. Let us consider a flow characterized by an arrival curve $\gamma_{r,b}$ entering a system with a service curve λ_R with the constraint $r \leq R$. Let us compute $(\gamma_{r,b} \otimes \lambda_R)(t)$.

For $t < 0$:

$$(\gamma_{r,b} \otimes \lambda_R)(t) = 0$$

For $t > 0$:

$$(\gamma_{r,b} \otimes \lambda_R)(t) = \inf_{0 \leq s \leq t} \{\gamma_{r,b}(t-s) + \lambda_R(s)\}$$

$$= \inf_{0 \leq s < t} \{\gamma_{r,b}(t-s) + \lambda_R(s)\} \wedge \inf_{s=t} \{\gamma_{r,b}(t-s) + \lambda_R(s)\}$$

$$= (\gamma_{r,b}(t) + \lambda_R(0)) \wedge (\gamma_{r,b}(0) + \lambda_R(t))$$

$$= (rt + b + 0) \wedge (0 + Rt) = (rt + b) \wedge Rt$$

A graphical representation is given in Figure 4.15.

The rate-latency function $\beta_{R,T}$ is a service curve with a limited rate R and a delay T. Note that, $\beta_{R,0} = \lambda_R$.

$$\beta_{R,T} = \begin{cases} 0 & \text{if} \quad t < T \\ Rt & \text{otherwise} \end{cases}$$

Convolution is shown graphically on Figure 4.15.

If several flows aggregate into one, the arrival curve is the sum of their service curves.

Delay and Backlog. Most of the DNC theorems work only if the buffers of the network elements are large enough. Nevertheless, physical network have constraints on buffer size. Hence, an interesting value is the backlog which is the amount of data in the system, considering an input function R and an output function R^*:

$$backlog(t) = R(t) - R^*(t) \leq \sup_{s \geq 0} \{\alpha(t) - \beta(s)\}$$

This corresponds to the maximal vertical deviation between the input and the output functions.

To compute the delay, a constraint is that packets are delivered in FIFO order by the system. This delay is the maximal horizontal deviation between the input and the output functions.

$$d(t) = \inf_{t \geq 0} \{R(t) \leq R^*(t+s)\}$$

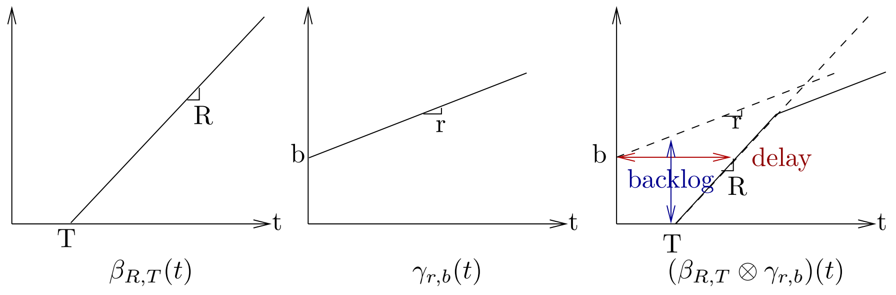


Figure 4.16: Min-plus convolution $(\beta_R \otimes \gamma_{r,b})(t)$. Maximum horizontal and vertical deviations show respectively the delay and backlog bound.

Figure 4.16 shows the graphical computation of the delay and the backlog of this system.

Delay and Backlog Bounds Under Multiplexing. A *multiplexing network element* merges several flows. This process of multiplexing different flows is called *arbitration* or *scheduling*. Some multiplexing network elements guarantee the temporal ordering between the packets of different flows, *i.e.*, the order of the packets at output is the same as the order of the packets at input. Furthermore, an assumption for the DNC analysis is that the packets of the same flow are handled in FIFO order.

The multiplexing policy of a greedy network element can be abstracted only if this abstraction is conservative. There are several ways for abstracting the multiplexing policy of the network element: priority-based, blind, FIFO, etc. The blind multiplexing is used when there is no information about the packet arbitration. This is equivalent to the priority-based multiplexing where the flow of interest has the lowest priority. If the aggregate input flow rate is smaller than the rate of the service curve, the time requires by a packet pass through the arbiter is finite.

Theorem 6.2.1 from [86] *Consider a node serving two flows, 1 and 2, with some unknown arbitration between the two flows. Assume that the node guarantees a strict service curve β to the aggregate of the two flows. Assume that flow 2 has α_2 as an arrival curve. Define $\beta_1(t) := [\beta(t) - \alpha_2(t)]^+$. If β_1 is wide-sense increasing, then it is a service curve for flow 1.*

The FIFO multiplexing can be used if the network element ensures ordering between the packets of the two flows. In other words, FIFO applies if a packet coming at t is transmitted after all the packets arrived strictly before t.

Proposition 6.4.1 from [86] (FIFO Minimum Service Curves) *Consider a lossless node serving two flows, 1 and 2, in FIFO order. Assume that packet arrivals are instantaneous. Assume that the node guarantees a minimum service curve β to the aggregate of the two flows. Assume that flow 2 has α_2 as an arrival curve. Define the family of functions β^1_θ by*

$$\beta^1_\theta(t) = [\beta(t) - \alpha_2(t-\theta)]^+ 1_{\{t>\theta\}}$$

Call $R_1(t)$, $R'_1(t)$ the input and output for flow 1. Then for any $\theta \geq 0$

$$R'_1 \geq R_1 \otimes \beta^1_\theta$$

As recalled in [86], the minimal β^1_θ when $\theta \geq 0$ has to be considered for computation.

Tandem of Network Elements. In this chapter, we saw how to compute the service curve of one network element and its consequence on flows. In real networks, the flows pass through several network elements. The DNC gives a direct computation of the service curve resulting of the concatenation of these network elements.

The service curve of a tandem of two of servers with respective service curves $\beta_1(t)$ and $\beta_2(t)$ is $(\beta_1 \otimes \beta_2)(t)$ This formula can be applied to series of network elements.

DNC is a complete framework to compute backlog and delay bounds of flows within a network. To be tightly analyzable, a network has to fulfil some requirement such that the ordering of the packets in the links.

DNC is a very general framework. For a particular network, a subset can be sufficient. We use a very simple subset of the DNC for bounding the delays on the Kalray MPPA2 NoC, nevertheless this provide good results.

4.2.8 Classical Artificial Instances of Flows

The efficiency of a routing algorithm depends on the application. As a consequence, the only way to compare algorithms for NoC is to compare them on the same instance of flows. There exist plenty of classical instances of flow for 2D grid networks. In this section, we present some well-known traffics that we use in our work to compare routing algorithms.

Unicast. There are four classical instances of flows [6] often used for unicast routing algorithm comparison.

Figure 4.17a shows Bit-Complement which consists in creating flows between each source of id S and their ones' complement D (bit-wise not).

$$D = not\ S$$

Figure 4.17b shows Bit-Reverse which consists in creating flows between each source S and the destination D corresponding to the reversed bits of S.

$$\forall i \in [0, B], D(i) = S(B - i)$$

with B, the number of bits to encode the id (this implies power of two numbers for a grid of dimension N).

Figure 4.17c shows Shuffle which consists in creating flows between each source S and the destination D with the following relation:

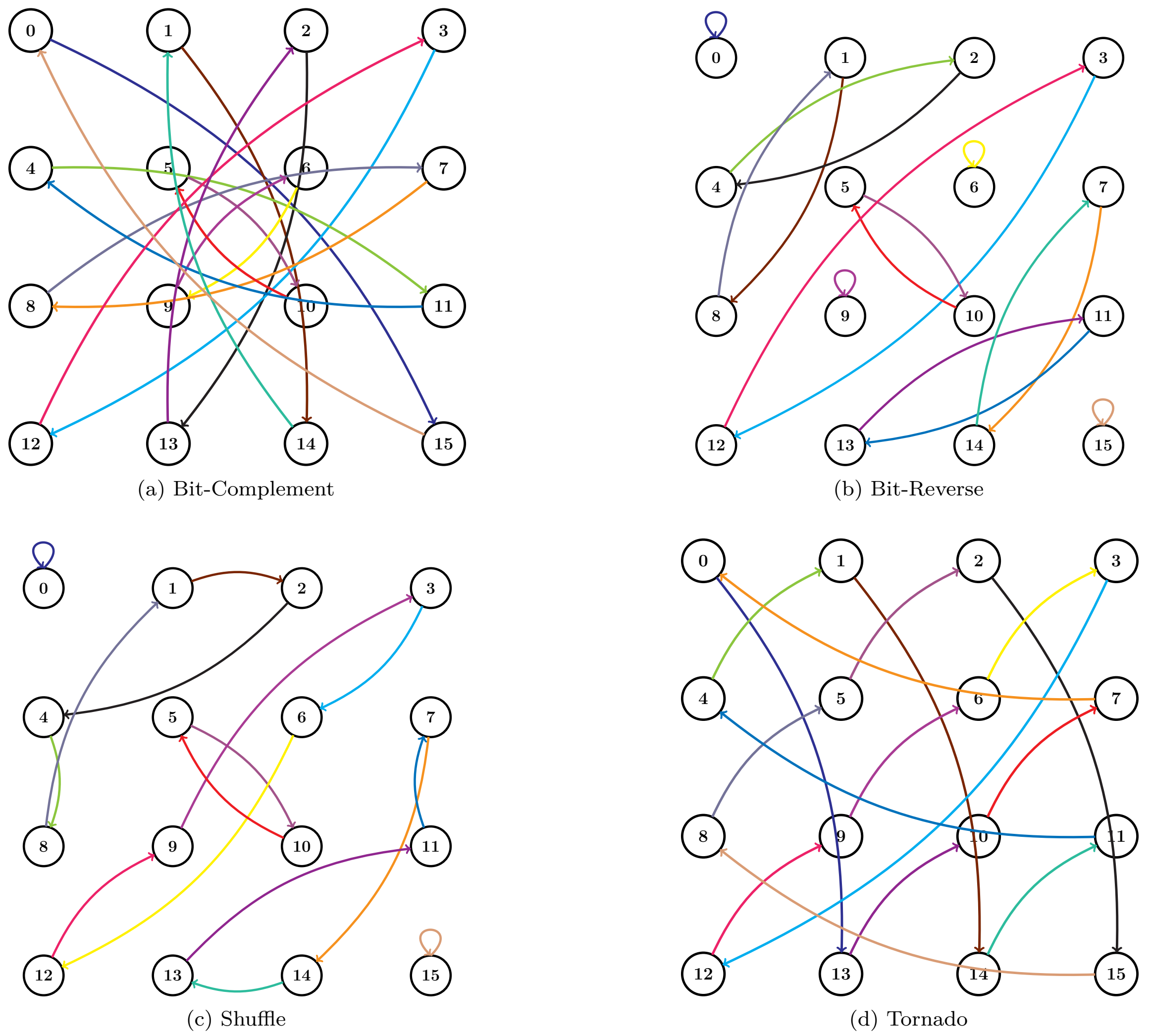
$$D = ((S << 1)\&(N-1))|((S >> (log2(N)-1))\&1);$$

with N, the number of nodes. As in C, << and >> are bit shifts, & and | are respectively `and` and `or` operations.

Figure 4.17d shows Tornado. Previous instances of flows were defining a relation between identification numbers S and D of the nodes whereas Tornado defines a relation using the position of the nodes on a grid. A relation between the coordinates (S_x, S_y) and (D_x, D_y) is defined. The dimension N of the grid must be power of two.

$$D_x = \left(S_x + \frac{\sqrt{N}}{2} - 1\right) mod\sqrt{N}$$

$$D_y = \left(S_y + \frac{\sqrt{N}}{2} - 1\right) mod\sqrt{N}$$



(a) Bit-Complement

(b) Bit-Reverse

(c) Shuffle

(d) Tornado

Figure 4.17: Classical Instances of Flows

Multicast. For multi-cast, the unicast instance of flows do not correspond since some are symmetrical (as bit-complement or bit-reverses), hence they are not adapted to connect more than two nodes.

One classic method to generate flows is the *uniform* traffic where each node has the same probability to reach any node. An alternative is *hotspot* where some nodes have a higher probability to be reached.

Rent is a law where nodes close to each other have a higher probability to communicate [18]. The following formula from [18] gives the probability of two nodes to communicate in function of their distance d.

$$P(d) = \frac{1}{4d}\left[(1 + d(d-1))^p - (d(d-1))^p + (d(d+1))^p - (1 + d(d+1))^p\right]$$

with p the Rent coefficient. Measures on actual application have shown that coefficient between 0.55 and 0.75 are representative of real applications [7].

4.3 The Kalray MPPA2

In this section, we present the Kalray MPPA2 with emphasis on the time-predictable configuration. This is the platform used for the experiments. As a consequence, the features useful for high-performance computing (HPC) usage of the processor are eluded but the feature required to make this document self-contained are presented.

The Kalray MPPA-256 Bostan[4] also called MPPA2 is a commercial many-core processor. It has been designed to offers high performance [77], low power consumption and predictability.

4.3.1 Cores

4.3.1.1 VLIW vs. Out of Order Pipeline

The cores of the Kalray (called k1) are Very Long Instruction Word (VLIW). This technology ensures good performance thanks to instruction-level parallelism while relying on in-order pipeline. For out-of-order pipeline, a dynamic engine schedules the ready instructions on the free execution unit. This leads to unprecise WCET analysis [127].

A VLIW processor has numerous instruction level parallelisms and the processor is not responsible for the schedule [53]. A VLIW core executes bundles that correspond to an aggregate of simple instructions. The content of a bundle is limited by execution units presents in the core and by the data-dependencies. On the Kalray cores, the execution units can be arithmetic logic unit (ALU), load/store unit (LSU), floating-point unit (FPU) or branch unit (BCU). We can highly benefit from instruction level parallelism if there are several of them. A reservation table gives the required execution unit for each instruction.

The construction of the bundles is done at compile time and this construction is similar to a two-dimension bin-packing problem where one dimension is time, and the other dimension is the execution units. The execution order of the bundles on the processor is the same as the order of the bundles in the binary.

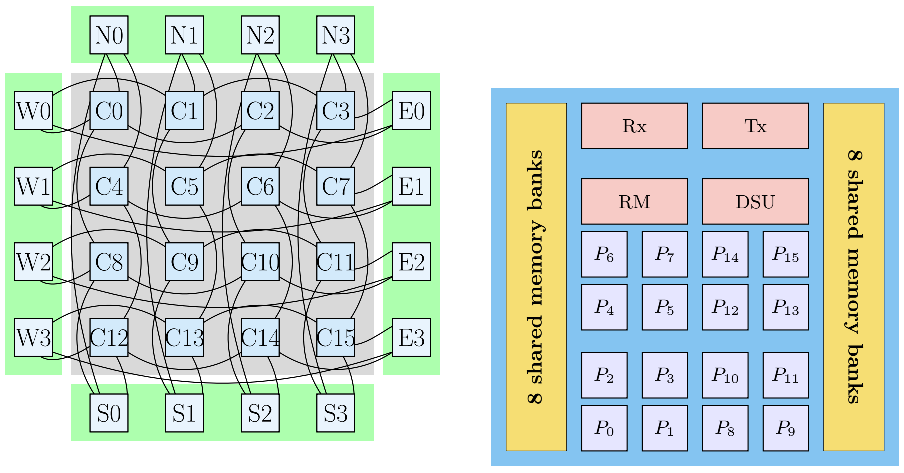
An optimized schedule for the instructions in the bundles can be complex to compute, but this computation is done at compile-time and not by hardware. This ensures simple hardware.

Each core has private caches, two timers and can access the network-on-chip.

4.3.1.2 Caches

Cache behavior. Each core has one private instruction cache and one private data cache. Both of them follow the least recently used (LRU) policy which ensures good cache predictability (as explained in Section 4.1).

[4]`http://kalray.eu`



(a) Kalray MPPA2 (b) One compute cluster
(Courtesy of Hamza Rihani)

Figure 4.18: (a) Overview of the Kalray MPPA2 with the 16 Compute Clusters in blue and the I/O Clusters in green. (b) On compute cluster with the 16 cores and the shared-memory.

The data cache (DC) is associated with a write buffer (WB) whose purpose is to reduce the number of writes issued to the memory due to in-place modifications thanks to a merging mechanism.

When data are read, if there is a cache miss, data are fetched from the memory and stored in the DC. If needed the LRU policy evicts the oldest line. When a value is written, it goes to the WB either by modifying the already present value or by going to a free space. The WB tries to always keep one free space to fasten the allocation of a new element.

The cache is not write-allocate, *i.e.*, if the write hits in both the DC and the WB, the modification is applied to both of them. If only the WB hits, the data is not allocated in the DC.

Software-based cache coherency. In multi-core processors, the coherency of the cache is an important parameter. A cache is said non coherent if for the same address, the value can be different at different places of the system. For instance, the address `0x10020` can have a different value in the DC of core 1, in the WB of core 2 and in the memory.

There are two ways to restore the coherency: either with a hardware cache coherency mechanism or by software. In the MPPA2, the caches are non-coherent but the coherency can be enforced with maintenance instruction to force the DC to refresh the data from the memory or to force the WB to commit the data in the memory.

For a hardware cache coherency, the analysis is difficult since a memory access of one core can lead to a cache miss of another core. The software-based coherency optimizes the real-time analysis since each core is responsible for updating its own cache.

4.3.2 Cluster

The Kalray MPPA2 is organized in clusters. Each cluster is comparable a multi-core with its own memory and a set of cores. In a cluster, communication is done in shared memory. Figure 4.18b shows one compute cluster.

There are 16 cores in each compute clusters. They are called PE (Processing Element) named $P_0 ... P_{15}$. Additionally, a resource manager (RM) core can be configured to have more rights than PEs.

A debug support unit (DSU) provides a global clock for the chip and a trace support. The reception unit (RX) and the transmission unit (TX) are responsible of the NoC transmissions.

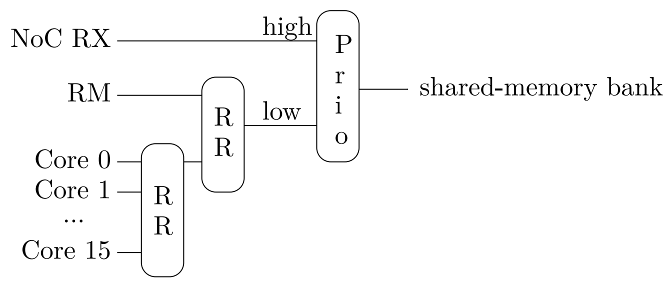


Figure 4.19: Arbiter for one memory bank of a MPPA2 Compute Cluster.

4.3.2.1 Shared-Memory

The shared memory of the MPPA2 is a static random access memory (SRAM). This memory does not need refresh and has a constant access time.

This memory is organized into independent banks such that an access to one bank has no impact of the access time to another bank. The addressing mode can either be configured in blocked mode or in interleaved mode. In blocked mode, the addresses relative to one memory bank are contiguous. This mode eases the mapping of the data in memory. In interleaved mode, the sequential addresses are moving from one bank to another every 64 Bytes. This mode offers a better average performance since load due to sequential access is balanced among the different memory banks.

Each bank has a private memory arbiter. Figure 4.19 shows the three arbitration levels. In the first stage, the priority is given to the data received from the NoC, then a round-robin is made between the resource manager core (RM) and the cores, then a last level arbitrates between the cores in a round-robin way. This RM executes operating system for some specific operations (such as asynchronous NoC transfer), nevertheless if such operations are not used, RM does not interfere with the other cores.

4.3.2.2 Input/Output Clusters

Figure 4.18a shows the I/O Clusters of the MPPA2 in green. They are named North, South, East and West. A topological difference with Compute Cluster is that I/O Clusters have 4 routers, *e.g.*, the routers of the I/O Clusters West are: W0, W1, W2, W3. I/O Clusters have a direct access to the external SDRAM memory, the PCIe (Peripheral Component Interconnect Express), the Ethernet ports and some serial ports. These I/O clusters have only 4 cores and the cache is coherent.

These I/O clusters are the entry point to configure and run the compute clusters. The MPPA2 can be used either as an accelerator for an x86 computer of as a stand-alone card. In this case, an I/O Cluster is the master and can be configured to fetch the code from an external flash memory.

4.3.2.3 Clocks and Synchronizations Mechanisms

In this section, we explain the synchronization mechanisms of the MPPA2. They are necessary to implement time-triggered and event-triggered execution.

Clocks are said *isochronous* if they are equals. Clocks are said *mesochronous* if they have the same frequency but the phase can be different. All the clocks of the MPPA2 have the same frequency.

In each cluster, there is a special timer called debug support unit timer (DSU timer). It is reset when the processor is powered on. As a consequence it is isochronous since it has the same value in all the clusters. Clocks of the cores are mesochronous since they are reset when the core is powered on.

In each cluster, some special registers called *message_ram* are accessible from any core and are able to generate synchronous event to all the cores. The set of subscribers to this event can be configured. A signal is emitted when *message_ram* reaches a special value which is the maximum

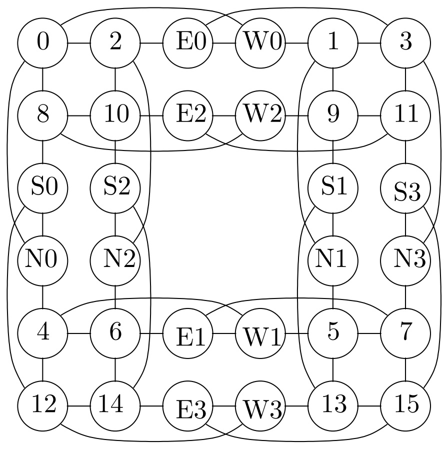


Figure 4.20: A 2D-grid view of the Kalray MPPA2 topology (also known as Pierre's topology).

value (`0xFFF...F`). Any core of the cluster can transmit a value to this register and a bit-wise *or* operation is computed between the current value and the transmitted value.

4.3.3 Network-on-Chip

The MPPA2 has a network-on-chip which is the only means of communicating between the clusters and between the I/O clusters and the other clusters. For instance, if the PE of a compute cluster wants to read data from the external memory, the data will be transmitted through the NoC and an I/O cluster is responsible for this transmission.

The functioning of the NoC is *remote direct memory access* (RDMA) since data can be directly written in distant memory. This remote memory write feature is handled by the hardware directly. The remote memory read has to be implemented with software protocol and special NoC configuration.

Topology and 2D-Grid. The Kalray has a double-mesh topology ensuring that the maximum distance between two nodes is 4 hops. As depicted in Figure 4.18a, there are two kinds of links: the short links and the long links. Short links connect odd and even clusters together if the processor. long links connect I/O cluster of the opposite side. There is one router per compute cluster and four routers per I/O cluster.

By rearrangement of the nodes of this network, a regular 2D-grid topology can be deduced. Figure 4.20, shows the North, South, East and West I/O clusters with a letter and a number. The compute clusters are named with numbers. We will show in Section 7.5.3 how the hole in the middle of the grid is handled when computing the routes.

Overview. Figure 4.21 shows the functioning of the NoC. A *TX engine* is composed of a packet shaper and a bandwidth limiter. The *packet shaper* forges packets with an header and a payload. The header is composed of the route and an identifier of the destination buffer. The route indicates the direction to take at each router. The *bandwidth limiter* blocks the packet until there are enough credits to send it.

A *RX engine* is a DMA writing the payload in the shared-memory. It is configured with the address of the destination buffer and an unique identifier.

Route. Unicast routes are encoded as a list of directions (North, South, East or West) in the packet's header. This list is *consumed* at each router to select the next link. The termination is encoded with a direction leading to the previous node. As a consequence, a route which goes back to the previous node cannot be programmed. One direction is encoded on two bits.

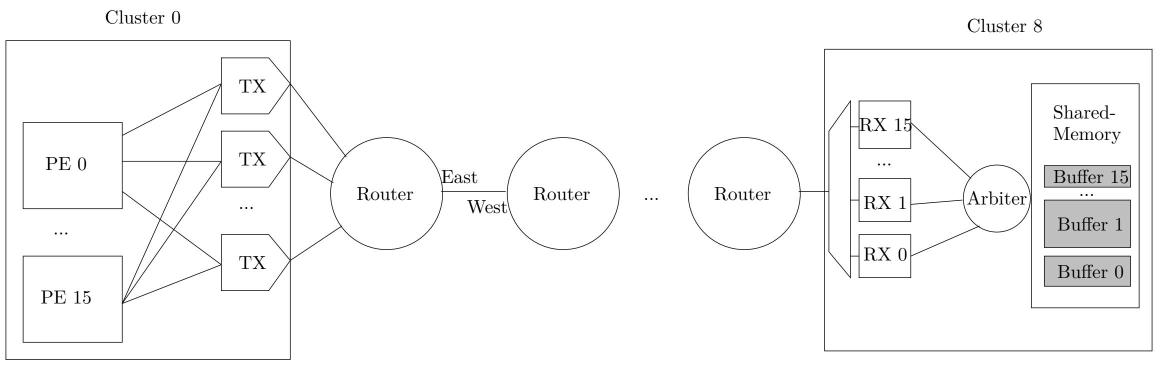


Figure 4.21: Schematic of the functioning of the MPPA2 NoC.

Similarly to unicast-routes, a multi-cast route is a path through routers. A each router, the packet can be delivered to the cluster. Consequently, three bits are required to encode each hop: two bits for the direction and one bit to enable the delivery.

Packet Shaper and Hardware Bandwidth Limiter. There are 8 packet-shapers in each cluster. To be able to send a packet, a core needs to configure a packet shaper with the header and the configuration of the bandwidth limiter. The header contains the route and an identification of the destination buffer.

A packet shaper is responsible for adding the header to the packet before sending the packet. The minimal size and maximal size of the packet can be configured. A buffer in the packet shaper stores the data before they are sent. If this buffer contains more data than authorized, the data is trunked in several packets. The packet-shaper does not send data if there is not enough data except if a flush is performed. In this case, padding is added to the packet to guarantee a minimal size.

The Hardware bandwidth limiter is based on the leaky bucket principle and has the same functioning as the one presented in Section 4.2.5.

Configuration of the reception. There are 128 RX engine in each cluster. They are configured with the address and size of the destination buffer in the shared-memory. The identifier of the RX engine is included in the packet header to select the right destination buffer. This RX engine can generate an event to the cores when a packet is received.

4.3.4 Summary of the Kalray MPPA2 Features

In this section, hardware feature of the MPPA2 relative to the timing predictability have been presented. They can be summarized as follows:

- Time-predictable cores with LRU policy caches
- Banked shared-memory with independent access from the cores
- Software-based cache coherency
- Efficient synchronization of the cores in the clusters
- Wormhole NoC with static routes, packet shaper
- Configurable packet size

All these form a basis for the implementation of time-critical software, nevertheless, they require special configurations that will be discussed in Chapter 7.

5 Method Overview: Dataflow Synchronous Languages to a Many-Core Processor

Previous chapters where dedicated to the prerequisite to understand the method. In this chapter, we explain the global method for the parallel implementation of a Dataflow Synchronous Program on a many-core processor.

The *Parallel Intermediate Representation* (PIR) describes the program as a set of computing entities communicating together. This PIR divides the work into two sub-problems: extraction of the parallelism from the input program discussed in Chapter 6 and the implementation of this PIR on the many-core discussed in Chapter 7.

5.1 Parallel Intermediate Representation Definition

The PIR has to be precise enough to make the implementation of the program possible but abstract enough to reason about it.

The straightforward PIR of a data-flow program takes advantage of node division. The behavior of the program is abstracted from the PIR. Hence, due to the data-flow property of Scade and Lustre, we consider the PIR as a set of nodes communicating together.

For data-flow Synchronous programs, the communication graph and the dependency graph are not equal due to the delayed communications (`pre` operator).

Nodes, functional code and tasks. A *node* is a sub-program in the main data-flow program. It is a candidate for parallelism, nevertheless, it does not necessarily lead to a parallel task. The *functional code* is the sequential compilation of the behavior of one node. The parallelism extraction, extracts the parallel tasks from the program. This step judges whether a node has to become a *task*. Finally, the tasks are mapped on the cores of the processor and scheduled. A task consists of the functional code of the node plus the communication code.

5.1.1 Task Graph and Communication Graph

The PIR of a data-flow Synchronous program is composed of a task graph and a communication graph. Figure 5.1, shows the PIR of a program on the top of the figure where all the top-level nodes are implemented into tasks. The dependency graph represents the partial order of the nodes and the communication graph represents both the instantaneous and the delayed communications.

The *dependency graph* is a directed acyclic graph (DAG). Its vertices are tasks and its edges are precedence constraints.

The *communication graph* can be cyclic if there is at least one delay for each feedback loop. The vertices are tasks and the edges are *communication channels*. A channel is a relation between two tasks. It carries the size of the transmission and the number of delays. If there is no delay, the channel is also a precedence relation.

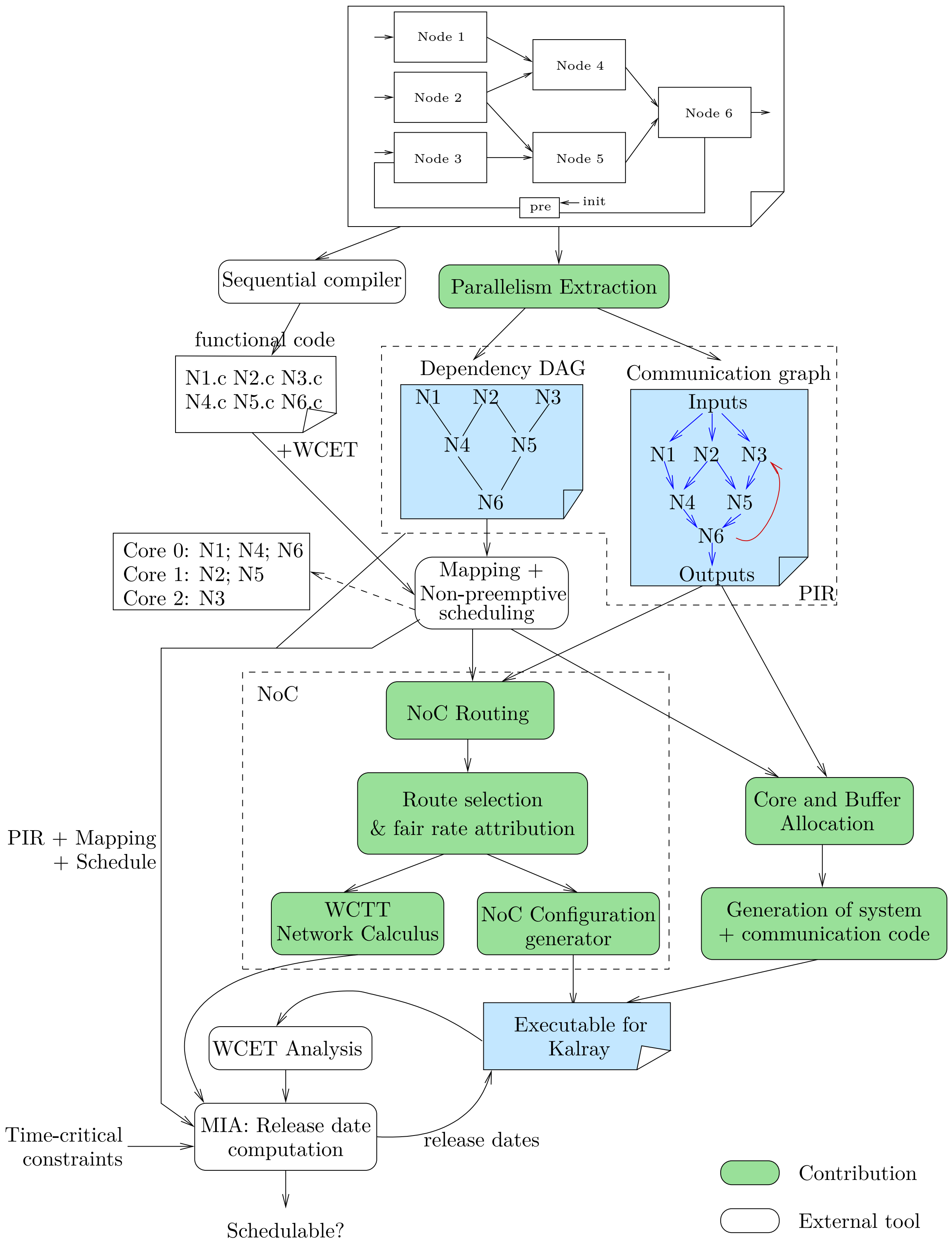


Figure 5.1: General framework for parallel implementation of Lustre/Scade with time-critical constraints on the Kalray MPPA2. White boxes are external tools while colored boxes are our contributions.

The dependency graph is used for scheduling the program and generating the synchronizations, while the communication graph is required to implement the communications. Consequently both graphs are required.

5.2 From Program to PIR: Task Extraction

The role of the **[Parallelism Extraction]** step from Figure 5.1 is to extract the parallel tasks from the program and thus to generate the PIR. This step is detailed in Chapter 6.

In our method, the tasks are extracted from the top-level node of the program. In this node, each sub-node is considered as candidate for parallelism and thus one task is created from each of them. A syntax analysis of the program extracts the tasks and the channels of communication between them.

This top-level method is similar to methods based on an architecture description languages such as Prelude or Giotto. In fact, we consider the top-level node as an ADL of the program.

For each extracted task, the functional code is compiled from the corresponding node using a standard Lustre/Scade compiler.

5.3 Implementation Choices to Take Advantage of the Banked Shared-Memory

In this section, we present the problems due to the shared-memory and the technical choices adopted in our method.

5.3.1 Inter-Task Communications: Remote Read vs. Remote Write

We consider platforms with a distributed memory or a banked memory with assignation of private banks to the core (similar to a scratchpad memory). We talk about *remote write* communications when data are copied by a core to the memory of another core. Conversely, we talk about *remote read* if a core reads the data in a remote memory.

The main difference between these two approaches is the location of the communication buffer. For the remote write, a task directly writes in the input buffer of other tasks, while for remote read a task reads in the output buffer of other tasks.

In our work, we choose the remote write communication for the shared-memory communications to be coherent with the NoC communications since the NoC of the Kalray MPPA2 offers direct remote memory access in write only.

5.3.2 Time-Triggered Execution Model

In a data-flow program, nodes are triggered by the availability of the input data. When this data-flow program is implemented by a set of tasks executed on a parallel architecture, this way of triggering tasks has to be reproduced. This can be done with an even-triggered mechanism where a synchronization is set between the source of the data and the destination node. This execution is also called “best effort”. The execution is said time-triggered if the input data are read at a specific date. To preserve the semantics of the program, we need to ensure that the inputs are arrived at this date.

The granularity of the interference analysis is at the task level. Two tasks can interfere as soon as they execute concurrently and they perform access in the same memory bank. Figure 5.2 shows Task A running on Core 0 and Task B running on core 1. A request from a task can be delayed by an access from the other task. In this case the actual duration of the task can exceed the WCET in isolation. Nevertheless, a Worst-Case Reaction Time (WCRT) taking into account the interference can be computed.

Figure 5.3 shows two executions of the same program. The tasks start as soon as possible. In the first execution, Task B does not interfere with Task C. In the second execution, Task A executes

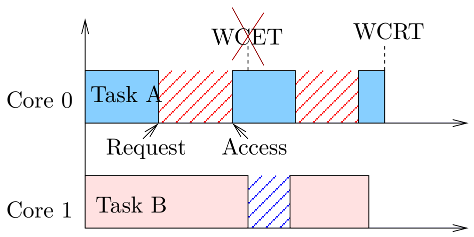


Figure 5.2: Impact of concurrent memory access on the WCET. The dashed parts represent the delays due to concurrent memory accesses.

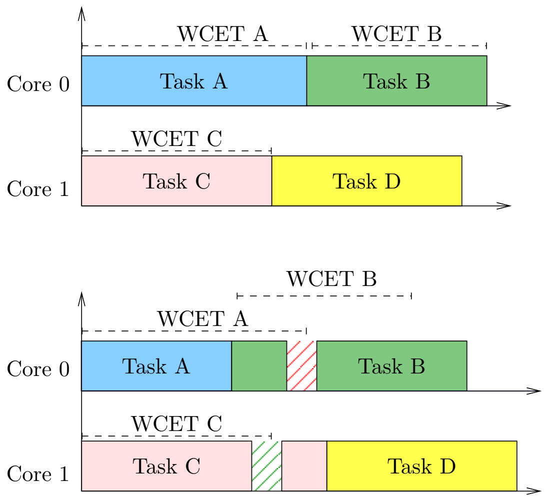


Figure 5.3: Two executions of the same program. In the second execution, a faster execution of Task A leads to a slower overall execution. The dashed parts are the delays due to concurrent memory accesses.

faster making Task B and Task C interfering. This leads to a slower execution of Task C and this leads to a slower overall execution.

A solution from Skalistis *et al* [119] is to force precedence between tasks using synchronization. For instance, in Figure 5.3, a precedence constraint would be added between Task B and Task C.

In our work, we rely on a time-triggered execution where a release date is fixed for each task, such that the tasks cannot start earlier if they are ready.

Rihani *et al.* [113] present a time-triggered execution model and a tool to compute the release dates of the tasks ensuring the availability of the inputs. As explained in Section 4.1.1.2, this time-triggered execution make WCRT bound less pessimistic. Consequently, we choose a time-triggered execution for the tasks.

5.4 Implementation of a PIR on a Many-Core

Once the PIR has been extracted, it can be implemented on the platform. The implementation must preserve the semantics of the input program and match the time-critical requirements.

5.4.1 Static Mapping-Scheduling

The [**Mapping+Scheduling**] step from Figure 5.1 in an external tool that finds a good placement of the tasks on the clusters and cores. Then it computes a schedule compatible with the data-flow.

One classical optimization criterion for the mapping-scheduling algorithm is the duration of the critical path of the program. Optimizing this duration requires the knowledge of the execution time of each task. Nevertheless, these execution times depend on the shared memory congestion which depends on the mapping. They also depend on the implementation of the communications.

This interdependency between the mapping-scheduling and the execution can be seen as a fixed-point problem. We choose to break the loop by using the WCET in isolation of each task as an approximation of the execution time. Consequently, the mapping-scheduling tool takes two inputs: the dependency graph and the WCET in isolation of the tasks.

The computed schedule is static and non-preemptive. This avoids cache-related preemption delays and unpredictable behaviors in order to ease the WCRT computation. Any mapping and scheduling technique can be used as long as it takes into account the task dependencies. The schedule can be checked using the PIR dependency graph. Gorcitz *et al.* [62] shown that mapping and scheduling of multi-periodic applications using integer linear programming does not scale when the number of tasks and the number of constraint are high. Thus this step requires heuristics. Our work relies on the tool from Nguyen *et al.* [100] which considers the effect of private caches in the execution time.

Figure 5.1 shows an example of mapping of one input program on three cores of the same cluster. Each core executes a sequence of tasks: core 0 executes N1; N4; N6, core 1 executes N2 and N5 and core 2 executes N3.

One memory bank is associated to each core. The code, data and communication buffers belonging to this core are mapped on this bank. The **[Code and Buffer Allocation]** step from Figure 5.1 statically allocates the code, data and buffer of each task to the corresponding core. Consequently, each core accesses exclusively its own bank, except when it communicates.

5.4.2 Release Dates and Final Executable

The [**WCET Analysis**] step from Figure 5.1 computes the execution time of the tasks in isolation. The WCET analysis tools OTAWA [8] and AiT [50] support the Kalray MPPA2 processor.

The [**MIA: Release date computation**] step computes a release date for each task taking into account both precedence constraint and memory interference. We used the MIA tool [113] but there is a similar method from Skalistic *et al.* [120]. The principle of the MIA tool is to compute iteratively the durations of the tasks and the release dates of the tasks. The durations are initialized with the WCET in isolation of the tasks. At each iteration, the durations are increased with the memory interferences. We describe one iteration of the algorithm:

1. The *release date* of each task is computed such that it is the smallest date where all the dependencies are satisfied. The *end date* is the *release date* + the current *duration* of the task.

2. For each task, the set of interfering tasks is computed. Two tasks interfere if their executions overlap, *i.e.*, there exists a date t which is between the *release date* and the *end date* of both tasks. Furthermore, interference is only considered if two tasks access the same shared device.

3. The *duration* of each task is updated with its WCET in isolation plus the delay due to the interfering accesses to shared-devices The algorithm is executed until convergence.

When the algorithm has converged, the WCRT of a task equals its *duration* if the program is time-triggered using the *release dates* computed by this algorithm. A deadline can be set for each task to check the time-critical constraints. If a deadline cannot be satisfied, the program is rejected since it is not schedulable.

The **[Generation of system + communication code]** step generates the system code powering up the cores, the implementation code of the static schedule and the communication code. In time-triggered implementation, this step requires the release date of the tasks. Nevertheless, these release dates depend on the WCET of the tasks which can be known only by analyzing the final binary since the compilation can change the layout of the binary.

There is interdependency between steps [**Generation of system + communication code**] and [**MIA: Release date computation**] since MIA computes the release dates with the WCET analysed on the final binary. Consequently, as represented in Figure 5.1, the release dates are integrated by modification of the final binary.

5.4.3 NoC: Routes and WCTT

When tasks from different clusters communicate, the communication channel is implemented using the NoC. We call flow, one communication through the NoC.

We choose to configure the NoC with a static route and a constant bandwidth limitation for each flow. Consequently, the NoC configuration requires two parameters for each flow: a route and a bandwidth to configure the bandwidth limiter. This configuration eases the Worst-Case Traversal Time (WCTT) computation which is an upper bound of the transmission latency through the NoC. The WCTT is required by the [**MIA: Release date computation**] step.

The [**NoC Routing**] step computes the set all the possible routes for a flow. For instance, if the chosen algorithm is XY, there is only one possible route for the flow, conversely, if the algorithm is HOE, there can be several routes.

The [**Route selection & fair rate attribution**] step selects one route per flow while optimizing a global criterion. The criterion is the max-min fairness which maximizes the rate of flows of minimum rate. This step is explained in Section 7.5.

Then, the [**NoC Configuration generator**] step generates the the packets' header with the selected route and the configuration of the hardware bandwidth limiter.

For each flows, their bandwidth and their route, the [**WCTT Network Calculus**] computes the WCTT. This step is explained in Section 7.6.

5.5 The CAPACITES Project

CAPACITES is a French project with academic and industrial partners. The topic of this project is the design of critical embedded software for a many-core architecture.

The framework represented in Figure 5.1 has been designed as part of the CAPACITES project. The WCET analysis, the MIA tool, and the scheduling tool from Nguyen *et al.* [100] are from this project.

5.6 Conclusion

In this section, we have briefly presented each step of a framework to compile data-flow Synchronous programs on a many-core processor under time-critical constraints. The framework preserves the semantics of the input program.

The hardware is configured to ease the WCET computation: the shared-memory is configured in banked mode and the buffer, code and data are allocated in cores' private bank in order to minimize interferences. The NoC is used in a predictable way with bandwidth limiter and static routes. The worst-case traversal time of communications through the NoC is bounded thanks to the deterministic network calculus framework. The execution is time-triggered to ease the WCET computation.

6 Parallelization of Synchronous Programs

This chapter details the **[Parallelism Extraction]** step which creates the parallel intermediate representation (PIR) from a hierarchical dataflow Synchronous program. An execution period can be specified for each node in the dataflow. We present several solutions where the nodes are considered as the minimal unit for parallelism.

6.1 Main Criteria to Select a Parallelization Method

In this section, we present the main characteristics of the parallelization methods. These characteristics are properties of the code generated using these methods.

6.1.1 Centralized Execution

The parallel methods either provide a centralized execution or a decentralized execution. The *centralized execution* is symmetric since the initiator of a task is also waiting for its completion. In decentralized execution, the task responsible for task creation and the task waiting for the completion can be different. Most of the time, the purpose of waiting for the completion of a task is to make sure the output data are available.

For instance, considering an initiator task `I` triggering a task `A`. In centralized execution, the initiator `I` is also responsible for waiting for completion of `A` and the information of termination. In *decentralized execution*, an initiator task `I` can create a task `A` while a task `B` can wait for the completion of `A`.

6.1.2 Hierarchical Parallelism Extraction

A hierarchical data-flow is composed of nodes which can be either leaf nodes or nodes containing other nodes. Leaf nodes can be basic operators such as `+, -, *, /` or external function call.

The parallelism extraction is the selection of some nodes of the data-flow as candidate for a parallel execution. The parallelism extraction is *hierarchical* if the candidate nodes are selected at any level of the hierarchy.

For instance, in Figure 6.1, the level of nodes `D` and `E` is different from the level of `I` and `J` since `D` and `E` are direct children of `B`, whereas `I` and `J` are direct children of `F`. Colored nodes are selected for parallelism. Each selected nodes will be considered as one sequential task. This is visible for node `H` which is selected entirely which means that it will be compiled into a single sequential task including its sub-nodes `K` and `L`.

6.1.3 Code Traceability

Traceability between the code and the implementation is important for debugging and certification. It ensures that any line of the final compiled code can be linked to the original Scade or Lustre code.

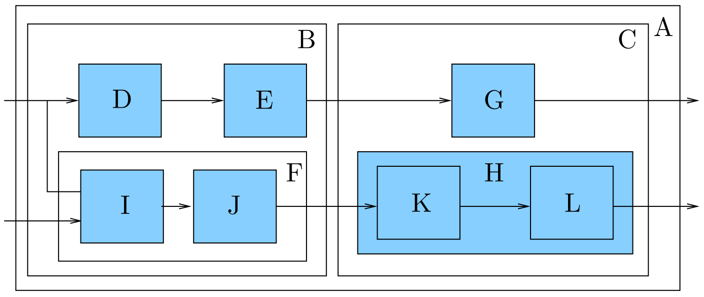


Figure 6.1: Hierarchical data-flow.

One solution to achieve this goal is to select the node as the compilation unit. Consequently, the code corresponding to each node is visible in the final binary.

6.1.4 Special Case for Clocks and Delayed Communications

Clocks in Lustre are the expression of the conditional execution of nodes. They are usually compiled into `if` statements by the Lustre compiler. In the case of periodic clocks, the execution of the nodes can be statically scheduled, replacing these dynamic conditions by a regular pattern of execution (hyper-period expansion).

The delayed communications (`pre` operator) can be handled in a centralized way or in decentralized way. In centralized, the delayed communication can be implemented with a double buffer managed by the initiator of the computations. In decentralized, it can be replaced with a special node which, at each execution, provides the previous input as output and stores the new input value.

6.2 Parallelization of Lustre and Scade

In this section, we detail some approaches for the parallelization of Lustre or Scade.

6.2.1 Fork-Join for Dataflow Synchronous Languages

Fork-Join is a centralized construct for parallel execution. The fork operation (denoted with $\uparrow$) is the creation of a new parallel task. It takes a function as parameter. The join operation (denoted with $\downarrow$) waits for a parallel task to finish.

The inputs are transmitted when the task is forked and the outputs are read when the task is joined. Scheduling fork, join and the local computations of the nodes is a multi-criteria optimization problem that depends on the execution time of the nodes.

We apply the fork-join parallelization method on the example of Figure 6.2. We recall that the execution is statically scheduled. We extract as much parallelism as possible.

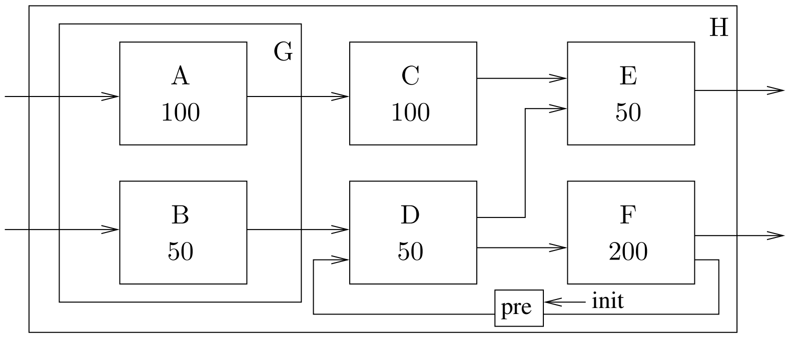


Figure 6.2: Example of hierarchical data-flow Synchronous program.

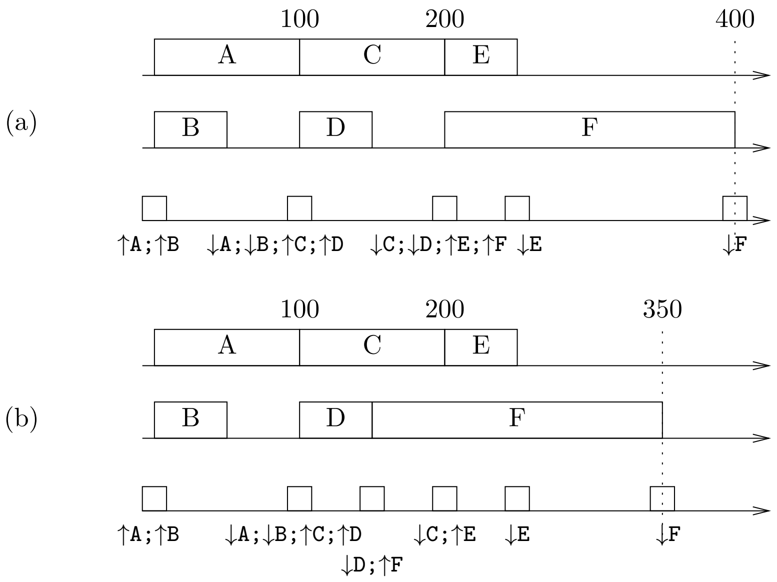


Figure 6.3: Examples of fork-join

The schedule is static, meaning that the order of the fork is predefined. Many schedules are possible. Two of them are presented:

Implementation (a):

```
H{ G{ ↑A; ↑B; ↓A; ↓B }; ↑C; ↑D; ↓C; ↓D; ↑E; ↑F; ↓E; ↓F}
```

Implementation (b):

```
H{ G{ ↑A; ↑B; ↓A; ↓B }; ↑C; ↑D; ↓D; ↑F; ↓C; ↑E; ↓E; ↓F}
```

Figure 6.3 shows two fork-join implementations of the same parallel program. Implementation (b) takes advantage of the duration of the tasks to minimize the execution duration. The sequence `↑A; ↑B; ↓A; ↓B` is used in both implementations since it corresponds to the execution of the same node `G`. Preserving the `G` frontier enables modular compilation of the program and the traceability of the code.

Both implementations (a) and (b) use one core dedicated to the execution of the fork and join operations. Nevertheless, this is not necessary and an equivalent of Implementation (a) exists where a core executes both fork/join operations and computations, *e.g.*, `A()`, `B()`, `C()`.

```
H{ G{ ↑B; A(); ↓B }; ↑D; C(); ↓D; ↑F; E(); ↓F}
```

We recall that the fork operation can only be performed on a task if the input data are present. In other word, since the execution is centralized, the join operation must be performed on all the dependent tasks before forking a new task. An equivalent of Implementation (b) on two cores giving a duration of 350 is not possible under these conditions.

6.2.2 Decoupling of the Nodes Using Future in Lustre

Future [54] is a standard language construct to express parallelism. It exists in C++ and Java. The beginning of the computation is decoupled from the usage of the result. In other words, it allows launching asynchronous computations and only the access to the result of these computations is blocking. In the following code, the function `f` is computed in parallel with `g`:

```
Future x = async f(i1);
int o1 = g(i2);
int o2 = h(!x);
```

The statement `!x` blocks until the computation of `f` has finished, then `h` can be computed.

Future in Lustre. Cohen *et al* [36] present an implementation of Future for Lustre. Combined with clocks, this implementation allows launching a computation and getting the result at different logical instants. Here is the introductory example from the paper [36]:

```
node a_slow_fast () = (y : float)
  var big : bool ; yf , v : float ; ys : future float;
let
  big = period <<3>>();
  ys = (async 0.0) fby (async slow (y when big));
  yf = fast ( v whenot big );
  y = merge big (!ys) (yf);
  v = 0.0 fby y ;
tel
```

The built-in node `period«n»()` produces a Boolean flow true every `n` ticks. The expression `async` *`exp`* launches the asynchronous execution of *`exp`*. The expression `merge` *`clk exp1 exp2`* has the value *`exp1`* when *`clk`* is true and the value *`exp2`* otherwise.

If the `async` and the `!` keywords, are removed, the program keeps the same semantics. The behavior of the program is the following:

At the first instant, `big` is true and then `slow` is activated with the initial value 0. The output `y` takes the value 0.0. The expression `async 0.0` is a Future with the value 0.0, it is required since `ys` if a variable of type `future`. At the second and third instants, `big` is false, `fast` is activated and output `y` is the output of `fast`. Then at the fourth instant, `slow` is activated and the output of the program is the output of `slow`.

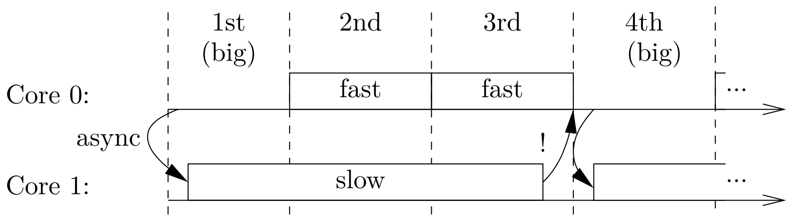


Figure 6.4: Futures in Lustre for Parallel Computation.

To decouple the execution of `slow` in parallel with the two `fast` execution, the `async` and `!` keywords are added.

6.2.3 Parallel Subset in Scade

Pagano *et al.* [101] introduce a modification of the Scade KCG compiler to support expression of parallelism in the Scade model. A *parallel subset* is a group of nodes selected for a parallel execution. The specification of the parallel subsets is done with special pragma in the code or with an external file.

Parallel subsets are a particular case of fork-join where the fork is done on a group of independent nodes. The implementation of the communications and the scheduling is not generated since it depends on the target platform. Consequently, the program must be completed with the system code to schedule and execute tasks on the target platform, and the communication code to transfer the buffers.

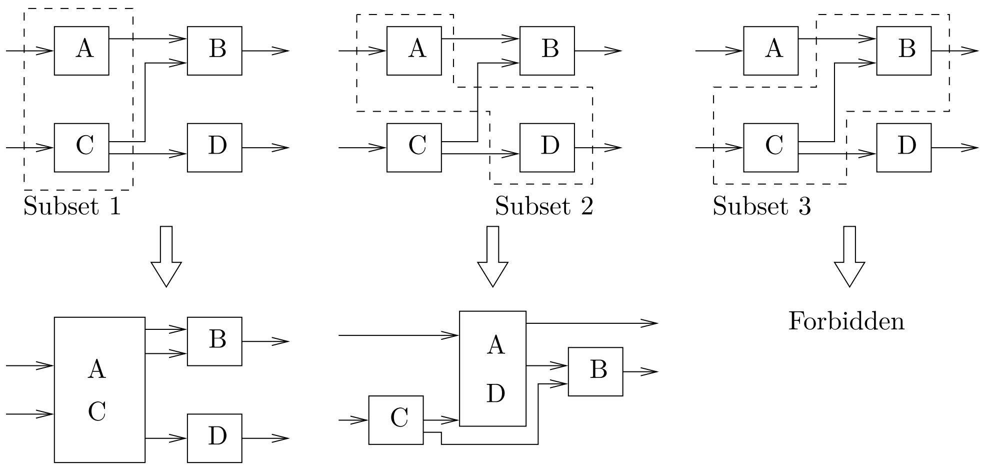


Figure 6.5: The Scade Parallel Subsets.

The compiler checks if the program is parallelizable and rejects the programs with invalid parallel subset. The compiler checks that there are no dependencies between the nodes of the subset. Figure 6.5 shows some examples of valid and invalid subsets.

When the parent node is compiled, the ↑ and ↓ statements have to be scheduled with the local computation of the node. To do this, the Scade compiler considers the subset as one virtual node as depicted in Figure 6.5. This node is scheduled with the other nodes of the parent node. Then, the virtual node is replaced with a sequence of ↑, one for each node of the subset, followed by a sequence of ↓, one for each node of the subset.

This method does not allow executing in parallel nodes from different parents. Figure 6.8 a program which is not parallelizable with this method since nodes cannot be divided. Nodes `A` and `B` are in the same node `D` and `C` has a different parent. This limitation exists to preserve traceability and enable modular compilation.

For the program of Figure 6.2, the only possible implementation maximizing the parallelism is the following:

```
H{ G{ ↑A; ↑B; ↓A; ↓B; }; ↑C; ↑D; ↓C; ↓D; ↑E; ↑F; ↓E; ↓F}
```

This schedule is represented in Figure 6.3(a).

Similarly to classical fork-join, the core responsible for forking can execute the following code:

```
H{ G{ ↑A; B(); ↓A; }; ↑C; D(); ↓C; ↑E; F(); ↓E}
```

We presented the Parallel subsets as a restricted fork-join. Fork-join is centralized since all communications between the tasks are managed by the forking core. This drawback is presented in

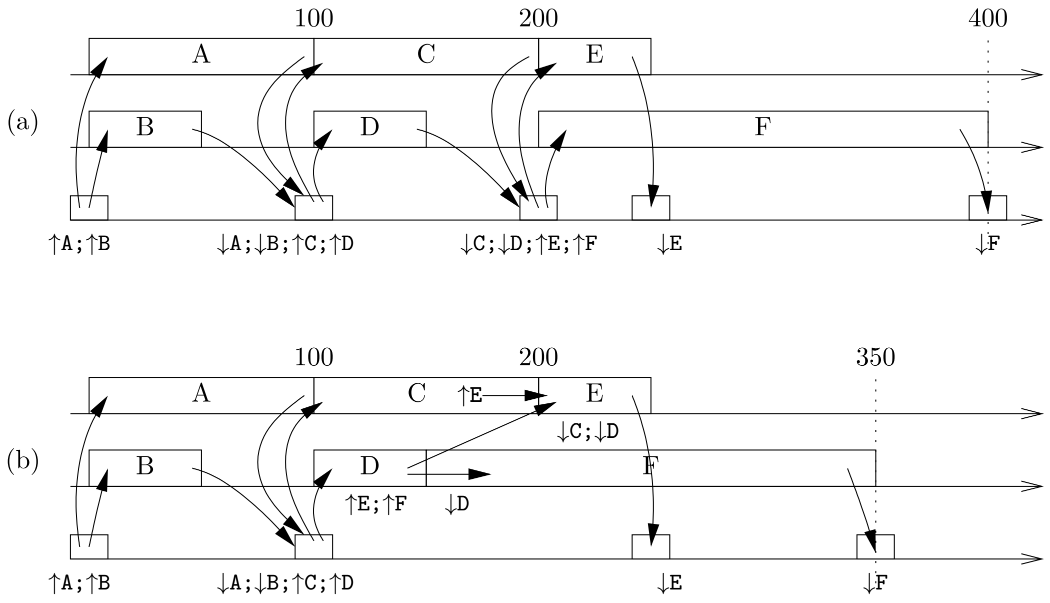


Figure 6.6: Comparison of the fork-join execution (a) and parallel subsets execution (b) with decentralization enabled.

Figure 6.6(a) where the forking core centralizes communications that could be done directly, *e.g.*, the communication between node `C` and node `E`.

The parallel subsets method offers a special optimization enabling data-flow communications between the different sections. The principle is that the tasks of two parallel subsets can communicate directly if they are in the same parent node. This schedule is shown in Figure 6.6(b) where nodes `C`, `D`, `E`, `F` communicate directly. Communication between nodes `A`, `B`, `C` and `D` is not direct to enable modular compilation. This optimization does not enable direct communication for the inputs of node `D` since the `pre` operator is a local computation of the forking node (see Figure 6.2).

In this example, the ↑ and ↓ operators have a slightly different meaning. Instead of representing fork and join operation carrying both control and communications, they only correspond to a communication. Then, the implementation of the ↑ operator has to ensure that the tasks do not start until all the inputs are present. The implementation of ↑ and ↓ is not generated by the KCG compiler and will be explained in Chapter 7.

6.2.4 Top-Level Node as an Architecture Description Language

In the Top-Level method, only one level of the program is considered. All the sub-nodes of the top-level node are the candidates for parallelism. This is the common method used by ADLs such as Prelude and Giotto.

The nodes are mapped on the cores of the processor and they are launched when their input data are present. If there are more than one node per core, a schedule has to be computed. The `pre` operator is handled in a data-flow way.

Figure 6.7 represents one schedule of the program of Figure 6.2 on two cores. Node `G` is candidate for parallelism and consequently compiled as a sequential node.

The main problem of this method is that the node hierarchy may forbid some parallelism since it creates useless synchronizations. For instance, in Figure 6.8, node `A` and `B` are the only computations of node `D` and both `A` and `B` are required before the end of node `D`. Due to the dependency, node `A` must be computed before `B`. Furthermore, node `C` depends on node `D`. Finally, `A`, `B` and `C` are in

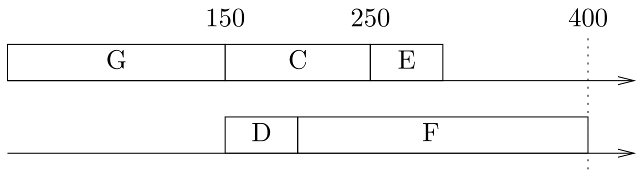


Figure 6.7: Schedule for the program of Figure 6.2 using the Top-Level Node method.

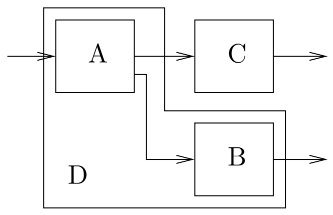


Figure 6.8: Example of Non-Parallelizable Data-Flow Node.

sequence thus this program is not parallelizable with this method. Furthermore, node `G` of Figure 6.2 is compiled as a sequential node even though it contains independent nodes.

6.2.5 Data-Flow Flattening

Data-flow flattening [125] (or graph flattening) is a method to convert a hierarchical data-flow into a flat data-flow by replacing each node with its internal data-flow. Figure 6.9 shows an application of graph flattening method to the example of Figure 6.2.

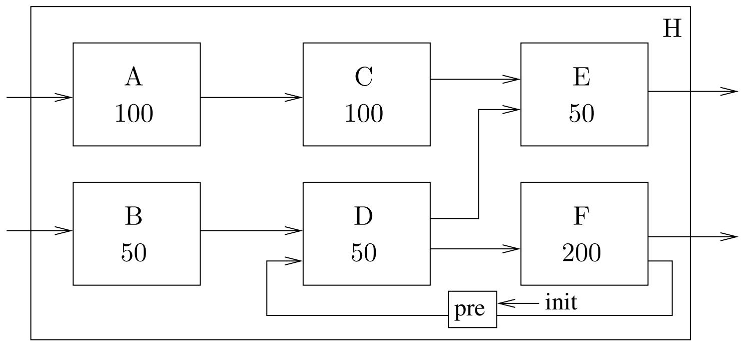


Figure 6.9: Example of graph flattening for the program of Figure 6.2.

Tripakis *et al.* applied this method to Static Data-Flows (SDF) where the rates of the nodes are written on the arrow making the destination node executed several times. This method causes graph explosion for SDF graphs if a node with a high rate is replaced with its contents since the contents are duplicated according to the rate. In Data-flow Synchronous programs, this rate is always 1 consequently the do not suffer from graph explosion.

This method makes more parallelism appear by removing the synchronization due to the structure of the program. The drawback is the reduction of the traceability since some nodes are lost. This flattening process yields some nodes with a low execution time. Each new task introduces new communications. For small tasks, the cost of communication can be higher than the time saved by parallelism.

To deal with this problem of minimal node execution time, the data-flow flattening can be made partial. If the execution time (or an approximation) of each node is known, some nodes can be kept unchanged if their flattening would produce too small nodes. Then, the resulting data-flow can be parallelized with the top-level node method.

6.2.6 Methods Comparison

We need to choose a parallelization method which authorizes enough parallelism while matching some constraints. The solution has to ensure the traceability of the generated code and it must authorize an efficient static scheduling.

Characteristics	Future	Fork-Join	Parallel Subset	Top-Level	Flattening
Hierachical	yes	yes	yes	no	yes
Centralized	yes	yes	yes/hybrid	no	no
Traceability	yes	yes	yes	yes	partial
Clocks	yes	no	no	no	no
`pre` support	yes	no	no	yes	yes

Table 6.1: Comparison of the Parallelization Methods for Data-flow Synchronous Programs.

Table 6.1 summarizes the main characteristics of the methods. Future and Fork-Join are centralized, hence they are not adapted for many-core architectures. In the case of Future, the compiler needs to know the task durations to generate efficient code. Furthermore, these methods require a modification of the source program to handle parallel execution. Future is the only method managing the Lustre clocks to authorize decoupling. All other methods can handle multi-period programs by hyper-period expansion. A static schedule of Fork-Join sometimes requires an extra core managing the fork and join operations.

Parallel Subset is a centralized method. Nevertheless it authorizes decentralized communications between some tasks. Some tasks start as soon as they are ready; some others are triggered with a programmed fork operation. This method is hierarchical and the source program does not require any modification. Traceability is ensured. Nevertheless, the `pre` operator is handled in a centralized way as there is no special case for data-flow delayed communication.

The top-level node method is fully data-flow. The data-flow flattening can be considered as a pre-transformation of the data-flow that may lost some traceability. Nevertheless, the top-level node method offers traceability and a special case for the `pre` operator.

We selected the top-level node and the parallel subset methods since they offer traceability and the modification of the source program is not required.

6.3 Contribution: Extraction of PIR from a Dataflow Synchronous Program

In the previous section, several parallelization methods have been compared and we argued for the selection of the Top-Level Node method and the Parallel Subsets. In this section, we present the extraction of the PIR from a program using these two methods. In practice, we applied the top-level method to the Lustre language, using the academic Lustre compiler to compile sequential parts of the program, and the Parallel Subsets method to Scade, using the KCG Multi-Core compiler.

6.3.1 Parallelization Method 1: Lustre Top-Level

The PIR corresponding to a single synchronous data-flow node is quite straightforward to generate. If this node contains local computations, each of them has to be considered as a node.

The dependency DAG is obtained thanks to a syntax analysis of the data-flow. The dependencies due to delayed communication are ignored.

The communication graph contains more information. Both the direct and delayed communications are extracted and represented with channels. The channel contains the source, destination, transmitted variables and the data type. If the communication is delayed, the number of delays and the constant initialization value are recorded. Each sub-node is compiled separately.

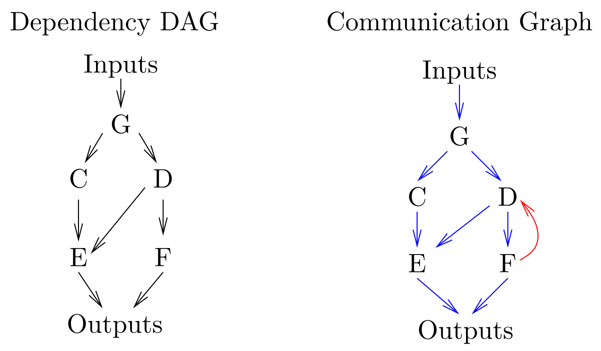


Figure 6.10: PIR corresponding to the example of Figure 6.2 for the top-level node method.

Figure 6.10 represents the PIR corresponding to the example of Figure 6.2. The dependency graph can be used to generate or validate a schedule. Channels are represented with arrows on the communication graph. The channel from `F` to `D` has one delay and an initialization value `init`.

The PIR is represented as a structured Yaml file. This file is used to convey the information about the tasks at each step of the toolchain. It describes the tasks and the channels. The mapping, schedule and the timing information about the tasks will be appended later.

ExtractDependencies is a tool part of our toolchain which extracts the communications from a Lustre node. A syntax analysis of the program is performed. It creates the channels and interprets the `pre` operator as a channel with a delay. Then, each node is compiled separately as a fully-fledged program. The resulting `step` function is the functional code of one task.

Our contributions are the extraction of the PIR from the program and the implementation of the schedule and the communications channels. The implementation of the schedule and the channel for the Kalray MPPA2 are detailed in Chapter 7.

6.3.2 Parallelization Method 2: Parallel Subsets from KCG Multi-Core

This section presents our second implementation of parallelization, based on the KCG Multi-Core compiler. The KCG Multi-Core compiler provides sufficient information to build the PIR from the program. It lists channels and dependencies but does not provide the schedule and the communication implementation. A contribution of our work is this implementation. It will be detailed in Chapter 7.

6.3.2.1 Simple Data-Flow Program

The Parallel Subset method is centralized since it is based on fork-join, *i.e.*, a task is responsible for forking and joining the parallel nodes. This task alternates between sequential code with fork and join statements and idle parts. Consequently, this task can be split into several sequential tasks.

The execution is partially centralized, hence this creates loop in the dependency graph. Consequently, in the PIR corresponding to the example of Figure 6.2 we split H into three sequential parts that correspond to fork and join instructions. Figure 6.11 shows the resulting PIR. The dependency graph is alternating between sequential parts (H1, H2 and H3) and parallel execution. The schedule of the parallel nodes is straightforward since the DAG gives the partial order. A single function for node `H` is generated. It performs fork, joins and some computations. Consequently, it is mapped on a dedicated core.

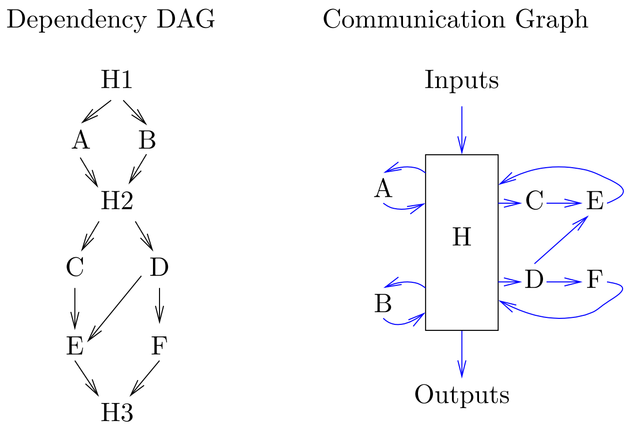


Figure 6.11: PIR corresponding to the example of Figure 6.2 for the parallel subsets method.

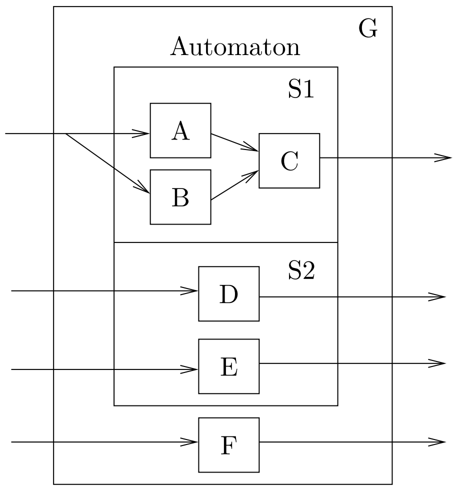


Figure 6.12: Example of Scade program with conditional execution.

The communication graph describes the communications between the main node and the parallel nodes and between the parallel nodes. This graph has no delayed communications since the `pre` operator is considered by KCG as a local computation of `H` and is replaced with an instantaneous communication.

6.3.2.2 Special Case for Automata

Scade automata have been presented in Section 3.1.2.2.They offer a way to conditionally execute nodes. Fully static schedule of automata is not possible in the general case since the set of executed nodes depends on some dynamic computations. Nevertheless, fully dynamic scheduling is not suitable for time-critical systems. *Quasi-static* scheduling is a class of scheduling methods which computes schedules that are mostly static except when it is absolutely necessary [66]. Our approach for the quasi-static compilation of automata is to compute the static schedule corresponding to each state of the automata. Then, the WCET analysis can be done by taking into account the longest state.

For a task belonging to an automaton state, the PIR provides the name of the automata and the name of the state.

Figure 6.12 shows a Scade program composed of an automaton with two states. State S1 contains nodes `A`, `B` and `C` and state S2 contains nodes `D` and `E`. The current state of the automaton is computed by the main node. Node `F` is outside the automaton thus always executed. A static schedule for this program on two cores could be:

For the state S1:

Core 1: `A; C`
Core 2: `B; F`

And for the state S2:

Core 1: `D`
Core 2: `E; F`

In order to express alternative schedules, we introduce the notation `Branch(S1, S2)` to specify that exactly one of the schedules `S1` and `S2` is executed. This notation can only refer to exclusive nodes, hence, only nodes of the same automaton can be specified. The corresponding schedule is:

Core 1: `Branch({A; C}, D)`
Core 2: `Branch(B, E); F`

To be able to express every schedule, we need an extra operation called ε. We consider the following schedules:

Schedule for state S1:

Core 1: `A;B;C`
Core 2: `F`

And for state S2:

Core 1: `D`
Core 2: `E; F`

The schedule of Core 1 relies on the `Branch` statement only. The schedule of Core 2 is the conditional execution of `E` followed by the unconditional execution of `F`. The node `F` can only start if `E` has been executed or if the current state is S1. This is expressed with the `Branch` and the ε operator:

```
Core 1:  Branch({A;B;C},D)
  Core 2:  Branch(ε,E);F
```

The ε is used whenever expressing the absence of task is required. Implementation of the schedules will be detailed in Chapter 7.

6.4 Conclusion

In this chapter, we compared several parallelization methods for Lustre and Scade. We discussed the choice of two of them offering a decentralized execution and the traceability of the generated code. The first is a pure data-flow solution; the second is based on fork-join. We detailed the generation of the PIR for both of them and handled the case of conditional execution. Implementation of the PIR is detailed in Chapter 7.

7 Time-Critial Implementation on the Kalray MPPA2

In this chapter, we present the implementation of the PIR on a many-core in order to preserve the semantics of the source program and to guarantee the real-time properties. The experiments have been done on the Kalray MPPA2, consequently, for each part, we state whether the implementation applies to a generic platform or whether it is specific to this platform.

This chapter is the result of a significant analysis of the hardware architecture in order to find the sources of interferences. We provide a configuration and a usage mode of the hardware minimizing the interferences.

In Sections 7.2, 7.3 and 7.4 we detail the [**Generation of system + communication code**] step. In Section 7.5, we describe the [**NoC Routing**] and [**Route selection & fair rate attribution**] steps. Finally, in Section 7.6, details the [**WCTT Network Calculus**] step and the final NoC configuration step [**NoC Configuration generator**].

7.1 Related Work on Multi-Core Execution Models

The software can be tailored to take advantage of hardware features of predictable multi-core and many-core platforms. There are two ways to limit interferences: spatial and temporal isolation. This isolation may be implemented through hardware or software mechanisms. Thus, interference can be completely eliminated, for instance, by decoupling the memory accesses and computation [103] and scheduling the tasks in a way that forbids concurrent memory accesses [96, 47]. Work from Becker *et. al* [10] eliminates any interference by reserving a memory bank for communications and shared variables. A static time-triggered schedule eliminates concurrent access to this bank. In our work, we do not aim at fully eliminating the interference. Instead, we use the hardware and software properties to minimize and bound interference.

Another feature in execution model concerns the way the tasks are triggered. Time-triggering is a way to improve predictability. Kopetz [83] has defined the Time-Triggered Architecture (TTA) that allows execution of real-time applications on a heterogeneous distributed system. Clocks of the system are synchronized and a Time Division Multiple Access (TDMA) bus ensures time-triggered communications without any interference. Caspi *et al.* [28] automated the compilation of Lustre programs for this architecture. Another approach of time-triggered implementation of synchronous data-flow for TTA taking execution modes into account is given in [27]. TTA and TTEthernet are used in the industry for connecting devices. Carle *et al.* [26] introduce a scheduling algorithm for data-flow synchronous program. It avoids interference and it is based on a time reservation table of the computation and communication resources of the many-core: cluster, link and DMA. This algorithm provides a safe WCET bound of the parallel application. NoCs such as Argo [79] offer TDMA communications. In our work, we use time-triggering to address unpredictable shared-memory.

7.2 Static Schedule Implementation

In this section, we describe the implementation of the static schedule and the quasi-static schedule if the execution is conditioned by automata. This implementation is not specific to the MPPA2 and can be applied to any multi- or many-core architecture.

7.2.1 Static Schedule

The mapping of the nodes on this core and the execution order are given by the **[Mapping-Scheduling]** step. Core 1 executes nodes 1, 4, 6 in sequence, core 2 executes nodes 2 and 5 in sequence and core 3 executes node 3. For each core, we generate the code implementing the schedule.

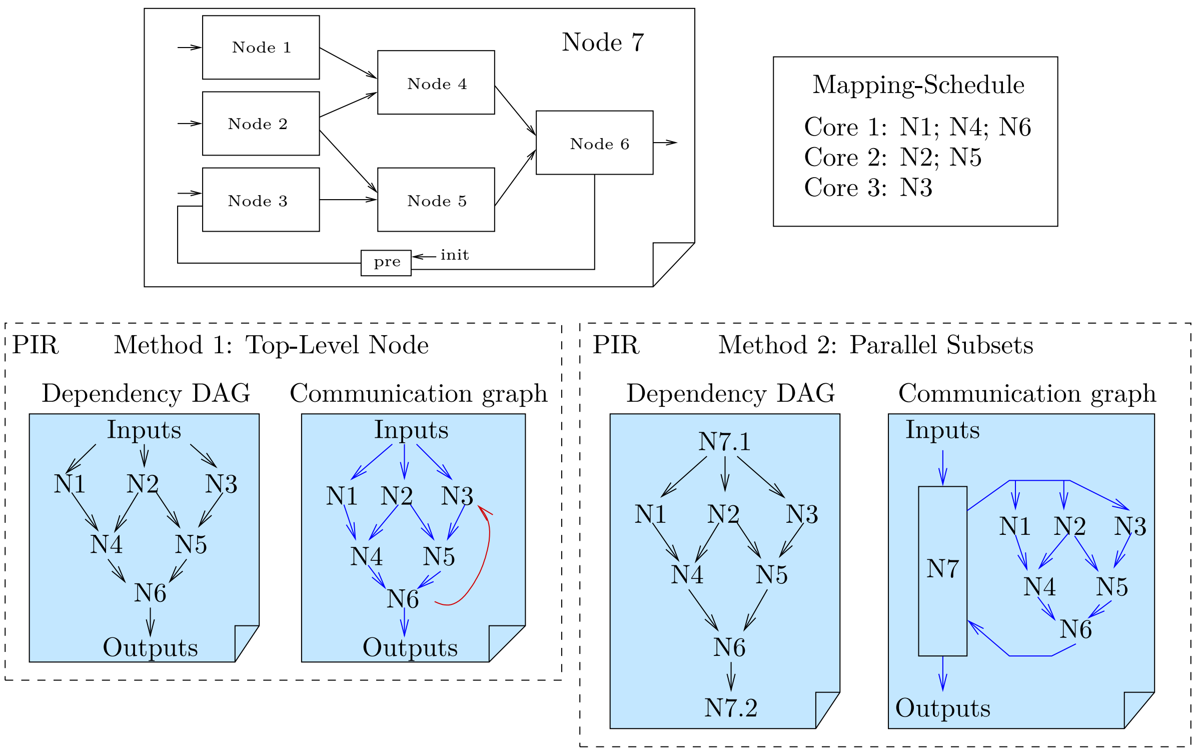


Figure 7.1: Mapping-scheduling and PIR for both parallelization methods.

We show the schedule corresponding to the example of Figure 7.1.

```
// Core 1              // Core 2              // Core 3
while(true) {          while(true) {          while(true) {
  task_N1();             task_N2();             task_N3();
  task_N4();             task_N5();           }
  task_N6();           }
}
```

This code varies slightly depending whether the program is implemented with the parallelization methods 1 or 2. For the parallel subsets method from KCG multi-core, Core 0 executes Node 7. For the Top-Level node parallelization method, the `pre` operator is managed by an extra task which is added to the schedule. Details on delayed communications are given in Section 7.4.2.

The tasks are functions responsible for reading the input of the node, executing the functional code of the node and writing the outputs. We show an example for N2:

```
void task_N2() {
  wait_data_N1();
  N1_step(&ctx_N1);
  send_data_N1();
}
```

For Top-Level node method, we generate the `task_*` functions, for parallel subsets method, the KCG Scade compiler generates them. In both cases, we create the `wait_data_*` and `send_data_*` functions from the communication graph of the PIR since they depend on the mapping and the target architecture.

We choose a remote write implementation, as discussed in Section 5.3.1. Consequently, when a task finishes, it sends outputs to all the requiring tasks. For instance, in the program of Figure 7.1, when N2 completes, it sends its outputs to N4 and N5.

7.2.2 Quasi-Static Schedule: Special Case of Automata

In Chapter 6, we introduced the `Branch` construct allowing expressing quasi-static schedules to handle automata. In this section, the implementation of this construct is presented.

A straightforward implementation of schedule of automata requires to broadcast the current state of the automata to all cores. The nodes corresponding to the current automata are then executed. Nevertheless, the KCG multi-core does not provide a direct way to know the current state at runtime.

The Scade language ensures that at each logical instant, the automaton is in exactly one state. Furthermore, instances of the same node located in different states are distinguishable and are considered as different tasks. For these reasons, if in a logical instant, we can identify at least one task activated by the automaton, we know the current state of this automaton. Since the different cases of a `Branch` statement must refer different states of the same automaton, we can deduce the nodes to execute from the available inputs.

In Chapter 6, we introduced the ε statement which allows expressing the absence of task. For instance, `Branch(B,`ε`)` means that task `B` may be executed if the current state of the automaton contains this task `B`, else the statement has no effect.

Implementing ε requires expressing the condition: *the current state is not X*. This Boolean condition can be expressed using extra communications from the other automaton states. For instance, if the automaton has three states X, Y and Z, the Boolean is set to `true` by a core executing a task of state Y and a core executing a task of state Z. Hence, if a core executes a task t in state X and nothing if the state is not X, it has to wait for the task t inputs or the Boolean `epsilon`. Then, when the Boolean has been consumed it is set to `false`.

Example 15 shows the implementation of a quasi-static schedule of an automaton.

Example 15. We consider an automaton with three states and each state contains two tasks:
- State X: `A` and `B`
- State Y: `C`, `D` and `E`
- State Z: `F` and `G`

Furthermore, nodes `H`, `I` and `J` are outside the automaton.

We define the following schedule on four cores:
- Core 1: `H; Branch(A, D)`
- Core 2: `Branch(B,` ε`); I`
- Core 3: `Branch(C,` ε`); J`
- Core 4: `Branch(E, {F;G})`

To implement this schedule, we define some Boolean variables: `epsilon1`, `epsilon2` initialized to `false` when the program starts. Variable `epsilon1` is `true` when the current state is not X and variable `epsilon2` is `true` when the current state is not Y.

Variable `epsilon1` is set to `true` once in every states different from X and `epsilon2` is set to `true` once in every states different from X. In the listing of Figure 7.2, `epsilon1` is set when core 4 executes state Y or state Z. The variable `epsilon2` is set when core 4 executes state Z and in core 1 when state X is executed.

To implement `Branch(B, ε)`, the core waits until the inputs of `B` are ready or if the state is not X. Then, `B` is executed if it is ready, and otherwise nothing is executed.

```
// Core 1
task_H();
while(A not ready and D not ready)
 {} // wait
if(A ready) { //Branch(A,D)
    epsilon2 = true;
    task_A(); // State X
} else {
    task_D(); // State Y
}
```

```
// Core 2
while(B not ready and not epsilon1)
 {} // wait
epsilon1 = false;
if(B ready) { //Branch(B,epsilon)
    task_B(); // State X
}
task_I();
```

```
// Core 3
while(C not ready and not epsilon2)
 {} // wait
epsilon2 = false;
if(C ready) { //Branch(C,epsilon)
    task_C(); // State Y
}
task_J();
```

```
// Core 4
while(E not ready and F not ready)
 {} // wait
if(E ready) { //Branch(E,{F;G})
    epsilon1 = true;
    task_E(); // State Y
} else {
    epsilon1 = epsilon2 = true;
    task_F(); // State Z
    task_G();
}
```

Figure 7.2: Implementation of a quasi-static schedule.

A correct implementation of the `Branch` statement guarantees that there is no circular dependency between the tasks of the different states. With the presented method, each implementation of the `Branch` statement waits until some tasks corresponding to specific states or any other states represented with `epsilon`. Consequently, whatever the current state is, one branch is selected: either the branch corresponding to a task or the `epsilon` branch. Then the `epsilon` variable is set to `false`.

The implementation of an `epsilon` condition is correct if for an automaton this condition is made true in any state except one. Furthermore, this `epsilon` condition cannot depend on another `epsilon` condition. Finally, if a Boolean condition "*the current state is not X*" is set exactly one time in each other state, *i.e.*, when a task of a different state from X is activated, then the implementation of `epsilon` does not depend on tasks of X. Consequently, there is no cycle dependency in the implementation of `epsilon`.

In the Example 15, the location of the `epsilon` variable assignment has been arbitrary set, *e.g.*, `epsilon2` can be set when state X is executed in core 1 or core 2. Nevertheless, its location has an impact on the execution time.

Now, in Figure 7.2 we consider that the automaton condition does not depend on the result of `task_H`. In this case, the execution of `task_J` cannot start before the completion of `task_H` even though it does not depend on `task_H`.

In conclusion, our implementation of quasi-static schedule of automata can cause extra communication which leads to extra synchronization and extra delays. Nevertheless, all these communications are statically known which eases the WCRT computation.

7.3 System Configuration and Backend Library

In this section we first describe the boot sequence of the MPPA2. We explain how executables are created and how the platform is started.

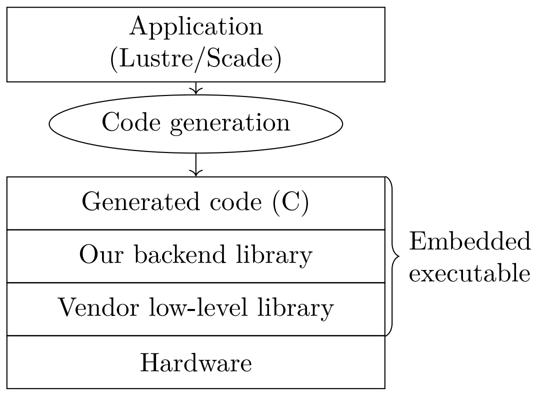


Figure 7.3: Code generation and low-level API

Figure 7.3 shows an overview of the code generation. The generated code relies on a backend library whose purpose is to abstract the processor. We generate platform agnostic code and the backend library implements the platform specific system code. This library is implemented using the Kalray MPPA2 low-level libraries or direct hardware configuration. Consequently, a new hardware platform can be handled by changing the backend library without changing the code generation. The functions provided by the backend library are prefixed with `backend_`.

The configuration of the MPPA2 is performed by executing the configuration code to initialize the synchronization mechanism (Section 7.3.2), initialize the communication buffers and configure the NoC. Finally, the cores execute the time synchronization code and begin the Synchronous program execution.

7.3.1 Boot Sequence

In this section we explain the boot sequence of the MPPA2. One executable is required for each cluster used: one executable per Compute Cluster plus one executable for the I/O Cluster. When the processor is powered on, it reads the I/O Cluster executable from the SDRAM memory. This executable is responsible for starting the Compute Clusters using a `spawn` function from the Kalray library parametrized with the cluster number and the name of the executable. The compute cluster binary is executed on core PE0 which is responsible for booting up the other cores. A core is started with the `pthread_create` function parametrized with the core number and the name of the thread function. Since there is no preemptive scheduler, this function is blocking until a core is free. A special parameter allows defining the target core sometimes called thread affinity.

7.3.2 Time and Event Synchronization

Efficient and predictable synchronization of the cores are required for the parallelization of time-critical software. The code generated by the [**Generation of system + communication code**] step can be time-triggered, event-triggered or both. The first requires time synchronization, the others requires efficient core synchronization mechanisms.

7.3.2.1 Inter-core Synchronization Mechanism on the MPPA2

In this section, we describe the implementation of the `backend_notify_all()`, `backend_notify_core(i)` and `backend_wait_event()` to synchronize cores of the same cluster. This implementation is specific to the MPPA2.

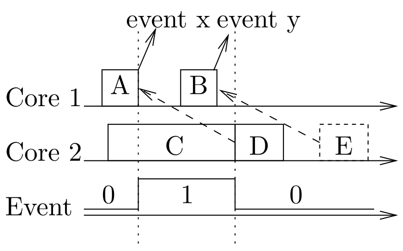


Figure 7.4: Events from `message_ram` can be overwritten.

The MPPA2 has 128 special registers called `message_ram` able to generate events for the cores. A core can be set in idle mode until an event occurs (see Section 4.3.2.3). Consequently, a power efficient synchronization is possible. The main drawback is that all the `message_ram` registers of the cluster generate the same event and the flag is overwritten if several events occur successively.

Example 16. In Figure 7.4, Task `D` depends on Task `A` and Task `E` depends on Task `B`. At the end of its execution Task `A` emits event `x` for Core 2. This event is not consumed instantaneously since Core 2 is executing `C`. Core 1 produces event `y` when it finishes `B`. Since, the core is able to store only one event for all the `message_ram` registers, it is overwritten and `E` cannot start even though `B` has been executed.

Example 16 shows that events can be squashed on the MPPA2, hence they cannot be discriminated. Our solution is to use a Boolean in the memory to discriminate the events beside the event notification. One variable is required for each pair of communicating tasks if they execute at the same frequency (see Section 3.1.3 for multi-periodic programs).

1-1 and 1-N Synchronization. The 1-1 synchronization allows one core to wake up another. The 1-N allows one core to wake up several others cores.

The `message_ram` register is configured with the set of destinations cores. The write of the maximal value on 64 bits to this register makes it generates an event to all the destination cores. Then a 1-1 synchronization is done by setting only one destination core and the 1-N is done by setting several cores of the cluster.

We then implement the `backend_notify_all()` function waking up all the cores of the clusters and the `backend_notify_core(i)` function waking up the core number `i` of the cluster. The `backend_wait_event` function turns the core in low power mode until an event is received.

Implementing these functions requires one `message_ram` register per cluster for the 1-N and one `message_ram` register per core for the 1-1 synchronizations.

7.3.2.2 Time Synchronization Protocol

In this section, we describe the clock synchronization protocol required to have a common clock accessible from any core of any cluster. The `backend_timer` function gives the current value of the clock.

Each core of the MPPA2 has a local timer. A core timer is private to its core and its value can be read and set without interfering with this other cores. The core timers are mesochronous, *i.e.*, they have the same frequency and only their phase is different.

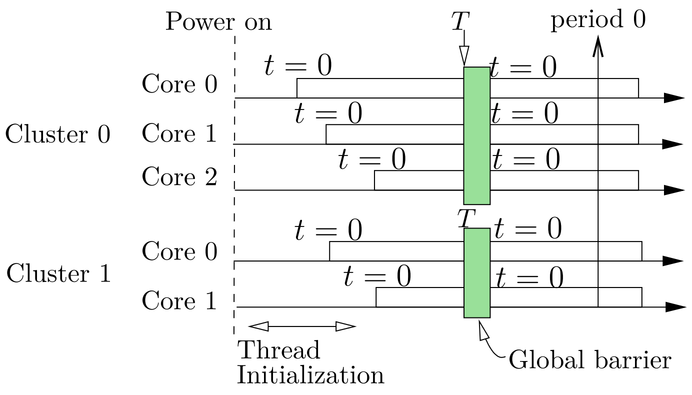


Figure 7.5: Clock Synchronization Protocol

There is one special timer in each cluster called DSU Timer. They are reset when the processor is reset. They are isochronous and thus can be used as a global time base. Nevertheless, it is only accessible through a single hardware bus. As a consequence, all accesses to this timer are serialized.

The protocol synchronizes all the cores using the DSU Timer without suffering of this serialization. In each cluster, when the cores are powered on, they go in idle mode, except one core which is responsible for synchronizing the others. This core waits for a time T on the DSU timer. When this time is elapsed, it wakes up the other cores of the cluster and all the cores of the cluster reset their timer or save the current time as the initial timestamp.

The synchronization of the cores in the cluster is done using the 1-N synchronization protocol presented in Section 7.3.2.1. The time T corresponds to the first time when all the cores are booted. A tight upper bound of this value is not necessary since this happens only at startup time. Instead, an arbitrary long value can be set and the initialization of the program can be stopped if the protocol did not work (forbidding the vehicle to start for instance).

7.3.3 NoC Configuration and Usage

In this section, we describe the library we implemented to configure and use the NoC. It provides a configuration procedures for the TX and RX engines. The TX engines is responsible for emitting packets on the NoC at a limited speed. It fills the packet header with the route and the destination buffer identifier. The RX engines are configured with an identifier, the address and the size of a buffer in memory. A RX engine stores the received payload in the buffer. The library is composed of the following procedures:

– `backend_noc_config_tx(id, dest_buffer_id, route, bandwidth)`: This function configures the TX engine identified by `id` to append a specific header at the beginning of each packet. This header contains the route `route` and the identifier of buffer `dest_buffer_id`. A maximal bandwidth can be specified.
– `backend_noc_config_rx(buffer_id, address, size)`: This function configures the RX engine to write all the packets whose destination buffer is `buffer_id` at `address`.
– `backend_send(id, address, size)`: This function uses the TX engine `id` to send a packet whose payload is `size` bytes at `address`.
– `backend_wait_event_NoC()`: This function makes the processor idle until a packet is received.

In practice, the configuration phase of the NoC is a sequence of `backend_noc_config_rx` and `backend_noc_config_tx` functions. The computation of the bandwidth and the route of each flow is explained in Section 7.5. The identifier of buffer and the TX engine are allocated statically when the channels are implemented.

The implementation of this library is done by directly accessing the registers of the NoC devices. This bare metal code eases the WCET analysis of the functions.

7.3.4 Cache Maintenance Functions

Non cache-coherent architectures require explicit modification of the cache state and explicit access to freshest data from memory:

– the `backend_write_barrier` makes sure that contents of the write buffer is committed to the memory
– the `backend_read_barrier` function invalidates the Data Cache to make sure that all next loads will be done directly in the memory
– the `backend_bypass_dcache` is used to directly read some data from the memory without changing the state of the cache. This function is to be preferred to `backend_read_barrier` to implement polling on one variable since it is faster.
– the `backend_bypass_wb` This function directly writes data to the memory without changing the state of the cache.

These functions are wrappers for the MPPA2 cache control instructions.

7.4 Communication for a Distributed and Multi-Banked Memory

As discussed in Chapter 5, our solution takes advantage of the banked shared-memory to minimize memory interferences. The **[Core and Buffer Allocation]** step associates one memory bank to each core of the processor and statically allocates the communication buffers, input data and code to the banks. This is done using a standard linker script and the configuration of the shared-memory to the blocked addressing mode where the addresses relative to the same memory bank are consecutive. The communications between the clusters are implemented with remote write through the NoC on the memory of another cluster.

Implementation is given for both the Top-Level Node method (toolchain 1) and the Parallel Subsets from the KCG compiler (toolchain 2) presented in Section 6.

7.4.1 Implementation of the Communications

We recall that a task `task_T` is composed of a communication function `wait_data_T` in sequence with the functional code `T_step` and the communication function `send_data_T`. In this section, we provide the implementation of the communication functions.

Basic Version. The basic implementation of input and output functions for Node 2 in shared memory is the following when the synchronization is event-triggered:

```
void wait_data_N2() {
  while(!token_N7_N2) {
    backend_wait_event();
  }
  token_N7_N2 = false;
}
```

```
void send_data_N2() {
  ctx_N4.i1 = ctx_N2.o1;
  ctx_N5.i2 = ctx_N2.o2;
  token_N2_N4 = true;
  backend_notify_core(5);
}
```

The `wait_data` function prevents executing the functional code until the input data is available. The basic implementation relies on a busy wait. A token mechanism ensures the data coherency. The `send_data` function copies the output data in the destination buffers and setup the token. It should be noted that, this synchronization is not required between N2 and N5 as they are executed on the same core in sequence.

Software-Based Cache Coherency. If the processor has a non-coherent cache architecture, this implementation would not work. As a consequence, we now present the modifications required to handle non-coherent caches.

```
void wait_data_N2() {
  while(!backend_bypass_dcache(
      token_N7_N2)){
    backend_wait_event();
  }
  backend_read_barrier();
  token_N7_N2 = false;
  backend_write_barrier();
}
```

```
void send_data_N2() {
  ctx_N4.i1 = ctx_N2.o1;
  ctx_N5.i2 = ctx_N2.o2;
  backend_write_barrier();
  token_N2_N5 = true;
  backend_write_barrier();
  backend_notify_core(5);
}
```

For `backend_wait_data`, the call to `backend_read_barrier` is required to fetch the new inputs data from the memory. The call to `backend_write_barrier` makes sure that the token modification is committed to the memory.

For `backend_send_data`, two calls to `backend_write_barrier` are required: one after the data have been written and one after the tokens have been written. The reason is that the memory system only guarantees the order of accesses for a single memory bank. If the data and the token are stored on several memory banks, the two write buffer flushes guarantee the ordering between them.

We now present the time-triggered implementation of the same program. The advantage of this implementation is that tight WCTT bounds can be guaranteed.

Time-Triggered Implementation. This version relies on the timer and the release date computation. It ensures safe bounds on the WCRT.

```
void wait_data_N2() {
  while(backend_timer() < i*period +
      release_N2){
    // Busy waiting
  }
  backend_read_barrier();
  token_N7_N2 = false;
  backend_write_barrier();
}
```

```
void send_data_N2() {
  ctx_N4.i1 = ctx_N2.o1;
  ctx_N5.i2 = ctx_N2.o2;
}
```

Time-Triggered plus Event-Triggered. In rare case, a release date could be missed for instance if a cosmic ray leads to a bit flip on the processor memory. For instance, a bit flip in a cache line leads to a cache refill which takes significantly longer than a cache hit. There are some hardware correction mechanisms on the platform; nevertheless, they cause memory access delays. Hence, taking them into account in all the release dates would make the WCRT over-pessimistic. Instead, the timing consequences of such rare events can be considered as a small reduction of the global performance while ensuring they can be handled when they happen.

We combine the time-triggered execution with an event-triggered check to ensure the functional correctness even if a release date has been missed. Obviously, the wait on release date must be performed before waiting on event.

NoC Communications. We statically associate one TX engine and one RX engine to each communication channel involving several clusters. This attribution is described in Section 7.6.6. The TX engines are configured with the route and a destination identifier using `backend_noc_config_tx`. The RX engines are configured with the destination buffer and an identifier using `backend_noc_config_rx`.

A structure carries the communication. There is one copy of this structure at each side of the channel. It contains the size to hold data and a token and finishes with a special field called `valid`. This field is used to know if a new value has been received. It is set to `true` before the transmission and the receiver set its local copy to `false` when it consumes the data. Furthermore, this field is located at the end of the structure to ensure it is written to the destination after the data.

Example of communication between from Node `Nsrc` to Node `Ndst`:

```
void wait_data_Ndst() {
  while(!backend_bypass_dcache(
      struct_Nsrc_Ndst.valid)){
    backend_wait_event_NoC();
  }
  backend_read_barrier();
  struct_Nsrc_Ndst.valid = false;
}
```

```
void send_data_Nsrc() {
  struct_Nsrc_Ndst.i1 = ctx_Nsrc.o2;
  struct_Nsrc_Ndst.valid = true;
  backend_send(NOC_Ndst_ID, &
      struct_Nsrc_Ndst, sizeof(
      struct_Nsrc_Ndst));
}
```

In `wait_data_Ndst`, `backend_bypass_dcache` and `backend_wait_event_NoC` is a blocking function returning when a new packet is received. The `backend_read_barrier` is required after the reception since RX engine does not changes the cache state.

In `send_data_Ndst`, the structure is filed with the data. No memory coherency function is required since the processor performs directly the copy of the data to the TX engine.

7.4.2 Delayed Communications

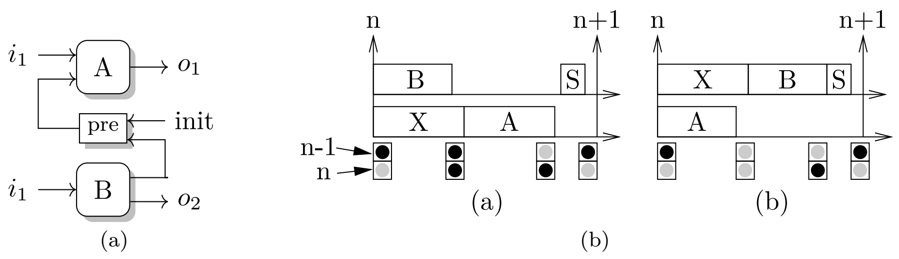


Figure 7.6: Buffer states are shown for two schedules of tasks `A`, `B`, `X` with a delayed communication from `B` to `A`. A light gray buffer has been consumed while a black buffer contains a non-consumed value. Swap task S ensures coherency between subsequent periods.

In the parallel subsets method, delayed communications are managed centrally by the top-level node. In the top-level node method, it is implemented it in a dataflow way. Delayed communications are replaced by direct communications with a swap task `S`. `S` is a regular task with an input buffer which writes its output to the dependent node buffer. This forms a double buffer. Some scheduling constraints are required for `S` to preserve the semantics.

Figure 7.6a represents a program with two nodes where `A` takes an output of `B` from the previous period and the output `init` for the first period. The task graph implementing this program has two tasks `A` and `B` and a delayed communication from `B` to `A`.

At the beginning of period n, the double-buffer contains only the output of `B` of period $n-1$. At the end, output $n-1$ has been consumed and output n has been produced. The swap task `S` moves the data in the double buffer from n to $n-1$ to prepare the next period.

Figure 7.6b shows the state of this buffer in two configurations. The purpose of the task `X` is pedagogical. It desynchronizes the `A` and `B` executions to show these two configurations. On the left side, producer `B` is executed before `A` completes, hence both $n-1$ and n outputs are presents when `B` finishes. On the right side, producer `B` is executed after `A` completes, hence the buffer is empty when `A` finishes.

The swap task `S` has two scheduling constraints due to the buffer state. `S` starts when output at n is present and the output at $n-1$ is absent, *i.e.*, it starts after both `A` and `B` have been executed.

If we apply this method to the example of Figure 7.1, one `S` task must be added. The schedule of Core 1 is changed to the following:

Core 1: `N1; N4; N6; S`.

Task `task_N6` performs a write to Task `S` input buffer and Task `S` waits for the completion of `task_N3` before sending the value to `task_N3`.

7.5 Contribution to Route Allocation

Network-on-Chip (NoC) and routing algorithms have been presented in Section 4.2. We considered the deadlock-free routing algorithms offering path diversity. We choose static routing for which the route is known before the transmission. We choose to keep only one route per flow to ensure the order of the transmitted packets. This route diversity lets a room for optimization.

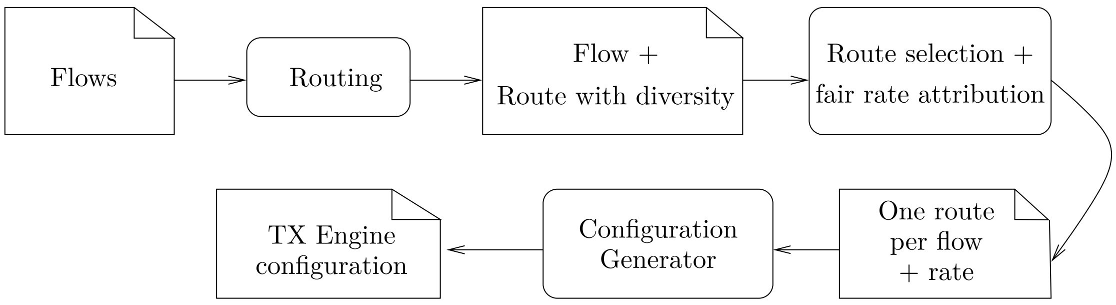


Figure 7.7: Method overview for routing and fair rate attribution.

Figure 7.7 shows the general method for routing and attributing the routes. The input is a set of unicast or multi-cast flows. The routing algorithm computes all the possible routes for this flow. The best route is selected for each flow optimizing a fair attribution of the flow rates. Finally, the configuration of the NoC TX engine is generated from the traffic limiter parameters and the route.

Contribution. In this section, we introduce an enumerative method and a heuristic based on linear programming to select the best max-min fair solution among all the routes combinations. We also present a variant of the HOE multi-cast algorithm for 2D-grid called HOE Dual-Path. This algorithm is both deadlock free and reaches all the recipient of a broadcast in two paths. These rate attribution and routing algorithms has been implemented in a network toolchain. This network toolchain is able, from a set of flows and the description of the network topology to configure the NoC and computing the end-to-end delay. The implementation of the complete toolchain is described in Section 7.6.6.

7.5.1 Route Selection and Flow Optimization

A routing algorithm with path diversity computes several routes for a couple of source and destination. Figure 7.8 shows two flows: f_1 from 1 to 9 in plain blue and f_2 from 3 to 7 in dashed red. There are three possible routes for each. Path diversity allows selecting a combination of routes which minimizes the concurrent usage of the link and then maximize the rate of the flows.

The criterion to maximize the rates could be the sum of the rates or the minimal rate. In both cases, this does not guarantee a max-min fair solution (see Section 4.2.6). As a consequence, we compare the different attributions using the lexicographical vector of flow rates.

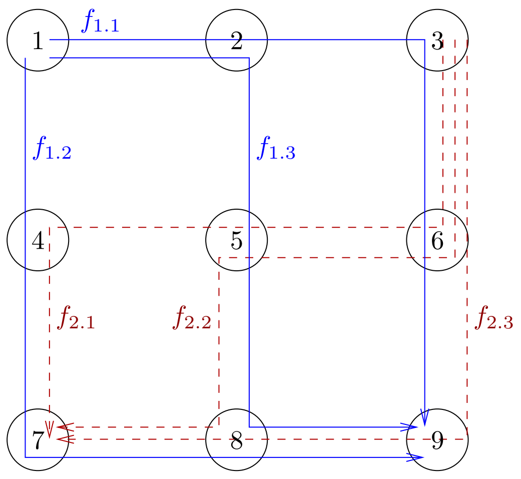


Figure 7.8: Example of path diversity for two flows.

To compare two vectors v_1 and v_2 of rates, we sort them in increasing order of rate. Then, we look for the first index i such that $v_1(i) \neq v_2(i)$. If $v_1(i) > v_2(i)$, then v_1 is greater than v_2 otherwise v_2 is greater than v_1. This method guaranteed the choice of the solution whose minimal rates are the highest.

We introduce two algorithms to select a combination of routes which maximizes the fair attribution of the flow rates. These algorithms also compute a rate for each flow. This corresponds to the route selection and fair rate attribution step from Figure 7.7: the Exploration for Unique Route Selection (EURS) algorithm and the LP-Based Heuristics for Unique Route Selection (LPURS) algorithm.

7.5.1.1 Exploration for Unique Route Selection (EURS)

Basic Exploration. At the output of a routing algorithm with path diversity several routes are possible for each flow. A basic solution to select the best max-min fair rate attribution is to compute this attribution for each combination of routes. The attribution is performed using the Water Filling [78] algorithm. For instance, in Figure 7.8, since both flows have tree alternative routes, there are $K = 9$ combinations hence, Water Filling is executed 9 times.

F is the set of flows. For each flow f_i, the set of possible routes is denoted r_i. One combination k is a set containing exactly one route r_i for each flow. K is the set of all route combinations k for the flows F. We look for the best fair combination of routes $k \in K$.

The comparison of the solutions is performed on the lexicographical vector of rates. Hence, for each $k \in K$, a max-min fair bandwidth allocation is computed with Water Filling [78] and v_k, the lexicographical vector of rates, is computed. At the end, the greatest vector v_k is selected.

Pruning. The basic algorithm can be significantly improved since the computation of Water Filling is sometime not necessary. In fact, an upper bound of the worst bandwidth can be computed. Our contribution to is to apply the pruning method to the space of exploration.

Instead of computing the Water Filling algorithm on all the combinations k, we first compute the worst bandwidth of every combination. To do this, for k we look for the link which is the most shared by flows. The worst bandwidth of k can be deduced from this information since it cannot be better than $1/n$ with n the number of flows sharing this link. A justification is that, if a link is shared by $1/n$ flows, the flow of minimal bandwidth taking this link has a bandwidth of a most $1/n$.

In Figure 7.8, if the combination is $k = \{f_{1.1}, f_{2.1}\}$, the most shared link is L36 with two flows. This leads to a worst bandwidth of $\frac{1}{2}$. For $k = \{f_{1.3}, f_{2.1}\}$, the links are not shared thus the worst bandwidth is 1.

The maximum of all the worst bandwidth is computed; then the combinations $k \in K$ whose worst bandwidths are under this maximum are removed from K. In the example, the $k \in K$ whose worst bandwidth is under 1 are removed.

For multicast traffic, the Water Filling algorithm has to be modified to set the same bandwidth to all the flows of each partition.

Despite this improvement, exhaustive enumeration of minimal path combinations becomes infeasible when the number of flows is high. To handle large problems, we designed a LP-based heuristic which avoids enumeration.

7.5.1.2 LP-Based Heuristics for Unique Route Selection (LPURS)

The presented EURS algorithm is based on an enumeration of non-splittable paths, meaning that a flow follows only one route. The drawback is that we need to consider all the combination of routes which leads to a combinatorial explosion.

In this section, we introduce an algorithm based on splittable paths, meaning that for a flow, several possible routes are followed concurrently. This avoids the combinatorial enumeration. Each alternative route of a flow is called *sub-flow*. For instance, in Figure 7.8, f_1 has three sub-flows: $f_{1.1}$, $f_{1.2}$ and $f_{1.3}$.

Our algorithm relies on the MMFSP [98] linear programming algorithm. MMFSP computes a max-min fairness rate attribution for splittable flows. It attributes max-min fair rates for all the sub-flows in such way that the sum of the sub-flows rates ≤ 1 flit/cycle. Each link has a variable in $[0, 1]$ which is set to the sum of the rates of all the sub-flows taking this link. A flow is considered as saturated either if all its sub-flows follow a saturated link or if the flow has a total rate of 1.

One iteration MMFSP maximizes the minimal sub-flows rates until they are saturated. Then, the saturated sub-flows are fixed and removed from the objective function. This algorithm iterates until all the flows are saturated and works in polynomial time.

The purpose of our LPURS algorithm it to take advantage of the fast rate attribution for splittable path performed by MMFSP while computing unsplit paths. This is performed by keeping only the sub-flows to whom MMFSP has attributed the highest bandwidth. The other sub-flows are set to 0.

If MMFSP attributes the same rate to all the sub-flows, hence our heuristics cannot be applied, then we enumerate the different solutions by setting successively each sub-flow to 0. Since there are several solutions, we compare the different combinations by computing the lexical vector of rates. Then, the solution with the greater vector of rates is selected.

The algorithm can be summarized as follows:

1. If there is only one sub-flow per flow, compute the lexicographical vector and exit.
2. Else, execute MMFSP until it converges
3. If for each flow, the rates of the sub-flow are the same, create as many instance of problem as there are sub-flows and restart the algorithm for each of them.
4. Else, for each flow, keep only the sub-flows of maximum rate then restart the algorithm.

For multicast traffic, the flows of all the partitions are encoded as the same flow in the linear program. This ensures that all the partition flows have the same bandwidth at the end.

Example 17 shows an application of the LPURS algorithm. It shows how the heuristic combined with MMFSP reduces the number of iterations compared to the EURS method.

Example 17. We apply the algorithm to the instance of flows of Figure 7.8. First, the MMFSP algorithm is executed. We obtain:

$$\left(f_{1.1} = \frac{1}{4}, f_{1.2} = \frac{1}{2}, f_{1.3} = \frac{1}{4}, f_{2.1} = f_{2.2} = f_{2.3} = \frac{1}{4}\right)$$

f_1 is saturated since it has a total rate of 1 and f_2 is saturated since all the sub-flow follows the L36 link which is saturated. Then, for each flow, we only keep the sub-flows with the best rate: $f_{1.1}$ and $f_{1.3}$ are removed. With MMFSP, we obtain:

$$\left(f_{1.2} = \frac{2}{3}, f_{2.1} = \frac{1}{3}, f_{2.1} = \frac{1}{3}, f_{2.1} = \frac{1}{3}\right)$$

All the sub-flows are equal, consequently, the algorithm enumerates the three combinations. First, $f_{2.2}$ and $f_{2.3}$ are set to 0. After execution of MMFSP, we obtain the vector of rates:

$$v_1 = \left(f_{1.2} = \frac{1}{2}, f_{2.1} = \frac{1}{2}\right)$$

Second, $f_{2.1}$ and $f_{2.3}$ are set to 0 and we obtain the vector of rates:

$$v_2 = (f_{1.2} = 1, f_{2.2} = 1)$$

Third, $f_{2.1}$ and $f_{2.2}$ are set to 0 and we obtain the vector of rates:

$$v_3 = (f_{1.2} = 1, f_{2.2} = 1)$$

Finally, we compare v_1, v_2 and v_3 and select the greatest vectors v_2 or v_3. In this example, we enumerated 3 cases where the EURS enumerated 9 cases.

7.5.2 Contribution to Multicast Path-Based Routing

As presented in Section 4.2.4 a common way of implementing the multicast is using several routes where each route reaches a subset of the destinations. Each subset is called partition. A path-based multicast algorithm has the following steps: the partitioning of the destinations into sets and the routing of each partition individually.

The main multi-cast partitioning algorithms are the Dual-Path (DP), Multi-Path [91] (MP) and Column-Path [19] (CP). In a 2D-grid dual-path leads to 2 partitions, multi-path to 4 partitions and column-path to at most $2n$ partitions where n is the number of columns of the grid.

On one hand, there is a limited number of TX engines. They are able to store only one route and their configuration takes time. On the other hand, dual-path provides an optimal number of partitions for a 2D grid since with a deadlock-free Hamiltonian routing it is possible to reach all the destinations with one ascending path and one descending path.

Routing the partitions using the Hamiltonian routing algorithm leads to long routes. As a consequence, HAMUM [42] has been presented and has been improved with the HOE [7] routing algorithm.

To the best of our knowledge, HAMUM and HOE have been applied to a multi-path and column-path partitioning but not on the dual-path partitioning. As a consequence, we implemented HOE Dual-Path (HOE DP) which combines both the advantages of a minimal number of partitions and the HOE adaptive and deadlock-free routing algorithm. By way of comparison, we also implemented the HAMUM Dual-Path (HAMUM DP) algorithm.

Evaluation of HOE DP and HAMUM DP compared to other multi-cast algorithms is given for a uniform traffic in Section 8.1.2.

7.5.3 Deadlock-Free Routing Algorithm for the Kalray MPPA2

In this section, we present the implementation of the routing step from Figure 7.7 for the MPPA2.

Most of the presented routing algorithms are designed for regular 2D-grid topologies. The MPPA2 NoC has a quite complex topology but can be seen as a 2D-Grid of Compute Clusters. This grid is presented in Figure 4.20 page 62.

In this topology, some routes reaching I/O Clusters are not possible. For instance there is no X-Y route from Node 12 to Node S2. The reason is that some routers are missing in the middle of the grid. Kumar *et al.* formalized this problem by introducing the *network regions* [84] and Hosmark *et al.* presented a deadlock-free algorithm for networks with region in the general case [75].

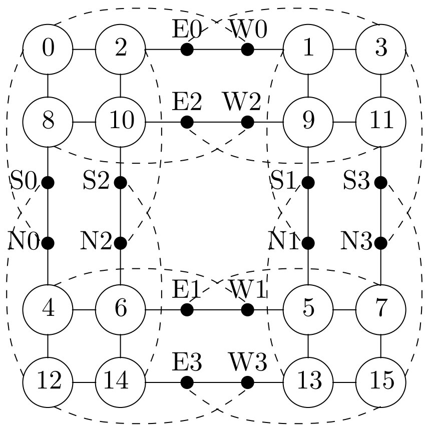


Figure 7.9: The Kalray MPPA2 NoC can be seen as a 2D-Grid of clusters. The I/O clusters are outside the grid.

We choose a quite simple solution which consists in replacing the hole in the middle of the grid by virtual links. For instance, Node 10 and Node 6 are linked with a virtual link composed of the 10-S2, S2-N2 and N2-6 links. Hence, the network region is ignored. This grid is represented in Figure 7.9. The dashed links are not part of the grid. This topology authorizes routing communications between any Compute Cluster with a standard 2D-Grid routing algorithm. At the end, the virtual links are replaced with hops through the I/O Cluster routers.

We now describe a procedure to compute a route from an I/O Cluster to any other cluster, or from a compute cluster to an I/O Cluster. Since I/O Clusters are outside the grid, an extra step to enter or exit the grid is required. The main part of the route is inside the grid.

The steps to enter and exit the grid rely if needed to the dashed links. Then, the route between the two compute clusters of the grid is computed. Finally, the different parts are concatenated. We provide some examples:

Example 18. The first example is a flow from node E2 to node 5. The routing is straightforward since the XY route from E2 to node 5 exists. Nevertheless, all the routes from an I/O Cluster to a Compute Cluster are not possible. We then present a second example:

Considering a flow from node 11 to node W1. The XY route does not exist. Consequently, we compute the XY route from node 11 to node 4. The grid is exited using the 4-W1 link.

As presented in this section, the routing on the MPPA2 requires the concatenation of a deadlock-free route and some steps to enter or exit the grid. We stated previously that concatenating several deadlock-free routes can lead to deadlock routes. We now explain why this specific concatenation cannot lead to deadlock-prone routes.

There are tree distinct sets of network elements. The first set E_{grid} contains the links and the turns inside the grid including the virtual links. For instance, the virtual links L2-1, L1-2 (through E0 and W0), the turns T2-WS and T2-SW are part of this set. Dashed links are not part of the set, *e.g.*, LW2-8 and L8-W2.

The second set E_{entry} contains the elements required to enter the grid: the dashed links from I/O cluster to cluster (LW0-0, LW2-8, LS0-12, T3-EW, etc) and the local input links of the I/O clusters ($E3_{egress}$, $N1_{egress}$, $N2_{egress}$, etc).

The last set E_{exit} contains the elements required to exit the grid: the dashed links from Cluster to I/O Cluster (L4-W1, L7-E1, L3-N3, etc), and the local output links of the I/O Clusters ($N1_{ingress}$, $E3_{ingress}$, etc).

We consider a deadlock-free route between two nodes of the grid. This route is only composed of network elements of E_{grid} and does not contain any elements of E_{entry} and E_{exit}. The entry and exit steps are deadlock-free.

This deadlock-free route between two nodes of the grid can be prefixed with a route to enter the grid (elements of E_{entry}) and suffixed with a route to exit the grid (elements of E_{exit}). Consequently, there exists a partial order between the elements of E_{entry} and the elements of E_{exit} and the flows traverse these resources only in this order. Then, the complete route is deadlock free, as required.

7.6 Deterministic Network Calculus (DNC) Tool

In this section, we present a tool for the computation of end-to-end latency (WCTT) over the network. This tool also computes the hardware FIFOs level and check for overflow. A rate and a packet size are attributed to each flow as presented in Section 7.5.1. Our DNC formulation is called Linear Formulation and has been done in collaboration with Benoît Dupont de Dinechin and published in [46].

7.6.1 Effects of Link Shaping

A bandwidth limiter guarantees an affine curve $\gamma_{r,b}$ for a flow. It injects the packets atomically if the flow has enough credits. Consequently, for a flow i, the value of b must be at least equal to the flow packet size l_i^{max}. The parameter r is configured to the flow rate.

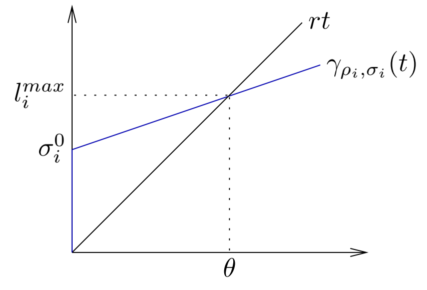


Figure 7.10: Effects of link shaping on burstiness.

We now discuss the effects of link shaping on the burstiness as presented in Figure 7.10. Before the shaping, the packet is considered to be transfered instantaneously, hence its burstiness is l_i^{max}. Nevertheless, the link has a capacity of 1 flit/cycle. Since this capacity is higher than the long-term flow rate, the link shaping can reduce the burstiness of the flow.

We look for the affine arrival curve $\gamma_{\rho_i,\sigma_i^0}$ for the flow f_i after link shaping. A link has a rate r, then, the date of end of transition of a packet of size l_i^{max} is $\theta = \frac{l_i^{\max}}{r}$. Furthermore, $l_i^{\max} \leq \sigma_i^0 + \rho_i\theta$. The long-term rate ρ_i keeps unchanged:

$$\forall f_i \in F : \sigma_i \geq \sigma_i^0 \triangleq l_i^{\max}\frac{r-\rho_i}{r} \tag{7.1}$$

For each flow, one arrival curve is computed at each network element of the route. Therefore, we compute a series of arrival curves. For each network element j traversed, one arrival curve γ_i^j is computed.

Effects of Link Shaping on Queues. Considering a queue q^j which receives the aggregates of flows F^j passing through it. The arrival curve is a leaky-bucket γ_{ρ^j,σ^j} with $\rho^j = \sum_{f_i \in F^j} \rho_i$ and $\sigma^j = \sum_{f_i \in F^j} \sigma_i^j$, shaped by the turn peak rate r. This yields the arrival curve $\min(rt, \sigma^j + \rho^j t)1_{t>0}$, which is a special case of the standard T-SPEC arrival curve $\alpha(t) = \min(M + pt, rt + b)1_{t>0}$ used in IntServ [52]. We use this arrival curve for burstiness augmentation due to FIFO and end-to-end delay computation.

7.6.2 Routers and Network Properties

In the routers of the MPPA2, there is one queue per turn. The output links are arbitrated at the packet size granularity with a round-robin policy.

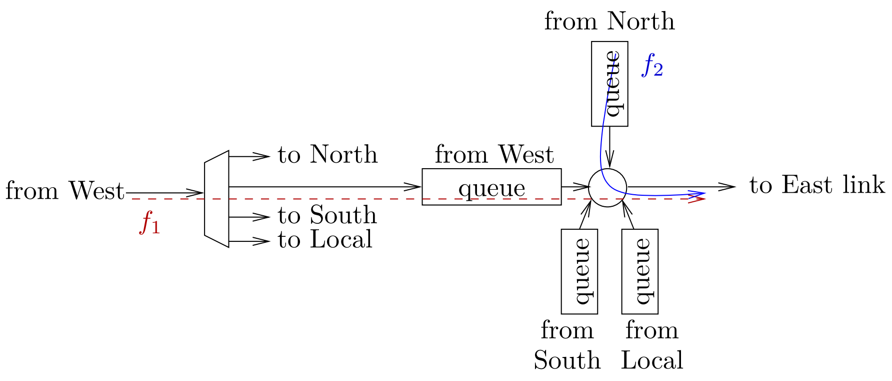


Figure 7.11: Router elements of the turns to the East link.

Figure 7.11 shows the complete path from the west input of a router to the east output. When a packet enters the router, it is directed to one output queue according to the route. Then, the queues destined to an output link are arbitrated. The dashed red flow takes the West to East turn, while the plain blue flow takes the North to East turn.

An arbiter is *active* if more than one queue is used for its output. It is not active either because there is no flow passing through it or only one queue is used. The number of queues used at input of the arbiter is denoted n^j.

In Section 4.2.7 we presented some common abstractions for network elements: the FIFO multiplexing and the blind multiplexing. The order of the packets entering an arbiter of the MPPA2 router is guaranteed for each queue, *i.e.*, for one turn. Nevertheless, the order is not guaranteed between packets of different queues of the arbiter, *i.e.*, for the same output link but different turns. Blind is the conservative policy that can be used when no other policy can be applied, consequently it is pessimistic. Some previous works only applied FIFO [89] or blind [20].

7.6.3 Arbiter Service Curve

We consider a network element arbitrating among several queues (from North, from South, from West, etc). Each queue receives an aggregated arrival curve. We compute the service curve offered to each queue.

Definitions.
- F^j is the set of flows for the active queue q^j
- n^k is the number of queues used at ingress of the arbiter
- A^k is the set of active queues in the link arbiter
- $B^j \triangleq A^k - \{q^j\}$ is the set of active queues in the link arbiter except q^j

Round-Robin. The round-robin arbiter tests each queue in a fixed order and accepts one packet if the queues is not empty. The round-robin policy is fair if the packets have the same size [1]. Consequently, the service curve of a round-robin is a stair function, nevertheless, it can be approximated as a long-term rate-latency function $\beta_{R,T}$ [20]. If the packets have the same size l^{max}, the queue q_j from the arbiter k benefits from a service curve with the following parameters:

$$R^j = \frac{r}{n^k} \text{ and } T^j = (n^k - 1)\frac{l^{max}}{r} \tag{7.2}$$

R corresponds to the fraction of bandwidth reserved to one queue and T corresponds to the time for each other queue to transmit one packet. With packets of different size, this rate is proportional to the packets size. The weighted round-robin service curve β_{R^j,T^j} for q^j is:

$$R^j = \frac{r l_{F^j}^{\min}}{l_{F^j}^{\min} + \sum_{k \in B^j} l_{F^k}^{max}} \text{ and } T^j = \frac{\sum_{k \in B^j} l_{F^k}^{max}}{r} \tag{7.3}$$

$l_{F^j}^{min}$ is the minimal packet size of the flows F^j and $l_{F^j}^{max}$ the maximal packet size. Since the packets in a queue q^j can have different sizes, a conservative hypothesis is to consider the minimum packet size for the queue q^j and the maximum packet size for the other queues of A^j.

Blind or Round-Robin. There are some restrictions on the usage of the round-robin policy. Figure 7.11 shows two flows f_1 and f_2 multiplexing into a round-robin. Assume respective rates $\rho_1 = \frac{1}{3}$ and $\rho_2 = \frac{2}{3}$. Assume the duration $T1$ to transmit one packet. If packets have the same size, the round-robin gives a long-term rate-latency β_{R^j,T^j} with $R = \frac{1}{2}$ and $T = T1$. Nevertheless, if the flows are sending continuously, the round-robin transmits one packet of f_1 and two packets of f_2 in a sliding window of $3 * T1$. Consequently, this service curve is not applicable since the actual service curve guaranteed for f_2 has a rate of $\frac{2}{3}$. Finally, if the rate of the aggregated input flows of one queue is greater than R^j from Equation 7.3, the round robin policy cannot be applied for this queue and the conservative blind multiplexing must be applied. In all other cases, one of blind and round-robin offering the least delay T is chosen.

Blind. Applying blind multiplexing to the round-robin arbiter consists in considering that this round-robin serves packet at peak rate r. The service curve offered to queue q_j is the leftover service curve if the flows of the other queues (Section 4.2.7). Application of **Theorem 6.2.1 (Blind Multiplexing)** [86] yields the blind multiplexing service curve $\beta^j = \beta_{R^j,T^j}$ for q^j:

$$R^j = r - \sum_{k \in B^j} \rho^k \text{ and } T^j = \frac{\sum_{k \in B^j} \sigma^k}{r - \sum_{k \in B^j} \rho^k} \tag{7.4}$$

7.6.4 Queue Service Curve

Considering a queue q_k entering a rate-latency arbiter of service curve β_{R^k,T^k} with a rate R and a latency T. Considering a flow of interest f_i with an arrival curve $\gamma_{\rho_i^k,\sigma_i^k}$. We want to compute the arrival curve $\gamma_{\rho_i^{k+1},\sigma_i^{k+1}}$ at the output of the arbiter. If the arbiter is active, the flow is delayed resulting in a burstiness increase leading to a new arrival curve $\gamma_{\rho_i^{k+1},\sigma_i^{k+1}}$.

If a flow is alone in the queue q_k, the rate-latency service curve β_{R^k,T^k} of the arbiter is directly applied to the arrival curve $\gamma_{\rho_i^k,\sigma_i^k}$ of the flow. The rate keeps unchanged.

$$\sigma_i^{k+1} = \sigma_i^k + \rho_i^k T^k \tag{7.5}$$

If several flows are multiplexed in the queue, this service curve β_{R^k,T^k} is shared among all the flows of the queue, therefore we have to compute the residual service curve $\beta_{R_i^k,T_i^k}$ guaranteed to the flow f_i.

[1] Deficit Round-Robin is a fair alternative with packets of different sizes [118, 24].

F_k is the set of flows of the queue q_k. The aggregate arrival curve corresponding to the competitors of f_i in the queue is an affine function γ_{ρ_2,σ_2} with

$$\rho_2 = \sum_{l \in F^k, l \neq i} \rho_l \text{ and } \sigma_2 = \sum_{l \in F^k, l \neq i} \sigma_l^k \tag{7.6}$$

We compute the residual service curve guaranteed to the flow f_i by applying **Corollary 6.2.3 (Burstiness Increase due to FIFO)** [86]. This yields the residual service curve $\beta_i^k = \beta_{R_i^k, T_i^k}$ for an active queue q^k traversed by f_i:

$$R_i^k = R^k - \rho_2 = \rho_i \text{ and } T_i^k = T^k + \frac{\sigma_2}{R^k} \tag{7.7}$$

We apply **Theorem 6.2.2 (Burstiness Increase Due to FIFO Multiplexing, General Case)** [86] where the flow of interest f_i is an affine curve and the aggregate arrival curve α_2 of f_2 is sub-additive. Due to link shaping $\alpha_2(t) = min(rt, \rho_2 t + \sigma_2)1_{t>0}$. The arbiter guarantees to the aggregate of the flows a rate latency service curve β_{R^k,T^k}.

If $\rho_i + \rho_2 < r$, f_i the arrival curve of f_i is $\gamma_{\rho_i^{k+1}, \sigma_i^{k+1}}$. The rate remains unchanged:

$$\sigma_i^{k+1} = \sigma_i^k + \rho_i \left(T^k + \frac{\sigma_2(r + \rho_i - R^k)}{R^k(r - \rho_2)} \right) \tag{7.8}$$

Since the network is feed-forward, the successive arrival curves for each flow can be computed. Then, the utilization level of a queue q_k can be checked with the following formula:

$$Q_{usage}^k = \sum_{l \in F^k, l} \sigma_l^k$$

The queue size Q_{max} depends on the hardware. If $Q_{usage}^k > Q_{max}$ the queue overflows and the quality of service cannot be guaranteed. In this case, the rate of the flows or the packet size can be changed make the flows comply with the hardware.

7.6.5 WCTT Bound

The WCTT is computed using a method similar to the Separated Flow Analysis (SFA) [20]. The difference is that our method relies on FIFO multiplexing for flow aggregation while SFA relies on blind multiplexing.

First, the service curve of each queue is computed and the residual service curve $\beta_{R_i^j, T_i^j}$ of each active queue q_j traversed by f_i is computed. The residual service curve is computed using Equation 7.3 if the round-robin policy is applied and Equation 7.4 if the blind policy is applied. If the queue is shared among several flows, the residual service curve is computed using Equation 7.7.

Second, equivalent service curve $\beta_{R_i^*, T_i^*}$ guaranteed by the NoC to flow f_i is computed. This is obtained by the convolution of the residual service curves $\beta_{R_i^j, T_i^j}$. $\beta_{R_i^*, T_i^*}$ is the convolution of rate-latency curves, hence, rate R_i^* is the minimum of the rates and latency T_i^* is the sum of the latencies:

$$R_i^* = \min_{j \in Q_i} R_i^j \text{ and } T_i^* = \sum_{j \in Q_i} T_i^j \tag{7.9}$$

Last, the end-to-end latency bound is computed as the maximum horizontal deviation between the arrival curve α_i of flow f_i and the service curve $\beta_{R_i^*, T_i^*}$. This flow is injected at rate ρ_i and burstiness σ_i, however it is subjected to link shaping at rate r as it enters the network. As a result, $\alpha_i = \min(rt, \sigma_i + \rho_i t)1_{t>0}$:

$$d_i^* = T_i^* + \frac{\sigma_i(r - R_i^*)}{R_i^*(r - \rho_i)} \tag{7.10}$$

Evaluation of the end-to-end latency will be given in Chapter 8.

7.6.6 NoC Configuration Toolchain for the Kalray MPPA2

In this section, we present the network toolchain able to generate the NoC configuration and to compute the end-to-end latency from a set of flows and the topology of the NoC.

Topology File Format. We define a Yaml file representing the network topology. This representation is precise enough to assign the physical delays.

This file consists of a list of nodes with the ingress and egress link and a list of links for which we attributed a traversal delay. This traversal delay depends on the physical length of the links. Then, the file contains the list of turns. The arbiters are described with several turns as input and one link as output. Finally, the switches demultiplexing the flows of one link into several turns are described.

The routes are computed with an abstract link and node representation, then they are converted into a concrete representation as described in the topology file.

Flow File Format. A Yaml file representing the flows is generated from the Parallel Intermediate Representation (PIR) of the program and the mapping. This file contains the set of NoC communications with the source node, destinations and packet size. This file is then processed and a new file is produced at each step of the network toolchain.

The routing tool takes the flow file and the topology file as input and produces a new flow file with the routes. Then, the rate attribution tool computes flow rates and produces a new file which is used at ingress of the DNC tool for the computation of the end-to-end latency and the queue overflow check which produce a new file. This file is used at ingress of the task release date computation tool. A NoC configuration tool also attributes one TX engine and one RX engine for each flow. Then it produces the packet shaper configuration and the bandwidth limiter parameters. This configuration is included in the final binary.

DNC Computation Tool. The DNC Computation Tool computes the end-to-end latency. Previous version of the DNC computation tool was relying on a LP solver to attribute the flow rate and compute the end-to-end latency bounds at the same time [44]. One contribution is an end-to-end computation taking advantage of the feed-forward property of the flows to perform a one pass computation. Our implementation computes a flow arrival curve at each queue. Then, it computes the service curve at each turn. The choice between blind and FIFO is done with the constraint formulated in Section 7.6.2. Furthermore, the blind policy can locally be chosen if it provides better delays than the round-robin policy. The delay of each flow is expressed with the convolution of the service curve crossed by the flow. Finally, the equations are sorted topologically before computation.

7.7 Conclusion

In this chapter we presented our implementation toolchain for parallel program on the Kalray MPPA2. The input of the toolchain is the PIR of the program, a mapping and a scheduling. The contributions of this chapter are the following:

- The configuration of the platform and a time-triggered execution model ensuring real-time constraints with a clock synchronization protocol
- The implementation of the schedule of data-flow program with automata and delayed communications.
- Generation of shared-memory communication code including software-based cache coherency.
- An enumerative fair rate attribution algorithm for the flows and a fast LP-based heuristics.
- A deadlock-free multi-cast routing algorithm and a deadlock-free routing algorithm for the MPPA2 topology.
- A complete toolchain of routing, static rate allocation and end-to-end delays bounding on the NoC generating the network configurations.

Evaluation of the implementation is performed in Chapter 8.

8 Experiments

In this chapter, we apply our toolchain to both synthetic, artificial benchmarks and to representative avionics case-studies. First, we compare our Network-on-Chip routing and flow optimization techniques with classical methods for Mesh-2D. Second, we compare our Deterministic Network Calculus (DNC) tool with alternative methods. Then we discuss the performance obtained on the case studies.

8.1 Route Selection and Routing Algorithm Comparison

We apply the EURS and LPURS algorithms to some artificial instances of flows (Bit-Complement, Bit-Reverse, Shuffle and Tornado presented in Section 4.2.8) and the two use cases E1 and E2 from Section 8.2. The flows are routed with Hamiltonian Odd-Even (HOE), Turn Prohibition (TP) and Simple Cycle Breaking (SCB). For the final rates attribution, we consider the minimal rate as a good indicator of fairness. Examples are named with the initial letters of the instance of flows followed by the initial letters of the routing algorithm.

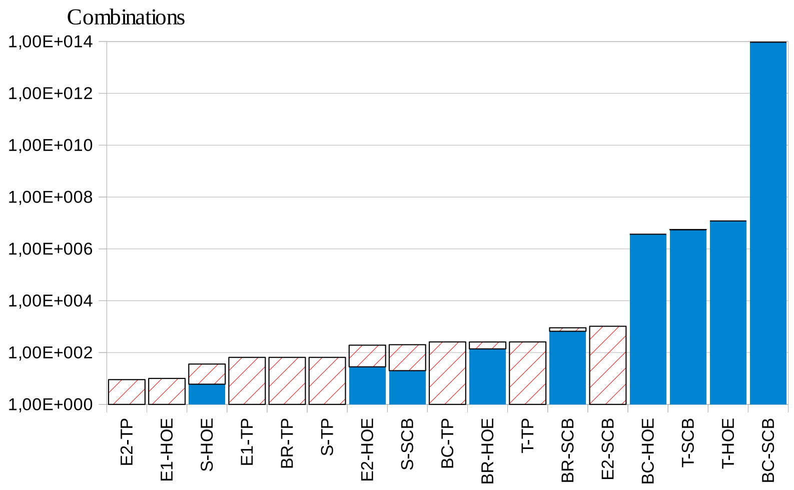


Figure 8.1: The top of each bar is the total number of combinations for a test case. In blue, the number of combinations computed after pruning Exploration for Unique Route Selection (EURS) algorithm. In white dashed of red, the number of pruned combinations. The y-axis has a logarithmic scale.

The Exploration for Unique Route Selection (EURS) algorithm is an improvement over the basic route selection algorithm which enables pruning of some combinations. EURS computes an upper bound of the smallest rate for each combination and avoids applying the Water Filling algorithm if this rate is not maximal. Figure 8.1 shows benefit of the pruning for all the test cases by comparing the number of combinations and the number of pruned combinations. The y-axis has a logarithmic scale. Pruning enables high reduction of the route combinations. Nevertheless, this reduction is highly variable depending on test case. For instance, for BC-HOE, T-SCB, T-HOE and BC-SCB, less than 1% of reduction is observed. A drawback of this pruning method is that it applies only on the link of the network which is shared by the highest number of flows. If all the combinations lead to the same maximum share, there is not gain. However, it should be noticed that EURS behaves very well for 8 out of 18 test cases and well for 4 others.

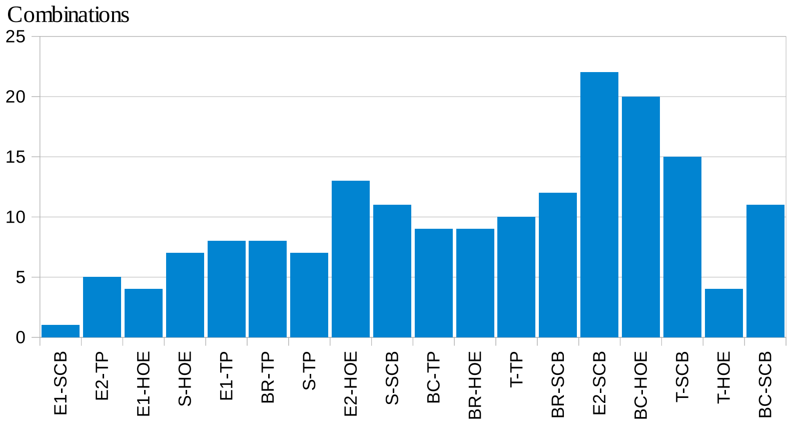


Figure 8.2: Evaluation of the number of final combinations computed by LPURS on the test case. Final combination are drastically reduced compared the EURS method (Figure 8.1).

The LP-Based Heuristics for Unique Route Selection (LPURS) algorithm reduces the number of evaluated combinations by considering at the same time all the paths of each flow. This leads to a high reduction of the number of final combinations. As depicted in Figure 8.2 there is a maximum of 22 solutions evaluated for E2-SCB.

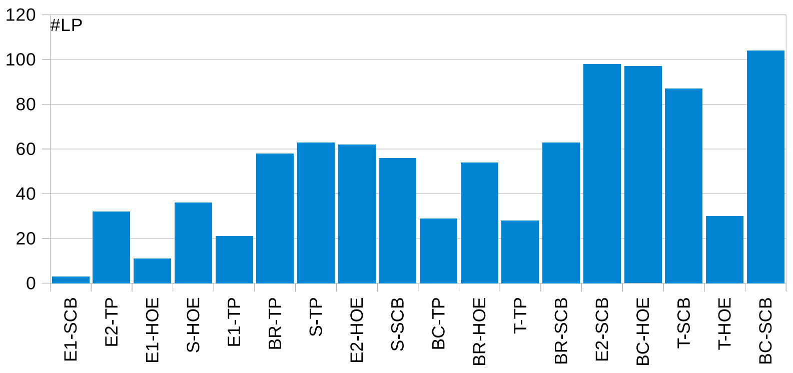


Figure 8.3: Comparison of the number of LP programs required by LPURS on the examples.

This very high reduction has to be put in perspective with the number of linear programs (LP) required to perform the algorithm and MMFSP. Figure 8.3 shows a maximum of 104 LPs for BC-SCB compared to the 10^{14} combinations evaluated with the EURS algorithm. Finally, if we compare the number of combinations of EURS to the numbers of LP programs of LPURS, we observe an average reduction of more than 10^{10} times.

Figure 8.4 compares the result of the route selection for the two algorithms LPURS and EURS. We compare the solutions with the minimum, maximum and average rate attributed to the flows. For 13 out of 24 examples, LPURS and EURS perform the same. If the criterion is the minimal rate both

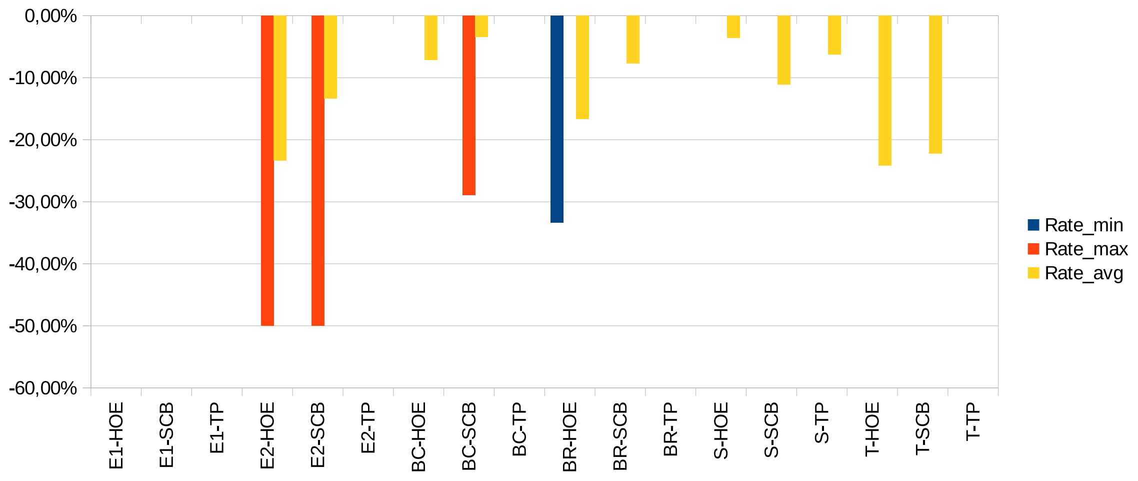


Figure 8.4: Rate reduction of the LPURS heuristics compared to the optimal EURS algorithm. Comparison is performed on the minimal, maximal and average rate. Longer lines mean worst LPURS performances.

algorithm perform similarly except for Bit-Reverse routed with HOE whose minimal rate is reduced of 33.3%. Globally, LPURS reduces the average rate by 5.8% and the maximal rate by 5.4%.

An enumerative solution is not applicable for big problems since it leads to a high number of combinations to evaluate. The EURS algorithm provides an optimization to reduce the number of combination. This reduction is still not sufficient. To conclude, we presented an LP-based heuristic LPURS whose result is comparable to the EURS but reduces the number of evaluated combinations.

8.1.1 Comparison of Deadlock-Free Unicast Routing Algorithms

In this section, we compare some deadlock-free routing algorithms for the MPPA2 NoC: XY, Hamiltonian Odd-Even (HOE) Simple Cycle Breaking (SCB) and Turn Prohibition (TP). The path diversity is the number of possible routes between two network nodes. We compare path diversity of each use case on the minimal and maximal number of possible routes observed for all the flows, and the average of the path diversity of all the flows. Then we compare the minimal rate resulting of a max-min fair rate attribution. The routing algorithms are applied to the four artificial instances of flows presented in Section 4.2.8 and the two use cases E1 and E2 presented in Figure 8.10 of Section 8.2.

For TP and SCB routing algorithms, the set of possible routes is obtained by enumeration of all the shortest paths. We used a modified Bellman-Ford algorithm for this purpose.

Figure 8.5 shows path diversity of the routing algorithm for different test cases. A high path diversity gives more alternatives to select the best combination. SCB provides the best path diversity. For the Bit-Complement instance of flows, it provides up to 21 alternative paths per flow. HOE provides better path diversity than TP.

We compare the minimal rate attribution for each flow using the optimal EURS method limited to the 50000 first combinations. A higher minimal rate is better since it is the result of a fair attribution. Glass and Ni [60] stated than a better path diversity results in better performances of the network. Figure 8.6 shows the minimal rate for each routing algorithm. It shows that the relation between path diversity and rate is not clear. For instance, SCB provides high path diversity for Bit-Complement, whereas it provides one of the worst rates for this instance of flow. Furthermore, XY provides the best minimal rate. The best minimal rate is always provided by HOE or XY.

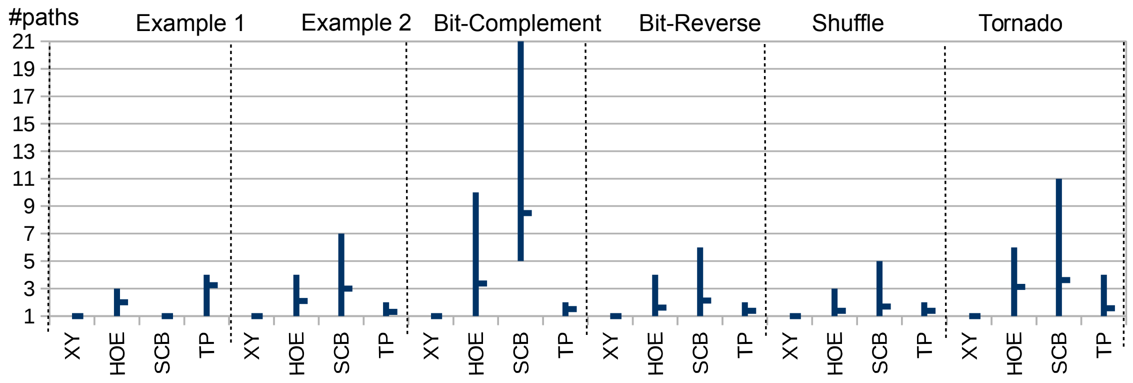


Figure 8.5: Minimal (resp. maximal) number of alternative paths among all the flows of the test case are represented with the bottom (resp. top) of line. Average number of alternative paths for the flows of each test case are represented with the horizontal line.

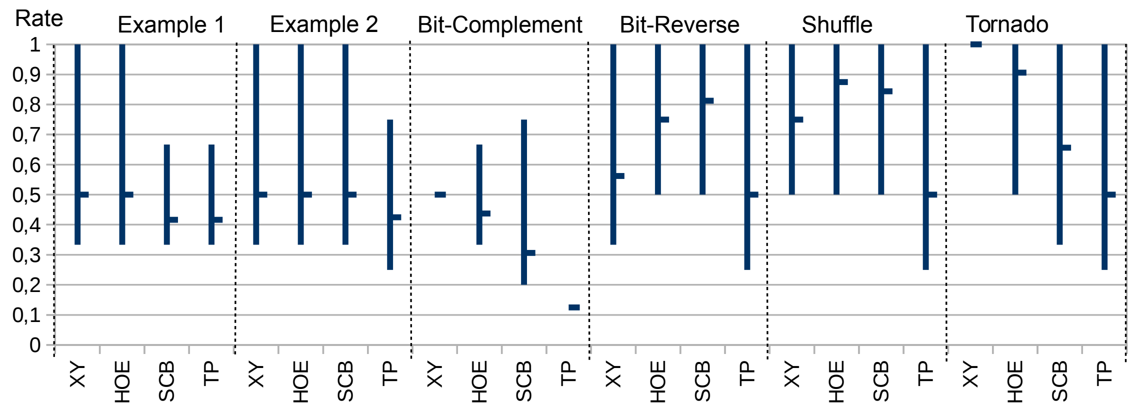


Figure 8.6: Rate for each test case after application of the EURS route selection on the flow. The rate of the slowest (resp. fastest) flow of the test case is represented with the bottom (resp. top) of the line. The average rate of all the flows of each test case is represented with the horizontal line.

Comparison of HOE and XY. Even though XY has no path diversity, it performs better than HOE on Bit-Complement and Tornado in term of minimal rate. The explanation is that HOE does not allow some XY path, as illustrated in Figure 8.7. Nevertheless, HOE offers efficient multi-cast routing algorithms as presented in the next section.

8.1.2 Evaluation of the HOE-DP Unicast-Based Multicast Routing Algorithm

In this section, we compare multicast routing algorithm based on the Dual Path (DP) and the Column Path (CP) partitioning algorithms. The classical DP [91] which is based on the Hamiltonian routing and the CP [19] which is based on XY routing are compared to the HOE DP and HAMUM DP introduced in Section 7.5.2.

The evaluation method is described in Section 7.5. The input flows are generated with a uniform traffic where each pair of nodes has a probability p to communicate. The max-min fair routes are selected using the EURS algorithm. The experiments is performed on $p \in [0.01, 0.5]$ with steps of 0.025. For each value of p, the result is the average of the minimal rates obtained for N=100 experiments.

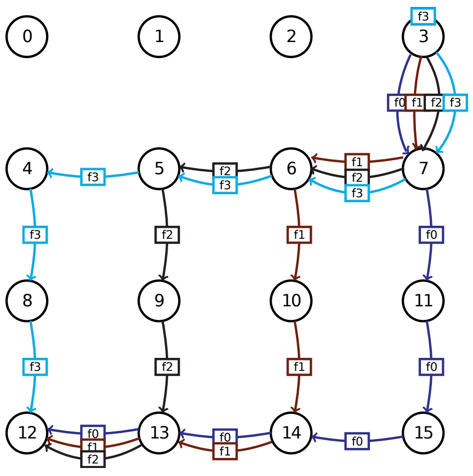


Figure 8.7: Enumeration of the possible HOE paths between 3 and 12 on a 4x4 2D-mesh.

The aim is to guarantee the best minimal rate for each flow. Figure 8.8 shows the minimal rate for several parameters p of the uniform law computed with the following formula:

$$E(p) = \frac{1}{N} \sum_{i \in [1,N]} \min_{f \in \text{flows}_i} \text{rate(f)}$$

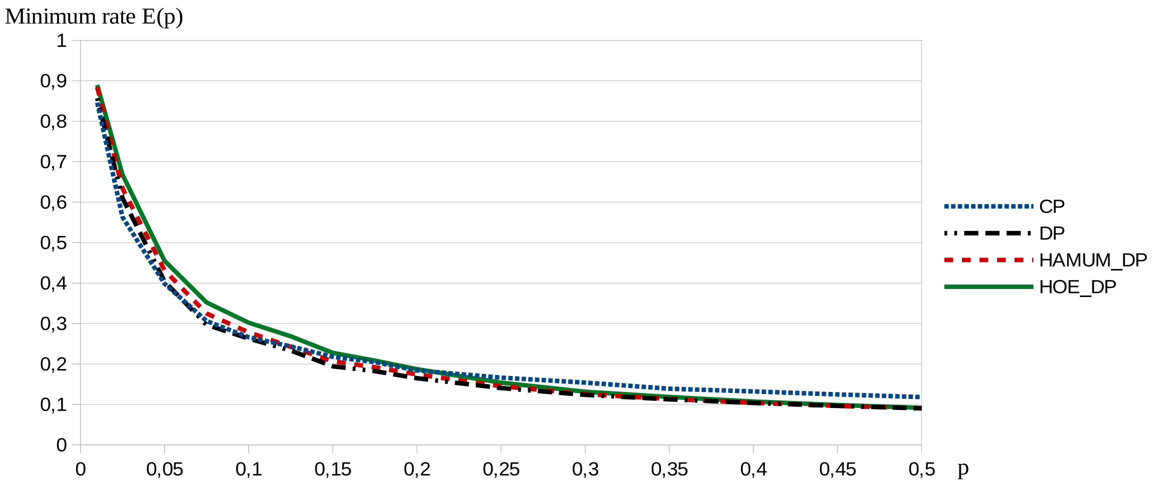


Figure 8.8: Comparison of the HOE DP, HAMUM DP with the CP and DP algorithms for a uniform traffic on a 4x4 grid. X-axis is the parameter of the uniform traffic and y-axis is the minimum rate for the flows.

The CP routing algorithm globally outperforms the other routing methods in term of minimal rate when the network usage is high ($p > 0.20$). The main reason is the reduction of the route length due to the increase of the number of partitions.

For small network usages ($p < 0.20$), HOE DP performs better since it takes advantage of the path diversity. Nevertheless, the benefit of path diversity is limited as soon as the network usage increases.

As expected both HAMUM DP and HOE DP always provide equal or better results than DP since they take advantage of path diversity.

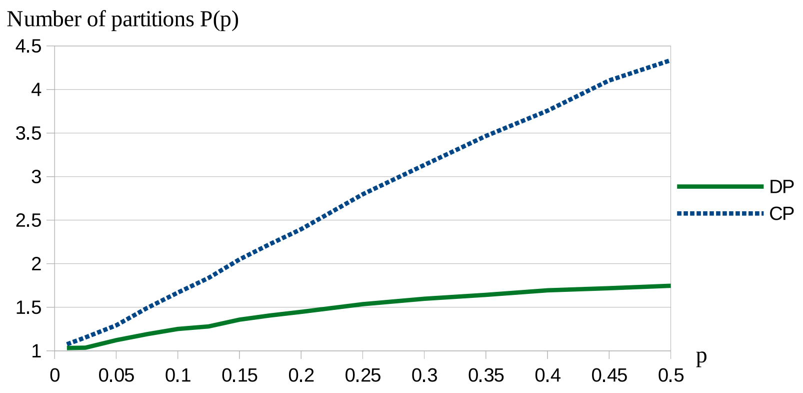


Figure 8.9: Comparison of the number of partitions for DP and CP partitioning methods. X-axis is the average number of partitions required for each multicast.

The other criterion of choice between CP and HOE DP is the number of partitions. Figure 8.9 shows the average number of unicast routes required by DP and CP partitioning computed as follows:

$$P(p) = \frac{1}{N} \sum_{i \in [1,N]} \left(\frac{1}{|\text{flows}_i|} \sum_{f \in \text{flows}_i} |\text{partitions}(f)| \right)$$

CP does not reach its maximum of 8 partitions for a 4x4 grid. Nevertheless, even for small network usage, it exceeds 2 partitions. As expected, DP stays under 2 partitions. Furthermore, CP provides a better minimal rate than HOE DP for $p < 0.20$. At $p = 0.20$, CP relies on more than 2.5 routes per flow in average which is more than DP which relies on 1.5 routes.

The multicast and unicast algorithms come in pairs since they must be compatible. For instance, the multicast CP algorithm is based on XY, hence it can be mixed with XY unicast traffic only. The HOE DP can be mixed with HOE unicast traffic only. As a matter of fact, in these cases the multicast algorithms give the same routing as the unicast algorithms if the “multicast” traffic with one destination is given.

The MPPA2 has a limited number of TX engines and the configuration of a TX engine takes time. Consequently, we recommend the HOE DP for multicast routing since it requires at most two TX engines per multicast. As a consequence, the HOE has to be chosen for the unicast routing also.

8.2 Comparison of the DNC Formulations

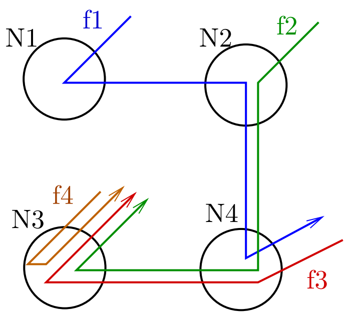
In this section, we perform a blackbox comparison of three methods to bound end-to-end latency of the wormhole NoC of the Kalray MPPA2. The first is the Linear Formulation method presented in Chapter 7 and implemented in our tool, the second is a DNC method originally designed for the Avionics Full-Duplex Switched Ethernet (AFDX) networks [21, 55, 22, 23] called Local formulation. The third from Ayed *et al.* [5] relies on the recursive calculus (RC) theory and has been designed to handle both the MPPA2 and the Tilera TILE64 many-core processor. They are compared to a naive formulation from [44] which was designed for the MPPA1 and is over-pessimistic when applied on the MPPA2.

The recursive calculus is an adaptation of the DNC. In introduces the Wormhole Sections which are network elements implementing the wormhole routers. The delays are expressed recursively as the sum of the delays of the interfering flows.

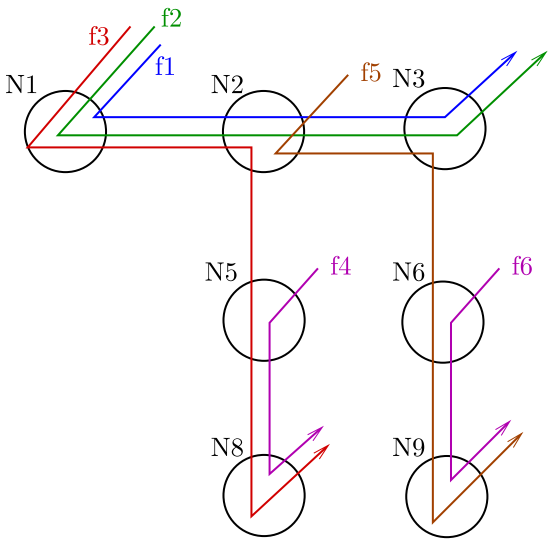
The main difference between the Linear and the Local formulation is the flows’ contract. In the Linear formulation, the flows are defined with a leaky bucket arrival curve. In AFDX, flows are defined with a maximal frame size and a minimal inter-frame time defining sporadic flows. Consequently, in Local formulation, the flows are inspired by AFDX and modeled as a staircase function while in Linear formulation, the staircase function is approximated with an affine function.

Another fundamental difference is that the Local formulation computes the latency of the flow in each queue while the Linear formulation is based on the Separated Flow Analysis (SFA) in which the burstiness is only accounted once. This principle is called *pay burst only once* (PBOO) [21].

The comparison of the methods is performed on the two use cases presented in Figure 8.10. We consider that packets are 17-flit long for use case 1. This corresponds to a packet with 1 flit of header and 16 flits (64B) of payload which often happens. The rates attributed to the flows with a max-min fair policy are shown in Table 8.1. For use case 2, the packets are 50-flit long and have a period of 1000 cycles [5]. This period is only taken into account for the Local method. For the Linear formulation, the period is approximated with a rate of 0.05 flits/cycle.



(a) Use case 1



(b) Use case 2: from [5]

Figure 8.10: Use cases of NoC traffic.

	f_1	f_2	f_3	f_4
Rate (flit/cycle)	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
Linear formulation (cycle)	25	110	100	34
Local formulation [21] (cycle)	25	170	136	34

Table 8.1: Bounds on NoC delay per flow in cycles for use case 1.

	f_1	f_2	f_3	f_4	f_5	f_6
Rate (flit/cycle)	5%	5%	5%	5%	5%	5%
NC, Naive form [44] (cycle)	4104	4104	5155	3153	5255	3153
Recursive Calculus [5] (cycle)	158	158	107	103	156	103
NC, Local formulation [21] (cycle)	57	57	54	62	153	58
NC, Linear formulation (cycle)	105	105	52	52	150	100

Table 8.2: Bounds on NoC delay per flow, in cycles for use case 2.

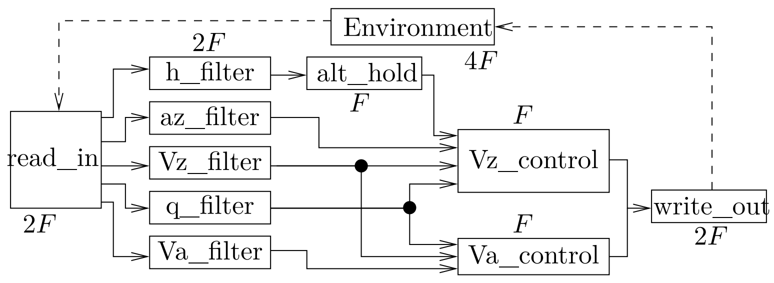


Figure 8.11: ROSACE Controller: Lustre implementation

Table 8.1 shows the delays for the flow of use case 1. The Linear formulation gives better results than the Local formulation. The reason is that Local formulation relies on a more pessimistic equation to compute the burstiness increase of a flow due to a FIFO. The Linear formulation relies on Equation 7.7 (page 101) which cannot be used by the Local formulation since it requires linearization of the arrival curve [21].

Table 8.2 shows the delays for the flows of use case 2. The Local formulation provides better delays.

Queue level. We check for both examples the queue maximum usage using our Linear formulation method. The limit is fixed to $Q_{max} = 517$ flits for the MPPA2.

We name Q^k_{usage}, usage of the hardware queue of the Turn k. For use case 1, the usage of the queue of the turn are: 12 flits for the Turn through node 3 from Local to Local (T3LL); 18 flits for T1LE, T4LW, T3NL and T2WS; and 24 flits for T4NW and T2LS. The maximum usage is 41 flits for queue T3EL. This set of flows can be implemented on the MPPA2 since the queues do not overflow.

For use case 2, the Q^k_{usage} are 51 flits for queues T5LS, T5NS, T6LS and T2WS; 53 flits for queues T2LE, T3WS and T6NS; 101 for 2WE, 3WL and T8NL. The highest queue usages are for T9NL with 103 flits and T1LE with 151 flits. In conclusion, this set of flows can be safely implemented on the MPPA2 since the queues do not overflow.

Conclusion. We compared the results of our DNC tool based on the Linear formulation with some other methods. We provide better bounds than earlier RC and Naive Form methods. Nevertheless, the Local formulation method is more mature and gives smaller delay bounds.

The main interest of our method is that it relies on a simple modeling offering low computation time. As a consequence, it allows on-line reconfiguration of the NoC.

8.3 Case Studies

This section presents applications of our framework on three use cases: the ROSACE case study, a sensor processing application and an application running on 64 cores. For them, the code is generated for the Kalray MPPA2. We compare the event-triggered method where each task starts as soon as possible, to the time-triggered method for which each task has a periodic release date. The event-triggered method is best effort and does not provide any timing guarantee while the time-triggered method offers guaranties on the WCRT. As the WCET analysis tools are not mature enough on this platform, the maximum duration of tasks is obtained by measurement.

8.3.1 ROSACE: Flight-Control Use-case

ROSACE [102] is a case-study whose structure is inspired by true avionic control applications. Only altitude is controlled, hence it does not require heavy computation. In order to experiment the influence

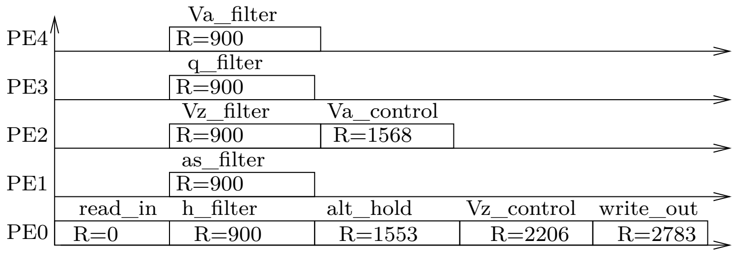


Figure 8.12: Static schedule the pure ROSACE controller tasks on 5 cores. First period is represented. For each task, the release is provided.

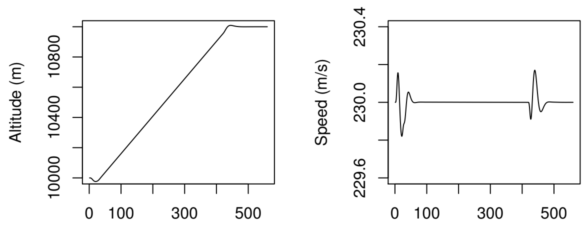


Figure 8.13: 600 periods of ROSACE running on the Kalray MPPA2. Step climb of 1000 m. Constant airspeed of 230 m/s.

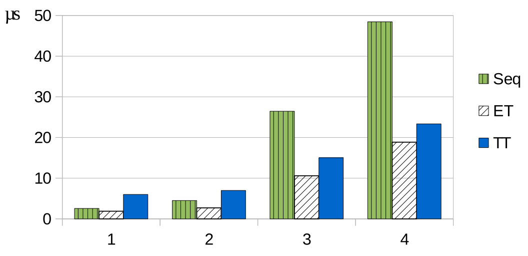


Figure 8.14: ROSACE: Communication only, pure ROSACE, ROSACE+100 cycles and ROSACE+200 cycles in each node are compared for Sequential, Event-Triggered and Time-Triggered methods.

Exp.	Seq/ET	Seq/TT	TT/ET	TT-ET
No Compute	1.36	0.43	3.18	4.11 μs
ROSACE	1.66	0.64	2.58	4.27 μs
ROSACE + 100	2.49	1.76	1.42	4.43 μs
ROSACE + 200	2.57	2.07	1.24	4.05 μs

Table 8.3: ROSACE: Comparison of event-triggered (ET), time-triggered (TT) and sequential version for each level of computation.

of computation load on the execution time, we created 4 versions of the application with different computation loads in each node of the controller. The "pure-ROSACE" version, the "ROSACE+100" where 100 cycles are added in each node, the "ROSACE+200" where 200 cycles are added in each node. In addition, to benchmark the communication time, we created the "No Compute" version where all the computations are removed from the controller's nodes.

Figure 8.11 shows the structure of the ROSACE application as presented in [102]. It is composed of a simulator of the environment and a controller. The purpose of the environment is to test the controller part, but only the controller part has real-time properties. We choose to execute the environment on an I/O Cluster and the controller on a Compute Cluster.

We look for the shortest period T such that the controller is schedulable:

– In Event-Triggered (ET), we experimentally measure the longest period while running the application on the Kalray MPPA2 chip.
– In Time-Triggered (TT) mode, we run the application on the MPPA2 and check that we do not get any timing violation. By construction, the period is known at compile-time.

The Time-Triggered plus Event Triggered mode has the same period as the TT mode, plus a few cycles: it only adds a polling loop that iterates only once if there is no timing violation. The difference with TT is too small to be relevant in measurements. In these experiments, we use the TT mode which has the advantage of letting us check that no timing violation occurs. Measurements are done on a Kalray MPPA2 running at 400 MHz.

In all cases, we compute the frequency as $F = 1/T$. Figure 8.11 shows the relative execution frequencies of the nodes. Figure 8.12 gives a schedule on 5 cores. The frequency F for pure ROSACE is 64 kHz. The release date (R) of each task is given. The first period (and odd periods) is the critical path since it contains all the tasks on F and $2F$. Figure 8.14 gives the first period duration for each experiment.

Table 8.3 shows that the speedup of the Event-Triggered (ET) is always better than sequential and Time-Triggered (TT) but ET does not provide any guarantee of WCET. The difference between TT and ET is due to the over-approximation of the WCET of each task compared to the actual execution and due to the WCET of the communication code. As expected, benchmarks with very lightweight tasks ("no compute" and "ROSACE") have an important synchronization overhead (TT is slower than Seq in these cases), but as soon as the tasks perform non-negligible computations, the synchronization overhead is compensated by the speedup due to parallelism.

8.3.2 Sensors Processing Case Study

We now consider the Lustre implementation of an avionic sensor processing program presented in [93]. This application reads a matrix of $8 * 512$ floats from the input/output cluster of the Kalray MPPA2, transposes it and dispatches the flows to 8 FFTs blocks. Results are gathered in a single matrix. In the pipelined version (see Figure 8.15), all the nodes are concurrently executed on 10 cores. In the non-pipelined version the FFT are executed in parallel on 8 cores. Figure 8.15b shows the schedule of the pipelined version.

Figure 8.16 compares TT and ET for the non-pipelined version. In the TT version, the WCET approximation and the MIA analysis leads to a nearly constant overhead of 21%. Nevertheless, the

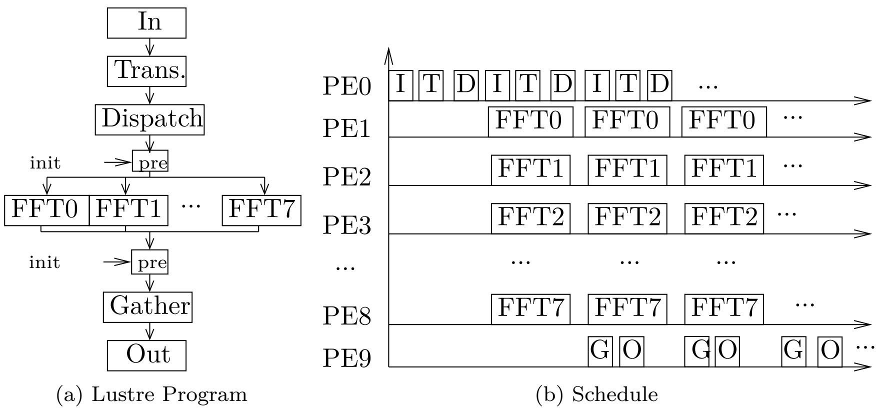


Figure 8.15: Processing on 8 sensors pipelined on 3 stages.

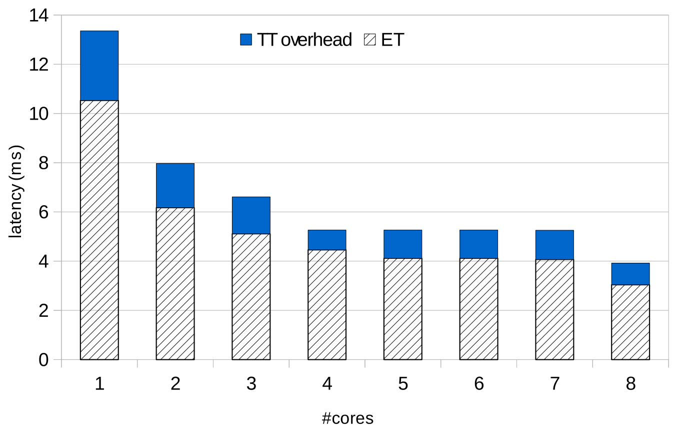


Figure 8.16: Non-pipelined sensor processing: Latency on 1 to 8 cores for TT and ET versions.

		1 core	8 cores	10 cores pipelined
ET	Throughput	389K	1348K	3242K
	Speedup	1x	3.46x	8.33x
	Latency	10.5ms	3ms	3.8ms
TT	Throughput	306K	1045K	2712K
	Speedup	1x	3.40x	8.86x
	Latency	13.3ms	3.9ms	4.53ms

Table 8.4: Sensors processing: pipelined *vs.* parallel version. Throughputs are in Kilo-float per second. Latency pipelined version is for three stages of pipeline.

duration of the TT version is an upper bound whereas ET is an average duration and does not offer any guarantee.

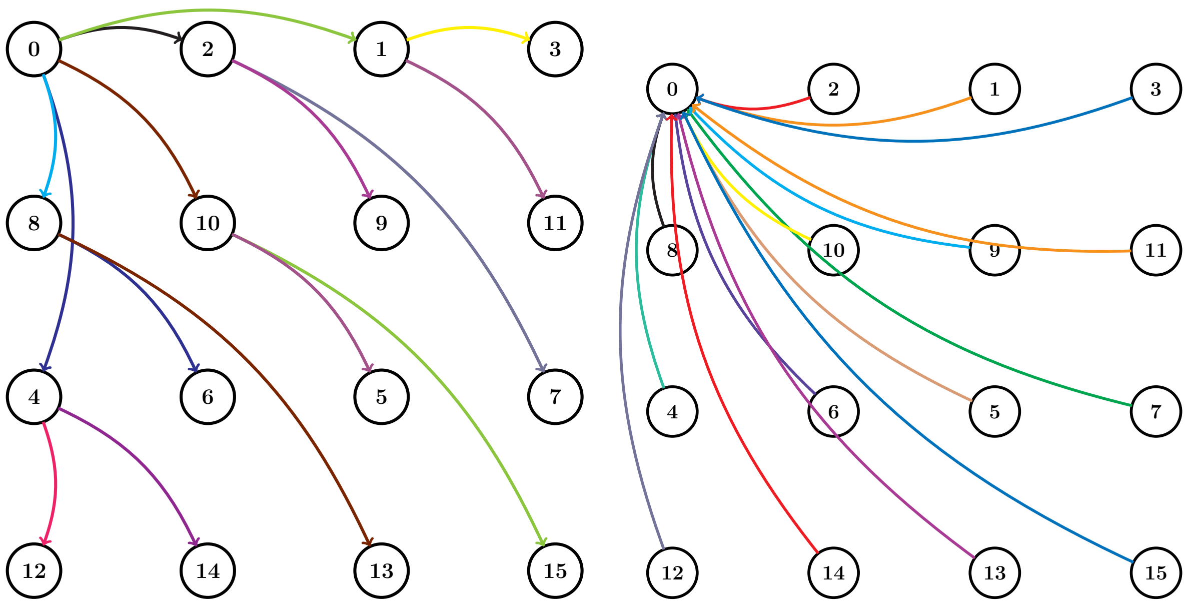
Between 4 to 7 cores, one core necessarily computes 2 FFTs in sequence, hence this explains the small difference. The small speedup increase between 4 and 7 is due to the instruction cache effect because each core has less code to execute.

Table 8.4 compares the pipelined and the parallel version. The pipelined version provides a better throughput than the non-pipelined version on 8 cores, but its latency is higher although better than the sequential version.

8.3.3 Synthetic Benchmark

The previously presented application are too small to be executed on several clusters. Consequently, we developed a use-case adapted to the highly parallel MPPA2. This application runs on 16 clusters and in each clusters, the program executes 4 tasks in parallel on 4 cores.

This synthetic benchmark allows demonstrating the automated code generation method on several clusters. It allows experimenting the NoC routing, configuration and worst-case traversal time (WCTT) computation.



(a) Dispatch from Cluster 0 (b) Gather to Cluster 0

Figure 8.17: Overview of the synthetic benchmark involving 16 clusters.

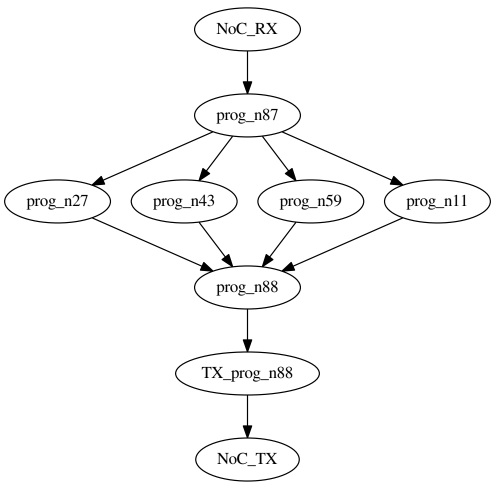


Figure 8.18: In each cluster, 4 cores execute tasks in parallel.

Figure 8.17 shows NoC communications involved in the application. Since the MPPA2 has a limited number of RX and TX engines in each cluster and since we statically allocate each of them to a specific communication, cluster 0 cannot dispatch the inputs to all the other clusters. Consequently, we chose to carry the inputs to each cluster using a tree structure as represented in Figure 8.17a. Cluster 0 dispatches the inputs to clusters 1, 2, 4, 8 and 10. Then, each of these clusters, in turn, dispatches to two other clusters. The outputs computed by each clusters are directly sent to cluster 0 as represented in Figure 8.17b.

As shown in Figure 8.18, in each cluster a dispatch task (`prog_n87`) receives the inputs from the NoC. The results are centralized by a gather task and sent through the NoC by a specific task (`TX_prog_n88`). For instance, Tasks `prog_27`, `prog_43`, `prog_59` and `prog_11` are computation tasks.

NoC Configuration. The flows are routed with HOE as recommended in Section 8.1.2. The obtained minimal rates are $\frac{1}{5}$ flit/cycle for the flows of Figure 8.17a and $\frac{1}{15}$ flit/cycle for the flows of Figure 8.17b flits/cycle. The bandwidth limiters are configured accordingly.

The packets are 17-flit long. In average the WCTT are 98 cycles for the flows of Figure 8.17a and 1227 cycles for the flows of Figure 8.17b. Since the packet size is the same, the difference is explained by the high usage of the ingress local link of cluster 0.

Release Date and Interference Analysis. The multi-core interference analysis (MIA) tool allows computing the release date of each task considering the memory interference. We explain how to model the NoC transfer and the WCTT in MIA. In the model, we create tasks to mimic the NoC latency. To avoid problems, we allow only one task per cluster to send data through the NoC.

As presented in Section 4.3, in the MPPA2, the NoC RX engines have priority on the bank arbiter. Consequently, their accesses do not have the same consequence on the concurrent access delay as the accesses from cores. To model this, MIA allows representing RX tasks which are special tasks accessing banks using this high priority.

Figure 8.19 shows an example of NoC communications modeled using MIA for release date computation. The example is composed of two tasks T1 and T2 running respectively on clusters 0 and 1. They are isolated and they do not suffer from interference. Clusters 2 executes tasks T3, T4

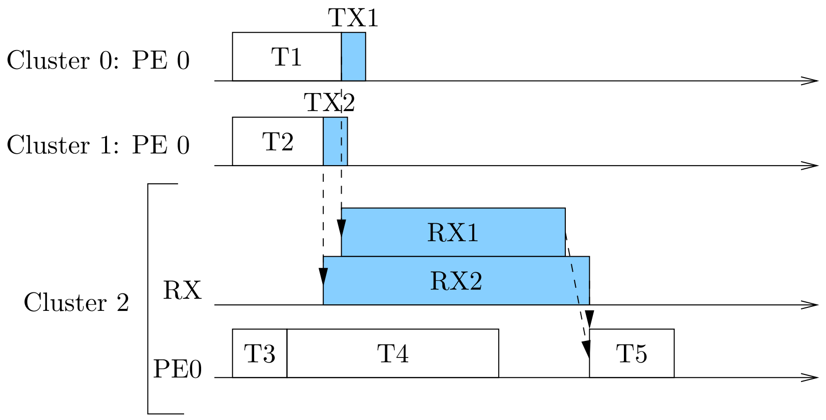


Figure 8.19: Representation of NoC communications in the MIA release date computation tool. The dashed arrows are the dependencies. White tasks are the tasks from the application while blue ones are tasks created to represent the NoC transmission in MIA.

and T5. T5 is waiting for packets coming from T1 and T2. All these tasks are represented in white and implement the application functionality.

Tasks in blue are created to encode the NoC behavior. For instance, in the communication from T1 to T5, tasks TX1 and RX1 are added. TX1 corresponds to the transmission initiation. RX1 performs writes to the bank of Cluster 2, PE0. Consequently, since tasks T4 and RX1 are concurrent, they interfere together. Similarly, TX2 and RX2 are added to represent the NoC transfer from T2 to T4. Note that the interference between RX1 and RX2 is accounted in the WCTT analysis.

To understand the need for the TX1 and TX2 tasks, we recall that the WCTT and the WCET do not provide any information about the best case. Consequently, if the NoC transfer is initiated immediately after the end of T1, the packet can possibly be send right after the beginning of T1 in case T1 terminates early, then interfering with T3.

The solution is to represent the tasks in two phases [113] to separate the computation and the shared-memory communications T1 of the NoC transfer initiation TX1. MIA then computes a release date for the second phase which corresponds to the end of T1. Hence, the NoC transfer starts somewhere between the beginning and the end of TX1. Consequently, the duration of RX1 is the WCET of the NoC transfer initiation plus the duration of RX1: WCTT + WCET(TX1). The same applies for RX2 = WCTT + WCET(TX2).

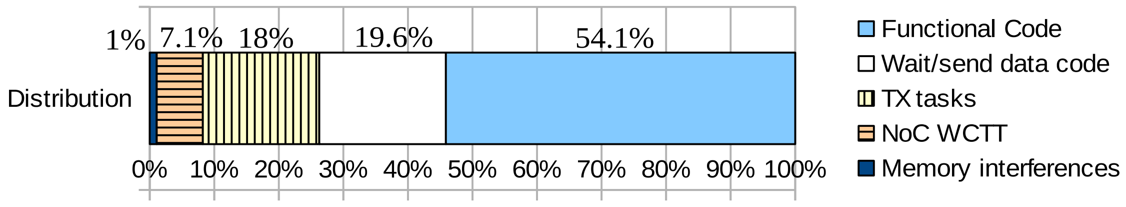


Figure 8.20: Time distribution of the critical path (WCRT) in percentage of the time-triggered (TT) execution.

Time Distribution. In order to understand the main bottleneck of our parallel implementation, we computed the fraction of the critical path spent in communication and computation. Figure 8.20 shows the fraction of the WCRT attributed to each work. The functional code represents 54.1% of the WCRT which means that less than half of the execution time is attributed to the communications.

More than 37.6% of the WCRT is attributed to the `send` and `wait` procedures. These procedures are called before and after the functional code in the task to ensure the presence of the inputs and write the outputs to the requiring tasks. The large fraction of the WCRT attributed to these procedures is due to the copy of the data and the access to the NoC TX engine. In the program, the transferred data are 20B each. More precisely, 19.6% of the WCRT is attributed to the `send` and `wait` procedures relative to the SMEM communications and 18% are attributed to the NoC TX tasks responsible for accessing the NoC TX engine.

The impact of NoC transfers is 7.1% of the WCRT. This number corresponds to the latency between the NoC TX engine of the emitter and NoC RX engine of the receiver. The WCETs of the transfer initiation through the TX engine are accounted in the TX tasks.

Finally, memory interferences are responsible for 1% of the WCRT. This low fraction has to be put into perspective with the low number of cores used in each cluster (4). It also shows the benefits of our execution model.

To evaluate the speedup of the implementation we define a theoretical speedup. It is obtained by the Amdahl law taking into account that some tasks are sequential, some are parallelizable on 16 cores, some others on 64 cores. This ideal speedup is 26.8.

	Execution time	Speedup
Sequential	212780 cycles	x1
TT	22264 cycles	x9.5
ET	11741 cycles	x18.12
Ideal	7939 cycles	x26.8

Table 8.5: Execution time and resulting speedup obtained for each method.

Table 8.5 shows the execution time of the application using the three methods compared to the ideal speedup. The ET methods provides a good speedup of 18. We recall that the time-triggered method (TT) is the only one with a guaranteed WCRT. The TT execution is 9.5 faster than the sequential execution. The TT is 47.3% slower than ET due the worst-case estimations.

8.4 Conclusion

In this chapter, we evaluated the two route attribution algorithms and showed that our LP-based heuristics performs similarly to the enumeration method while reducing drastically the complexity of the algorithm compared to the exploration method. We evaluated our network calculus tool computation.

We showed that our multicast routing algorithm leads to an efficient max-min fair attribution while minimizing the number of partitions and thus the number of TX engines required for the transmission.

Finally, we applied our complete code generation toolchain on three realistic use-cases: two from avionics and one designed for its heavy computing demand. We showed a great latency reduction of the parallel version compared to the sequential version and a minimal overhead due to the WCRT guaranties.

9 Conclusion and Future Work

9.1 Summary and Contributions

In this thesis, we presented a complete toolchain allowing parallel code generation from a data-flow synchronous program on a multi- or many-core processor. This work targets the code generation for critical, hard real-time systems. These systems require functional correctness. In particular, the code generation should enable software validation. Thus, the solution must be based on simple methods, simple hardware mechanisms and it must enable traceability of the generated code. These systems should be time-predictable to fit real-time constraints such as minimal bandwidth or maximal latency.

The purpose of our work is to give a complete implementation method of real-time software on a multi- and many-core. We apply this method to the Kalray MPPA2.

9.1.1 Task Extraction and Parallel Intermediate Representation (PIR)

The first step is to identify and extract parallelism from the data-flow Synchronous program. Extracting as much parallelism as possible from a data-flow application is not efficient since the cost of the communication becomes higher than the execution time of the tasks. Consequently, we develop the Top-Level Node method which automatically selects all the sub-nodes of the top-level node of the program as candidate for parallel execution. We compare our method with the KCG multi-core Scade compiler offering a fork-join-like parallel annotation. Each selected node and its sub-nodes are compiled into a single functional code. These methods ensure code traceability.

We introduce a Parallel Intermediate Representation (PIR) to describe the direct and delayed communications, and the dependencies between the tasks. Since the problem of mapping the tasks on the core is already subject to an active research, we rely on an external tool for this purpose. We focus our work on the generation of the code required to deploy the tasks on the hardware and make them communicate. Consequently, the second step of our toolchain is the automatic implementation of the PIR on the hardware. Each task is composed of the functional code, the communication and the synchronization codes.

9.1.2 Latency-Bounded Communication Under Interference

The main difficulty with the multi- and many-core architectures compared to the single-cores is the existence of shared-resources leading to timing interferences. Furthermore, these shared-resources interferences have to be taken into account when generating the code.

Shared-Memory. We have chosen an approach where the memory interferences are not avoided. Our method minimizes them using data placement in banked memory. We explain that a time-triggered execution of the tasks is a solution to minimize memory interferences and to guarantee a bound on the Worst-Case Response Time (WCRT).

Hamza Rihani developed the MIA [113] tool which computes a release date for each task taking into account the dependencies and the memory interferences. Together with him, we developed a method to adapt MIA to our code generation tool and a time-triggered execution model.

Network-on-Chip. The Network-on-Chip (NoC) offers low latency communications on the chip. A NoC is composed of shared links and routers. Nevertheless, some NoCs offer some guarantees of service thanks to hardware bandwidth limiters. Our contribution is a complete tool which computes a NoC configuration ensuring tight end-to-end latency bounds of the communications.

Routers of the MPPA2 NoC follow the wormhole policy. This offers low latencies but is deadlock-prone. Consequently, we compare several deadlock-free routing algorithms and show how to adapt them to the MPPA2. We also introduce the deadlock-free HOE Dual-Path algorithm offering efficient multi-cast communications while minimizing the number of packets required to implement one transmission.

Some routing algorithms offer path-diversity meaning that for each pair of communicating nodes, several paths are possible. We develop a method to exploit these different paths by selecting a combination of static routes offering a fair rate attribution. In particular, we design an enumerative method called Exploration for Unique Route Selection (EURS) and a LP-based heuristic called LP-Based Heuristics for Unique Route Selection (LPURS) to efficiently compute the combination offering the best fair attribution. We then presented a tool relying on the Deterministic Network Calculus theory to compute the end-to-end latency of the communications.

9.1.3 Results

We compare the multi-cast routing algorithm to classical routing algorithm and show good results while managing with the hardware constraints. We also evaluate the EURS and the LP-based heuristic LPURS and shown that LPURS performs considerably faster while keeping accurate results.

We apply our toolchain to the ROSACE [102] avionic case study which implements an academic flight control program regulating speed and altitude. We generate parallel and time-triggered code which guarantees the WCRT. We then compare the guaranteed execution time to a best effort execution time on the platform showing a small overhead of 21% for our method.

We apply our toolchain on a sensor processing application inspired from a true avionic industry [93]. The result is a 22% overhead between our method and a best effort execution time. We have shown a version parallelized on 1 to 8 cores and a pipelined version on 10 cores.

Finally, we present an application running on 16 clusters and 64 cores. The WCRT of this application is guaranteed thanks to NoC end-to-end latency analysis and memory interference computation. Furthermore, we provide a method to compute safe release dates for the tasks including both NoC and shared-memory communications. This application shows a good speedup with overhead of 47% compared to the best effort.

9.2 Future Work

Code generation of real-time applications on a multi-core is a wide topic and offers many opportunities for improvement.

One lesson learned from the implementation of applications using our method is that the main bottleneck is the memory. We shown with ROSACE for instance, that our computation time was beyond the requirements, nevertheless, the size of the MPPA2 shared-memory was a bottleneck for the implementation of the signal processing application.

This memory limitation is first due to the hardware characteristics, for instance the size of the shared-memory memory, but also to the data-flow paradigm which requires a lot of copies. Since the implementation is static and time-triggered, the usage of each buffer can be know, then a better buffer allocation can be invented to minimize memory usage.

Another reason for this memory shortage is the use of the processor in banked mode which requires strict memory alignments. We think that these alignments could be made less strict while preserving low WCRT. This requires modifications in both the release dates computation and the code generation.

Another line of inquiry is overlaying. It consists in storing only a partial code and data in the memory. Then, data and code are transferred when needed. The phase of transmission and the phase of execution have to be pipelined to minimize the execution time. Furthermore, the order of the phases and the amount of transmitted data has to be know at compile time to ensure predictability.

Finally, we fully implement our toolchain for the Kalray MPPA2 but we are convinced that the general concepts of this method apply on a large range of hardware. As a future work, we consider the implementation for the T-CREST platform, PULP or Infineon AURIX.

9.3 Future of the Multi-Core for Critical Applications

Multi-core for real-time and critical systems will become increasingly important within the next few years. But there is still a long way to go before we reach the same level of safety as single-core architectures.

High-performance hardware is too complex to be analyzed and prone to timing anomalies. Nevertheless, some multi-cores are specifically designed to be time-predictable. Their main common characteristic is the presence of a scratchpad memory private to each core. Apart from that, the characteristics such as the presence of cache, their policy, the organization in clusters, the availability of a global address space, the presence of bus or NoC do not reach a consensus. As a matter of fact, the certification problems caused by the multi-cores are well listed in the CAST-32A [48] document but there is no norm and standard to define real-time multi-cores. Consequently, a complete analysis is required for each of them to identify sources of interference and the timing behavior of the cores. In particular a specific execution model has to be defined for each multi-core architecture taking into account for instance the different memory levels.

From two cores to some hundreds, multi- and many-cores offer a lot of computation power. Obviously, one benefit of using many-core architecture is the extra computing power they provide, but another important benefit is to allow integrating more applications in the same processor. The integration of several applications is similar to the integrated modular avionics (IMA).

Nevertheless, if we integrate several applications in the same processor, these applications are isolated spatially rather than temporally as in IMA. Then, the input/output ports and the external memory can be simultaneously accessed and shared among all the applications. This creates a new bottleneck and a new source of interferences. In particular, the hardware has to handle the multiplexing between the applications in a generic way to support the diversity of external devices that can be connected to the processor. Finally, bounded communication latencies have to be guaranteed by the software and hardware architecture. Perret *et al.* [106] defines an execution model to implement IMA on the MPPA2.

Finally, there is still a lot of work and a lot of research to do before usage of multi-/many-core for critical applications becomes reality. Despite that, we provided a toolchain to solve some of the main problems due to the execution of critical programs on a multi-core. Our toolchain provides an important base of work for future research on real-time code generation for a multi-core. This thesis was also a great opportunity to join the academic world with the industrial world since as part of the CAPACITES project, we provided expertise on the Kalray MPPA2 architecture to the laboratories.

Bibliography

[1] Abbaspour, S., Brandner, F., and Schoeberl, M. (2013). A time-predictable stack cache. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–8. IEEE.

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

[3] Alras, M., Caspi, P., Girault, A., and Raymond, P. (2009). Model-based design of embedded control systems by means of a synchronous intermediate model. In *Embedded Software and Systems, 2009. ICESS'09. International Conference on*, pages 3–10. IEEE.

[4] Amaldi, E., Coniglio, S., Gianoli, L. G., and Ileri, C. U. (2013). On single-path network routing subject to max-min fair flow allocation. *Electronic Notes in Discrete Mathematics*, 41:543–550.

[5] Ayed, H., Ermont, J., Scharbarg, J.-l., and Fraboul, C. (2016). Towards a unified approach for worst-case analysis of tilera-like and kalray-like noc architectures. In *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*, pages 1–4. IEEE.

[6] Bahn, J. and Bagherzadeh, N. (2009). A generic traffic model for on-chip interconnection networks, Netw. *Chip Arch*, 22.

[7] Bahrebar, P. and Stroobandt, D. (2015). The Hamiltonian-based odd–even turn model for maximally adaptive routing in 2D mesh networks-on-chip. *Computers & Electrical Engineering*, 45:386–401.

[8] Ballabriga, C., Cassé, H., Rochange, C., and Sainrat, P. (2010). OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS 2010*, pages 35–46.

[9] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. (2002). Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 73–78.

[10] Becker, M., Dasari, D., Nicolic, B., Åkesson, B., Nélis, V., and Nolte, T. (2016). Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24.

[11] Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., et al. (2008). Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598. IEEE.

[12] Benini, L. and De Micheli, G. (2002). Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 418–419. IEEE.

[13] Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282.

[14] Berg, C. (2006). Plru cache domino effects. In *OASIcs-OpenAccess Series in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[15] Berry, G. (2007). SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer.

[16] Berry, G. and Cosserat, L. (1984). The ESTEREL synchronous programming language and its mathematical semantics. In *International Conference on Concurrency*, pages 389–448. Springer.

[17] Bertsekas, D. P., Gallager, R. G., and Humblet, P. (1992). Data networks, vol. 2. *Prentice Hall International, Englewood Cliffs, New Jersey*, 7632:493–536.

[18] Bezerra, G. B., Forrest, S., Moses, M., Davis, A., and Zarkesh-Ha, P. (2010). Modeling NoC traffic locality and energy consumption with rent's communication probability distribution. In *Proceedings of the 12th ACM/IEEE international workshop on System level interconnect prediction*, pages 3–8. ACM.

[19] Boppana, R. V., Chalasani, S., and Raghavendra, C. (1998). Resource deadlocks and performance of wormhole multicast routing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):535–549.

[20] Bouillard, A. and Stea, G. (2015). Worst-Case Analysis of Tandem Queueing Systems Using Network Calculus. *Quantitative Assessments of Distributed Systems: Methodologies and Techniques*, pages 129–173.

[21] Boyer, M., Dupont de Dinechin, B., Graillat, A., and Havet, L. (2018). Computing routes and delay bounds for the network-on-chip of the kalray mppa2 processor. In *ERTS 2018-9th European Congress on Embedded Real Time Software and Systems.*

[22] Boyer, M., Migge, J., and Navet, N. (2011). An efficient and simple class of functions to model arrival curve of packetised flows. In *Proceedings of the 1st International Workshop on Worst-Case Traversal Time*, pages 43–50. ACM.

[23] Boyer, M., Navet, N., and Fumey, M. (2012a). Experimental assessment of timing verification techniques for afdx. In *6th European Congress on Embedded Real Time Software and Systems.*

[24] Boyer, M., Stea, G., and Sofack, W. M. (2012b). Deficit round robin with network calculus. In *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on*, pages 138–147. IEEE.

[25] Breaban, G., Stuijk, S., and Goossens, K. (2017). Efficient synchronization methods for let-based applications on a multi-processor system on chip. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1721–1726. IEEE.

[26] Carle, T., Djemal, M., Potop-Butucaru, D., De Simone, R., and Zhang, Z. (2014). Static mapping of real-time applications onto massively parallel processor arrays. In *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*, pages 112–121. IEEE.

[27] Carle, T., Potop-Butucaru, D., Sorel, Y., and Lesens, D. (2015). From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation. *Leibniz Transactions on Embedded Systems(LITES)*, 2(2):01–1.

[28] Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. (2003). From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, pages 153–162. ACM.

[29] Caspi, P., Girault, A., and Pilaud, D. (1999). Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE T Software Engineering*, 25(3):416–427.

[30] Caspi, P., Scaife, N., Sofronis, C., and Tripakis, S. (2008). Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):15.

[31] Champion, A., Mebsout, A., Sticksel, C., and Tinelli, C. (2016). The Kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer.

[32] Charette, R. N. (2009). This car runs on code. *IEEE spectrum*, 46(3):3.

[33] Chen, S. and Nahrstedt, K. (1998). Maxmin fair routing in connection-oriented networks. In *Proc. Euro-Parallel and Distributed Systems Conf*, pages 163–168.

[34] Chiu, G.-M. (2000). The odd-even turn model for adaptive routing. *IEEE Transactions on parallel and distributed systems*, 11(7):729–738.

[35] Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., and Pouzet, M. (2006). N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. *ACM SIGPLAN Notices*, 41(1):180–193.

[36] Cohen, A., Gérard, L., and Pouzet, M. (2012). Programming Parallelism with Futures in Lustre. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT, pages 197–206, New York, NY, USA. ACM. Un restriction est mise: les future doivent rester local au noeud et ne peuvent pas être passés en paramètre. Cela évite d'avoir recours à un GC.

[37] Conti, F., Rossi, D., Pullini, A., Loi, I., and Benini, L. (2014). Energy-efficient vision on the PULP platform for ultra-low power parallel computing. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE.

[38] Cordovilla, M., Boniol, F., Forget, J., Noulard, E., and Pagetti, C. (2011). Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In *RTNS'11.*

[39] Cruz, R. L. (1991). A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131.

[40] Dally, W. J. and Seitz, C. L. (1987). Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36(5):547–553.

[41] Dally, W. J. and Towles, B. P. (2004). Principles and practices of interconnection networks (the morgan kaufmann series in computer architecture and design).

[42] Daneshtalab, M., Ebrahimi, M., Xu, T. C., Liljeberg, P., and Tenhunen, H. (2011). A generic adaptive path-based routing method for MPSoCs. *Journal of Systems Architecture*, 57(1):109–120.

[43] Davis, R. I., Altmeyer, S., Indrusiak, L. S., Maiza, C., Nelis, V., and Reineke, J. (2017). An extensible framework for multicore response time analysis. *Real-Time Systems.*

[44] Dupont de Dinechin, B., Durand, Y., Van Amstel, D., and Ghiti, A. (2014). Guaranteed services of the noc of a manycore processor. In *Proceedings of the 2014 International Workshop on Network on Chip Architectures*, pages 11–16. ACM.

[45] Dupont de Dinechin, B. and Graillat, A. (2017a). Feed-forward routing for the wormhole switching network-on-chip of the kalray mppa2 processor. In *Proceedings of the 10th International Workshop on Network on Chip Architectures*, NoCArc'17, pages 10:1–10:6, New York, NY, USA. ACM.

[46] Dupont de Dinechin, B. and Graillat, A. (2017b). Network-on-chip service guarantees on the kalray mppa-256 bostan processor. In *Proceedings of the 2nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*, pages 35–40. ACM.

[47] Durrieu, G., Faugere, M., Girbal, S., Pérez, D. G., Pagetti, C., and Puffitsch, W. (2014). Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*.

[48] FAA, Certification Authorities Software Team (November 2016). Multi-core Processors, CAST-32A. *Position Paper.*

[49] Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[50] Ferdinand, C. (2004). Worst case execution time prediction by static program analysis. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 125. IEEE.

[51] Fidler, M. and Einhoff, G. (2004). Routing in Turn-Prohibition Based Feed-Forward Networks. In *NETWORKING 2004, Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communication, Third Int. IFIP-TC6 Networking Conference, Athens, Greece, May 9-14.*

[52] Firoiu, V., Boudec, J. Y. L., Towsley, D., and Zhang, Z.-L. (2002). Theories and models for internet quality of service. *Proc.of the IEEE*, 90(9):1565–1591.

[53] Fisher, J. A. (1983). *Very long instruction word architectures and the ELI-512.* ACM.

[54] Flanagan, C. and Felleisen, M. (1995). The semantics of future and its use in program optimization. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220. ACM.

[55] Frances, F., Fraboul, C., and Grieu, J. (2006). Using network calculus to optimize the afdx network.

[56] Geyer, F. and Carle, G. (2016). Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches. *IEEE Communications Magazine*, 54(2):106–112.

[57] Girault, A. (2005). A survey of automatic distribution method for synchronous programs. In *International workshop on synchronous languages, applications and programs, SLAP*, volume 5.

[58] Girault, A., Nicollin, X., and Pouzet, M. (2006). Automatic Rate Desynchronization of Embedded Reactive Programs. *TECS*, 5(3):687–717.

[59] Glass, C. J. and Ni, L. M. (1992). The turn model for adaptive routing. *ACM SIGARCH Computer Architecture News*, 20(2):278–287.

[60] Glass, C. J. and Ni, L. M. (1994). The turn model for adaptive routing. *Journal of the ACM (JACM)*, 41(5):874–902.

[61] Goossens, K., Koedam, M., Nelson, A., Sinha, S., Goossens, S., Li, Y., Breaban, G., van Kampenhout, R., Tavakoli, R., Valencia, J., et al. (2017). Noc-based multiprocessor architecture for mixed-time-criticality applications. *Handbook of Hardware/Software Codesign*, pages 491–530.

[62] Gorcitz, R., Kofman, E., Carle, T., Potop-Butucaru, D., and De Simone, R. (2015). On the scalability of constraint solving for static/off-line real-time scheduling. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 108–123. Springer.

[63] Graillat, A., Moy, M., Raymond, P., and Dupont De Dinechin, B. (2018). Parallel Code Generation of Synchronous Programs for a Many-core Architecture. In *DATE 2018 - Design, Automation and Test in Europe*, Dresden, Germany.

[64] Graillat, A., Rihani, H., Maiza, C., Moy, M., Raymond, P., and Dupont de Dinechin, B. Implementation Framework for Real-Time Data-Flow Synchronous Programs on Many-Cores. *Real-Time Systems*. Submitted for review.

[65] Grieu, J. (2004). *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis.

[66] Ha, S. and Lee, E. A. (1991). Quasi-static scheduling for multiprocessor dsp. In *1991., IEEE International Sympoisum on Circuits and Systems*, pages 352–355 vol.1.

[67] Halbwachs, N. (1993). *Synchronous programming of reactive systems*. Kluwer Academic Pub.

[68] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320.

[69] Hamann, A., Dasari, D., Kramer, S., Pressler, M., and Wurst, F. (2017). Communication Centric Design in Complex Automotive Embedded Systems. In *LIPIcs*, volume 76. Schloss Dagstuhl-LZI.

[70] Harel, D. and Pnueli, A. (1985). On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer.

[71] Haverkort, B. R. et al. (1998). *Performance of computer communication systems: a model-based approach*. Wiley Online Library.

[72] Henzinger, T., Horowitz, B., and Kirsch, C. (2001). Giotto: A time-triggered language for embedded programming. In *Embedded software*, pages 166–184. Springer.

[73] Henzinger, T. A., Kirsch, C. M., and Matic, S. (2003). Schedule-carrying code. In *International Workshop on Embedded Software*, pages 241–256. Springer.

[74] Henzinger, T. A., Kirsch, C. M., and Matic, S. (2005). Composable code generation for distributed Giotto. In *ACM SIGPLAN Notices*, pages 21–30. ACM.

[75] Holsmark, R., Palesi, M., and Kumar, S. (2006). Deadlock free routing algorithms for mesh topology NoC systems with regions. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, page 696–703. IEEE.

[76] Hugues, J. and Delange, J. (2015). Model-based design and automated validation of ARINC653 architectures. In *2015 International Symposium on Rapid System Prototyping (RSP)*, pages 3–9.

[77] Ishii, M., Detrey, J., Gaudry, P., Inomata, A., and Fujikawa, K. (2017). Fast modular arithmetic on the kalray mppa-256 processor for an energy-efficient implementation of ecm. *IEEE Transactions on Computers*.

[78] Jafari, F., Yaghmaee, M. H., Talebi, M. S., and Khonsari, A. (2008). Max-min-fair best effort flow control in network-on-chip architectures. In *International Conference on Computational Science*, pages 436–445. Springer.

[79] Kasapaki, E., Schoeberl, M., Sørensen, R. B., Müller, C., Goossens, K., and Sparsø, J. (2016). Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492.

[80] Kehr, S., Quiñones, E., Böddeker, B., and Schäfer, G. (2015). Parallel execution of AUTOSAR legacy applications on multicore ECUs with timed implicit communication. In *DAC'15*, page 42. ACM.

[81] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R. (2014). Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154.

[82] Kluge, F., Schoeberl, M., and Ungerer, T. (2016). Support for the logical execution time model on a time-predictable multicore processor. *ACM SIGBED Review*, 13(4):61–66.

[83] Kopetz, H. and Bauer, G. (2003). The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126.

[84] Kumar, S., Jantsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., and Hemani, A. (2002). A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 117–124. IEEE.

[85] Land, I. and Elliott, J. (2009). Architecting arinc 664, part 7 (afdx) solutions. *Xilinx, May.*

[86] Le Boudec, J.-Y. and Thiran, P. (2001). *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media.

[87] Lee, C.-G., Hahn, H., Seo, Y.-M., Min, S. L., Ha, R., Hong, S., Park, C. Y., Lee, M., and Kim, C. S. (1998). Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713.

[88] Lee, E. A. and Seshia, S. A. (2017). *Introduction to Embedded Systems - A Cyber-Physical Systems Approach.* MIT Press, second edition edition.

[89] Lenzini, L., Martorini, L., Mingozzi, E., and Stea, G. (2006). Tight end-to-end per-flow delay bounds in fifo multiplexing sink-tree networks. *Performance Evaluation*, 63(9-10):956–987.

[90] Levitin, L., Karpovsky, M., and Mustafa, M. (2009). Deadlock prevention by turn prohibition in interconnection networks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–7. IEEE.

[91] Lin, X., McKinley, P. K., and Ni, L. M. (1994). Deadlock-free multicast wormhole routing in 2-D mesh multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):793–804.

[92] Ljung, M. (1999). Formal modelling and automatic verification of Lustre programs using np-tools. *Master's thesis, Prover Technology AB and Department of Teleinformatics, KTH, Stockholm.*

[93] Lo, M., Valot, N., Maraninchi, F., and Raymond, P. (2016). Implementing a real-time avionic application on a many-core processor. In *42nd European Rotorcraft Forum (ERF)*, Lille, France.

[94] Lv, M., Yi, W., Guan, N., and Yu, G. (2010). Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349. IEEE Computer Society.

[95] Maraninchi, F. (1991). The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, volume 3.

[96] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. (2015). Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, pages 87–96.

[97] Muljadi, E. and Butterfield, C. P. (2001). Pitch-controlled variable-speed wind turbine generation. *IEEE transactions on Industry Applications*, 37(1):240–246.

[98] Nace, D. (2002). A Linear Programming Based Approach for Computing Optimal Fair Splittable Routing. In *Proc.of ISCC'02*, ISCC '02.

[99] Naumann, N. (2009). Autosar runtime environment and virtual function bus. *Hasso-Plattner-Institut, Tech. Rep*, page 38.

[100] Nguyen, V. A., Hardy, D., and Puaut, I. (2017). Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:22.

[101] Pagano, B., Pasteur, C., Siegel, G., and Knížek, R. (2018). A Model Based Safety Critical Flow for the AURIX Multi-core Platform. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*.

[102] Pagetti, C., Saussié, D., Gratia, R., Noulard, E., and Siron, P. (2014). The ROSACE case study: From Simulink specification to multi/many-core execution. In *IEEE RTAS'14*, pages 309–318.

[103] Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., and Kegley, R. (2011). A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279.

[104] Pellizzoni, R., Schranzhofer, A., Chen, J.-J., Caccamo, M., and Thiele, L. (2010). Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 741–746.

[105] Perret, Q., Maurere, P., Noulard, E., Pagetti, C., Sainrat, P., and Triquet, B. (2016a). Predictable composition of memory accesses on many-core processors. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

[106] Perret, Q., Maurere, P., Noulard, E., Pagetti, C., Sainrat, P., and Triquet, B. (2016b). Temporal isolation of hard real-time applications on many-core processors. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–11. IEEE.

[107] Plateau, F. (2010). *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Paris 11.

[108] Pree, W., Templ, J., Hintenaus, P., Naderlinger, A., and Pletzer, J. (2011). TDL-Steps Beyond Giotto: A Case for Automated Software Construction. *Int. J. Software and Informatics*, 5(1-2):335–354.

[109] Rahimi, A., Loi, I., Kakoee, M. R., and Benini, L. (2011). A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE.

[110] Raymond, P. (2006). Vérification de programmes synchrones avec Lustre/Lesar . In *Systèmes temps réel 1 – techniques de description et de vérification*, chapter 6. Hermes science – Lavoisier.

[111] Reineke, J., Liu, I., Patel, H. D., Kim, S., and Lee, E. A. (2011). PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*, pages 99–108. IEEE.

[112] Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., and Becker, B. (2006). A definition and classification of timing anomalies. In *OASIcs-OpenAccess Series in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[113] Rihani, H., Moy, M., Maiza, C., Davis, R. I., and Altmeyer, S. (2016). Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *RTNS'16*, pages 67–76. ACM.

[114] Saidi, S., Ernst, R., Uhrig, S., Theiling, H., and Dupont de Dinechin, B. (2015). The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press.

[115] Schmitt, J. B. and Zdarsky, F. A. (2006). The disco network calculator: a toolbox for worst case analysis. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 8. ACM.

[116] Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., et al. (2015). T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471.

[117] Schroeder, M. D., Birrell, A. D., Burrows, M., Murray, H., Needham, R. M., Rodeheffer, T. L., Satterthwaite, E. H., and Thacker, C. P. (1990). *Autonet: a high-speed, self-configuring local area network using point-to-point links.* Digital Equipment Corporation Systems Research Center.

[118] Shreedhar, M. and Varghese, G. (1995). Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review*, volume 25, pages 231–242. ACM.

[119] Skalistis, S., Angiolini, F., Simalatsar, A., and De Micheli, G. (2017). Safe and Efficient Deployment of Data-Parallelisable Applications on Many-Core Platforms: Theory and Practice. *IEEE Design & Test.*

[120] Skalistis, S. and Simalatsar, A. (2016). Worst-Case Execution Time Analysis for Many-Core Architectures with NoC. pages 211–227.

[121] Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. (2016). Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46.

[122] Starobinski, D., Karpovsky, M., and Zakrevski, L. (2002). Application of Network Calculus to General Topologies using Turn-Prohibition. In *IEEE INFOCOM.*

[123] Stefan, R. A., Molnos, A., and Goossens, K. (2014). daelite: A tdm noc supporting qos, multicast, and fast connection set-up. *IEEE Transactions on Computers*, 63(3):583–594.

[124] Thiele, L., Chakraborty, S., and Naedele, M. (2000). Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE.

[125] Tripakis, S., Bui, D., Geilen, M., Rodiers, B., and Lee, E. A. (2013). Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):83.

[126] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36.

[127] Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. (2009). Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978.

[128] Yip, E., Girault, A., Roop, P., and Biglari-Abhari, M. (2016). The ForeC Synchronous Deterministic Parallel Programming Language for Multicores. In *MCSoC'16*, Lyon, France. IEEE.

[129] Ziccardi, M., Schoeberl, M., and Vardanega, T. (2015). A time-composable operating system for the Patmos processor. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1892–1897. ACM.

Most critical systems are subject to hard real-time requirements. These systems are more and more complex and the computational power of the predictable single-core processors is no longer sufficient. Multi- or many-core architectures are good alternatives but interferences on shared resources must be taken into account to avoid unpredictable timing effects. For many-core, the Network-on-Chip (NoC) must be configured such that deadlocks are avoided and a tight Worst-Case Traversal Time (WCTT) of the communications can be computed. The Kalray MPPA2 is a many-core architecture with good timing properties.

Dataflow Synchronous languages such as Lustre or Scade are widely used for avionics critical software. In these languages, programs are described by networks of computational nodes. We introduce a method to extract parallel tasks from synchronous programs. Then, we generate parallel code to deploy tasks on the chip and implement NoC and shared-memory communications. The generated code enables traceability. It is based on a time-triggered execution model which relies on a static schedule and minimizes the memory interferences thanks to usage of memory banks. The code enables the computation of a worst-case execution time bound accounting for the memory interferences and the WCTT of NoC transmissions. We generate a configuration of the platform to enable a fair bandwidth attribution on the NoC, bounded transmissions through the NoC and clock synchronization. Finally, we apply this toolchain on avionic case studies and synthetic benchmarks running on 64 cores.

La plupart des systèmes critiques sont dits «temps-réel dur» puisqu'ils requièrent des garanties temporelle fortes. Ces systèmes sont de plus en plus complexes et les processeurs mono-cœurs traditionnels ne sont plus assez puissants. Les multi-cœurs et les pluri-cœurs sont des alternatives plus puissantes, cependant ils contiennent des ressources partagées. Les accès concurrents à ces ressources provoquent des interférences qui doivent être prises en compte puisqu'elles rendent les délais d'accès non prédictibles. Pour les pluri-cœur, le réseau sur puce (NoC) doit être configuré pour éviter les interblocages et garantir des pires temps de traversée précis. Le MPPA2 de Kalray est un pluri-cœur avec de bonnes propriétés temporelles.

Les langages Synchrones flot de données tels que Lustre ou Scade sont largement utilisés dans l'industrie aéronautique. Les programmes sont des réseaux de nœuds de calcul communicants. Nous présentons une méthode pour extraire le parallélisme des programmes Synchrones. Nous générons du code pour déployer les tâches parallèles sur la puce et pour implémenter les communications en mémoire partagée ou à travers le NoC. Notre solution permet la traçabilité du code. Elle est basée sur un modèle d'exécution dirigé par le temps où chaque tâche a une date de début. L'ordonnancement est statique et minimise les interférences grâce à l'utilisation de bancs mémoire. Une borne de pire temps d'exécution (WCET) est calculée. Elle inclut les interférences mémoire et les pires temps de traversée NoC. Nous générons la configuration du processeur qui permet une allocation équitable des bandes passantes sur le NoC, la garantie de temps de traversées bornés et la synchronisation des horloges. Enfin, nous appliquons notre outils sur des exemples de programmes aéronautiques et un exemple synthétique utilisant 64 cœurs.