

Minimising communication costs impact when scheduling real-time applications on multi-core architectures

Benjamin Rouxel

▶ **To cite this version:**

Benjamin Rouxel. Minimising communication costs impact when scheduling real-time applications on multi-core architectures. Computer Science [cs]. Université de Rennes 1, 2018. English. NNT : . tel-01945456v2

HAL Id: tel-01945456
https://inria.hal.science/tel-01945456v2

Submitted on 21 Dec 2018

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

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

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathématique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Benjamin ROUXEL »

« Minimising communication costs impact when scheduling real-time applications on multi-core architectures »

Thèse présentée et soutenue à RENNES , le 19 Décembre 2018
Unité de recherche : Irisa – UMR6074

Rapporteurs avant soutenance :

Claire Pagetti Ingénieur de recherche ONERA, Toulouse

Matthieu Moy Maître de conférences Université de Lyon

Composition du jury :

Président :	Daniel Chillet Professeur ENSSAT Lannion – Université Rennes 1
Examinateurs :	Joël Goossens Professeur Université Libre de Bruxelles
	Frédéric Pétrot Professeur ENSIMAG, Grenoble
Dir. de thèse :	Isabelle Puaut Professeur Université Rennes 1
Co-dir. de thèse :	Steven Derrien Professeur Université Rennes 1



Il vaut mieux se tromper en allant de l'avant que d'avoir raison en reculant.
par Frédéric Dard

TABLE OF CONTENTS

Résumé de thèse **7**

Introduction **11**

1 Hard real-time systems and multi-core platforms **17**
1.1 Task models and their expressiveness . . . 18
1.2 Predictable multi-core architectures . . . 21
1.3 Towards parallel and predictable execution models . . . 24
1.4 Task and Inter-core communication . . . 26
1.5 Worst-case execution time estimation . . . 28
1.6 Real-time scheduling: a state of the art . . . 29
1.6.1 Classification of single-core schedulers . . . 29
1.6.2 Multi-core partitioned scheduling . . . 30
1.6.3 Multi-core global scheduling . . . 31
1.6.4 Multi-core hybrid scheduling . . . 32
1.6.5 Shared resource management on single-core and multi-core architectures 32
1.7 Conclusion . . . 33

2 Generic static scheduling frameworks with worst-case contention **35**
2.1 Basic predictable multi-core architectures . . . 36
2.2 Software Model . . . 38
2.2.1 Inter-core communication . . . 39
2.3 Scheduling framework . . . 39
2.3.1 Example . . . 40
2.3.2 Integer Linear Programming (ILP) formulation . . . 41
2.3.3 Forward List Scheduling algorithm . . . 44
2.4 Experiments . . . 46
2.4.1 Scalability of the ILP formulation . . . 47
2.4.2 Quality of the heuristic compared to the ILP . . . 48
2.4.3 Impact of T_{slot} on the schedule . . . 49
2.5 Conclusion . . . 50

3 Computing the precise contention to build contention-aware schedules **51**
3.1 Motivating example . . . 52
3.2 Improving worst-case communication cost . . . 54
3.3 Resource-aware scheduling techniques . . . 56
3.3.1 Integer Linear Programming (ILP) formulation . . . 56
3.3.2 Forward List Scheduling algorithm . . . 58
3.4 Experiments . . . 62
3.4.1 Quality of the heuristic compared to ILP . . . 63

3.4.2 Quality of the heuristic compared to worst-contention communications . . 64
3.4.3 Quality of the heuristic compared to contention-free communications . . . 64
3.5 Related work . . . 65
3.6 Conclusion . . . 66

4 Hiding communication latencies in contention-free schedules 69
4.1 Hardware support . . . 70
4.2 Software & execution model support . . . 71
4.3 Motivating example . . . 71
4.4 SPM allocation scheme . . . 74
4.5 Non-blocking contention-free scheduling techniques . . . 75
4.5.1 Integer Linear Programming (ILP) formulation . . . 75
4.5.2 Forward List Scheduling algorithm . . . 79
4.6 Experiments . . . 83
4.6.1 Quality of the heuristic compared to the ILP . . . 83
4.6.2 Blocking vs non-blocking communications . . . 84
4.6.3 Impact of fragmentation strategy . . . 86
4.6.4 Impact of topological sorting algorithm . . . 88
4.7 Related Work . . . 88
4.8 Conclusion . . . 90

Conclusion 91

Appendices 94
STR2RTS benchmark suite . . . 94

Bibliography 95

List of Figures 109

List of Algorithms 111

List of Publications 112

RÉSUMÉ DE THÈSE

L'année 1969 marqua le point culminant de la course à l'espace lorsque Neil Armstrong, Buzz Aldrin et Michael Collins firent un grand pas pour l'humanité lorsqu'ils marchèrent sur la Lune. Leur voyage commença avec le lancement de la fusée *Saturn V* qui est considéré comme le premier système critique. Pour assurer son contrôle, cette fusée inclue un ordinateur de guidage *(Apollo Guidance Computer – AGC)* qui exécute un système d'exploitation temps-réel. Celui-ci permettait aux astronautes d'entrer des commandes simples afin de commander la fusée. L'architecture matérielle inclue, entre autres, un processeur 16-bits simple-cœur offrant 64 Koctets de mémoire approximativement, et opère à une fréquence de 2,048 MHz [1]. Une comparaison naïve serait d'opposer ces caractéristiques techniques avec les téléphones portables intelligents de notre époque. En effet, ceux-ci offrent une puissance de calcul mille fois supérieure à celle qu'il a fallu pour envoyer des astronautes sur la Lune, et surtout pour les récupérer en un seul morceau. Depuis lors, la demande de puissance de calcul n'a cessé d'augmenter, amenant les fabricants de matériels a continuellement amélioré leurs processus de fabrication des puces informatiques. Les deux principales améliorations possibles ont pendant longtemps été l'augmentation de la densité des transistors et l'augmentation de la fréquence d'horloge. Ces augmentations suivirent la loi de Moore [Sch97] jusqu'en 2004, où elles atteignirent une limite technologique connue sous le nom du Mur de Puissance (Power Wall) [Kur01].

Á cause du Mur de Puissance, l'augmentation de la densité et de la fréquence ne sont donc plus possible avec la technologie actuelle. La solution trouvée par les constructeurs, afin continuer d'augmenter la puissance de calcul des processeurs, est d'accroître le nombre de cœurs à l'intérieur d'une même puce. Il est dorénavant très facile de trouver des processeurs commerciaux grand public avec 4, 8 cœurs, ou même plus. Par exemple, les processeurs de la marque Intel© séries Core™i[3-9] contiennent de 2 à 18 cœurs. Les systèmes critiques ne peuvent échapper à l'évolution des processeurs, et, peu à peu, intègrent les processeurs multi-/pluri-cœurs au sein de leur plateforme dans des domaines tel que l'automobile, l'aviation, ou l'espace [BDN+16 ; HMC+16].

L'informatique temps-réel-dur portent sur les applications calculant non seulement le bon résultat, mais, et encore plus important, qui le calculent dans le temps imparti. En plus des contraintes imposées aux systèmes classiques (e.g. performance, énergie), les systèmes temps-réels-dur ajoutent des contraintes temporelles (eg. temps de relâche, échéances, . . .) . De plus, manquer une contrainte temporelle, dans un système temps-réel-dur, mène à un échec total du système, ce qui peut avoir des conséquences désastreuses. Par exemple, dans le domaine de l'aviation, un manquement d'échéance peut causer la perte de vies humaines ; dans le domaine de l'aéro-spatiale, des milliards de dollars peuvent être gâchés avec le satellite brûlant inopinément dans l'atmosphère.

Concevoir un système temps-réel-dur nécessite des garanties strictes et plus d'attention que pour tout autre système, en particulier en ce qui concerne les contraintes temporelles. Afin d'obtenir les garanties les plus fortes, les performances pire cas sont calculées *a priori* avec,

1. https ://en.wikipedia.org/wiki/Apollo_Guidance_Computer

entre autres, l'estimation statique du plus long temps d'exécution (Worst-Case Execution Time – WCET) et de politiques d'ordonnancement statiques. Les plates-formes multi-cœurs sont une solution très attrayante pour la mise en œuvre de systèmes temps-réel-dur. Lorsque ces plateformes sont spécifiquement conçues dans un souci de prévisibilité, elles offrent à la fois de bonnes performances et la possibilité de détermine des performances pire cas précises. De plus, les applications parallèles permettent d'exploiter pleinement toutes les ressources disponibles au sein des architectures multi-cœurs. La meilleure représentation pour les applications parallèles fournit explicitement à la fois les dépendances et les concurrences entre les tâches, car ces informations sont obligatoires pour calculer de manière statique des performances pire des cas précises. Avant d'exécuter une application sur une plateforme multi-cœur, une étape obligatoire est de décider sur quel cœur et quand exécuter les tâches de l'application. Un ordonnanceur place les tâches sur les cœurs, et ordonnent leur exécution sur le cœur. Les décisions de placement et d'ordonnancement ont un impact sur le temps d'exécution global de l'application sur les processeurs multi-cœurs. Le point central de cette dissertation est la construction d'une politique d'ordonnancement dont l'objectif est de minimiser la taille de l'ordonnancement.

Cette thèse présente deux méthodes d'ordonnancement statiques ayant pour but d'optimiser les performances globales d'une application donnée tout en garantissant des contraintes temporelles fortes. Elles peuvent être classées comme partitionnées statiques et non préemptives. La mise en œuvre des stratégies d'ordonnancement proposées inclut à la fois une formulation à base de programmation en nombre entier (Integer Linear Programming – ILP) et un algorithme approximatif basé sur une heuristique gloutonne similaire à l'ordonnancement par liste. Les formulations ILP fournissent une description non ambiguë du problème étudié, et servent également de base pour évaluer la qualité de la proposition d'algorithme heuristique. Les deux stratégies d'ordonnancement ciblent des plateformes multi-cœurs dans lesquelles, les cœurs sont inter-connectés par un bus arbitrée avec une politique de tournoi à la ronde juste (FAIR-Round-Robin). Chaque cœur est également supposé avoir accès à une mémoire locale privée, ou mémoire bloc-notes (ScratchPad Memory – SPM). Dans cette dissertation, les applications parallèles sont considérées représentées sous forme de graphiques de tâches acycliques (Directed Acyclic Graphs – DAG). Afin de tirer parti au mieux d'architectures à mémoire bloc-notes, le modèle d'exécution utilisé est Acquisition Exécution Restitution (AER) [MNP+16]. Il force les accès mémoire à être isolés des phases de calcul. Cette séparation permet d'abord de lire les données d'entrée depuis la mémoire globale dans la SPM, puis d'exécuter le code sans interférence avec les autres cœurs, et enfin d'écrire les données produites de la SPM vers la mémoire externe. Dans les approches défendues dans cette thèse, les tâches de communication sont supposées ne transférer que des données via la mémoire principale (les communications de SPM à SPM ne sont pas autorisées). Dans un premier temps, les communications sont limitées à un mode *bloquant*, Chapitres 2 et 3, puis le mode *non bloquant* est pris en charge au Chapitre 4. Ce travail a pour objectif essentiel de minimiser l'impact de ces communications sur la durée globale d'exécution de l'application.

Avec les architectures multi-cœurs, plusieurs cœurs peuvent accéder au bus partagé en même temps. Par conséquent, le calcul des latences de transfert, entre les cœurs et la mémoire hors puce via le bus, doit prendre en compte un délai de contention. Ce délai dépend généralement du nombre de cœurs en conflit pour l'accès au bus partagé. Le calcul des latences de communication dans le Chapitre 2 utilise un délai de contention dans le cas le plus défavorable, qui considère toujours que tous les autres cœurs demandent également à accéder au

bus partagé, au même instant. Même si très pessimiste, ce cas défavorable fournit une base de référence permettant la comparaison lors de l'ajout, à l'ordonnancement, de conscience des conflit dans le Chapitre 3. Une évaluation empirique montre que la résolution d'un problème d'ordonnancement, tout en recherchant un optimum exact ne s'adapte pas aux grand problèmes (comme prévu). Pour les cas de test impliquant un grand nombre de tâches, le seul moyen d'obtenir une solution consiste à s'appuyer sur une méthode approximative. Ensuite, une évaluation empirique, dans la Section 2.4.3, montre que le paramètre spécifique T_{slot} d'un bus FAIR Round-Robin, a un impact négligeable sur la durée de l'ordonnancement.

Le modèle de conflit dans le cas le plus défavorable est un choix sûr par construction, mais il conduit à une sur-approximation importante qui est ensuite affinée dans le Chapitre 3. Dans ce chapitre, la méthode d'ordonnancement proposée utilise la connaissance de la structure de l'application conjointement à celle de l'ordonnancement courant, pour affiner, au moment de la conception et pour chaque phase de communication, la pire quantité réelle d'interférences. Cette méthode s'est révélée efficace avec l'évaluation empirique de la section 3.4 montrant une amélioration moyenne de 59% par rapport au pire des scénarios. Toutefois, des expériences ont également montré que, dans la plupart des cas, le fait de permettre à l'ordonnanceur de choisir entre éviter les contentions dans l'ordonnancement ou les autoriser avec un calcul précis de celles-ci, aboutissait à des ordonnancement exempts de conflits. Cette observation doit cependant être mise en perspective avec le fait que les travaux présentés ne considèrent que des graphes de tâches avec la sémantique AER.

Afin de poursuivre l'affinement des délais de contention, la contribution du Chapitre 4 décrit une méthode d'ordonnancement statique générant des ordonnancements sans contention et dont les communications sont non-bloquantes. L'ordonnanceur tire parti du moteur d'accès direct à la mémoire (Direct Memory Access engine – DMA) et de la SPM à double accès, afin de masquer les accès au bus lorsque l'unité de traitement est occupée à effectuer des calculs. L'approche présentée fragmente les communications pour augmenter les possibilités de chevauchement avec les calculs. L'évaluation empirique montre que, comparée à un scénario sans chevauchement, cette approche améliore la longueur d'ordonnancement 4% en moyenne sur les applications en flux-continu (8% sur les graphes de tâches synthétiques). Néanmoins, des expériences montrent également que le fait de permettre différentes stratégies de fragmentation peut augmenter encore le gain ; lorsque l'on évite des frais de transfert minimes, jusqu'à $6,5\%$ avec les applications de flux-continu. En outre, différentes méthodes de tri provenant de l'algorithme heuristique sont évaluées, mais aucunes d'elles ne semblent surpasser aucunes autres. Enfin, l'implémentation réalisée avec succès est résumée, elle porte sur la réalisation des ordonnancements générés par l'heuristique avec pour cible une grappe de la plateforme Kalray MPPA Bostan [DVP+14]. Sur cette plate-forme, le gain observé pour l'ordonnancement avec fragmentation par arc du graphe, par le paramètre D_{slot}, ou par DTU en mode *non bloquant* est respectivement : 36%, 22% et 24%. Avec des gains au-delà de nos attentes, cette mise en œuvre valide les avantages de notre stratégie d'ordonnancement.

Contenu de la thèse

Le reste de cette thèse est divisé en quatre chapitres principaux, résumés comme suit :

Chapitre 1 introduit le contexte de cette thèse à travers une revue des systèmes temps-réel et des plateformes multi-cœurs. Cela commence par les modèles d'applications fondamentaux et leur expressivité. Ensuite, des plateformes multi-cœurs sont présentées, dans lesquelles la prévisibilité et le déterminisme sont des caractéristiques essentielles. Des caractéristiques similaires sont ensuite prises en compte dans les modèles d'exécution qui spécifient comment les applications parallèles sont exécutées sur des plateformes multi-cœurs. Enfin, un examen des travaux précédents sur les stratégies d'ordonnancement / placement pour les applications en temps réel sur des architectures mono-cœurs / multi-cœurs est présenté.

Chapitre 2 présente la infrastructure d'ordonnancement extraite de la littérature et utilisée comme base pour les contributions suivantes. Notre infrastructure peut calculer un placement et un ordonnancement statiques pour une application s'exécutant sur un multi-cœur en tenant compte d'hypothèses initiales simples (par exemple, le cas le plus défavorable de conflits). Toutes ces hypothèses restrictives sont abandonnées dans les chapitres suivants, à savoir la prise de conscience de la contention dans le Chapitre 3, et l'évitement des conflits dans le Chapitre 4. L'infrastructure est d'abord détaillée avec une formulation ILP (Integer Linear Programming) et à l'aide d'un modèle de contention projetant le cas le plus défavorable. Ensuite, un algorithme basé sur l'ordonnancement par liste est présenté. Il utilise le même modèle de conflits mais permet de mieux s'adapter aux grandes applications. Certaines expériences initiales démontrent la viabilité de l'heuristique.

Chapitre 3 décrit une nouvelle technique permettant de prendre en compte la quantité réelle d'interférences lors de l'ordonnancement hors ligne. Il présente la méthodologie et sa mise en œuvre à la fois via une technique demandant un calcul intensif mais fournissant des résultats exacts, et via un algorithme plus rapide mais fournissant des résultats approximatifs. Enfin, les expériences démontrent que la sur-approximation de l'algorithme est limitée, suivi des gains de la méthode opposé à celle du chapitre **??** qui inclue un scénario de conflit pire cas. La dernière expérience discute les améliorations de cette méthode par rapport à une méthode existante adaptée dont les ordonnancements sont sans conflit.

Chapitre 4 relâche certaines des hypothèses restrictives sur le matériel afin de s'appuyer sur une plateforme plus réaliste. Tout d'abord, la restriction sur la taille de la mémoire SPM (ScratchPad Memory) est supprimée tandis que le temps de latence pour transmettre des données sur le bus est masqué. Pour augmenter les possibilités de masquage, la communication est fragmentée. Les expériences valident d'abord le comportement de l'heuristique par rapport à la formulation ILP. Ensuite, le gain est exprimé par rapport à l'infrastructure de base du Chapitre 2 incluant un scénario de communication non-masquant et non-fragmenté. La dernière expérience traite de la taille des fragments utilisée dans les stratégies de planification.

Chapitre 5 conclut cette thèse en résumant toutes les contributions présentées. Des travaux futurs possibles sont ensuite introduits afin d'optimiser davantage l'ordonnancement des systèmes temps-réel-dur sur du architectures multi-cœurs.

INTRODUCTION

In 1969, Neil Armstrong, Buzz Aldrin and Michael Collins made a giant leap for mankind when walking on the moon. Their journey started with the launch of the rocket *Saturn V* which is considered as the first safety-critical system. The rocket included the Apollo Guidance Computer (AGC) which ran a real-time operating system. This system enabled astronauts to enter simple commands to control the rocket. The hardware architecture included, among other devices, a 16-bits single-core processor which had approximately 64 Kbytes of memory and operated at 2.048 MHz [2]. As a naive comparison, modern smartphones offer a thousand times more computing capacity than it was required to launch and safely return astronauts from the moon. Computational demand has kept increasing ever since, and hardware manufacturers continuously improved chip manufacturing techniques, both in terms of transistors density and clock frequency. Until 2004, this growth followed the Moore's law [Sch97]. Until it reached a technological limit, known as the Power Wall [Kur01]. This limit initiated from power leakage and heat of the chip which both increase with the diminishing size and the raise of transistors frequency.

Due to the Power Wall, increasing the density and frequency was not an option anymore. The solution found to augment the processors computational power without increasing the frequency was to extend the number of cores within a chip. It is now very common to find commercial mainstream processors with 4, 8 or more cores. For example, the Intel© Core™i[3-9] includes from 2 to 18 cores. Critical systems cannot escape from this processors' evolution and now increasingly include multi-/many-core processors at the heart of their hardware, such as automotive, avionic or space industries [BDN+16; Per17]. Therefore, the innate idea of this dissertation integrates this evolution by targeting multi-core usage in safety-critical systems.

Real-time computing refers to applications that compute correct results but, and more importantly, applications that perform *on-time*. As pictured in Figure 1, their goal is not to be fast, but rather to be *on-time*. In addition to usual system constraints (e.g. performance, energy), real-time systems include timing constraints that need to be satisfied by the system in order to provide a safe computation. Examples of timing constraints are, among others, release time, deadline, worst-case execution time, ... In a safety-critical system with timing constraints, failing one constraint can lead to disastrous consequences. For example, in the avionic domain, a timing constraint failure can cause a waste of human lives in a crash; for a satellite, billion euros might end wasted with the satellite burning into the atmosphere. Depending on their criticality level, real-time systems can be classified in three categories [SR94]:

2. https://en.wikipedia.org/wiki/Apollo_Guidance_Computer

Hard: missing a timing constraint is a total system failure.

Firm: infrequent timing constraint misses are tolerable, but may degrade the system's quality of service.

Soft: the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service

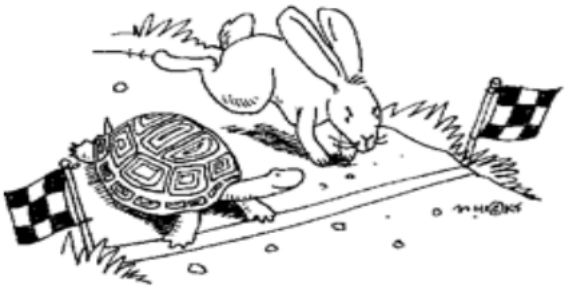


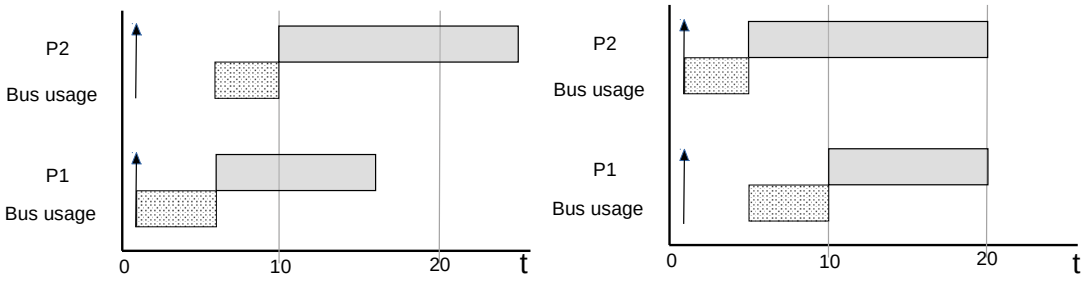
Figure 1 – Real-time is not real-fast

Designing a hard real-time system requires more attention and strict guarantees, especially on timing constraints, than any other systems, from Reineke [RGB+07] : "*An important part in the design of hard real-time systems is the proof of punctuality which is determined by the worst-case performance of the system.*". In this context, the worst-case performance corresponds to the execution scenario where the time to execute the software on the hardware is the longest possible. In addition, to enforce the strongest guarantees, this worst-case performance is computed *a priori*, which means without actually executing the software on the platform (this is also known as a *static* approach). However, determining the worst-case performance is extremely tedious, especially when determined *a priori*. Main issues originate from the absence of detailed information on the hardware parts. Indeed, in modern complex multi-core platforms, many components speculate on the execution flow (e.g. out-of-order pipelines, branch and value prediction, shared cache levels, ...) . Because results from speculations are hard to predict without actually executing the program, this ends in an under-utilisation of the platform capacity as in practice the over-provisioned worst-case has almost no chance to actually happen. In addition, manufacturers mostly hide, in their commercial platforms, such details about the architecture due to intellectual property restrictions. In some specific cases, hard real-time system designers mostly rely on custom architectures designed for their domain (e.g. the Leon processor from the European Space Agency (ESA) [Gai02] or on an abstraction of the platform [BB08]). Due to cost constraints (specific platforms are expensive) and the absence of constraints regarding the application domain, contributions of this dissertation show how to reduce this over-provisioning when computing worst-case performance on an abstracted multi-core architecture.

Multi-core hardware platforms, even when designed specifically for hard real-time systems, lead to unpredictability when two or more cores request an access to the same resource at an identical moment in time. In this case, it might be impossible [3] to determine, *a priori*, the actual granting order. However, this order has an impact on the worst-case performance as one task can complete before an other one. Figure 2 shows two executions where the X-axis denotes time and where solid arrows are release times. On both figures, a computing task (plain box) follows a bus request (dotted box). Each computing task is executed on an individual core, named $P1$ or $P2$. In addition, a shared bus connects the two cores. Because only one bus

3. This, obviously, depends on the arbitration policy, for example it is easy to determine the order with a TDM arbiter, but impossible with a RR one where the worst-case should be considered. See Section 1.4 for a review on arbiters.

request can be performed at a time (mutually exclusive), both of them initiate from different cores, therefore need to be scheduled. In Figure 2a, the overall makespan is 25 time units where it decreases to 20 time units in Figure 2b just by inverting the granting order of the two bus requests. It is therefore clear that the order of bus requests impacts the final overall execution time of the application. This *shared bus* is the central component of this study.



(a) Overall makespan is 25 time units (b) Overall makespan is 20 time units

Figure 2 – Illustrating example showing the importance of the execution order

Prior to run an application on a multi-core platform, a mandatory step is to decide on which core and when to execute application tasks. A scheduler maps tasks on cores and orders their execution on this core. As seen in Figure 2, mapping and scheduling decisions impact the overall execution time of a parallel application on a multi-core processor. The main point of this dissertation is the construction of scheduling policies where the objective is to minimise the schedule length while taking care of bus accesses.

Even when relaxing the indeterminism caused by context sensitive hardware decisions (speculative features, or arbiters), worst-case performance analyses can be unable to provide a result. Indeed, some common software features (e.g dynamic memory allocation, *goto*) lead to statically unpredictable behaviours. Despite programming rules to avoid such indeterministic behaviours, the compiler plays an important role for worst-case performance. At compile time, the timing behaviour can drastically change due to optimisations [PDC+18]. Thus, safety-critical applications require specific programming languages as well as verified compilers to determine and prove worst-case performance. Programming language features and compilers are out of the scope of this dissertation, and in this work, we only consider compilers where optimisation passes are disabled.

In summary, the presented work of this thesis targets hard real-time applications running on multi-core platforms. More specifically, we focus on bus requests management and aim at decreasing the impact of worst-case transmission latencies. This, in turns, decreases the over-provisioning of hardware resources, and inherently increases the utilisation of the platform reaching the goal of minimising the worst-case performance.

Challenges in real-time systems

Guaranteeing that timing constraints are met on both single-core and multi-core platforms is challenging. The first challenge lies in the computation of the Worst-Case Execution Times (WCET). WCET estimates correspond to a safe and tight upper bound of the execution time

of an application executing on a given platform. As represented by Figure 3, a safe WCET estimate must be proven to be higher or equal to any effective execution time. For hard real-time systems, static WCET analyses produce the strongest guarantees on the results in terms of both safety and accuracy. Achieving such complex analyses requires detailed information on the hardware platform (pipeline, caches, ...) to devise tight estimates [WEE+08]. A common practice is to perform the analysis at the binary level, as this is the closest representation to what is executed by the hardware. However, control flow information (e.g. loop bounds) are generally lost at compile time, although they are required to estimate WCETs [LPR14]. In order to be safe, the WCET estimation assumes conservative hypotheses, where reducing the induced pessimism remains a challenging issue.

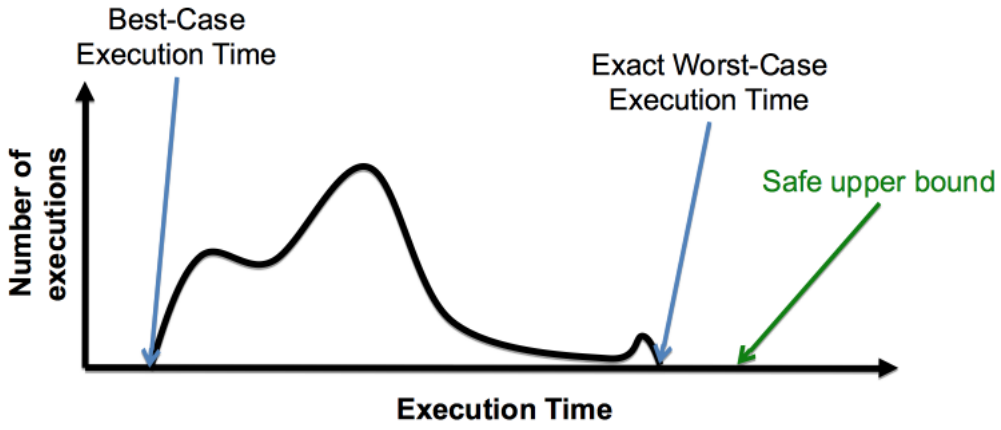


Figure 3 – Worst-Case Execution Times (WCET)

Once WCETs are estimated, a scheduling policy defines the execution order of tasks. Based on this policy, a task might be delayed, or pre-empted (paused and resumed), or forced to wait for a (software/hardware) resource to be available. The task is then blocked for a certain amount of time. The addition of a WCET estimate and a blocking time corresponds to the Worst-Case Response Time (WCRT). Therefore, computing this blocking time and the WCRT is a major challenge for real-time systems.

In addition, the schedule generated according to a given policy must enforce that the execution order implied will enforce a timing constraint for all tasks of the application. Proving the feasibility of a scheduling policy and bounding its computational complexity is still an open problem for certain combinations of policies and task models. For example, in a very recent work Ekberg et al. [EY17] have proven that using sporadic tasks with fixed priority scheduling on uniprocessor is a NP-Hard problem.

Moreover, multi-core platforms need a mapping of tasks on cores along with a schedule. Mapping analysis decides on what core a task will be mapped, Coffman et al. [CGJ96] have shown that this problem is NP-hard as it reduces to the well known bin-packing problem.

The main evolution from single-core to multi-core architecture focuses on the management of shared hardware resources (buses, shared last level of cache, ...) that are shared temporally and physically. When running multiple tasks on a single core, only a single task can execute on the processor and can have access to other resources at a given time instant. When dealing with multiple cores, multiple tasks will compete to access software/hardware re-

sources (e.g. bus, sensors, semaphores . . .) at an identical moment in time. Access requests must then be ordered/prioritized, since only a single request can be fulfilled at a time. These tasks, hence, interfere with each other and such interferences influence their predictability and timing behaviour. Identifying the region in which contention occur, and precisely estimate this contention is an other key challenge as it has a significant impact on the WCRT of tasks. Since it is difficult in general to guarantee the absence of resource conflicts during execution, current WCRT techniques either produce pessimistic WCRT estimates or constrain the execution to enforce the absence of conflicts, often at the price of a significant hardware under-utilisation.

Contributions

Contributions of this work address hard real-time applications running on multi-core platforms and are applied at design time. The general contribution focuses on bus requests management and aims at decreasing the impact of worst-case transmission latencies on schedule length. The presented work optimises the overall schedule makespan and increases the utilisation of the platform.

The first proposed method aims at determining the effective amount of interferences in a static schedule in order to reduce the pessimism of worst-case contention model. It consists in a *contention-aware* scheduling strategy that produces time-triggered schedules of the application's tasks. Based on knowledge of the application's structure, this scheduling strategy precisely estimates the *effective* contention, and minimises the overall makespan of the schedule. An Integer Linear Programming (ILP) solution of the scheduling problem is presented, as well as a heuristic algorithm that generates schedules very close to the ILP results (2 % longer on average), with a much lower time complexity. The heuristic improves by 59% the overall makespan of resulting schedules compared to a worst-case contention baseline.

We extend our first approach to further reduce the overall schedule makespan by relaxing previous assumptions on the hardware, which in turns, enables new scheduling opportunities. The second contribution therefore proposes techniques to select ScratchPad Memory (SPM) contents off-line, jointly with schedule generation, in a way such that the cost of SPM loading/unloading is hidden, thanks to overlapping communication and computation phases. More precisely, we take advantage of communication fragmentation to bring more opportunities to benefit from such overlapping. Experimental results show the effectiveness of the proposed techniques on streaming applications and on synthetic task-graphs. The length of generated schedules is reduced by 4% on average on streaming application (8% on synthetic task graphs) by overlapping communications and computations.

All the code of the scheduler is available at `https://gitlab.inria.fr/brouxel/methane`.

Thesis outline

The rest of this thesis is divided into four main chapters, summarised as follows:

Chapter 1 introduces the context of this thesis through a review of real-time systems and multi-core platforms. It starts with the fundamental application models and their expressive-

ness. Then, multi-core platforms are presented, in which predictability and determinism are key features. Similar features are considered, afterwards, with execution models which specify how parallel applications are executed on multi-core platforms. Finally, a review of prior results on scheduling/mapping strategies for real-time applications on single-/multi-core architectures is given.

Chapter 2 presents the scheduling framework extracted from the literature and used as a baseline for following contributions. Our framework can compute a static mapping and scheduling for an application mapped on a multi-core considering simple initial hypotheses (e.g. worst-case contention). All those restrictive hypothesis are lifted in following chapters, i.e. contention-awareness in Chapter 3, and free from contention in Chapter 4. The framework is first detailed with an Integer Linear Programming (ILP) formulation, and using a worst-case contention model. Then, a list-scheduling-based algorithm is presented which uses the same contention model to scale better on larger applications. Some initial experiments demonstrate the viability of the heuristic.

Chapter 3 describes a novel technique to account for the effective amount of interference in off-line scheduling algorithm. It presents the methodology and its implementation in both a computationally intensive but exact method and in a faster but approximate algorithm. Finally, experiments demonstrate the limited over-approximation of the algorithm, along with gains over the framework from Chapter 2 with a worst-case contention scenario. The last experiment discusses the improvements over an adapted state-of-the-art algorithm with contention-free scenario.

Chapter 4 lifts some of our restrictive hypotheses on the hardware in order to rely on a more realistic platform. First, the restriction on the size of the ScratchPad Memory (SPM) is removed while the latency to transmit data on the bus is hidden. To increase hiding opportunities, communication are fragmented. Experiments first validate the behaviour of the heuristic over the ILP formulation. Then, the gain is expressed over Chapter 2 baselines with a non-overlapping and non-fragmented communication scenario. The last experiment discusses the size of the fragment used in the scheduling strategies.

Chapter 5 concludes this dissertation by summing up all presented contributions. Then possible future works are devised to further optimise the schedule makespan of hard real-time software on multi-core hardware.

CHAPTER 1

HARD REAL-TIME SYSTEMS AND MULTI-CORE PLATFORMS

Designing hard real-time systems implies to carefully select all hardware and software components. In order to guarantee the timing behaviour of the system. This chapter lists different possibilities where the production of a safe system is the Holy Grail.

We assume that a real-time system contains a number of tasks, that fulfil the functional requirements. A task generally exhibits a variation in its execution time. The Worst-Case Execution Time (WCET) estimate of a task corresponds to an upper bound of any possible execution time for that particular task.

Then all tasks from the set will eventually be executed on a given core, but only one can execute at a time per core. Hence, a scheduling policy describes how to construct an execution order for a given task-set, also known as a schedule. At design time, a schedulability analysis proves (or disproves) that the pair tasks-set, scheduling policy is schedulable, i.e. a schedule exists and meets all timing constraints.

The task WCET estimation is influenced by the schedule, for example, the initial cache state for a task depends on the task previously executed. Moreover, analysing the schedulability of a system requires each task WCET estimate. As a consequence, both WCET estimation and schedulability analysis are co-dependent, as each one impacts the other.

The evolution to multi-core architectures increases this co-dependence. In the general case, mapping decisions influence the WCET estimation accuracy. Indeed, two cores impact each other WCET estimations when mapped on different cores with overlapping request time on a shared resource.

To break this co-dependence, nowadays techniques manage to enforce timing isolation. The isolation in time enables to study the timing behaviour of a component in independence from the others with the guarantee that no other components will affect the result. Applying this timing isolation principle can be accomplished at different levels from the hardware with cache locking or ScratchPad Memory (SPM) (e.g. [PP07]), on the execution model with the separation of the memory access from the computation (e.g. [PBB+11]), or on the communication model with contention-free mechanism (e.g. [BDN+16]).

This chapter reviews these different key concepts to, at the end, focus on state-of-the-art techniques on scheduling. It is organised as follows. First, Section 1.1 introduces different task models and Section 1.2 different hardware designs for real-time systems. Then, Section 1.3 presents execution models enforcing timing isolation, followed by Section 1.4 describing available communication configurations. Next, Section 1.5 summarises methods to determine WCETs. Finally, Section 1.6 presents a review on real-time scheduling targeting multi-core architectures before concluding in Section 1.7.

1.1 Task models and their expressiveness

A task model, a.k.a workload model, describes the properties of a real-time application. At a coarse grain, tasks compose the *application*. A *task* corresponds to a piece of code. At a finer grain, tasks infer task instances, known as *jobs*. Each job corresponds to an instance of a task that is effectively scheduled and executed on the platform. A task can generate multiple jobs depending on its timing properties and the studied time window. Jobs are ordered and job i must complete before job $i + 1$. Depending on the level of abstraction and expressiveness of further task models, properties defined for tasks and/or jobs include :

— WCET: an upper bound of any possible execution time,

— period: the frequency at which the task/job is ready for execution,

— deadline: the time at which the execution must be complete,

— predecessors/successors: execution order constraints.

Differences between task models essentially come from the amount of exhibited information (expressiveness), the amount of applications it can represent (flexibility), and the applied generalisation (abstraction level). This section does not attempt to be exhaustive as the multiplicity of task models is tremendous, but it covers the wide range of possibilities. For a list of other task models, the reader is advised to look into [SY15].

The seminal work from Liu and Layland [LL73] introduced the periodic task model. It represents applications where jobs arrival times appear at a known and strict frequency after the first one (defined at designed time). Then, each job must terminate before a deadline relative to its arrival time. The lack of flexibility in term of restrictive periodicity was latter relaxed by Mok [Mok83] with sporadic tasks. This allows releasing tasks at later time point as long as at least a minimum time interval as elapsed between two firings (pseudo-period). Both cases are said periodical task models as they generate an infinite sequence of job instances, released periodically. In addition each job must end before the next one is released.

Three other sub-categories classify the aforementioned task models depending on the deadline property for the entire tasks set. If for all tasks the deadline is equal to the period [LL73], the tasks set is *implicit*. If all deadlines are inferior or equal to the period then the category is *constrained* deadlines, while *arbitrary* deadline is otherwise used [Mok83].

Above-mentioned task models lack of flexibility. As an example, a MPEG decoder sequentially receives, decodes and displays video frames. In such codec, a frame is periodically bigger than others, and thus requires more computational time. Applying the same task/function to all frames results in an over-provisioning of the system as most frame computation need less processor time. The Multi-Frame model [MC96] and its Generalisation (GMF) [BCG+99] allow to configure different properties, such as WCET estimates, per jobs originating from an identical task.

Another class of embedded real-time systems concerns signal processing applications. They mainly focus on images, sounds or any digital signals, e.g. a wireless router, surveillance camera, ... This type of application processes an uninterrupted flow of information. The expressiveness of seminal periodic task model does not capture this flow of information which is transmitted from task to task. In [Ack82], Ackerman expressed the *data-flow program graph* task model, known as Directed Acyclic Graph (DAG) when no cycle are present. This representation increases the expressiveness of the task-model by exhibiting dependencies between

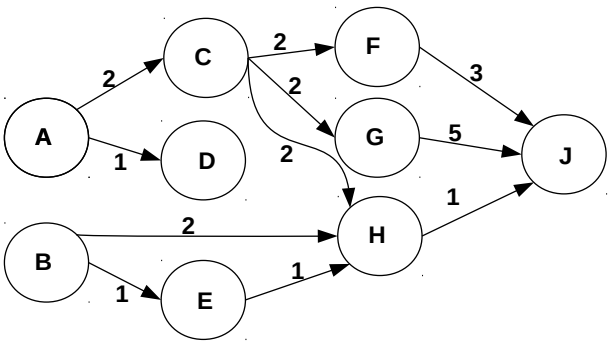


Figure 1.1 – Example of an application represented by a DAG

tasks, thus exposing the parallelism of the application. In data-flow graphs, nodes represent computations (tasks) and edges represent communications between tasks. An edge is present when a task is causally dependent on another one, meaning the source of the edge needs to complete prior to run the target. The edge corresponds to a First In First Out (FIFO) channel where the source produces a certain amount of tokens, and the sink consumes all of them. An example is presented by Figure 1.1 where labels on edges represent the number of tokens.

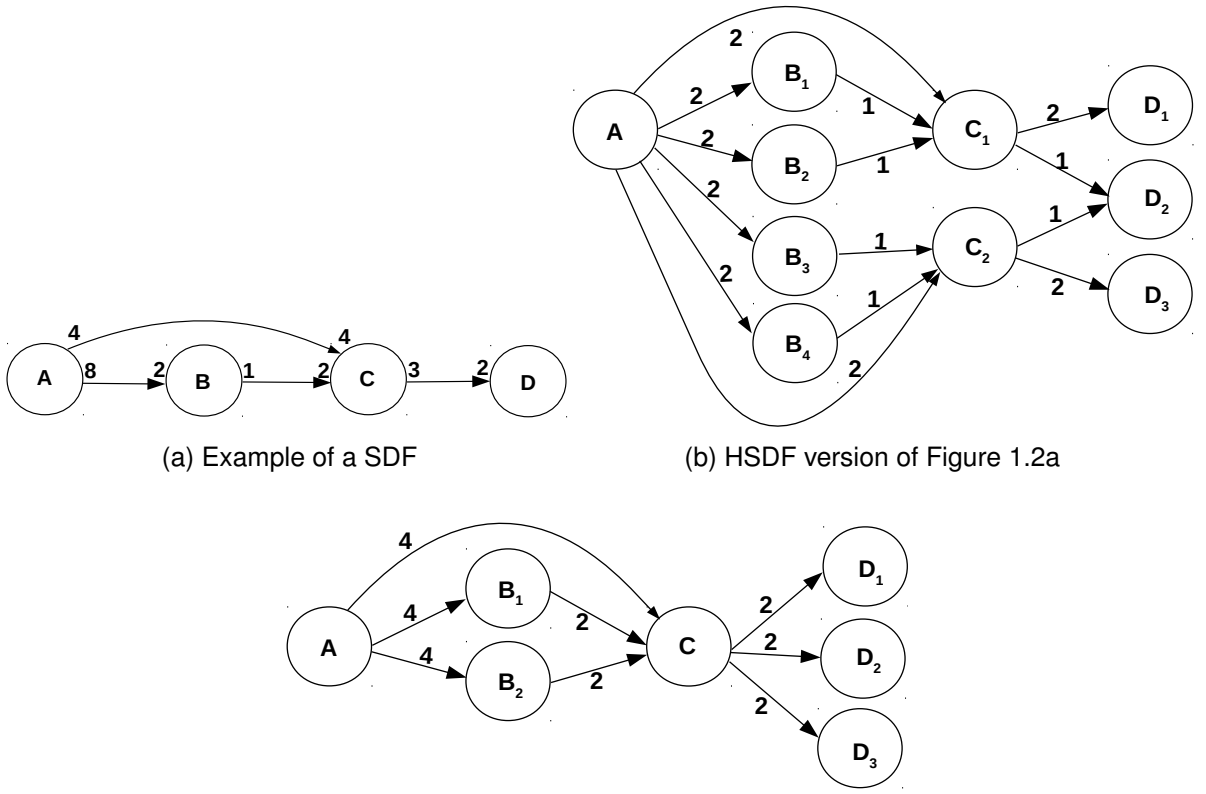
Similarly to periodic task models, a data-flow graph instance is called an *iteration* and a job is a task instance inside an *iteration*. Then, the DAG may iteratively executes until the end of time (or platform is shutdown). Hence, jobs execution order follows aforementioned constraint, job i finished before job $i+1$. But, the iteration $j+1$ can start before the completion of iteration j as long as jobs dependencies are satisfied. This allows to exploit job parallelism, i.e. pipelining [TPG+14].

In data-flow graphs, timing properties (e.g. period, deadline) can be attached to graph itself and not anymore stated for each task and job. All tasks must therefore complete their execution between the release time and the deadline of the whole graph. A multi-task application is then a multi-DAG application as in [Per17].

In general DAGs, edge sinks consume all tokens produced by the corresponding source in one job execution. To overcome this limitation, Synchronous Data-Flow (SDF) graphs [LM87] allow different production/consumption rates between two actors of an edge. An additional constraint on SDF forces the amount of transiting tokens to be known at compile time which allows static analysis on the graph, see Figure 1.2a.

Due to different rates of production and consumption of tokens on an edge, SDF graphs need an expansion pass prior to be scheduled. The larger expansion builds an Homogeneous Data-Flow graphs (HSDF) [LM87], where all production/consumption rates are equal to 1 (there is as much tokens produced than consumed). Despite of the exponential complexity when expanding SDFs, HSDF representations are required [GHK+13] to determine, *a priori*, the amount of node repetitions and all number of transmitted tokens. Figure 1.2b presents the example HSDF obtained after expanding the SDF from Figure 1.2a.

Due to the inherent complexity of building a HSDF, middle size graph representations have been proposed by Zacki [Zak13]. Partial Expansion Graph (PEG) exposes more parallelism than SDF, with potentially less tasks than HSDF. Therefore, using strength from both initial representations.



(a) Example of a SDF

(b) HSDF version of Figure 1.2a

(c) One possible PEG from the SDF in 1.2a

Figure 1.2 – An SDF example and its transformation to HSDF and one possible PEG

In streaming applications, fork-join graphs [TKA02] correspond to an adaptation of SDF graphs. Identically to SDF, they can include different production/consumption rates. They also need to be expanded for further analyses. In contrast to SDF, all actors, except specific fork and join nodes, can have one and only one predecessor and successor. Figure 1.3 exhibits an expanded version of a sample fork-join graph.

Applying the multi-rate idea from GMF to SDF, Bilsen et al. [BEL+96] introduced the Cyclo-Static Data-Flow graph (CSDF). CSDF graphs allow to have different production/consumption rates within a period to jobs of the same task. The example from Figure 1.4 present a CSDF where, for example, actor C has a firing rate of $(2, 1)$ on its output edge. Then actor C alternatively produces 2 tokens then 1 token, then 2 ....

Synchronous Data-Flow applications can be represented with different languages, such as Esterel [BC84], StreamIT [TKA02] or Prelude [PFB+11]. They all have their specificities, Prelude targets multi-periodic synchronous system, while StreamIT generates fork-join graphs targeting streaming applications. All help building parallel applications represented by graphs.

These DAGs do not necessarily need to be built from scratch, which would require an important engineering effort. It is possible to extract tasks from legacy sequential code as in [FDC+13; CM12; CEN+13].

The literature abounds of other graph-based task model, e.g. Hierarchical Task Graph [GP94], Dynamic Data Flow graphs [BDT13]. Only the most common were presented here and

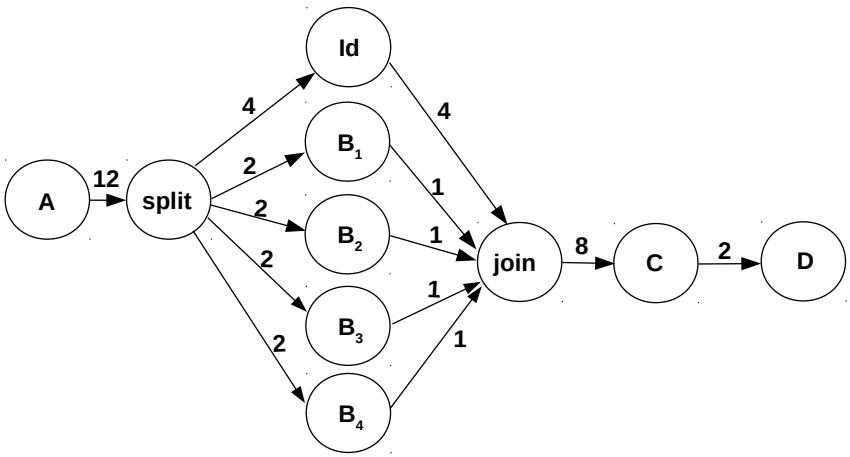


Figure 1.3 – Expanded example of a fork-Join graph

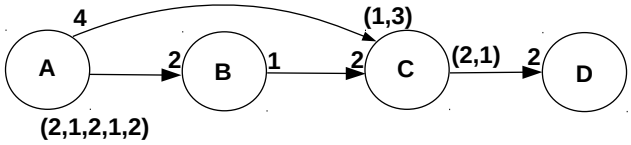


Figure 1.4 – Example of a CSDF

not all of them are suitable for critical applications. An attractive task model for real-time system lies in its expressiveness (no ambiguities, expose parallelism) without too much flexibility (concise) and too high abstraction (remain implementable), therefore allowing proof of timing behaviour.

1.2 Predictable multi-core architectures

For a long time, increasing the computational power of a processor meant increasing the processor clock frequency. Then, manufacturers started to increase the number of cores within a chip instead of the clock frequency, due to technological limits.

In general-purpose multi-core processors, speculative features are key concepts to increase average case performance but mostly worsen WCET estimates. These features are also major sources of indeterminism at different levels: micro-architecture, predictors, memory hierarchy, inter-connection medium, and arbiters. Compared to single-core architectures, in multi-core platforms, the indeterminism comes mainly from interferences when cores try to access a shared hardware element. To fulfil all requirements, hard real-time systems need specific architectures with both properties :

— *Determinism*: it corresponds to the absence of randomness. With identical given inputs, a deterministic system always produces identical results. Then, with real-time systems, determinism also includes timing constraints where these identical results require identical production times.

— *Predictability*: it corresponds to the ability to determine the produced result in advance (without executing the system) for given inputs. Again, with real-time systems, predictability includes the ability to guarantee that timing constraints are met without actually executing the system.

Real-time-oriented multi-core architectures are classified according to the presence of timing anomalies or domino effects [WGR+09; HRW15]. Timing anomalies on the WCET arise when the local worst-case does not entail the global worst-case [CHO12]. The often cited example [LS99] comes from an instruction cache miss that is the local worst-case of a cache analysis, but turns out to not lead to the global worst-case in presence of an out-of-order pipeline. The domino effect is a specific kind of anomalies where an initial local state have an expected impact on the local WCET, but no convergence is possible on the global WCET. For example, the execution time of a loop body differs according to the initial state (caches, pipeline, ...) when entering the loop [Ber06]. Following four general architecture categories classify the further next multi-core architectures:

— *Composable*: the timing of a program running on a core is not influenced by ones running on other cores (no domino effect) and is free from timing anomalies, such as Patmos+Argo [SSP+11; KS14] ;

— *Full timing compositional*: platforms do not exhibit timing anomalies but core analysis are dependent (domino effect), such as Kalray [DVP+14];

— *Compositional with constant bound effect*: these exhibit timing anomalies but no domino effects, such as TriCore [WCM16];

— *Non-compositional*: all other architectures, especially general purpose ones, that include timing anomalies and domino effect.

Following is a, non exhaustive, list of architectures designed for predictability and determinism.

The academic core *Patmos* from Shoeberl et al. [SSP+11] is available on FPGA. It contains a simple RISC in-order pipeline with five stages, see Figure 1.5. All timing information of the Instruction Set Architecture (ISA) are available in [SBH+15]. Different caches/local memories configurations make this core a good candidate for research work. It includes the common instruction/data caches with Least Recently Used (LRU) policy, which is known to be predictable [RGB+07]. But it also comes with other cache types: method cache [DHP+14], heap cache [SHP13; HPS10], stack cache [ABS13]. Split caches separate the source of indeterminism from dynamic allocation and split analyses. Instead of split caches, it is possible to connect the core pipeline to a ScratchPad Memory (SPM). Such memories are more predictable than caches [PP07; MB12] as they are managed by software. Then a compiler decides what and when to store/load data to/from it. To integrate Patmos on a multi-core platform, it is shipped with two fully predictable Networks on Chip (NoC):

i) a bluetree NoC [SCP+14] to access the off-chip memory with a Time Division Multiplexing arbiter ;

ii) a 2-D Mesh NoC [SBS+12; KS14; KS15; KSS+16] to enable inter-core communication, also using time division for the arbitration policy.

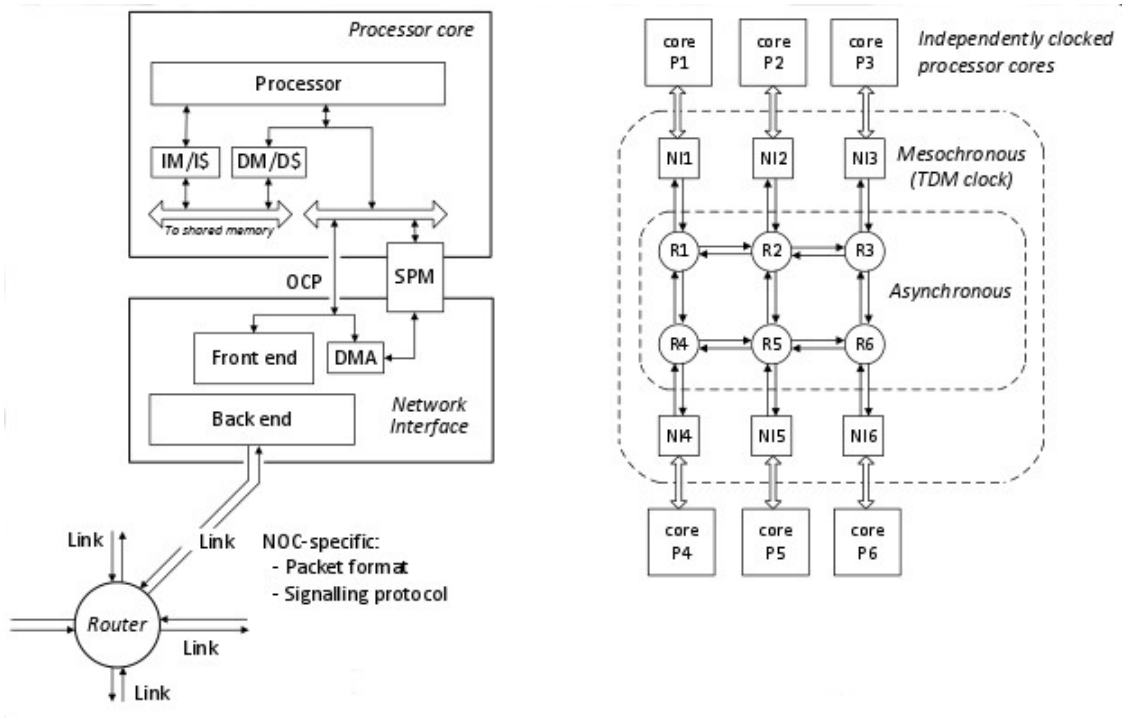


Figure 1.5 – Patmos core (left side) and its Argo NoC (right side), from [KS14]

The Patmos core+Argo NoC is a fully composable architecture, free from timing anomalies, and allows composable analysis due to the full timing isolation thanks to Time Division Multiple Access (TDMA) arbiters.

The commercial platform *Kalray MPPA 256* [DVP+14] contains 256 cores organised in 16 tiles of 16 cores each (see Figure 1.6). All tiles are connected to a 2-D bi-torus NoC enabling packets to transit between tiles, to the off-chip memory, or I/O devices. Inside a tile, the 16 cores are connected to every 16 memory banks (SMEM) through 8 buses (cores are paired). In addition, the SMEM can be configured to enable a private memory to each core, similarly to the SPM in Patmos core. All cores are based on a simple RISC pipeline with seven stages. A vast literature dealing with the Kalray MPPA exists, e.g. [HMC+16; BDN+16; SS16; Per17].Its strengths include an important computational power and design choices with predictability in mind. Some sources of indeterminism still subsist, for example when a tile receives a packet from the NoC: the fixed priority arbiter gives a higher priority to packets coming from the NoC, then they are store in the SMEM; this process adds a delay to cores inside the tile that would also access the same memory bank, and packets coming from the NoC are hardly predictable. The Kalray MPPA is a fully compositional architecture, where WCETs of tasks running on each core are influenced by other cores or NoC incoming packets (domino effects).

From the automotive domain, Wang et al. [WCM16] use a TC27x TriCore micro-controller from Infineon [1]. This platform includes a SPM attached to each core. With its three cores, two identical cores TC1.6P and another core TC1.6E, this processor is heterogeneous. But all these three cores execute the same instructions set. Two independent on-chip buses allow

1. TriCore Microcontroller, http://www.infineon.com/

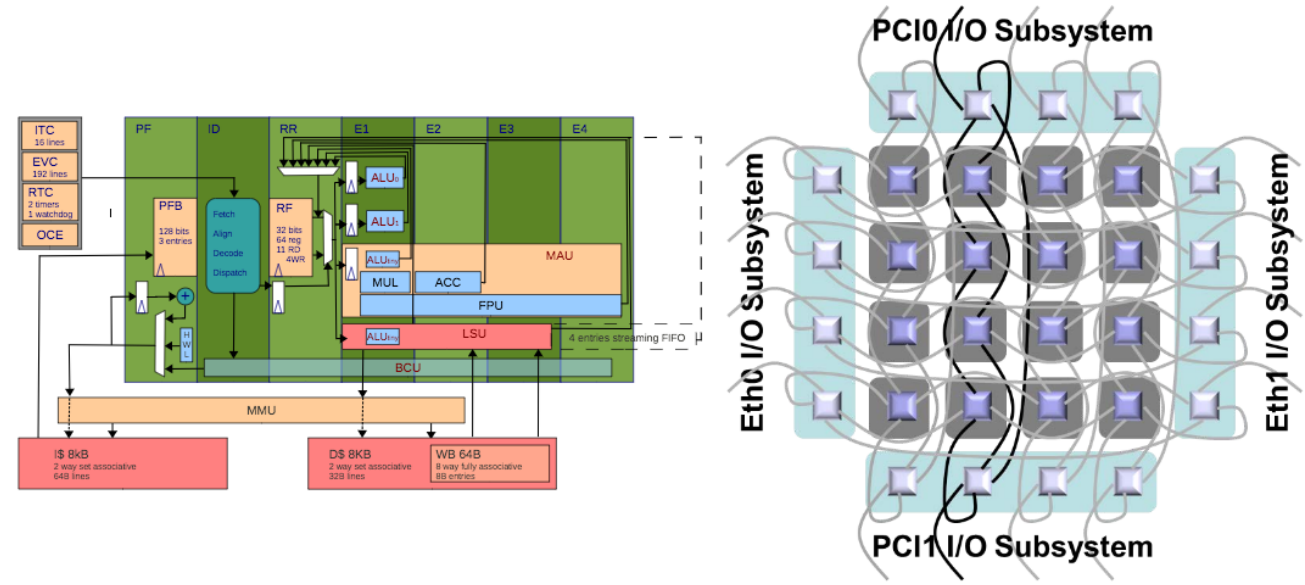


Figure 1.6 – Kalray core (left side) and its NoC (right side), from [DVP+14]

shared resources interconnection, as well as, system peripherals accesses. A Direct Memory Access (DMA) engine connects the global off-chip memory and cores. The TriCore is assumed, but not formally proven, to belong to the category of compositional architectures with constant-bounded anomalies [AEF+14].

1.3 Towards parallel and predictable execution models

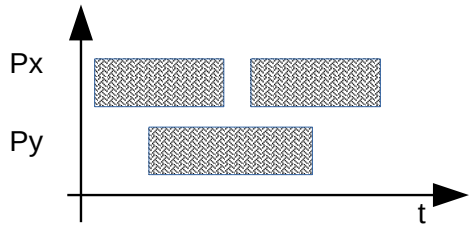
An execution model defines how programs are processed by the hardware. The very first execution model was described by Turing [Tur37]. He detailed how to manipulate symbols on a strip of tape according to a table of rules. This principle was then adapted into von Neumann architecture [Neu82], or in sequential programming language such as C [Rit93].

To benefit from the multiplicity of cores in modern architectures, execution models expose the parallelism of an application. OpenMP [MB12] provides an extension to C code based on pragmas to expose possible parallel regions of code. GPU architectures exhibit a specific execution model [ND10] based on group of instructions, i.e. a warp, where all instructions inside a group execute in parallel.

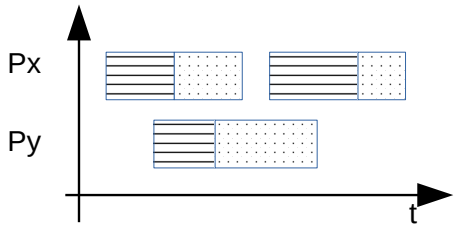
Task models based on graph, e.g. SDF graph [LM87], naturally expose the potential concurrency of applications. Gordon et al. [GTA06] present a method to benefit from all sources of parallelism implied by the SDF task model, in particular they use task pipelining to maximise the throughput of the application.

In hard real-time systems, timing verification can be facilitated with following execution models. For example, Pushner et al. [Pus03] limit the program to a single-path, leaving the exploration of the longest path trivial, see Section 1.5 for WCET estimation details.

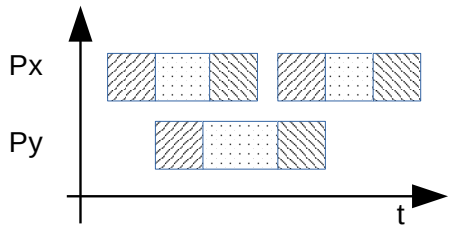
Figures 1.7 oppose the three execution models commonly used in the literature with respect to accesses to the main memory. Figure 1.7a presents three tasks (rectangles) running on two



(a) Example of three tasks with original execution model. No possible differentiation between computation and communication.



(b) Example of three tasks using PREM, a memory phase accesses the off-chip memory to write or read data, then the execute phase can compute without any more accesses to the shared memory.



(c) Example pf three tasks using AER, a read phase first fetch data from the main memory, then an execute phase executes without any access to the shared memory, finally a write phase writes back data to the main memory.

Figure 1.7 – Task representation with different execution model

cores (Px, Py), where the X-axis is the time. In this original execution model, memory accesses and computation are undistinguishable. Therefore it is very difficult to determine if and when memory accesses arise, and moreover if they interfere. A safe solution is therefore to compute a pessimistic WCET for each task where all instructions accessing the memory are considered interfering with similar instructions from other cores.

With the PRedictable Execution Model (PREM), Pellizzoni et al. [PBB+11] decompose a task in two parts : i) unload modify data produced by the preceding task to the off-chip memory if need be, then load code and data required by the task into a local private memory; ii) execute the code without anymore access to the off-chip memory. Figure 1.7b sketches an example composed with three tasks (rectangles). Each task is divided in two parts, horizontal line boxes for memory accesses and dotted boxes for computation phases. This decomposition enforces composability in parallel execution by isolating memory access phases from execution ones. The WCET of the execution phase can then be computed in isolation. Hence, if coupled with a SPM, a software-managed memory, it increases the likelihood of predictability as memory are located in space and time. Lately, PREM became more and more popular, e.g. [MDC14; AWP15; BMV+15; MBB+15; BDN+16]. The main reason lies in the predictability improvement by isolating memory accesses. In addition, Light-PREM [MDC14] allows to automatically adapt any (legacy-)application to the PREM model.

Durrieu et al. [DFG+14] extend PREM with Acquisition Execution Restitution (AER) to en-

able this timing isolation principle with dependent tasks. AER [MNP+16] unloads the data produced by a task just after its execution in a third phase. Figure 1.7c displays an example composed with, again, three tasks (rectangles). Each task is divided in three parts, slashed lines boxes for *acquisition* phase, dotted boxes for *execution* phases, and backslashed lines boxes for *restitution* phases. Coupled with a SPM-based architecture, AER is very powerful to isolate multi-core interferences, hence it improves the predictability. With such an execution model, it is possible to load all the data into the SPM, execute the code without any further access to the main memory and then write back results to the off-chip memory.

1.4 Task and Inter-core communication

Aside from the execution model, the communication model defines how tasks exchange data, and how data transit on the hardware between computational units. First, most common communication mediums are either a bus or a Network on Chip (NoC). Second, the data transmission scheme organises sending and receiving point of data. When dealing with a task model that contains only independent tasks, this last transmission setting becomes optional. NoCs are out of the scope of this dissertation and will not be discussed anymore.

Communication medium A bus is limited by a bandwidth, also known as a bitrate in this specific case, which corresponds to the amount of data transmitted per time unit. The major source of hardware interferences originates on bus accesses generated by cores. Indeed, the bus connects every core together and with other devices (e.g. I/O devices, off-chip memory), leaving only one possible path for data transmission. Bus requests are scheduled with the help of an arbiter, that enforces a mutual exclusion on bus accesses as only one can access the bus at a time.

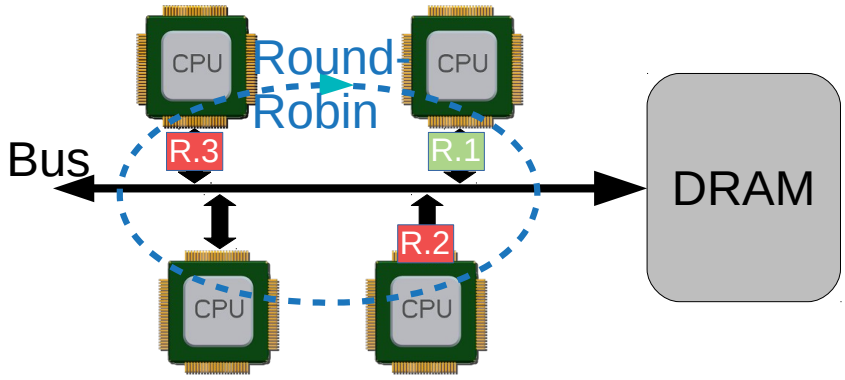


Figure 1.8 – Representation of the Round-Robin policy principle. Access order is $R.1$, $R.2$, $R.3$

The probably most used bus arbitration policy is the Round-Robin (RR) [Ens77]. Access requests are enqueued (one queue per core) and served in a round-robin fashion, pictured in Figure 1.8. In order to improve the predictability, a FAIR-RR arbitration policy [Ens77] limits the request access time for each core to the bus with a time sharing mechanism.

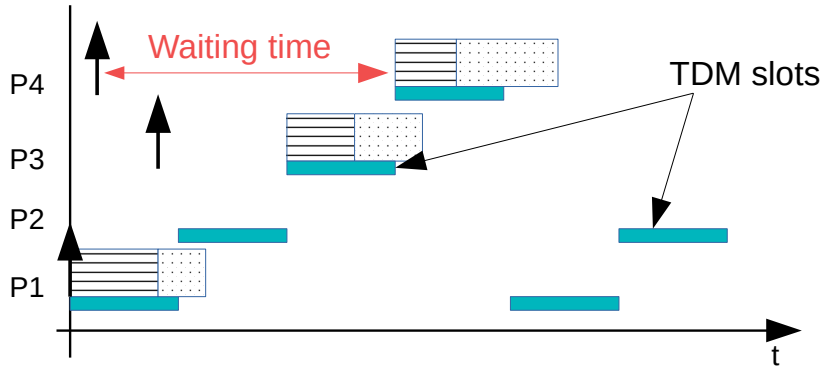


Figure 1.9 – Example of a TDMA arbitration scheme. Blue filled boxes are TDM slots, granting time for the corresponding core where horizontal-lined boxes represent memory accesses as in PREM in Figure 1.7b

Time Division Multiple Accesses (TDMA) arbitration policy [Ens77] enforces timing isolation and increases the predictability of the hardware [SCT10]. Figure 1.9 pictures a TDMA arbiter example. Each blue-filled boxes are TDM slots, and horizontal-lined boxes are memory accesses as in the PREM example from Figure 1.7b. Each core receives a time slot (TDM slot) in which the arbiter grants the access to the communication medium. TDM slots are guaranteed to not overlap, thus ensuring an access in mutual exclusion to the shared resource. The arbiter stores a table with the granting rules when booting up the platform. Then, the core is allowed to access the bus only when its time slot is active. TDMA arbiters suffer from an under-utilisation of hardware resources. A core must wait for its allocated time interval to process its request, and a time slot is wasted if the allocated core has no pending request to transmit [WS11]. To overcome this situation, Poseidon [SSP+14] tries to build the table with application structure knowledge and scheduling time slots in order to minimise the waiting time for the shared resource.

Round-Robin arbiters do not suffer from the drawbacks of TDMA ones, but they lack of predictability. Hence, FAIR-RR arbiters fill this gap by overcoming all drawbacks from vanilla Round-Robin arbiters and TDMA arbiters with the increase of predictability and no waste of resources time. FAIR-RR arbiters seem therefore to be the best candidate for multi-core architectures in the context of embedded real-time systems.

Data transmission scheme Data transmission schemes are architecture dependent. They depend on the hardware capabilities in terms of available hardware elements and configuration options.

When a Direct Memory Access (DMA) is available, data transmissions can occur while the core, which issued the transmission, is processing some computation; transmission are therefore *non-blocking* [GPP09]. If no DMA is available, cores initiate their requests and wait until the end of the transmission; transmissions are therefore *blocking* [PNP15].

When a SPM is available, if a producer and a consumer are mapped on the same core, then their shared data do not have to be transferred to the main memory. They can use the shared SPM, thus saving two bus requests; communications use shared memory with placement op-

timisation [RDP17]. However, choosing this type of communication must be carefully thought as data produced may be used by other tasks running on other cores. When only caches are available, the software can not decide what to store in an other cache than its own. Therefore, tasks running on cache-based architectures should exchange their data through the off-chip memory; communications use the *shared memory* [PNP15].

Lastly, if the hardware offers a global addressing policy along with SPMs, every memory cell has a different address SPM/off-chip/. . . , then producers can store data into the consumer SPM or the consumer can read data from the producer SPM; communications are *direct* [SPS+15].

1.5 Worst-case execution time estimation

Worst-Case Execution Time (WCET) estimation has been widely studied over the years. This section aims at giving the key techniques to perform this analyse and not an in depth study. For more information on WCET estimation, a survey is available in [WEE+08].

WCET estimation corresponds to an upper bound of any possible execution time for a software task on a hardware platform. WCET estimates properties include both safety and tightness. A safe WCET estimate is guaranteed above any possible execution of the program. While a tight WCET estimate limits the over-approximation (see Figure 3 from the Introduction).

When a WCET analysis considers a task executed on a dedicated hardware, without any other task, the WCET estimate is said to be in isolation from the system. Main studies identify WCET estimates in isolation and WCET estimates including interferences from other tasks.

From [WEE+08], there exists three major categories of WCET estimation techniques:

— Measurement based techniques: they run the software several times on the hardware (or a simulator) with different inputs to generate the worst-case behaviour. This method results in tight estimates by nature. But there is no guarantee that the worst-case behaviour is found, if the worst-case input is never exercised. Therefore this method is not safe for critical systems;

— Probabilistic methods: they construct a probabilistic curve from WCET measurements. They differ from the above-mentioned techniques in the number of runs which is lower. These methods aim at giving a worst-case execution time with a confidence probability. Tightness in results depends on the probabilistic law applied and safety can be discussed with the probabilistic confidence factor;

— Static methods: they are from far the most safe, but safety comes at the cost of a possible over-approximation. To generate tight and safe results, these analyses run at the binary level. It is usually divided in two steps to first account for the micro-architecture timing behaviour; i.e. data address analysis, cache analysis, branch predictor analysis, pipeline analysis and so on . . . . And second, a higher level analysis computes the worst-case execution path of the application and then the WCET estimate, e.g. Implicit Path Enumeration Technique (IPET).

Apply state-of-the-art WCET analysis techniques, with multi-core architectures, impose to guarantee the absence of interference caused by other cores, or further analyses are required to augment WCET estimates by accounting for these interferences. Kelter [Kel15] accounts for the contention on the bus. Potop et al. [PP13] add the communication cost into the IPET

problem. Ozaktas et al. [ORS13] computes the stall time induced by critical sections and add them to the IPET problem.

1.6 Real-time scheduling: a state of the art

Real-time scheduling determines the execution order of tasks on a processor. Then, targeting multi-core architectures, mapping algorithms determine on which core a task will run. Most of the times, scheduling on single-core architecture is NP-hard, e.g. with fixed-priority and sporadic tasks [EY17]. Mapping+scheduling on multi-core architectures is therefore classified as a NP-hard problem [CGJ96], because the mapping step reduces to the known bin-packing problem. For the sake of simplicity, when not explicitly stated in the following, a multi-core scheduling algorithm refers to both a mapping and a scheduling algorithm.

The literature on multi-core real-time scheduling is tremendously vast. Davis and Burns [DB11] sort scheduling algorithms in three main categories: i) partitioned, ii) global, and iii) hybrid. Following the classification of single-core schedulers, further review focuses on multi-core architectures where the main content focuses on partitioned (including static/off-line) scheduling, while other scheduling policies are summarised. More details on scheduling strategy and schedulability analysis are available in this very same survey [DB11].

1.6.1 Classification of single-core schedulers

The two first major categories of schedulers represent the construction moment of the selection order: *off-line* or *on-line.* The former, statically pre-computes off-line an execution sequence for each core. Then at runtime, a dispatcher executes the generated schedule. The latter executes *on-line*, and determines the next task to execute according to some criteria, the most common being assigned priorities and current execution state.

Priorities are assigned to tasks dynamically or statically (also known as fixed). Rate Monotonic (RM) [LL73] scheduling policy statically affects, at design time, higher priorities to tasks with shorter periods. In contrast, Earliest Deadline First (EDF) [LL73] determines priorities according to the current execution state of tasks and set higher priorities to unfinished tasks closer to their deadlines. Multiple other examples exist for fixed or dynamic priorities scheduling policies targeting single-core architectures, e.g. Deadline Monotonic (DM) [LW82] or Least Laxity First (LLF) [DM89].

On-line schedulers select the task to execute at run time upon task arrivals or terminations (except few particular cases, e.g. Least Laxity First [DM89] scheduling policy where decisions are taken at each time instant). Due to the nature of WCET that is likely to not happen, on-line schedulers are generally more flexible than off-line ones. But their implementation is more costly. Another solution is to combine off-line and on-line schedulers [CFL+05] to benefit from the flexibility and low implementation cost.

Because on-line schedulers construct the schedule at run-time, a schedulability analysis is mandatory, also known as a feasibility test. It statically proves that no task misses a deadline. In contradiction, off-line schedulers validate at schedule time that no task misses a deadline by building the schedule in advance [CC10; CC11].

Pre-emptive scheduling policies, e.g. [LL73], allow a task to be paused/resumed by an other task. Pre-emption depends on the priority of the current executing task. A higher priority task pre-empts a lower priority one. Pre-emptive schedules require a Real-Time Operating System (RTOS) to handle context switches when a task pre-empts another one. Between these two extreme cases, limited pre-emptive scheduling policy offers an alternative which limits the number of context switches but remain flexible [BBY13].

An infinite schedule does not need to be built for periodic task models. Instead, the analysis computes the minimum time a schedule needs to enter in a repetitive pattern. This interval, known as the *hyper-period*, corresponds to the Least Common Multiple (LCM) of all periods in the task set [GGC16]. This principle is extended by Puffitsch et al. [PNP15] for dependent task set where all dependent tasks have different periods.

1.6.2 Multi-core partitioned scheduling

Partitioned approaches group tasks into partitions. Each partition is assigned to a core and scheduling algorithms designed on uniprocessor can then be applied (e.g. RM [LL73], LLF [DM89]). Baruah et al [BB08] find the best partition set for sporadic tasks under constrained deadlines. Then EDF or RM from [LL73] will schedule each partition on a core.

Off-line schedule problems are solved with different types of algorithm depending on the wished level of optimality. Integer Linear Programming (ILP) [PNP15; GKC+15] and Constraint Programming (CP) [GKC+15; Per17] solvers formulate the problem as a series of constraints, then solving the constraint system results in an optimal solution according to an objective function. Satisfiability Modulo Theory (SMT) methods [TPG+14; GKC+15] also result in optimal schedules but logic constraints are applied instead of algebraic ones. Mapping a task set onto a multi-core is NP-hard [CGJ96], therefore such techniques do not well scale with large problems. It is a common practice to provide a heuristic aside from a method generating optimal results, for example in [BDN+16] both an ILP and a heuristic solving the same problem are given. Such heuristic targets the same goal but with a small possible over-approximation. Hence, the goal of the heuristic algorithm is to generate a close to optimal result within a much smaller solving time. In between, meta-heuristics aim at limiting the over-approximation to get closer to the optimal solution with a control on the solving time ; e.g. Particle Swarm Optimisation (PSO) [Zak13], genetic algorithms [PK06; WGW+14].

Different objectives drive scheduling algorithms. Among all the different possibilities, the literature offers to :

— minimise the overall schedule makespan in [YHZ+09; Per17],

— find a valid schedule (not specifically the shortest) [BDN+16],

— optimise the energy consumption [CFL+18],

— maximise throughput [CLC+09],

— minimise total communication cost [TPG+14].

In [KM08; TPG+14] a multi-step process schedules communication and tasks from SDF. They first augment the graph by adding nodes to model DMA transfers. Then a SMT solver outputs a schedule before allocating buffers for data transfer. In [TPM14], Tendulkar et al. add jobs pipelining to there SMT formulation from [TPG+14] to increase the throughput of the schedule while it was already present in Kudlur et al. work from [KM08].

Following works manage to capture the essence of the targeted hardware platform to generate more specialised schedules.

Alhammad and Pellizzoni [AP14] propose a heuristic to map and schedule a fork/join graph onto a multi-core architecture in a contention-free manner. They split the graph in sequential or parallel segments, and then schedule each segment. They consider only code and local data access in contention estimations, leaving global shared variables in the main external memory, where a worst concurrency scenario is assumed when accessing them.

Puffitsch et al. [PNP15] statically schedule periodical dependent tasks on many-core platforms. They account for the memory loading such as SPM, and places preloading time in case of caches in order to achieve a timing isolation process. They also allow to parametrise the objective function between the number of used cores, and a threshold of contention along the bus.

Becker et al. [BDN+16] propose an ILP formulation and a heuristic to schedule periodic sporadic independent tasks on one cluster of a Kalray MPPA processor. They systematically create a contention-free schedule.

Nguyen et al. [NHP17] statically schedule DAG on one cluster of a Kalray MPPA architecture. The objective is to build a cache-conscious schedule in which tasks sharing information are mapped on the same core, and in an order that will maximise cache lines reuse.

Skalistis and Simalatsar [SS17] build a partitioned off-line schedule from DAGs, and minimise the overall schedule makespan. Because an off-line schedule uses the WCET of tasks, which is likely to not happen, an on-line scheduler can fire a job earlier as long as off-line scheduling decisions are fulfilled.

1.6.3 Multi-core global scheduling

Global methods schedule tasks dynamically and globally on all cores, at a job level. Jobs from the same task can migrate between cores. They are often adaptations of uniprocessor scheduling strategies ; EDF becomes Global-EDF [DL78], RM becomes Global-RM [ABJ01].

Because global scheduling may imply a lot of context switches and migration, Anderson et al [ABD08] improve the initial work from Dhall et al. [DL78] on Global-EDF to limit the number of migrations.

When dealing with sporadic tasks, new tasks appear with a minimum interval during the execution of the real-time system. With an online global scheduler, to decide whether to accept the task or not, the system must recheck the schedulability in an on-line fashion. Therefore, the admission control algorithm requires an efficient schedulability analysis to limit the decision overhead. Two methods are proposed by Zhou et al. [ZLL+18]. They identified factors affecting the efficiency of RTA methods – i.e. unnecessary recalculation of WCRTs under Global-EDF. They also improved the computational method. However checking the schedulability of a system under Global-EDF and Global-RM with sporadic tasks remain pseudo-polynomial.

In [MHN+17], an on-line global scheduler decides which tasks to run in parallel. Decisions imply to know the memory demand, and the scheduler determine the amount of interferences it allows. If the scheduler detects more interferences than expected and above a threshold, then tasks are re-prioritised and pre-empted to drop down the contention.

1.6.4 Multi-core hybrid scheduling

Because partitioned approaches might under-utilise the hardware, and global scheduling techniques induce possible high overhead with migration costs, then hybrid scheduling methods try their best to improve these two factors of performance.

First, semi-partitioned approaches limit the fragmentation of spare capacity. They generally build an initial partition set which is assigned to a core but this assignation does not remain immutable, as following defined.

Cannella et al. [CBS14] build a semi-partitioned heuristic scheduler : First-Fit Decreasing Semi-Partitioned (FFD-SP). As opposed to regular partition algorithms, they allow a whole task to temporarily migrate into another partition.

Burns et al [BDW+12] give a method to compute partitions which will then be executed with EDF. The specificity is the possibility for a task to migrate when pre-empted. A task can then start on a core, and ends on another one, but returns to the initial core at the next firing.

Second, cluster approaches group processors with related properties (e.g. shared caches). Then a global EDF is applied on the group of cluster [SEL08].

Third, federated scheduling [LCA+14] group tasks depending on their processor utilisation. The idea is to identify heavy tasks from light ones, where heavy tasks load more the processor than light ones.

In [JGL+17], heavy tasks are considered on their own. Light tasks are federated and a federation is considered as sequential sporadic and scheduled at once.

1.6.5 Shared resource management on single-core and multi-core architectures

Shared resources represent anything that tasks are required to share with other tasks. On the hardware side, the processor core is the first obvious shared resource. But other devices also can be shared, e.g. bus, memory hierarchy, I/O devices, ... On the software side, shared variables are the primal shared resources. Because a shared resource usually only allows one access at a time, ordering access requests require an arbitration policy or a sharing protocol. Because a request might be delayed due to interferences with other ones, real-time systems require to compute the worst-case waiting time. This waiting time is commonly named *blocking time* and is added to issuing task's WCET to form the Worst-Case Response Time (WCRT) of the tasks [JP86]. The general term *blocking time* can be specialised depending on the context, when a task pre-empts (blocks) an other one, it is known as the *pre-emption delay*; when a task is delayed du to interferences on the communication medium, the blocking time is known as the *contention delay*.

Negrean et al. [NSE09] provide a method to compute the blocking time induced by concurrent tasks (pre-emption delay) in order to determine their response time. But some studies might choose to reduce [HPP09], or even avoid [SM08] interferences on shared resources leading to a decrease, or the removal, of the blocking time.

Aforementioned schedulers order accesses to the processor core. From Section 1.4, among other arbiters, RR and TDMA arbiters order accesses to the shared bus. A contention analysis is then defined to determine the worst case delay for a task to gain access to the resource (see [FAQ+14] for a survey). Some shared resources may directly implement timing isolation mechanism between cores, such as TDMA buses, making contention analysis straightforward. Further Chapters 3 and 4, in their respective related work sections, detail bus sharing mechanisms

and blocking time computations. Therefore, following policies from this section summarise key concepts in other hardware-oriented (e.g. cache, SPM) and general resource sharing (e.g. software objects).

On uniprocessor with priority-based scheduling, when a higher priority task pre-empts another, the cache replacement policy can evict lines useful to lower priority tasks. Then the resumed task can suffer from a blocking time to refill its evicted cache lines. In this case, the blocking time corresponds to the Cache Related Pre-emption Delay (CRPD) [AB11; ADM11]. In multi-core architectures, the CRPD for shared caches results in an important pessimism. Therefore, with cache partitioning, Suhendra et al. [SM08] fixes the CRPD to zero at the cost of reduced performance with a smaller cache area per task. SPM-based architectures can suffer from the same issue when a higher priority task evict useful data/code from the SPM for a lower pre-empted task. Whitman et al. [WDA+12] port the CRPD to SPM with the Scratchpad Pre-emption Delay (SRPD).

Other works on shared caches tend to minimise the interference. In [DLM13], the mapping of tasks minimise interference on L2 cache. In [NHP17] the scheduler decides pairs of task mapped on the same core contiguously scheduled that will maximise cache reuse. Hardy et al. [HPP09] reduce interference on shared L2 cache by identifying at compile time, what basic blocks are uniquely used. Then these blocks are not stored in the shared cache avoiding cache pollution.

Dealing with shared software objects is not new, and there now exists several techniques adapted from the single-core systems. Most of them are based on priority inheritance. In particular Jarrett et al. [JWA15] apply priority inheritance to multi-cores and propose a resource management protocol which bounds the access latency to a shared resource. For single-core architecture Priority Ceiling Protocol (PCP) [SRL90] and Shared Resource Protocol (SRP) [Bak91] are widely accepted as the most efficient technique to deal with software resource accessed in mutual exclusion. It is known to avoid priority inversion and limit the blocking time. The PCP policy defines a ceiling attached to each semaphore as the maximum priority among all tasks that can possibly lock the semaphore. The SRP is similar to the PCP, but it has the additional property that a task is never blocked once it starts executing. Both have analogous methods targeting multi-core architectures, MPCP [Raj12] and MSRP [GLD01]. MPCP extends the concept of blocking time to include also a remote blocking (when a job has to wait for the execution of a task of any priority assigned to another processor). MSRP separates local and global resources. For local resources, a classic SRP is used, while for a global resource, if the resource is already locked by some other task on another processor, then the task performs a busy wait (also called spin lock). A comparison of both MPCP and MSRP [GDL+03] showed that no method outperforms the other. MSRP is easier to implement with a lower overhead while MPCP has a higher schedulability bound.

1.7 Conclusion

This chapter presented a literature review on the different components required to design real-time systems. The task model section described the software with different level of expressiveness and abstraction from the seminal periodic task model to multi-rate CSDF graphs. Then, a classification of hardware platforms helped identifying predictable multi-core architec-

tures properties. Moreover, the execution model followed by the inter-core communication completed design possibilities, by describing how tasks are executed and how they communicate with each other. The consecration of these design choices lead to the estimation of the WCET. Then, a review of mapping and scheduling algorithms was presented to determine on which core and when to execute application tasks. Finally, hard real-time systems include software and hardware resources to which only a single-task can access. Therefore, a review of different methods and protocols presented how to integrate several sharing mechanisms. Next chapter will present the context of this dissertation with respect to the aforementioned material.

CHAPTER 2

GENERIC STATIC SCHEDULING FRAMEWORKS WITH WORST-CASE CONTENTION

As shown in previous chapter, real-time systems need performance and predictable hardware platforms. While multi-core architectures increase the computation power and so the performance, they also add a dose of uncertainty which impacts the predictability. Therefore, specific multi-core platforms, that also improve the predictability, generally enforce timing isolation with their design choices, e.g. Kalray MPPA [DVP+14]. Moreover, SPM-based multi-core architectures further improve the predictability [PP07; MB12], because the content of the SPM is decided at compile time. Nowadays, it is more and more common to see NoC connecting cores, but a bus remains a valid choice. A bus can be arbitrated with different policies where the balance between performance (e.g. Round-Robin arbiters) and predictability (e.g. TDMA arbiters) needs to be met. We believe that SPM-based multi-core architectures with a FAIR Round-Robin arbitrated bus should remain of major concern in the future.

Some existing works on multi-core scheduling consider that the platform workload consists of independent tasks, e.g. [BDN+16; CCR+17]. As parallel execution is the most promising solution to improve performance, we envision that within only a few years from now, real-time workloads will evolve toward parallel programs. The timing behaviour of such programs is challenging to analyse because they consist of *dependent* tasks interacting through complex synchronization/communication mechanisms. We believe that models offering a high-level view of the behaviour of parallel programs allow a precise estimation of shared resource conflicts. In this thesis, we assume parallel applications modelled as DAGs. This task model exhibits both dependency and concurrency. In addition with AER execution model and SPM-based architectures, memory accesses are explicit which increases the predictability of the whole system.

Scheduling for multi-core platforms was the subject of many research works, surveyed in [DB11]. We believe that static mapping of tasks to cores (partitioned scheduling) and non-preemptive time-triggered scheduling on each core allow to have control on hardware resource sharing, and thus allow to better estimate worst-case contention delays. This chapter introduces two scheduling frameworks extracted from the state-of-art, and used as initial reference for further scheduling methods in Chapter 3 and Chapter 4. The first one is an Integer Linear Programming (ILP) formulation which gives an optimal solution. The second technique is a heuristic algorithm which gives approximate results with a faster solving time. The included ILP formulation gives a non ambiguous description of the problem under study, and also serves as a baseline to evaluate the quality of the proposed heuristic technique. They both respond to the same scheduling problem and both use

The proposed scheduling techniques are empirically evaluated. As expected, the ILP is shown to not scale with large DAGs (scheduling is NP-hard [CGJ96]), thus requiring a heuristic. To validate the quality of the heuristic, the schedule length generated by the heuristic is compared to its equivalent baseline from the ILP formulation. Finally, the effect of schedule parameters on schedule length is discussed such as the duration of one slot of the round-robin bus.

The outline of this chapter is as follows: Section 2.1 presents the hardware platform and the computation of communication costs between a core and the off-chip memory. Section 2.2 describes the task model and the execution model used on tasks. Section 2.2.1 details how cores communicate with each other. Section 2.3 presents an example of the basic scheduling problem before introducing both the ILP formulation and the heuristic frameworks. Then, Section 2.4 gives some statistics and experiments picturing the efficiency of previous scheduling techniques.

2.1 Basic predictable multi-core architectures

Multi-core architectures with a private ScratchPad Memory (SPM) per core are a very attractive solution for executing time-critical embedded applications. They perfectly fit the need for performance, with the computational power of multi-cores, and the need for predictability, with the presence of SPMs. Examples of such architectures are the academic core Patmos [SBH+15] or the Kalray MPPA [DVP+14] (with local memory banks configured as *blocked*). Such a private memory allows, after having first fetched data/code from the main memory to the SPM, to perform computations without any access to the shared bus. Figure 2.1 depicts an abstraction of the multi-core architecture used all along this dissertation.

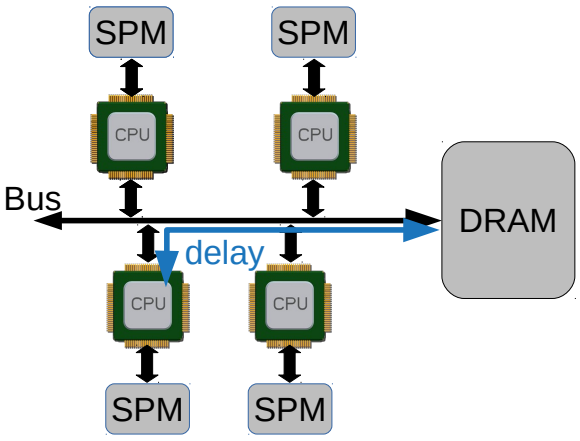


Figure 2.1 – A multi-core architecture abstraction

The shared bus is arbitrated using a FAIR-RR policy to benefit from more predictability over a classic Round-Robin [Ens77], and to avoid the loss of performance from TDMA arbiters. Access requests are enqueued (one queue per core) and served in a round-robin fashion. A maximum duration of T_{slot} is allocated to each core, to transfer D_{slot} data words to the external memory (a data word needs T_{slot}/D_{slot} time units to be sent). If a core requires more time

than T_{slot} to send all data, then the data are split in *chunks* to be sent in several intervals of length T_{slot} (see equation (2.1a)), plus some additional *remaining time* (see equation (2.1b)). If a full T_{slot} duration is not needed to send some data, the arbiter processes the request from the next core in the round. As an example in Figure 2.2, taking a D_{slot} of 1 data word, T_{slot} of 2 time units, and a core requesting a transfer request of 5 data words, results in two periods of duration T_{slot} and a remaining time of T_{slot}/D_{slot}.

In the worst case for each chunk, we assume all cores active and requesting access to the bus. This worst-case scenario can be improved (see Chapter 3). Then, a chunk will be delayed by $NbCores - 1$ pending chunks from the other cores (with $NbCores$ being the number of available cores), see equation (2.2). For each interfering core, the communication delay is augmented with a *waiting time*. This *waiting time* refers to the accumulated time (for all chunks), the core waits to access the shared bus to transmit a piece of data, equation (2.1c).

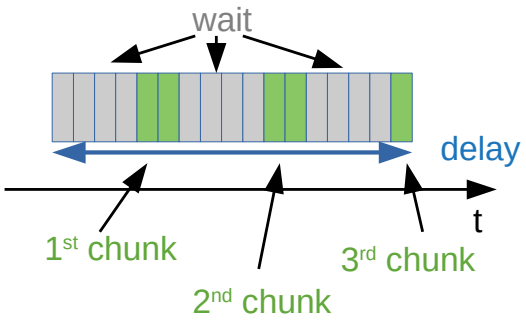


Figure 2.2 – Delay representation.
Configuration: $T_{slot} = 2$ time units, $D_{slot} = 1$ data-word, 3 cores
Request: 5 data words gives 3 chunks.
Each chunk is delayed by 2 interferences $\times\ T_{slot}$

Overall, equation (2.1d) derives the worst latency, delay, to transmit some data with a round-robin arbitration policy. This equation could be refined to account for DRAM access costs, as done in [KBC+14] with a constant value. But the impact on the presented method is negligible. Chapter 3 alleviates the worst-case contention in a contention-aware manner. And, Chapter 4 presents a contention-free version.

$$chunks = \lfloor data/D_{slot} \rfloor \tag{2.1a}$$

$$remainingTime = (data \bmod D_{slot}) \cdot (T_{slot}/D_{slot}) \tag{2.1b}$$

$$waitingSlots = \lceil data/D_{slot} \rceil \tag{2.1c}$$

$$delay = \underbrace{T_{slot} \cdot waitingSlots \cdot interf}_{\text{Total waiting time}} + \underbrace{T_{slot} \cdot chunks + remainingTime}_{\text{Total access time}} \tag{2.1d}$$

$$interf = NbCores - 1 \tag{2.2}$$

The round-robin arbiter is predictable as the latency of a request can be statically estimated, as long as the configuration of the arbiter (parameters T_{slot} and D_{slot}) and the amount of data to be transferred (*data*) are known at design time [KHM+13].

2.2 Software Model

This work considers applications modelled as Directed Acyclic task Graphs (DAG), in which nodes represent computations (tasks) and edges represent communications between tasks. Nevertheless, this work supports multiple DAGs with identical periods as it is. With different periods, the following methods are applicable at the job level on the hyper-period. A task graph G is a pair (V, E) where the vertices in V represent the tasks of the application. The set of edges E represents the data dependencies. An edge is present when a task is causally dependent on another, meaning the target of the edge needs the source to be completed prior to run. An example of a simple task graph is presented by Figure 2.3 with labelled edges corresponding to number of exchanged data words.

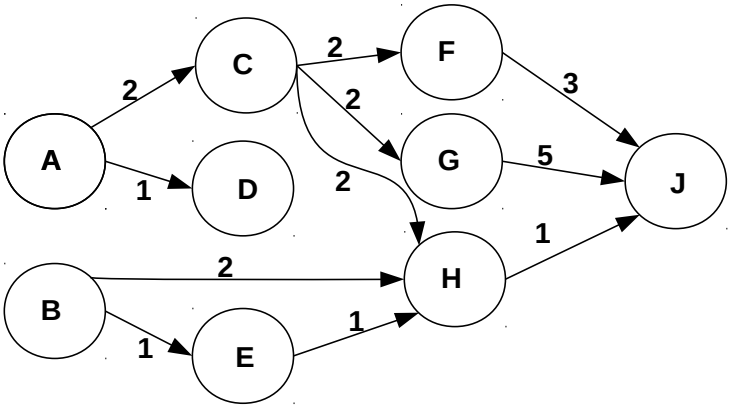


Figure 2.3 – Task graph example

Each task is then divided in three phases (or sub-tasks) according to the *Acquire-Execute-Release* semantics (AER), as first defined in [DFG+14]. The *acquire* phase reads/receives the mandatory code and/or data from main memory to SPM, such that the *execute* phase can proceed without accessing the bus. Finally, the *release* phase writes/sends the resulting data back to the main memory. In the rest of this document, *read* and *write* will refer to the *acquire*/*release* communication phases of tasks respectively.

The obvious interest of the *AER* semantics is to isolate the sub-tasks that use the bus. Therefore, contention analysis can focus only on these sub-tasks. For the sake of simplicity, this chapter considers that the entire code and data fit in the off-chip memory, and there is always enough space left in SPMs for loaded and produced data. Since the code in our experimental evaluation, is generally small and likely to be reused along the execution of the application, for simplicity reasons we assume that the code is preloaded in the SPM at startup. However, enabling code prefetching could be easily done, by including the size of the code of a task in the amount of data to be fetched by its *read* phase.

A task i is defined by a tuple $< \tau_i^r, \tau_i^e, \tau_i^w >$ to represent its *read*, *execute*, and *write* phases. An edge is defined by a tuple $e =< \tau_s^w, \tau_t^r, D_{s,t} >$ where τ_s^w is the *write* phase of the source task s, τ_t^r is the *read* phase of the target task t. $D_{s,t}$ is the amount of data exchanged between s and t.

The WCET of the *execute* phase, noted C_i, can be estimated in isolation from the other tasks considering a single-core architecture, because there is no access to the main memory (all the required data and code have been loaded into the SPM before the task's execution). The communication delay of the *read* and *write* phases (respectively noted $delay_i^r$ and $delay_i^w$) depend on several factors: amount of data to be transferred, number of potential concurrent accesses to the bus. For this initial chapter, a worst-case contention is assumed. Worst-case contention assumes that all cores are requesting the bus at the same time instant. This will represent a baseline for further improvements and is quite generic in the literature.

2.2.1 Inter-core communication

In this chapter, communications are considered *blocking* and *indivisible*. The sender core initiates a memory request, then waits until the request is fully complete (*blocking* communications), i.e. the data is transferred from/to the external memory. There is no attempt to reuse processor time during a communication by allocating the processor to another task (*indivisible* communication). Execution on the sending computing core is stalled until communication completion.

All communications go to the main memory, the *read* phase fetches data from the main memory to the SPM and the *write* phase sends data from the main memory to the SPM.

The conjunction of *blocking* communication and *all communications via external memory* constraint maximises the waiting time spent to communicate data between tasks. Hence, it represents the worst-case behaviour regarding the impact of communications on the schedule length.

While these assumptions are met in Chapter 3, Chapter 4 relaxes them with the presence of a DMA engine and a dual-ported SPM in the architecture. Both hardware modifications enable *non-blocking* communications ; i.e. the core is not stalled.

2.3 Scheduling framework

With multi-core architectures, static scheduling algorithms aim at finding a mapping (where to execute tasks), and a schedule per core (when to fire tasks). Hard real-time systems need strong guarantees on the timing behaviour. Such guarantees are offered by construction with static off-line scheduling algorithms, as opposed to dynamic scheduling policies which require a schedulability analysis. When a static scheduler finds a valid schedule, as long as the WCET are safe, no deadline misses is guaranteed.

The proposed scheduling policy co-schedules and maps both computation and communication phases. A one-step strategy avoids a source of pessimism induced by a multi-step method, as in [TPG+14] where mapping and scheduling are performed in separate processes.

This section presents the scheduling framework used all along this document. Both an ILP formulation and a heuristic targeting the same problem are detailed. They are further improved

in next chapters (Chapter 3 for contention-aware schedules, Chapter 4 for a contention-free with limited-size SPM). The main outcome of both techniques is a static mapping/scheduling for one single application represented by a DAG. According to the terminology given in [DB11], the proposed scheduling techniques are partitioned, time-triggered and non-pre-emptive and operate at the task level.

The scheduling framework, from this chapter, is not a contribution on its own. It corresponds to an aggregation of principles extracted from the literature. Moreover, in this dissertation the following constraints and algorithms are used as baseline and are later refined according to the solved problem. In this case, they represent a simple mapper+scheduler generating schedules as presented in the following example.

2.3.1 Example

Table 2.1 – WCET values used for the task graph example in Figure 2.3

Task	A	B	C	D	E	F	G	H	J
WCET in time units	5	6	10	15	15	15	15	10	5

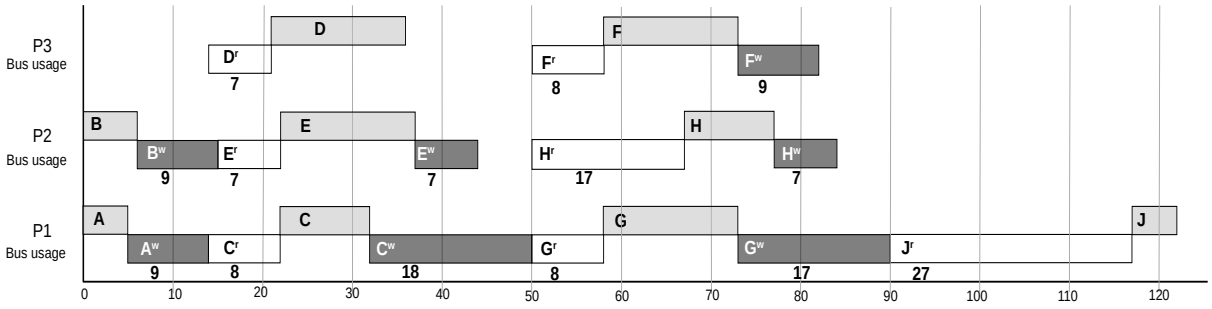


Figure 2.4 – Resulting schedule for task-graph from Figure 2.3 with a worst-case contention policy targeting a tri-core architecture equivalent to Figure 2.1. Overall makespan is 122 time units.

Figure 2.4 presents the results of a static scheduler using following ILP formulation. Inputs include the task graph from Figure 2.3 with WCET estimate values from Table 2.1. To compute communication delays, a worst-case contention policy is applied targeting a tri-core architecture equivalent to Figure 2.1. In Figure 2.4, the X-axis represents the time in *time units* starting at time 0 to the overall makespan of the schedule, here 122 time units. The Y-axis corresponds to processors timeline and bus usage timeline. Notice that there is only one bus on the targeted architecture, but three timelines on the resulting figure are clearer.

Three grey levels identify each phase of each task : white boxes for read phases, light grey boxes for exec phases, and dark grey boxes for write phases. Exec phases are placed on a processor core timeline, while communication phases are placed under on a bus usage timeline. Read/Write phases include both waiting time and effective bus access time. All three phases are contiguously mapped on the schedule (a phase starts right after its preceding one) and ordered according to the AER execution model.

On a processor's timeline, no execution phases overlap in time. The execution order is determined by the scheduler according to dependencies found in the graph. Thus there exist other solutions than this one – e.g. tasks G and D could be executed in reverse order – but no other resulting schedules can have a shorter overall makespan which is 122 time units here.

Fusing the two bus usage timelines results with between 0 and 3 communication phases activated at the same moment in time. But communication phases do not access the bus in parallel, they are interleaved according to a mutual exclusive principle of buses arbitrated with a FAIR-RR policy. The worst-case contention effectively arises when 3 communication phases are effectively active at the same time ; e.g. write phases C^r, E^r, and D^r. Numbers under each communication phase correspond to delays to transmit the data from the SPM to the main memory. Equation (2.1d), introduced in Section 2.1, computes this delay using a worst-case concurrency (equation (2.2)) and $T_{slot} = 3,\ D_{slot} = 3$; e.g. write phase G^w transmit 5 tokens, $delay^w_G = \underbrace{3 \cdot 2 \cdot (3-1)}_{\text{Total waiting time}} + \underbrace{3 \cdot 1 + 2}_{\text{Total access time}} = 17$ time units.

In Figure 2.4, there is no overlapping in time between a bus usage and an exec phase on the same core, therefore enforcing blocking communications.

2.3.2 Integer Linear Programming (ILP) formulation

An Integer Linear Programming (ILP) formulation consists of a set of integer variables, a set of constraints and an objective function. Constraints describe the problem to be solved in the form of linear inequalities and equalities [1]. Solving an ILP problem consists in finding a valuation for each variable satisfying all constraints with the goal of minimising/maximising the objective function. When scheduling and mapping a task graph on a multi-core platform, the objective is to minimise the overall schedule makespan.

Table 2.2 summarises the notations and variables used in this ILP formulation. Uppercase elements correspond to constants, lowercase elements are variables of the ILP formulation. For a concise presentation of constraints, the two logical operators $\vee, \wedge$ are directly used in the text of constraints. These operators can be transformed into linear constraints in order to properly use ILP solvers using simple transformation rules from [BD07], and given in Table 2.3.

Big-M notation The technique named *nullification* allows to activate or withdraw a constraint depending on the value of a binary variable. To apply a nullification on a constraint within an ILP system, a portable choice lies in the big-M notation [GNS09] [2].

As an example, if a constraint $x \leq z$ must be satisfied to validate the model only if a binary variable y is equal to 1 then the inequalities is written: $x \leq z + \mathcal{M} \cdot (1-y)$. If $y = 1$ then $\mathcal{M} \cdot (1-1) = 0$ and $x \leq z + 0$ holds. Otherwise $y = 0$ and $x \leq z + \mathcal{M}$ still holds if $\mathcal{M} = \infty$ The value of $\mathcal{M}$ is very important as it must be greater than any possible value in the model. But computers do not code ∞, so, one could use the biggest integer available on the hardware running the solver.

1. An equality can be defined as 2 inequalities, but lowering the verbosity is always appreciable

2. In the commercial CPLEX solver from IBM, the *indicator* feature allows to activate a constraint depending on a binary variable. But this feature is not available on other solver such as LP solve or Gurobi. Its usage is therefore withdrawn in this document for genericity purposes.

Table 2.2 – Notations & ILP variables

Sets	T	the set of tasks
	P	the set of processors/cores
Functions	$predecessors(i)$	returns the set of direct predecessors of task i
	$successors(i)$	returns the set of direct successors of task i
Constants	C_i	task i execute phase's WCET computed in isolation as stated in Section 2.2
	$DELAY_i^r$ $DELAY_i^w$	task i *read*, *write* phases' WCET from Equations (2.1d) and (2.2)
Int. var.s	Θ	schedule makespan
	ρ_i^r, ρ_i^e, ρ_i^w	start times of *read*, *execute*, and *write* phases of task i
Bin. var.s	$p_{i,c} = 1$	task i is mapped on core c
	$m_{i,j} = 1$	tasks i & j are mapped on the same core
	$a_{i,j}^{ee} = 1$	represents the causality between task i and task j, in the sense $\rho_i^r \leq \rho_j^r$ The precision of concerned type of phase, ee for two execute phases, is used here to remain consistent with ILP formulation from next chapters
	$am_{i,j}^{ee} = 1$	same as $a_{i,j}^{ee}$ but on the same core

Table 2.3 – Transformations of $\vee, \wedge$ to linear inequalities where a, b, c are binary variables

$c = a \wedge b$	$c + 1 \geq a + b$	$c \leq a$	$c \leq b$
$c = a \vee b$	$c \leq a + b$	$c \geq a$	$c \geq b$

In multi-core scheduling, the makespan of a sequential schedule on one core bounds the multi-core overall schedule makespan – i.e. a multi-core schedule with a higher response time than on a single-core is not relevant. The sequential schedule makespan is a good candidate value for the big-M constant. Equation (2.3) then computes the sum of WCETs and delays, which is the worst scenario that can arise.

$$\mathcal{M} = \sum_{i \in T} (C_i + DELAY_i^r + DELAY_i^w) \tag{2.3}$$

Objective function The goal is to minimise the makespan of the schedule, that is minimising the end time of the last scheduled task. The objective function, given in equation (2.4a), is to minimise the makespan Θ. Equation (2.4b) constrains the completion time of all tasks (starting of *write* phase, ρ_i^w, plus its WCET, $DELAY_i^w$) to be inferior or equal to the schedule makespan.

$$minimize\ \Theta \tag{2.4a}$$

$$\forall i \in T; \rho_i^w + DELAY_i^w \leq \Theta \tag{2.4b}$$

Problem constraints Some basic rules of a valid schedule are expressed in the following equations. Equation (2.5a) ensures the unicity of a task mapping. Equation (2.5b) indicates if two tasks are mapped on the same core. Notice that Equation (2.5b) doesn't need to be added for all tasks, but an optimisation on the number of constraints would be to consider that $m_{i,j} = m_{j,i}$. This optimisation will then decrease the solving time without sacrificing the quality of results.

Variable $a^{ee}_{i,j}$ represents the causality of the two tasks i, and j in the sense τ^e_i is scheduled before τ^e_j, thus Equation (2.5c) enforces a mutual exclusion in the causal order of phases. One of the $a^{ee}_{i,j}, a^{ee}_{j,i}$ must be equal to 1, but both can not be equal to 1. In the case of $\tau^e_i = \tau^e_j$, both cases $a^{ee}_{i,j} = 1$, or $a^{ee}_{j,i} = 1$ are possible but the solver will have to chose one depending on the remaining state of the system. However, if such a case ($\tau^e_i = \tau^e_j$) should arise in a multi-core environment, it means that tasks i and j are mapped on different cores, and the aware reader can notice that in the following ILP formulation, variables $a^{ee}_{i,j}/a^{ee}_{j,i}$ incidences are therefore limited.

Finally Equations (2.5d) unifies Equations (2.5b) and (2.5c) to order *exec* phases only on the same core.

$$\forall i \in T; \quad \sum_{c \in P} p_{i,c} = 1 \tag{2.5a}$$

$$\begin{aligned} &\forall (i,j) \in T \times T; i \neq j, \\ &m_{i,j} = \sum_{c \in P} (p_{i,c} \wedge p_{j,c}) \end{aligned} \tag{2.5b}$$

$$\forall (i,j) \in T \times T; i \neq j, a^{ee}_{i,j} + a^{ee}_{j,i} = 1 \tag{2.5c}$$

$$\forall (i,j) \in T \times T; i \neq j, am^{ee}_{i,j} = a^{ee}_{i,j} \wedge m_{i,j} \tag{2.5d}$$

Read-execute-write semantics constraints Equations (2.6a) and (2.6b) constrain the order of all phases of a task to be *read* phase, then *exec* phase, then *write* phase. Equations (2.6a) and (2.6b) impose all three phases to execute contiguously without any delay between them. The start time of the *execute* phase of task i (ρ^e_i) is immediately after the completion of the *read* phase (start of *read* phase ρ^r_i + communication cost $DELAY^r_i$). Similarly, the *write* phase starts (ρ^w_i) right after the end of the *execute* phase (start of *read* phase ρ^e_i + WCET C_i).

$$\forall i \in T, \rho^e_i = \rho^r_i + DELAY^r_i \tag{2.6a}$$

$$\forall i \in T, \rho^w_i = \rho^e_i + C_i \tag{2.6b}$$

Absence of overlapping on the same core Equation (2.7) forbids the overlapping of two tasks when mapped on the same core by forcing one to execute after the other. When i and j are mapped on the same core, and task i is meant to be scheduled before task j, $am^{ee}_{i,j} = 1$. Then read phase of j (ρ^r_j) must start after the write phase of i completes ($\rho^w_i + DELAY^w_i$). If not mapped on the same core, or tasks are scheduled in reverse order, then $am^{ee}_{i,j} = 0$ and the constraint holds with the help of the $\mathcal{M}$ notation.

$$\begin{aligned} &\forall i, j \in T \times T; i \neq j, \\ &\rho^w_i + DELAY^w_i \leq \rho^r_j + \mathcal{M}\ (1 - am^{ee}_{i,j}) \end{aligned} \tag{2.7}$$

Data dependencies in the task graph Equation (2.8) enforces data dependencies by constraining all tasks to start after the completion of all their respective predecessors. For a *read* phase (ρ^r_i) its predecessor is the *write* phase ($\rho^w_j + DELAY^w_j$) of the task producing the corresponding data ($j \in predecessors(i)$).

$$\forall i \in T, \forall j \in predecessors(i); \quad \rho^w_j + DELAY^w_j \leq \rho^r_i \tag{2.8}$$

Computing communication phases interference This chapter assumes a worst-case contention model, equation (2.1d) allows to compute the communication cost of each read and write phases statically. Thus, $DELAY^\chi_i$ with $\chi \in \{r, w\}$ and $i \in T$ is a constant in the ILP formulation.

2.3.3 Forward List Scheduling algorithm

Due to the NP-hardness [CGJ96] of the scheduling problem, the afore-described ILP formulation does not scale with large use cases. A heuristic algorithm helps support them at the cost of a possible over-approximation. In the late 60's, Graham [Gra66] proposed the List Scheduling (LS) algorithm to minimise the overall schedule makespan of a set of jobs on an homogenous parallel machine. This greedy approximation algorithm was later shown to perform quite well [CSS98] in terms of quality of results (over-approximation compared to the optimal is small) and with a much lower solving time.

LS algorithms first orders input elements, then add them one by one in the schedule without backtracking. They are classified in two types, forward or backward. While the former starts a new task as soon as possible, the later works from the deadline of a job and plans for just-in-time completion order. This dissertation presents a Forward List Scheduling (FLS) method in order to minimise the schedule length.

This chapter considers two sorting algorithms, all respecting dependencies. Scheduled elements are sorted with classic vanilla algorithms to walk-through the graph: i) Depth First Search (DFS), and ii) Breath First Search (BFS). For all two sorting solutions, we used the element memory footprint as tie breaking rule (larger footprint to be scheduled first). But, we could not find any sorting algorithm that outperforms the other one, so we generate two schedules each resulting from one sorting algorithm, and we select the best one as the heuristic's solution. Chapter 4 briefly discusses the importance of different algorithms in its experiment section (Section 4.6.4).

ALGORITHM 2.3.1: Forward list scheduling

Input : A task graph G $= (\mathsf{T}, \mathsf{E})$ and a set of processors P
Output : A schedule

```
1  Function ListSchedule(G = (T, E), P)
2      Elist ← BuildListElement(G)
3      Qready ← TopologicalSortNode(Elist)
4      Qdone ← ∅
5      schedule ← ∅
6      while t ∈ Qready do
7          Qready ← Qready \{t}
8          Qdone ← Qdone ∪{t}
           /* tmpSched contains the best schedule for the current task */
9          tmpSched ← ∅ with makespan = ∞
10         foreach p ∈ P do
11             copy ← schedule
12             map t on p in copy
13             ScheduleElement(copy, Qdone, t, p)
14             tmpSched ← min_makespan(tmpSched, copy)
15         schedule ← tmpSched
16     return schedule
```

ALGORITHM 2.3.2: Build list scheduled element

Input : A task graph G $= (\mathsf{T}, \mathsf{E})$
Output : A list

```
1  Function BuildListElement(G = (T, E))
2      return {τ_i^e | i ∈ T}
```

The FLS algorithm is sketched in Algorithm 2.3.1. It uses the task graph as input, extracts scheduled elements (line 2), sorts them to create a list (line 3), and then a loop iterates on each element while there exists one to schedule (lines 6-15). The algorithm schedules an element on a core (line 13), and then selects the core where the mapping+scheduling minimises the overall makespan (line 14).

In this first version, scheduled elements include the whole task at once which is summarised in the *exec* phase (Algorithm 2.3.2) for clarity regarding algorithms in following chapters. Besides, only *exec* phases need to be mapped on a core, communication phases *de facto* initiates from this same core and map on the unique bus.

Scheduling an element Algorithm 2.3.3 sketches the method to determine the start time of the considered element (here all the three phases at once). This heuristic uses an *As Soon As Possible* (ASAP) strategy when scheduling an element. It tries to schedule the element as early as possible on every processor.

First, each element must start after all their causal predecessors (line 2). Then, lines 3-7 enforces that no tasks overlap their execution/communication on the same core at the same time. Finally, lines 8 and 9 fulfil the AER requirements.

ALGORITHM 2.3.3: Schedule an element

Input : the list of scheduled element, the current element to schedule, the current core
Output : The input schedule modified with the newly added element

1 **Function** `ScheduleElement`$(Qdone, cur_elt, cur_p)$
2 $\quad \rho^r_{cur_elt} \leftarrow max_{e \in pred(cur_elt)}(\rho^w_e + DELAY^w_e)$
3 $\quad siblings \leftarrow \{e \in Qdone | e \; is \; mapped \; on \; cur_p\}$
4 $\quad$ `sort`$(\{(e, e') \in siblings | e \neq e' \wedge \rho^r_e < \rho^r_{e'}\})$
5 $\quad$ **foreach** $e \in siblings$ **do**
6 $\quad\quad$ **if** e *and* cur_elt *are active at the same time* **then**
$\quad\quad\quad$ `/* postpone the start of the current element */`
7 $\quad\quad\quad \rho^r_{cur_elt} \leftarrow \rho^w_e + DELAY^w_e$
8 $\quad \rho^e_{cur_elt} \leftarrow \rho^r_{cur_elt} + DELAY^r_{cur_elt}$
9 $\quad \rho^w_{cur_elt} \leftarrow \rho^e_{cur_elt} + WCET_{cur_elt}$

2.4 Experiments

Experiments were conducted on synthetic task graphs generated using the Task Graph For Free (TGFF) [DRW98] graph generator. Due to the intrinsic complexity of solving the scheduling problem using ILP, we need for small task graphs such that the ILP is solved in reasonable time. TGFF is used when there is a need to generate a large number of task graphs with a wide range of topologies. It is first used to evaluate the quality of the heuristic against the ILP formulation. This validation step shows the over-approximation induced by the heuristic is acceptable. The second experiment shows the impact of the T_{slot} hardware property on the schedule.

We generated two sets of task graphs: one with relatively small task graphs (referred to as STG), and another with bigger task graphs (referred to as BTG). With the latest version of the TGFF task generation software, the generator builds task graphs with chains of tasks with different lengths and widths, including both fork-join graphs and more evolved structures (e.g. multi-DAGs).

The resulting parameters of both sets are presented in Table 2.4. The table includes for both sets the number of task graphs, their number of tasks, the width of the task graph, the range of WCET values for each task[3] and the range of amount of exchanged data in bytes between pairs of tasks. The TGFF parameters for STG (average and indicator of variability) are set in such a way that the average values for task WCETs and volume of data exchanged between task pairs correspond to the analogous average values from real applications found in the STR2RTS benchmarks suite [RP17].

All reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). In all experiments T_{slot} is explicit, and a transfer rate of one data word (8 bits) per time unit is used.

3. The reader may notice that the WCET average value is not perfectly in the middle of the min and max values. This is due to the generation of random numbers in TGFF (pseudo-random, not perfectly random) combined to the limited number of values generated.

Table 2.4 – Task-graph parameters for synthetic task-graphs

	#Task-graphs	#Tasks <min,max,avg>	Max. width <min,max,avg>	WCET	Exchanged data
STG	50	5, 69, 22	3, 17, 8	$[5; 6000[$	$[0; 192]$
BTG	1000	18, 447, 150	2, 23, 9	$[5; 6000[$	$[0; 200]$

2.4.1 Scalability of the ILP formulation

Solving an ILP problem for a mapping/scheduling problem on multi-cores is known to be NP-hard [CGJ96]. Thus, the running time of the ILP formulation is expected to explode as the number of tasks grows. To evaluate the scalability of the ILP formulation with the number of tasks, a large number of different configurations is needed, explaining why we used synthetic task graphs for the evaluation. For each task graph in set STG we vary the number of cores in $\{2, 4, 8, 12\}$ values and vary T_{slot} in interval $[1; 10]$. With those varying parameters, the total number of scheduling problems to solve is $50 \cdot 4 \cdot 10 = 2000$. The ILP solver used is CPLEX v12.7.1 [4] with a timeout of 11 hours.

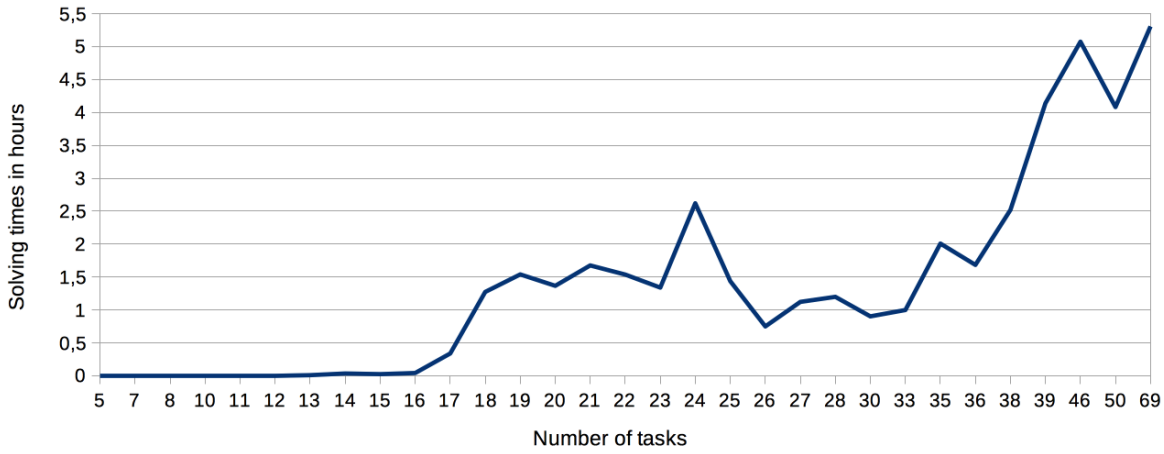


Figure 2.5 – Scalability of ILP formulation (synthetic task graphs / STG)

Figure 2.5 draws the evolution of the solving time depending on the number of tasks per graph. The solving time corresponds to the average value among graphs with the same number of tasks. As expected, when the number of tasks grows, the average solving time explodes, thus motivating the need for a heuristic that produces schedules much faster. As noticeable from Figure 2.5, the average solving time does not reach the timeout of 11h, because for some test cases the solver proves the infeasibility and stops quickly. Some other test cases include few linear chains of tasks and the solver quickly finds an optimal solution and stops. But still,

4. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

the number of tasks is of major influence regarding the solving time.

2.4.2 Quality of the heuristic compared to the ILP

The following experiments aim at estimating the gap between makespans of schedules generated by the heuristic (see Section 2.3.3) opposed to solutions found by the ILP formulation (see Section 2.3.2). The gap is expected be small. To perform the experiments we used the 50 task-graphs from the STG task set with the same parameters' variation as previously: number of cores $\in \{2, 4, 8, 12\}$ and $T_{slot} \in [1; 10]$. The heuristic is implemented in C++ and CPLEX was configured with a timeout of 11 hours.

Table 2.5 – Degradation of the heuristic compared with the ILP (synthetic task graphs / STG)

% of exact results (ILP only)	degradation <min,max,avg> %
73%	0%, 22%, 2%

Table 2.5 summarises the results. The first column of Table 2.5 presents the percentage of exact results the ILP solver is able to find in the granted time. We only refer to the exact solutions for the comparison, as the feasible ones (i.e not exact) might bias the conclusion on the quality of the heuristic compared to the ILP. Therefore, remaining 27% include feasible solutions as well as problems where timeout was reached (neither optimal solution found, nor proved infeasible). The next column presents the minimum, maximum and average degradation in percent, computed using makespans with equation (2.9).

$$degradation = \frac{heuristic - ILP}{ILP} \tag{2.9}$$

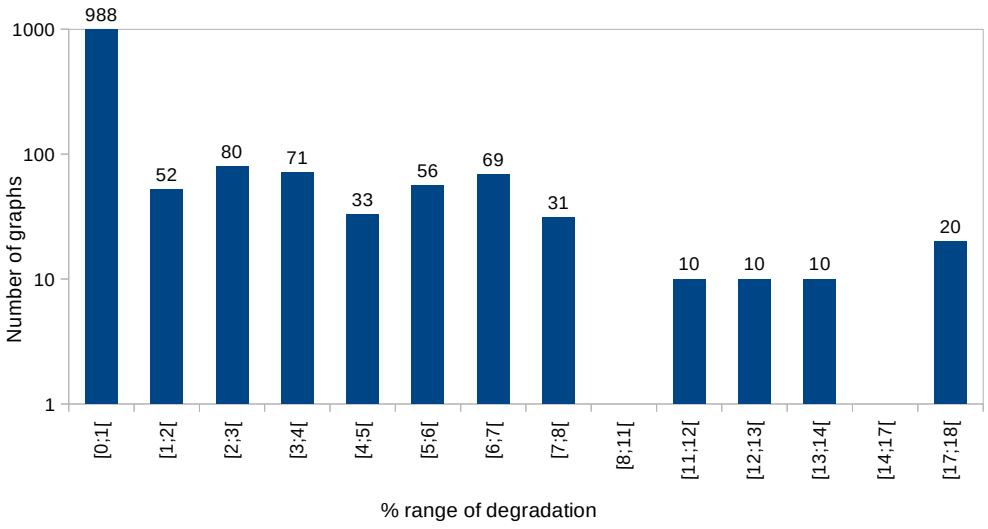


Figure 2.6 – Distribution of the degradation of the heuristic against the ILP formulation using STG task set. (logarithmic scale)

The average degradation is low, which means the heuristic has acceptable quality. A deeper analysis of the distribution of degradation, Figure 2.6, shows that 95% of the heuristic schedules are less than 10% worse than the ILP formulation solutions. We also observed a schedule generation time far much lower for the heuristic than the ILP solving time on the STG task set (maximum 0.5 second, average 0.08 second).

The influence of task sorting has a significant impact on the heuristic output. We choose a topological order with random tie breaking as stated in Section 2.3.3. These sorting algorithms explain the under-performance of 22% on the worst-case. No comparative study on sorting algorithms is included here, this is done in Chapter 4 with the addition of a third sorting technique.

2.4.3 Impact of T_{slot} on the schedule

Finally, with the heuristic, we studied the influence of the duration T_{slot} on the overall makespan, assuming the overhead negligible when switching between slots. We choose to remain on synthetic task graphs to benefit from a wider range of different test cases. Here the BTG task set is employed. For each graph, we generated three versions of the same topology but with different amounts of exchanged data between tasks. First case concerns few exchanged data $[0; 20]$, then reasonable amount of data $[20; 100]$ and large amount of data $[100; 200]$. With duration T_{slot} in the range $[1; 40]$, it covers all scenarios to exchange data in one or several chunks.

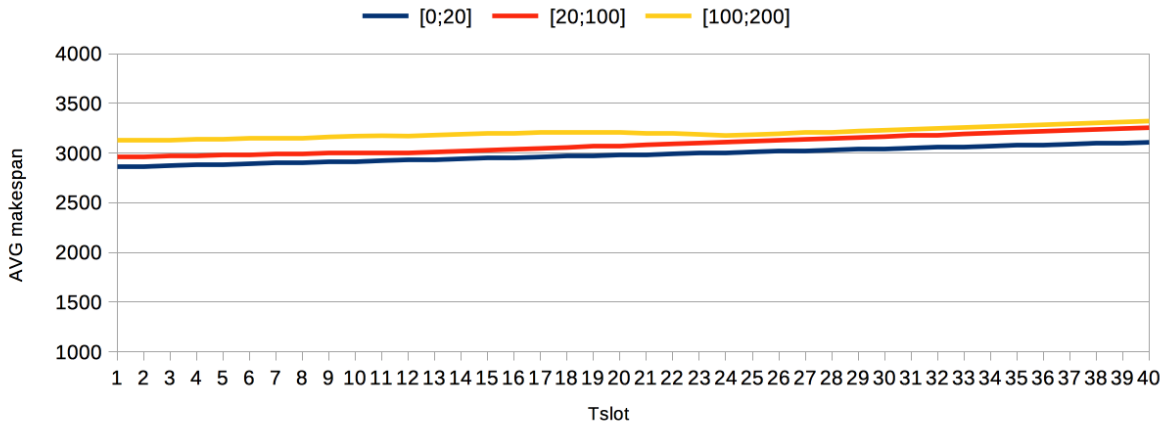


Figure 2.7 – Average makespan when varying T_{slot} (synthetic task graphs / BTG)

Results are presented in Figure 2.7 where the three curves correspond to the average makespan of each category over the value of T_{slot}. We observe that T_{slot} has a very little impact on schedule makespan. The exposed results confirm that it is a better choice to keep this T_{slot} small to reduce the waiting time between each slot even if there are several chunks. This allows small packets of data to be handled faster when in competition with bigger packets.

2.5 Conclusion

This chapter presented the design choice for the remaining chapters of this document. First the hardware design introduces the SPM-based multi-core architecture with cores connected through a bus arbitrated by a FAIR Round-Robin policy. We believe that these choices are valid as existing architectures are available for research studies with, for example, the academic core Patmos [SSP+11] and the commercial platform Kalray MPPA [DVP+14].

The software design includes parallel applications represented by DAGs. In conjunction with an AER execution model, memory accesses are isolated from computation in different phases. This separation allows to compute the WCET of computation phase in isolation as the computing core do not suffer from interferences from other cores. Timing isolation of bus accesses is, therefore, enforced without sacrificing performance and expressiveness. Moreover, the complexity of analysing real-time systems is decreased, while the accuracy of results is increased.

Initial contention model depicts a worst-case scenario where all contending cores try to access the shared resource – i.e. the bus – at the same time. Even if very pessimistic, this represents a baseline open for improvements with contention awareness from Chapter 3. In addition to the communication model, blocking bus accesses represent a worst-case scenario which is improved in Chapter 4.

Then, two scheduling techniques allow to build static schedules that are partitioned, time-triggered and non-pre-emptive. First, an ILP clearly states the problem under study. And by generating optimal schedules, ILP solvers results serve as baseline for approximate algorithm. Second, a heuristic algorithm based on list scheduling generates approximate schedules with the same general constraints than the ILP formulation.

Empirical evaluation showed that solving scheduling problem with exact method, ILP formulation, does not scale with large problems, as expected. For larger test cases an approximate method is required with a heuristic algorithm. Due to the intrinsic over-approximation of heuristic methods based on list scheduling, the quality of their results is compared with an exact method. Experiments revealed that the degradation of heuristic results does not exceed 22% with an acceptable average value of 2%.

Then, an empirical evaluation showed that the T_{slot} parameter has negligible impact over the schedule length.

The code of this scheduling framework is available at `https://gitlab.inria.fr/brouxel/methane`.

Next chapter relaxes the assumption on the contention scenario used in this chapter. It details how to compute the effective amount of interference at schedule time.

CHAPTER 3

COMPUTING THE PRECISE CONTENTION TO BUILD CONTENTION-AWARE SCHEDULES

This chapter marks the first main contribution of this dissertation: a mapping/scheduling strategy featuring bus *contention awareness*. Again, this method applies to SPM-based multi-core platforms where a round-robin bus connects cores and off-chip memory, as described in Section 2.1. The scheduling techniques, detailed in Chapter 2, employed a worst-case contention, which was a safe bound (but pessimistic) for the access latency: *NbCores - 1* contending tasks can access the bus (with *NbCores* as the number of available cores). The scheduling method in this chapter takes into consideration the application structure and information on the schedule under construction. The scheduler is then able to precisely estimate the *effective* contention by counting the worst-case degree of interference, which increases the accuracy of worst-case bus access latencies.

Like in Chapter 2, the proposed scheduling strategy generates a non pre-emptive time-triggered partitioned schedule with blocking communications. Again, both an ILP formulation and a heuristic algorithm model the scheduling problem, and all elements of the context remains identical to this same previous chapter – i.e. hardware, software, execution and communication models, and SPM assumptions.

The proposed scheduling techniques are evaluated experimentally. The schedule's length generated by the heuristic algorithm is compared to its equivalent baseline scheduling technique, from Chapter 2, accounting for the worst case contention. The experimental evaluation also studies the interest of allowing concurrent bus accesses as compared to related work where concurrent accesses are systematically avoided in the generated schedule. The experimental evaluation uses a subset of the STR2RTS streaming benchmarks [RP17], as well as synthetic task graphs using the TGFF graph generator [DRW98].

The contributions of this chapter are threefold:

i) The first proposed novel approach derives precise bounds on worst-contention on a shared round-robin bus. Compared to state-of-the-art, this method employs knowledge of the application structure (task graph) and mapping decisions to determine tasks that effectively execute in parallel. Therefore, the worst-case bus access delays are tightened with the accurate amount of interferences, compared to a worst-contention scenario.

ii) Second, two scheduling techniques, an ILP formulation and a heuristic algorithm, generate time-triggered partitioned schedules. They augment the scheduling frameworks from Chapter 2. The novelty with respect to existing scheduling techniques lies on the ability of the scheduler to select the best strategy regarding concurrent accesses to the shared

bus (allow to forbid concurrency) to minimise the overall makespan of the schedule.

iii) Third, empirical experiments evaluate the benefit of precise estimation of contentions as compared to the baseline estimation from previous Chapter 2. Moreover, we discuss the interest of allowing concurrency (and thus interference) between tasks as compared to state-of-the-art techniques such as [BDN+16] where contentions are systematically avoided.

The method presented in this chapter has been published in [RDP17]. Nevertheless some differences exist between this chapter and the aforementioned publication. In the paper, the scheduler implements an optimisation regarding the mapping of tasks which stands at follows: considering two tasks exchanging data, if they are mapped on the same core, then these data are not written then read to/from the main memory but remain in the SPM. This optimisation saves some time by not creating traffic over the bus. Motivations to not include it in this dissertation are twofold: i) complexity in solving time, ii) and coherency. The contribution in following Chapter 4 does not include this optimisation, because, with *non-blocking* communications, this optimisation is too costly in term of solving time. This chapter, therefore, does not include it as a matter of coherency. Nonetheless, conclusions remain identical as in the publication, thus strengthening the proposed contribution.

This chapter is organised as follows: Section 3.1 presents an example motivating and briefly explaining the detailed method in Section 3.2. Then Section 3.3 applies this method to capture the effective contention within two scheduling techniques. Section 3.4 validates the approximate algorithm against the ILP formulation before showing the benefit of using the contention-aware method. Finally, Section 3.5 compares this study again the literature before concluding in Section 3.6.

3.1 Motivating example

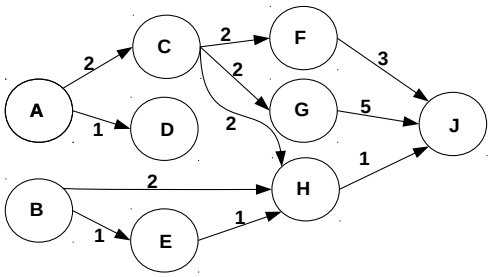
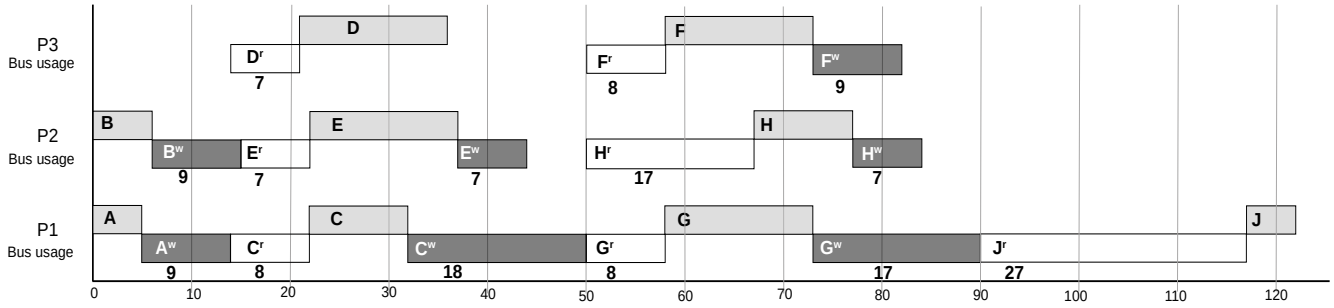


Figure 3.1 – Running task graph example, identical to Figure 2.3

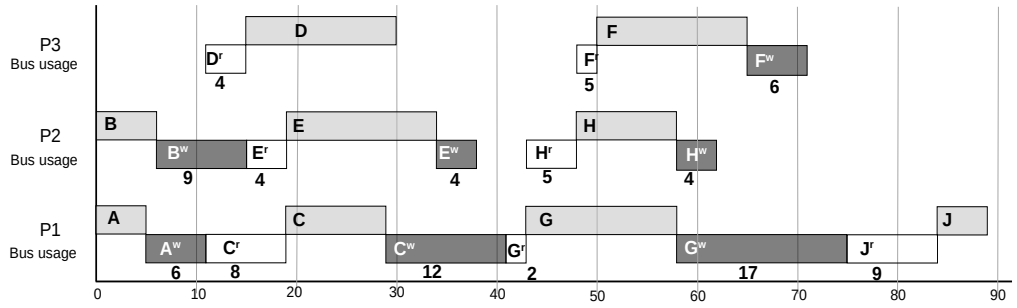
Figures 3.1 and 3.2a are identical to Figure 2.3 and Figure 2.4 respectively, and act as a reminder from Chapter 2. Figure 3.1 represents an application task-graph, while Figure 3.2a shows the resulting schedule of this task-graph on a tri-core with a worst-contention model as in Section 2.4 with identical WCET estimates from Table 2.1.

Figure 3.2b depicts the resulting schedule of the following method with contention-awareness. This schedule is generated using subsequent ILP formulation. Same as previously, the X-axis is time, the Y-axis is cores and bus timelines, white/grey/dark boxes display read/exec/write

phases respectively with a number under communication phases indicating the communication delay.



(a) Resulting schedule with a worst-contention scenario targeting – overall makespan is 122 time units.



(b) Resulting schedule with a contention-aware contention scenario – overall makespan is 89 time units.

Figure 3.2 – Motivation example

With the help of the application structure and the mapping decision, the scheduler computed a tighter worst-case communication delay. For example, write phase of E^w interferes only with one other communication phase, i.e. write phase of C. Therefore, the worst-case delay for E^w reduces to 4 time units instead of 7. Same considerations apply for most communication phases except for B^w, C^r, and G^w that still conflict with two other phases (worst-case situation from Figure 3.2a).

In addition, assuming that the application executes on its own on the architecture, then no parallel requests occur at the time of the *read* phase of task J because of the structure of the application. Similarly on the same example, the *write* phase of task A can only be delayed by the *write* phase of task B (assuming an optimal schedule with at least three cores).

It is also interesting to note that the three read phases of tasks F, G, H have been delayed. In this case, the solver preferred to postpone the activation of these two tasks to avoid adding interferences. This results with very shorter worst-case communication delay for F^r, G^r and H^r.

Finally, contention-aware schedule makespan is 89 time units, while worst-contention schedule makespan was 122 time units. This represents an improvement of 37%.

3.2 Improving worst-case communication cost

The communication cost for a communication phase depends on how much interference this phase suffers from. The interference is due to tasks running in parallel on the other cores. The number of such tasks depends on scheduling decisions (task placement in time and space). Considering a task i, only tasks that are assigned to a different core may interfere with i, and only tasks that execute within a time window overlapping with that of i actually interfere. This section presents, using a top-down approach, how a precise estimation of communication costs is obtained. For the whole document, *concurrent* tasks is used for tasks with no data dependencies that *may* be executed in parallel, while *parallel* tasks is used for tasks that are effectively scheduled to run in overlapping time windows.

Accounting for the actual concurrency in applications Equation (2.1d) (Chapter 2, Page 37) statically computes communication costs assuming all cores (equation (2.2)) execute a communicating phase and thus add a waiting time to every memory access, which is pessimistic. A first solution to reduce this pessimism lies in exploiting knowledge about the application itself to determine the amount of concurrent tasks.

A pair of tasks is concurrent if they do not have data dependencies between each other, i.e. tasks that may execute in parallel. As an example, in Figure 3.1, tasks F and G are concurrent and task G and task A are *not* concurrent. Determining if two tasks are concurrent is usually NP-complete [Tay83]. However, with the DAG task model properties, in particular the presence of statically-paired communication points, evaluation of concurrency is polynomial. Two tasks are concurrent if there exists no path connecting them in the task graph. By building the transitive closure of the task graph, using for example the classical Warshall's algorithm [War62], two tasks i and j are concurrent if there is no edge connecting them in the transitive closure of the DAG. In the following, function *are_conc(i,j)* will be used to indicate task concurrency according to the method described in this section. It returns *1/true* when tasks i and j are concurrent and *0/false* otherwise.

According to the knowledge of the structure of the task graph, equation (2.2) (amount of interference) can then be refined as follows. Instead of considering $NbCores - 1$ contentions for every memory access, the worst-case number of contenders with a task i is now as in equation (3.1).

$$\begin{aligned} &\forall i \in T, \chi \in \{r, w\}; \\ &\qquad interf_i^{\chi} = min(NbCores - 1, \sum_{j \in T} are_conc(i, j)) \end{aligned} \tag{3.1}$$

Further refining the worst-case degree of interference Keeping the example from Figure 3.1, if the two concurrent tasks A and B are mapped on the same core, then their communication phases do not interfere anymore. Knowledge of tasks' scheduling (tasks placement and time window assigned to each task), when known, can further help refining the amount of interference suffered by a task.

Reasoning in reverse, two phases do *not* overlap if one ends before the other starts, which leads, for tasks with *read-execute-write* semantics, to equation (3.2). For two tasks i and j,

taking two phases τ_i^χ and τ_j^ψ, where χ and ψ can either represent a *read* or a *write*, equation (3.2) states that if phase τ_j^ψ ends before phase τ_i^χ starts or vice versa, then the two phases τ_i^χ and τ_j^ψ do *not* overlap. Notice, ρ_i^χ defines the start time of phase χ from task i, and $\rho_i^\chi + DELAY_i^\chi$ defines the end time this same phase. We consider here the end date as the first discrete time point at which the task is over, thus no overlapping occurs when $\rho_j^\psi + DELAY_j^\psi$ and ρ_i^χ are equal.

$$\rho_j^\psi + DELAY_j^\psi \leq \rho_i^\chi \vee \rho_i^\chi + DELAY_i^\chi \leq \rho_j^\psi \tag{3.2}$$

Then, by negating equation (3.2), equation (3.3) is true if two tasks have overlapping execution windows. In the following, $are_OL(\tau_i^\chi, \tau_j^\psi)$ returns *1/true* if the communication delay of phases τ_i^χ and τ_j^ψ overlap, and *0/false* otherwise.

$$\begin{aligned} are_OL(\tau_i^\chi, \tau_j^\psi) &= \neg(\rho_j^\psi + DELAY_j^\psi \leq \rho_i^\chi \vee \rho_i^\chi + DELAY_i^\chi \leq \rho_j^\psi) \\ &\equiv \rho_j^\psi + DELAY_j^\psi > \rho_i^\chi \wedge \rho_i^\chi + DELAY_i^\chi > \rho_j^\psi \end{aligned} \tag{3.3}$$

Assuming the schedule is known, the degree of interference a task can suffer from can be determined by counting the number of other tasks that overlap in the schedule. Only concurrent tasks (function are_conc) can overlap, because dependent (not concurrent) tasks have data dependencies.

As constrained by the AER execution model (Section 2.2), only communication phases request accesses to the bus, thus only the amount of interference of the *read* and *write* phases needs to be computed. Looking at write phase B^w from example in Figure 3.2b, both A^w, C^r, and D^r interfere with it, resulting in an amount of interferences greater than the worst-case – i.e. $3 \geq NbCores - 1$. Therefore, only one of the two contending tasks from core $P1$ should be accounted for as they are both interfering tasks mapped on the same core. Then equations (3.4a) and (3.4b) counts the number of cores interfering, not only the number of phases.

In complement for clarity, a cast from boolean to integer allow to sum boolean values – i.e. $true = 1$, $false = 0$. The big disjunction is equal to 1, if at least one phase of a concurrent task j is mapped on the concerned core p, and is overlapping with communication of task i. The disjunction guarantees that no core with several overlapping phases will be counted more that once. Finally, there is no need to exclude the core on which i is mapped. Indeed, the scheduler enforces that bus requests are mutually excluded when originated from the same core.

$\forall i \in T$;

$$interf_i^r = \sum_{p \in P} \left[\bigvee_{j \in T | are_conc(i,j)} ((are_OL(\tau_i^r, \tau_j^r) \vee are_OL(\tau_i^r, \tau_j^w)) \wedge j \; is \; mapped \; on \; p) \right] \tag{3.4a}$$

$$interf_i^w = \sum_{p \in P} \left[\bigvee_{j \in T | are_conc(i,j)} ((are_OL(\tau_i^w, \tau_j^w) \vee are_OL(\tau_i^w, \tau_j^r)) \wedge j \; is \; mapped \; on \; p) \right] \tag{3.4b}$$

The values of $interf_i^r$ and $interf_i^w$ from equations (3.4a) and (3.4b) then replace the pessimistic value of *NbCores - 1* from equation (2.2). This new interference value tighten the worst-case delay in equation (2.1d) for communication phases. As a reminder, equation (2.1d) is repeated here.

$$delay_i^\chi = T_{slot} \cdot waitingSlots_i^\chi \cdot interf_i^\chi + T_{slot} \cdot chunks_i^\chi + remainingTime_i^\chi \tag{2.1d}$$

The two scheduling techniques described in Section 3.3 use these formulas jointly with schedule generation. But this new contention refinement depends on the knowledge of the schedule. Therefore, It creates an interdependence between delays computation and tasks scheduling. The scheduler can no longer use a constant value for communication delays, which is dependent on tasks mapping, which in turn drastically increases the time complexity of the problem to solve.

3.3 Resource-aware scheduling techniques

This section modifies the scheduling framework introduced in Section 2.3 including both the ILP formulation and the heuristic algorithm. They are upgraded to compute the precise worst-case amount of interference for communication phases. The main outcome of both techniques is still a static partitioned time-triggered non-pre-emptive schedule, for one single application represented by a DAG.

3.3.1 Integer Linear Programming (ILP) formulation

Section 2.3.2 introduced the baseline scheduling ILP formulation. This section gives only modifications, additions or suppressions to apply. As a reminder, initial formulation included constraints aiming at:

— ensuring the unicity of mapping tasks on cores,

— detecting if two tasks are mapped on the same core,

— ordering tasks and ordering tasks on the same core,

— enforcing the AER execution model,

— enforcing that no pair of phases of the same type (computation/communication) overlaps on the same core ,

— enforcing data dependencies.

All these previous constraints remain un-modified. Identically, the objective remains: the overall schedule makespan minimisation. Moreover, boolean operators are still linearised using [BD07] as in Section 2.3.2.

Table 3.1 summarises the notations and variables removed from the initial framework in Chapter 2, while Table 3.2 summarises the one added to the ILP formulation.

Table 3.1 – Removed Notations

Constants	$DELAY_i^r$	task i *read*, *write* phases' are now fragmented. Therefore,
	$DELAY_i^w$	a communication delay is attached to each fragment in the following ILP

Table 3.2 – Added Notations & ILP variables

Func.	$are_conc(i,j)$	return true if i and j are concurrent, as defined in Section 3.2
Cst.	D_i^r D_i^w	the cumulated amount of data number task i reads/writes
	$CHUNK_i^r$ $CHUNK_i^w$	the number of full slots to *read/write* D_i^r/D_i^w used by eq. (2.1a)
	$REMTIME_i^r$ $REMTIME_i^w$	the remaining time to *read/write* that do not fit in a full slot from eq. (2.1b)
	$WAITSLOT_i^r$ $WAITSLOT_i^w$	the number of full slots a *read/write* phase must wait used by eq. (2.1c)
Int. var.s	$interf_i^r$ $interf_i^w$	the number of interfering tasks of the *read/write* phases of i from eq. (3.4a) and (3.4b)
	$delay_i^r$ $delay_i^w$	task i *read*, *write* phases' WCET from equations (2.1d)
Bin. var.s	$ov_{i,j}^{\chi\psi} = 1$	phase χ of i overlaps with phase ψ of j $(\chi,\psi) \in \{rr, ww, rw, wr\}$
	$ovc_{i,p}^r = 1$ $ovc_{i,p}^w = 1$	read or write phase of i interferes with a least one communication phase mapped on p

Computing communication phases interference All the following equations implement, using linear equations, the refinement of contention duration presented in Section 3.2. With the use of the function $are_conc(i,j)$ tasks that never interfere with each other are excluded from the search space. Equations (3.5a) to (3.5d) implement function are_OL derived from equation (3.3). For each pair of communication phases, the equations indicate if they are overlapping in the schedule ($ov_{i,j}^{\chi\psi} = 1$, with $\chi \in \{rr, ww, rw, wr\}$).

Note that when there is no data for the considered communication phase ($D_i^r = 0, D_i^w = 0$), then there is no possible overlapping, and then each $ov_{i,j}^{\chi\psi}$ is forced to be equal to 0 to decrease the solving time.

$\forall i \in T, \forall j \in are_conc(i,j)$;

$$ov_{i,j}^{rr} = \begin{cases} 0 & \text{if } D_i^r = 0 \vee D_j^r = 0 \\ \rho_i^r < \rho_j^e \wedge \rho_j^r < \rho_i^e & \text{otherwise} \end{cases} \tag{3.5a}$$

$$ov_{i,j}^{ww} = \begin{cases} 0 & \text{if } D_i^w = 0 \vee D_j^w = 0 \\ \rho_i^w < \rho_j^w + delay_j^w \wedge \rho_j^w < \rho_i^w + delay_i^w & \text{otherwise} \end{cases} \tag{3.5b}$$

$$ov_{i,j}^{rw} = \begin{cases} 0 & \text{if } D_i^r = 0 \vee D_j^w = 0 \\ \rho_i^r < \rho_j^w + delay_j^w \wedge \rho_j^w < \rho_i^e & \text{otherwise} \end{cases} \tag{3.5c}$$

$$ov_{i,j}^{wr} = \begin{cases} 0 & \text{if } D_i^w = 0 \vee D_j^r = 0 \\ \rho_i^w < \rho_j^e \wedge \rho_j^r < \rho_i^w + delay_i^w & \text{otherwise} \end{cases} \tag{3.5d}$$

Then, as stated in Section 3.2, two communication phases from the same core interfering with the considered one, should not be counted twice. The amount of interferences corresponds to the amount of interfering cores, but not the amount of interfering communication phases. Equations (3.6a) and (3.6b) implement both equations (3.4a) and (3.4b) for both read and write phases respectively. They count the number of interfering cores in $interf_i^r$ and $interf_i^w$ for respectively the *read* and *write* phases of task i.

$\forall i \in T$;

$$interf_i^r = \sum_{\forall p \in P} \left[\bigvee_{\forall j \in T | are_conc(i,j)} (ov_{i,j}^{rr} \vee ov_{i,j}^{rw}) \wedge p_{j,p} \right] \tag{3.6a}$$

$$interf_i^w = \sum_{\forall p \in P} \left[\bigvee_{\forall j \in T | are_conc(i,j)} (ov_{i,j}^{ww} \vee ov_{i,j}^{wr}) \wedge p_{j,p} \right] \tag{3.6b}$$

Finally, equations (3.7a), (3.7b) and (3.7c) highlight three optimizations that constrain the overlapping variables, to improve the solving time.

$$ov_{i,j}^{rr} = ov_{j,i}^{rr} \tag{3.7a}$$

$$ov_{i,j}^{ww} = ov_{j,i}^{ww} \tag{3.7b}$$

$$ov_{i,j}^{wr} = ov_{j,i}^{rw} \tag{3.7c}$$

Estimation of worst-case communication duration Previous Chapter 2 used a constant to represent the communication delay ($DELAY_i^\chi$), in this chapter, the delay is dependent on the current state of the schedule and must therefore be an integer variable ($delay_i^\chi$).

Estimating the communication cost of a read or write phase is then the application of equation (2.1d) with the right amount of interference from above equations (3.6a) and (3.6b), as in equation (3.8).

$$\forall i \in T; \chi \in \{r, w\}$$
$$delay_i^\chi = T_{slot} \cdot WAITSLOT_i^\chi \cdot interf_i^\chi + T_{slot} \cdot CHUNKS_i^\chi + REMTIME_i^\chi \tag{3.8}$$

3.3.2 Forward List Scheduling algorithm

Section 2.3.3 introduced the scheduling framework for the heuristic algorithm. This section gives only modifications, additions or suppressions to apply. As a reminder, previous approximate algorithm is based on list scheduling. The technique scheme orders tasks, then schedules them one by one as soon as possible without backtracking in order to minimise the overall

makespan. The same scheme applies to this new version, again tasks ordering is built upon two BFS and DFS algorithms. The algorithm keeps trying to map tasks on each core and select the mapping minimising the schedule length. But this new version introduces the effective worst-case amount of interference in the communication cost. Algorithm 3.3.1 presents this new version where only novelties are detailed below.

ALGORITHM 3.3.1: Forward list scheduling

Input : A task graph G $= (T, E)$ and a set of processors P
Output : A schedule

```
1  Function ListSchedule(G = (T, E), P)
2      Elist ← BuildListElement(G)
3      Qready ← TopologicalSortNode(Elist)
4      Qdone ← ∅
5      schedule ← ∅
6      while t ∈ Qready do
7          Qready ← Qready \{t}
8          Qdone ← Qdone ∪{t}
           /* tmpSched contains the best schedule for the current task */
9          tmpSched ← ∅ with makespan = ∞
10         foreach p ∈ P do
11             copy_aw ← schedule
12             map t on p in copy_aw
13             ScheduleElement(copy_aw, Qdone, t, p)
14             AdjustScheduleEffectiveContention(copy_aw, Qdone, t)
15             copy_free ← schedule
16             map t on p in copy_free
17             ScheduleElement(copy_free, Qdone, t, p)
18             AdjustScheduleContentionFree(copy_free, Qdone, t)
19             tmpSched ← min_makespan(tmpSched, copy_aw, copy_free)
20         schedule ← tmpSched
21     return schedule
```

Finding the best solution between contention-aware and contention-free Previous example from Section 3.1 depicted the case where the solver postpones a task (not respecting the ASAP property). With the above ILP formulation, the ILP solver had the opportunity to select, on a per task basis, the best solution between two options: synchronize every communication phase (perform them in mutual exclusion) to obtain a contention-free schedule, or enable concurrency if it results in a shorter global schedule.

The heuristic uses a similar approach. Two schedules are computed: one allowing concurrent tasks overlapping communication phases (lines 11-14) and one avoiding this overlapping (lines 15-18). Then the shortest of the two schedules is selected among with the previously computed schedule (line 19).

ALGORITHM 3.3.2: Adjust schedule in contention-free mode : Updating the schedule to avoid contention

Input : An incomplete schedule schedule, the list of already mapped tasks Qdone and the newly mapped task cur_task
Output : An updated schedule

```
1 Function AdjustScheduleContentionFree(schedule, Qdone, cur_task)
2     mark cur_task as contention-free
3     while schedule is not stable do
4         foreach t ∈ Qdone| do
5             if (are_OL(t,cur_task) then
6                 ρ^r_cur_task ← ρ^w_t + delay^w_t
7     return schedule
```

Scheduling contention-free task Algorithm 3.3.2 guarantees that the added task does not suffer from any contention by shifting it forward along its processor time line. That process is repeated until a stable state is reached, because shifting a task on a core can create interference with another task on another core. Convergence is always reached as the task is moved in only one direction, and at some point this task will be the last one scheduled.

Because list scheduling algorithms enforce a non-backtracking policy, when a task is deemed to be contention-free, the scheduler must keep track of this decision. Algorithm 3.3.2 fulfils this requirement. It marks the task to avoid future creation of interference with itself when adding new tasks.

Updating the schedule to cope with interference The contribution of this algorithm is to compute the effective worst-case amount of interference in communication cost at schedule time. This cost depends on past and future mapping decisions. Therefore, after scheduling each task, the communication costs in relation with the newly scheduled task must be recomputed and tasks must be moved on the time line of each involved core to ensure a valid schedule, i.e. a schedule accounting for all interferences (line 14 in Algorithm 3.3.1).

Each time a task is added, new interferences caused by the addition of the task in the schedule must be added, and the delay of (some) communication phases must be recomputed.

As an example, Figure 3.3a depicts a partial initial schedule extracted from example from Section 3.1. White/grey/dark boxes still represent read/exec/write phases. A task F mapped on $P1$ sends 3 tokens to a task not yet in the schedule and a task H receiving 5 tokens with a $T_{slot} = 3$ and $D_{slot} = 3$, thus $delay^w_F = 3$ and $delay^r_H = 5$ (equation (3.8), in this situation none of them suffers from any parallel task). Figure 3.3b sketches the addition of task G on $P2$, it writes 5 tokens. The writing delay $delay^w_C$ and reading delay $delay^r_H$ must be adjusted now to account for the interference introduced by G^w. Thus, the new writing delay for task F becomes $delay^w_F = 6$, and reading delay for task H becomes $delay^r_H = 11$, as now there is 1 interference. As a consequence; the read task H^r moves to guarantee that only one bus request is active at a time. Then, exec phase of task H moves to guarantee the blocking communication restriction (Section 2.2.1).

Whenever a communicating task already mapped needs to be rescheduled/delayed, it may

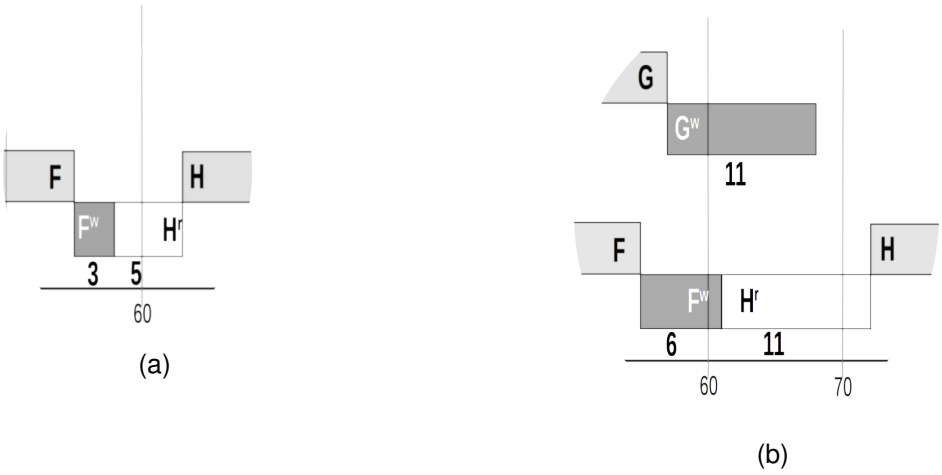


Figure 3.3 – Example of adjustments that occur while scheduling. (3.3a) initial schedule of 2 tasks. (3.3b) adjusted communication delays after the addition of task G

change the number of interfering tasks. The communication delay of all tasks impacted by this change must therefore be recomputed, since they may in turn also create interference. The partial communication delay calculation must therefore proceed iteratively until no task is impacted anymore.

ALGORITHM 3.3.3: Adjust schedule in contention-aware mode : Updating the schedule to cope with interference

Input : An incomplete schedule schedule, the list of already mapped tasks Qdone and the newly mapped task cur_task
Output : An updated schedule

```
1  Function AdjustScheduleEffectiveContention(schedule, Qdone, cur_task)
2      while schedule.length is not stable do
3          rSet ← BuildRelated(schedule, Qdone, cur_task)
4          foreach t ∈ rSet do
5              if t is not marked contention-free then
6                  re-Compute delay_t^r and delay_t^w using equations (2.1d) and (3.4a)/(3.4b)
7          foreach (t, t') ∈ rSet ∪ rSet | t¬t' ∧ ρ_t^r < ρ_t'^r do
8              if (t and t' are mapped on the same core ∨ t is marked contention-free) ∧ are_OL(t,t')
                 then
9                  ρ_t'^r ← ρ_t^w + delay_t^w
10     return schedule
```

To reduce the number of tasks impacted by each adjustment, algorithm 3.3.3 first computes the set of *related* tasks (line 3), i.e. tasks that can be impacted by the addition of the current task. To build this set, it is simple as looking into the schedule for tasks with a *recursively* overlapping communication phase or *scheduled later*.

Then from the related task set, Algorithm 3.3.3 recomputes the delay of each communica-

tion phase (line 6). Due to previous tasks movement on the processor time lines, lines 7-9 shift forward tasks to guarantee execution constraints: no two tasks execute on the same core at the same time, and synchronisation decision are still met. This process is then repeated until the length of the schedule becomes stable.

Convergence of Algorithm 3.3.3 is always reached since, at worst, every concurrent task will interfere and no previous decision regarding placement and ordering are discussed (non-backtracking). In addition, tasks can be shifted (line 9), but in only one direction, therefore the increase of the release time is monotonic.

3.4 Experiments

This chapter presented a modification of the ILP formulation and the heuristic algorithm from Chapter 2. Therefore, the first presented experiment aims at validating the quality of the new heuristic algorithm against the ILP formulation, as in previous experiments in Section 2.4. Then, an empirical evaluation shows the benefits of counting the effective worst-contention using the approximate method. Experiments were conducted on real code, in the form of the open-source refactored StreamIT [TKA02] benchmark suite: STR2RTS [RP17], as well as on synthetic task graphs generated using the TGFF [DRW98] graph generator.

Applications from the *STR2RTS benchmark suite* are modelled using fork-join graphs and come with timing estimates for each task and amount of data exchanged between them. We were not able to use all the benchmarks and applications provided in the suite due to difficulties when extracting information (e.g. task graph, WCET) or because some test cases are linear chains of tasks with no concurrency. A table summarising the used benchmarks is available in appendix (page 94).

Task Graph For Free (TGFF) is used, as in Chapter 2 when there is a need to generate a large number of task graphs. It is used to evaluate the quality of the heuristic against the ILP formulation. Due to the intrinsic complexity of solving the scheduling problem using ILP, we need for that experiment small task graphs such that the ILP is solved in reasonable time.

The same STG set from previous chapter, Section 2.4 is reused. Table 3.3 reminds the resulting parameters of the task set. The table includes the number of task graphs, their number of tasks, the width of the task graph, the range of WCET values for each task and the range of amount of exchanged data in bytes between pairs of tasks.

Table 3.3 – Task graph parameters for synthetic task graphs

	#Task graphs	#Tasks	Width	WCET	Amount of bytes exchanged
		<min,max,avg>			
STG	50	5, 69, 22	3, 17, 8	$[5; 6000[$	$[0; 192]$

As previously, all reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). Because last chapter in Section 2.4.3 showed that the value of T_{slot} has a little impact on the schedule makespan, but it is rather better to keep it small, in all experiments $T_{slot} = 3$ as in [KHM+13]. The transfer rate is one word (4 bytes) per time unit.

3.4.1 Quality of the heuristic compared to ILP

The following experiments aim at quantifying the over-approximation induced by the heuristic algorithm in the same manner than in Section 2.4.2. This gap is still expected to be small. Again, this experiment employs the STG task set with number of varying in $\in \{2, 4, 8, 12\}$. The heuristic is implemented in C++ and CPLEX was configured with a timeout of 11 hours.

Table 3.4 – Degradation of the heuristic compared with the ILP (synthetic task graphs / STG)

% of exact results (ILP only)	degradation <min,max,avg> %
66%	0%, 20%, 2%

Table 3.4 summarises the results, the first column presents the percentage of exact results the ILP solver is able to find in the granted time. As in Section 2.4.2, we only refer to the exact solutions for the comparison as the feasible ones (i.e not exact) might bias the conclusion on the quality of the heuristic compared to the ILP. Therefore remaining 34% include feasible solutions as well as problems where timeout was reached (nor optimal solution found, neither proved infeasible). The second column presents the minimum, maximum and average degradation in percent.

The average degradation is low, which means our heuristic has acceptable quality. A deeper analysis of the distribution of degradation, Figure 3.4, shows that 96% of the heuristic schedules are less than 10% worse than the ILP formulation solutions. We also observed a schedule generation time far much lower for the heuristic than the ILP solving time, < 1 second on average, with a maximum observed of 2 seconds.

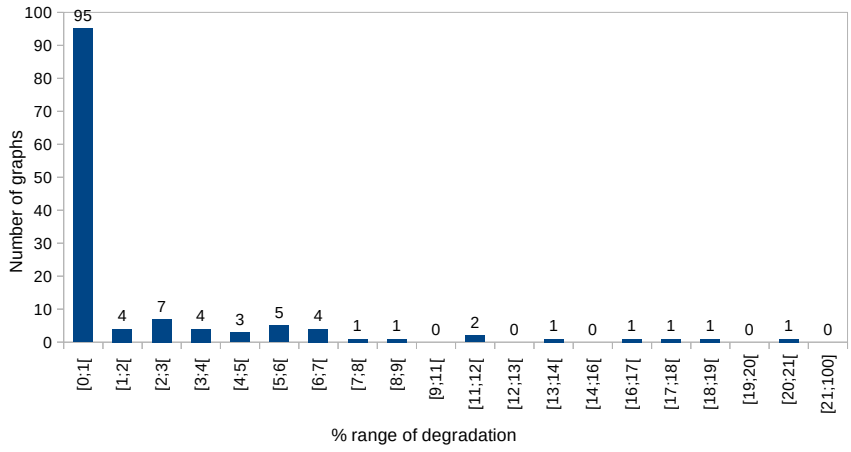


Figure 3.4 – Distribution of the degradation of the heuristic against the ILP formulation using STG task set.

3.4.2 Quality of the heuristic compared to worst-contention communications

We estimated the gain when using the method presented in this chapter to tighten communication delays over the same heuristic using the worst-contention scenario from Chapter 2. The higher is the gain the tighter is the proposed estimation of communication delays. Experiment were performed on the STR2RTS benchmarks suite previously described. Hardware parameters were identical as in previous experiments.

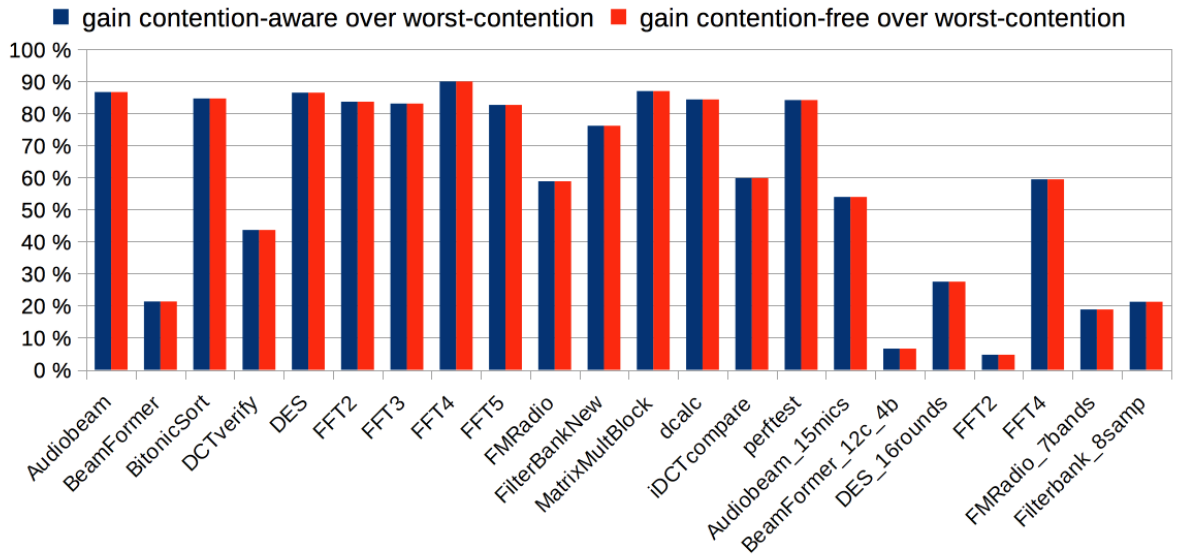


Figure 3.5 – Gain in % obtained with contention-aware scenario (heuristic, STR2RTS benchmarks)

Results are depicted in Figure 3.5 by *blue* bars. The gain is computed using equation 3.9.

$$\frac{worst\ concurreny - accurate\ interference}{worst\ concurrency} \tag{3.9}$$

Results show that the gain to use the accurate degree of interference (contention-aware) decreases the overall makespan of 59% on average over the worst-contention, demonstrating the benefits of precisely computing the degree of interference at schedule time.

3.4.3 Quality of the heuristic compared to contention-free communications

Recent papers [PBB+11; AP14; BDN+16] suggested to build contention-free schedules to nullify interference cost. Due to the different task models and system models in the aforementioned works, a direct comparison with them is hard to achieve. Thus, we modified our heuristic to produce a schedule without any contention and to be as close as possible to the ideas defended in the mentioned papers. The gain of a contention-free heuristic against a worst-contention one is depicted for the StreamIt benchmarks in Figure 3.5 by red bars.

Among the contention-aware and contention-free variants of our heuristic, no method outperforms the other for all benchmarks. Moreover, the difference between the schedule makespans

using the two variants is very small. The average difference is 0,08%, with a worst value of 0,3%. Contention-free communications scenario (red bars) give better results for *fft3, fft5, filterbank, fm, hdtv, mp3, tconvolve and vocoder*; the proposed heuristic (blue bars) gives better for *audiobeam, beamformer, mpd*; the results for all other benchmarks are identical. Regarding schedule generation duration for the STR2RTS benchmarks, *contention-free* solutions are found in less than 30 seconds on average, while *contention-aware* once need less than 3 minutes on average. The shortest schedule generation times were obviously observed when generating contention-free schedules, because no estimation of interference costs has to be performed at all.

3.5 Related work

Shared resources in multi-core systems may be either shared software objects (such as variables) that have to be used in mutual exclusion or shared hardware resources (such as buses or caches) that are shared between cores according to some resource arbitration policies (TDMA, round-robin, . . .). Dealing with shared objects is not new, and there now exists several techniques adapted from the single-core systems. Section 1.6.5 pictured the wide range of possibilities to manage such shared resources. Beyond shared objects, multi-core processors feature hardware resources that may be accessed concurrently by tasks running on different cores. A contention analysis then has to be defined to determine the worst case delay for a task to gain access to the resource.

Approaches to estimate contention delays for round-robin arbitration differ according to the nature and amount of information used to estimate contention delays. For architectures with caches, Dasari et al. [DN12; DNA16] only assume task mapping known, whereas Rihani et al. [HMC+16] assume both mapping and execution order on each core known. Schliecker et al. [SNE10] tightly determine the number of interfering bus requests. In comparison with these works, this chapter jointly calculates task scheduling and contentions with the objective of minimising the schedule makespan by letting the technique decide when it is necessary to avoid or to account for interference.

Altmeyer et al. [ADI+15] created a framework to compute the response time of a sporadic task set previously mapped and scheduled on a multi-core. This work has been implemented to target the Kalray MPPA [DVP+14] by Rihani et al. [HMC+16]. In [ADI+15], they focus on the computation of memory accesses delay according to different memory hierarchy architecture (caches, SPMs, mixed of both), and different bus arbitration (FIFO, Round-Robin, TDMA . . .). Their work is orthogonal to ours as we deal with an AER execution model, and a SPM utilisation that excludes any bus access during the execution. Therefore, our isolation of memory requests allow us to relax their conservative assumptions regarding where and when in the task memory accesses occur. However, we can improve our method with their work to more precisely compute communication delays when two or more communications phases partially overlap, which is left for future work.

Becker et al. [BDN+16] proposed an ILP formulation and a heuristic aiming at scheduling periodic independent PREM-based tasks on one cluster of a Kalray MPPA processor. They systematically create a contention-free schedule. The presented work in this chapter differs in the considered task model as well as the goal to reach. They consider sporadic independent

tasks to which they aim at finding a valid schedule that meets each tasks' deadline. In contrast, we consider one iteration of a task graph and we aim at finding the shortest schedule. In addition, resulting schedules might include overlapping communications due to scheduler decision, while [AP14; BDN+16] only allow contention-free communication.

Using legacy code, or legacy schedule, Martinez et al. [MHP17] reduced contention in input schedules by introducing slack time between the execution of pairs of tasks consecutively assigned to the same core, which limits the contention between concurrent tasks.

Giannopoulou *et al* proposed in [GSH+16] a combined analysis of computing, memory and communication scheduling in a mixed-criticality setting, for cluster-based architectures such as the Kalray MPPA. Similarly to our work, the authors aim, among others, at precisely estimating contention delays, in particular by identifying tasks that may run in parallel under the FTTS schedule, that uses synchronization barriers. However, to the best of our knowledge they do not take benefit of the application structure, in particular dependencies between tasks to further refine contention delays.

To quantify memory interference on DRAM-banks, [KDA+14; YPV15] proposed two analyses, *request-driven* and *job-driven*. The former one bounds memory request delays considering memory interference on the DRAM bank, while the latter adds the worst-contention on the data-bus of the DRAM. Their work is orthogonal to ours: the *request-driven* analysis would refine the access time part in our delay, while our method could refine their *job-driven* analysis by decreasing the amount of concurrency they use.

In [BCS+16], an off-line/on-line combined technique monitors the bandwidth to access an off-chip memory. In the off-line mode, they assume a worst-contention scenario. Then on-line, when a contention threshold is reached best-effort tasks are suspended to leave real-time tasks executing. Then when a down threshold is reached again, best-effort tasks are resumed. This approach differs in many point as this dissertation di not mix off-line and on-line modes, no mixed criticality tasks. But this dissertation computes the affective amount of interference off-line.

3.6 Conclusion

This chapter showed how to take advantage of the structure of parallel applications, along with their target hardware platforms, to obtain tight estimates of contention delays. The presented approach builds on a precise model of the cost of bus contention for a round-robin bus arbitration policy, which we use to define a mapping/scheduling strategy. This strategy is implemented in a ILP formulation that formally describes the solved problem, and with an approximate algorithm that solves the problem faster. The empirical evaluation demonstrates the validity of the heuristic algorithm by showing that the over-approximation, induced by such technique, remains low (2% in average). Then, the experimental results show that, compared to a scenario accounting for a worst case contention, this approach improves the schedule makespan by 59% on average. Nevertheless, experiments also show that allowing the scheduler to chose between contention-free and contention-aware scheduling methods mostly results in schedules free from contention. In addition, generating a contention-free schedule needs less computation time/power.

This conclusion, regarding contention-free schedules, must be balanced with the context of

this dissertation. The usage of AER execution model in concordance with SPM-based architecture enforces an isolation of memory accesses. However, in legacy software, it is difficult to identify where the memory accesses occur within a task (or the scheduler does not have this knowledge). Scheduling all tasks in a contention-free manner (no task can run in parallel on different cores) would be equivalent to a sequential schedule on a single-core, and the benefit of using a multi-core architecture will be lost. We are quite confident that applying our contention-aware scheduling method with such execution model will result in tighter schedule length than both worst-contention and contention-free solutions at the task level.

The code of this scheduler using both techniques (ILP, heuristic) and accounting for contention is available at `https://gitlab.inria.fr/brouxel/methane`.

One of the limitation in the presented approach is its restriction to blocking communications. A natural extension of this work is therefore to relax this constraint and introduce support for asynchronous communications, which are notoriously more challenging to support in a real-time context. Next Chapter 4 develops this idea in details.

CHAPTER 4

HIDING COMMUNICATION LATENCIES IN CONTENTION-FREE SCHEDULES

Efficient and predictable management of SPMs are facilitated by application models that offer a high-level view of parallel programs. As previously, this chapter focuses on applications modelled as DAGs, consisting of dependent tasks that exchange data through shared FIFO channels. The AER execution model [MNP+16] is well-suited for SPM-based architectures, as only communication phases accesses the main memory and computation phases can perform in isolation from other tasks. Using proper scheduling techniques, communication can perform without contention which has been shown in previous Chapter 3 to be equivalent in terms of makespan to a precise evaluation of contention with a much slower solving time. Therefore, this chapter embraces this conclusion by adopting a contention-free scheduling strategy.

Blocking communication configuration, from previous chapters, limits the performance of applications by stalling the computing core while data is transmitted between the SPM and the off-chip memory. Relaxing this blocking constraint decreases the impact of communication latencies on the schedule and allow to overlap computation and communication. It consists in adding a pinch of asynchrony with *non-blocking* communication. Then, the core is not stalled, and bus requests can be performed in parallel with computation on cores. To further increase the number of hiding opportunities, fragmented communications enable to cut the communication phases into pieces that are small enough to fit under small computation tasks. *Non-blocking* fragmented communication is the explored path in this chapter to further optimise the schedule length.

To enable *non-blocking* communication, the assumed hardware from previous chapters is updated to integrate both a Direct Memory Access (DMA) and a *dual-ported* SPM. The DMA is in charge of actually executing the transfer between the SPM and the off-chip memory. A *dual-ported* SPM supports concurrent loading and unloading information from both the DMA and the core. Concurrent accesses raise a question of data integrity, if the two actors write some data at the same memory location. However, with a proper SPM allocation scheme decided at design time, a compiler is able to enforce this data integrity issue. Also, a SPM allocation scheme helps to relax the unrealistic unlimited-size SPM assumption from Chapter 2 and Chapter 3.

Additionally, and in comparison with the recent related works (such as [KM08; COK+12; TPG+14]), we permit fragmentation of communication phases, thus allowing a single communication phase to be hidden in multiple execution phases. We contrast with most other works dealing with SPM, e.g. [DP07; BMD+18], by allocating memory regions for the whole data and code, when they aim at deciding the SPM content (what should be stored in SPM or left in the off-chip memory).

In summary, the contributions of this work are:

i) We propose a strategy to map and schedule a task graph onto cores coupled with a

SPM allocation scheme. The generated static contention-free non-preemptive schedules allow, when possible, to overlap communications and computations, through *non-blocking* loading/unloading of information into/from SPM. Communication phases are *fragmented* to maximise the duration of overlapping between communications and computations. The proposed strategy is formulated using an Integer Linear Programming (ILP) formulation, producing optimal schedules, and a heuristic based on list-scheduling.

ii) We provide an experimental evaluation showing our method improves the overall makespan, up to 16% on both real-life and synthetic applications, compared to equivalent schedules generated with blocking communications.

iii) We evaluate the impact of different granularities for communication fragments on the schedule makespan, gain increases with smaller fragments.

The method detailed in this chapter is presented in [RSD+19]. The related paper also describes the implementation of schedules, generated with the following method, on a Kalray MPPA Bostan [DVP+14].

The rest of this chapter details the proposed techniques and is organised as follows. Sections 4.1 and 4.2 summarise the modifications to apply on respectively the hardware and the execution model, leaving other elements of the context identical to the ones used in the previous chapters. A motivating example is presented in Section 4.3. Then, Section 4.4 presents the basic principles of the SPM allocation scheme. The scheduling/mapping/allocation strategies (ILP formulation and heuristic) are detailed in Section 4.5. Section 4.6 presents experimental results. Finally, Section 4.7 presents related studies, before concluding in Section 4.8.

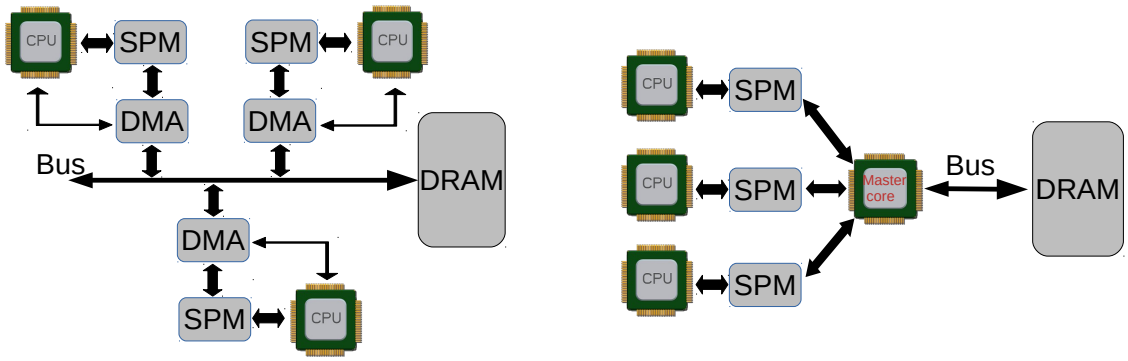
4.1 Hardware support

With the same spirit than in previous chapters, we consider multi-core architectures where every core has access to a SPM. Cores are connected through a bus to the off-chip memory with a FAIR-round-robin arbitration [Ens77]. All communications are non-preemptable and go through the shared global memory (no SPM to SPM communication).

This chapter allows both communication and computation to be performed in parallel. As opposed to previous chapters, the multi-core platform is equipped with a Direct Memory Access (DMA) and a *dual-ported* SPM. At a specific time, the DMA is able to perform the transfer between the main memory and SPM, which is essential to overlap computation and communication. A *dual-ported* SPM integrates two ports, one for core accesses, one for DMA accesses, hence allows concurrent accesses from both DMA and CPU.

Provided support may be either a hardware DMA engine or a specific core acting as a DMA software engine (as long as this specific core has access to other core SPM). Figure 4.1a presents a hardware abstraction with a hardware DMA engine for each core (inspired by **??**), while Figure 4.1b depicts an abstract view of a software DMA where a master core dispatch data between the DRAM and SPMs (inspired by [CLC+09]). At any time, in these two examples, the computing core is able to access the external memory without a request to the DMA. These assumptions are realistic, and are met in both academic and commercial processors (e.g. Patmos [SBH+15], Kalray MPPA [DVP+14]).

Communications can be implemented in *blocking* mode or *non-blocking* mode. In *blocking* mode (Chapters 2 and 3), the CPU is in charge of transfers between its SPM and the shared



(a) Multi-core platform abstraction with a hardware DMA and a dual-ported SPM

(b) Multi-core platform abstraction with a master core acting as a software DMA and a dual-ported SPM

Figure 4.1 – Hardware abstraction

memory, and is stalled during every transfer. In *non-blocking* mode, transfers are managed asynchronously, allowing the CPU to execute other jobs during every transfer.

4.2 Software & execution model support

Assumptions on the application model remain from previous chapters. Applications are still modelled as directed acyclic graphs (DAG). Again, each task is divided in three phases according to the AER execution model.

Compared to the original AER model, we split each communication into *fragments*. A *fragment* is some division of the total amount of data that a task produces/consumes, required by its successive/preceding tasks, written/read to/from the main memory. How the data are divided is determined by the fragmentation scheme, detailed in Section 4.6.3. Thus, a task τ_i is a tuple $\tau_i =< F_i^r, \tau_i^e, F_i^w >$, where τ_i^e is the *exec* phase, and F_i^r (resp. F_i^w) is the set of fragments read (resp. written) by the task. A single fragment f of task i that is read (resp. written) is denoted as $\tau_{(i,f)}^r \in F_i^r$ (resp. $\tau_{(i,f)}^w \in F_i^w$).

Since the code in our experimental evaluation, is generally small and likely to be reused along the execution of the application, for simplicity reasons we assume that the code is preloaded in the SPM at startup. However, enabling code prefetching could be easily done, by including the size of the code of a task in the amount of data to be fetched by its *read* phase. The SPM is then used to store both the code and data.

4.3 Motivating example

Figure 4.3 illustrates the optimal solutions for the application in Figure 4.2, targeting a tri-core architecture with identical WCET estimates as in Table 2.1. All figures display a contention-free scenario, where there are never two communication phases active at each time. As previously, communication phases are carried by the DMA and corresponds to tokens exchanged

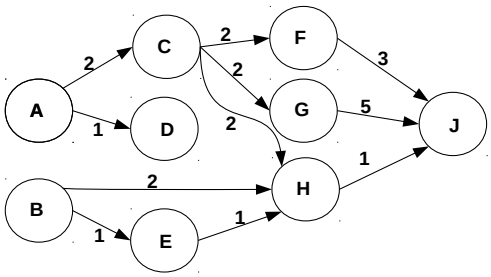


Figure 4.2 – Running task graph example, identical to Figure 2.3

between tasks, the code being previously loaded into the SPM and local data accesses are only through the SPM.

Again in Figure 4.3, for each core, the top time-line depicts the *exec* phases scheduling (grey boxes) and the bottom one the bus usage with read phases (white boxes, black font) and write phases (dark boxes, white font). Again, below each communication phase, the corresponding communication cost is indicated for a contention-free schedule. The SPM is assumed large enough to store all information (code, data, communication buffers), this assumption will be relaxed in Section 4.4.

In Figure 4.3a, the schedule is generated with the ILP formulation from Chapter 3 with *forced contention-free worst-case blocking* communications. All parts of the same task are scheduled contiguously on the same core, and the CPU is stalled while accessing the bus. The *read* and *write* phases are not fragmented as it would not bring any benefit in *blocking* mode.

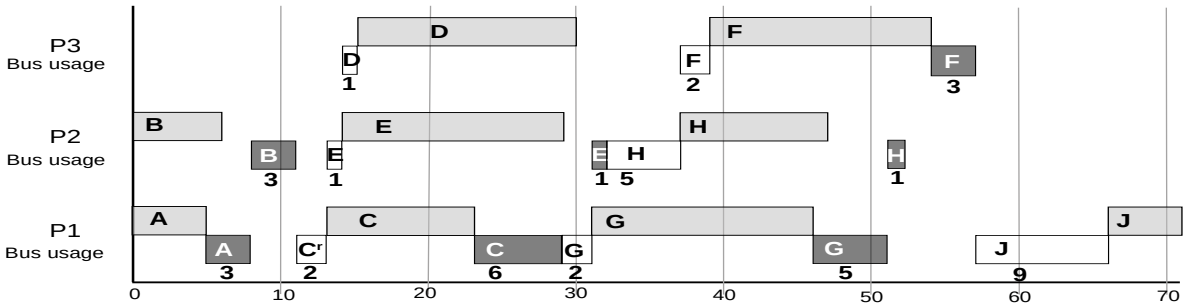
The schedule from Figure 4.3b represents an intermediary step with the introduction of *non-blocking* communication without fragmentation. Therefore, communication phases overlap with computation which hides their latency underneath (e.g.: τ_F^r overlaps with τ_D^e). The communication latencies of this phase are hidden behind the *exec* phase of task τ_D. The resulting overall makespan is 69 time units which show an improvement of 2% over the *blocking* scheme.

We observed from Figure 4.3b that their are very little opportunities to overlap communication and computation with this application on this architecture. In addition to *non-blocking*, fragmented communications offer more flexibility as pictured by the schedule from Figure 4.3c with the default fragmentation policy of one fragment per incoming/outgoing edge. This figure points out more opportunities with *non-blocking fragmented* communication to overlap with *exec* phases, e.g. $\tau_{(C,2)}^w$ and $\tau_{(C,3)}^w$ overlap with τ_G^e, thus hiding the communication delay. Having prefetched all required data into the SPM, the *exec* phase of τ_G can start as soon as possible after task τ_C.

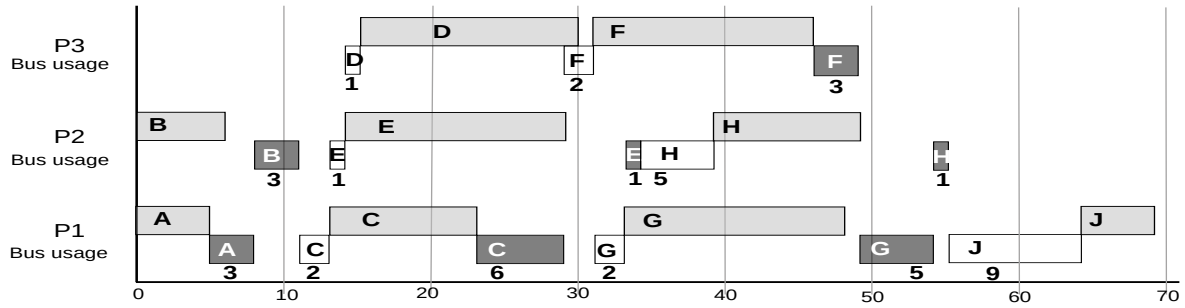
The gain in schedule length from Figure 4.3c is obtained by introducing the following flexibilities in the scheduling of communication fragments, while respecting *read-exec-write* phase's order:

i) communication phases of different tasks can have a different order than their respective *exec* phases, as long as there is no data dependencies between them, e.g. $\tau_{(B,2)}^w$ is scheduled after $\tau_{(E,1)}^r$, in reverse order compared to τ_B^e and τ_E^e.

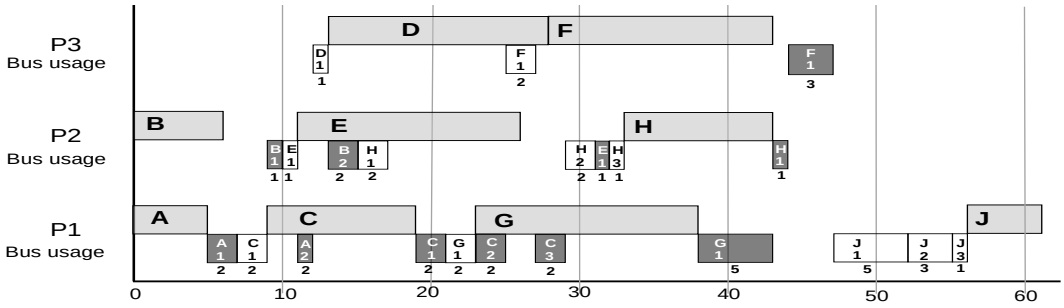
ii) communication phases and *exec* phase of the same task, do not need to be contiguous in time, e.g. $\tau_{(B,1)}^r$ and τ_B^e are not.



(a) Resulting schedule from Chapter 3 with *forced* contention-free communication, *blocking, non-fragmented* mode – overall makespan 71 time units



(b) (Intermediate) Resulting contention-free schedule with *non-blocking, non-fragmented* mode – overall makespan 69 time units



(c) Resulting contention-free schedule from Chapter 4, *non-blocking, fragmented* mode – overall makespan 61 time units

Figure 4.3 – Motivating example

iii) communications are fragmented, a task with multiple successors does not write all its data at once, e.g. task B has two successors, thus writing two fragments.

The last point (fragmented communications) is new, as compared to related works (Section 4.7). Considering each fragment individually allows additional overlaps between communications and calculations that were not previously possible. In the example, it allows to hide part of the *write* phase of task τ_A, and part of the *read* phase of task τ_B, which was not possible without fragmentation. Thus, splitting communications allows each source/sink of the task graph to hide part of its communications. However, in the example from Figure 4.3c, the remaining first

part A^w ($\tau^w_{(A,1)}$) can still not be hidden, but is however smaller than in Figure 4.3b. The overall makespan of the resulting schedule in *non-blocking* mode with fragmented communications (Figure 4.3c) is 61 time units, resulting in a gain of 20% over the *blocking* scheme.

4.4 SPM allocation scheme

In previous motivational example, we assumed the SPM large enough to store all information required to execute the entire application (code, data, communication buffers). To account for limited SPM capacity, our scheduling strategy comes with a SPM allocation strategy that allocates an SPM area (called hereafter *region*) to each communication fragment and execution phase. *Fragment-to-region mapping* is performed by the scheduler off-line. The same region can be used by different fragments, the scheduler guarantees that the time ranges of the concerned fragments do not overlap.

To isolate bus accesses from computation, we impose that *all* information accessed by a task is loaded into SPM beforehand. This comes in opposition to most SPM allocation policies that decide which information should be stored in the SPM and which information should remain in the global main memory (e.g. [DP07]). Our fragment-to-region mapping is inspired by the method proposed in [KBC+14].

The regions assigned to the fragments F^r_i contain the input data, fetched from the main memory, which are required by the task's *exec* phase. These regions contain the data produced by all predecessor tasks. The unique region assigned to τ^e_i contains any kind of information used locally by the task (code, constants, local data, usually stack-allocated). The regions assigned to F^w_i contain the data produced by the task.

The size of a region obviously depends on the amount of data required by the associated fragment (i.e. amount of data produced by a predecessor in case of a *read* fragment). Considering a mapping of tasks to cores and a mapping of fragments to SPM regions, the sum of the sizes of regions on a core must not exceed the SPM size.

Let us consider the example of Figure 4.3c, in which for simplicity we concentrate on the communication fragments and ignore the execution phases. If the size of the SPM is 1 Kbytes then on processor $P2$ the SPM can be partitioned in eight regions $SPM = \{\tau^w_{(B,1)}, \tau^w_{(B,2)}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(H,1)}, \tau^r_{(H,2)}, \tau^r_{(H,3)}, \tau^w_{(H,1)}\}$ with respective sizes in bytes $\{1, 2, 1, 1, 2, 2, 1, 1\}$ (according to the amount of data exchanged between tasks, taken from Figure 4.2). The sum of the regions' sizes is 11 bytes, which is less than the SPM size. If we now restrict the SPM size to 10 bytes, the previous partitioning of SPM in regions is not valid anymore. However, once $\tau^w_{(B,2)}$ is completed, the data produced by task τ_B has been committed to the global shared memory, therefore its assigned region can be reused. In this example, $\tau^w_{(H,1)}$ starts after the completion of $\tau^w_{(B,2)}$. Thus, the fragments $\tau^w_{(B,1)}$ and $\tau^w_{(H,1)}$ can be assigned to the same SPM region, leaving a partitioning of seven regions: $SPM = \{\tau^w_{(B,1)}, \{\tau^w_{(B,2)}, \tau^w_{(H,1)}\}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(H,1)}, \tau^r_{(H,2)}, \tau^r_{(H,3)}\}$ with respective sizes (in bytes) $\{1, max(2,1), 1, 1, 2, 2, 1\}$. The sum of all regions' sizes is 10 bytes, which can fit in the SPM.

In the example, the pair $(\tau^w_{(B,2)}, \tau^w_{(H,1)})$ could share the same region, because their lifespans do not overlap. On the other hand, in Figure 4.3c, $\tau^r_{(E,1)}$ cannot share the same region as $\tau^r_{(H,1)}$, because the data consumed by task τ_E are in use from the start of the read phase T^r_E up to the end of the execution of τ_E. This leads to define the *live range* of regions for each type of

fragment. Definition 1 defines the live range for a region assigned to a *read* fragment, while Definitions 2 and 3 give live ranges for regions assigned respectively to an *exec* and a *write* fragment.

Definition 1. Data fetched from the main memory by a *read* fragment are alive from its start time to the end of the *exec* phase.

Definition 2. Local information used by an *exec* phase (code, stack data area) are alive for the whole execution time of the application.

Definition 3. Data written back to main memory by a *write* fragment are alive from the start time of the *exec* phase to its transmission end time.

We assume read/written data can be consumed/produced at any time in the *exec* phase of the task. Therefore, the live range in Definitions 1 and 3 include the duration of the *exec* phase.

The scheduler maps fragments to regions, but does not decide the addresses of the regions in the SPM, which is left to the compiler/code generator. Since the number and size of regions is decided off-line, address assignment is straightforward, and does cause *external* fragmentation. Fragmentation of the SPM can only arise inside regions (*internal* fragmentation) when two (or more) phases are assigned to the same region but store different amounts of data.

4.5 Non-blocking contention-free scheduling techniques

This section modifies the scheduling framework introduced in Section 2.3 including both the ILP formulation and the heuristic algorithm. They are upgraded to integrate *fragmented non-blocking* communication and a *SPM allocation* scheme in contention-free scheduling techniques. The main outcome of both techniques is still a static partitioned time-triggered non-preemptive schedule, for one single application represented by a DAG.

4.5.1 Integer Linear Programming (ILP) formulation

Section 2.3.2 introduced the baseline scheduling ILP formulation. This section gives only modifications, additions or suppressions to apply. As a reminder, initial formulation included constraints aiming at:

- ensuring the unicity of mapping tasks on cores,
- detecting if two tasks are mapped on the same core,
- ordering tasks and particularly those on the same core,
- enforcing the AER execution model,
- enforcing that no pair of phases of the same type (computation/communication) overlaps on the same core ,
- enforcing data dependencies.

All these previous constraints remain un-modified. Identically, the objective remains: the overall schedule makespan minimisation. Moreover, boolean operators are still linearised using [BD07] as in Section 2.3.2.

Table 4.1 summarises the notations and variables removed from the initial framework in Chapter 2, while Table 4.2 summarises the one added to the ILP formulation.

Table 4.1 – Removed Notations & ILP variables

Constants	$DELAY_i^r$	task i *read*, *write* phases' WCET now need to be computed
	$DELAY_i^w$	by the ILP solver as the amount of interference depends on the schedule

Table 4.2 – Notations & ILP variables

Sets	F_i^r, F_i^w $F = \forall i \; inT, F_i^r \cup F_i^w$	sets of fragments for task $i \in T$ sets of all fragments from all tasks in T
	R	set of regions intialised with a region per phase
	$(j,g) \in pred(i,f)$	fragment g from task j is a direct predecessor of fragment f from task i
Fn	$i = task(f)$	utility to retrieve the task of a fragment, fragment f belongs to task i
Constants	SS_i	local (stack) data of τ_i^e
	CS_i	code size of τ_i^e
	$D_{(i,f)}^r, D_{(i,f)}^w$	data in bytes of fragment f in task i
	$DELAY_{(i,f)}^r$ $DELAY_{(i,f)}^w$	fragment f of task i, *read*/*write* latency from Equation (4.1)
	$SPMSIZE$	core local memory size (SPM). There is only one SPM size for every core, but specialising this constant per core is straightforward.
Int. vars	$\rho_{(i,f)}^r$, ρ_i^e, $\rho_{(i,g)}^w$	start times of *read* fragment f, *exec* phase and *write* fragment g of task i
	$\sigma_{(i,f)}$, σ_i	*spm reservation* start times of *read/write* fragment f, *exec* phase of task i
	$\omega_{(i,f)}$, ω_i	*spm reservation* end times of *read/write* fragment f, *exec* phase of task i
	$spmsr_z^c$	computed size of SPM region z on core c
Bin. vars	$a_{(i,f),(j,g)}^{\chi\psi} = 1$	$\tau_{(i,f)}^{\chi}$ is scheduled before $\tau_{(j,g)}^{\psi}$, in the sense $\rho_{(i,f)}^{\chi} \leq \rho_{(j,g)}^{\psi}$ $\chi \in \{r,w\}$ and $\psi \in \{r,w\}$
	$spmp_{z,i} = 1$ $spmp_{z,(i,f)} = 1$	τ_i^e is allocated to SPM region z fragment f from task i is allocated to SPM region z
	$spmm_{(i,f),(j,g)} = 1$	fragment f from task i and fragment g from task j are assigned to the same region (similar to $m_{i,j}$ from Table 2.2)
	$spma_{(i,f),(j,g)} = 1$	fragment f from task i is causally before fragment g from task j (similar to $a_{i,j}$ from Table 2.2)
	$spmam_{(i,f),(j,g)} = 1$	fragment f from task i is causally before fragment g from task j, and both are assigned to the same region (similar to $am_{i,j}$ from Table 2.2)

Computing the communication delays Calculation of communication delays for fragments f of a task i ($DELAY_{(i,f)}^r$ and $DELAY_{(i,f)}^w$ in the ILP formulation), is identical to computing the

WCET of the communication phase from Section 2.3.2 with equation (2.1d) (repeated below). But for the scope of this chapter, we generate contention-free schedules, thus no contention delay is paid when accessing the shared bus, and the duration of a data transfer of d bytes is very simply calculated by equation (4.1). This equation could be refined to account for DRAM access cost, as done in [KBC+14].

$$delay_i^\chi = T_{slot} \cdot waitingSlots_i^\chi \cdot interf_i^\chi + T_{slot} \cdot chunks_i^\chi + remainingTime_i^\chi \tag{2.1d}$$

$$DELAY_{i,f}^\chi = (D_{i,f}^\chi / D_{slot}) \cdot T_{slot} \tag{4.1}$$

Problem constraints Ordering fragments is similar to ordering exec phases. Hence, equation (2.5c) is duplicated into (4.2) but now targets fragments instead. When $a_{(i,f),(j,g)}^{\chi\psi} = 1$ then $\tau_{(i,f)}^\chi$ is scheduled before $\tau_{(j,g)}^\psi$.

$$\begin{aligned} \forall (i,j) \in T \times T; i \neq j, \{X,Y\} \in \{\{r,r\},\{w,w\},\{r,w\}\}, f \in F_i^X, g \in F_j^Y; \\ a_{(i,f),(j,g)}^{XY} + a_{(j,g),(i,f)}^{XY} = 1 \end{aligned} \tag{4.2}$$

In Equation (4.2), no equation forces to have the same ordering between *exec* phases than between *read* phases. The same remark applies to *exec* phases and *write* phases.

Absence of overlapping Equation (4.3) applies the ordering from the previous decision variable $a_{(i,f),(j,g)}^{\chi\psi}$. It forbids to have more than one active DMA request at a time to produce contention-free schedules. This equation is adapted from equation (2.7) which forbids more than one activated exec phase at a time per core.

$$\begin{aligned} \forall (i,j) \in T \times T; i \neq j, \{X,Y\} \in \{\{r,r\},\{w,w\},\{r,w\}\}, f \in F_i^X, g \in F_j^Y; \\ \rho_{(i,f)}^X + DELAY_{(i,f)}^X \leq \rho_{(j,g)}^Y + \mathcal{M}\ (1 - a_{(i,f),(j,g)}^{XY}) \end{aligned} \tag{4.3}$$

Equation (4.3) must be activated only if the two elements are scheduled in a specific order. Thus, a nullification method is applied using of a big-M notation [GNS09], detailed in Chapter 2, see equation (2.3).

Read-exec-write semantics constraints Equations (4.4a) and (4.4b) constrain the order of all phases of a task to be *read* phase, then *exec* phase, then *write* phase. But in contrast to Chapter 2, these phases will not necessarily be scheduled contiguously – i.e. replacing previous equations (2.6a) and (2.6b). The start date of the *exec* phase of task i (ρ_i^e) must be some time after the completion of all *read* fragments (start of *read* fragment $\rho_{(i,f)}^r$ + latency $DELAY_{(i,f)}^r$). Similarly, each *write* fragment ($\rho_{(i,f)}^w$) starts some time after the end of the *exec* phase (start of *exec* phase ρ_i^e + WCET C_i).

$\forall i \in T$,

$$\forall f \in F_i^r, \rho_i^e \geq \rho_{(i,f)}^r + DELAY_{(i,f)}^r \tag{4.4a}$$

$$\forall f \in F_i^w, \rho_{(i,f)}^w \geq \rho_i^e + C_i \tag{4.4b}$$

Data dependencies in the task graph To enforce data dependencies, equation (4.5) replaces equation (2.8) from Chapter 2. It constraints all *read* fragments to start after the completion of all their respective predecessors. For a *read* fragment its direct predecessor is the *write* fragment of the task that produced the corresponding data.

$$\begin{aligned} &\forall i \in T, \\ &\forall f \in F_i^r, \forall (j,g) \in pred(i,f); \\ &\rho_{(j,g)}^w + DELAY_{(j,g)}^w \leq \rho_{(i,f)}^r \end{aligned} \tag{4.5}$$

Assigning SPM region Mapping phases to region, and mapping tasks to cores are analogous. Equations (4.6a) & (4.6b) force every element (*exec* phase and fragments) from task i to be mapped on one and only one region z. These two equations mirror the integer variable $p_{i,c}$ from equation (2.5a) to map a task on a core.

$$\sum_{z \in R} spmp_{z,i} = 1 \tag{4.6a}$$

$$f \in F, i = task(f); \quad \sum_{z \in R} spmp_{z,(i,f)} = 1 \tag{4.6b}$$

Equations (4.7a) and (4.7b) set the size ($spmsr_z^c$) of region z on core c to be the largest amount of data that will be stored in it. The data stored by an *exec* phase includes both code (CS_t) and local data (SS_t, stack data). The data stored by a *read* or *write* fragment ($D_{(i,f)}^X$) includes all data consumed (or produced) by a task from one predecessor (or one successor). To store data into a given region of a core, both mapping variables for the region $spmp_{z,(i,f)}^X$ and the core $p_{i,c}$ must be set to 1.

$\forall c \in P, \forall z \in R, \forall i \in T,$

$$spmsr_z^c \geq \ (SS_i + CS_i)\ (spmp_{z,i} \wedge p_{i,c}) \tag{4.7a}$$

$$\forall \chi \in \{r, w\}, \forall f \in F_i^\chi; spmsr_z^c \geq \ D_{(i,f)}^\chi\ (spmp_{z,(i,f)} \wedge p_{i,c}) \tag{4.7b}$$

Equation (4.8) limits the sum of size for each region for a core to the available size of the SPM.

$$\begin{aligned} &\forall c \in P, \\ &\sum_{z \in R} spmsr_z^c \leq SPMSIZE \end{aligned} \tag{4.8}$$

Delimiting the usage time of a region by an element relies on Definitions 1, 2 and 3. Equation (4.9a) sets the allocation start time $\sigma_{(i,f)}$ of the *read* fragment f from task τ_i to be equal to its schedule start time and the allocation end time $\omega_{(i,f)}$ to be the end of the *exec* phase. Equation (4.9b) forces the lifetime of the region used by the *exec* phase to be the whole duration of the schedule (recall that Θ represents the overall makespan). Equation (4.9c) sets the allocation start time $\sigma_{(i,f)}$ of the write fragment f from task i equal to the beginning of the *exec* phase and the allocation end time $\omega_{(i,f)}$ equal to its arrival time.

$\forall i \in T$

$$\forall f \in F_i^r; \quad \sigma_{(i,f)} = \rho_{(i,f)}^r \quad and \quad \omega_i = \rho_i^e + C_i \tag{4.9a}$$

$$\sigma_i = 0 \quad and \quad \omega_i = \Theta \tag{4.9b}$$

$$\forall f \in F_i^w; \quad \sigma_{(i,f)} = \rho_i^e \quad and \quad \omega_{(i,f)} = \rho_{(i,f)}^w + DELAY_{(i,f)}^w \tag{4.9c}$$

Mapping elements (*exec* phases and communication fragments) to SPM region is very similar to mapping tasks on cores. Therefore, following equations (4.10a), (4.10b), (4.10c) and (4.10d) mimic the behaviour of respectively (2.5b), (2.5c), (2.5d) and (2.7) by replacing variables $m_{i,j}, a_{i,j}$ and $am_{i,j}$ with $spmm_{i,j}, spma_{i,j}$ and $spmam_{i,j}$. As a reminder, (4.10a) detects if two fragments are assigned to the same region from the same core, (4.10b) represents the causality of a fragment compare to another, and (4.10c) represents this causality on the same region. Finally, (4.10d) imposes the mutual exclusion of the reservation time .

$\forall (f,g) \in F \times F, f \neq g, i = task(f), j = task(g)$

$$spmm_{(i,f),(j,g)} = \sum_{z \in R} (m_{i,j} \wedge spmp_{z,(i,f)} \wedge spmp_{z,(j,g)}) \tag{4.10a}$$

$$spma_{(i,f)} + spma_{(j,g)} = 1 \tag{4.10b}$$

$$spmam_{(i,f),(j,g)} = spma_{(i,f)} \wedge spmm_{(i,j),(j,g)} \tag{4.10c}$$

$$\omega_{(i,f)} \leq \sigma_{(j,g)} + \mathcal{M}\ (1 - spmam_{(i,f),(j,g)}) \tag{4.10d}$$

4.5.2 Forward List Scheduling algorithm

This section details the modification added to the heuristic algorithm presented in Chapter 2. It now integrates the *non-blocking* and *fragmented* communication at schedule time with the SPM phase-to-region mapping scheme discussed in Section 4.4.

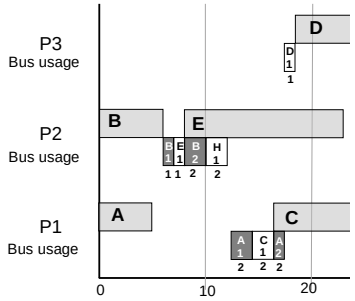


Figure 4.4 – Partial schedule of the task graph from Figure 4.2 with a DFS sorting strategy in the heuristic algorithm.

New sorting algorithm In addition to the framework presented in Chapter 2, this heuristic adds a third sorting algorithm for scheduled elements (tasks+fragments). The new sorting algorithm is a modification of the vanilla Depth First Search (DFS) that delays *read* fragments in the $Qready$ list. The purpose is to avoid too early reading that might delay other fragments in the schedule. When constructing the $Qready$ list from Figure 4.2 with a DFS exploration,

With a DFS exploration of Figure 4.2 and starting with task B, the $Qready$ list extracted can look like: $Qready = (\tau^e_B, \tau^w_{(B,1)}, \tau^r_{(E,1)}, \tau^e_E, \tau^w_{(E,1)}, \tau^r_{(H,3)}, \tau^w_{(B,2)}, \tau^r_{(H,1)}, \tau^e_A, \tau^w_{(A,2)}, \tau^r_{(D,1)}, \tau^e_D, \ldots)$. This would result in the partial schedule from Figure 4.4 (some of the listed fragment do not appear in the figure as they are placed later on the time line schedule). In this example, fragments $\tau^w_{B,2}$ and $\tau^r_{(H,1)}$ are scheduled before fragments from task A, task C and task D, with the consequence of retarding the beginning of these tasks. But τ^e_H cannot be scheduled right after $\tau^r_{(H,1)}$ due to other dependencies with task E and task C. Thus, the delay imposed to task C and D should be avoided.

This new sorting strategy postpones the insertion $\tau^w_{(B,2)}$ and $\tau^r_{(H,1)}$ in the $Qready$ list. The *DFS with delay* sorting algorithm inserts $\tau^r_{(H,1)}$ as soon as all its siblings ($\tau^r_{(H,2)}$ and $\tau^r_{(H,3)}$) are eligible for insertion (preceding fragments are in $Qready$). Moreover, it inserts $\tau^w_{(B,2)}$ in $Qready$ as soon as $\tau^r_{(H,1)}$ is eligible for insertion as well.

This leaves three sorting algorithms which are evaluated in Section 4.6.4 i) vanilla DFS, ii) vanilla Breadth First Search (BFS), and iii) DFS with delayed fragments. Nevertheless, no sorting algorithms outperform the other, because ordering is a topic on its own, the heuristic generates three schedule versions and selects the one with the shortest schedule length. Section 4.6.4 gives some hints to motivate three versions.

Forward list scheduling with non-blocking fragmented communications and SPM phase-to-region mapping Following the principle from previous Section 2.3.3, the forward list scheduling algorithm orders scheduled elements (line 3), then schedules them one by one as soon as possible (lines 6-23) without backtracking in order to minimise the overall makespan. If the element to schedule is a fragment (line 9), then there is no need to map it on a core, but it must be scheduled to avoid interference. If it is an *exec* phase, then a core is selected and the mapping minimises the overall makespan (line 16).

SPM regions can be assigned to a task, only once *all* its phases are properly scheduled

ALGORITHM 4.5.1: Forward list scheduling

Input : A task graph G $= (T, E)$ and a set of processors P
Output : A schedule

```
1  Function ListSchedule(G = (T, E), P)
2      Elist ← BuildListElement(G)
3      Qready ← TopologicalSortNode(Elist)
4      Qdone ← ∅
5      schedule ← ∅
6      while elt ∈ Qready do
7          Qready ← Qready \{elt}
8          Qdone ← Qdone ∪{elt}
9          if elt is a read fragment ∨ elt is a write fragment then
10             ScheduleElement(Qdone, elt, null)
11         else if elt is an exec phase then
               /* tmpSched contains the best schedule for the current task */
12             tmpSched ← schedule with makespan = ∞
13             foreach p ∈ P do
14                 copy ← schedule
15                 ScheduleElement(Qdone, elt, p)
16                 tmpSched ← min_makespan(tmpSched, copy)
17             schedule ← tmpSched
18         if all fragments and exec phase of the task i containing elt are in Qdone then
19             AssignRegion(schedule, Qdone, τ_i^e, SS_t + CS_t, 0, infinity)
20             foreach f ∈ F_i^r do
21                 AssignRegion(schedule, Qdone, f, D^r_(i,f), ρ^r_(i,f), ρ_i^e + C_i)
22             foreach f ∈ F_i^w do
23                 AssignRegion(schedule, Qdone, f, ρ_i^e, ρ^w_(i,f) + DELAY^w_(i,f))
24     return schedule
```

ALGORITHM 4.5.2: Build list scheduled element

Input : A task graph G $= (T, E)$
Output : A list

```
1  Function BuildListElement(G = (T, E))
2      return {F_i^r ∪ τ_i^e ∪ F_i^w | i ∈ T}
```

and mapped to a core (lines 19-23). When scheduling the read fragments, the core mapping information are not yet available. Additionally, when mapping the *exec* phase, we still do not have the information regarding the *write* fragments that have not been scheduled yet. While assigning the region (lines 19-23), the *exec* is allocated into a region first, then the communication phases. This order is motivated to better handle resident code in SPM and avoid SPM space to be stolen by communication fragments. For example, if there are 5 units of free space (not assigned yet) and the *exec* phase needs 5 units while two fragments need 2 units each, then the task can be mapped. The *exec* phase will take the remaining free space, while the communication fragments can share an already created but available (in time) region (see Definitions

1 and 3).

ALGORITHM 4.5.3: Schedule an element

Input : the list of scheduled element, the current element to schedule, the current core or null if the element is a fragment

Output :

1 **Function** `ScheduleElement`$(Qdone, cur_elt)$

`/* wct` $\rightarrow$ `Worst-Case Timing,` $DELAY^{\alpha}_{\beta}$ `or` C_{β} `*/`

`/* X and Y depend on the type of the corresponding element */`

2 $\rho^{X}_{cur_elt} \leftarrow max_{p \in pred(cur_elt)}(\rho^{Y}_{p} + wct_{p})$

3 **foreach** $e \in Qdone$ **do**

4 **if** cur_elt *is a fragment and* e *is not a fragment*

5 $\vee$ cur_elt *is an exec phase and* e *is not an exec phase*

6 $\vee$ cur_elt *is an exec phase and* e *is not mapped on core* cur_proc **then**

7 **continue**

8 **if** `are_OL`(e, cur_elt) **then**

9 $\rho^{X}_{cur_elt} \leftarrow \rho^{Y}_{e} + wct_{e}$

Scheduling an element In this version, scheduled elements include both all fragments and exec phases, as shown by Algorithm 4.5.2.

Algorithm 4.5.3 sketches the method to determine the start time of the considered element. First, each element must start after its causal predecessors (line: 2) Then, lines 3-9 enforce that no *exec* phase executes on the same core at the same time, and that no core accesses the bus at the same time (line 8).

ALGORITHM 4.5.4: Allocate a SPM region to a phase

Input : A schedule, the list of scheduled element, the current task and properties of the phase to map on a region

Output : A schedule

1 **Function** `AssignRegion`$(cur_elt, \quad data, start, end)$

2 **if** $data == 0$ **then return**

3 $proc \leftarrow$get the core on which the task containing cur_elt is mapped to

4 $existing \leftarrow$ Build the set of existing regions on core $proc$ where : size $\geq$ data $\wedge$ last reservation time ends before $start$

5 **if** $existing \neq \emptyset$ **then**

6 Assign the smallest existing region to cur_elt

7 **else if** $free\ SPM\ size\ in\ proc \geq data$ **then**

8 Create new SPM region for cur_elt on $proc$ with size $data$ where the reservation time is $[start; end]$

9 **else**

10 Throw Unschedulable

Allocation of SPM regions Algorithm 4.5.4 associates a SPM region to an element (*exec* phase, fragment). If there is data to store in the SPM (line 2), then it first tries to reuse an

existing region (lines 4-6), thus minimising the required memory size. If no existing region can be shared, then a new one is created (lines 7-8). Sharing a region imposes that the selected region is big enough to handle the current amount of data and free for use at the required time interval (line 4).

4.6 Experiments

This chapter presented a modification of the ILP formulation and the heuristic algorithm from Chapter 2. Therefore, the first presented experiments aim at validating the quality of the new approximate algorithm against the ILP formulation, as in previous experiments Section 2.4. Then, an empirical evaluation shows the benefits of non-blocking fragmented communication using the presented heuristic algorithm. The default fragmentation scheme is questioned to determine if finer fragmentation modes further increase the gain. Finally, the three sorting algorithms employed by the approximate algorithm are compared.

Experiments were conducted on real codes, in the form of the open-source refactored StreamIT [TKA02] benchmark suite: STR2RTS [RP17], as well as on synthetic task graphs generated using the TGFF [DRW98] graph generator.

Applications from the *STR2RTS benchmark suite* are modelled using fork-join graphs and come with timing estimates for each task and amount of data exchanged between them. We were not able to use all the benchmarks and applications provided in the suite for the very same reasons than in Section 3.4 (pre-requisite information extraction, linear chains of tasks, ...). A table summarising the used benchmarks is available in appendix (page 94).

The identical STG task graphs set from Chapter 2 is reused. Table 4.3 reminds the resulting parameters of the task set with the addition of two columns: i) the code size range for each task, ii) and their local storage size range.

Table 4.3 – Task graph parameters for synthetic task graphs

	#Task graphs	#Tasks	Width	WCET	Amount of bytes exchanged	Code size	Local size
		<min,max,avg>					
STG	50	5, 69, 22	3, 17, 8	$[5; 6000[$	$[0; 192]$	$[3; 3920[$	$[1; 60]$

As previously, all reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). Because Chapter 2 in Section 2.4.3 showed that the value of T_{slot} has a little impact on the schedule makespan, but it is rather better to keep it small, in all experiments $T_{slot} = 3$ as in [KHM+13]. The transfer rate is one word (4 bytes) per time unit.

4.6.1 Quality of the heuristic compared to the ILP

The following experiments aim at quantifying the over-approximation induced by the heuristic algorithm in the same manner than in Section 2.4.2. This gap is still expected to be small. Again, this experiment employs the STG task set with number of varying in $\in \{2, 4, 8, 12\}$. The heuristic is still implemented in C++ and CPLEX was configured with a timeout of 11 hours.

For each graph, we varied the number of cores in $\{2, 4, 8, 12\}$ and the sizes of the SPM vary in $\{2KB, 4KB\}$. SPM sizes allow to cover three situations i) all test-cases fit in the SPM (4KB size), ii) some test-cases do not entirely fit in SPM (2KB size), iii) some test-cases are too large, hence unschedulable (2KB size, biggest benchmarks).

Table 4.4 – Degradation of the heuristic compared to the ILP on the synthetic task-graphs

% of exact results (ILP only)	degradation <min,max,avg> %
68%	0%, 20%, 3%

Table 4.4 presents the combined results for all different configurations. First, it shows the number of optimal (including unfeasible) results the ILP solver is able to find in the given time-out – 68%. The remaining 32% includes all other cases where the solver reaches the timeout without neither an optimal solution nor an infeasibility verdict. Then Table 4.4 presents the minimum/maximum and average degradation induced by the heuristic over the ILP. As displayed, the average degradation is low thus showing the quality of our heuristic.

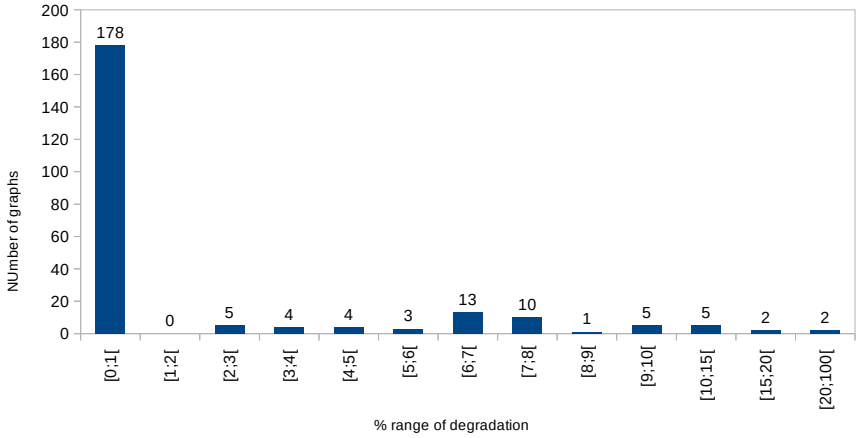


Figure 4.5 – Distribution degradation heuristic vs ILP (232 test-cases)

Figure 4.5 presents the degradation of the heuristic compared to the ILP. Most of the results (77%) have a degradation below 1%. For all cases we investigated, the degradation is due to the topological order at the beginning of the heuristic and the non-backtracking policy implied by forward-list-scheduling technique. In addition, we have 10% (40 graphs) of the test-cases for which the ILP is able to find an optimal solution whereas the heuristic answer is unfeasible (no schedule found due to lack of SPM memory space). Again, this comes from the non-backtracking policy of forward-list-scheduling technique.

4.6.2 Blocking vs non-blocking communications

Further experiments rely only on the heuristic as most task-graphs from STR2RTS benchmark suite are too large to be scheduled using the ILP solver in reasonable time. To compare

the benefit of hiding communication latency, the method must be opposed to a scheduler that does not hide it. We preferred to modify our heuristic to implement both the *blocking* and *non-blocking* methods instead of reusing a state-of-the-art algorithm. The main reason, as detailed in Section 4.7, is that related work have characteristics that are hardly compatible with our proposal: different task model [TMW+16], SPM big enough to store all code/data [BDN+16; RDP17], lack of information on SPM management [BMD+18], different interconnect [DFG+14]. Another reason for this choice is to guarantee that the deviation between the results from the two communication modes will not be affected by any other technical implementation decision (e.g.: sorting algorithm).

To summarise the modifications applied to the heuristic in order to get the blocking mode: i) we forbid to have more than one phase active at a time (both communication and computation as in the example of Figure 4.3a) ii) we do not fragment communications. As previously, we varied the number of cores in $\{2, 4, 8, 12\}$, and the SPM sizes in $\{4KB, 2MB\}$ ($2MB$ is the SMEM size in one cluster of the Kalray MPPA [DVP+14]). All aforementioned three situations regarding the SPM size are covered with these configurations. Note that STR2RTS benchmarks are larger in term of memory space than synthetic benchmarks. We then calculate the gain of the *non-blocking* mode versus the *blocking* mode that we expect to be positive.

$$gain = \frac{blocking - NONBlocking}{blocking} * 100 \tag{4.11}$$

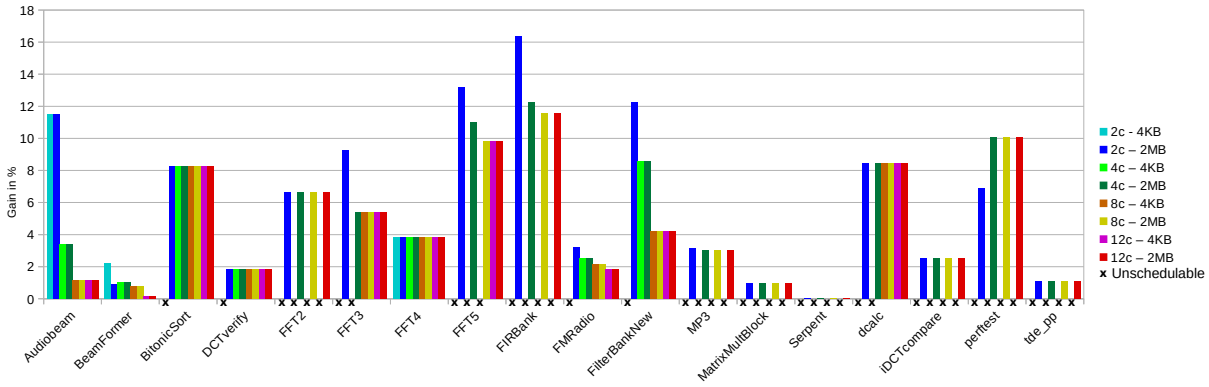


Figure 4.6 – Gain of *non-blocking* communications over *blocking* on STR2RTS benchmarks per cores/SPM configuration – avg: 4%

Figure 4.6 presents the average gain per benchmark for all configurations (computed using Equation (4.11)), e.g. *2c-2MB* stands for *2-cores and SPM size of 2MB*. The maximum gain is 16% (*FIRBank* on 2 cores with 2MB SPM), whereas the average is 4%.

Figure 4.6 shows that some benchmarks are *unschedulable* for some configurations, e.g. *FFT2* with 2c-4KB. This comes from a lack of SPM space to place all code and all data. This might be relaxed with code pre-fetching in *read* phase, which is left for future work.

Lower gains are observed when the amount of parallelism is low due to the lack of opportunity to hide communications. For example, *Serpent* is a chain of fork-joins containing 2 concurrent tasks only, as opposed to *FIRBank* which includes only one fork-join construct with several long chains of tasks. In addition, higher gains are observed on hardware configurations with lower number of cores – i.e. 6% on average with 2-cores as opposed to 4% with 12-cores.

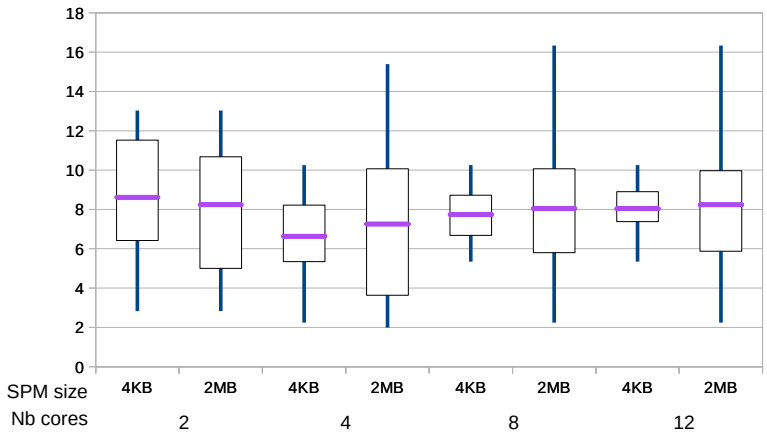


Figure 4.7 – Gain of *non-blocking* communications over *blocking* on TGFF benchmarks – average: 8%

To evaluate the impact of graph shapes on gains, we experimented our heuristic technique on synthetic task graphs, the ones used previously to validate the heuristic. In contrast to STR2RTS graphs, that are fork-join graphs, synthetic task graphs are arbitrary graphs. Results are depicted in Figure 4.7. We observe that arbitrary graphs offer more opportunities to hide communication, with an average gain of 8% in total.

4.6.3 Impact of fragmentation strategy

Through the chapter, we have split *read*/*write* phases according to tasks dependencies (one fragment per edge in the task). We experimented with two more fine-grain splitting strategies:

— splitting by D_{slot}: each fragment will fit in a T_{slot} bus period, each transmitting D_{slot} bytes – a task transmitting 5 floats (20 bytes) with a $T_{slot} * D_{slot}$ of $3 * 4$ bytes per request will result to 2 fragments, generating 2 communications.

— splitting by *data-type unit (DTU)*: an application exchanging only *floats* will have a DTU of 1 float (4 bytes). If a task produces 5 floats, then there is 5 fragments.

We conducted the experiments by applying our heuristic on the STR2RTS benchmarks, with the very same experimental setup as before. We include in the comparison scheduling in *non-blocking* mode without communication fragmentation (label *no frag* in Figure 4.8). We expect the gain to increase as the fragment granularity gets smaller.

Figure 4.8 presents the results with four granularities: *no frag*, *edge* (default configuration), D_{slot} (12 bytes) and *DTU* (4 bytes). Fragmenting communications always result in shorter schedules than the *no frag* configuration. In addition, in most cases the smaller granularity results in higher gains. However the better the results are, the higher the schedule generation time is, as given in the legend of the figure. Schedules are generated in less than 1 second on average for *no frag* and *edge*, whereas several minutes are required on average for fine-grain fragments.

To validate our theory on a real platform, we successfully implemented schedules generated with our heuristic targeting one cluster of a Kalray MPPA Bostan platform [DVP+14]. The

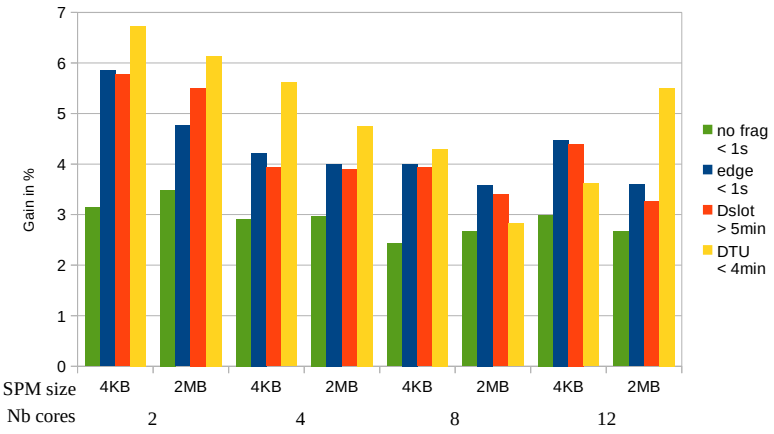


Figure 4.8 – Average gain of *non-blocking* schedule length over *blocking* one depending on fragmentation strategy

final code is largely auto-generated (only the code of the exec phase of each task has to be inserted manually in the generated code). At the time of writing, we managed to run benchmark *BeamFormer_12ch_4b* from the STR2RTS benchmark suite [RP17]. This benchmark is made of 56 tasks with a DAG width of 12.

A master core, in the cluster, is reserved to act as a software DMA engine as in [CLC+09]. Then the thread on that core contains the statically generated sequential schedule of *read*/*write* phases only. This ensures a contention-free scenario to access the memory (no read and write can be concurrently active), and emulate the behaviour of a hardware DMA engine that acts on behalf of other computing cores.

We were able to generate the following versions of the benchmark:

— *blocking* mode (S_{bl}),

— *non-blocking* mode without communication fragmentation (S_{nbl}),

— *non-blocking* mode with fragmentation by edges (S_{nbl}^{edge}),

— *non-blocking* mode with fragmentation by D_{slot} (12 bytes) (S_{nbl}^{dslot}),

— *non-blocking* mode with fragmentation by DTU, fragment size: 4-bytes (1 float) (S_{nbl}^{dtu}).

In terms of implementation overheads, there is no overhead to set up the software implemented DMA at run-time, since *channel connectors* are initialised only once at application start. A jitter of 32 cycles due to the scheduling of communication and execution phases is taken into account.

For this experiments, WCETs of computations and communications were estimated using measurements, adding an arbitrarily chosen margin of 20% for safety. Taking into account implementation overheads, as expected, the overall schedule makespans are: $S_{bl} > S_{nbl} > S_{nbl}^{dslot} > S_{nbl}^{edge}$. The gain of S_{nbl} schedule over the S_{bl} schedule is 1%, the gain of S_{nbl}^{edge} schedule over the S_{nbl} schedule is 36%, and the gain of $S_{n}^{dslot}bl$ over S_{nbl} is 22%.

However, the finer fragmentation policy suffers from an overhead on this platform. The degradation of S_{nbl}^{dtu} over S_{nbl}^{edge} is 24%. The source of this overhead mainly originates from

read phases measured time where reading one float takes as much time as reading four floats. Nevertheless, we observe a small decrease in *write* phases measured time depending on the amount of data exchanged (approximately 1000 cycles on average).

4.6.4 Impact of topological sorting algorithm

In the heuristic, Algorithm 4.5.1, we first build a sorted list of all elements to be scheduled (execution phases+fragments). As mentioned before, we use three different topological sorting algorithms: i) a vanilla DFS, ii) a DFS with delayed fragments, iii) a vanilla BFS. We could not find one sorting algorithm outperforming the others for all graphs and configurations (number of cores, SPM size). Using the same experimental setup as before, we observed an average difference in the schedule makespan of 3% with a maximum of 18%.

4.7 Related Work

Accessing the global shared memory has always been the performance bottleneck. To overcome this issue, a prefetching mechanism brings data/instructions closer to the processor before it is actually needed. Hardware prefetchers will speculatively request data or instructions based on access pattern [Mic16]. Software prefetchers give control to the developer or compiler to introspect the code by adding prefetching instructions [KL91]. Hybrid prefetchers usually use helper-threads to prefetch data and some synchronisation hardware support [KST11]. In this chapter we propose a *software* prefetcher that adds prefetching based on a schedule generated off-line.

Most of other works considering SPM aim at deciding what should be stored into the SPM and when to evict data, and in cases some information cannot be stored in SPM it stays in main memory. Considered metrics for SPM allocation are *average-case* performance [DHC+13; LXK10], *power consumption* [TTT10], *WCET* [DP07], schedule makespan [BMD+18]. In contrast to these studies, our work, in order to control resource contention, requires all information to be stored in SPM.

Wasly and Pellizzoni [WP13] add a hardware component to manage the SPM, a Real-Time Scratchpad Memory Unit (RSMU). This RSMU acts similarly as a traditional Memory Management Unit (MMU) except it will also use a previously computed schedule for loads/unloads of code/data from mixed-critical tasks. To use our method, no specific hardware component needs to be added ; only a DMA engine that we believe to be quite common nowadays. In addition, we do not account for mixed-critical tasks. Giorgi et al. [GPP09] introspect the code to add the RSMU behavior in order to prefetch global data from the global external memory into a local memory on many-core tiled architecture. They modified the compiler to isolate load into specific basic blocks and added synchronisation mechanism (before the usage of the data) to block the thread if the mandatory data are not ready for use. However their study do not include any real-time guarantee on the blocking time of the threads. We can guarantee the data will be ready for use without blocking time.

Kim et al. [KBC+14] present an algorithm to map a function to a specific SPM region, that inspired our phase to region mapping step. They aim at storing the basic blocks into the SPM

in order to improve the WCET of an application on a single-core. We improve their work to map multiple tasks on multi-cores.

Che et al. [CC11] schedule stream applications on SPM-based single-core architectures. They also split the SPM in region but with specific purposes. One region is reserved for libraries and global data, one region per task, two regions for the stack and the head and one more used as exchanged buffer for inter-tasks communication.

Cheng et al. [CCR+17] derive a speed-up and a resource augmentation factors when partitioning memory banks with minimum interference. At the opposite we have a complete off-line schedule with phase to region allocation on single bank SPM memory.

On a single-core, using PREM, Soliman et al. [SP17] hide the communication latency at a the basic-block level. They modify the compiler toolchain LLVM to hide this latency while other parts of the code are executing. Wastly and Pellizzoni [WP14] proposed to dynamically co-schedule, without preemption, DMA accesses and sporadic tasks on a SPM-based single-core. The SPM is split in 2 parts: one assigned to the currently executing task, while the other load information for the next scheduled task. Our work makes a better use of the SPM by allowing more than two regions alive at the same time. This last work has been extended to multi-core in [AWP15].

In order to reduce the impact of communication delays on schedules, [GTK+02; CLC+09] hide the communication request while a computation task is running. This accounts with the asynchronism implied by DMA requests. However they use a worst-case contention. And neither SPM management nor fragmented communication are employed in this article.

The technique proposed in [BMD+18] generates contention-free off-line schedules with periodic dependant tasks. Dealing with the SPM, they aim at deciding if a task should be resident in SPM or be fetched before each execution from the global memory. Unfortunately they do not provide information on SPM allocation, raising questions about address allocation and SPM fragmentation. With our region allocation scheme, SPMs are allocated while managing fragmentation is proposed.

A technique to hide transfers behind calculations is presented in [TMW+16]. Similarly to [WP14] and [AWP15], the SPM is split in two regions, one used by the application while the other is being loaded. Our work differs from the work in [TMW+16] by the task model under use (dependant tasks in our work, sporadic independent tasks in their work). Moreover, our work make better use of SPM by allowing more than two SPM regions to be alive simultaneously.

Kudlure et al [KM08] created a full compilation chain including partitioning, mapping, scheduling, buffer allocation while hiding memory latencies. To hide memory access latencies, they benefit from pipelining mechanism which allows to schedule several iterations in a time window. Thus, the communication initiated jobs of an iteration are hidden under the execution of jobs from next iterations. In our work, communication are fragmented and hidden under computation of the same iteration.

The work presented in [DFG+14] proposes an off-line scheduling scheme for flight management systems using a PREM-like task model. The proposed schedule avoid interferences to access the communication medium. However, in contrast to our work, there are still interferences in their schedule, due to communications between tasks assigned to different cores.

Other works very close to our research, such as [KM08; CLC+09; TPG+14; SS17], statically schedule applications represented by SDF with some form of buffer checking. However, they do not use the PREM/AER model like us [TPG+14; SS17], and none of them fragment the

communications, which allows us to drastically increase the hiding opportunities.

4.8 Conclusion

This chapter showed how to minimise the impact of the communication latency when mapping/scheduling a task graph on a multi-core, by overlapping communications and computations. We also argued that this kind of technique should always be coupled with a memory allocation scheme to guarantee the integrity of the accessed data. The presented approach fragments communication to increase the possibility of overlapping them with computation, which we use to define a mapping/scheduling mapping strategy. This strategy is implemented in an ILP formulation that formally describes the solved problem, and with an approximate algorithm that solves the problem faster. The empirical evaluation demonstrates the validity of the heuristic algorithm by showing that the over-approximation, induced by such technique, remains low (3% in average). Then, the experimental results show that, compared to a scenario not overlapping communications and computations, this approach improves the schedule makespan by 4% on average on streaming applications (8% on synthetic task graphs). Nevertheless, experiments also show that allowing different fragmentation strategies can further increase the gain when avoiding small transfer overhead, up to 6.5% with streaming applications. In addition, we evaluated different sorting algorithm within our heuristic, but could not find one that outperforms the other.

This conclusion, regarding smaller size fragmentation, must be balanced with the context of this dissertation. We did not include an overhead for the creation and transfer of fragments. When dealing with a real system, and not just an abstraction, overheads must be added for the creation of the fragment, for the time-triggered implementation of the schedule and for possible other DMA costs. However, this cost is constant and can be added to the computation of the delay. In order to validate our theory on a physical system, we successfully implemented schedules generated with our heuristic algorithm targeting one cluster of a Kalray MPPA Bostan platform [DVP+14]. Implementation details are available in the publication containing the full study from this chapter [RSD+19]. On this real platform, the observed gain for the fragmentation by edge, D_{slot} and DTU over *non-blocking* and *non-fragmented* schedules are respectively: 36%, 22%, and 24%.

The code of this scheduler using both techniques (ILP, heuristic) with fragmented and hidden communication is available at `https://gitlab.inria.fr/brouxel/methane`. Also, the code generated for the Kalray MPPA is available in the resource folder of the same project.

CONCLUSION

Designing a hard real-time system requires more attention and strict guarantees, especially on timing constraints, than any other systems. To enforce the strongest guarantees, this worst-case performance is computed *a priori* with, among other analyses, static WCET estimations, and static scheduling policies. Multi-core platforms are a very attractive solution to implement hard real-time systems. When specifically designed with predictability in mind, they provide both performance and the ability to devise tight worst-case performance. Moreover, parallel applications provide means to fully exploit all available resources within multi-core architectures. The best representation for parallel applications explicitly provides both dependency and concurrency between tasks, as these information are mandatory to compute tight worst-case performance on multi-core hardware in a static manner. To run a parallel application on a multi-core architecture, a scheduling policy describes on which core and when tasks is executed.

This dissertation presents two static scheduling methods to optimise the global performance of given application while enforcing strong temporal constraints. Both can be classified as static partitioned and non-pre-emptive. The implementation of all proposed scheduling strategies include both an ILP formulation and a heuristic algorithm loosely based on greedy heuristic similar to list scheduling. ILP formulations provide a non ambiguous description of the problem under study, and is also used as a baseline to evaluate the quality of our proposed heuristic algorithm. Our scheduling strategies target multi-core platforms in which cores use a bus based interconnect arbitrated with a FAIR-RR policy. Each core is also assumed to have access to a local private memory, which is a SPM. We consider parallel applications represented as acyclic task graphs. To take advantage of the SPM-based architecture, we employ an execution model where memory accesses are isolated from computation phases, following the AER [MNP+16] principle. This separation allows us to first read input data from the global memory to the SPM, then execute the code without interferences with other cores, and finally write back the produced data from the SPM to the external memory. In our approach, communicating tasks are assumed, in the whole document, to only transfer data through the main memory (SPM to SPM communication are not considered). We first restrict ourself to *blocking* communications in Chapters 2 and 3, and show how to support *non-blocking* communications in Chapter 4. In essence, the goal of this work is to minimise the impact of these communications on the overall schedule makespan of the application.

With multi-core architectures, several cores can access the shared bus at the same time instant. Therefore computing transfer latencies, between cores and the off-chip memory through the bus, must consider a contention delay, which usually depends on the number of contending cores. The computation of communication latencies in Chapter 2 used a worst-case contention delay, which always considers that all other cores may also access the shared bus, at the same time instant. Even if very pessimistic, this provides a baseline against which we compare when dealing with contention awareness in Chapter 3. Empirical evaluation shows that solving scheduling problem while looking for an exact optimum does not scale with large problems (as expected). For test cases involving a large number of tasks, the only way to obtain a solution it to rely on an approximate method. Then, an empirical evaluation, in Section 2.4.3, shows

that specific bus parameter T_{slot}, in a FAIR Round-Robin bus, has a negligible impact over the schedule length.

The worst-case contention model is a safe choice by construction, but it leads to a large over-approximation which is then refined in Chapter 3. In this chapter, our proposed scheduling method uses the knowledge of the application structure and the knowledge of the currently considered schedule to refine, at design time and for each communication phase, the effective worst-case amount of interferences. This method is shown to be effective with the empirical evaluation from Section 3.4, where we show an average improvement of 59% over the worst-case scenario. However, experiments also show that, in most cases, allowing the scheduler to choose between contention-free and contention-aware scheduling methods results in schedules free from contention. This observation must, however, be put in perspective with the fact that we only consider task graph with AER semantics.

To further refine the contention delay, the contribution in Chapter 4 describes a static scheduling method that generates contention-free schedules with non-blocking communications. The scheduler takes advantage of DMA transfers and a dual-ported SPM to hide bus accesses while the processing unit is busy with computations. We also show that this kind of technique should always be coupled with a memory allocation scheme to guarantee the integrity of the accessed data. The presented approach fragments communications to increase the opportunities of overlapping them with computation. The empirical evaluation demonstrates that, compared to a scenario without overlapping, this approach improves the schedule makespan by 4% on average on streaming applications (8% on synthetic task graphs). Nevertheless, experiments also show that allowing different fragmentation strategies can further increase the gain when avoiding small transfer overhead, up to 6.5% with streaming applications. In addition, we evaluate different sorting algorithms within our heuristic, but could not find any of them that outperformed the other. Finally, we successfully implemented a scheduled program generated with our heuristic algorithm targeting one cluster of a Kalray MPPA Bostan platform [DVP+14]. On this platform, the observed gain for the fragmentation by edge, D_{slot} and DTU over *non-blocking* and *non-fragmented* schedules were respectively: 36%, 22%, and 24%. With gains beyond our expectations, this implementation validates the advantages of our scheduling strategy.

The whole source code of the scheduler used along this thesis is available at `https://gitlab.inria.fr/brouxel/methane`.

Future Work

All good thinks eventually end, so is this dissertation, which leaves room for further improvements.

Our first contribution generates contention-aware schedules in which the effective contention is computed at design time. However this contribution includes an over-approximation that can be refined to further optimize the overall schedule makespan. Indeed, two communication phases with partially overlapping worst-case access time do not suffer from interferences for this entire time interval. It is therefore possible to further refine reduce this over-approximation, in order to further reduce the schedule length.

Moreover, this work considers only non-pre-emptive scheduling techniques. We believe that

our approach could be extended to include pre-emptive schedules. Pre-emptive scheduling techniques may generate new additional interferences when accessing the code and data for pre-emptive tasks. Also, the memory allocation scheme needs to be re-evaluated to guarantee the integrity of the data stored inside the SPM.

This work did not address the problem of off-chip memory accesses, considering DRAM-based platform. The worst-case access time to data in DRAM banks is impacted by several factors such as the locality and the DRAM refresh cycles. Different strategies could be exploited by a scheduler to improve the worst-case access delay of communication phases/fragments. For example, memory accesses could be scheduled at a time where no extra delay are added due to DRAM refresh cycles, or the choice of fragmentation could exploit DRAM row locality and read/write switching of the accesses.

An aspect that could be further explored is fine-grained fragmentation of the data into categories such as read-only data. This may help in the following way: read-only data used by multiple tasks deployed on different cores could potentially be loaded simultaneously. Assuming that the architecture is bus based, then if the data is broadcasted on the bus, then multiple SPM controllers could pick up the data as it is streamed by the DMA engine.

This work addresses scheduling techniques targeting bus-based multi-core architectures. Because network on chip based many-core architectures are getting more and more attention, a natural extension of this work would be the inclusion of such interconnect architectures. As a matter of fact, since network on chip links carry packets, we expect that communication fragmentation to have even more benefits.

Appendices

STR2RTS benchmark suite

Following Table 4.5 characterises used benchmarks from STR2RTS benchmark suite. The first column presents the number of tasks and the second column the width of the graph. Then, it gives the average data in bytes sent along all edges. Following is the average memory footprint of all tasks within a benchmark, it includes the code size and the stack size. Last column shows average, among all tasks, WCET estimates. All this information are shipped with the benchmark suite and target a Patmos single core architecture [SBH+15].

Table 4.5 – Benchmarks characteristics

Name	#Tasks	Width	avg data (bytes)	average memory footprint (bytes)	average WCET (time unit)
Audiobeam	20	15	12 B	108 B	41
Beamformer	56	12	18 B	246 B	2718
BitonicSort	122	8	49 B	109 B	30
DCTverify	7	2	513 B	506 B	10045
FFT2	26	2	551 B	2 KB	618
FFT3	82	16	84 B	208 B	120
FFT4	10	2	6 B	32 B	11
FFT5	115	16	52 B	1 KB	38
Firbank	340	12	505 B	2 KB	670
FMRadio	67	20	6 B	191 B	235
FilterbankNew	53	8	35 B	180 B	144
MP3	116	36	3502 B	19 KB	12222
MatrixMultiBlock	23	2	793 B	1 KB	726
Serpent	234	2	1013 B	709 B	922
dcalc	84	4	106 B	685 B	174
IDCTcompare	13	3	454 B	685 B	4557
perftest	16	4	8267 B	21 KB	5269
tde_pp	55	2	25344 B	16 KB	2931

BIBLIOGRAPHY

[AB11] Sebastian Altmeyer and Claire Maiza Burguière, « Cache-related preemption delay via useful cache blocks: Survey and redefinition », *in*: *Journal of Systems Architecture* 57.*7* (2011), pp. 707–719 (cit. on p. 33).

[ABD08] James H Anderson, Vasile Bud, and UmaMaheswari C Devi, « An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems », *in*: *Real-Time Systems* 38.*2* (2008), pp. 85–131 (cit. on p. 31).

[ABJ01] Björn Andersson, Sanjoy Baruah, and Jan Jonsson, « Static-priority scheduling on multiprocessors », *in*: *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, IEEE, 2001, pp. 193–202 (cit. on p. 31).

[ABS13] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl, « A time-predictable stack cache », *in*: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, IEEE, 2013, pp. 1–8 (cit. on p. 22).

[Ack82] William B. Ackerman, « Data flow languages », *in*: *Computer 2* (1982), pp. 15–25 (cit. on p. 18).

[ADI+15] Sebastian Altmeyer, Robert I Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke, « A generic and compositional framework for multicore response time analysis », *in*: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ACM, 2015, pp. 129–138 (cit. on p. 65).

[ADM11] Sebastian Altmeyer, Robert I Davis, and Claire Maiza, « Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems », *in*: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, IEEE, 2011, pp. 261–271 (cit. on p. 33).

[AEF+14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al., « Building timing predictable embedded systems », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 13.*4* (2014), p. 82 (cit. on p. 24).

[AP14] Ahmed Alhammad and Rodolfo Pellizzoni, « Time-predictable execution of multithreaded applications on multicore systems », *in*: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, IEEE, 2014, pp. 1–6 (cit. on pp. 31, 64, 66).

[AWP15] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni, « Memory efficient global scheduling of real-time tasks », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, IEEE, 2015, pp. 285–296 (cit. on pp. 25, 89).

[Bak91] Theodore P. Baker, « Stack-based scheduling of realtime processes », *in*: *Real-Time Systems* 3.*1* (1991), pp. 67–99 (cit. on p. 33).

[BB08] Sanjoy Baruah and Enrico Bini, « Partitioned scheduling of sporadic task systems: an ILP-based approach », *in*: *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image Processing*, Citeseer, 2008 (cit. on pp. 12, 30).

[BBY13] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao, « Limited preemptive scheduling for real-time systems. a survey », *in*: *IEEE Transactions on Industrial Informatics* 9.*1* (2013), pp. 3–15 (cit. on p. 30).

[BC84] Gérard Berry and Laurent Cosserat, « The ESTEREL synchronous programming language and its mathematical semantics », *in*: *International Conference on Concurrency*, Springer, 1984, pp. 389–448 (cit. on p. 20).

[BCG+99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok, « Generalized multiframe tasks », *in*: *Real-Time Systems* 17.*1* (1999), pp. 5–22 (cit. on p. 18).

[BCS+16] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, and Gilles Muller, « Maximizing parallelism without exploding deadlines in a mixed criticality embedded system », *in*: *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, IEEE, 2016, pp. 109–119 (cit. on p. 66).

[BD07] Gerald G Brown and Robert F Dell, « Formulating integer linear programs: A rogues' gallery », *in*: *INFORMS Transactions on Education* 7.*2* (2007), pp. 153–159 (cit. on pp. 41, 56, 75).

[BDN+16] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte, « Contention-free execution of automotive applications on a clustered many-core platform », *in*: *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, IEEE, 2016, pp. 14–24 (cit. on pp. 7, 11, 17, 23, 25, 30, 31, 35, 52, 64, 65, 66, 85).

[BDT13] Shuvra S Bhattacharyya, Ed F Deprettere, and Bart D Theelen, « Dynamic dataflow graphs », *in*: *Handbook of Signal Processing Systems*, Springer, 2013, pp. 905–944 (cit. on p. 20).

[BDW+12] Alan Burns, Robert I Davis, P Wang, and Fengxiang Zhang, « Partitioned EDF scheduling for multiprocessors using a C = D task splitting scheme », *in*: *Real-Time Systems* 48.*1* (2012), pp. 3–33 (cit. on p. 32).

[BEL+96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete, « Cycle-static dataflow », *in*: *IEEE Transactions on signal processing* 44.*2* (1996), pp. 397–408 (cit. on p. 20).

[Ber06] Christoph Berg, « PLRU cache domino effects », *in*: *OASIcs-OpenAccess Series in Informatics*, vol. 4, Schloss Dagstuhl-Leibniz-Zentrum f"ur Informatik, 2006 (cit. on p. 22).

[BMD+18] Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte, « Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints », *in*: *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, IEEE Press, 2018, pp. 560–567 (cit. on pp. 69, 85, 88, 89).

[BMV+15] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna, « A memory-centric approach to enable timing-predictability within embedded many-core accelerators », *in*: *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, IEEE, 2015, pp. 1–8 (cit. on p. 25).

[CBS14] Emanuele Cannella, Mohamed A Bamakhrama, and Todor Stefanov, « System-level scheduling of real-time streaming applications using a semi-partitioned approach », *in*: *Proceedings of the conference on Design, Automation & Test in Europe*, European Design and Automation Association, 2014, p. 363 (cit. on p. 32).

[CC10] Weijia Che and Karam S Chatha, « Scheduling of synchronous data flow models on scratchpad memory based embedded processors », *in*: *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, IEEE, 2010, pp. 205–212 (cit. on p. 29).

[CC11] Weijia Che and Karam S Chatha, « Scheduling of stream programs onto SPM enhanced processors with code overlay », *in*: *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, IEEE, 2011, pp. 9–18 (cit. on pp. 29, 89).

[CCR+17] Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo, « Memory Bank Partitioning for Fixed-Priority Tasks in a Multi-core System », *in*: *Real-Time Systems Symposium (RTSS), 2017 IEEE*, IEEE, 2017, pp. 209–219 (cit. on pp. 35, 89).

[CEN+13] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel, « Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms », *in*: *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, IEEE, 2013, pp. 1–10 (cit. on p. 20).

[CFL+05] Jacques Combaz, Jean-Claude Fernandez, Thierry Lepley, and Joseph Sifakis, « QoS control for optimality and safety », *in*: *Proceedings of the 5th ACM international conference on Embedded software*, ACM, 2005, pp. 90–99 (cit. on p. 29).

[CFL+18] Gruia Calinescu, Chenchen Fu, Minming Li, Kai Wang, and Chun Jason Xue, « Energy optimal task scheduling with normally-off local memory and sleep-aware shared memory with access conflict », *in*: *IEEE Transactions on Computers 1* (2018), pp. 1–1 (cit. on p. 30).

[CGJ96] Edward G Coffman Jr, Michael R Garey, and David S Johnson, « Approximation algorithms for bin packing: a survey », *in*: *Approximation algorithms for NP-hard problems*, PWS Publishing Co., 1996, pp. 46–93 (cit. on pp. 14, 29, 30, 36, 44, 47).

[CHO12] Franck Cassez, Ren'e Rydhof Hansen, and Mads Chr Olesen, « What is a Timing Anomaly? », *in*: *OASIcs-OpenAccess Series in Informatics*, vol. 23, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012 (cit. on p. 22).

[CLC+09] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge, « Stream compilation for real-time embedded multicore systems », *in*: *Code generation and optimization, 2009. CGO 2009. International symposium on*, IEEE, 2009, pp. 210–220 (cit. on pp. 30, 70, 87, 89).

[CM12] Daniel Cordes and Peter Marwedel, « Multi-objective aware extraction of task-level parallelism using genetic algorithms », *in*: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, IEEE, 2012, pp. 394–399 (cit. on p. 20).

[COK+12] Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha, « Executing synchronous dataflow graphs on a SPM-based multicore architecture », *in*: *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012, pp. 664–671 (cit. on p. 69).

[CSS98] Keith D Cooper, Philip J Schielke, and Devika Subramanian, « An experimental evaluation of list scheduling », *in*: *TR98* 326 (1998) (cit. on p. 44).

[DB11] RI Davis and A Burns, « A survey of hard real-time scheduling algorithms for multiprocessor systems », *in*: *in ACM Computing Surveys* (2011) (cit. on pp. 29, 35, 40).

[DFG+14] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch, « Predictable flight management system implementation on a multicore processor », *in*: *Embedded Real Time Software (ERTS'14)*, 2014 (cit. on pp. 25, 38, 85, 89).

[DHC+13] Boubacar Diouf, Can Hantacs, Albert Cohen, Özcan Özturk, and Jens Palsberg, « A decoupled local memory allocator », *in*: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.*4* (2013), p. 34 (cit. on p. 88).

[DHP+14] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl, « A method cache for Patmos », *in*: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, IEEE, 2014, pp. 100–108 (cit. on p. 22).

[DL78] Sudarshan K Dhall and Chung Laung Liu, « On a real-time scheduling problem », *in*: *Operations research* 26.*1* (1978), pp. 127–140 (cit. on p. 31).

[DLM13] Huping Ding, Yun Liang, and Tulika Mitra, « Shared cache aware task mapping for WCRT minimization », *in*: *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, IEEE, 2013, pp. 735–740 (cit. on p. 33).

[DM89] Michael L. Dertouzos and Aloysius K. Mok, « Multiprocessor online scheduling of hard-real-time tasks », *in*: *IEEE Transactions on software engineering* 15.*12* (1989), pp. 1497–1506 (cit. on pp. 29, 30).

[DN12] Dakshina Dasari and Vincent Nelis, « An Analysis of the Impact of Bus Contention on the WCET in Multicores », *in*: *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, IEEE, 2012, pp. 1450–1457 (cit. on p. 65).

[DNA16] Dakshina Dasari, Vincent Nelis, and Benny Akesson, « A framework for memory contention analysis in multi-core platforms », *in*: *Real-Time Systems* 52.*3* (2016), pp. 272–322 (cit. on p. 65).

[DP07] Jean-Francois Deverge and Isabelle Puaut, « WCET-directed dynamic scratch-pad memory allocation of data », *in*: *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, IEEE, 2007, pp. 179–190 (cit. on pp. 69, 74, 88).

[DRW98] Robert P Dick, David L Rhodes, and Wayne Wolf, « TGFF: task graphs for free », *in*: *Proceedings of the 6th international workshop on Hardware/software codesign*, IEEE Computer Society, 1998, pp. 97–101 (cit. on pp. 46, 51, 62, 83).

[DVP+14] Benoît Dupont de Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager, « Time-critical computing on a single-chip massively parallel processor », *in*: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, IEEE, 2014, pp. 1–6 (cit. on pp. 9, 22, 23, 24, 35, 36, 50, 65, 70, 85, 86, 90, 92).

[Ens77] Philip Enslow Jr, « Multiprocessor organization—A survey », *in*: *ACM Computing Surveys (CSUR)* 9.*1* (1977), pp. 103–129 (cit. on pp. 26, 27, 36, 70).

[EY17] Pontus Ekberg and Wang Yi, « Fixed-priority schedulability of sporadic tasks on uniprocessors is np-hard », *in*: *Real-Time Systems Symposium (RTSS), 2017 IEEE*, IEEE, 2017, pp. 139–146 (cit. on pp. 14, 29).

[FAQ+14] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J Cazorla, « Contention in multicore hardware shared resources: Understanding of the state of the art », *in*: *OASIcs-OpenAccess Series in Informatics*, vol. 39, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014 (cit. on p. 32).

[FDC+13] Antoine Floc, Steven Derrien, Francois Charot, Christophe Wolinski, Olivier Sentieys, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, et al., « GeCoS: A framework for prototyping custom hardware design flows », *in*: *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2013, pp. 100–105 (cit. on p. 20).

[Gai02] Jiri Gaisler, « A portable and fault-tolerant microprocessor based on the SPARC v8 architecture », *in*: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, IEEE, 2002, pp. 409–415 (cit. on p. 12).

[GDL+03] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca, « A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform », *in*: *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, IEEE, 2003, pp. 189–198 (cit. on p. 33).

[GGC16] Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean, « Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms », *in*: *Real-time systems* 52.*6* (2016), pp. 808–832 (cit. on p. 30).

[GHK+13] Robert de Groote, Philip KF Hölzenspies, Jan Kuper, and Hajo Broersma, « Back to basics: Homogeneous representations of multi-rate synchronous dataflow graphs », *in*: *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, IEEE, 2013, pp. 35–46 (cit. on p. 19).

[GKC+15] Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert De Simone, « On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling », *in*: *Formal Modeling and Analysis of Timed Systems*, Springer, 2015, pp. 108–123 (cit. on p. 30).

[GLD01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale, « Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip », *in*: *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, IEEE, 2001, pp. 73–83 (cit. on p. 33).

[GNS09] Igor Griva, Stephen G Nash, and Ariela Sofer, *Linear and nonlinear optimization*, vol. 108, Siam, 2009, ISBN: 0898716616 (cit. on pp. 41, 77).

[GP94] Milind Girkar and Constantine D Polychronopoulos, « The hierarchical task graph as a universal intermediate representation », *in*: *International Journal of Parallel Programming* 22.*5* (1994), pp. 519–551 (cit. on p. 20).

[GPP09] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic, « Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture », *in*: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 1–8 (cit. on pp. 27, 88).

[Gra66] Ronald L Graham, « Bounds for certain multiprocessing anomalies », *in*: *Bell System Technical Journal* 45.*9* (1966), pp. 1563–1581 (cit. on p. 44).

[GSH+16] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin, « Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources », *in*: *Real-Time Systems* 52.*4* (2016), pp. 399–449 (cit. on p. 66).

[GTA06] Michael I Gordon, William Thies, and Saman Amarasinghe, « Exploiting coarse-grained task, data, and pipeline parallelism in stream programs », *in*: *ACM SIGOPS Operating Systems Review*, vol. 40, 5, ACM, 2006, pp. 151–162 (cit. on p. 24).

[GTK+02] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al., « A stream compiler for communication-exposed architectures », *in*: *ACM SIGPLAN Notices*, vol. 37, 10, ACM, 2002, pp. 291–303 (cit. on p. 89).

[HMC+16] Rihani Hamza, Moy Matthieu, Maiza Claire, Davis Robert I., and Altmeyer Sebastian, « Response Time Analysis of Synchronous Data Flow Programs on a Many-core Processor », *in*: *In proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS 2016)*, ACM, 2016 (cit. on pp. 7, 23, 65).

[HPP09] Damien Hardy, Thomas Piquet, and Isabelle Puaut, « Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches », *in*: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, IEEE, 2009, pp. 68–77 (cit. on pp. 32, 33).

[HPS10] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl, « WCET driven design space exploration of an object cache », *in*: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM, 2010, pp. 26–35 (cit. on p. 22).

[HRP17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut, « The Heptane Static Worst-Case Execution Time Estimation Tool », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, vol. 8, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017, pp. 1–812 (cit. on p. 112).

[HRW15] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm, « Towards compositionality in execution time analysis: definition and challenges », *in*: *ACM SIGBED Review* 12.*1* (2015), pp. 28–36 (cit. on p. 22).

[JGL+17] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi, « Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors », *in*: *arXiv preprint arXiv:1705.03245* (2017) (cit. on p. 32).

[JP86] Mathai Joseph and Paritosh Pandya, « Finding response times in a real-time system », *in*: *The Computer Journal* 29.*5* (1986), pp. 390–395 (cit. on p. 32).

[JWA15] Catherine E Jarrett, Bryan C Ward, and James H Anderson, « A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems », *in*: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ACM, 2015, pp. 3–12 (cit. on p. 33).

[KBC+14] Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastaval, « WCET-aware dynamic code management on scratchpads for software-managed multicores », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, IEEE, 2014, pp. 179–188 (cit. on pp. 37, 74, 77, 88).

[KDA+14] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar, « Bounding memory interference delay in COTS-based multi-core systems », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, IEEE, 2014, pp. 145–154 (cit. on p. 66).

[Kel15] Timon Kelter, « WCET analysis and optimization for multi-core real-time systems », PhD thesis, 2015 (cit. on p. 28).

[KHM+13] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk, « Evaluation of resource arbitration methods for multi-core real-time systems », *in*: *OASIcs-OpenAccess Series in Informatics*, vol. 30, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013 (cit. on pp. 38, 62, 83).

[KL91] Alexander C Klaiber and Henry M Levy, « An architecture for software-controlled data prefetching », *in*: *ACM SIGARCH Computer Architecture News*, vol. 19, 3, ACM, 1991, pp. 43–53 (cit. on p. 88).

[KM08] Manjunath Kudlur and Scott Mahlke, « Orchestrating the execution of stream programs on multicore platforms », *in*: *ACM SIGPLAN Notices*, vol. 43, 6, ACM, 2008, pp. 114–124 (cit. on pp. 30, 69, 89).

[KS14] Evangelia Kasapaki and Jens Sparsø, « Argo: A time-elastic time-division-multiplexed NOC using asynchronous routers », *in*: *Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on*, IEEE, 2014, pp. 45–52 (cit. on pp. 22, 23).

[KS15] Evangelia Kasapaki and Jens Sparso, « The Argo NOC: Combining TDM and GALS », *in*: *Circuit Theory and Design (ECCTD), 2015 European Conference on*, IEEE, 2015, pp. 1–4 (cit. on p. 22).

[KSS+16] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christoph Müller, Kees Goossens, and Jens Sparsø, « Argo: A real-time network-on-chip architecture with an efficient GALS implementation », *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.*2* (2016), pp. 479–492 (cit. on p. 22).

[KST11] Md Kamruzzaman, Steven Swanson, and Dean M Tullsen, « Inter-core prefetching for multicore processors using migrating helper threads », *in*: *ACM SIGPLAN Notices* 46.*3* (2011), pp. 393–404 (cit. on p. 88).

[Kur01] Tadahiro Kuroda, « CMOS design challenges to power wall », *in*: *Microprocesses and Nanotechnology Conference, 2001 International*, IEEE, 2001, pp. 6–7 (cit. on pp. 7, 11).

[LCA+14] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah, « Analysis of federated and global scheduling for parallel real-time tasks », *in*: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, IEEE, 2014, pp. 85–96 (cit. on p. 32).

[LL73] Chung Laung Liu and James W Layland, « Scheduling algorithms for multiprogramming in a hard-real-time environment », *in*: *Journal of the ACM (JACM)* 20.*1* (1973), pp. 46–61 (cit. on pp. 18, 29, 30).

[LM87] Edward Ashford Lee and David G Messerschmitt, « Static scheduling of synchronous data flow programs for digital signal processing », *in*: *IEEE Transactions on computers* 100.*1* (1987), pp. 24–35 (cit. on pp. 19, 24).

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou, « Traceability of flow information: Reconciling compiler optimizations and WCET estimation », *in*: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ACM, 2014, p. 97 (cit. on p. 14).

[LS99] Thomas Lundqvist and Per Stenstrom, « Timing anomalies in dynamically scheduled microprocessors », *in*: *Real-time systems symposium, 1999. Proceedings. The 20th IEEE*, IEEE, 1999, pp. 12–21 (cit. on p. 22).

[LW82] Joseph Y-T Leung and Jennifer Whitehead, « On the complexity of fixed-priority scheduling of periodic, real-time tasks », *in*: *Performance evaluation* 2.*4* (1982), pp. 237–250 (cit. on p. 29).

[LXK10] Lian Li, Jingling Xue, and Jens Knoop, « Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 10.*2* (2010), p. 28 (cit. on p. 88).

[MB12] Andrea Marongiu and Luca Benini, « An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs », *in*: *Computers, IEEE Transactions on* 61.*2* (2012), pp. 222–236 (cit. on pp. 22, 24, 35).

[MBB+15] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo, « Memory-processor co-scheduling in fixed priority systems », *in*: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ACM, 2015, pp. 87–96 (cit. on p. 25).

[MC96] Aloysius K Mok and Deji Chen, « A multiframe model for real-time tasks », *in*: *Real-Time Systems Symposium, 1996., 17th IEEE*, IEEE, 1996, pp. 22–29 (cit. on p. 18).

[MDC14] Renato Mancuso, Roman Dudko, and Marco Caccamo, « Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems », *in*: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, IEEE, 2014, pp. 1–10 (cit. on p. 25).

[MHN+17] Theodoros Marinakis, Alexandros-Herodotos Haritatos, Konstantinos Nikas, Georgios Goumas, and Iraklis Anagnostopoulos, « An efficient and fair scheduling policy for multiprocessor platforms », *in*: *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 1–4 (cit. on p. 31).

[MHP17] Sébastien Martinez, Damien Hardy, and Isabelle Puaut, « Quantifying WCET reduction of parallel applications by introducing slack time to limit resource contention », *in*: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ACM, 2017, pp. 188–197 (cit. on p. 66).

[Mic16] Pierre Michaud, « Best-offset hardware prefetching », *in*: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 469–480 (cit. on p. 88).

[MNP+16] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez, « A closer look into the aer model », *in*: *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, IEEE, 2016, pp. 1–8 (cit. on pp. 8, 26, 69, 91).

[Mok83] Aloysius K Mok, « Fundamental design problems of distributed systems for the hard-real-time environment », *in*: (1983) (cit. on p. 18).

[ND10] John Nickolls and William J Dally, « The GPU computing era », *in*: *IEEE micro* 30.*2* (2010) (cit. on p. 24).

[Neu82] John von Neumann, « First draft of a report on the EDVAC », *in*: *The Origins of Digital Computers*, Springer, 1982, pp. 383–392 (cit. on p. 24).

[NHP17] Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut, « Cache-conscious offline real-time task scheduling for multi-core processors », *in*: *29th Euromicro Conference on Real-Time Systems (ECRTS17)*, 2017 (cit. on pp. 31, 33).

[NSE09] Mircea Negrean, Simon Schliecker, and Rolf Ernst, « Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources », *in*: *Proceedings of the Conference on Design, Automation and Test in Europe*, European Design and Automation Association, 2009, pp. 524–529 (cit. on p. 32).

[ORS13] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat, « Automatic WCET Analysis of Real-Time Parallel Applications. », *in*: *WCET*, 2013, pp. 11–20 (cit. on p. 29).

[PBB+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley, « A predictable execution model for COTS-based embedded systems », *in*: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, 2011, pp. 269–279 (cit. on pp. 17, 25, 64).

[PDC+18] Isabelle Puaut, Mickael Dardaillon, Christoph Cullmann, Gernot Gebhard, and Steven Derrien, « Fine-Grain Iterative Compilation for WCET Estimation », *in*: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, vol. 58, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018 (cit. on p. 13).

[Per17] Quentin Perret, « Predictable execution on many-core processors », PhD thesis, University of Toulouse, 2017 (cit. on pp. 11, 19, 23, 30).

[PFB+11] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens, « Multi-task implementation of multi-periodic synchronous programs », *in*: *Discrete event dynamic systems* 21.*3* (2011), pp. 307–338 (cit. on p. 20).

[PK06] Ruxandra Pop and Shashi Kumar, « On Performance Improvement of Concurrent Applications Using Simultaneous Multithreaded Processors as NoC Resources », *in*: *Norchip Conference, 2006. 24th*, IEEE, 2006, pp. 191–196 (cit. on p. 30).

[PNP15] Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti, « Off-line mapping of multi-rate dependent task sets to many-core platforms », *in*: *Real-Time Systems* 51.*5* (2015), pp. 526–565 (cit. on pp. 27, 28, 30, 31).

[PP07] Isabelle Puaut and Christophe Pais, « Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison », *in*: *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, IEEE, 2007, pp. 1–6 (cit. on pp. 17, 22, 35).

[PP13] Dumitru Potop-Butucaru and Isabelle Puaut, « Integrated Worst-Case Execution Time Estimation of Multicore Applications. », *in*: *WCET*, 2013, pp. 21–31 (cit. on p. 28).

[Pus03] Peter Puschner, « The single-path approach towards WCET-analysable software », *in*: *Industrial Technology, 2003 IEEE International Conference on*, vol. 2, IEEE, 2003, pp. 699–704 (cit. on p. 24).

[Raj12] Ragunathan Rajkumar, *Synchronization in real-time systems: a priority inheritance approach*, vol. 151, Springer Science & Business Media, 2012 (cit. on p. 33).

[RDP16] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut, « Resource-aware task graph scheduling using ILP on multi-core », *in*: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, 2016 (cit. on p. 112).

[RDP17] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut, « Tightening contention delays while scheduling parallel applications on multi-core architecture », *in*: *Embedded Software (EMSOFT), 2017 International Conference on*, ACM, 2017 (cit. on pp. 28, 52, 85, 112).

[RGB+07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm, « Timing predictability of cache replacement policies », *in*: *Real-Time Systems* 37.*2* (2007), pp. 99–122 (cit. on pp. 12, 22).

[Rit93] Dennis M Ritchie, « The development of the C language », *in*: *ACM Sigplan Notices* 28.*3* (1993), pp. 201–208 (cit. on p. 24).

[RP17] Benjamin Rouxel and Isabelle Puaut, « STR2RTS: Refactored StreamIT benchmarks into statically analyzable parallel benchmarks for WCET estimation & real-time scheduling », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, vol. 57, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017 (cit. on pp. 46, 51, 62, 83, 87, 112).

[RSD+19] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut, « Hiding communication delays in contention-free execution for SPM-based multi-core architectures », *in*: *submitted to RTAS*, 2019 (cit. on pp. 70, 90, 112).

[SBH+15] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch, « Patmos reference handbook », *in*: *Technical University of Denmark, Tech. Rep* (2015) (cit. on pp. 22, 36, 70, 94).

[SBS+12] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki, « A statically scheduled time-division-multiplexed network-on-chip for real-time systems », *in*: *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, IEEE, 2012, pp. 152–160 (cit. on p. 22).

[Sch97] Robert R Schaller, « Moore's law: past, present and future », *in*: *IEEE spectrum* 34.*6* (1997), pp. 52–59 (cit. on pp. 7, 11).

[SCP+14] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparso, « A time-predictable memory network-on-chip », *in*: *14th International Workshop on Worst-Case Execution Time Analysis*, 2014, p. 53 (cit. on p. 22).

[SCT10] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele, « Timing analysis for TDMA arbitration in resource sharing systems », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, IEEE, 2010, pp. 215–224 (cit. on p. 27).

[SEL08] Insik Shin, Arvind Easwaran, and Insup Lee, « Hierarchical scheduling framework for virtual clustering of multiprocessors », *in*: *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on*, IEEE, 2008, pp. 181–190 (cit. on p. 32).

[SHP13] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch, « Data cache organization for accurate timing analysis », *in*: *Real-Time Systems* 49.*1* (2013), pp. 1–28 (cit. on p. 22).

[SM08] Vivy Suhendra and Tulika Mitra, « Exploring locking & partitioning for predictable shared caches on multi-cores », *in*: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, IEEE, 2008, pp. 300–303 (cit. on pp. 32, 33).

[SNE10] Simon Schliecker, Mircea Negrean, and Rolf Ernst, « Bounding the shared resource load for the performance analysis of multiprocessor systems », *in*: *Proceedings of the conference on design, automation and test in Europe*, European Design and Automation Association, 2010, pp. 759–764 (cit. on p. 65).

[SP17] Muhammad Refaat Soliman and Rodolfo Pellizzoni, « WCET-Driven dynamic data scratchpad management with compiler-directed prefetching », *in*: *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017 (cit. on p. 89).

[SPS+15] Rasmus Bo Sorensen, Wolfgang Puffitsch, Martin Schoeberl, and Jens Sparso, « Message Passing on a Time-predictable Multicore Processor », *in*: (2015) (cit. on p. 28).

[SR94] Kang G Shin and Parameswaran Ramanathan, « Real-time computing: A new discipline of computer science and engineering », *in*: *Proceedings of the IEEE* 82.*1* (1994), pp. 6–24 (cit. on p. 11).

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky, « Priority inheritance protocols: An approach to real-time synchronization », *in*: *IEEE Transactions on computers* 39.*9* (1990), pp. 1175–1185 (cit. on p. 33).

[SRP19] Martin Schoeberl, Benjamin Rouxel, and Isabelle Puaut, « A Time-predictable Branch Predictor », *in*: *Proceedings of the 34rd Annual ACM Symposium on Applied Computing*, ACM, 2019 (cit. on p. 112).

[SS16] Stefanos Skalistis and Alena Simalatsar, « Worst-case execution time analysis for many-core architectures with NoC », *in*: *International Conference on Formal Modeling and Analysis of Timed Systems*, Springer, 2016, pp. 211–227 (cit. on p. 23).

[SS17] Stefanos Skalistis and Alena Simalatsar, « Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees », *in*: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2017, pp. 752–757 (cit. on pp. 31, 89).

[SSP+11] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, and Tommy Thorn, « Towards a time-predictable dual-issue microprocessor: The Patmos approach », *in*: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, vol. 18, 2011, pp. 11–21 (cit. on pp. 22, 50).

[SSP+14] Rasmus Bo Sorensen, Jens Sparso, Mikkel Rath Pedersen, and Jaspur Hojgaard, « A metaheuristic scheduler for time division multiplexed networks-on-chip », *in*: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, IEEE, 2014, pp. 309–316 (cit. on p. 27).

[SY15] Martin Stigge and Wang Yi, « Graph-based models for real-time workload: a survey », *in*: *Real-Time Systems* 51.*5* (2015), pp. 602–636 (cit. on p. 18).

[Tay83] Richard N Taylor, « Complexity of analyzing the synchronization structure of concurrent programs », *in*: *Acta Informatica* 19.*1* (1983), pp. 57–84 (cit. on p. 54).

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe, « StreamIt: A language for streaming applications », *in*: *Compiler Construction*, Springer, 2002, pp. 179–196 (cit. on pp. 20, 62, 83).

[TMW+16] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo, « A real-time scratchpad-centric os for multi-core embedded systems », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, IEEE, 2016, pp. 1–11 (cit. on pp. 85, 89).

[TPG+14] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler, « Many-core scheduling of data parallel applications using SMT solvers », *in*: *Digital System Design (DSD), 2014 17th Euromicro Conference on*, IEEE, 2014, pp. 615–622 (cit. on pp. 19, 30, 39, 69, 89).

[TPM14] Pranav Tendulkar, Peter Poplavko, and Oded Maler, « Strictly periodic scheduling of acyclic synchronous dataflow graphs using SMT solvers », *in*: *Verimag Research Report TR-2014* (2014), p. 501 (cit. on p. 30).

[TTT10] Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada, « Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems », *in*: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, IEEE, 2010, pp. 1124–1129 (cit. on p. 88).

[Tur37] Alan M Turing, « On computable numbers, with an application to the Entscheidungsproblem », *in*: *Proceedings of the London mathematical society* 2.*1* (1937), pp. 230–265 (cit. on p. 24).

[War62] Stephen Warshall, « A theorem on boolean matrices », *in*: *Journal of the ACM (JACM)* 9.*1* (1962), pp. 11–12 (cit. on p. 54).

[WCM16] Wenhao Wang, Fabrice Camut, and Benoit Miramond, « Generation of schedule tables on multi-core systems for AUTOSAR applications », *in*: *Design and Architectures for Signal and Image Processing (DASIP), 2016 Conference on*, IEEE, 2016, pp. 191–198 (cit. on pp. 22, 23).

[WDA+12] Jack Whitham, Robert I Davis, Neil C Audsley, Sebastian Altmeyer, and Claire Maiza, « Investigation of scratchpad memory for preemptive multitasking », *in*: *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, Ieee, 2012, pp. 3–13 (cit. on p. 33).

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al., « The worst-case execution-time problem—overview of methods and survey of tools », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 7.*3* (2008), p. 36 (cit. on pp. 14, 28).

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand, « Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.*7* (2009), p. 966 (cit. on p. 22).

[WGW+14] Andreas Weichslgartner, Deepak Gangadharan, Stefan Wildermann, Michael Glaß, and Jurgen Teich, « DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems », *in*: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, IEEE, 2014, pp. 1–10 (cit. on p. 30).

[WP13] Saud Wasly and Rodolfo Pellizzoni, « A dynamic scratchpad memory unit for predictable real-time embedded systems », *in*: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, IEEE, 2013, pp. 183–192 (cit. on p. 88).

[WP14] Saud Wasly and Rodolfo Pellizzoni, « Hiding memory latency using fixed priority scheduling », *in*: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, IEEE, 2014, pp. 75–86 (cit. on p. 89).

[WS11] Jack Whitham and Martin Schoeberl, *The limits of TDMA based memory access scheduling*, tech. rep., Technical Report YCS-2011-470, University of York, 2011 (cit. on p. 27).

[YHZ+09] Ying Yi, Wei Han, Xin Zhao, Ahmet T Erdogan, and Tughrul Arslan, « An ilp formulation for task mapping and scheduling on multi-core architectures », *in*: *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.* IEEE, 2009, pp. 33–38 (cit. on p. 30).

[YPV15] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan, « Parallelism-aware memory interference delay analysis for cots multicore systems », *in*: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, IEEE, 2015, pp. 184–195 (cit. on p. 66).

[Zak13] George F Zaki, « Scalable techniques for scheduling and mapping DSP applications onto embedded multicore platforms », PhD thesis, University of Maryland, College Park, 2013 (cit. on pp. 19, 30).

[ZLL+18] Quan Zhou, Guohui Li, Jianjun Li, and Chenggang Deng, « Execution-Efficient Response Time Analysis on Global Multiprocessor Platforms », *in*: *IEEE Transactions on Parallel and Distributed Systems* (2018) (cit. on p. 31).

LIST OF FIGURES

1 Real-time is not real-fast . . . 12
2 Illustrating example showing the importance of the execution order . . . 13
3 Worst-Case Execution Times (WCET) . . . 14

1.1 Example of an application represented by a DAG . . . 19
1.2 An SDF example and its transformation to HSDF and one possible PEG . . . 20
1.3 Expanded example of a fork-Join graph . . . 21
1.4 Example of a CSDF . . . 21
1.5 Patmos core (left side) and its Argo NoC (right side), from [KS14] . . . 23
1.6 Kalray core (left side) and its NoC (right side), from [DVP+14] . . . 24
1.7 Task representation with different execution model . . . 25
1.8 Representation of the Round-Robin policy principle. Access order is $R.1$, $R.2$, $R.3$ 26
1.9 Example of a TDMA arbitration scheme. Blue filled boxes are TDM slots, granting time for the corresponding core where horizontal-lined boxes represent memory accesses as in PREM in Figure 1.7b . . . 27

2.1 A multi-core architecture abstraction . . . 36
2.2 Delay representation. Configuration: $T_{slot} = 2$ time units, $D_{slot} = 1$ data-word, 3 cores Request: 5 data words gives 3 chunks. Each chunk is delayed by 2 interferences $\times\, T_{slot}$ . . . 37
2.3 Task graph example . . . 38
2.4 Resulting schedule for task-graph from Figure 2.3 with a worst-case contention policy targeting a tri-core architecture equivalent to Figure 2.1. Overall makespan is 122 time units. . . . 40
2.5 Scalability of ILP formulation (synthetic task graphs / STG) . . . 47
2.6 Distribution of the degradation of the heuristic against the ILP formulation using STG task set. (logarithmic scale) . . . 48
2.7 Average makespan when varying T_{slot} (synthetic task graphs / BTG) . . . 49

3.1 Running task graph example, identical to Figure 2.3 . . . 52
3.2 Motivation example . . . 53
3.3 Example of adjustments that occur while scheduling. (3.3a) initial schedule of 2 tasks. (3.3b) adjusted communication delays after the addition of task G . . . 61
3.4 Distribution of the degradation of the heuristic against the ILP formulation using STG task set. . . . 63
3.5 Gain in % obtained with contention-aware scenario (heuristic, STR2RTS benchmarks) . . . 64

4.1 Hardware abstraction . . . 71
4.2 Running task graph example, identical to Figure 2.3 . . . 72
4.3 Motivating example . . . 73

4.4 Partial schedule of the task graph from Figure 4.2 with a DFS sorting strategy in the heuristic algorithm. 80
4.5 Distribution degradation heuristic vs ILP (232 test-cases) 84
4.6 Gain of *non-blocking* communications over *blocking* on STR2RTS benchmarks per cores/SPM configuration – avg: 4% 85
4.7 Gain of *non-blocking* communications over *blocking* on TGFF benchmarks – average: 8% 86
4.8 Average gain of *non-blocking* schedule length over *blocking* one depending on fragmentation strategy 87

LIST OF ALGORITHMS

2.3.1 Forward list scheduling 45
2.3.2 Build list scheduled element 45
2.3.3 Schedule an element 46

3.3.1 Forward list scheduling 59
3.3.2 Adjust schedule in contention-free mode : Updating the schedule to avoid contention 60
3.3.3 Adjust schedule in contention-aware mode : Updating the schedule to cope with interference 61

4.5.1 Forward list scheduling 81
4.5.2 Build list scheduled element 81
4.5.3 Schedule an element 82
4.5.4 Allocate a SPM region to a phase 82

LIST OF PUBLICATIONS

— Benjamin Rouxel et al., « Hiding communication delays in contention-free execution for SPM-based multi-core architectures », *in*: *submitted to RTAS*, 2019

— Martin Schoeberl, Benjamin Rouxel, and Isabelle Puaut, « A Time-predictable Branch Predictor », *in*: *Proceedings of the 34rd Annual ACM Symposium on Applied Computing*, ACM, 2019

— Benjamin Rouxel, Steven Derrien, and Isabelle Puaut, « Tightening contention delays while scheduling parallel applications on multi-core architecture », *in*: *Embedded Software (EMSOFT), 2017 International Conference on*, ACM, 2017

— Benjamin Rouxel and Isabelle Puaut, « STR2RTS: Refactored StreamIT benchmarks into statically analyzable parallel benchmarks for WCET estimation & real-time scheduling », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, vol. 57, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017

— Damien Hardy, Benjamin Rouxel, and Isabelle Puaut, « The Heptane Static Worst-Case Execution Time Estimation Tool », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, vol. 8, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017, pp. 1–812

— Benjamin Rouxel, Steven Derrien, and Isabelle Puaut, « Resource-aware task graph scheduling using ILP on multi-core », *in*: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, 2016

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre: Minimiser l'impact des communications lors de l'ordonnancement d'application temps-réels sur des architectures multi-cœurs

Mot clés : systèmes temps-réel, multi-cœurs, ordonnancement, contention

Les architectures multi-coeurs utilisant des mémoire bloc-notes sont des architectures attrayantes pour l'exécution des applications embarquées temps-réel, car elles offrent une grande capacité de calcul. Cependant, les systèmes temps-réel nécessitent de satisfaire des contraintes temporelles, ce qui peut être compliqué sur ce type d'architectures à cause notamment des ressources matérielles physiquement partagées entre les coeurs. Plus précisément, les scénarios de pire cas de partage du bus de communication entre les coeurs et la mémoire externe sont trop pessimistes.

Cette thèse propose des stratégies pour réduire ce pessimisme lors de l'ordonnancement d'applications sur des architectures multi-coeurs. Tout d'abord, la précision du pire cas des coûts de communication est accrue grâce aux informations disponibles sur l'application et l'état de l'ordonnancement en cours. Ensuite, les capacités de parallélisation du matériel sont exploitées afin de superposer les calculs et les communications. De plus, les possibilités de superposition sont accrues par le morcellement de ces communications.

Title: Minimising shared resource contention when scheduling real-time applications on multi-core architectures

Keywords : real-time system, multi-cores, scheduling, contention

Abstract : Multi-core architectures using scratch pad memories are very attractive to execute embedded time-critical applications, because they offer a large computational power. However, ensuring that timing constraints are met on such platforms is challenging, because some hardware resources are shared between cores. When targeting the bus connecting cores and external memory, worst-case sharing scenarios are too pessimistic.

This thesis propose strategies to reduce this pessimism. These strategies offer to both improve the accuracy of worst-case communication costs, and to exploit hardware parallel capacities by overlapping computations and communications. Moreover, fragmenting the latter allow to increase overlapping possibilities.