Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture

Nicolas Ventroux

N^o d'ordre : 3364

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITÉ DE RENNES 1*

Mention : ÉLECTRONIQUE

PAR

Nicolas VENTROUX

Équipe d'accueil : CEA LIST / IRISA

École Doctorale : MATISSE
Composante universitaire : IFSIC

Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture

Soutenue le 19 septembre 2006 devant la commission d'examen

MM.	A.A. Jerraya	Directeur de Recherche CNRS au TIMA	Rapporteurs
	M. Auguin	Directeur de Recherche CNRS à l'I3S	
	O. Sentieys	Professeur à l'ENSSAT	Examinateurs
	J.Y. Mignolet	Ingénieur Chercheur à l'IMEC	
	E. Flamand	Ingénieur chez ST/AST	
	D. Lavenier	Directeur de Recherche CNRS à l'IRISA	Directeur de thèse
	T. Collette	Ingénieur Chercheur au CEA LIST	Codirecteur de thèse

Je dédie cette thèse à mes parents,
et à ma grande sœur Magali,

Remerciements

Ce travail de thèse a été réalisé au sein du Laboratoire Calculateurs Embarqués et Image (LCEI) du Laboratoire d'Intégration des Systèmes et des Technologies (LIST) au Commissariat à l'Energie Atomique (CEA) de Saclay. Aussi, je tiens à remercier Laurent Letellier pour m'avoir accueilli dans son laboratoire et permis de travailler dans les meilleures conditions jusqu'à la fin de la rédaction de ce mémoire.

J'adresse mes remerciements à monsieur Thierry Collette, chef du service Architectures et Conception du CEA LIST, pour m'avoir encadré et prodigué de nombreux conseils tout au long de ces trois années. Je tiens à lui exprimer toute ma gratitude pour la confiance qu'il a su m'accorder et pour m'avoir offert la possibilité de m'épanouir professionnellement.

Je remercie monsieur Dominique Lavenier, directeur de recherche CNRS à l'IRISA, qui m'a fait l'honneur de diriger cette thèse. Malgré la distance qui nous a séparé, il a toujours été d'un grand soutien tout au long de la thèse et plus particulièrement lors de la rédaction de ce manuscrit.

Je tiens à exprimer ma profonde reconnaissance à monsieur Raphaël David, ingénieur chercheur au CEA LIST, pour tout le temps qu'il a consacré à mon encadrement et pour les nombreuses discussions que l'on a eues ensemble. Ce travail de thèse n'aurait jamais pu être possible sans son soutien exemplaire et j'aimerais partager avec lui les résultats obtenus pendant cette thèse.

Je tiens à remercier monsieur Olivier Sentieys, professeur à l'ENSSAT, pour m'avoir fait l'honneur de présider le jury de cette thèse.

J'adresse mes remerciements à messieurs Ahmed Amine Jerraya, directeur de recherche CNRS au TIMA, et Michel Auguin, directeur de recherche CNRS à l'I3S, pour avoir accepté de juger mon travail en tant que rapporteurs.

Je remercie par ailleurs messieurs Eric Flamand, directeur des activités Computing chez ST/AST, et Jean-Yves Mignolet, ingénieur chercheur à l'IMEC, pour leur participation dans mon jury de thèse.

Je tiens également à remercier monsieur Vincent David, ingénieur chercheur au CEA LIST, pour tous ses conseils et ses remarques toujours très pertinentes. Il a toujours su se montrer disponible et ses connaissances approfondies dans les systèmes temps-réel m'ont été d'un très grand secours tout au long de cette thèse.

Je remercie monsieur Jean-Loup Leroy, ingénieur ESIEE, qui a travaillé à mes côtés pendant son stage de fin d'études sur la validation de mes travaux de thèse. Je remercie par ailleurs, madame Eva Dokladalova, ingénieur chercheur au CEA LIST, pour son travail important sur l'application MPEG4-AVC.

J'aimerais remercier tous mes collègues et amis du LCEI avec qui il a été très agréable de travailler. Merci pour tous les bons moments passés ensemble et l'aide que vous avez su m'apporter pendant toute la durée de cette thèse. Plus particulièrement, je tiens à remercier Stéphane Chevobbe pour avoir toujours été présent dans les moments difficiles et pour m'avoir fait l'honneur d'être son ami.

Je souhaite remercier très chaleureusement mes parents qui m'ont soutenu et offert des conditions exceptionnelles pour étudier. Je leur exprime ma profonde reconnaissance. Je tiens

également à remercier ma grande sœur Magali et son mari Gaëtan pour avoir été présents ces dernières années de thèse. C'est une immense joie de vous savoir de nouveau près de moi.

Que mes amis soient remerciés pour leur soutien et tous les bons moments que l'on a passés ensemble. Merci à Guillaume, Claudia, Cyril, Marie, Magali, David, Aurélie, Jeff, Xavier et tous les autres qui se reconnaîtront. Merci à Pierre et Marie-Claire Cleenwerck qui ont toujours fait preuve d'une très grande générosité et sur qui l'on peut toujours compter.

Merci finalement à tous les membres de l'association MAIOT avec qui j'ai partagé de très bons moments. J'espère que nous avons réussi à créer un lieu de rencontre, de détente et d'entraide pour tous les doctorants du CEA.

Table des matières

Table des figures

Liste des tableaux

Introduction

Sommaire

Les systèmes embarqués sont aujourd'hui omniprésents dans les sociétés modernes. Pourtant, nous ignorons bien souvent la présence des dispositifs électroniques composant tous ces objets qui nous entourent. Leur intelligence les rend par la même occasion de plus en plus indispensables. Ces dispositifs influent sur l'évolution de notre société. Ils transforment la nature de nos métiers, nous permettent de nous déplacer, de communiquer et d'avoir accès à des informations auxquelles nous n'aurions jamais pu accéder autrement.

Ces systèmes embarqués offrent aujourd'hui de nombreuses fonctionnalités. Les assistants personnels numériques (PDA : Personal Digital Assistant) ne servent plus seulement d'agenda. Ils permettent de téléphoner, d'écouter de la musique, de visionner des films, voire même le positionnement avec le GPS (Global Positioning System). Ces dispositifs mobiles apporteront de plus en plus de services comme la synthèse de la parole, la télévision numérique, la vidéoconférence, ou les loisirs numériques. Par ailleurs, ils se connecteront via de multiples standards de communication (UMTS : Universal Mobile Telecommunications System, WiFi : Wireless Fidelity, WiMax : Worldwide Interoperability for Microwave Access, Bluetooth...) et s'adapteront dynamiquement au réseau présent et aux conditions d'utilisation lors de chacun de nos déplacements. Ils deviendront ainsi de véritables objets communiquants.

Les architectes de ces dispositifs devront s'efforcer de proposer des solutions toujours plus performantes et suffisamment dynamiques pour que leurs structures puissent s'adapter aux besoins de chacun. L'utilisateur pourra, par exemple, envoyer des courriels ou visionner un film ou des diapositives, tout en profitant de la vidéoconférence avec plusieurs autres personnes. Les coûts et la consommation d'énergie de ces architectures devront dans le même temps être maîtrisés afin de respecter les contraintes de mobilité des systèmes embarqués. Mais dans quelles mesures les solutions actuelles ne peuvent-elles pas répondre à ces exigences ?

Le contexte de l'étude

Le premier microprocesseur commercial Intel 4004 a été conçu par Marcian Hoff au début des années 70. Depuis, même si de nombreuses évolutions technologiques ont été proposées, les microprocesseurs reposent presque tous sur le même principe d'exécution, dit von Neumann. La complexité de ces dispositifs est passée de 2 250 transistors, pour l'Intel 4004, à des centaines de millions pour les microprocesseurs actuels. Pendant la même période, la puissance de traitement a augmenté de 60 000 instructions exécutées par seconde, à plusieurs milliards pour les processeurs les plus performants. Ceci a permis de passer de la simple manipulation arithmétique, pour concevoir des calculatrices ou des montres, aux applications complexes que nous connaissons aujourd'hui.

Ce rythme d'évolution effréné provient des avancées technologiques des circuits intégrés et de leurs architectures. Dans ce manuscrit, nous nous intéresserons plus particulièrement aux solutions architecturales. Il est vrai que l'amélioration des procédés a grandement contribué à réduire les temps d'exécution par l'utilisation de transistors plus rapides et de plus faibles dimensions. Cependant, le temps de commutation et la taille des transistors atteindront bientôt leurs limites. A mesure que l'on s'approche de la taille de l'atome, des phénomènes parasites deviennent prépondérants et dégradent le fonctionnement des transistors. De plus, lorsque la taille du transistor diminue, il devient plus sensible aux effets du champ électrique et sa tension d'alimentation doit être réduite. Pour maintenir ses performances, sa tension de seuil doit alors être abaissée. Ceci génère des courants de fuite et contribue à augmenter la consommation d'énergie de l'architecture.

Les progrès d'intégration des transistors ont également enrichi les architectures de nouvelles fonctionnalités afin d'accroître encore la fréquence d'exécution des instructions. En fait, ces apports architecturaux ont principalement contribué à réduire le temps passé à exécuter une instruction, ou à augmenter le nombre d'instructions exécutées en parallèle. Pour favoriser la production rapide de résultats, le traitement d'une instruction a été segmenté (pipeline) et l'alimentation en instructions et en données a été accélérée par la hiérarchisation de la mémoire. Des mécanismes de prédiction de branchement complexes ont permis dans le même temps de soutenir la fréquence d'exécution.

D'autre part, pour exécuter plusieurs instructions simultanément, les ressources de calcul élémentaires ont été parfois multipliées. Les architectures superscalaires puis VLIW (Very Large Instruction Word) ont ainsi vu le jour, ainsi que de complexes mécanismes d'allocation et de spéculation. Néanmoins, le parallélisme d'instructions est restreint dans chaque application et séquence d'instructions. Par conséquent, exécuter davantage d'instructions en parallèle nécessite toujours plus de complexité. En fait, l'augmentation du nombre ou de la taille des dispositifs matériels ou logiciels nécessaires aux améliorations de ces architectures, contribue à la dégradation de leur efficacité (voir figure 1). Par exemple, la part des transistors utilisés pour effectuer des calculs n'est plus que de l'ordre de 10% pour les dernières architectures superscalaires. Le reste est utilisé pour extraire le parallélisme d'instructions ou alimenter le pipeline d'exécution. La consommation d'énergie ou les coûts de développement nécessaires à leur mise en œuvre sont alors excessifs compte tenu des performances obtenues.

Ainsi, le fait de maintenir les efforts de recherche et de développement autour du parallélisme d'instructions, pendant toutes ces années, a engendré une utilisation inefficace des transistors. La mise en œuvre de ces architectures a atteint ses limites et induit des performances insuffisantes. Pire, les limitations des principaux facteurs d'amélioration de performance compromettent ces modèles d'architecture. Les mécanismes de spéculation nécessaires pour leur

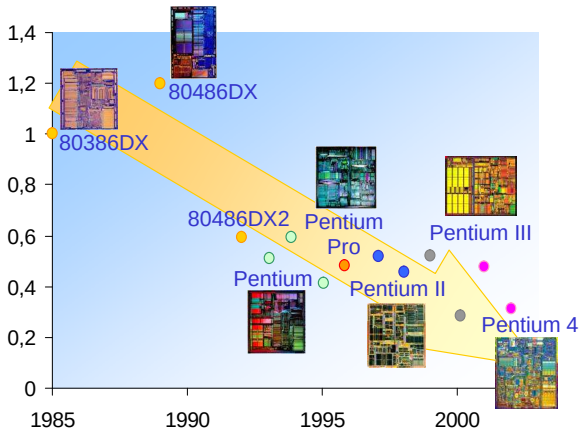

FIG. 1 – Efficacité des transistors des processeurs Intel 32 bits - nombre d'instructions exécutées en 1 cycle par transistor utilisé. Les temps de commutation des transistors pour chacune de ces architectures ont été adaptés pour une technologie CMOS 0,13μm

exploitation les rendent de moins en moins prévisibles et empêchent leur utilisation dans des environnements embarqués. Continuer à améliorer ce modèle d'exécution ne permettrait donc pas d'atteindre le niveau de performance et de flexibilité requis par les nouvelles applications embarquées.

Une solution pour continuer à améliorer les performances consiste à exploiter le parallélisme au niveau tâche et à multiplier le nombre de ressources de calcul intégré sur une même puce. Les architectures exploitant ce parallélisme, que nous appellerons *architectures parallèles* par la suite, offrent une grande flexibilité et de bonnes performances. De plus, elles réutilisent aisément des ensembles fonctionnels existants et hétérogènes. D'autre part, elles réduisent la consommation d'énergie et sont capables d'exécuter efficacement plusieurs applications en parallèle, voire des applications disposant d'un fort parallélisme au niveau tâche. Ainsi, l'exploitation supplémentaire de ce parallélisme, en plus des différentes avancées technologiques actuelles, semble être une alternative intéressante pour répondre aux futures exigences des applications. Néanmoins, la complexité de ces structures nécessite un contrôle adapté.

La problématique de l'étude

Le parallélisme au niveau tâche peut nécessiter un contrôle tout aussi complexe que celui mis en œuvre pour exploiter le parallélisme d'instructions. Ceci est d'autant plus vrai si l'ordre d'exécution des tâches doit être modifié dynamiquement. L'architecture parallèle doit alors maîtriser les communications internes, les échanges de données, les accès et le partage des ressources de mémorisation. Les différentes tâches doivent être allouées et exécutées dynamiquement sur ces ressources hétérogènes, tout en maîtrisant des contraintes de temps d'exécution (temps-réel) et de consommation d'énergie. De plus, certaines applications nécessitent de maîtriser avec exactitude les temps d'exécution et la prédictibilité du système. La complexité et le comportement des architectures de contrôle doivent donc pouvoir être évalués précisément. Enfin, ces dernières sont tenues de proposer les fonctionnalités nécessaires aux systèmes temps-réel et d'exécuter dynamiquement les applications.

La difficulté est que la solution de contrôle mise en œuvre a une influence inévitable, à la fois sur le modèle de programmation et sur le modèle d'exécution. Elle contribue finalement à la flexibilité de l'architecture tout en maîtrisant l'ensemble de la complexité du système. Dans le même temps, cette dernière ne doit ni dégrader sa programmabilité, ni pénaliser ses performances.

Pour exploiter le parallélisme d'exécution au niveau tâche de ces architectures, il est d'usage d'utiliser un microprocesseur pour supporter l'implémentation d'un système d'exploitation. Celui-ci prend en charge l'ordonnancement des tâches, les interfaces utilisateurs et les accès mémoires. Cependant, cette solution de contrôle comporte de nombreux défauts. La structure d'un microprocesseur optimisé pour le calcul ne peut pas constituer une solution pour le contrôle, tant les fonctionnalités visées sont différentes. Elle induit alors une consommation d'énergie et une taille sur le silicium importantes et inutiles. Enfin, son emploi nécessite l'utilisation de couches d'abstraction et de mécanismes d'interruption enclins à des problèmes de disponibilité et de réactivité. En effet, les signaux de contrôle échangés entre le système d'exploitation et l'architecture de calcul doivent traverser plusieurs couches logicielles. Ces temps de traversées varient lors de l'exécution et inhibent la prise en compte d'informations immédiates. Leur utilisation répond au seul souci de programmabilité des systèmes qu'il doit contrôler. La plupart des améliorations de performance envisagées pour les systèmes d'exploitation a ainsi bien souvent uniquement consisté à adapter leurs fonctionnalités aux besoins applicatifs. Par conséquent, le système d'exploitation est un élément peu optimisé. Il est considéré comme une contrainte pour l'architecte et une nécessité pour le développeur.

Même si pendant longtemps la programmabilité a été le seul critère de sélection, depuis quelques années des solutions matérielles pour accélérer les primitives en charge de la gestion des systèmes temps-réel ont été imaginées. Le fait de déporter une partie des services offerts normalement par le système d'exploitation améliore sensiblement le déterminisme et autorise une grande réactivité. Par ailleurs, elle induit un faible coût d'intégration, réduit la consommation d'énergie et s'affranchit des couches d'abstraction tout en améliorant considérablement les performances du système. Ainsi, le contrôle de systèmes multiprocesseurs pourrait être géré efficacement par un accélérateur matériel de système d'exploitation. Néanmoins, comme nous le verrons par la suite, les solutions matérielles existantes n'offrent pas assez de flexibilité et de fonctionnalités pour répondre aux futurs besoins des systèmes embarqués. Le but de cette thèse est donc de proposer *une architecture pour le contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués* appelée *Operating System accelerator on Chip* ou *OSoC*.

L'organisation du manuscrit

Le premier chapitre revient sur la notion de parallélisme. En détaillant les limitations du parallélisme d'instructions et de données, nous montrons pourquoi le parallélisme de tâches doit être exploité pour continuer à améliorer les performances des architectures. Ensuite, les différentes formes de parallélisme de tâches sont étudiées. Nous comprenons alors dans quelle mesure celles-ci ne répondent pas aux besoins des futurs systèmes embarqués. Un nouveau modèle d'exécution plus approprié est donc proposé en conclusion de ce chapitre.

Dans le second chapitre, les différentes solutions existantes pour contrôler l'architecture multiprocesseur envisagée sont présentées. Il montre qu'une solution logicielle ne pourrait pas être utilisée, car elle pénaliserait sensiblement les performances globales du système. Des approches alternatives sont alors étudiées. Elles consistent à utiliser des éléments matériels

pour accélérer des primitives de contrôle. Les gains obtenus sont conséquents et une approche multiprocesseur devient possible.

Dans le troisième chapitre, une nouvelle architecture multiprocesseur est proposée. Elle est appelée *Loosely-Coupled SCalable Multi-Processor* (*SCMP-LC*). Elle se compose d'une partie calcul, constituée de multiples ressources hétérogènes et d'un contrôleur matériel dénommé *OSoC*. Une analyse détaillée de sa structure et de son principe d'exécution est effectuée. C'est l'occasion de comprendre en quoi sa gestion particulière des communications et des ressources de mémorisation rend son contrôleur original.

Au sein du quatrième chapitre, l'architecture *OSoC* est définie. Ses mécanismes innovants d'ordonnancement, d'allocation, de synchronisation des tâches et de gestion des ressources de mémorisation sont présentés. L'architecture *OSoC* est la première architecture de contrôle temps-réel à proposer ce niveau de parallélisme et de flexibilité. Elle offre une gestion particulière des tâches afin de répondre au mieux aux exigences des systèmes embarqués.

La validation de l'*OSoC* dans l'architecture *SCMP-LC* fait l'objet du cinquième chapitre. Les modèles fonctionnels et les résultats de synthèse sont présentés. Nous montrons que l'architecture *OSoC* constitue une solution unique en terme de services offerts et de flexibilité. La comparaison avec le noyau temps-réel μCOS-II confirme un gain significatif sur le temps global d'exécution d'un encodeur MPEG-4 AVC. D'autres résultats d'implémentation mettent en évidence les gains en performance obtenus avec l'architecture *SCMP-LC* et plus particulièrement l'*OSoC*.

Finalement, dans le chapitre de conclusion, nous résumons les contributions de ce travail de recherche. Nous discutons des améliorations possibles des architectures *SCMP-LC* et *OSoC*. Ce dernier chapitre propose également des évolutions et des perspectives de recherche à plus long terme.

Chapitre 1

Les architectures parallèles pour les systèmes embarqués

Sommaire

Les applications sont de plus en plus exigentes en terme de performance et d'efficacité. Ceci est particulièrement vrai pour les systèmes embarqués qui apportent, en plus, de nouvelles contraintes technologiques. En effet, ces systèmes doivent répondre à de forts impératifs d'intégration et de consommation d'énergie. Pendant près de 40 années, les innovations technologiques se sont succédées dans le but de réduire les temps d'exécution. Certaines ont consisté à réduire le temps de traitement et à améliorer les fréquences de fonctionnement. Ces techniques sont néanmoins limitées par les possibilités physiques d'intégration. D'autres ont tenté d'augmenter le débit de traitement des instructions, c'est-à-dire le nombre de traitements effectués par unité de temps. Les architectures parallèles étaient nées. Elles consistent, en fait, à segmenter le traitement des instructions ou à accroître le nombre de ressources pour exécuter plus d'instructions simultanément.

Ce premier chapitre définit tout d'abord la notion de parallélisme. Ensuite, il apporte une nouvelle classification de ces architectures en intégrant les propositions qui sont apparues ces dernières années. C'est l'occasion de découvrir l'ensemble des procédés employés pour exploiter le parallélisme d'instructions et de tâches. Chacune de ces solutions répond à un besoin applicatif particulier et à des critères de consommation d'énergie, de performance ou de surface. L'examen de leurs caractéristiques permet de comprendre pourquoi elles apportent un gain de performance et en quoi elles sont limitées. Notre contexte d'étude particulier, avec de fortes contraintes d'intégration et de consommation d'énergie, nous oriente alors finalement vers une solution multiprocesseur hétérogène particulière. Les solutions couramment utilisées par les supercalculateurs et autres machines de calcul parallèle (machines vectorielles ou parallèles [1]) ne seront pas abordées dans ce chapitre, puisqu'elles ne peuvent pas être intégrées dans les systèmes embarqués. Nous nous intéresserons essentiellement aux solutions intégrées sur une même puce.

1.1 Définition du parallélisme et limitations

En architecture, le parallélisme est lié à la possibilité de pouvoir exécuter plusieurs traitements simultanément. On définit deux types de parallélisme : le parallélisme de données et le parallélisme de contrôle. Le parallélisme de données consiste à exécuter une même opération sur des unités de calcul et des données différentes. A l'inverse, le parallélisme de contrôle exécute des opérations différentes simultanément. Pour cela, les différents traitements exécutés en parallèle doivent être *indépendants*. Deux traitements sont *indépendants sur les données* lorsque leurs opérandes peuvent être utilisées de manière concurrente et lorsque le résultat d'un des traitements n'a pas d'incidence sur l'autre traitement. Une formalisation de ce principe a été établi par H.J. Bernstein en 1966 [2]. On dit également qu'ils sont *indépendants sur le séquencement* si l'exécution d'un des traitements n'est pas conditionné par l'exécution de l'autre traitement.

Statistiquement, la probabilité d'occurence des dépendances croît avec le nombre d'instructions que l'on désire exécuter en parallèle. Autrement dit, le parallélisme est restreint dans chaque application et séquence d'instructions [3, 4]. En pratique, des techniques particulières sont utilisées pour limiter l'effet de ces dépendances. Celles-ci se déclinent en deux approches : la première consiste à accélérer le transfert entre les éléments qui produisent l'information et ceux qui l'utilisent. La seconde prédit, par des techniques statistiques, les valeurs que prendront ces résultats intermédiaires [5]. Les valeurs spéculées permettent de poursuivre l'exécution des instructions tant que celles-ci ne remettent pas en cause l'état des mémoires et des registres. On parle alors d'exécution *spéculative*. Dès que les valeurs des résultats intermédiaires sont connues, les exécutions spéculatives sont transformées en exécutions normales ou sont détruites.

Pour ne pas trop pénaliser les performances, les mécanismes de prédiction doivent être les plus efficaces possibles. D'autres techniques peuvent être utilisées pour réduire les dépendances de données. Elles consistent notamment à *renommer* dynamiquement les registres physiques qui contiennent les dernières valeurs des registres utilisés par le programme. Enfin, des techniques de compilation particulières sont employées pour limiter le coût de l'extraction du parallélisme au cours de l'exécution. Les techniques de compilation permettent de supprimer des dépendances de contrôle et de donnée, ainsi que d'extraire, par exemple, le parallélisme dans les cœurs de boucle.

En fait, deux programmes peuvent être exécutés en parallèle s'ils sont indépendants, si la méthode d'extraction du parallélisme choisie est appropriée et s'il existe suffisamment de ressources pour qu'ils puissent s'exécuter simultanément. En effet, si deux traitements doivent partager une ressource commune, leur exécution concurrente ne garantit plus l'intégrité et la cohérence des données. Des mécanismes de synchronisation doivent alors être mis en place au niveau applicatif ou architectural sous la forme, par exemple, de *sémaphores* ou de *verrous*. Le sémaphore permet la coordination asynchrone entre des traitements parallèles, alors que le verrou peut être utilisé pour rendre exclusif l'accès aux ressources partagées. Cela se traduit en général, au niveau architectural, par la mise en place de registres de synchronisation. Bien souvent, même si les traitements sont considérés comme étant indépendants, les synchronisations demeurent inévitables. Par exemple, une application peut être constituée d'une tâche pour l'affichage des résultats et d'une autre pour leur élaboration. A la fin de l'obtention des résultats, la tâche de calcul se synchronise avec la tâche de visualisation pour les afficher. Ceci contribue à réduire les performances du système puisque la non-disponibilité d'une ressource empêche le traitement de se poursuivre.

Ainsi, il est malheureusement impossible de paralléliser des programmes dans leur intégralité. Ceux-ci sont toujours composés d'une partie séquentielle et incompressible S et d'une partie parallélisable P. Soit s le temps d'exécution de la partie S et p le temps d'exécution de la partie P, la loi d'Amdahl [6] considère que l'accélération maximale qu'il est possible d'obtenir en parallélisant un programme est égale à :

$$A = (s + p)/(s + \tfrac{p}{n})$$

En d'autres termes, quel que soit le nombre de ressources de calcul disponibles, le temps d'exécution d'un programme est toujours borné par sa partie séquentielle. Il faut donc que l'architecture parallèle puisse exécuter efficacement des traitements séquentiels. La loi de Gustafson [7] modère néanmoins les conclusions de la loi d'Amdahl. Il a en effet remarqué que la partie parallèle est composée de boucles qui traitent les données. Ainsi, si le nombre de données à traiter progresse, la contribution de la partie séquentielle tend à se réduire avec le nombre n de ressources de calcul utilisées. Soit a la taille de la partie parallèle attribuée à chaque processeur, l'accélération devient alors :

$$A = (s + a \cdot n)/(s + a)$$

Par conséquent, plus la taille a est importante et plus l'accélération tend vers le nombre de ressources de calcul n, soit l'accélération maximale. Le parallélisme peut donc dépasser la limite établie par la loi d'Amdahl à condition d'augmenter la quantité d'informations à traiter par chacune des ressources de calcul.

Les limitations du parallélisme d'instructions (ILP : Instruction-Level Parallelism) peuvent ainsi être réduites en exploitant davantage un parallélisme de grain plus important. On parle alors de *parallélisme au niveau tâche* (TLP : Thread-Level Parallelism), en considérant une tâche comme un ensemble d'instructions. Ces tâches peuvent être constituées de manière à créer des groupes d'instructions indépendants. Et même si elles peuvent avoir besoin de se synchroniser ou d'échanger des informations avec d'autres tâches, elles accélèreront les temps d'exécution si elles sont allouées sur des ressources de calcul différentes. Une *application monotâche* possède au plus une seule tâche concurrente, tandis qu'une *application multitâche* en contient au moins deux. De même, une *architecture monotâche* ne peut traiter qu'une seule tâche à la fois, alors qu'une *architecture multitâche* peut en exécuter au moins deux simultanément.

Ces architectures peuvent prendre diverses formes en fonction des besoins applicatifs, de contraintes énergétiques, de performance ou de surface. Nous allons donc tout d'abord présenter les contraintes imposées par les systèmes embarqués. Puis nous présenterons l'ensemble des architectures exploitant le parallélisme d'instructions ou de tâches, encore appelées *architectures parallèles* dans la suite de ce manuscrit.

1.2 Contraintes des systèmes embarqués

Afin de confronter ces structures variées, nous proposons différents critères de comparaison. Tout d'abord, il est intéressant de comparer le coût ou la taille de la solution obtenue. En effet, ce paramètre influence directement les coûts de production et de développement du circuit. Il affecte par ailleurs le plus souvent sa consommation d'énergie. C'est pourquoi nous proposons de définir l'*efficacité transistor* comme premier critère. Il représente le nombre de milliards d'opérations par seconde et par millions de transistors ($GOPs/MT$). A titre d'exemple, l'efficacité transistor d'un MIPS 24KE [8], qui est un cœur optimisé pour le calcul, est d'environ 0,6 GOPs/MT. Cette efficacité correspond à une bonne utilisation des transistors pour effectuer du calcul. Par conséquent, une solution intéressante pour les systèmes embarqués doit avoir une efficacité de cet ordre.

La consommation d'énergie est également un moyen de comparaison incontournable. Les systèmes visés sont pour la plupart mobiles et doivent avoir une autonomie suffisante avec une batterie de taille raisonnable. Une autre grandeur caractéristique est donc le nombre de millions d'opérations par seconde et par milliwatt ($MOPs/mW$). Par exemple, une application de téléphonie de troisième génération comme l'UMTS a une complexité d'environ 12 GOPs et doit consommer moins de 500 mW [9]. Par conséquent, une *efficacité énergétique* de 20 MOPs/mW est recommandée pour les futurs systèmes embarqués. Par la suite, ces grandeurs seront normalisées pour une technologie CMOS 130 nm.

Un autre critère comme la complexité de la programmation ou de la mise en œuvre des synchronisations nous semble important à considérer. De même, la capacité de l'architecture à utiliser l'ensemble de sa puissance de calcul permet d'atteindre une meilleure efficacité transistor. Enfin, les applications embarquées doivent pouvoir répondre à des contraintes temps-réel. L'architecture doit donc faciliter la migration des tâches entre les ressources de calcul, ainsi que leur *préemption*. La préemption est l'interruption d'une tâche en cours d'exécution pour une tâche considérée comme plus prioritaire.

A partir des critères que nous venons de présenter, nous allons maintenant nous intéresser aux techniques qui ont été développées pour exploiter le parallélisme d'instructions puis le parallélisme de tâches. La section suivante présente une classification de ces architectures, incluant les architectures multitâches et multiprocesseurs.

1.3 Classification des architectures parallèles

La première classification des architectures parallèles a été proposée par M. Flynn [10]. Elle classe les architectures suivant les relations qui existent entre les unités de traitement et les unités de contrôle. Elle définie quatre modèles d'exécution : *SISD* (Single Instruction Single Data), *SIMD* (Single Instruction Multiple Data), *MISD* (Multiple Instruction Single Data) et *MIMD* (Multiple Instruction Multiple Data).

Le modèle SISD correspond au modèle classique de Von Neuman pour lequel une seule ressource de traitement reçoit un seul flot d'instructions pour traiter un seul flot de données. Dans une architecture SIMD, une seule unité de contrôle distribue plusieurs flots d'instructions identiques simultanément. Comme chaque unité de traitement traite un flot de données différent, la même opération est appliquée à plusieurs données en parallèle. Les architectures MISD appliquent sur un seul flot de données plusieurs traitements en parallèle. Ce sont les architectures *pipeline*. Comme dans une chaîne de montage industrielle, chaque flot de données est exécuté par autant de modules matériels travaillant successivement. Enfin, plusieurs unités de contrôle gèrent chacune une ou plusieurs unités de traitement dans les architectures MIMD. Ce dernier modèle correspond à la grande majorité des architectures parallèles.

Se restreindre à cette classification des architectures parallèles empêche de mettre en évidence leurs différences et leurs caractéristiques. Par exemple, il existe des différences importantes au niveau architectural dans les architectures SIMD et MIMD. Entre les architectures *EPIC* (Explicitly Parallel Instruction Computing) et superscalaire, ou les architectures multitâches et multiprocesseurs, les principes d'exécution sont tellement différents qu'ils ne peuvent pas être regroupés si l'on souhaite comprendre leurs spécificités. Par ailleurs, les architectures SIMD ne sont utilisées que dans les machines parallèles, qui ne font pas l'objet de ce chapitre. Une nouvelle classification est donc proposée figure 1.1.

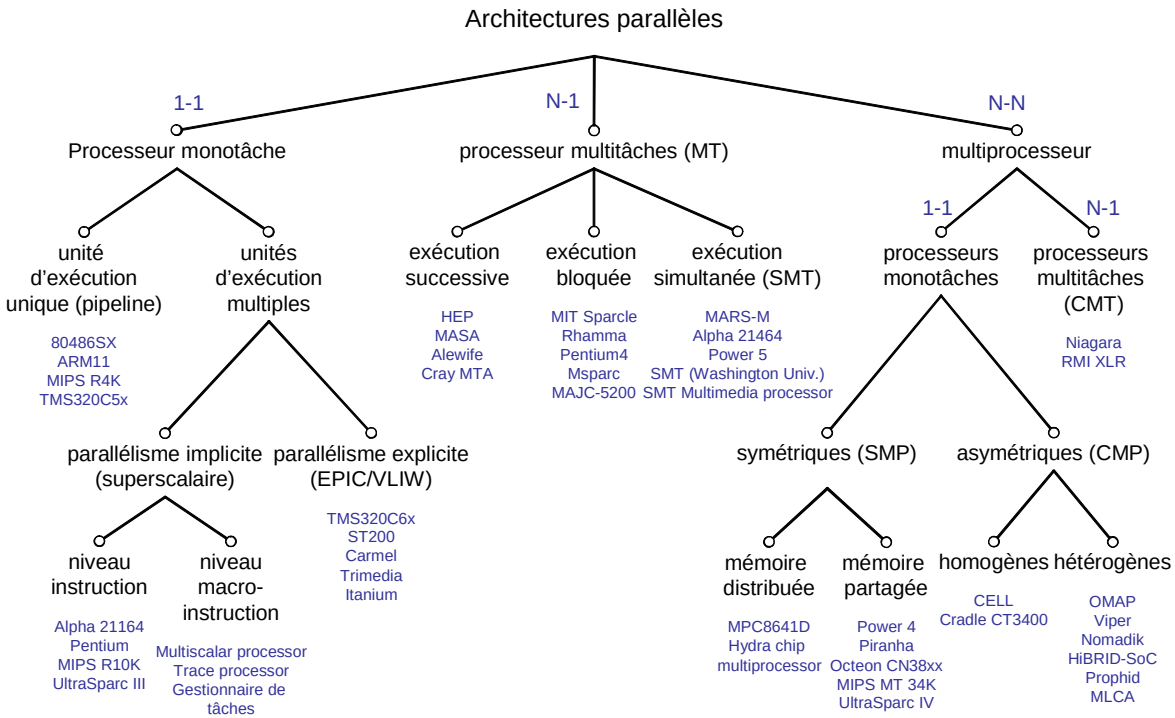

FIG. 1.1 – Classification des architectures parallèles sur une même puce

Elle propose trois grandes catégories : les *processeurs monotâches*, les *processeurs multitâches* et les *multiprocesseurs*. Pour la suite, nous considérons un *processeur* comme étant une architecture de calcul. Dans les processeurs monotâches, nous retrouvons toutes les architectures qui exploitent le parallélisme d'instructions pour accélérer le traitement d'une tâche. Les processeurs multitâches regroupent les architectures qui sont capables d'exécuter

plusieurs tâches simultanément sur une seule unité de traitement. Enfin, les multiprocesseurs ou MPSoC (Multi-Processor System on Chip) sont des architectures constituées de multiples processeurs monotâches ou multitâches. Les architectures multiprocesseurs considérées dans cette thèse sont toutes intégrées sur une même puce.

Les sections suivantes présentent dans l'ordre ces trois grandes familles d'architectures parallèles. Les critères introduits précédement seront utilisés comme éléments de comparaison et permettront de discuter l'intérêt de chacune des solutions étudiées. Dans un premier temps, nous nous intéresserons aux techniques d'accélération utilisées par les processeurs monotâches, avec une ou plusieurs unités d'exécution. Ce sera l'occasion de revenir sur le fonctionnement des architectures pipeline, superscalaire ou EPIC. Ces solutions n'exploitent que le parallélisme d'instructions et restent limitées pour répondre aux besoins des futures applications mobiles. Néanmoins il est intéressant de comprendre leurs limitations et leur fonctionnement, car elles sont les éléments de base des solutions multiprocesseurs.

1.4 Techniques d'accélération des processeurs monotâches

Pour accélérer le traitement sur un processeur monotâche, deux solutions sont possibles. La première appelée *parallélisme temporel* consiste à réduire le temps d'exécution en découpant le traitement d'une instruction en plusieurs étapes successives. Elle représente la seule solution qu'il est possible de mettre en œuvre pour réduire le temps de traitement à partir d'une seule unité d'exécution. La seconde solution nommée *parallélisme spatial* repose sur la multiplication des ressources de calcul.

1.4.1 Unité d'exécution unique

La technique principale utilisée pour accélérer la vitesse de traitement d'un processeur, avec une seule unité d'exécution, consiste à l'organiser en pipeline. Ce mécanisme, inventé par Michael J. Flynn en 1966 [11], consiste à scinder le traitement d'une instruction en plusieurs étages successifs. Comme le montre la figure 1.2, le traitement d'une instruction se compose de 4 étages principaux. L'instruction est chargée depuis la mémoire d'instruction, puis elle est décodée pour configurer et sélectionner l'unité d'exécution. Après son exécution, le résultat de l'instruction est rangé en mémoire. Dès qu'une instruction quitte le premier étage de chargement pour passer dans l'étage de décodage, une nouvelle instruction vient occuper le premier étage. En régime établi, plusieurs instructions sont donc en exécution dans la machine à des degrés d'avancement différents. Avec un pipeline de 4 étages, il suffit de 7 cycles pour exécuter 4 instructions.

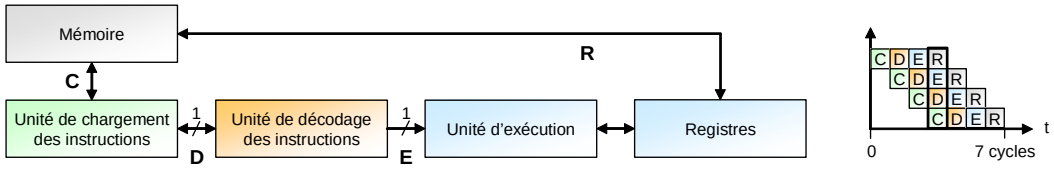

FIG. 1.2 – Architecture pipeline et principe d'exécution

Aujourd'hui, les architectures pipeline sont très répandues. On peut citer à titre d'exemple les architectures Intel 80486SX [12], ARM 11 [13], MIPS R4K [14] ou le processeur *DSP* (Digital Signal Processing) TMS320C5x de Texas Instrument [15].

Avec 5 étages de pipeline, l'ARM 11 a une efficacité transistor de 0,24 GOPs/MT pour une efficacité énergétique de 1,5 MOPS/mW. Même si la puissance de calcul de ce processeur est insuffisante, ces chiffres traduisent une assez bonne utilisation des transistors. En fait, plus le traitement d'une instruction est découpé et plus le débit des résultats augmente. Néanmoins, les temps d'accès aux instructions et aux données ne doivent pas être supérieurs au temps de chaque étage du pipeline. Il faut donc hiérarchiser la mémoire pour réduire ses temps d'accès. Ensuite, plus le pipeline est profond, plus les pénalités induites par des dépendances de données ou de contrôle sont importantes. Par exemple, dans les dernières architectures, une mauvaise prédiction sur un branchement peut entraîner 30 cycles de latence supplémentaires. Ainsi, cette technique d'accélération nécessite des mécanismes de branchement performants. Par conséquent, leur complexité augmente avec la profondeur du pipeline. Ceci tend à limiter l'accélération apportée par ce parallélisme temporel.

1.4.2 Unités d'exécution multiples

Augmenter les performances du pipeline peut se faire en décodant et en exécutant plusieurs instructions simultanément. Ceci est rendu possible en multipliant le nombre de ressources de calcul (UF : Unité Fonctionnelle) disponible. Ces architectures sont également de type MISD. Elles peuvent cependant intégrer des unités fonctionnelles capables d'exécuter une opération sur plusieurs données d'un même flot. Ces unités couramment appelées SIMD sont en fait des opérateurs *SWP* (Sub-Word Parallelism). L'expression du parallélisme dans ce type d'architecture peut-être *explicite* ou *implicite*. Le parallélisme est explicite lorsque le compilateur gère les dépendances de données et le flot de contrôle. Il garantit alors la disponibilité des ressources. Le contrôle est alors relativement simple et permet d'utiliser des fréquences d'horloge plus élevées. Au contraire, lorsque l'architecture s'occupe dynamiquement de tous ces aléas d'exécution, le parallélisme est exploité de manière implicite.

Parallélisme implicite

La technique superscalaire, apparue en 1989 [16], consiste à dupliquer les différents étages du pipeline pour pouvoir lire et exécuter plusieurs instructions simultanément (*superpipeline*). Un procédé de distribution et d'allocation est ajouté au cycle d'exécution afin de sélectionner dynamiquement les instructions à traiter. Les premiers processeurs superscalaires utilisent un chargement *statique* des instructions (Intel Pentium [17], DEC Alpha 21164 [18]). Les instructions sont lues par paquet et exécutées simultanément lorsqu'elles sont admissibles. Sinon, lorsque l'exécution est impossible, le reste des instructions est exécuté au cycle suivant. Avec cette approche, le parallélisme dépend essentiellement du compilateur et du nombre de groupes d'instruction qui peuvent être exécutés en parallèle. Des instructions d'un groupe peuvent néanmoins nécessiter le résultat du groupe suivant. En d'autres termes, il faut être capable d'exécuter les instructions de manière non-ordonnée pour utiliser toutes les unités fonctionnelles. C'est pourquoi, des mécanismes de spéculation et de renommage sont joints à la distribution des tâches dans les architectures à chargement *dynamique*. Les architectures MIPS R10K [19] ou Sun UltraSparc III [20] sont des exemples caractéristiques.

Dans les architectures superscalaires à chargement dynamique, les instructions sont sélectionnées dans l'ordre imposé par le programme. Elles sont exécutées dans le désordre en veillant seulement à respecter les dépendances sur les données. Le dernier étage assure le réordonnancement des résultats calculés afin de paraître extèrieurement comme un processeur von Neumann. Toutes les techniques dynamiques qui sont apparues dans ces processeurs ont permis d'optimiser l'utilisation des ressources de calcul et de détecter les problèmes de parallélisme entre les instructions. Comme le montre la figure 1.3, dans une architecture superscalaire d'ordre 2 (avec 2 UF), le temps d'exécution des 4 instructions est réduit à seulement 5 cycles avec un pipeline de 4 étages.

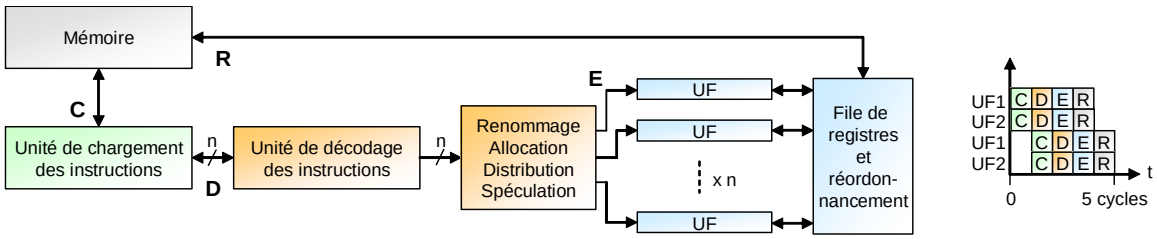

FIG. 1.3 – Architecture superscalaire et principe d'exécution

Bien que la plupart des architectures superscalaires exécutent des instructions élémentaires, d'autres solutions ont été développées. Elles permettent de sélectionner des groupes d'instruction appelés *macro-instructions*. Ces macro-instructions sont des ensembles d'instructions indépendantes qui sont exécutés sans spéculation par les unités fonctionnelles. A titre d'exemple, il existe le Multiscalar Processor de l'université du Wisconsin [21], le Trace Processor [22] de l'université de la Caroline du Nord, ainsi que le Gestionnaire de tâches de X. Verians [23]. Ce dernier exécute des flots d'instructions avec des techniques similaires à celles employées dans les architectures superscalaires. Il est néanmoins particulier dans le sens où les files d'attente des tâches à exécuter contiennent l'information de dépendance entre les blocs d'instructions. Cependant, ceci n'empêche pas l'exécution spéculative des tâches et les techniques de prédiction de banchement sont, par exemple, transposées pour un grain de calcul plus important.

Les avantages de l'approche superscalaire sont la simplicité de la description du parallélisme et sa gestion dynamique durant l'exécution. L'inconvénient est la nécessité de disposer de ressources spécialisées à même d'effectuer cette extraction. D'ailleurs, dans de telles architectures, une faible partie de la surface est utilisée pour le calcul. La complexité des mécanismes d'exécution non-ordonnée ou de prédiction de branchement engendre des mises en œuvre inefficaces. Pour mieux s'en rendre compte, prenons l'exemple du processeur Intel Pentium cadencé à 133 MHz. Ce dernier a une efficacité transistor égale à 0,18 GOPs/MT et une efficacité énergétique de seulement 0.02 MOPs/mW. Il est vrai que ce processeur est destiné aux traitements généralistes, mais par rapport à une architecture pipeline simple, l'efficacité transistor a été environ divisée par 1,5 et l'efficacité énergétique par 75.

L'approche architecturale qui exploite un parallélisme d'instructions explicite subit moins ces inconvénients. Nous allons étudier son principe afin de mieux comprendre ses limitations.

Parallélisme explicite

Les architectures consituées de plusieurs unités de traitement qui supportent le parallélisme explicite choisissent statiquement les instructions à exécuter. Dans cette technique d'exécution, le processeur reçoit directement de la mémoire de longs mots qui contiennent plusieurs instructions sans dépendances relatives. Ces architectures sont couramment appelées EPIC ou VLIW. La principale différence entre ces deux approches réside dans la taille de l'instruction à exécuter. Dans une architecture EPIC la taille de l'instruction est variable. Cela permet de réduire partiellement les défauts d'instruction qui surviennent lorsque le mot de l'instruction est plus grand que l'ensemble des instructions à exécuter en parallèle. Un exemple d'architecture EPIC est l'Intel Itanium [24].

Dans les architectures VLIW telles que le processeur TMS320C6x de Texas Instrument [25], le ST200 de ST Microelectronics [26], le processeur Carmel de RealChip [27], ou le Trimedia de Philips Semiconductors [28], la fenêtre d'instructions est constante. Sans compilateur performant il est difficile de ne pas sous-occuper l'ensemble des ressources de calcul disponibles. Néanmoins, certaines approches permettent de supprimer un grand nombre de branchements conditionnels et donc de réduire d'autant l'importance de leur prédiction. En effet, les deux branches de la condition sont exécutées simultanément. Le choix du résultat est effectué seulement après l'obtention de la condition de branchement.

La figure 1.4 montre bien la simplification de l'étage de distribution et d'allocation. La simplification du contrôle permet d'améliorer grandement l'efficacité transistor et énergétique. Par exemple, l'efficacité transistor du Trimedia atteint 0,96 GOPs/MT pour une efficacité énergétique de 0,88 MOPs/mW. Lorsque le parallélisme est bien exploité, ou encore que les performances efficaces sont proches des performances crêtes, la simplicité de ces architectures en fait des solutions très efficaces.

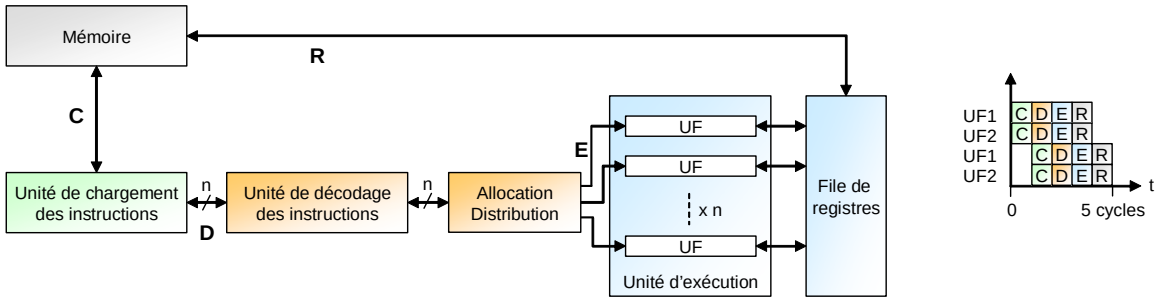

FIG. 1.4 – Architecture EPIC/VLIW et principe d'exécution

Pourtant, de nombreux registres sont nécessaires pour sauvegarder les différents opérandes et résultats. De plus, la largeur de l'instruction sélectionnée est contrainte par la bande passante de la mémoire d'instructions, qui possède un nombre de ports limité. Enfin, une incompatibilité forte existe entre les codes exécutables de différentes architectures VLIW. L'architecture Itanium développée par Intel parvient malgré tout à modérer cet inconvénient. Cependant, son domaine d'utilisation trop généraliste et la mise en œuvre d'un trop grand nombre d'unités de calcul lui impose un contrôle plus important et entraîne une consommation d'énergie excessive. Celle-ci est supérieure à 170 W. En effet, le parallélisme d'instructions est limité et une utilisation optimale des ressources nécessite un contrôle complexe si le compilateur n'est pas suffisamment efficace.

1.4.3 Conclusion

Cette première section a permis de mieux comprendre le fonctionnement et la complexité des différentes techniques d'accélération monotâches. Les architectures pipeline ou EPIC disposant d'un contrôle simple sont des solutions acceptables en terme d'efficacité transistor ou énergétique. Par contre, dès que l'on tente d'exagérer un peu le concept en augmentant la taille de la fenêtre d'instructions, très vite l'architecture devient inefficace et inadaptée aux systèmes embarqués. Elle devient limitée par le parallélisme d'instructions et la performance des compilateurs. De plus, la préemption d'une tâche a peu d'impact sur les performances puisque le nombre de cycles de pénalité est de l'ordre du nombre d'étages du pipeline. Cependant, plus le pipeline est profond et moins l'architecture est réactive. Par ailleurs, nous avons vu que le parallélisme d'instructions est limité dans chaque programme ou séquence d'instructions. Réduire l'accélération d'une architecture à l'exploitation du parallélisme d'instructions ne peut donc que conduire à une augmentation de la complexité pour peu de gain en performance supplémentaire. La solution est donc d'exploiter le parallélisme qui existe entre les tâches dans une application multitâche. Néanmoins, l'exécution de plusieurs tâches sur ces architectures n'apporterait aucun gain en performance puisqu'elles seraient exécutées séquentiellement. C'est pourquoi il faut intégrer des mécanismes de contrôle supplémentaires et concevoir des processeurs multitâches.

1.5 Les processeurs multitâches

La première observation que l'on peut faire sur les architectures monotâches est qu'elles ne peuvent pas exploiter la totalité de leurs ressources de calcul à chaque cycle d'exécution [29]. Ceci est principalement dû aux mécanismes de spéculation qui n'ont pas un comportement optimal et au parallélisme d'instructions qui est limité. Le principe des architectures multitâches est donc d'occuper ces ressources par d'autres tâches. Ces architectures exécutent alors plusieurs flots d'instructions indépendants sur plusieurs flots de données. Selon la classification de M. Flynn, les processeurs multitâches sont des architectures MIMD. Comme le montre la figure 1.5, ces architectures sont constituées de plusieurs compteurs de programmes indépendants, ainsi que de mécanismes permettant la sélection de plusieurs instructions issues de tâches indépendantes. Ces architectures optimisent le temps de commutation d'une tâche à une autre. Ce temps peut être nul ou valoir seulement quelques cycles d'exécution.

De manière générale, une architecture multitâche garde le principe d'exécution des architectures superscalaires ou EPIC. Néanmoins, certains étages se complexifient pour exécuter plusieurs flots de données et d'instructions différents. Ainsi, la première étape sélectionne plusieurs instructions en parallèle à des emplacements mémoires différents. Ces différentes adresses sont générées par les multiples compteurs de programmes qui permettent d'exécuter autant de tâches simultanément. Pour pouvoir accéder à plusieurs flots indépendants et soutenir l'alimentation en instructions du superpipeline, plusieurs files d'instructions sont utilisées. De plus, l'étage de renommage et de spéculation qui est dépendant du flot d'instructions traité est multiplié. Enfin, il faut également augmenter le nombre des registres pour sauvegarder les différents opérandes et résultats.

Il existe de nombreuses architectures multitâches [30, 31, 32], mais elles peuvent être regroupées en trois grandes catégories. La première technique appelée *exécution successive* ou

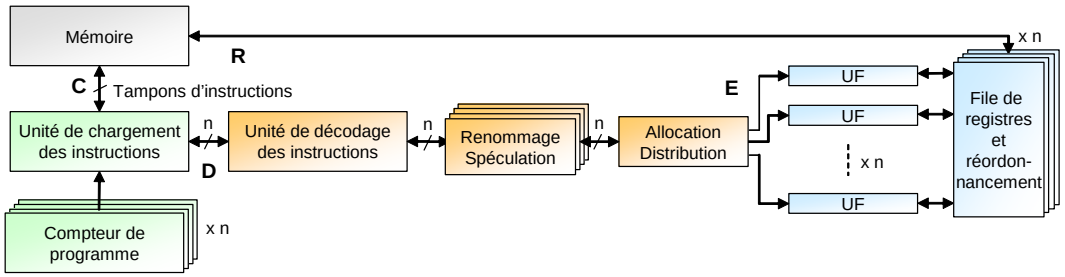

FIG. 1.5 – Architecture multitâche et principe d'exécution

interleaved multithreading consiste à exécuter les tâches actives les unes à la suite des autres à chaque cycle d'exécution. La seconde approche, nommée *exécution bloquée* ou *blocked multithreading*, exécute successivement les tâches chaque fois qu'une tâche est bloquée. A titre d'exemple, une tâche peut être bloquée si une de ses instructions a besoin du résultat produit par une autre instruction, ou si la mémoire ne contient pas l'instruction à exécuter. Enfin, la troisième solution est capable de traiter simultanément plusieurs tâches d'un programme. Elle est appelée *exécution simultanée* ou *simultaneous multithreading*.

1.5.1 Exécution successive

Les architectures multitâches à exécution successive change la tâche à exécuter après chaque cycle d'exécution (figure 1.6-a). Les architectures HEP [33], MASA [34], Alewife [35] ou Cray MTA [36] illustrent ce principe. L'avantage de cette solution est qu'il n'est plus nécessaire de disposer de matériel complexe pour spéculer l'exécution des instructions. Les dépendances de contrôle et de données sont éliminées. De plus, le pipeline peut atteindre un bon niveau d'occupation. Par ailleurs, la latence d'accès aux instructions ou du changement de contexte n'a pas d'incidence sur les temps d'exécution. Cependant, cette architecture nécessite d'exécuter simultanément au moins autant de tâches qu'il y a d'étages de pipeline. Il est cependant possible de permettre l'exécution successive d'instructions indépendantes d'une même tâche. Mais ceci est parfois difficile à obtenir par les outils de compilation. Le fait d'avoir conçu une architecture devant exécuter plusieurs tâches les unes à la suite des autres limite la puissance de calcul disponible pour une seule tâche.

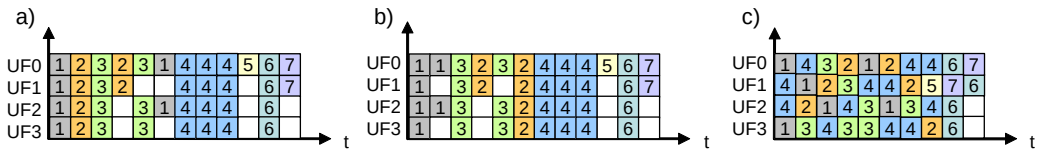

FIG. 1.6 – Exemple d'exécution dans une architecture multitâche. Pour cet exemple, un ensemble de tâches indépendantes $\{T1, T2, T3, T4, T5, T6, T7\}$ est choisi. Dans le cas d'une exécution successive (a), l'exécution de chacune des tâches est alternée à chaque cycle d'exécution. Au contraire, les tâches peuvent s'exécuter les unes après les autres jusqu'à l'arrivée d'un évènement : c'est l'exécution bloquée (b). Par exemple, la tâche bloquée $T3$ permet l'exécution de la tâche $T2$. Enfin, l'exécution simultanée exécute indifféremment les tâches en fonction de la disponibilité des ressources de calcul (c).

Le processeur MTA développé par Cray a, par exemple, une efficacité transistor de 0,62-GOPs/MT et une efficacité énergétique inférieure à 0,02 MOPs/mW. Par conséquent, l'efficacité transistor a été améliorée par rapport aux architectures superscalaires, malgré un grand nombre de registres pour l'exécution successive de ses 128 tâches. Ceci est principalement dû à la simplification de l'architecture et de son contrôle.

1.5.2 Exécution bloquée

L'approche avec exécution bloquée consiste à exécuter chaque tâche jusqu'à ce qu'elle rencontre une situation bloquante (figure 1.6-b). En pratique, cette situation intervient lors d'un accès à une ressource partagée ou en cas de dépendance de données ou d'instructions. Contrairement à la solution précédente, un nombre inférieur de tâches est nécessaire et une tâche peut s'exécuter sans interruption jusqu'au prochain changement de contexte. En fait, les performances de l'exécution d'une seule tâche sont similaires à celles obtenues avec des architectures superscalaires. Le changement de contexte peut être obtenu statiquement ou dynamiquement.

Dans les architectures avec changement de tâche statique, le changement de contexte est généré par le compilateur. L'avantage est que le changement de contexte est prévu et que sa latence est réduite à zéro. Au contraire, les architectures avec changement de tâche dynamique souffrent d'une pénalité élevée lors de chaque blocage. Celle-ci correspond au temps nécessaire pour recharger le pipeline d'exécution.

Plusieurs techniques dynamiques peuvent être utilisées. Tout d'abord, lorsqu'une sauvegarde ou un chargement échoue en mémoire, le temps nécessaire pour recharger la mémoire cache est utilisé pour exécuter une autre tâche. Ensuite, d'autres techniques permettent le changement de tâche lorsque la donnée utile n'est toujours pas prête ou lors de branchements conditionnels.

En pratique, les architectures multitâches mettent en œuvre plusieurs politiques de changement de tâche. Par exemple, les processeurs Sparcle du MIT [37] ou MSparc de l'université de Oldenburg [38] permutent entre deux tâches lors de défauts de cache et sur l'occurence d'autres signaux de contrôle gérés dynamiquement. Les processeurs Rhamma de l'université de Karlsrühe [39] ou MAJC-5200 de Sun [40] intègrent des changements de tâche statique et dynamique. Par ailleurs, Intel a récemment proposé une technique appelée *Hyperthreading* [41] utilisée dans le pentium 4 [42]. Elle réunit des mécanismes de changement de tâche dynamique et peut exécuter jusqu'à deux tâches simultanément.

Le processeur Sun MAJC-5200 a une efficacité transistor de 0,46 GOPs/MT et une efficacité énergétique de 0,29 MOPs/mW. Le processeur Intel pentium 4, avec une efficacité transistor de 0,22 GOPs/MT et une efficacité énergétique de 0,15 MOPs/mW est, quant-à lui, moins performant. Par rapport à une exécution successive, l'efficacité énergétique est améliorée de façon conséquente. Ceci est principalement dû à une meilleure utilisation des ressources de calcul. En effet, ces architectures obtiennent de bons résultats même si le nombre de tâches à exécuter en parallèle est inférieur à la taille du pipeline. De plus, il n'est plus nécessaire de disposer de multiples registres de sauvegarde puisque les tâches s'exécutent les unes après les autres. Cependant, la mise en œuvre d'un changement de tâche dynamique a une complexité bien plus importante. Ceci se ressent sur l'efficacité transistor.

1.5.3 Exécution simultanée

Contrairement aux deux autres techniques d'exécution qui sont utilisées avec des processeurs pipeline ou VLIW, l'exécution simultanée est associée aux architectures superscalaires. Cette technique est communément appelée *Simultaneous Multithreading* ou SMT. Elle hérite des architectures superscalaires la possibilité d'exécuter de multiples instructions à chaque cycle d'exécution [43]. La principale différence réside dans le fait qu'elle peut accéder simultanément à plusieurs flots d'instructions indépendants. Chaque cycle peut conduire à exécuter des opérations appartenant à des tâches différentes (figure 1.6-c). Les pénalités qui interviennent dans l'exécution de chacune des tâches sont réduites considérablement par l'exécution simultanée d'autres instructions. Les processeurs MARS-M [44], Compaq Alpha 21464 [45], Power 5 [46], SMT de l'Université de Washington [29], ou SMT Multimedia Processor de l'université de Karlsrühe [47] sont quelques exemples d'architecture SMT.

Les avantages de ces architectures ne sont plus à démontrer. Elles constituent sans nul doute la solution la plus aboutie pour exploiter le parallélisme de tâches. Elle offre un grand niveau de flexibilité et de performance. Néanmoins, ces architectures ne peuvent pas être utilisées dans les systèmes embarqués. Elles sont exclusivement réservées pour les supercalculateurs ou les machines parallèles. En effet, de nombreux étages de pipeline doivent être multipliés par le nombre d'instructions à exécuter en parallèle. De même, le nombre de ports de la mémoire d'instructions doit permettre l'exécution simultanée de plusieurs tâches. Pour s'en rendre compte, intéressons-nous aux caractéristiques du processeur Alpha 21464 de la société Compaq. Ce processeur multitâche a une efficacité transistor de 0,064 GOPs/MT et une efficacité énergétique de 0,064 MOPs/mW. Ces résultats, très inférieurs à ceux obtenus jusqu'alors, ne sont pas surprenants dans la mesure où ils complexifient encore le modèle superscalaire.

1.5.4 Synthèse des processeurs multitâches

Les architectures multitâches améliorent l'occupation des unités fonctionnelles des architectures monotâches [48]. Quelle que soit la façon dont les tâches sont exécutées, ces architectures offrent une solution au problème d'exploitation de programmes multitâches. Elles permettent de passer très rapidement d'une tâche à une autre et sont donc particulièrement disposées pour la préemption de tâches. La programmation de ces architectures reste simple puisque toute la complexité est supportée par l'architecture ou par le compilateur. Néanmoins, ces architectures souffrent toutes d'une trop grande complexité parce qu'elles enrichissent encore des solutions jugées inefficaces pour les systèmes embarqués. Par ailleurs, elles sont limitées par la bande passante de la mémoire d'instructions et consomment toutes énormément d'énergie. Par exemple, le processeur Alpha 21464 consomme environ 250 Watts. On est très loin des 500 mW requis pour les systèmes embarqués. Enfin, ces architectures ne sont pas adaptées pour répondre aux contraintes temps-réel. En effet, l'ordre d'exécution des tâches ne peut pas respecter des contraintes de priorité et l'exécution des tâches est très peu déterministe. Pour toutes ces raisons, ces solutions sont définitivement à prohiber pour des systèmes embarqués mobiles. Il faut donc considérer une approche sensiblement différente pour exploiter à la fois le parallélisme de tâches et respecter les contraintes que nous nous sommes fixées. Pour cela, nous allons étudier des solutions qui consistent à regrouper sur un même circuit plusieurs processeurs monotâches ou multitâches.

1.6 Les multiprocesseurs

Pour exploiter le parallélisme de tâches, la première solution a consisté à améliorer et à enrichir les modèles d'architecture initialement conçus pour exploiter le parallélisme d'instructions. Ces processeurs multitâches sont capables d'exécuter plusieurs instructions issues de différentes tâches simultanément. Néanmoins, comme nous avons pu le constater, ces solutions souffrent d'une importante complexité qui les rendent inutilisables dans les systèmes embarqués mobiles. Il existe heureusement une autre solution pour exploiter le parallélisme de tâches avec une meilleure efficacité. Celle-ci consiste à multiplier les processeurs comme nous avons multiplié les unités d'exécution dans les architectures superscalaires [49]. A performance égale, ces solutions peuvent conserver des fréquences de fonctionnement et une tension d'alimentation réduites. Par conséquent, elles conduisent à une réduction de leur consommation d'énergie et à une augmentation de leur performance. A titre d'exemple, avec le nombre de transistors utilisés dans le Alpha 21464, il est possible d'intégrer près de 90 processeurs ARM 11, soit un gain théorique en performance de 4. La première solution multiprocesseur étudiée dans cette section est certainement la plus simple. Elle repose sur la mise en œuvre de deux cœurs de processeurs monotâches indépendants avec un bus partagé.

1.6.1 Avec des processeurs monotâches

Symétriques

L'association de plusieurs processeurs indépendants sur un bus partagé conduit à des solutions appelées *multiprocesseurs symétriques* (SMP : Symmetric Multiprocessor). Ces architectures proposent d'allouer statiquement ou dynamiquement des tâches sur les différents processeurs qui composent l'architecture. Par exemple, dans la figure 1.7, le premier processeur élémentaire PE0 exécute les tâches 3 puis 5, alors que le processeur PE3 traite uniquement la tâche 4. Ce modèle d'architecture est très simple, mais il introduit un problème que l'on n'avait pas encore rencontré jusqu'alors : le partage des données. La tâche 3 peut, par exemple, avoir besoin de partager des données avec la tâche 4. Bien sûr, il est possible de partitionner des ensembles de tâches lors de l'allocation, afin de garantir que toutes les tâches disposent localement de leurs données. Mais le partitionnement est sous-optimal et ne peut conduire qu'à une faible exploitation des ressources de calcul [50]. Ainsi, on sent bien que le problème majeur réside dans la programmation, la mise en œuvre des synchronisations et le partage des données. Deux modèles s'opposent alors : le modèle avec mémoire distribuée et celui avec mémoire partagée.

La première solution consiste à distribuer la mémoire entre tous les processeurs, comme le représente la figure 1.7-a. Les architectures MPC8641D de Freescale [51] ou Hydra Chip Multiprocessor de l'université de Stanford [52] empruntent ce procédé. Il a l'avantage de rendre exclusif les accès en écriture des données. En effet, chaque mémoire étant gérée localement par un des processeurs, deux écritures simultanées sont impossibles. Reste que bien souvent, les tâches ont besoin de partager des données. Ainsi, une tâche peut nécessiter des résultats ou des données produits par une autre tâche qui s'est exécutée ou qui s'exécute sur une ressource de calcul distante. Dans ce cas, il faut mettre en place une communication explicite par envoi de messages, qui peut se révéler très pénalisante en terme de performance. En plus de limiter les performances, ces synchronisations sérialisent le flot de contrôle et chaque nouvelle dépendance oblige la tâche à interrompre son exécution.

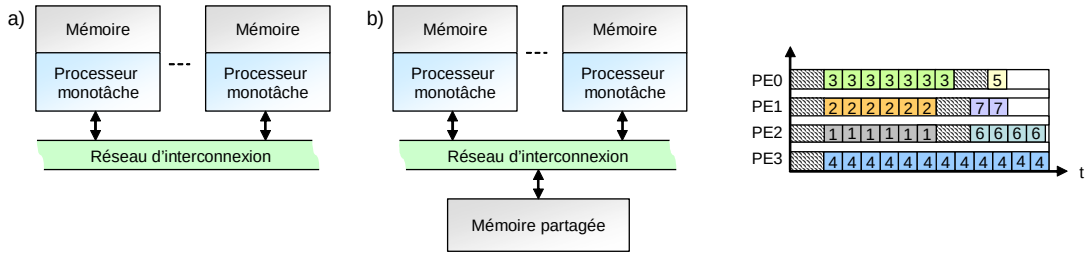

FIG. 1.7 – Architecture multiprocesseur symétrique constituée de multiples processeurs monotâches. Soit un ensemble de tâches indépendantes $\{T1, T2, T3, T4, T5, T6, T7\}$, chaque processeur élémentaire (PE) se voit attribuer un groupe de tâches à exécuter. L'ensemble des données peut être réparti dans des mémoires distribuées (a), ou être regroupé dans une mémoire partagée (b).

Pour s'affranchir de la pénalité induite par les communications entre les tâches, un autre modèle peut être utilisé. Celui-ci repose sur l'utilisation d'une mémoire partagée entre les processeurs distants. L'avantage est que les différentes tâches manipulent le même espace d'adressage. Elles peuvent alors facilement se partager des données ou se synchroniser. C'est le modèle retenu par les processeurs MIPS MT 34K [53], IBM Power 4 [54], Sun UltraSparc IV [55], Piranha [56], ou Octeon CN38xx de la société Cavium Networks [57]. Néanmoins, il faut protéger les accès concurrents aux ressources de mémorisation partagée à l'aide de mécanismes de synchronisation, au risque de diminuer les performances du système. En effet, il faut séquentialiser les accès en lecture et en écriture si la mémoire ne peut lire ou écrire qu'une donnée à la fois. Si la mémoire offre des accès multiples, il faut par ailleurs protéger les données en écriture car deux écritures simultanées à un même emplacement mémoire peut provoquer une erreur de mémorisation.

Pour ne pas trop pénaliser les temps d'exécution, les architectures SMP conservent localement une copie des données partagées. Il peut donc y avoir des problèmes de *cohérence*. Si la mise à jour des données par une tâche n'est pas prise en compte par les autres tâches, chacune travaille sur des données différentes et les résultats produits ne sont plus cohérents. Par conséquent, il faut mettre en place des solutions matérielles [58]. La première consiste à utiliser un contrôleur de cache (*Multiple Write Through*). Il assure que la mémoire cache de chaque ressource de calcul est cohérente avec les données présentes en mémoire globale. Ce type de solution présente l'avantage d'un partage immédiat des données, facilitant ainsi la programmation. Mais ces mécanismes ont un coût d'intégration élevé et induisent des baisses de performance significatives dûes, notamment, à la forte charge du réseau. Sans compter que celle-ci augmente avec le nombre de ressources de calcul. Ainsi, la gestion de la cohérence a un impact important sur la bande passante du système. Par ailleurs, il reste à la charge du programmmeur de vérifier la cohérence des données critiques en synchronisant leurs accès.

Une autre technique, plus simple à mettre en œuvre, repose sur l'espionnage du bus partagé (*Multiple Write Back*). Chaque contrôleur de cache espionne les transactions sur le bus et compare l'adresse de transfert en cours avec le contenu de sa mémoire cache. Si l'un des caches possède une copie de la donnée transférée, une mise à jour est effectuée. Cette technique a une complexité moindre mais suppose l'utilisation d'un unique réseau de communication.

Bien que ces solutions paraissent simples à mettre en œuvre, elles introduisent des problèmes d'accès aux données et des problèmes de cohérence. Ils peuvent être résolus au moment de la programmation, mais empêchent d'exploiter dynamiquement les ressources de calcul et

limitent les performances de l'architecture. Pour simplifier les problèmes de programmation, il est néanmoins possible d'enrichir l'architecture de nombreux mécanismes de cohérence et de protection de l'intégrité des données. Même si ceci contribue à réduire l'efficacité de l'architecture. A titre d'exemple, le processeur Power4 d'IBM a une faible efficacité transistor de 0,05 GOPs/MT et une efficacité énergétique de 0,12 MOPs/mW. Des travaux de recherche ont proposé des solutions appelées *hybrides flot de données* pour ne pas avoir besoin de rajouter de complexité matérielle [59]. Ces modèles d'architecture réduisent les synchronisations en reliant les processeurs entre eux pour réaliser un réseau similaire aux réseaux de processus de Kahn [60]. Cependant, ils n'offrent que peu de flexibilité et ne peuvent pas convenir à nos critères de sélection. Nous allons donc nous intéresser à d'autres solutions multiprocesseurs. Ces solutions présentent la particularité d'avoir un contrôle centralisé et ainsi de simplifier les synchronisations et les échanges de données.

Asymétriques

Dans tous les modèles d'architecture étudiés jusqu'alors, les processus dominés par le contrôle sont exécutés avec les mêmes ressources que celles qui exécutent les processus critiques ou intensifs. Autrement dit, les processeurs supportent l'exécution de processus *réguliers* et *irréguliers*. Un traitement régulier exécute une suite d'instructions dont l'ordre d'exécution est prévu et non-conditionné. Au contraire, un traitement irrégulier regroupe toutes les instructions conditionnelles nécessaires au contrôle des applications. Ainsi, un processus régulier ne nécessite pas de prédiction de branchements ou de mécanismes de contrôle évolués. L'utilisation d'une structure unique pour effectuer des traitements de nature différente empêche leur optimisation. Les multiprocesseurs asymétriques, constitués de multiples processeurs monotâches, tentent d'y apporter une solution. Ces architectures sont constituées d'une ressource de contrôle particulière, souvent un processeur RISC (Reduced Instruction Set Computer) et de multiples ressources de calcul homogènes ou hétérogènes. Elles sont appelées Chip Multiprocessors (CMP).

Comme le montre la figure 1.8, ce modèle a des similitudes avec les architectures superscalaires. Néanmoins, la sélection, la gestion des synchronisations et l'allocation des tâches sont réalisées par un processeur de contrôle distant des autres processeurs de calcul. Ce processeur décide statiquement ou dynamiquement de l'allocation des tâches. Il a pour avantage de connaître l'ensemble des tâches en cours d'exécution. L'autre intérêt est la simplicité du partage des données qui pose tant de problèmes dans les modèles SMP. En effet, toutes les informations partagées entre les tâches peuvent être gérées par le processeur de contrôle. Par conséquent, les problèmes de cohérence et d'intégrité des données sont simplifiés. Ensuite, des processeurs plus disposés aux calculs intensifs, comme les processeurs DSP ou VLIW, peuvent être utilisés de façon efficace comme processeurs de calcul.

Homogènes Les premières solutions CMP disposent de processeurs de calcul identiques. Elles sont dîtes *homogènes*. C'est par exemple le cas pour les processeurs CELL d'IBM [61, 62] ou CT3400 de Cradle Technologie [63]. L'efficacité transistor du processeur CELL est de 0,68 GOPs/MT et l'efficacité énergétique supérieure à 2 MOPs/mW. Ainsi, cette solution atteint une bonne efficacité transistor sans trop pénaliser l'efficacité énergétique. Mais cette dernière reste insuffisante. Avec ses 80 W en technologie CMOS 90 nm, elle ne peut pas convenir à nos attentes. Cette architecture n'a pas été conçue pour consommer peu d'énergie puisqu'elle se destine principalement aux supercalculateurs, ou aux consoles de jeux vidéos.

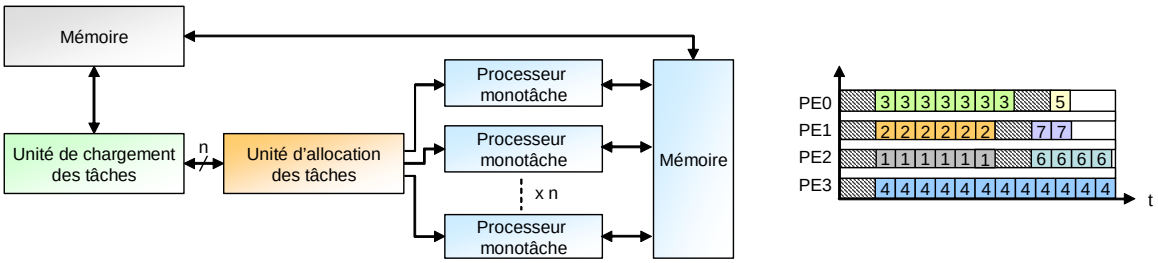

FIG. 1.8 – Architecture multiprocesseur asymétrique constituée de multiples processeurs monotâches. Soit un ensemble de tâches indépendantes $\{T1, T2, T3, T4, T5, T6, T7\}$, l'allocation des tâches est effectuée en-ligne par une unité de contrôle spécialisée. Si les processeurs élémentaires sont homogènes, les tâches peuvent s'exécuter sur n'importe quelle ressource de calcul.

Hétérogènes A l'inverse, les multiprocesseurs asymétriques hétérogènes sont des solutions optimisées pour les systèmes embarqués mobiles [64]. Les processeurs OMAP de Texas Instrument [65], Nomadik de ST Microelectronics [66], Viper de Philips Semiconductors [67], HiBRID-SoC [68] ou Prophid [69], disposent toutes de processeurs de calcul optimisés et hétérogènes. Ceci leur permet de répondre à un besoin applicatif précis avec une ressource de calcul optimisée. Les performances sont alors améliorées ainsi que l'efficacité énergétique. L'allocation des tâches sur ces ressources est souvent malheureusement trop statique. De même, l'absence d'homogénéité empêche des algorithmes d'allocation de répartir efficacement la charge de calcul entre plusieurs ressources. Des travaux sont actuellement en cours et un modèle d'exécution appelé MLCA (Multilevel Computing Architecture) permet d'apporter plus de souplesse lors de l'allocation des tâches en transformant des techniques propres aux architectures superscalaires [70]. Par ailleurs, les communications entre des éléments de nature différente sont difficiles à mettre en œuvre, même si l'utilisation d'un contrôleur centralisé réduit ce besoin. Pour une meilleure occupation des ressources de calcul, il peut être intéressant de disposer de moyens de communication entre les processeurs de calcul [71].

L'approche CMP minimise la taille finale de l'architecture en préférant augmenter le nombre de ressources plutôt que le nombre d'instructions exécutées par une même unité d'exécution. De plus, le fait d'utiliser un processeur de calcul simple améliore l'efficacité de l'architecture. En effet, pour une surface et une densité d'intégration identiques, la complexité d'un multiprocesseur composé de 16 processeurs monotâches n'est pas supérieure à un processeur SMT pouvant exécuter 12 instructions en parallèle. Avec ce potentiel de parallélisme supplémentaire, le CMP obtient de meilleures performances [29, 31]. En effet, les processeurs multitâches sont limités en nombre d'instructions à chaque cycle d'exécution par la bande passante de la mémoire d'instruction. De plus, la complexité de l'interconnexion entre les différents bancs mémoires induit une latence supplémentaire qui existe à chaque accès. Ainsi, même pour décoder de multiples instructions d'une même tâche ou d'un même banc mémoire, le traitement souffre toujours d'un surcoût qui n'existe pas dans les architectures CMP. En effet, les mémoires sont simples et les temps d'accès ne souffrent d'aucune pénalité. Sans compter que des études ont montré que les modèles CMP consomment moins d'énergie que les processeurs multitâches [72, 73].

Néanmoins, il est vrai que les CMP souffrent d'une mauvaise occupation des ressources de calcul si le parallélisme de tâches est inférieur au nombre de processeurs. Au contraire, les processeurs SMT apportent une solution efficace en partitionnant dynamiquement les tâches sur les unités d'exécution. Ainsi, les processeurs multitâches s'adaptent mieux aux

besoins applicatifs. De plus, la latence engendrée par le contrôle et le réseau d'interconnexion pénalise la réactivité de l'architecture CMP et empêche l'optimisation des ressources de calcul. Enfin, pour exploiter correctement ce modèle d'architecture, le parallélisme de tâches doit être explicite. Ceci nécessite une programmation particulière des applications. Ainsi, même si ces structures sont très bien adaptées aux contraintes de l'embarqué, elles peuvent encore être améliorées pour mieux répondre aux besoins des futures applications mobiles.

Même si les architectures CMP sont les modèles les plus disposés à répondre à nos contraintes, elles manquent de souplesse et nécessitent de nombreuses améliorations. Pour tenter d'apporter plus de flexibilité et notamment permettre la préemption et la migration des tâches, une dernière solution consiste à combiner la souplesse des processeurs multitâches et la simplicité des structures asymétriques.

1.6.2 Avec des processeurs multitâches

Le dernier modèle d'architecture que nous allons étudier propose un compromis entre les solutions SMT et CMP. Il est appelé Chip Multithreading (CMT) [74]. Il intègre plusieurs processeurs SMT de complexité réduite autour d'un modèle SMP (figure 1.9). Le processeur UltraSparc T1 (Niagara) de Sun Microsystems [75], ainsi que le RMI XLR de la société Raze Microelectronics [76] utilisent ce modèle d'exécution.

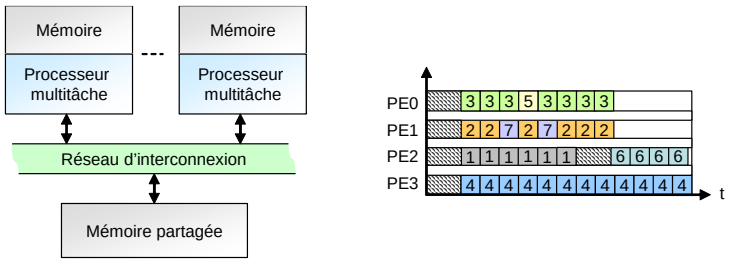

FIG. 1.9 – Architecture multiprocesseur constituée de multiples processeurs multitâches. Soit un ensemble de tâches indépendantes $\{T1, T2, T3, T4, T5, T6, T7\}$, chaque processeur élémentaire peut exécuter plusieurs tâches simultanément. L'attribution des tâches est considérée comme définitive.

L'avantage de ces solutions réside dans une meilleure occupation des ressources de calcul. Il maintient de bonnes performances avec peu de tâches en parallèle. Comme nous l'avons vu avec les architectures SMT, ce modèle d'exécution résout en partie les problèmes liés aux limitations du parallélisme d'instructions. En effet, il supporte l'exécution simultanée de plusieurs tâches au sein même des processeurs distribués. Par conséquent, il améliore les temps d'exécution dans chacun des processeurs multitâches. Par ailleurs, il a l'avantage de posséder des mécanismes de changement de tâches entraînant peu ou aucune pénalité. Au sein même d'un processeur, la préemption est donc facilitée.

Par contre, dès que des synchronisations ou des migrations de tâches doivent se faire entre des processeurs distants, les mêmes inconvénients que ceux rencontrés dans les architectures SMP surviennent. Par ailleurs, selon Sun Microsystems, la complexité nécessaire pour transformer un processeur monotâche en un processeur multitâche capable d'exécuter 4 tâches simultanément, est de l'ordre de 25 % [77]. Même si les performances sont améliorées d'environ 2,1 fois, l'efficacité transistor du processeur UltraSparc T1 est de 0,08 GOPs/MT pour

une efficacité énergétique supérieure à 0,33 MOPs/mW. Ceci est bien en dessous des caractéristiques du processeur CELL d'IBM par exemple. Avec 70 millions de transistors en moins, le processeur CELL atteint plus de 600 % de performance en plus.

Le modèle CMT propose une solution intermédiaire aux modèles d'architecture étudiés jusqu'alors. Malheureusement, cette solution s'éloigne des contraintes imposées par les systèmes temps-réel. Elle garde finalement les inconvénients des diverses solutions dont elle est issue. Néanmoins, cette architecture peut se révéler performante pour les machines parallèles ou les serveurs. Mais qu'en serait-il d'un multiprocesseur de type CMP avec de multiples ressources de calcul multitâches ? La conclusion aurait été sensiblement identique, malgré la séparation des processeurs de contrôle et de calcul. Ceci est très fortement lié à la non-utilisation de processeurs de calcul optimisés. Pourtant, la solution CMP a des caractéristiques intéressantes pour les systèmes embarqués. Sans doute faut-il tenter d'apporter une solution à ses inconvénients en conservant une efficacité transistor et énergétique importante.

1.7 Synthèse

Dans la première section de ce chapitre, nous avons défini la notion de parallélisme et montré en quoi le parallélisme d'instructions est limité. Par conséquent, pour répondre aux besoins des futures applications mobiles, nous nous sommes intéressés à une autre forme de parallélisme : le parallélisme de tâches. Ensuite, le domaine de conception regroupant l'ensemble des solutions matérielles existantes a été exploré. Le tableau 1.1 résume les différentes solutions étudiées.

Architectures parallèles	Type de parallélisme	Efficacité transistor (GOPs/MT)	Efficacité énergétique (MOPS/mW)	Puissance consommée (W)
Processeurs monotâches				
MIPS 24KE [8]	pipeline	0,6	3,45	0,365
ARM 11 [13]	pipeline	0,24	1,5	0,44
Intel Pentium [17]	superscalaire	0,18	0,02	14
Philips Trimedia [28]	EPIC/VLIW	0,96	0,88	6
Processeurs multitâches				
Cray MTA [36]	exécution successive	0,62	0,02	≈ 64
Sun MAJC-5200 [40]	exécution bloquée	0,46	0,29	≈ 15
Compaq Alpha 21464 [45]	exécution simultanée	0,064	0,064	250
Multiprocesseurs				
IBM Power4 [54]	SMP	0,05	0,12	115
IBM CELL [62]	CMP	0,68	2	≈ 80
Sun UltraSparc T1 [75]	SMT	0,08	0,33	≈ 72

TAB. 1.1 – Récapitulatif des architectures parallèles en fonction de l'efficacité transistor et énergétique. Les valeurs présentées se basent sur les performances crêtes des architectures et sont normalisées pour une technologie CMOS 130 nm.

L'étude de chacune de ces solutions n'a finalement pas permis de révéler une solution répondant à nos besoins particuliers. Une solution intéressante pour les systèmes embarqués a une efficacité transistor supérieure à 0,6 GOPs/MT et une efficacité énergétique de 20 MOPs/mW. Néanmoins, les architectures multiprocesseurs asymétriques ont montré des propriétés intéressantes. Ce sont du moins les seules à atteindre une efficacité transistor et énergétique

acceptable. Par conséquent, nous retiendrons une approche mettant en œuvre un contrôle séparé et des ressources de calcul simples et optimisées. Reste que ces solutions souffrent de nombreux inconvénients :

- l'occupation des ressources de calcul est limitée ;
- la réactivité due au contrôle distant et au réseau d'interconnexion est faible ;
- la préemption et la migration sont difficiles et fortement pénalisantes ;
- les synchronisations sont peu efficaces ;
- et la programmation de ces architectures est complexe.

Ainsi, une architecture multiprocesseur efficace pour les systèmes embarqués doit pouvoir répondre à toutes ces limitations. Il faut remarquer que ces problèmes n'existent pas ou peu dans les processeurs multitâches. Ceci est dû à la localité de l'exécution des tâches, mais aussi au contrôle complexe qui existe dans ces architectures. Ainsi, un contrôle plus intelligent et plus réactif adapté aux architectures CMP serait une réponse à tous ces désavantages. Dans ce but, le prochain chapitre présente les diverses solutions de contrôle qu'il est possible d'utiliser. Nous verrons que l'utilisation d'une solution matérielle peut se révéler être une approche efficace. Un modèle d'exécution et de programmation plus adapté sera également proposé dans le chapitre 3.

Chapitre 2

Le contrôle des architectures multiprocesseurs

Sommaire

Le chapitre précédent a mis en évidence la nécessité d'utiliser des architectures multiprocesseurs pour répondre aux besoins des futures applications mobiles. Néanmoins, les solutions proposées dans l'état de l'art ne peuvent pas répondre aux contraintes des systèmes embarqués. Le nombre de ressources de calcul n'est pas suffisant pour atteindre le niveau de performance requis. Il est important de considérer une solution de contrôle adaptée à ces architectures. Elle doit être associée à un modèle d'exécution facile à mettre en œuvre afin d'engendrer une bonne efficacité énergétique et transistor. D'autre part, le modèle de programmation doit rester compatible avec les méthodologies existantes.

Ainsi, l'association de multiples ressources de calcul sans dispositif de contrôle adapté risque de limiter les performances. Ceci est encore plus vrai lorsque le contrôle est centralisé puisqu'il doit intervenir, par exemple, pour toutes les synchronisations, les demandes d'exécution ou les accès aux données. Le contrôle des architectures multiprocesseurs asymétriques a donc un rôle fondamental sur ses performances globales. Il doit gérer de manière efficace l'exécution des tâches et l'occupation des ressources de calcul sous des contraintes de consommation et de temps-réel.

Ce second chapitre dresse un état de l'art des solutions logicielles ou matérielles pour le contrôle des architectures multiprocesseurs. Tout d'abord, il présente les solutions de contrôle logiciel et propose une classification des systèmes d'exploitation pour les architectures multiprocesseurs. Ensuite, la mise en évidence de leurs limitations introduira les solutions de contrôle matériel. Nous verrons qu'un gain significatif peut être apporté en accélérant matériellement certaines fonctions de contrôle, même si aucune solution répondant aux besoins des futures applications mobiles et des architectures multiprocesseurs n'existe dans la littérature.

2.1 Le contrôle logiciel des multiprocesseurs asymétriques

Les techniques de contrôle logicielles sont presque toujours utilisées avec les architectures multiprocesseurs. La première approche consiste à partitionner les applications en tâches indépendantes en tenant compte de la structure de l'architecture. Cette méthode est par exemple exploitée par l'outil *SynDEx* (Synchronized Distributed Execution) de l'INRIA Rocquencourt [78, 79] ou *SPEAR* (Signal Processing Environments and ARchitectures) du groupe Thales [80]. Elle repose sur la distribution statique des tâches et n'a alors pas besoin d'ordonnancement. L'avantage de cette solution est sa simplicité de mise en œuvre matérielle. Néanmoins, l'ordonnancement statique des tâches ne permet pas la création dynamique d'application et souffre d'une faible *réactivité*. La réactivité est une grandeur caractérisant le temps de prise en compte des informations provenant des ressources matérielles et d'exécution du contrôle correspondant. Ceci limite la flexibilité du système et l'empêche de s'adapter aux évènements ou aux contraintes extérieures.

Contrairement à cette approche statique, la seconde solution utilise un programme qui agit comme un intermédiaire entre l'utilisateur et le matériel. Il est appelé *système d'exploitation* (SE) [81, 82]. Son but est d'abstraire la complexité de l'architecture et de faciliter sa programmation. Il sélectionne également les tâches à exécuter, les ordonnance et les alloue sur une ressource de calcul adaptée. Avant d'étudier quelles sont les solutions employées dans les multiprocesseurs asymétriques, nous allons détailler leur fonctionnement et tenter d'explorer l'ensemble des solutions qu'il est possible d'utiliser. Ceci justifiera l'intérêt d'une approche matérielle par la suite.

2.1.1 Les systèmes d'exploitation

Les systèmes d'exploitation constituent une famille extrêmement vaste qu'il est difficile d'explorer dans sa totalité. Cette première sous-section tente malgré tout de rendre compte de leur complexité et de décrire leurs principaux mécanismes. Elle introduit l'ensemble des systèmes d'exploitation envisageables pour les multiprocesseurs qui seront présentés par la suite.

Principe et structure

Le noyau Le système d'exploitation est principalement composé d'un noyau qui assure les communications avec les périphériques et les entrées-sorties, ainsi que la gestion des tâches à exécuter et les accès aux ressources de mémorisation. Le noyau n'est pas une tâche mais un ensemble de fonctions pouvant être appelées par les différents *processus*. Un processus est un ensemble d'instructions qui possède un espace d'adressage, un ensemble de registres, un

compteur d'instructions et une pile. Un processus peut contenir plusieurs tâches qui partagent son espace d'adressage. Le rôle d'un noyau est de garantir l'ordonnancement de ces processus et d'allouer les ressources nécessaires à leur exécution. Par ailleurs, le noyau permet d'abstraire le traitement de processus simultanés.

La structure Comme le montre la figure 2.1, le noyau est structuré en couches. Ainsi, l'envoi d'une information par la couche applicative nécessite de traverser toutes les couches logicielles qui composent le noyau. Ceci induit une latence importante lors de la prise en compte d'informations en provenance des ressources matérielles. Cependant, l'avantage de ces couches est qu'elles procurent un niveau d'abstraction qui simplifie la programmation et qui réduit les temps de développement. Ainsi, la programmation d'une application ne nécessite pas de connaître l'architecture, mais seulement l'ensemble des fonctions de haut niveau qui composent l'interface d'abstraction logicielle ou *Application Programming Interface* (API). Seule la dernière couche logicielle est dépendante de l'architecture. Elle est encore appelée *Hardware Abstraction Layer* (HAL).

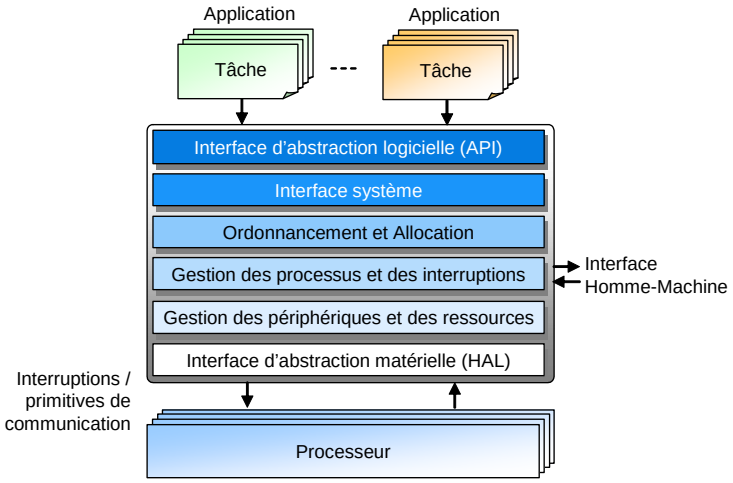

FIG. 2.1 – Structure en couche d'un système d'exploitation

L'ensemble des noyaux peut être divisé en deux catégories : les noyaux *monolithiques* et les *micro-noyaux*. Un noyau monolithique regroupe l'ensemble des fonctions du système et des logiciels nécessaires à l'utilisation des ressources matérielles. Une approche moins rigide appelée *noyau monolithique modulaire* regroupe seulement les fonctions principales et dispose de modules indépendants. Ceci offre plus de stabilité et adapte la taille du noyau au besoin du système. Cependant, ces deux solutions manquent de robustesse et de fiabilité. Plus la taille du noyau, ou le nombre de couches, est important et moins le système d'exploitation est réactif.

Au contraire, les micro-noyaux minimisent les fonctionnalités dépendantes du noyau. La plus grande partie des services du système d'exploitation est placée à l'extérieur. Ceci induit des communications importantes entre les différents services extérieurs au noyau (IPC : Inter-Process Communication). Malgré tout, ils sont largement utilisés dans les systèmes embarqués qui demandent une grande réactivité. Ces micro-noyaux sont le plus souvent combinés à des services de gestion mémoire et des entrées-sorties. Ils sont en outre capables de respecter des contraintes de temps d'exécution : ils sont appelés *exécutifs temps-réel*.

Les tâches Le noyau maintient une liste de tâches à exécuter. Pratiquement, chacune de ces tâches peut être dans l'état suivant : *en cours d'exécution*, *prête*, *suspendue*, *dormante* ou *bloquée* (figure 2.2). Quand le noyau est invoqué, il vérifie les tâches prêtes et si la prochaine tâche est éligible. Dans ce cas, la tâche prête passe dans la liste de tâches *en cours d'exécution*. Par ailleurs, le noyau contrôle l'état des ressources et met à jour les listes de tâches *suspendues* et *bloquées*. Les tâches bloquées en attente d'une ressource peuvent alors être prêtes à être exécutées.

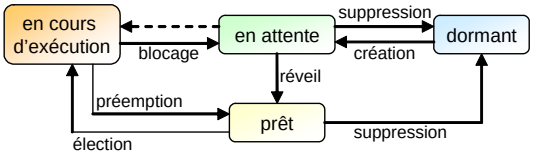

FIG. 2.2 – Diagramme d'état des tâches temps-réel

Ces micro-noyaux ou exécutifs temps-réel peuvent être monotâches ou multitâches. Cependant, seuls les noyaux multitâches conviennent à notre étude puisque nous nous intéressons uniquement à la gestion de multiples ressources de calcul. En effet, dans cette étude un noyau monotâche comme Palm-OS [83] n'a aucun intérêt puisqu'il ne peut pas demander l'exécution de plusieurs tâches en parallèle.

Le multitâche peut être *coopératif* lorsque chaque tâche permet explicitement aux tâches suivantes de s'exécuter. Cette approche comporte plusieurs inconvénients : le système peut s'arrêter si une tâche est bloquée et le partage des ressources est inefficace. Pour remédier à cette situation, les noyaux ont évolué vers une approche nommée *multitâche préemptif*. Une tâche en cours d'exécution peut être interrompue par une autre tâche. Cette solution apporte de nombreux avantages : elle rend plus stable le système et les ressources disponibles sont utilisées plus efficacement.

Cette première analyse des systèmes d'exploitation met en évidence la complexité de leur structure. Ces solutions logicielles structurées en couches augmentent la latence de prise en compte des interruptions ou des informations provenant des ressources matérielles. Le choix de cette structure est essentiellement motivé par la simplification de la programmation et du développement d'applications. Des aménagements ont été apportés pour les systèmes embarqués, mais la répartition du noyau en services indépendants engendre d'importantes communications internes.

Classification des systèmes d'exploitation pour les multiprocesseurs

L'étude du fonctionnement ou de la structure du système d'exploitation n'est pas suffisante pour envisager leur utilisation dans les architectures multiprocesseurs. Il faut également étudier comment le système d'exploitation peut être exécuté par les multiples ressources de calcul disponibles. En fait, ceci peut se faire de deux façons. Son exécution peut être répartie sur toutes les ressources de calcul : ce sont les *systèmes d'exploitation distribués*. Dans le cas contraire, ils seront nommés *systèmes d'exploitation parallèles* (figure 2.3) [84].

Les systèmes d'exploitation distribués reposent sur la répartition des services offerts par le système d'exploitation [85, 86]. Les systèmes UNIX [87], Amoeba [88] ou CHORUS [89] sont des exemples de SE distribués. Chacun des processeurs possède un SE simplifié qui

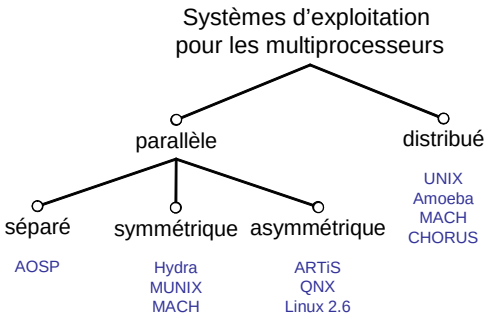

FIG. 2.3 – Espace de conception des systèmes d'exploitation pour les multiprocesseurs

communique avec un serveur pour accéder à des fonctionnalités supplémentaires. Ce noyau possède quatre services de base : il gère l'exécution des processus et des tâches, fournit une gestion de bas niveau de la mémoire, assure les communications et contrôle les entrées-sorties. L'exécution parallèle des tâches est basée sur l'utilisation d'un processus commun de communication, capable de créer des tâches sur un processeur distant. Les systèmes distribués sont très complexes à mettre en œuvre et engendrent des coûts de développement importants. Ceci est d'autant plus vrai pour un ensemble de processeurs hétérogènes. Par ailleurs, le contrôle étant distribué, ces SE se destinent essentiellement aux architectures multiprocesseurs symétriques. Par conséquent, nous ne nous y intéresserons pas davantage.

Dans les systèmes d'exploitation parallèles, les services ne sont pas distribués et le système d'exploitation est identique sur chacun des processeurs [90]. Les SE parallèles peuvent être soit *séparés*, *symétriques* ou *asymétriques* (figure 2.4). Dans les SE parallèles séparés, comme dans AOSP [91], chaque processeur possède une copie privée du système d'exploitation et accède à un espace mémoire privilégié. Les processus ne peuvent pas migrer d'un processeur à l'autre. Ceci empêche de répartir convenablement la charge de calcul. Enfin, les données partagées entre tous les processeurs nécessitent un contrôle permanent de leur cohérence et de leur intégrité. Au contraire, les SE symétriques peuvent être exécutés sur n'importe quel processeur. Une copie unique du système d'exploitation est accessible dans une mémoire partagée. Chaque processeur n'exécutant pas de tâche peut alors supporter l'exécution du système d'exploitation. Pour éviter des problèmes d'intégrité, un mécanisme de protection par verrou doit être utilisé lors de chacun des accès à la mémoire partagée. Les systèmes d'exploitation Hydra [92], MUNIX [93] et MACH [94] sont quelques exemples de SE symétriques.

Ces solutions ne sont pas intéressantes dans notre cas d'étude, car chacune d'elle repose encore sur un contrôle distribué qui nécessite des mécanismes de synchronisation complexes. Même si les machines parallèles n'en souffrent pas trop grâce à leur forte puissance de calcul, ces solutions ne sont pas envisageables pour un multiprocesseur embarqué. La dernière solution simplifie la plupart de ces problèmes de cohérence. Elle se rapproche davantage de nos besoins en terme de contrôle. Les systèmes d'exploitation asymétriques ou *maître-esclave* sont exécutés par une unique ressource de calcul. A titre d'exemple, il existe les systèmes d'exploitation ARTiS [95], QNX [96] ou le noyau Linux 2.6. Dans ce modèle de contrôle, le processeur *maître* est responsable de l'ordonnancement et de la distribution des tâches sur les processeurs *esclaves*. Lorsqu'un processeur esclave a terminé l'exécution d'une tâche ou d'un processus, il le signale au SE et se met en attente d'une nouvelle tâche. Ainsi, il n'est plus possible qu'un processeur soit surchargé alors que d'autres sont au repos. Néanmoins, cette

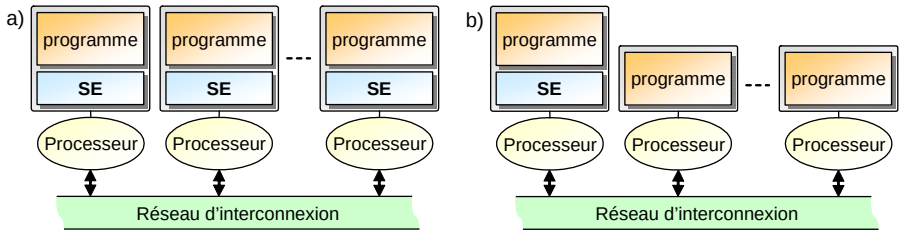

FIG. 2.4 – Les systèmes d'exploitation parallèles. Ils sont dits *séparés* lorsque chacune des ressources de calcul possède une copie du système d'exploitation (a), *symétriques* lorsqu'une seule copie peut être exécutée indépendamment sur une des ressources disponibles et enfin *asymétriques* si un unique processeur est attribué spécifiquement au contrôle (b).

solution souffre d'un problème de réactivité du contrôle dû à la latence des communications et à la sérialisation des traitements sur des requêtes parallèles.

La solution asymétrique est certainement la plus proche de la solution logicielle optimale pour les multiprocesseurs asymétriques. Cependant, elle ne résout en rien les défauts liés à l'asymétrie de la structure qui repose principalement sur un contrôle réactif. Maintenant que nous savons ce qu'est un système d'exploitation et comment l'exécuter sur nos multiples processeurs, il nous faut nous intéresser aux systèmes d'exploitation qui répondent plus spécifiquement aux besoins des systèmes embarqués.

2.1.2 Les exécutifs temps-réel

Les exécutifs temps-réel se différencient des autres noyaux par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Autrement dit, le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés [97]. La plupart des systèmes embarqués nécessitent un noyau pouvant respecter des contraintes sur la durée d'exécution des tâches. Par exemple, un algorithme de décompression d'images numériques doit respecter une fréquence d'affichage correcte.

En pratique, on distingue le temps-réel *strict* ou *souple* suivant l'importance accordée aux contraintes temporelles. Le temps-réel strict ne tolère aucun dépassement de ces contraintes pour des questions de sécurité critique. Le concepteur doit être capable, le plus souvent, de prouver que les limites temporelles ne seront jamais franchies quelle que soit la situation. Pour cela, des critères d'ordonnançabilité sont étudiés ou des analyses hors lignes des comportements sont effectuées. A l'inverse, le temps-réel souple tolère des débordements exceptionnels qui seront rattrapés à l'exécution suivante. Une mesure statistique sur un prototype peut se révéler suffisante.

Les systèmes d'exploitation temps-réel pour des architectures multiprocesseurs sont essentiellement distribués (Chimera [98], MARS [99]), symétriques (RT-Mach [100], Montavista Linux [101]) ou séparés (OSE [102], VxWorks [103]). Aucune des solutions existantes ne propose une structure asymétrique comme nous le préconisons [104]. Ceci a plusieurs raisons : les structures multiprocesseurs asymétriques sont présentes sur le marché depuis peu de temps et les algorithmes nécessaires à la mise en œuvre du contrôle restent trop complexes, ou inefficaces, pour être exécutés sur des processeurs dédiés au calcul. Ainsi, les solutions adoptées

étendent toutes les fonctionnalités utilisées par les monoprocesseurs en intégrant des mécanismes de communication particuliers. Nous allons malgré tout décrire leur fonctionnement car les propositions que nous ferons par la suite s'appuient sur ces premiers principes.

Définitions et notions générales

Le premier rôle d'un exécutif temps-réel est d'ordonnancer l'exécution des tâches et de protéger l'accès aux ressources partagées. En effet, les mécanismes qui décident de l'ordre d'exécution, ou de l'attribution des ressources, influent considérablement sur la capacité du système à respecter ses contraintes de temps. La classification des ordonnancements temps-réel se révèle être extrêment vaste [105, 106]. Chacune des solutions étudiées dans la littérature répond à un besoin précis en terme de complexité ou de sûreté de fonctionnement. Dans un premier temps, il nous faut définir quelles sont les solutions les mieux adaptées aux futures applications mobiles.

En ligne ou hors ligne Un algorithme d'ordonnancement *hors ligne* planifie l'exécution des tâches en fonction de tous ses paramètres temporels. Cette séquence ou ce *scénario* est connu avant l'exécution des tâches et peut être mis en œuvre très efficacement. Néanmoins, cette solution est trop rigide au vu de nos besoins applicatifs. A l'inverse, un algorithme d'ordonnancement *en ligne* choisit, au cours de l'exécution, la prochaine tâche à exécuter. Il s'adapte aux évènements extérieurs et aux besoins de l'utilisateur. Cette approche dynamique donne des solutions moins bonnes puisqu'on utilise moins d'informations pour prendre les décisions. Par ailleurs, elle est plus complexe à mettre en œuvre et induit un surcoût temporel plus important. Elle est cependant indispensable dans notre cas d'étude.

Préemptif ou non-préemptif Dans le cadre des systèmes temps-réel, la préemption est la seule solution qui permette d'interrompre une tâche moins prioritaire. Elle a donc un rôle très important pour les ordonnancements en ligne si le comportement du système n'est pas prédictible. Ceci est notre cas puisque l'utilisateur peut demander ou interrompre l'exécution d'une tâche à n'importe quel moment.

ET ou TT L'ordonnancement peut être cadencé par les évènements (*ET : Event-Triggered*) ou par l'écoulement du temps (*TT : Time-Triggered*) [107]. Les systèmes temps-réel ont communément une architecture cadencée par les évènements, donc l'ordonnancement ET est souvent préféré. Pourtant, le fait d'ordonnancer des tâches à des intervalles de temps réguliers permet de mieux prédire le comportement du système. Il devient plus déterministe et plus aisé à valider. En contrepartie, il engendre une sous-occupation des ressources de calcul (figure 2.5).

Niveaux de Priorité ou priorité temporelle La priorité des tâches est définie de deux manières. La première consiste à attribuer un *niveau de priorité* à chacune des tâches. La tâche ayant le niveau le plus élevé est la tâche en cours d'exécution. L'avantage est de pouvoir exécuter une tâche plus prioritaire quel que soit l'état des autres tâches moins prioritaires. La simplicité de cette solution l'a rendu très populaire dans les noyaux temps-réel commerciaux [108]. A titre d'exemple, μCOS-II est un noyau préemptif capable de gérer 64 tâches possédant un niveau de priorité unique [109]. Néanmoins, ces systèmes ne garantissent pas le respect des contraintes de fin d'exécution qui sont essentielles pour respecter les cadences d'affichage, la périodicité des évènements, ou la fréquence des flux de données ou de contrôles. A moins d'une

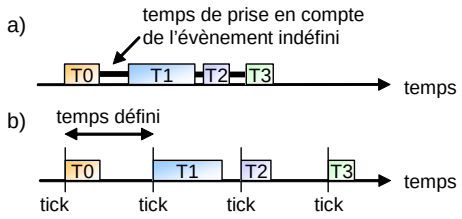

FIG. 2.5 – Ordonnancement ET ou TT. L'ordonnancement ET exécute successivement les tâches en fonction d'évènements de fin d'exécution (a). La durée d'ordonnancement ou de prise en compte des interruptions est difficilement prédictible. Au contraire, l'ordonnancement TT prend en compte les évènements à chaque nouveau *tick* d'ordonnancement (b). L'échelle du temps est mieux maîtrisée.

analyse hors ligne, ces ordonnancements n'assurent pas le respect des contraintes temps-réel. Au contraire, les ordonnancements à *priorité temporelle* prennent leur décision en fonction de la périodicité ou de l'*échéance* des tâches et cela au cours de l'exécution. L'échéance d'une tâche est la date de sa fin d'exécution au plus tard permettant de respecter sa contrainte temps-réel.

Priorités statiques ou dynamiques Enfin, l'ordonnancement des tâches fonde ses décisions sur des paramètres assignés aux tâches avant leur activation. Ces algorithmes *statiques* ont l'avantage de minimiser le surcoût de l'ordonnancement, mais ne permettent toujours pas de s'adapter à des conditions d'exécution imprévues ou dont les dates d'arrivée sont variables. Par ailleurs, ils peuvent se révéler parfois moins performants que les approches *dynamiques*. Cette dernière solution supporte le changement des priorités au cours de l'exécution. Elle permet de rendre compte de l'urgence de l'exécution d'une tâche en fonction de son échéance. Par exemple, une tâche proche de son échéance pourra avoir une priorité différente de celle qu'elle aurait eu si elle disposait d'un temps important pour s'exécuter.

En conclusion, nous devons privilégier une approche en ligne et préemptive. De même, les priorités doivent être évaluées dynamiquement et dépendre pour chacune des tâches de son échéance. Le choix d'un ordonnancement ET ou TT est plus délicat et sera davantage étudié par la suite. Ainsi, les contraintes imposées par les futurs besoins applicatifs nécessitent la mise en œuvre d'un algorithme capable d'adapter son comportement au cours de l'exécution. Les besoins applicatifs varient de manière indéterminée en fonction des envies de l'utilisateur. Par conséquent, seuls les algorithmes en lignes et préemptifs sont adaptés à nos exigences. Pour la même raison, les priorités des tâches doivent pouvoir évoluer au cours de l'exécution en fonction de contraintes temporelles.

Les ordonnancements préemptifs à priorité temporelle dynamique

Les deux algorithmes les plus utilisés respectant ces contraintes d'ordonnancement sont le *Earliest Deadline First* (EDF) et le *Least Laxity First* (LLF). L'algorithme d'ordonnancement EDF consiste à assigner la priorité la plus élevée à la tâche dont l'échéance relative courante est la plus proche [110, 111]. L'évaluation des priorités des tâches est effectuée à chaque nouvel ordonnancement. Celui-ci a lieu à la fin de l'exécution d'une tâche, lors de la libération d'une ressource partagée, ou à l'activation d'une nouvelle tâche à ordonnancer. Cet algorithme est optimal dans la classe des algorithmes à priorités dynamiques pour des configurations de tâches *périodiques indépendantes* avec des échéances inférieures ou égales à leur période. Ainsi, si un ordonnancement existe pour un ensemble de tâches, l'ordonnancement EDF est

possible. De la même façon, si un ordonnancement EDF n'est pas admissible, aucun autre ordonnancement n'est envisageable [112].

L'algorithme LLF consiste, quant à lui, à affecter la priorité des tâches en fonction de leur *laxité* [110]. Comme le montre la figure 2.6, la laxité est la durée restante à la tâche pour s'exécuter. A un instant donné, la tâche dont la laxité est la plus petite est la tâche la plus prioritaire. Cet algorithme est également optimal, mais induit un nombre plus important de changements de contexte. Par ailleurs, il engendre un surcoût élevé puisque les priorités doivent être évaluées à chaque mise à jour du temps courant. Ce modèle d'ordonnancement TT occasionne également une sous-exploitation des ressources de calcul. Pour toutes ces raisons, il n'a jamais été implémenté dans un noyau temps-réel pour les systèmes embarqués.

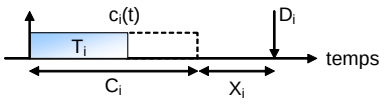

FIG. 2.6 – Définition de la laxité. Soient C_i la durée d'exécution maximale sans préemption (WCET : Worst Case Execution Time) de la tâche T_i, D_i son échéance relative et c_i le temps déjà passé à exécuter cette tâche. Comme l'illustre la figure, la laxité X_i est égale à $X_i(t) = D_i - (t + C_i - c_i(t))$.

Chacun de ces algorithmes d'ordonnancement peut prendre en compte des tâches *apériodiques* au cours de l'exécution. Une tâche apériodique est une tâche réveillée par un évènement dont la date d'occurence est inconnue avant l'exécution. Le but est d'assurer le respect des contraintes temporelles des tâches dont l'ordonnancement était prévu, tout en acceptant l'exécution de tâches supplémentaires. Il s'agit alors de minimiser le temps de réponse (*contrainte relative*) ou de maximiser le nombre de tâches qui peuvent être acceptées (*contrainte stricte*) [113, 114]. L'ordonnancement de tâches apériodiques à contraintes relatives peut avoir lieu lorsque le processeur est au repos (*traitement d'arrière plan*), ou lors de l'activation d'une tâche périodique particulière appelée *serveur de tâches* [111, 115, 116, 117]. A l'inverse, l'ordonnancement de tâches apériodiques à contraintes strictes consiste à attribuer aux tâches apériodiques une pseudo-période. Celle-ci peut être équivalente au temps minimal entre deux occurences successives, ou être définie hors ligne de manière à utiliser les temps de repos du processeur [118].

Les tâches de l'application peuvent être indépendantes ou non les unes des autres. Une tâche est dépendante si elle possède une relation de précédence qui conditionne son exécution, ou si elle doit accéder à une ressource partagée. Dans le premier cas, les tâches sont transformées en tâches indépendantes en définissant leurs échéances relatives en fonction de leurs contraintes de précédence [119, 120]. Si la tâche T_j précède la tâche T_i alors l'échéance de T_j est inférieure à celle de la tâche T_i. Dans le second cas, des algorithmes supplémentaires doivent être utilisés pour interdire les accès simultanés aux ressources et pour éviter les interblocages (figure 2.7). Par exemple, lorsqu'une tâche plus prioritaire souhaite accéder à une ressource utilisée par une tâche moins prioritaire, le protocole de la priorité héritée (PIP : Priority Inversion Protocol) inverse les priorités [121]. La tâche utilisant alors la ressource devient plus prioritaire. Au contraire, le protocole de la priorité plafonnée (PCP : Priority Ceiling Protocol) autorise l'accès à une ressource partagée seulement si la priorité de la tâche est strictement supérieure aux priorités des tâches qui pourraient utiliser la ressource [121]. Enfin, le protocole de la priorité de pile (SRP : Stack Resource Policy) autorise une tâche à démarrer son exécution seulement si toutes les ressources nécessaires sont disponibles [122].

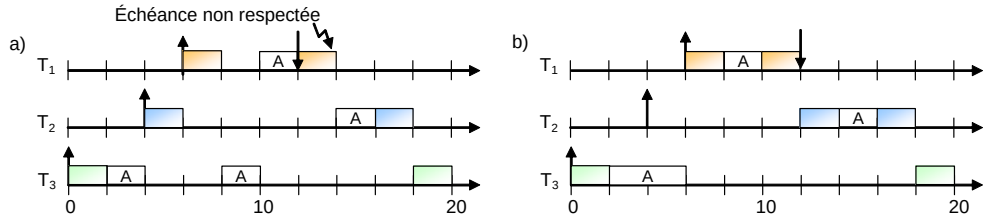

FIG. 2.7 – Ordonnancement EDF avec le protocole PIP (a) et avec les protocoles PCP ou SRP (b). La ressource partagée est notée *A*. L'arrivée d'une tâche est représentée par une flêche montante, son échéance par une flêche descendante.

En ce qui concerne l'ordonnancement en ligne multiprocesseur, les algorithmes d'ordonnancement vus précédemment s'appliquent en leur associant un algorithme d'allocation particulier. L'allocation choisit par exemple le meilleur processeur disponible pour exécuter la tâche et respecter ses contraintes. Si aucune ressource de calcul n'est disponible, la tâche est assignée à la ressource exécutant la tâche la moins prioritaire. En pratique, aucun de ces algorithmes n'a été mis en œuvre pour des raisons de complexité. En fait, seuls les ordonnancements effectués hors ligne permettent d'obtenir un ordonnancement optimal [123, 124, 120]. Un résultat fondamental, présenté par M. Sahni, affirme qu'il n'existe pas d'algorithme en ligne optimal pour un système comportant plus de deux processeurs [125]. Pire, comme nous le verrons dans le chapitre 4, des anomalies d'ordonnancement peuvent avoir lieu. Mais alors, quelles sont les solutions pour respecter des contraintes temps-réel ?

2.1.3 Exemples de techniques employées dans les architectures CMP

En pratique, on recherche avant tout la simplicité et on s'accomode d'un contrôle apportant peu de flexibilité. Cependant, deux solutions s'opposent : les approches multiprocesseurs hétérogènes et homogènes.

Les multiprocesseurs asymétriques hétérogènes

Les solutions de contrôle pour les multiprocesseurs hétérogènes favorisent toutes un noyau offrant une interface graphique conviviale. Ces noyaux sont appelés *systèmes d'exploitation mobiles*. Les plus évolués proposent des mécanismes permettant d'exécuter plusieurs tâches sur une seule ressource de calcul et de gérer de multiples protocoles de communication. A titre d'exemple, citons les systèmes d'exploitation Symbian [126], SavaJe [127], Palm OS [128] et Microsoft Windows CE [129].

Le système d'exploitation Symbian est utilisé par la plupart des fabricants de téléphones portables, tandis que Palm OS détient la part de marché la plus importante pour les assistants personnels. Ce dernier est un noyau multitâche coopératif intégrant des fonctionnalités bureautiques particulières comme la reconnaissance d'écriture manuscrite. Ce noyau n'est donc pas temps-réel puisqu'il n'y a pas d'ordonnancement à effectuer. Quant-à Windows CE, il supporte un ordonnancement suivant huit niveaux de priorité. Les 32 processus qui peuvent être actifs simultanément sont ordonnancés selon un algorithme appelé *tourniquet* ou *round-robin*. L'algorithme répartit équitablement les temps d'exécution des tâches de même priorité. Par conséquent, ces systèmes d'exploitation ne sont pas capables de respecter dynamiquement des contraintes temporelles. De plus, ils ne peuvent pas exécuter en parallèle des tâches sur

de multiples ressources de calcul. Celles-ci sont utilisées séquentiellement sans profiter du parallélisme de l'architecture.

Comme le montre la figure 2.8, l'occupation des ressources de calcul est très faible contrairement à ce que l'on aurait pu espérer atteindre avec un contrôle plus adapté. En effet, le taux d'occupation passe de 91 à 33% et la durée totale d'exécution est multipliée par deux. Pourtant, l'efficacité et la puissance théorique sont identiques. L'intérêt d'un contrôle plus efficace est ici largement démontré ; son absence pénalise grandement les performances de ces architectures.

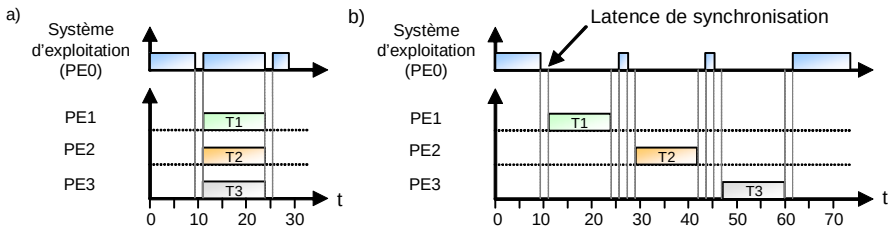

FIG. 2.8 – Ordonnancement de tâches dans les architectures CMP hétérogènes. Pour l'exemple, nous considérons une application composée d'une tâche principale devant exécuter des tâches T_1, T_2 et T_3 sur des ressources de calcul distantes. Les temps de synchronisation et les temps des tâches ont été choisis arbitrairement. Le premier résultat correspond à un ordonnancement théorique avec un contrôle multitâche adapté (a) et le second à l'ordonnancement privilégié dans les architectures CMP hétérogènes (b).

Les multiprocesseurs asymétriques homogènes

Dans les multiprocesseurs asymétriques homogènes, des solutions plus adaptées sont proposées. C'est le cas par exemple dans le processeur CELL de la société IBM [61, 62]. Le procédé employé pour exploiter ses 8 processeurs de calcul secondaires consiste à utiliser un noyau Linux 2.6 pour le contrôle et une technique de compilation particulière appelée *Octopiler* [130, 131].

Le noyau Linux utilise un algorithme d'ordonnancement préemptif à priorités dynamiques. Les 140 niveaux de priorités sont attribués de manière dynamique en fonction de l'occupation de la tâche sur le processeur hôte, de son interactivité ou de son historique. Ensuite, les tâches sont exécutées suivant une politique de tourniquet et le temps accordé à chacune des tâches est basé sur leur priorité. Plus le niveau de priorité est important et plus la durée d'exécution allouée à la tâche est conséquente. Enfin, le système utilise un algorithme d'allocation particulier qui distribue équitablement la charge de calcul entre tous les processeurs. Pour cela, les migrations des tâches sont autorisées.

Mais le processeur CELL n'emploie pas cet algorithme d'allocation et un autre modèle d'exécution et de programmation a été privilégié. Celui-ci est basé sur un compilateur issu de l'interface de programmation *OpenMP* [132]. Tout programme étant constitué d'une partie séquentielle et d'une partie parallélisable, essentiellement formée de nids de boucles, l'idée principale repose sur la répartition du calcul des boucles sur les différents processeurs. Un programme OpenMP commence avec une seule tâche maître. Lorsque le programme rentre dans une partie parallélisable, celle-ci est répartie entre tous les processeurs disponibles. A la fin de son exécution, le programme reprend le traitement de la tâche principale. Ce modèle

est appelé *orphaning* et est basé sur le langage de programmation *Fortran* [133]. Il est assez aisé à mettre en œuvre dans une machine multiprocesseur à mémoire partagée.

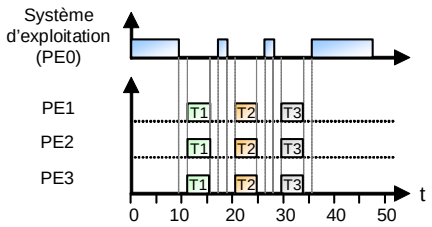

FIG. 2.9 – Ordonnancement de tâches dans les architectures CMP homogènes

La figure 2.9 reprend l'exemple de la figure 2.8 pour comparer le gain obtenu. L'occupation des ressources de calcul est de l'ordre de 53% pour une durée d'exécution juste égale à la moyenne des deux solutions présentées auparavant. Elle apporte donc un gain important par rapport aux solutions utilisées dans les architectures multiprocesseurs hétérogènes. Cependant, le taux d'occupation des ressources pourrait encore être amélioré et l'ordonnancement choisi peut difficilement respecter des contraintes temps-réel. Il serait très pénalisé en cas de préemption puisqu'il faudrait sauvegarder l'ensemble des tâches réparties sur les différents processeurs secondaires. Enfin, ce modèle n'est pas adapté pour les structures hétérogènes car la parallélisation nécessite des mécanismes complexes de compilation.

2.1.4 Limitations de l'approche logicielle

Il existe une réelle inadéquation entre les solutions de contrôle disponibles et les besoins des systèmes embarqués. Il n'existe pas de solutions temps-réel pour les multiprocesseurs répondant aux contraintes que nous nous sommes fixées. L'utilisation d'un ordonnancement à priorité statique avec un ordonnancement à partage de temps (*timesharing*) ne permet pas de s'adapter au cours de l'exécution aux besoins de l'utilisateur. Il ne garantit pas le respect des contraintes temps-réel. De plus, un système embarqué doit être en mesure d'exécuter efficacement des tâches non soumises à des contraintes de temps. Mais aucune solution combinant efficacement ces deux aspects n'a été proposée. Les choix de l'algorithme d'ordonnancement tiennent plus compte de la simplicité de la mise en œuvre que des performances. Ces solutions logicielles ne permettent pas de s'affranchir de ce compromis.

D'autre part, l'utilisation d'un système d'exploitation pour les systèmes embarqués, engendre des problèmes de disponibilité et de déterminisme. En effet, ces systèmes sont basés sur les interruptions pour dialoguer avec les ressources matérielles. Ces interruptions sont associées à des niveaux de priorité pour ordonner le traitement de plusieurs interruptions. Par conséquent, la prise en compte d'une information peut être retardée par une autre interruption plus prioritaire. Sans compter que si la charge du processeur est trop importante, l'exécution des tâches peut nécessiter un temps indéfiniment long. De plus, comme nous le verrons dans le chapitre 5, la durée de l'ordonnancement ou de certains mécanismes de contrôle dépend du nombre de tâches à traiter. Le système est donc imprévisible avant son exécution.

Ces temps d'accès au noyau ne sont pas seulement non-déterministes. Ils sont longs. En effet, chaque traitement d'une information provenant des ressources matérielles nécessite de

traverser toutes les couches logicielles qui composent le SE. Ce modèle en couche facilite la programmation, mais réduit considérablement les temps d'exécution et la réactivité du système [64]. Pire, comme le montre la figure 2.10, ces temps de traversée introduisent des *sections critiques* supplémentaires pendant lesquelles aucune information ne peut être prise en compte. Par ailleurs, le manque de parallélisme de ces solutions engendre un goulot d'étranglement lors des synchronisations.

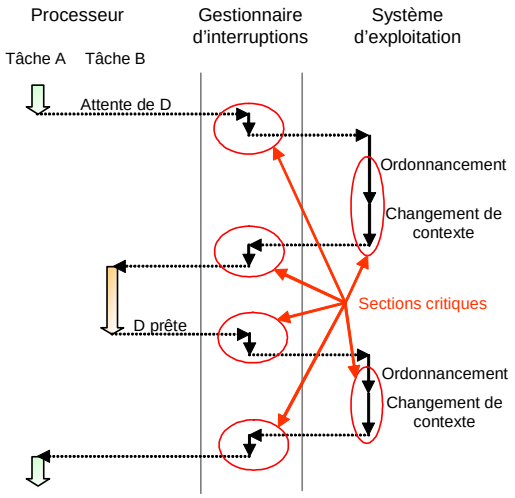

FIG. 2.10 – Limitations de l'approche logicielle. La tâche A s'exécute jusqu'à ce qu'elle ait besoin d'un résultat nommé D. Le système d'exploitation ordonnance alors la prochaine tâche plus prioritaire. Ici la tâche B produisant le résultat D est choisie et un changement de contexte préempte la tâche bloquée.

Enfin, l'utilisation d'une ressource conçue pour le calcul ne peut pas être une solution optimale pour effectuer le contrôle. Tout d'abord, elle utilise une place importante sur le silicium. Par exemple, un processeur ARM 9 occupe 3,26 mm^2 en technologie 0,13 μm. Il intègre une mémoire cache de 2 Koctets pour les instructions et les données, ce qui est tout juste suffisant pour contenir un exécutif temps-réel comme μCOS-II. Windows CE nécessite quant à lui 32 Koctets de mémoire, ce qui représente environ 1 mm^2 de surface silicium supplémentaire pour une technologie CMOS 0,13μm. Par ailleurs, l'exécution des services du système d'exploitation occupe en moyenne 22 % et jusqu'à 90 % du temps total d'exécution [134, 135]. Par conséquent, l'énergie dissipée par le SE est en moyenne de 50 mW et peut atteindre 200 mW sur une architecture ARM 9. Au vu des problèmes présentés précédement, on peut considérer que cette consommation d'énergie est peu justifiée. Une gestion des dispositifs de réduction d'énergie des ressources matérielles permettrait de mieux maîtriser la consommation d'énergie du système. Cependant, pour des architectures multiprocesseurs, aucune solution combinant le respect des contraintes temps-réel et de consommation d'énergie n'a été mise en œuvre.

Aujourd'hui, les structures multiprocesseurs semblent incontournables et les techniques d'ordonnancement ne sont pas assez abouties pour y répondre. Une vraie alternative technologique doit être apportée. Pour cela, depuis quelques années, quelques solutions de contrôle matérielles tendent à émerger.

2.2 Le contrôle matériel des multiprocesseurs asymétriques

Dans la conception de systèmes électroniques, deux mondes se complètent et s'opposent. D'un côté, les architectes sont spécialisés dans la conception de solutions matérielles et tentent de relever tous les défis qu'ils rencontrent, en proposant essentiellement des évolutions matérielles. D'un autre côté, les développeurs de logiciels choisissent d'abstraire la complexité des architectures pour concevoir des applications de plus en plus complexes. Malheureusement, ceci empêche de pouvoir pleinement exploiter les améliorations des architectures qui doivent être de plus en plus évoluées pour continuer à apporter suffisamment de performances aux applications. Ainsi, le système d'exploitation est un élément non optimisé et est bien souvent considéré comme étant une contrainte pour l'architecte, bien qu'il soit une nécessité pour le développeur.

Ce paradoxe existe depuis de nombreuses années. La situation est restée inchangée malgré les tentatives de simplification des structures des systèmes d'exploitation (micro-noyaux), la réduction de leur fonctionnalité et les améliorations des techniques de compilation. En pratique, le souci de garder une programmabilité importante sans changer le modèle de programmation reste un frein important. De ce point de vue, la plateforme de programmation *OpenMP* [132] est une avancée majeure. Les architectes ont alors tenté d'apporter des solutions supplémentaires. Celles-ci ont principalement consisté à étendre le jeu d'instructions en y intégrant des primitives de contrôle particulières, ou encore à se passer d'un contrôle logiciel en concevant des solutions dédiées (ASIC : Application Specific Integrated Circuit) [136]. Ainsi, des solutions sont apparues pour tenter d'accorder davantage les besoins applicatifs avec les architectures. Mais l'exercice est difficile car il n'est pas aisé de maîtriser l'ensemble du flot de conception d'un système embarqué.

Tout d'abord, les architectes ont accéléré quelques primitives de contrôle à l'aide d'un co-processeur ou d'un accélérateur matériel. Puis, ils ont intégré de plus en plus de fonctionnalités, jusqu'à mettre en œuvre des systèmes d'exploitation temps-réel sur silicium. Le fait de déporter une partie des services offerts normalement par le système d'exploitation, permet d'améliorer sensiblement la latence du contrôle, la prise en compte des interruptions, la précision des horloges ou le temps de disponibilité du processeur pour l'exécution des tâches.

2.2.1 Principe et état de l'art des accélérateurs de noyau temps-réel

L'accélération matérielle de quelques primitives habituellement traitées par le système d'exploitation temps-réel améliore les temps de réponse du système d'exploitation et réduit la charge du processeur hôte. L'idée principale consiste à effectuer un partitionnement logiciel/matériel du système d'exploitation en fonction des performances et de la prédictibilité des processus de gestion du système. La première solution utilise un co-processeur pour exécuter les primitives de contrôle les plus critiques.

Co-processeurs

Par exemple, le co-processeur *Ada TAsking Coprocessor* (ATAC) conçu par l'European Space Agency facilite les synchronisations et l'ordonnancement des tâches [137] : il dispose d'un ensemble de sémaphores, d'un mécanisme d'inversion de priorité et de délais, de différentes files de tâches et d'un jeu de 64 instructions de haut niveau. L'ensemble des 16 interruptions provenant du processeur est redirigé vers le co-processeur ATAC. Il ne perturbe

alors l'exécution du processeur que si une tâche plus prioritaire est en attente. Ceci permet d'améliorer les performances du processeur puisque toutes les fonctions traitées par ATAC sont exécutées de manière concurrente. L'architecture ATAC supporte 32 tâches temps-réel sur 64 niveaux de priorité et jusqu'à 2048 tâches avec une mémoire externe. Le co-processeur est intégré dans un circuit VLSI de 215 mm^2 en technologie CMOS 2 μm.

Architectures dédiées

La seconde solution repose sur des accélérateurs matériels reconfigurables (FPGA : Field Programmable Gate Array) ou dédiés (ASIC). Elle offre davantage de performance car elle s'appuie sur une structure de contrôle dédiée, qui se prête mieux au parallélisme de contrôle. Ces architectures dédiées accélèrent uniquement l'ordonnancement comme l'architecture *cs2* de l'université de Waterloo [138] ou l'architecture de l'université de Rostock qui implémente un algorithme temps-réel proche du LLF [139]. D'autres mettent en œuvre des algorithmes d'ordonnancement plus simples mais permettent de réveiller des tâches sur des interruptions extérieures. C'est le cas des architectures *F-Timer* de l'université fédérale de Rio Grande [140], *RTM* de l'université du Maryland [141] ou *CHS* de l'Institut Technologique de Georgie [142]. Enfin, des architectures offrent en plus un ensemble de sémaphores pour gérer les dépendances entre les tâches et les synchronisations. Deux solutions monoprocesseurs existent actuellement. La première, nommée *Sierra-RTμK*, est commercialisée par la société *RealFast* [143]. Elle est issue des travaux réalisés pour l'architecture *RTU94* qui sera étudiée plus en détail par la suite. Elle ordonnance jusqu'à 16 tâches périodiques selon des priorités statiques, supporte l'utilisation de 16 sémaphores et de 8 interruptions extérieures. La deuxième solution est l'architecture *Silicon TRON* de la société *TRON* [144] qui est présentée plus en détail ci-après.

L'architecture Silicon TRON

L'architecture *Silicon TRON* supporte une partie des primitives existantes dans le noyau temps-réel *μITRON* [145]. Ce composant est lié au noyau *μITRON* en charge des autres primitives supportant la gestion du système et de l'interface avec l'utilisateur (figure 2.11). Les primitives logicielles permettent de maintenir la cohérence avec la partie applicative ainsi que l'évolutivité et la programmabilité du système.

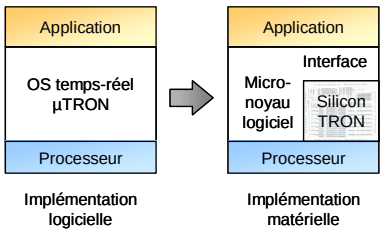

FIG. 2.11 – Le concept de l'architecture Silicon TRON

Comme le montre la figure 2.12, l'architecture *Silicon TRON* s'interface aisément au niveau système grâce à un ensemble d'évènements et d'interruptions extérieurs. Elle gère les interruptions provenant du noyau, ainsi que les tâches temps-réel et leur ordonnancement. Elle est composée de 3 modules élémentaires : le module *flag* dont chaque objet est de 1 bit, le module *sémaphore* (3 bits) et enfin le module *horloge* (8 bits). Tout d'abord, l'ensemble des flags conserve le statut des évènements et des interruptions. Ensuite, le groupement des

sémaphores permet les synchronisations entre les tâches. Enfin, les horloges supportent les fonctions temporelles liées aux tâches temps-réel. *Silicon TRON* est également constitué d'un bloc appelé *tâches* contenant toutes les informations relatives à l'exécution des tâches. De plus, il est composé d'un ordonnanceur et d'un bloc de contrôle administrant l'ensemble du système et des signaux d'entrées/sorties. L'ordonnancement est basé sur les niveaux de priorité qui sont définis avant l'exécution. Lorsque plusieurs tâches ont le même niveau de priorité, l'ordonnanceur traite les tâches suivant leur date d'arrivée (FCFS : *First Come First Serve*) ou en attribuant à chaque tâche un temps d'exécution limité de type *tourniquet*. Par ailleurs, l'architecture supporte 8 niveaux de priorité pour gérer l'ensemble des interruptions extérieures.

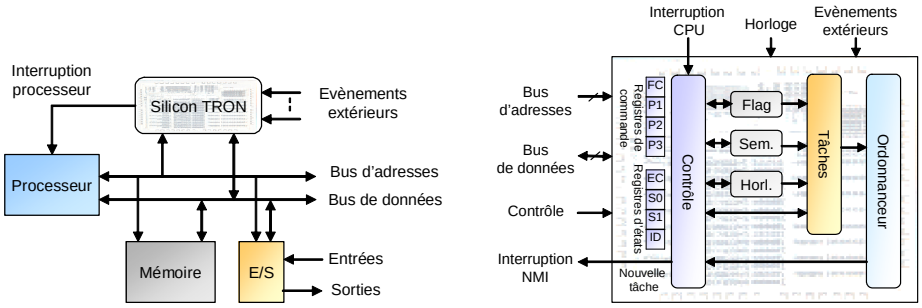

FIG. 2.12 – Détail de l'architecture Silicon TRON et son environnement

Silicon TRON réduit les temps de traitement et de prise en compte des interruptions. Dans un système d'exploitation temps-réel logiciel, chaque interruption nécessite l'intervention de multiples requêtes avec le contrôleur d'interruptions. Ceci génère un surcoût temporel non négligeable sur le temps d'exécution (figure 2.13). Au contraire, l'architecture Silicon TRON permet la prise en compte asynchrone d'évènements extérieurs. Mais surtout, la matérialisation et la centralisation de la gestion des tâches évitent l'utilisation d'un contrôleur d'interruption. Par ailleurs, pour éviter que la prise en compte d'une interruption soit retardée, toutes les interruptions sont traitées avant de demander l'exécution des actions correspondantes. Pour cela, elles sont stockées dans des files d'attente spécifiques. Par ailleurs, les temps de réactivité de Silicon TRON ne dépendent pas du niveau de priorité des tâches, comme pour un système d'exploitation logiciel.

L'architecture Silicon TRON a donné lieu à un circuit d'environ 40 000 portes équivalentes en technologie BiCMOS 0,5μm [146]. Elle supporte l'exécution de 31 tâches et l'implémentation de 16 modules élémentaires. Une comparaison avec un système composé du processeur SH-1 d'Hitachi et du noyau $\mu ITRON$ a également été effectuée. Elle montre une accélération d'environ 200%. L'ordonnancement est réalisé en 8 cycles d'horloge pour un nombre maximal de 7 tâches [147]. Mais ce temps est négligeable puisque l'ordonnancement est effectué de manière concurrente avec le noyau. D'autre part, l'avantage de ce couplage indépendant entre les systèmes d'exploitation logiciel et matériel permet un plus fort parallélisme d'exécution et apporte une plus grande réactivité. En effet, toutes les opérations effectuées par le processeur comme l'ordonnancement des tâches, les synchronisations, ou les communications, génèrent un surcoût temporel puisque les applications ne peuvent pas être traitées simultanément.

Toutes ces solutions ont démontré l'intérêt d'accélérer matériellement les exécutifs temps-réel. Des algorithmes d'ordonnancement efficaces, mais habituellement inexploitables, retrouvent un sens. Mieux, avec peu de complexité matérielle supplémentaire, ces solutions

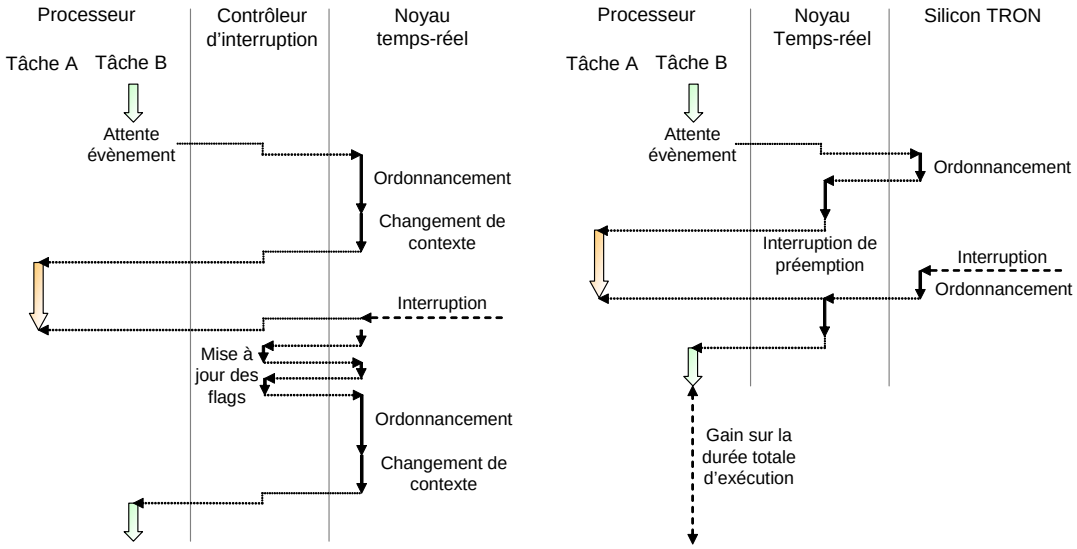

FIG. 2.13 – Analyse fonctionnelle de la prise en compte des interruptions par un système d'exploitation temps-réel logiciel et matériel

de contrôle réduisent considérablement les temps d'exécution et augmentent d'autant les performances. Ainsi, de telles améliorations profiteraient très certainement au contrôle des architectures multiprocesseurs. C'est ce que nous allons étudier dans la sous-section suivante.

2.2.2 Les accélérateurs de noyau temps-réel pour les multiprocesseurs

Dans le cadre des architectures multiprocesseurs, deux approches se distinguent. La première utilise un élément matériel pour simplifier la gestion des ressources partagées. La seconde tente, à la manière de Silicon TRON, d'apporter une solution matérielle capable de rassembler un grand nombre de services habituellement supportés par les systèmes d'exploitation.

Gestion des ressources partagées

La gestion des ressources partagées engendre un surcoût important dans le contrôle de structures multiprocesseurs. En effet, il faut permettre aux différents processeurs d'accéder à des ressources communes en garantissant des accès exclusifs et l'intégrité des données. Pour limiter son impact sur les performances, de nombreuses solutions matérielles ont vu le jour. Par exemple, l'architecture *Real Time Processor Operating System* (RTPOS) a été développée afin de gérer la complexité du partage de multiples ressources de mémorisation et de calcul [148]. Il permet de réserver l'utilisation d'une ressource par une seule tâche ou de partager une ressource parmi plusieurs tâches concurrentes.

D'autre part, l'Institut de Technologie de Georgie et l'université de Mälardalen ont mis au point une méthodologie de partitionnement de noyau temps-réel [149]. L'utilisateur peut choisir d'accélérer quelques fonctions d'un noyau temps-réel en utilisant une liste d'éléments matériels. Ces derniers regroupent un mécanisme de protection de cache pour les opérations de synchronisation (*SoCLC : System-on-Chip Lock Cache*) [150, 151], une unité de détection de blocages pour les systèmes multiprocesseurs (*SoCDDU : System-on-Chip Detection Unit*)

[152] et un gestionnaire de mémoire dynamique appelé *System-on-Chip Dynamic Memory Management Unit* (SoCDMMU) [153].

L'architecture *SoCDMMU* permet une gestion rapide, déterministe et dynamique d'une mémoire locale partagée [154]. Elle alloue ou désalloue des emplacements mémoires et gère la distribution des ressources de mémorisation pour une meilleure utilisation de la mémoire partagée. Le SoCDMMU contient une table d'adresses virtuelles qui référencient des emplacements dans la mémoire partagée (voir figure 2.14). Chaque adresse virtuelle est associée à une ressource de calcul. Le bloc responsable de l'allocation appelé *Basic SoCDMMU* effectue l'allocation et la libération suivant un algorithme appelé *First Fit*. Celui-ci consiste à parcourir la mémoire et à réserver le premier emplacement de taille suffisante. Cet élément matériel a été synthétisé pour un système constitué de 4 processeurs ARM9TDMI et d'une mémoire SRAM de 16 Mo divisée en 256 blocs de 64 Ko [153]. Il représente 41 500 portes équivalentes en technologie 0,5μm. La comparaison avec une solution complètement logicielle montre une accélération de l'ordre de 440%.

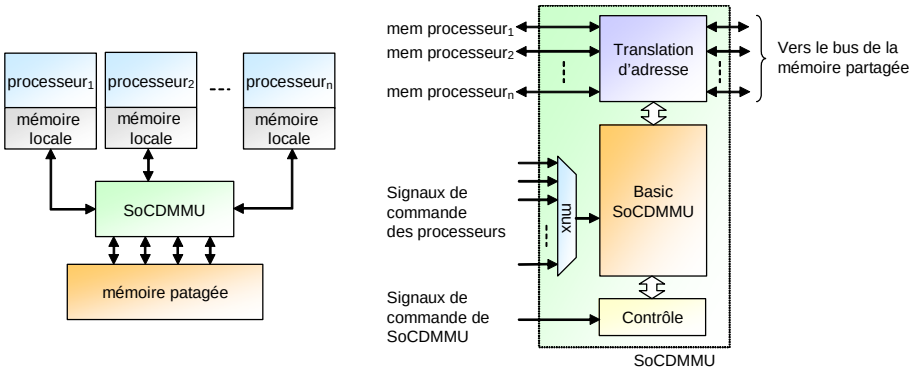

FIG. 2.14 – Modèle fonctionnel de l'architecture SoCDMMU

Gestion de l'ordonnancement et de l'allocation des tâches

La seconde approche tente d'apporter une solution pour l'ordonnancement et l'allocation de tâches temps-réel. Nous avons vu précédement qu'aucune solution logicielle n'existe pour les multiprocesseurs asymétriques. Au contraire, toutes les solutions matérielles en charge de l'ordonnancement reposent sur un contrôle centralisé. Aujourd'hui, seulement 3 solutions existent : l'architecture *Spring*, *SystemWeaver* et *RTU94*.

Spring L'architecture Spring est un accélérateur d'algorithme d'ordonnancement dynamique prenant en charge l'exécution de tâches sur des structures multiprocesseurs [155, 156]. Cette heuristique appelée *myopic* génère, tâche par tâche, une suite finie de tâches à ordonnancer et détermine la faisabilité de tous les ordonnancements partiels générés. Contrairement aux algorithmes d'ordonnancement temps-réel classiques, celui-ci prend en compte l'état des ressources. En fait, l'ordonnancement est pseudo-dynamique car il génère au cours de l'exécution des ordonnancements partiels statiques. Ceci est possible puisque les préemptions ne sont pas autorisées, mais cet algorithme limite le taux d'occupation des ressources et la réactivité du système. De plus, cet algorithme prend en compte les contraintes de précédence entre les tâches. L'ensemble de ces informations est agencé sous forme matricielle. La taille de la matrice dépend alors du nombre de tâches supportées [157]. En fait, les tâches dépendantes sont

regroupées en un ensemble de tâches disposant d'une seule échéance. Cette architecture a été synthétisée avec une technologie CMOS 0,8μm pour 16 tâches et 16 ressources (6,4 mm^2) et pour 32 tâches et 32 ressources (59,84 mm^2).

SystemWeaver La société Ignios propose une architecture appelée *SystemWeaver* [158]. Cette solution permet l'ordonnancement et l'allocation de tâches sur une structure homogène ou hétérogène. Elle supporte également les synchronisations et la sélection des tâches suivant des contraintes de précédence. Les multiples ressources sont regroupées en nœuds de distribution définis à la compilation. Chaque nœud est associé à une politique d'ordonnancement particulière basée sur les niveaux de priorité ou des algorithmes de type *round-robin*. Enfin, les différentes tâches sont attribuées dynamiquement aux nœuds de distribution en fonction de leur charge de calcul. Cette solution a donné lieu à un circuit d'environ 100 000 portes équivalentes, soit 0,4 mm^2 en technologie 90 nm. Le temps d'ordonnancement est d'environ 1 μs pour un taux d'occupation des ressources de calcul de l'ordre de 90%.

RTU94 L'élément de contrôle appelé *Real Time Unit 94* (RTU94) développé par l'université de Mälardalen supporte jusqu'à 64 tâches sur 8 niveaux de priorités [159, 160, 161]. Il permet l'ordonnancement et l'allocation de tâches sur trois processeurs homogènes. Il est composé de 4 files de tâches prêtes, d'une interface de bus VME et d'un registre d'état et de contrôle accessible de l'extérieur par les 3 processeurs (voir figure 2.15).

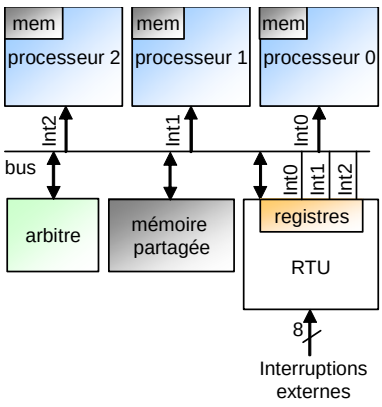

FIG. 2.15 – Modèle fonctionnel de l'architecture RTU94

Comme le montre la figure 2.16, chacune de ces files est associée à un processeur, sauf une qui contient des tâches pouvant s'exécuter sur n'importe quel processeur disponible. Le banc de registres *Old* contient l'identifiant de la tâche en cours d'exécution et le banc de registres *New* l'identifiant de la tâche suivante. Lors d'un changement de tâche, tous les registres présents dans la *file de registres*, sont sauvegardés dans la *mémoire TCB* à l'emplacement indiqué par l'identifiant de la tâche en cours d'exécution. Lorsque la tâche courante se termine, le contenu du registre *New* est transféré dans celui du registre *Old* et la tâche suivante la plus prioritaire, présente dans la file de *tâches prêtes*, est transférée dans le banc de registres *New*. Le contenu de la *mémoire TCB* correspondant à la nouvelle tâche est alors transféré dans la *file de registres* partagée avec le processeur, pour permettre son exécution.

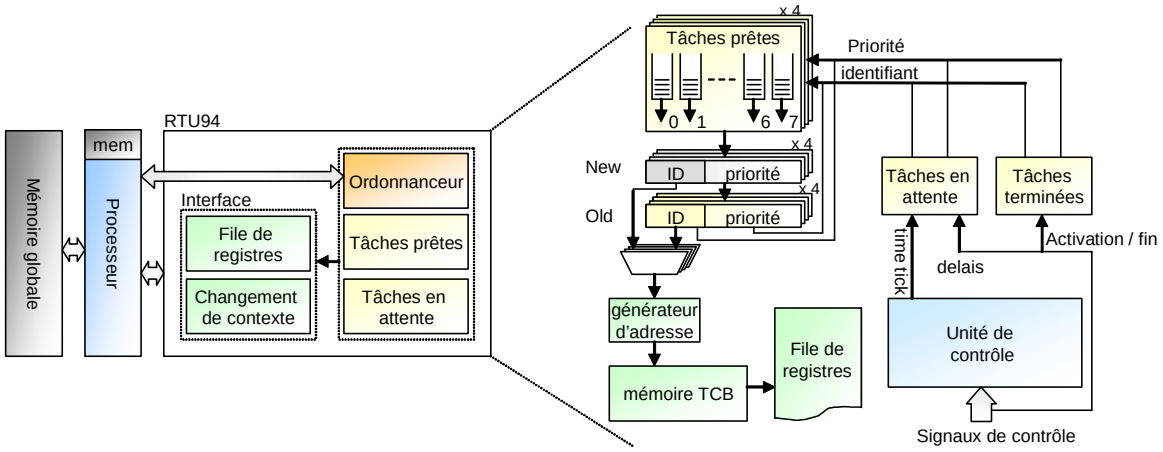

FIG. 2.16 – Modèle fonctionnel de l'architecture RTU94

Le prototype développé représente environ 250 000 portes équivalentes en technologie 0,25 μm [162]. Son utilisation montre un gain de 36% lors de l'exécution d'une application mobile composée de 30 tâches. RTU94 réduit les durées de préemption, de communication et de synchronisation entre les tâches, ainsi que la plupart des services liés à l'ordonnancement [163]. De plus, une version monoprocesseur du RTU94 présente une consommation d'énergie d'environ 2,5 mW lors de l'exécution d'une application pour les télécommunications [164].

2.2.3 Synthèse de l'approche matérielle

L'utilisation d'un composant matériel pour accélérer une ou plusieurs primitives d'un système d'exploitation apporte toujours un gain très important en terme de performance. Ceci est d'autant plus vrai pour les architectures multiprocesseurs qui nécessitent davantage de contrôle. Finalement, rajouter un élément matériel supplémentaire ne contribue pas à la complexité du circuit. Ces quelques millimètres carrés de silicium simplifient les ressources de calcul et améliorent de manière très significative les performances du système.

Contrairement à l'approche logicielle, les accélérateurs matériels traitent avec beaucoup d'efficacité les défauts liés à l'asymétrie de la structure multiprocesseur. La réduction des latences de synchronisation ou d'ordonnancement envisage une approche architecturale inconcevable jusqu'alors. De plus, ils permettent d'utiliser des ordonnancements dynamiques, voire même des ordonnancements inutilisables avec des solutions logicielles comme le LLF.

D'autre part, cette approche maîtrise et réduit les durées d'exécution du contrôle. Autrement dit, elle améliore le déterminisme et les temps de réponse de manière conséquente, en partie grâce à l'abandon du modèle en couches pour communiquer avec les ressources matérielles. Ceci a un effet très positif sur l'occupation des ressources de calcul. D'ailleurs, l'architecture *SystemWeaver* parvient à des taux d'occupation encore jamais atteints, malgré un algorithme de répartition de charge de faible complexité.

Ainsi, l'utilisation d'un composant matériel pour soutenir le système d'exploitation semble être incontournable lors de l'exécution de primitives de contrôle critiques. Le contrôle des architectures multiprocesseurs hétérogènes passe par une accélération du noyau temps-réel.

2.3 Synthèse

Ce second chapitre a permis d'explorer l'ensemble des solutions existantes pour contrôler les architectures multiprocesseurs. La première section a mis en évidence les limitations liées à l'utilisation d'une solution logicielle. L'emploi de couches d'abstraction logicielles simplifient la programmabilité des architectures, mais au détriment des performances. Là est tout le paradoxe que nous avons tenté de mettre en évidence. Finalement, aucune solution logicielle n'a été trouvée pour répondre à nos contraintes temps-réel.

Dans la seconde section, nous nous sommes intéressés à des solutions de contrôle accélérées par des éléments matériels. Elles mettent en œuvre des procédés complexes d'ordonnancement ou de synchronisation, sans trop pénaliser le système. Ainsi, elles sont les seules à pouvoir supporter la complexité du contrôle dans les architectures multiprocesseurs asymétriques. Cependant, les approches qui intègrent des algorithmes d'ordonnancement ne sont pas suffisantes pour répondre aux besoins des futures applications mobiles. Elles ne permettent pas :

- de créer ou d'interrompre dynamiquement des tâches ou des applications ;
- de gérer avec efficacité les synchronisations et les précédences entre les tâches ;
- d'ordonnancer à la fois des tâches temps-réel et non-temps-réel ;
- de supporter l'exécution de tâches non-périodiques temps-réel ;
- de garantir au cours de l'exécution le respect des contraintes temporelles ;
- de préempter et de changer dynamiquement la priorité ;
- de gérer les ressources de mémorisation et de calcul hétérogènes ;
- de migrer efficacement les tâches entre les ressources de calcul ;
- de gérer la consommation d'énergie ;
- et de faciliter la programmation.

Ainsi, une architecture de contrôle efficace pour les systèmes embarqués doit pouvoir répondre à toutes ces exigences. Dans un premier temps, il nous faut néanmoins définir un modèle de multiprocesseur asymétrique qui doit être en mesure d'apporter une réponse aux problèmes soulevés dans la section 1.7 du premier chapitre. Dans un deuxième temps, les spécifications de l'architecture en charge du calcul doivent permettre de concevoir une architecture de contrôle adaptée et de définir des solutions répondant de manière plus générale au contrôle de ces structures.

Chapitre 3

L'architecture SCMP-LC

Sommaire

Le premier chapitre a mis en évidence l'intérêt de concevoir des architectures multiprocesseurs pour répondre aux besoins des futures applications embarquées. Il a exploré les différentes solutions architecturales et souligné les avantages d'une approche asymétrique. Le second chapitre s'est donc intéressé au contrôle de ces architectures. Il a montré qu'une solution logicielle ne pourrait pas être utilisée, car elle pénaliserait les performances du système. En effet, même si un contrôle centralisé apporte de nombreux avantages, il devient la principale cause de limitation du système s'il n'est pas suffisamment réactif. Heureusement, l'utilisation d'un accélérateur matériel envisage à nouveau cette approche asymétrique.

Ainsi, ce chapitre présente une architecture multiprocesseur appelée *(Loosely-Coupled SCalable Multi-Processor)* (SCMP-LC). Avant d'introduire cette nouvelle architecture, il identifie les problèmes liés aux architectures multiprocesseurs, afin d'apporter une solution répondant

à toutes ces limitations. Ensuite, le modèle d'exécution de l'architecture SCMP-LC est présenté. Tout en étant proche du modèle CMP présenté dans le premier chapitre, notre nouveau modèle d'exécution facilite la préemption et la migration des tâches. Il met en œuvre des mécanismes particuliers pour les communications et la gestion des ressources de mémorisation. Enfin, ce chapitre caractérise les différents éléments qui la composent et son principe d'exécution.

3.1 Identification des problèmes liés aux multiprocesseurs

De manière générale, les architectures composées de multiples ressources de calcul souffrent toutes de la mise en œuvre complexe des synchronisations, du partage des données et des dépendances de contrôle et de données. De plus, dans les structures hétérogènes, les problèmes de communication entre des entités de nature différente nécessitent l'utilisation d'interfaces supplémentaires [49]. D'autre part, des moyens de communication sous-dimensionnés et des temps d'exécution peu maîtrisés contribuent au non-déterminisme du système, ainsi qu'à une mauvaise utilisation de ses ressources. Pour concevoir un multiprocesseur performant, chacun de ces aspects doit être pris en compte et une réponse appropriée apportée afin de garantir des performances optimales. Dans un premier temps, nous nous intéressons aux synchronisations et aux dépendances de contrôle et de données.

3.1.1 Les dépendances de contrôle et de données

Les dépendances de contrôle et de données induisent une réduction de l'occupation des ressources et des performances de l'architecture. En effet, chaque nouvelle dépendance peut potentiellement bloquer la tâche en cours d'exécution pendant un temps indéterminé. La donnée ou l'information de contrôle est produite par une tâche concurrente dont la date d'activation est incertaine. Il y a même un risque d'interblocage dans la mesure où l'exécution de la tâche concurrente peut être conditionnée par une information produite plus tard par la tâche en attente. Enfin, ces dépendances nécessitent des moyens de communication et de synchronisation entre les processeurs. Par conséquent, elles engendrent une complexité matérielle supplémentaire.

Le fait d'identifier les dépendances et les synchronisations avant de demander l'exécution des tâches permet de s'affranchir, dans de nombreux cas, des mécanismes de protection des données. Des conflits d'écriture et de lecture peuvent être évités. De plus, les mécanismes de protection induisent des surcoûts temporels non négligeables pour accéder à des ressources de mémorisation partagées. Nous avons vu, dans le premier chapitre (section 1.6.1), qu'une solution consistait à distribuer les mémoires afin de rendre leurs accès privilégiés et de s'affranchir de ces moyens de protection. Néanmoins, certaines applications ont besoin de partager des données entre des processus concurrents. Dans ce cas, des mécanismes de communication par envoi de messages entre les ressources de calcul distantes sont utilisés. Les temps d'accès aux données devenant plus longs, les performances globales tendent à diminuer.

Une solution consiste à partager physiquement ces mémoires distribuées entre toutes les ressources de calcul. La figure 3.1 illustre ce principe qui a largement été étudié dans la littérature sous le nom de *Parallel Random Access Machine* [165]. En effet, le problème d'écriture simultanée dans une unique mémoire n'existe plus si chaque producteur possède une mémoire

locale non partagée en écriture. En revanche, une latence supplémentaire est à considérer lors de lectures simultanées (figure 3.2). Il faut alors envisager le pire cas pour définir la durée d'acquisition des données qui correspond à l'instant où toutes les tâches susceptibles d'écrire ou de lire dans la mémoire y accèdent simultanément. Cette pénalité n'a pas d'incidence sur l'intégrité des données. Par ailleurs, si le nombre de ressources de mémorisation le permet, les données produites peuvent être dupliquées ou écrites dans des mémoires différentes pour chaque consommateur. Alors, il n'y a plus de pénalité à considérer lors de l'accès en lecture.

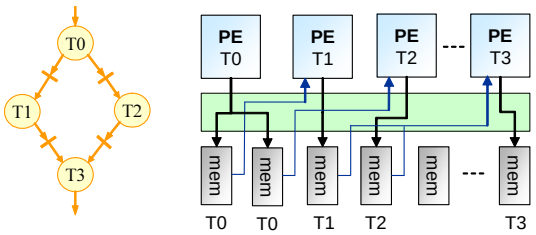

FIG. 3.1 – Modèle d'architecture avec mémoires distribuées partagées. Chacun des processeurs élémentaires (PE) accède à une des mémoires partagées lors du traitement d'une tâche. Le réseau d'interconnexion permet à chacun des PE de communiquer avec une des mémoires distribuées et partagées. Ce modèle réduit les pénalités liées aux accès aux données.

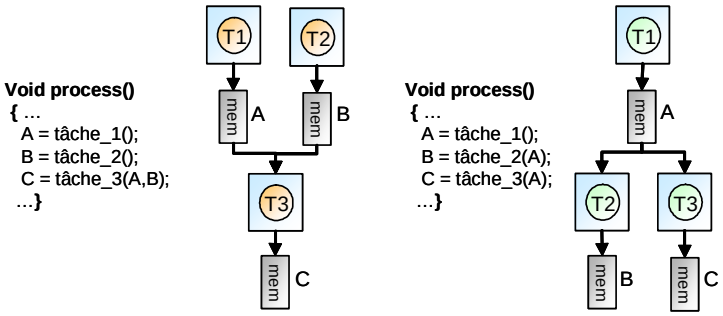

FIG. 3.2 – Partage des données dans un modèle d'architecture avec mémoires distribuées partagées. A gauche, les tâches $T1$ et $T2$ produisent un résultat utilisé par la tâche $T3$ (convergence en ET). A droite, la tâche T1 produit un résultat consommé par les tâches $T2$ et $T3$ (divergence en ET). La lecture simultanée de la donnée A, à des adresses différentes, augmente les temps d'accès en lecture. La bande passante sur chacun des liens est divisée par deux.

Les variables globales

Dans le cas des variables globales, la première solution consiste à multiplier les informations en les distribuant physiquement dans chacune des mémoires locales partagées. Néanmoins, ceci n'est pas possible avec d'importantes quantités de données et pose surtout des problèmes de cohérence en cas de modification de ces variables. Par conséquent, la meilleure solution consiste à regrouper les variables globales dans une mémoire partagée et à protéger ses accès en utilisant des mécanismes de verrou. Il faut alors considérer pour chaque accès aux variables globales un temps de latence qui dépend du nombre de tâches ayant besoin de ces données. Pour de faibles quantités de variables, la pénalité est heureusement peu importante. Si, au contraire, les variables globales sont trop nombreuses, des synchronisations explicites entre les tâches devront être spécifiées.

La cohérence des données

Le respect de la cohérence des données est complexe à garantir, ce qui conduit à réduire considérablement les performances des systèmes distribués. Pour les systèmes utilisant un espace mémoire distribué, la cohérence des données est plus difficile à maintenir si chaque ressource de calcul est autorisée à écrire ailleurs que dans sa propre mémoire locale. Ce qui est le cas avec des mémoires partagées et distribuées. La figure 3.3 illustre le fonctionnement d'une architecture *UMA : Uniform memory Access* [166, 167] lors d'une convolution par partie d'une image. Ce dispositif évite qu'une division de l'image subisse plusieurs traitements successifs. Pour cela, l'état d'avancement des différents traitements doit être connu par chacune des ressources de calcul distribuées.

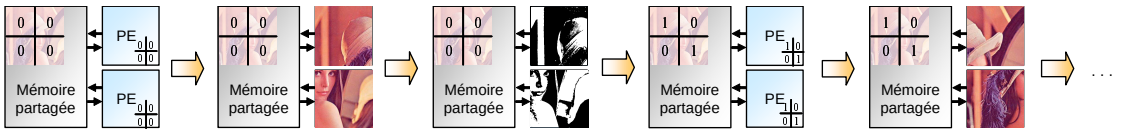

FIG. 3.3 – Modèle d'exécution avec mémoire partagée et mécanisme de cohérence de cache. L'exemple choisi est une application de filtrage sur une image constituée de 4 imagettes. Chaque ressource de calcul peut effectuer cette opération morphologique sur une imagette. Elle dispose localement des informations nécessaires pour identifier les parties de l'image à traiter. Un 0 indique qu'aucun traitement n'a eu lieu, alors qu'un 1 signifie que la partie de l'image correspondante a subi une transformation. Des problèmes de cohérence peuvent donc survenir si les informations ne sont pas mises à jour à temps.

Cependant, la description explicite des précédences d'exécution ou des synchronisations avant l'exécution des tâches résout la plupart des problèmes de cohérence avec un modèle d'architecture à mémoires distribuées et partagées. En effet, la modification par l'une des ressources de calcul de sa mémoire locale n'a pas d'influence directe sur l'exécution des autres ressources. Les écritures concurrentes ne sont pas possibles si toutes les tâches productrices de données disposent d'une mémoire unique.

Comme le montre la figure 3.4, l'extraction des dépendances de contrôle et de données assure la cohérence des traitements sans utilisation de procédés particuliers. Chaque traitement peut avoir lieu sans se soucier de l'état d'avancement des autres parties de l'image. Un simple mécanisme d'allocation suffit pour identifier les ressources libres.

Un modèle d'exécution général

Ce modèle présente de nombreux avantages. Il garantit des accès exclusifs aux mémoires comme dans le modèle à mémoires distribuées. Il partage cependant l'ensemble des données entre toutes les ressources de calcul et ne nécessite pas de mécanismes de protection particuliers. Enfin, il ne requiert pas un partitionnement explicite des tâches. Il apporte suffisamment de flexibilité pour autoriser leur exécution sur n'importe quelle ressource de calcul. Il est donc possible d'envisager la migration des tâches puisque les données sont visibles par toutes les ressources de calcul.

Ce modèle d'exécution général inclut la possibilité de faire du flot de données (figure 3.5). Mais avant tout, il permet à une tâche d'utiliser des données qui ne sont pas en cours de production et qui ont été générées par des tâches qui ne la précédaient pas immédiatement. Ainsi des tâches devant s'échanger des données n'ont pas besoin d'être simultanément présentes sur les ressources de calcul. Ceci n'est possible que si les échanges de données ont lieu

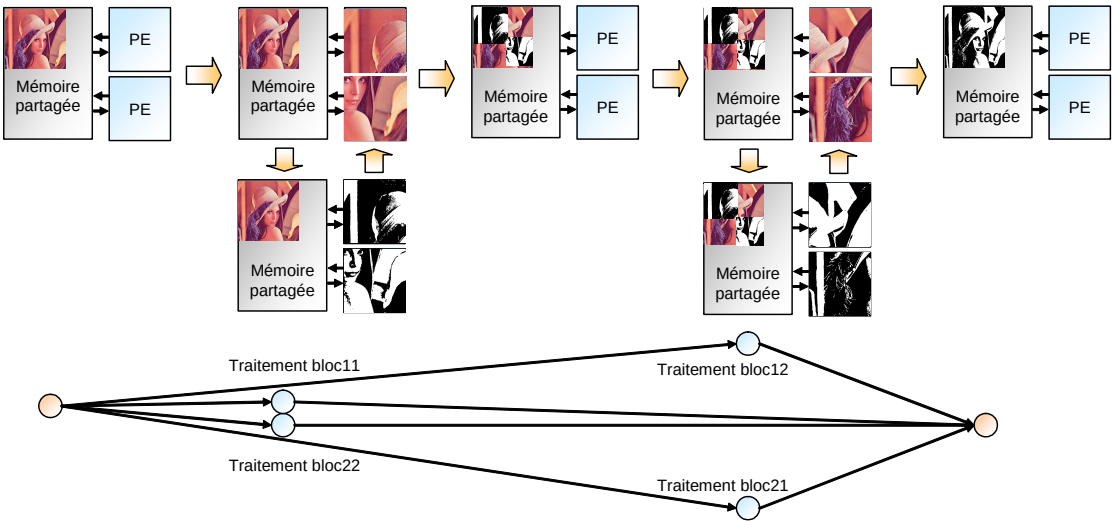

FIG. 3.4 – Modèle d'exécution avec mémoires partagées et distribuées et gestion des dépendances de contrôle et de données. Le respect de ces dépendances permet de garantir la cohérence des données.

via des mémoires partagées. Eventuellement, lorsque le consommateur et le producteur sont présents, un chemin direct peut être établi pour améliorer les performances. L'utilisation de mémoires pour les communications résout par ailleurs les problèmes de communication entre des entités de nature différente.

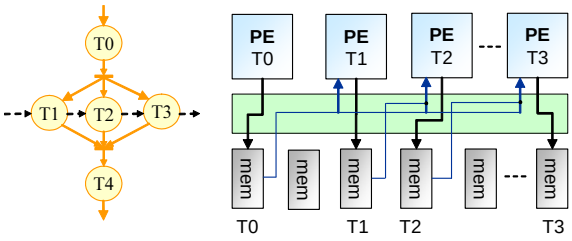

FIG. 3.5 – Modèle d'exécution flot de données de l'architecture SCMP-LC. L'exécution des tâches commence dès que les résultats intermédiaires sont prêts. Les synchronisations locales sont effectuées via les mémoires locales.

Cette approche n'est néanmoins possible que si les emplacements mémoires sont réservés aux tâches pendant toute la durée de leur traitement. Par ailleurs, le réseau doit être correctement dimensionné pour autoriser l'ensemble des communications avec les différentes mémoires. Chaque processeur doit pouvoir accéder à toutes les mémoires partagées, afin d'avoir la possibilité d'accueillir toutes les tâches éligibles. Les tâches en cours d'exécution n'ont pas besoin de tenir compte des modifications de données réalisées par d'autres tâches concurrentes ou de leur état d'avancement. Ainsi, chaque traitement doit être contraint par rapport aux autres. Il doit attendre l'exécution des traitements précédents pour s'exécuter et accéder aux données modifiées. L'exécution des tâches ne peut donc pas être interrompue. Cette simplification du partage des données demande, en contrepartie, un contrôle efficace de l'exécution des tâches. Elle interdit les intéractions entre les tâches. Pour ces raisons, ce modèle n'a jamais été mis en œuvre mais l'emploi d'un contrôleur performant permettrait d'envisager dorénavant son utili-

sation. En effet, la complexité du contrôle nécessaire pour gérer les dépendances de contrôle et de données, les ressources de mémorisation, ainsi que les échanges entre les tâches, deviendrait peu pénalisante.

3.1.2 L'occupation des ressources de calcul

Le taux d'utilisation des ressources de calcul dépend de nombreux facteurs. Il est dépendant du nombre de tâches indépendantes, de l'adéquation entre les applications à exécuter, du type des ressources disponibles, des algorithmes d'allocation et d'ordonnancement utilisés, ou encore de l'efficacité des mécanismes de synchronisation. D'un point de vue plus matériel, le taux d'occupation est également lié à l'efficacité du réseau d'interconnexion, aux temps d'accès des mémoires, ou à la réactivité du contrôle. Chacun de ces points sera détaillé dans les sections suivantes.

La faiblesse des architectures CMP est liée au faible taux d'occupation de ses ressources. La première approche pour l'améliorer consiste à utiliser des mécanismes de préemption. Nous avons vu que la préemption est nécessaire pour permettre un ordonnancement en ligne efficace et capable de s'adapter aux besoins de l'application. Comme le montre la figure 3.6, si la préemption n'est pas autorisée, l'arrivée de la tâche $T5$ empêche le système de respecter ses contraintes temps-réel. Si une tâche avait pu s'interrompre pour exécuter cette tâche plus prioritaire, son échéance aurait pu être respectée. Bien sûr, un système surdimensionné garantirait le respect des échéances, mais il engendrerait une moindre occupation des ressources de calcul.

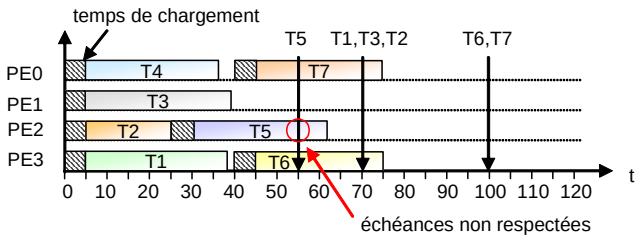

FIG. 3.6 – Ordonnancement temps-réel sans préemption. Soit un ensemble de tâches $\{T1, T2, T3, T4, T5, T6, T7\}$ tel que l'exécution des tâches $T6$ et $T7$ ne peut avoir lieu qu'à la fin de l'exécution des tâches $T1$, $T2$, $T3$ et $T4$. La tâche $T5$ est indépendante et sa date d'arrivée exacte est inconnue. Le système est supposé ordonnançable. Pourtant, l'arrivée de la tâche $T5$ en t=18 ne peut être prise en compte qu'à la fin de la tâche $T2$: $T5$ ne peut pas respecter son échéance.

Si maintenant la préemption est autorisée, dans l'exemple de la figure 3.6, sans possibilité de continuer l'exécution de la tâche interrompue sur une *autre* ressource de calcul, l'exécution ne respecte toujours pas ses contraintes temporelles (figure 3.7). Pourtant, des processeurs inutilisés pourraient être exploités pour exécuter la tâche préemptée. Ainsi, la préemption peut induire une sous-utilisation importante des ressources disponibles. Elle conduit à exécuter peu de tâches pour être sûre de respecter toutes les échéances d'exécution.

Une solution consiste à continuer l'exécution de la tâche interrompue sur une autre ressource de calcul libre. C'est la migration de tâche. Elle a cependant un coût important lié au transfert du contexte d'exécution de la tâche vers une mémoire partagée, ou vers une autre ressource de

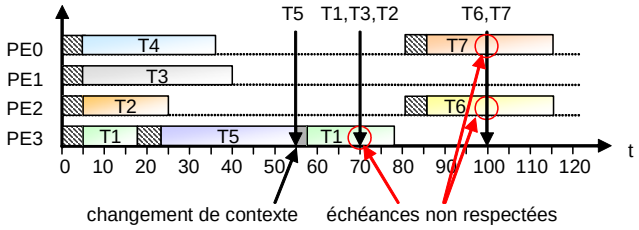

FIG. 3.7 – Ordonnancement temps-réel sans préemption et sans migration. Soit le même ensemble de tâches que précédemment, l'arrivée de la tâche $T5$ en t=18 préempte la tâche la moins prioritaire $T1$. L'exécution de la tâche $T1$ reprend sur le même processeur et les contraintes temps-réel suivantes ne sont pas respectées.

calcul. Pour cette raison, la migration n'est pas utilisée dans les architectures multiprocesseurs. Néanmoins, l'utilisation de mémoires distribuées et partagées, telles qu'elles sont présentées précédemment, apporte un avantage certain.

En effet, le contexte d'exécution est sauvegardé dans une mémoire accessible par toutes les autres ressources de calcul. Il n'y a alors aucun transfert à effectuer entre les ressources de calcul et l'exécution n'est pas pénalisée. Si l'on considère le même exemple, l'occupation des ressources de calcul passe désormais de 54,4% à 75,8% en permettant juste la migration des tâches (figure 3.8). Par ailleurs, le temps d'exécution global est amélioré de 30%. Ceci laisse envisager des contraintes temporelles plus proches d'une exécution optimale. Par conséquent, il devient possible de traiter davantage de tâches pour une même durée d'exécution ou d'avoir besoin de moins de ressources matérielles pour le même traitement.

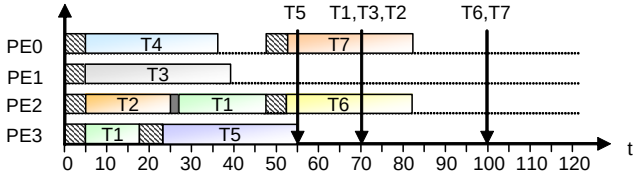

FIG. 3.8 – Ordonnancement temps-réel avec préemption et migration. Soit le même ensemble de tâches que précédemment, l'arrivée de la tâche $T5$ en t=18 préempte la tâche la moins prioritaire $T1$. L'exécution de la tâche $T1$ reprend sur le premier processeur disponible. L'occupation des ressources est mieux maîtrisée et les contraintes temps-réel sont respectées.

La préemption et la migration des tâches permettent une meilleure occupation des ressources de calcul et une plus grande réactivité du système. Cependant, ces procédés induisent un coût non négligeable sur les performances et sur la consommation d'énergie. En effet, le système est soumis à une activité supplémentaire et les *temps de préemption* ou de chargement des tâches augmentent les durées d'exécution des tâches. Le temps de préemption considère le temps de sauvegarde du contexte d'exécution et le temps de chargement de la nouvelle tâche à exécuter. Au vu du gain qu'il est possible d'obtenir avec un contrôle efficace et une migration simplifiée, il est intéressant d'évaluer sa mise en œuvre. Par ailleurs, pour améliorer encore l'occupation des ressources de calcul, il faut optimiser le déterminisme de l'architecture. Cela revient à pouvoir évaluer précisément les durées d'exécution des tâches et le comportement du système.

3.1.3 Le déterminisme

La conception d'un système temps-réel doit tenir compte du comportement prédictible de l'exécution. Les sources de non-déterminisme sont nombreuses. Elles peuvent être liées à un réseau sous-dimensionné, à un contrôle inadapté (sous-section 2.1.4) ou à l'utilisation de mémoires caches. Dans un premier temps, cette sous-section s'intéresse essentiellement aux mémoires caches. L'étude du réseau sera effectuée ultérieurement.

Ces mémoires sont des éléments de mémorisation locaux exploitant la virtualisation de l'espace mémoire. Elles sont capables de mémoriser les informations que le processeur est le plus susceptible de demander. La localité des informations utiles à l'exécution améliore les temps d'alimentation en instructions et les performances du système. Le problème majeur de ces mémoires est qu'elles ne permettent pas de garantir si la prochaine donnée ou instruction sera présente. Par conséquent, les temps d'accès aux données sont inprévisibles et le WCET des tâches ne peut pas être évalué précisément [168].

Ces problèmes peuvent modifier les temps d'exécution prévus et empêcher une tâche de respecter son échéance d'exécution. Bien sûr, il est toujours possible d'estimer le pire temps d'exécution (WCET) de chaque tâche en tenant compte des mémoires caches. Il faut alors considérer le pire comportement lors de chaque accès à la mémoire [169]. Cependant, ceci engendre une surévaluation du WCET de chaque tâche et réduit considérablement le taux d'occupation des ressources de calcul.

Par la suite, nous considérons que les mémoires sont locales. Seul ce choix garantit des durées d'exécution prédictibles et donc un taux d'occupation élevé des ressources de calcul sous des contraintes temps-réel.

3.1.4 Bilan des solutions proposées

Nous avons vu que les ressources de mémorisation physiquement distribuées et logiquement partagées répondent à la plupart des problèmes liés aux architectures multiprocesseurs. Cependant, il est nécessaire de mettre en œuvre un contrôle complexe capable d'ordonner l'exécution des tâches suivant des contraintes de temps-réel, de consommation d'énergie et de précédence.

En effet, l'exécution de chaque tâche doit être conditionnée par ses dépendances de contrôle et de données. Dès lors, toutes les tâches sont indépendantes et accèdent de manière exclusive en écriture pendant toute la durée de leur traitement. Elles échangent des informations uniquement via les mémoires partagées avec les autres tâches. De plus, chaque tâche doit pouvoir accéder à plusieurs résultats produits par des tâches antérieures différentes.

Par conséquent, il faut décrire explicitement les dépendances de contrôle ou de données et décomposer toutes les applications en un ensemble de processus indépendants. A leur tour, chacun de ces processus doit être découpé en un ensemble fini de tâches dont l'exécution ne nécessite pas de synchronisation avec d'autres tâches. La taille minimale des tâches peut également être soumise à des contraintes particulières nécessaires au bon fonctionnement du système. Tout ceci introduit un nouveau modèle de programmation qui est détaillé dans le chapitre suivant.

Par ailleurs, pour améliorer les performances et le déterminisme du système, des mémoires locales et la migration entre les tâches doivent être utilisées. Ceci s'accomode parfaitement avec le modèle de mémoire choisi, mais nécessite malgré tout une grande réactivité du contrôle afin de ne pas trop pénaliser les performances. Par la suite, un effort particulier sera porté en vue d'améliorer l'occupation des ressources et la prédictibilité du système. Plus les ressources de calcul sont utilisées et plus les durées d'exécution des applications sont réduites ou la taille du système diminuée.

3.2 Présentation et caractéristiques de l'architecture SCMP-LC

L'architecture SCMP-LC est un multiprocesseur asymétrique composé d'un contrôleur matériel. Elle a la particularité d'avoir un espace mémoire physiquement distribué et logiquement partagé. Elle facilite également la préemption et la migration des tâches. Les sous-sections suivantes présentent son principe d'utilisation, le modèle de tâche utilisé et le modèle d'exécution.

3.2.1 Couplage et principe d'utilisation

L'architecture SCMP-LC est vue comme un co-processeur par le système d'exploitation. Le système d'exploitation continue de supporter toutes les opérations de contrôle liées aux calculs ne nécessitant pas une forte accélération. Par exemple, il prend en charge l'exécution des interfaces avec l'utilisateur, ou des processus simples et périodiques. Les processus complexes nécessitant une forte puissance de calcul sont exécutés par notre contrôleur matériel appelé *OSoC*. Ceci permet de disposer d'un contrôleur efficace et de réserver les ressources contrôlées par l'OSoC aux processus critiques. L'architecture OSoC sera présentée dans le chapitre suivant.

Plusieurs modes de couplages sont possibles avec le processeur [170]. Le premier effectue les échanges par l'intermédiaire d'un bus d'entrées/sorties. C'est le couplage en mode périphérique. Le second intègre directement le co-processeur au processeur principal. Il est alors situé sur le chemin de données interne du processeur, au même titre qu'une unité entière ou flottante. Le dernier mode, appelé *couplage co-processeur*, connecte le co-processeur au bus local du processeur. Ils partagent alors le même espace mémoire et le co-processeur peut être piloté via des interruptions ou des instructions particulières. Ce mode facilite son intégration tout en maintenant des liens de communication relativement étroits et performants.

L'architecture SCMP-LC utilise un couplage co-processeur pour s'interfacer aisément sur un réseau d'interconnexion standard avec les autres entités de la plate-forme (figure 3.9). Ceci facilite son intégration dans des systèmes existants tout en conservant une latence de communication relativement faible pour ne pas trop pénaliser les performances. Par ailleurs, cette flexibilité lui offre un potentiel commercial important car les développements nécessaires pour son intégration sont élémentaires.

Il suffit que chacune des entités distantes dispose de registres partagés et d'une interruption pour échanger toutes les commandes et les informations nécessaires au contrôle des applications. Par ailleurs, ce couplage distant permet de limiter les efforts de développement pour les processus non-critiques. Seules les applications nécessitant une forte puissance de calcul pourront être portées sur l'architecture SCMP-LC. L'architecture SCMP-LC peut être associée avec n'importe quelle solution de plate-forme et système d'exploitation propriétaire.

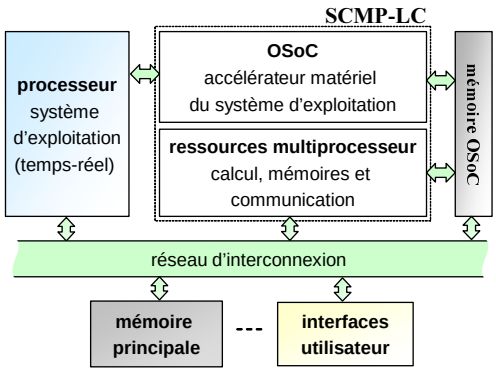

FIG. 3.9 – Couplage et utilisation de l'architecture SCMP-LC. Elle intègre un accélérateur de système d'exploitation temps-réel appelé OSoC. Ce dernier contrôle dynamiquement les multiples ressources de calcul et l'exécution des calculs intensifs. Ses mécanismes de communication avec le système d'exploitation, basés sur les interruptions, contribuent à faciliter son intégration dans n'importe quel réseau d'interconnexion standard. Des liens directs peuvent être établis avec le processeur, ou toutes les communications peuvent avoir lieu via le medium utilisé par le système.

Le processeur peut demander l'exécution d'applications intensives sur l'architecture SCMP-LC en fonction des besoins de l'utilisateur (figure 3.10). Par conséquent, l'OSoC doit supporter dynamiquement la création, la suspension, le réveil ou l'arrêt d'une application. Lors de la reprise d'une tâche périodique, celle-ci doit se synchroniser automatiquement sur sa période initiale d'activation. De plus, de multiples applications et plusieurs instances d'une même application doivent pouvoir s'exécuter concurremment, par exemple, si l'utilisateur souhaite effectuer une vidéoconférence avec plusieurs personnes simultanément. Finalement, l'OSoC doit être en mesure de permettre au processeur de contrôler l'exécution des applications, ainsi que le respect de leurs contraintes temps-réel.

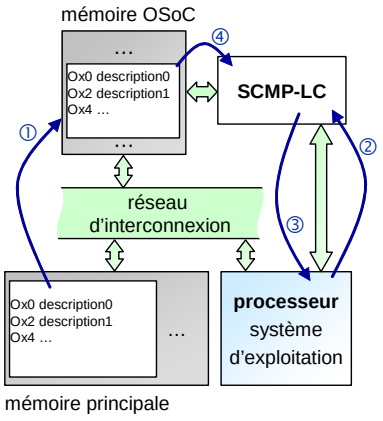

FIG. 3.10 – Couplage entre le système d'exploitation et l'architecture SCMP-LC. Pour exécuter une nouvelle application, le système d'exploitation doit transférer toutes les données et instructions nécessaires dans la mémoire de l'OSoC (1). Lorsque le transfert des informations est terminé, le SE envoie une demande d'exécution à l'OSoC (2). Il donne alors l'identifiant de l'application et son numéro d'instance. Après avoir retourné un acquittement au processeur (3), l'OSoC charge les informations nécessaires et demande l'exécution de l'application (4). A la fin du traitement, un message est envoyé au processeur pour l'en informer.

3.2.2 Caractérisation d'une tâche

L'exécution de chaque tâche doit comporter une étape de configuration, d'acquisition, de traitement et de transmission des données comme le décrit la figure 3.11. Les blocs d'acquisition et de transmission permettent de dialoguer avec le réseau. Ils s'occupent de l'alimentation de la tâche en données et de la restitution des résultats. Leur complexité varie suivant les besoins de la tâche et les capacités du réseau. Par exemple, la tâche peut accéder à des données au début de son exécution, effectuer un traitement et transmettre les résultats vers un élément de mémorisation. Une tâche peut encore nécessiter continuellement des synchronisations avec un élément producteur ou une interface de données. De même, elle peut effectuer les traitements et la restitution des données de manière itérative.

Dans ce dernier cas, il est indispensable que la tâche comporte des mécanismes d'attente et de synchronisation si le réseau n'en dispose pas. La prise en compte des caractéristiques du réseau doit être faite au moment de l'extraction des tâches afin de garantir le bon fonctionnement des communications. En d'autres termes, plus la complexité des blocs d'acquisition et de transmission est importante, et plus la description du contrôle est simplifée.

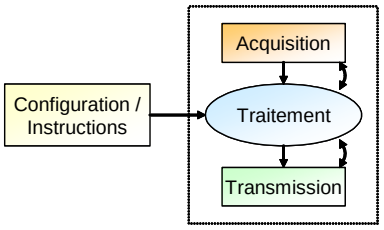

FIG. 3.11 – Caractérisation d'une tâche. Chaque tâche peut être découpée en 4 parties principales : la configuration, l'acquisition, le traitement et la transmission.

Le bloc de traitement a en charge l'exécution d'instructions non conditionnées par des évènements extérieurs comme les cœurs de boucles, ou des ensembles de traitements séquentiels. Enfin, le bloc de configuration correspond à l'étape initiale de configuration de la ressource de calcul. Il permet d'accéder à toutes les informations utiles à leur exécution, comme l'adresse de la tâche à exécuter dans le cas d'un processeur, ou le bitstream de configuration pour une ressource reconfigurable.

Durant son exécution, une tâche ne peut pas accéder à des données autres que celles qui lui ont été destinées lors de l'étape d'extraction. En fait, l'exécution d'une tâche ne peut accéder qu'à des données produites par une des tâches dont l'exécution a déjà eu lieu. Elle respecte l'ordre d'exécution établi par l'élément de contrôle. Par ailleurs, les informations liées aux tâches ou les données produites doivent pouvoir être contenues dans les mémoires réservées pour chacune des tâches.

D'autre part, dans les applications embarquées, les échéances sont liées à des critères applicatifs, comme la fréquence d'affichage ou le débit des données. Il est donc nécessaire d'associer à un ensemble de tâches une contrainte temporelle. Ceci est différent de l'approche habituellement utilisée qui associe à chaque tâche un niveau de priorité ou une échéance relative sans relation avec les autres tâches.

Enfin, les tâches doivent pouvoir s'exécuter sur des ressources de nature hétérogène. Pour cela, le partitionnement des tâches sur les différentes ressources doit être effectué au moment

de la compilation. En effet, il n'est pas envisageable de disposer pour chaque tâche d'un code équivalent pour tous les types de ressources de calcul, afin de maintenir une bonne efficacité transistor. Ainsi, chaque tâche peut être exécutée sur n'importe quelle ressource du moment qu'elle est identique à la cible de départ. Il n'y a donc pas de partitionnement matériel/logiciel au cours de l'exécution.

Cette section a présenté les principes d'utilisation de l'architecture SCMP-LC. Nous avons ensuite défini les caractéristiques des tâches exécutées par l'OSoC. Nous allons maintenant tenter de comprendre comment elles s'exécutent et étudier le *modèle d'exécution* de notre architecture. Un modèle d'exécution tend à donner une représentation fonctionnelle de l'exécution du système et s'attache plus précisément aux moyens mis en œuvre.

3.2.3 Modèle d'exécution

Notre modèle d'exécution est basé sur une exécution séparée du contrôle et des calculs. Il est lié à l'asymétrie de notre architecture. De plus, l'extraction de toutes les synchronisations et des dépendances entre les tâches permet de n'exécuter que des tâches indépendantes. Pendant son exécution, une tâche ne peut pas accéder à des données non prévues au moment de la compilation. Son exécution ordonnée l'empêche d'utiliser des données en cours de production. L'exécution de ces tâches non-bloquantes peut commencer dès que toutes ses opérandes sont disponibles. Les problèmes de cohérence n'existent donc pas. De même, l'utilisation d'un espace de mémorisation physiquement distribué et logiquement partagé garantit l'intégrité des données. Ceci simplifie grandement le modèle d'exécution et facilite la préemption et la migration des tâches.

Ce modèle s'apparente à un réseau de *processus de Kahn* particulier [171]. Dans ces réseaux, l'information transite par des canaux unidirectionnels à lecture bloquante et écriture non bloquante. Ce mécanisme de synchronisation est représenté par des FIFO de taille infinie qui permettent une communication asynchrone. Il impose une chronologie identique aux données produites et consommées par les composants du système. Pour plusieurs raisons, notre modèle d'exécution est différent de celui proposé par G. Kahn, mais garde son déterminisme d'exécution. Ainsi, au lieu d'utiliser des FIFO non bornées, des mémoires standards sont utilisées pour transférer les données. Les lectures ne sont donc pas destructives et plus d'un seul consommateur peut accéder à un même canal de communication.

En fait, notre modèle d'exécution est du type producteur/consommateur. Il n'induit aucune limitation s'il y a autant de mémoires que d'ensembles de données à produire ou à consommer. Toutefois, des écritures exclusives sont possibles dès que chaque producteur possède au moins une mémoire particulière. Autrement dit, chaque tâche est associée à un seul processeur et il faut au moins une mémoire locale par tâche active. Une gestion dynamique des mémoires maîtrise le taux d'utilisation des ressources de mémorisation. D'autre part, les exécutions et les préemptions peuvent être concurrentes sans conflit d'exécution.

La nature des ressources de calcul ne doit pas avoir d'incidence sur ce modèle d'exécution. Ceci est très important et doit permettre de disposer d'une solution efficace pour gérer l'hétérogénéité. Des mécanismes de configuration particuliers peuvent être mis en œuvre en fonction de la nature de la ressource à configurer. Cependant, cela doit rester transparent pour le contrôleur. Certaines ressources nécessitant des temps de configuration longs peuvent avoir besoin d'une configuration anticipée sur l'exécution afin de ne pas trop pénaliser le système.

Ainsi, notre modèle d'exécution pourra bénéficier d'une configuration des ressources avant leur utilisation pour l'exécution des tâches.

Un certain nombre de dispositions a dû être pris pour garantir et simplifier les accès aux données, la préemption ou la migration des tâches. D'autres sont nécessaires pour prédire efficacement le comportement du système et optimiser l'occupation des ressources de calcul. Pour cela, l'architecture SCMP-LC se base sur un modèle producteur/consommateur qui apporte une solution efficace aux problèmes rencontrés dans les plates-formes composées de multiples resssources de calcul. Nous allons maintenant détailler la partie en charge des calculs intensifs et nous reviendrons, dans le chapitre suivant, sur le contrôleur de notre architecture multiprocesseur.

3.3 Principe de l'exécution

L'architecture SCMP-LC est une architecture multiprocesseur asymétrique conçue pour répondre aux contraintes des systèmes embarqués. Son modèle d'exécution permet de simplifier la mise en œuvre de multiples ressources de calcul. Il facilite la migration, la préemption des tâches et améliore le déterminisme de l'exécution. Ceci permet d'adapter et de répartir la charge des ressources de calcul, afin de mieux respecter les contraintes d'exécution temps-réel. Comme le montre la figure 3.12, le partage explicite du contrôle et du calcul lui confère une structure innovante. Pour mieux comprendre son principe d'exécution particulier, nous allons maintenant détailler son fonctionnement suite à la réception d'une demande d'exécution provenant du système d'exploitation.

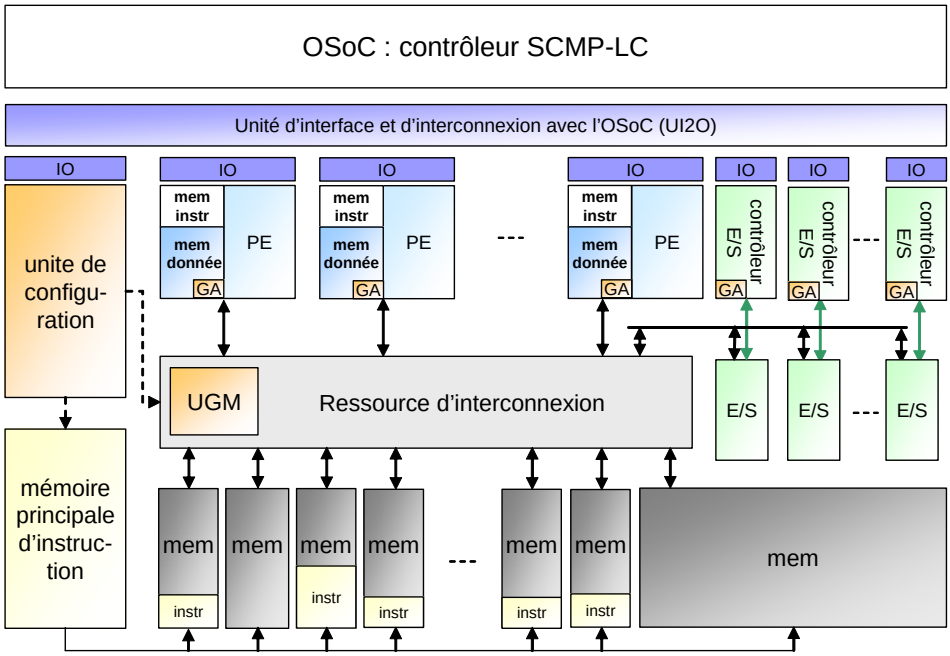

FIG. 3.12 – Structure de l'architecture SCMP-LC. L'unité d'interface et d'interconnexion de l'OSoC (UI2O) assure les communications entre les ressources de calcul et le contrôleur. Pour cela, chaque ressource dispose d'une interface avec l'OSoC (IO). L'unité de gestion de la mémoire (UGM) permet aux générateurs d'adresse (GA) d'accéder directement aux instructions et aux données de la tâche.

3.3.1 Exécution d'une application

Une fois la demande d'exécution reçue par l'OSoC, celui-ci doit effectuer un ordonnancement et configurer les nouvelles tâches actives. Chaque demande de configuration contient l'identifiant de la tâche et donc l'adresse de sa configuration dans la mémoire principale d'instruction. Alors, l'unité de configuration sélectionne une mémoire partagée libre et y pré-charge toutes les instructions nécessaires à la tâche. Puisqu'on dispose d'autant de mémoires qu'il y a de tâches actives, il existe au moins une mémoire libre pour chaque nouvelle demande de configuration. Ces configurations ont lieu séquentiellement, suivant les priorités des tâches et seulement une fois pour toute l'exécution de la tâche. Ainsi, aucun conflit n'est possible lors des accès à la mémoire principale d'instruction.

Une fois que la configuration d'une tâche est terminée, l'adresse de la mémoire sélectionnée et l'identifiant de la tâche correspondant sont écrits dans l'*unité de gestion de la mémoire* (UGM) appartenant au réseau d'interconnexion. Ceci permettra par la suite aux ressources de calcul de retrouver l'adresse physique de la mémoire à partir de l'identifiant de la tâche. Enfin, l'unité de configuration retourne un acquittement à l'OSoC pour lui indiquer la fin du pré-chargement des mémoires partagées.

Lorsqu'une tâche précédemment configurée devient éligible, celle-ci est allouée sur une ressource de calcul. A partir de l'identifiant de la tâche contenu dans la demande d'exécution, celle-ci retrouve l'adresse physique de la tâche grâce à l'UGM. Par la suite, chacune des ressources de calcul accède directement aux informations de la tâche via leur *générateur d'adresse* (GA). Celui-ci peut effectuer les transferts par paquet pour améliorer les échanges d'information. Si la mémoire interne du processeur élémentaire n'est pas suffisante pour contenir toutes les informations, l'acquisition et la transmission des données et des instructions ont lieu au cours de l'exécution. La gestion de ces transferts dépend du type de PE utilisé. Ils peuvent être explicites et décrits avant la compilation, ou être gérés matériellement.

La distribution des générateurs d'adresses et un réseau bien dimensionné permettent des communications concurrentes et des transferts de données entre les tâches. Les contrôleurs d'entrée-sortie sont vus par l'OSoC comme les autres processeurs élémentaires. Si plusieurs tâches doivent accéder en parallèle à une seule mémoire partagée, alors les temps d'exécution des tâches comprennent la latence supplémentaire engendrée. Afin de minimiser ce surcoût, il est possible d'allouer plusieurs mémoires par tâche et de dupliquer les données dans les différents bancs mémoires. Un compromis entre les temps d'écriture et de lecture des données devra alors être déterminé par l'utilisateur afin d'optimiser les performances globales du système.

L'exécution d'une tâche consiste simplement à traiter les données contenues dans ses mémoires locales, puis à stocker le résultat dans une ou plusieurs mémoires accessibles aux tâches suivantes. En effet, toutes les données nécessaires à l'exécution des tâches sont présentes et toutes les synchronisations ont été effectuées avant leur demande d'exécution.

Enfin, en fonction de l'algorithme d'ordonnancement choisi, une tâche en cours d'exécution peut être préemptée. Lors de la préemption d'une tâche moins prioritaire, son contexte d'exécution et les données intermédiaires produites sont sauvés dans la mémoire qui lui a été attribuée. Puisque toutes ces mémoires sont partagées par toutes les ressources de calcul, la migration de la tâche a lieu sans pénalité supplémentaire. Elle est même transparente pour le contrôleur de l'architecture. Par ailleurs, le temps de préemption est constant quel que soit le

nombre de tâches actives, puisque la mémoire de destination est réservée et les ressources de communication garanties. Ce temps inclut uniquement la sauvegarde de la mémoire interne du PE et la latence du réseau d'interconnexion. Par ailleurs, il est indispensable que ce temps soit inférieur à la période d'ordonnancement.

3.3.2 Exécution flot de données

L'exécution d'applications passe aussi par la mise en œuvre de modèles d'exécution flot de données. Il est vrai que le passage par une mémoire partagée pour échanger les informations entre les tâches, et surtout l'exécution des tâches sous des contraintes de précédence, n'est a priori pas adapté pour l'exécution flot de données. Il engendre des accès supplémentaires aux ressources de mémorisation. Pourtant, tous les mécanismes nécessaires sont disponibles pour son exploitation. Et surtout, ce modèle d'exécution général offre beaucoup plus de souplesse et s'adapte davantage à des domaines applicatifs très différents.

Si l'on considère l'application décrite figure 3.13-a, toutes les tâches doivent être activées simultanément, mais doivent s'exécuter suivant un ordre de priorité. En effet, la tâche T_0 par exemple doit être configurée et exécutée avant la tâche T_1 car sans données à consommer, les tâches ne peuvent pas s'exécuter.

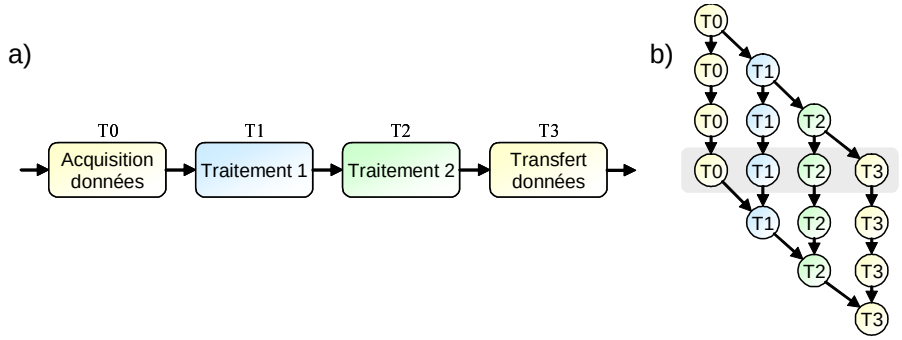

FIG. 3.13 – Exemple d'application flot de données (a) et d'exécution pipelinée (b).

Une solution consiste alors à décrire l'application en s'inspirant de la notion de pipeline logiciel introduite dans le premier chapitre (voir figure 3.13-b). Dans ce graphe d'application, chaque transition est validée par un signal indiquant que la mémoire de destination est pleine, ou que l'ensemble des données produit pour la tâche suivante est prêt. Par exemple, lorsque la tâche T_0 a rempli sa mémoire de destination, elle doit prévenir l'OSoC et se mettre en attente. Alors, la tâche T_1 s'exécute à partir des données produites par T_0. T_0 continue son exécution et à produire de nouvelles données. Dans un premier temps, une partie préliminaire a lieu jusqu'à la configuration de toutes les tâches de l'application. Ensuite, toutes les tâches sont actives simultanément et un ordre d'exécution est établi naturellement entre-elles. Enfin, une partie terminale est exécutée pour terminer l'application et s'assurer que les dernières données produites sont traitées. Le respect de ce modèle permet aux tâches de s'exécuter à des vitesses différentes sans empêcher l'exécution flot de données de l'application.

La figure 3.14 représente l'exécution de l'application décrite précédemment sur un nombre différent de ressources de calcul. Chaque fois que la mémoire de destination est pleine, un

trait vertical est présent pour indiquer une attente de décision de l'OSoC. Remarquons tout d'abord que quel que soit le nombre de ressources, il est toujours possible d'exécuter notre application. Mieux, lorsqu'une tâche plus prioritaire survient (ici la tâche T_n), elle n'empêche pas l'utilisation des autres ressources de calcul et ne bloque pas l'exécution de l'application.

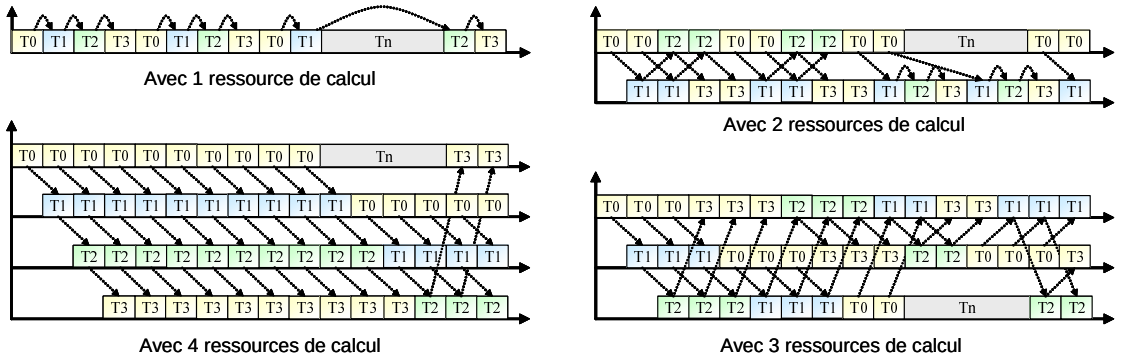

FIG. 3.14 – Principe de l'exécution flot de données sur plusieurs ressources de calcul

Cette troisième section a présenté le principe d'exécution ainsi que la structure de notre architecture. Nous avons vu que notre modèle d'exécution autorise l'exécution flot de données tout comme l'échange de données entre des tâches qui ne s'exécutent pas en parallèle. Il garantit les accès exclusifs aux mémoires et partage l'ensemble des données entre les tâches sans mécanismes de protection particuliers.

3.4 Caractérisation des ressources

Nous allons maintenant caractériser les différents éléments qui structurent l'architecture SCMP-LC. Dans un premier temps, les ressources de calcul puis de mémorisation sont caractérisées. Ensuite, les moyens mis en œuvre pour interconnecter l'ensemble de ces éléments sont discutés afin de garantir la concordance du système avec notre modèle d'exécution. Les ressources de calcul regroupent un très large espace de conception, mais des restrictions majeures doivent être respectées pour les ressources de mémorisation et d'interconnexion. En effet, les contraintes temps-réel et notre modèle d'exécution particulier imposent l'utilisation de solutions déterministes et efficaces.

3.4.1 Ressources de calcul

L'architecture SCMP-LC doit permettre une exploitation efficace de ressources homogènes ou hétérogènes. En fait, la nature de ses ressources ne doit pas avoir d'incidence sur le modèle d'exécution. Pour cela, il faut mettre en œuvre, pour chaque ressource de calcul, une interface permettant le dialogue avec l'OSoC et les ressources de mémorisation. Nous supposerons par la suite que ce dispositif existe pour tous les types de ressources envisagés.

Hétérogénéité

Les ressources de calcul doivent pouvoir être de nature hétérogène. L'hétérogénéité apporte de meilleures performances au système dans le traitement des tâches critiques pour un domaine d'application donné. Ceci offre une meilleure efficacité transistor et énergétique. Cependant, le fait de disposer de plusieurs ressources de calcul identiques permet une meilleure répartition de la charge de calcul et une plus grande flexibilité. La migration n'est possible que si les ressources sont de même nature. Le choix de la plate-forme dépend essentiellement du domaine d'application visé. Pour une utilisation plus généraliste, une solution majoritairement homogène sera préférée.

Les types de ressource

Plusieurs types de ressources de calcul existent : des processeurs monotâches généralistes ou dédiés pour le traitement du signal par exemple, des contrôleurs d'entrées/sorties (DMA : Direct Memory Access, capteurs vidéos...), des accélérateurs matériels dédiés (IP : Intellectual Property), ou des *architectures reconfigurables*. Un circuit reconfigurable peut adapter sa structure, statiquement ou dynamiquement, en fonction des besoins applicatifs [172]. Ceci lui confère une grande souplesse d'adaptation tout en permettant une accélération matérielle significative [173]. Une classification proposée par R. David distingue trois principaux niveaux de reconfiguration (figure 3.15) : le *niveau logique*, le *niveau fonctionnel* et le *niveau système* [174]. Ce dernier niveau pouvant être associé aux processeurs programmables évoqués dans le premier chapitre, nous ne retiendrons que les deux premiers niveaux de reconfiguration.

Au niveau logique Les architectures reconfigurables au niveau logique consistent en une matrice d'éléments configurables reliés par un réseau d'interconnexions. Les architectures FPGA sont les composants matériels à reconfiguration au niveau logique les plus utilisés [175, 176]. L'ensemble de leur structure et des éléments qui les composent peuvent être modifiés pour réaliser un circuit particulier. Ils sont constitués d'un réseau d'interconnexions, d'éléments de calcul élémentaires (LUT : Look-Up Table, logique séquentielle) et de blocs d'entrées/sorties. La reconfiguration consiste à programmer toutes ces ressources configurables afin de réaliser n'importe quelle fonction logique. Par ailleurs, d'autres approches intègrent des éléments de calcul élémentaires multi-fonctions travaillant sur des données de plusieurs bits. A titre d'exemple, nous citons les projets DREAM [177], MorphoSys [178], ou REMARC [179]. Ceci accroît les performances lors des traitements arithmétiques mais offre moins de flexibilité.

Au niveau fonctionnel L'architecture reconfigurable au niveau fonctionnel consiste en un ensemble d'opérateurs arithmétiques, dédiés ou programmables, reliés par un réseau d'interconnexions reconfigurable. A titre d'exemple, il existe les architectures RaPiD [180], XPP [181], DART [182], ou Systolic Ring [183]. La réduction sensible du volume de données de configuration facilite la reconfiguration dynamique. Dès lors, elles sont bien adaptées pour supporter l'exécution de calculs intensifs. De même, elles offrent un potentiel d'efficacité transistor et énergétique important.

Le choix d'une architecture reconfigurable améliore la flexibilité et l'efficacité de l'architecture. Il apporte la possibilité d'utiliser un unique composant pour effectuer différentes fonctions et d'adapter le multiprocesseur aux nouveaux standards numériques. Les solutions

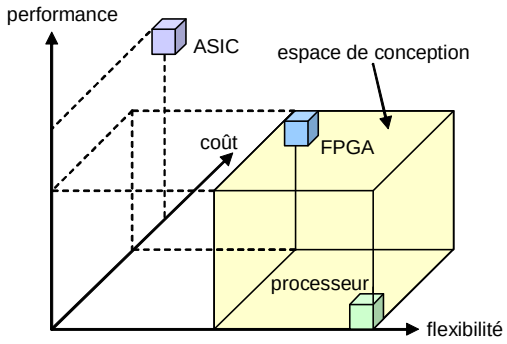

FIG. 3.15 – Espace de conception des architectures reconfigurables [174]. Cet espace peut être défini par trois grandeurs caractéristiques : la performance, la flexibilité et le coût. Les architectures reconfigurables sont des solutions intermédiaires entre les processeurs et les circuits dédiés.

dédiées sont à envisager lorsqu'aucune solution programmable ou reconfigurable n'est suffisamment performante. Elles sont les seules à être capables de s'acquitter de processus critiques nécessitant une forte puissance et densité de calcul. Tous ces processeurs élémentaires doivent être capables d'exécuter des tâches sans contrôle additionnel extérieur, de supporter la préemption et d'accéder à toutes les mémoires disponibles. Par ailleurs, pour limiter la pénalité du contrôle, il est préférable que les temps de préemption n'excèdent pas le temps de réactivité de l'OSoC. Enfin, elles doivent constituer un bon choix en terme d'efficacité transistor et énergétique. Dans cette optique, il faut s'intéresser aux solutions capables de gérer leur consommation d'énergie.

La consommation d'énergie

La réduction de la consommation d'énergie passe par l'utilisation d'opérateurs ayant une bonne efficacité énergétique. Pour cela de nombreuses techniques logicielles et matérielles ont été proposées [184, 185]. Plus récemment, des processeurs capables de fonctionner à de multiples tensions d'alimentation et fréquences de fonctionnement ont vu le jour (DVFS : Dynamic Voltage Frequency Scaling). La possibilité de réduire dynamiquement la tension d'alimentation et la fréquence de fonctionnement est intéressante afin d'adapter la consommation d'énergie du système en fonction des besoins applicatifs. En effet, le fonctionnement du processeur comporte des périodes d'inactivité pendant lesquelles il n'effectue aucun traitement utile mais continue à consommer de l'énergie. Cette technique se révèle très efficace puisque celle-ci est dominée par la consommation dynamique qui est proportionnelle au carré de la tension d'alimentation. Néanmoins, il faut tenir compte de la réduction de la fréquence de fonctionnement qui modifie les durées d'exécution.

A titre d'exemple, les processeurs PowerPC 405LP d'IBM [186], StrongARM-1100 d'ARM [187], lpARM de l'université de Berkeley [188], Crusoe de Transmeta [189], ou XScale d'Intel [190] peuvent être cités. La plupart de ces solutions ont des temps de transition relativement importants qui induisent une pénalité non néligeable sur les temps d'exécution. De plus, ces temps ne sont pas constants pour toutes les transitions. Par exemple, la mise en veille et le réveil du processeur Crusoe nécessite respectivement 10 μs et 25 μs, alors que le changement de fréquence de 300 MHz à 600 MHz dure 300 μs. Ceci empêche de prédire précisément les durées d'exécution si le changement du point de fonctionnement a lieu au cours de l'exécution.

Au contraire, le processeur XScale peut changer de fréquence en 1 μs. Ceci grâce à une mise en œuvre efficace des changements de mode DVFS qui ne nécessite pas de verrouiller à nouveau la PLL (Phase-Locked Loop). Comme l'illustre la figure 3.16, ce processeur supporte de nombreux modes de fonctionnement et adapte ainsi efficacement la consommation d'énergie à la charge de calcul. Nous verrons dans le chapitre suivant comment il est possible de les exploiter.

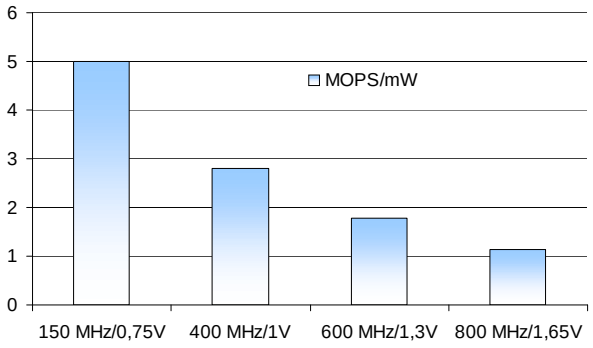

FIG. 3.16 – Evolution de l'efficacité énergétique en fonction des modes DVFS du processeur XScale. L'utilisation d'une fréquence et d'une tension moins élevées améliore l'efficacité énergétique de l'architecture (sources Intel Corporation 2001).

3.4.2 Ressources de mémorisation

Une autre ressource très importante est la mémoire. Notre modèle de calcul est basé sur l'utilisation de la mémoire. Elle est utile pour la mémorisation des données ou des instructions, mais est aussi un passage obligé pour les communications. Elles doivent pouvoir être utilisées par n'importe quelle ressource de calcul. De plus, chaque tâche doit pouvoir accéder à plusieurs résultats produits par des tâches antérieures. L'architecture SCMP-LC dispose de plusieurs types de mémoires : des mémoires très locales intégrées dans les ressources de calcul, l'ensemble des mémoires partagées et distribuées présentées précédemment et la mémoire de l'OSoC utilisée comme interface par le système d'exploitation. Nous avons vu précédemment que l'utilisation de mémoires caches est peu adaptée aux systèmes temps-réel. Ses mécanismes ne permettent pas d'évaluer précisément les durées d'exécution et augmentent significativement les coûts d'intégration et la consommation d'énergie.

La mémoire de l'OSoC

La mémoire de l'OSoC est utilisée par le système d'exploitation pour transférer les instructions et les données nécessaires à l'exécution de nouvelles applications. Dès lors, il est envisageable de segmenter la mémoire en blocs pour diminuer la taille de l'adresse fournie à l'OSoC. Le choix d'un espace segmenté peut conduire à une sous-exploitation de l'espace de mémorisation, mais dans notre cas la mémoire peut être facilement dimensionnée. De même, la fragmentation de la mémoire n'a pas d'incidence particulière sur les temps d'exécution. Au contraire, ceci diminue les temps d'accès puisque l'adressage est simplifié.

Caractérisation des mémoires intégrées dans les ressources de calcul

Les mémoires intégrées dans les ressources de calcul doivent permettre la localité physique des données et des instructions. Ces mémoires sont totalement distribuées et leur contenu ne peut en aucun cas être partagé avec une autre ressource de calcul. Plus sa taille est importante et moins les transferts sont nécessaires avec les mémoires partagées. Néanmoins, plus la quantité de données stockées est grande et plus les temps de sauvegarde du contexte d'exécution sont longs. Ainsi, pour une mémoire disposant d'un temps d'accès moyen de 4 ns, le transfert de 256 mots de 32 bits nécessite environ 1 μs.

Il est difficile de caractériser avec précision la taille de ces mémoires. Elle résulte surtout d'un compromis : plus la mémoire est petite et plus le temps de préemption est faible, alors que plus elle est grande et moins le nombre de transferts de données est important. Pour déterminer la solution optimale, il faut caractériser les besoins applicatifs. Cependant, nous choisirons par la suite une taille de 256 mots pour son temps raisonnable de préemption et parce qu'elle correspond à la taille des motifs de base du traitement d'image.

Caractérisation des mémoires partagées et distribuées

Au cours de l'exécution, chacune des mémoires partagées contient l'ensemble des informations nécessaires pour la tâche qui la possède, puis toutes les données nécessaires aux tâches suivantes. Avant l'exécution des tâches, toutes les mémoires sont pré-chargées par un contrôleur particulier appelé *unité de configuration* (figure 3.17). Le chargement de ces mémoires ne se fait qu'une seule fois lors de la configuration de chaque tâche. La séquentialité du contrôleur impose qu'aucun conflit en lecture ne peut survenir lors de l'accès à la mémoire contenant toutes les appplications. Cette mémoire est nommée *mémoire principale d'instruction*. Au cours de l'exécution, aucun accès à cette mémoire principale n'est possible. Autrement dit, toutes les instructions et la configuration de chaque tâche sont préchargées dans les mémoires associées à chacune d'elles. Si la taille des instructions nécessaires à la tâche est trop importante, plusieurs mémoires doivent être associées à chaque tâche.

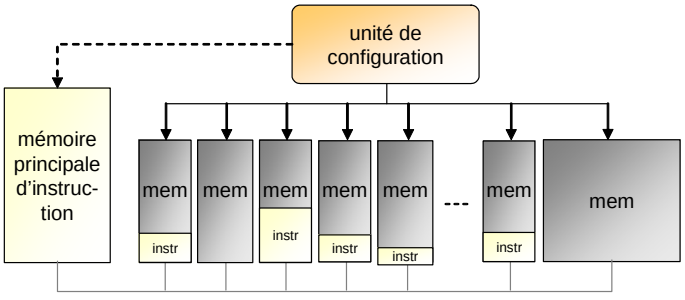

FIG. 3.17 – Modèle de la configuration des mémoires partagées et distribuées. Une unité de configuration charge séquentiellement les instructions nécessaires à chacune des tâches.

La taille et le nombre des mémoires partagées dépendent avant tout du compromis entre la surface, les performances et le parallélisme. Néanmoins, notre modèle d'exécution nous impose de disposer d'autant de mémoires qu'il y a de tâches actives. D'autre part, pour de meilleures performances il est souhaitable que ces mémoires soient des doubles ports pour permettre une lecture et une écriture simultanées. La taille dépend essentiellement des besoins applicatifs et de la granularité minimale des tâches obtenue lors du partitionnement.

A titre d'exemple, nous considérons la compression de flux vidéo qui est un domaine applicatif nécessitant une importante quantité de données. Ces applications découpent les images à traiter en imagettes appelées *macroblocs*. Ces ensembles de données ont une taille inférieure à 1 Koctet. Par conséquent, un système composé de multiples mémoires de données dont la taille varie entre 1 K à 4 Kmots semble suffisant. Pour accéder à un espace d'adressage plus important il est possible d'envisager l'intégration d'un contrôleur de mémoire externe, ou de plusieurs mémoires capables de stocker des images entières ou d'importantes quantités d'informations. Ces mémoires devront également disposer de mécanismes de protection par verrou pour la modification des variables globales. Enfin, des mémoires capables d'adapter leur taille au besoin applicatif doivent éventuellement être envisagées [191]. Ceci permettrait de limiter la consommation d'énergie et de mieux répartir l'espace de mémorisation. En effet, pour optimiser l'utilisation des mémoires partagées, la taille de la mémoire allouée doit dépendre des besoins de chacune des tâches.

Le contenu de ces mémoires est conservé jusqu'à ce qu'il soit consommé par une ou plusieurs tâches suivantes. L'unité de configuration a en charge la gestion des mémoires et le respect de l'intégrité de leur contenu. D'autre part, pour permettre une exécution flot de données, des mécanismes doivent être présents pour indiquer l'état d'occupation de la mémoire aux tâches concernées. Lors d'une tentative de lecture d'une mémoire vide, la tâche doit se mettre en attente d'une donnée. De même, lors d'une écriture dans une mémoire pleine, la tâche doit attendre et retourner un signal à l'OSoC. Celui-ci pourra éventuellement préempter la tâche pour permettre une meilleure occupation des ressources et éviter les blocages. Par ailleurs, l'OSoC doit permettre la libération et l'allocation des ressources de mémorisation.

3.4.3 Ressources d'interconnexion

Le réseau d'interconnexion dans une architecture multiprocesseur a un rôle fondamental pour garantir de bonnes performances. En particulier, notre modèle d'exécution repose sur des moyens d'interconnexion performants et déterministes. Des travaux de recherche sont actuellement en cours pour proposer une solution adaptée à nos besoins. Dans cette thèse, nous nous restreindrons à présenter les solutions existantes et à apporter un début de réponse.

L'utilisation de l'OSoC présuppose un certain nombre de particularités au niveau des ressources d'interconnexion. Pour de meilleures performances, le réseau doit ainsi permettre des accès simultanés aux données, ainsi que des accès par toutes les ressources de calcul à toutes les mémoires. Dans tous les cas, le passage par une mémoire partagée est indispensable pour respecter le modèle d'exécution. Ainsi, un réseau reliant directement les ressources de calcul n'est pas adapté. En effet, notre modèle de communication n'autorise pas seulement l'exécution flot de données.

Par ailleurs, toutes les ressources de calcul sont tenues de communiquer avec toutes les autres ressources sans latence indéterminée. En effet, dans le cadre d'une utilisation temps-réel, les latences d'accès doivent être garanties et prédictibles. Les durées des échanges de données doivent nécessairement être majorées et cela, quelle que soit la charge du réseau ou les conditions d'exécution. Ceci est fondamental pour le bon fonctionnement du système en vue de garantir le respect des contraintes temps-réel et le fonctionnement de l'OSoC. Enfin, les communications doivent prendre en compte l'hétérogénéité des ressources et il faut, pour cela, éventuellement envisager des solutions asynchrones.

L'espace de conception des réseaux d'interconnexion est très vaste [192]. Par la suite, nous tenterons de présenter les différentes approches et de discuter leur mise en œuvre dans l'architecture SCMP-LC. La première solution consiste à relier complètement tous les éléments entre eux sans utiliser d'éléments de routage particuliers. Cette approche pleinement connectée est envisageable si le nombre d'éléments à interconnecter n'est pas trop important. Dans le cas contraire, elle peut très vite nécessiter une surface silicium inacceptable et des temps de propagation longs. Au contraire, une seconde solution utilise un unique lien de communication : le bus.

Les bus

Le *bus* est le support de communication le plus simple à mettre en œuvre et, par conséquent, le plus utilisé par les industriels [193, 194]. Il est partagé par l'ensemble des éléments qui composent l'architecture (figure 3.18). Pour communiquer vers un élément distant, il faut réserver le medium pendant toute la durée du transfert auprès d'un contrôleur de bus appelé *arbitre*.

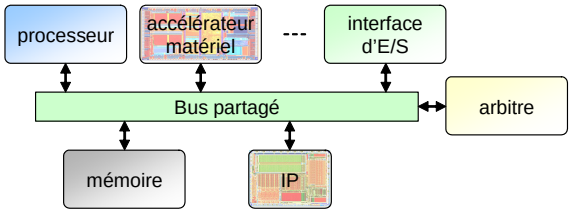

FIG. 3.18 – Modèle avec un bus partagé. Les différents modules interconnectés partagent le medium de communication. La gestion de ce partage est effectuée par un arbitre de bus.

Néanmoins, cette solution présente un certain nombre d'inconvénients. Tout d'abord, le fait de connecter un grand nombre d'éléments entre eux engendre une réduction de la bande passante et des délais de propagation importants. Une solution consiste à segmenter le bus en utilisant des répéteurs, mais ceci augmente encore la latence. Ensuite, les communications ne peuvent pas être concurrentes et nécessitent d'importants mécanismes de synchronisation. De plus, l'utilisation d'un mécanisme d'arbitrage pour échanger des informations introduit du non-déterminisme dans l'exécution. En effet, la disponibilité du bus dépend des communications concurrentes. Enfin, cette solution est limitée par le nombre de ressources à connecter. Elle nécessite des interfaces complexes pour interconnecter des éléments hétérogènes.

Les réseaux par aiguillage

Pour apporter une réponse aux limitations des bus, de nouvelles approches issues des réseaux entre ordinateurs ont été proposées [195, 196]. La première solution consiste à disposer de multiples éléments de routage et à établir un chemin avant toute communication. Ces *réseaux par aiguillage* sont appelés *circuit-switched* (figure 3.19-a). Le principal avantage de ces réseaux est qu'il permet de faire communiquer simultanément plusieurs éléments distants. Il permet également d'augmenter le nombre de ressources pouvant dialoguer entre elles. A titre d'exemple, citons les réseaux aSoC [197] et SoCBUS [198].

Ce mode de communication comporte néanmoins un inconvénient majeur lorsqu'il est sous-dimensionné. Ceci est souvent le cas puisqu'il faut trouver un compromis entre le nombre d'éléments de routage et la flexibilité. En effet, les délais de propagation sont d'autant plus

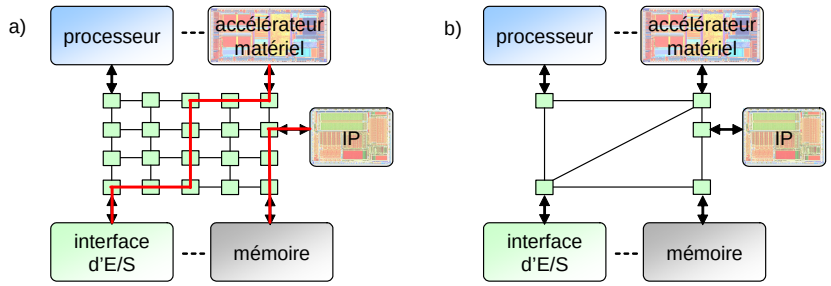

FIG. 3.19 – Les réseaux sur puce. Dans les réseaux par aiguillage (a), de multiples éléments de routage sont réservés pour établir des chemins exclusifs pendant toute la durée de la communication. Au contraire, dans les réseaux par paquet (b), chaque information envoyée contient une entête de routage qui lui permet d'atteindre automatiquement l'élément de destination.

longs qu'il y a d'éléments de routage à traverser. De même, plus il y a d'éléments de routage et plus le nombre de communications simultanées est important.

Pour augmenter les possibilités d'interconnexion, un routage dynamique peut être mis en place. Cependant, il y a toujours, à un instant donné, la possibilité pour qu'une tâche soit en attente de ressources de communication pour continuer son exécution. Le WCET est alors difficilement prédictible et les inversions de priorité ne peuvent pas être évitées.

Par conséquent, ce mode de communication, lorsqu'il est sous-dimensionné, est peu adapté pour les systèmes devant répondre à des contraintes de temps d'exécution. Dans le cas contraire, il devient coûteux et pénalise l'efficacité transistor de l'architecture.

Les réseaux par paquet

La solution suivante regroupe les *réseaux par paquet*, encore appelés *packet-switched* ou *wormhole routing networks* [199]. Ces réseaux sont constitués de nœuds de communication reliant toutes les ressources entre-elles (figure 3.19-b). Ils permettent de recevoir et d'envoyer des messages qui sont constitués d'une entête contenant des informations sur les données associées au paquet. Le nœud doit alors décoder l'entête de chaque message pour savoir s'il lui est destiné. Dans ce cas, son contenu doit être vérifié avant d'être accepté, car il a pu y avoir des pertes d'information dues à des collisions. Dans le cas contraire, le nœud redirige le message vers le chemin approprié ou le propage juste vers son unique voisin. Les réseaux AETHEREAL [200], NOSTRUM [201], Octagon [202, 203] ou SPIN [204] utilisent cette technique de routage. L'avantage est que le nombre de noeuds est inférieur à celui nécessaire au routage dans les réseaux de type *Circuit-Switched*.

Néanmoins, ce mécanisme nécessite l'implémentation de nœuds de communication complexes puisqu'ils doivent être en mesure de décoder et de rediriger les différents messages. Ensuite, les transmissions ne sont pas garanties, à moins d'empêcher les communications simultanées par une implémentation statique et déterministe [205]. Ceci signifie que l'ordre d'arrivée des messages, l'intégrité des informations reçues ou les temps de communication sont indéterminés. Certains réseaux comme le Silicon Backplane Micro-metwork de Sonics [206] ou les réseaux *Time Division Multiple Access* (TDMA), réservent des *time slots* pour éviter notamment les problèmes de collision, de congestion et pour garantir les transmissions.

Ces solutions sont bien adaptées pour les systèmes temps-réel mais nécessitent une synchronisation entre les différents éléments devant communiquer entre eux. Une solution asynchrone

consiste alors à transformer ces *time slots* en *physical slots*. Ce sont des registres ou des nœuds contenant temporairement des messages. Cette solution a été notamment brevetée par le CEA LIST sous la forme d'un réseau en anneau pipeliné dit *Réseau Intelligent Ouvert* ou *à passage de message* [207].

Les réseaux en anneau

L'utilisation d'un réseau en anneau garantit l'envoi de paquets et permet des écritures et des lectures simultanées. Pour ces raisons, ces réseaux sont bien adaptés aux contraintes temps-réel. Cependant, ils induisent plusieurs inconvénients. Tout d'abord, la latence augmente avec le nombre d'éléments constituant le système.

Ensuite, la latence et la priorité d'écriture dépend de l'emplacement de la ressource sur ce réseau. Il est possible de réserver un *physical slot* par élément interconnecté, mais ceci induit une latence trop importante et une sous-occupation du réseau. De plus, il est difficile de réserver plusieurs *physical slots* successifs pour envoyer plusieurs paquets d'information.

Enfin, lorsqu'une mémoire est connectée au réseau, la fréquence du réseau est contrainte par les temps d'accès de cette mémoire. Dans ce cas, ce réseau convient mieux aux applications dont les temps de traitement sont longs par rapport aux communications. En fait, quelle que soit la structure du réseau choisi, le fait de mettre des mémoires directement sur le réseau engendre des dégradations importantes des performances, car il faut bien sûr toujours considérer le pire cas. Ceci conduit à un sur-dimensionnement de l'architecture et donc à une dégradation de son efficacité transistor.

Proposition possible pour l'architecture SCMP-LC

L'architecture SCMP-LC est composée de plusieurs réseaux d'interconnexion. Le premier autorise les communications entre le contrôleur et les ressources de calcul, le second entre les ressources de calcul et les mémoires partagées et enfin le troisième entre la mémoire principale d'instruction et les mémoires partagées.

Pour le chargement des instructions provenant de la mémoire principale, un simple bus suffit puisque le contrôle garantit l'accès à la mémoire lors de leur configuration. Un mécanisme d'arbitrage supplémentaire n'est par ailleurs pas nécessaire.

D'autre part, pour communiquer avec l'OSoC, un réseau en anneau peut se révéler intéressant. En effet, il met en place simplement un réseau performant et de complexité matérielle réduite. De plus, ses inconvénients n'ont pas d'incidence majeure sur les temps de calcul. Néanmoins, d'autres solutions sont envisageables et peuvent parfaitement convenir pour dialoguer entre les ressources de calcul et l'OSoC.

Enfin, la solution à mettre en œuvre entre les ressources de calcul et les mémoires est celle qui a le plus d'incidence sur les performances. Il existe un grand nombre de possibilités pour faire communiquer plusieurs ressources de calcul entre-elles. Et alors même qu'une solution est choisie, un nombre considérable de topologies différentes peuvent être envisagées. Pourtant, il a été montré que seulement certains types de réseaux peuvent être considérés pour répondre aux contraintes de notre système. Les solutions retenues doivent être indépendantes des fréquences de fonctionnement des différents éléments et ne pas consommer d'énergie en cas de non-utilisation. Par ailleurs, elles doivent être adaptées aux contraintes d'exécution temps-réel.

Par conséquent, des solutions pleinement connectées ou des réseaux par aiguillage bien dimensionnés conviennent à nos besoins particuliers. De même, l'utilisation d'un réseau segmenté reconfigurable proche de celui proposé dans DART [182] peut se révéler intéressant. Des solutions existantes ont démontré la faisabilité de tels réseaux. Par exemple, le réseau d'interconnexion du processeur Niagara [75] constitué de 8 cœurs offre un débit théorique de 134 Go/s. La complexité est similaire à celle de l'architecture SCMP-LC puisque chaque cœur a accès à la mémoire de n'importe quel autre coeur. Dans le cadre de ces travaux de thèse, nous ne nous intéresserons pas plus précisément à la mise en œuvre de telles techniques.

3.5 Synthèse

Dans la première section de ce chapitre, nous avons identifié les problèmes liés aux architectures multiprocesseurs. Alors, différentes solutions ont été proposées et ont conduit à la définition de notre architecture SCMP-LC. Puis, l'examen de son principe d'exécution et la caractérisation des différentes ressources qui la composent ont apporté une description détaillée de son fonctionnement.

En fait, le système présenté repose sur un modèle d'exécution simple, puisqu'il consiste à exécuter des tâches indépendantes. Nous avons vu que ceci confère à notre modèle de nombreux avantages et simplifie l'exécution des tâches. Mieux, il offre une solution à l'exécution d'applications sur de multiples ressources de calcul hétérogènes. Ceci repose principalement sur une mise en œuvre séparée du contrôle.

Ce modèle d'exécution n'a jamais été réalisé pour plusieurs raisons. Tout d'abord, il comporte un certain nombre de limitations. La parallélisation des traitements sur plusieurs ressources de calcul n'est pas possible. Il faut pouvoir décrire explicitement les dépendances de contrôle et de données. D'autre part, les accès séquentiels et contraints aux données restreignent les exécutions concurrentes. Enfin, il repose sur la présence d'un fort parallélisme de tâches et donc sur une extraction efficace du parallélisme.

Ensuite, il nécessite un contrôle particulièrement performant et réactif. En effet, sa position centralisée peut en faire le principal facteur limitatif. Les solutions de contrôle existantes n'offrent pas la possibilité d'assurer un niveau de parallélisme suffisant, ou la même réactivité qu'une solution matérielle.

Cependant, ce modèle devient aujourd'hui intéressant parce que les systèmes embarqués doivent pouvoir exécuter simultanément un nombre important d'applications concurrentes. Et finalement, il est dorénavant efficace de partager l'exécution des applications sur de multiples ressources de calcul au lieu de paralléliser les tâches. Par ailleurs, la conception d'un accélérateur matériel de système d'exploitation peut constituer une réponse efficace au problème du contrôle. Dans cette optique, nous avons conçu une architecture capable de gérer toutes les ressources réparties, ainsi que de distribuer dynamiquement les tâches sous des contraintes de temps-réel et de consommation d'énergie. Ce contrôleur matériel appelé OSoC est décrit dans le chapitre suivant.

Chapitre 4

L'architecture OSoC

Sommaire

Le chapitre 3 a détaillé le modèle d'exécution de notre architecture multiprocesseur. Il a mis en exergue son principe d'exécution qui apporte une réponse efficace aux problèmes de synchronisation, d'échanges de données et d'exploitation du parallélisme. Ceci est notamment obtenu par des mémoires physiquement distribuées et logiquement partagées. En effet, le modèle d'exécution simplifie le partage des données entre des ressources hétérogènes et autorise l'exécution concurrente de tâches indépendantes.

Mais l'usage de ces mémoires nécessite une gestion efficace des dépendances et de l'ordonnancement des tâches. Pour garantir la cohérence et l'intégrité des données, chaque traitement doit être contraint par rapport aux autres. Une tâche ne peut s'exécuter que si son exécution n'est pas interrompue par une dépendance de contrôle ou de données. Ainsi, notre modèle

d'exécution repose sur une description et une gestion explicite des précédences d'exécution et des synchronisations.

Alors même que ces contraintes semblent limiter la programmabilité et les performances du système, elles simplifient par la suite l'exécution des tâches. Elles apportent une solution efficace pour répondre au problème des synchronisations, tout en facilitant la préemption et la migration des tâches. Ceci optimise l'occupation des ressources de calcul et adapte la charge de calcul aux besoins de l'utilisateur. Enfin, la maîtrise des temps d'exécution et du déterminisme offre un modèle de calcul adapté aux contraintes temps-réel.

Cependant, tout ceci n'est possible que si les ressources constituant l'architecture sont adaptées à nos besoins particuliers. Par exemple, le réseau d'interconnexion doit être correctement dimensionné pour garantir une bande passante suffisante et une latence constante. Mais avant tout, ce modèle d'architecture repose entièrement sur la disponibilité d'un contrôle dynamique et réactif. Pour que sa position centralisée ne limite pas les performances du système, le contrôleur doit engendrer peu de pénalités temporelles et énergétiques. Le chapitre 2 a bien démontré l'intérêt d'utiliser un accélérateur matériel de système d'exploitation pour améliorer les performances du contrôle, mais a aussi fait état des limitations des solutions existantes.

Ainsi, ce quatrième chapitre propose une architecture de contrôle adaptée aux architectures multiprocesseurs asymétriques. Cette architecture appelée *Operating System accelerator on Chip (OSoC)* met en œuvre une sélection dynamique des tâches à exécuter en fonction des dépendances de contrôle et de données. De ce fait, elle assure les accès exclusifs aux ressources partagées et répartit la charge de calcul entre ses ressources. D'autre part, elle intègre un ordonnancement en ligne global qui garantit le respect des contraintes de temps et minimise le temps global d'exécution des tâches non-temps-réel. Elle réduit également la consommation d'énergie en exploitant les mécanismes basse-consommation des ressources de calcul. Enfin, le procédé d'allocation des tâches gère les ressources hétérogènes et maîtrise les mécanismes de préemption et de migration.

4.1 Présentation et caractéristiques

Comme le montre la figure 4.1, l'architecture OSoC est constituée de sept éléments fonctionnels distincts. Le premier nommé *Control Interface* (CI) assure les communications avec le processeur qui supporte l'exécution du système d'exploitation. Le CI permet par ailleurs la configuration de l'OSoC et le chargement des nouvelles tâches à exécuter. Pour cela, il doit intégrer une interface de communication compatible avec le protocole de communication utilisé par le bus du système. Dans le cadre de ces travaux de thèse, des mécanismes basés sur les interruptions ont été mis en œuvre.

Le deuxième appelé *Task Execution and Synchronization Management Unit* (TSMU) est l'unité de sélection des tâches à exécuter. Elle s'occupe de la gestion des dépendances de données et de contrôle et assure les préséances d'exécution pour n'autoriser que des tâches indépendantes à s'exécuter. Elle garantit également l'accès aux ressources partagées et prévient les interblocages. Pratiquement, cet élément interprête directement des graphes flot de données et de contrôle (*CDFG : Control Data Flow Graph*) qui regroupent l'ensemble des informations nécessaires à l'élection de tâches indépendantes. Ce choix a été notamment motivé par la réutilisation partielle dans l'OSoC d'une architecture développée par notre laboratoire exploitant directement des CDFG. Plus exactement, cette architecture est appelée *Reconfigurable Architecture for the Control* (RAC) [208].

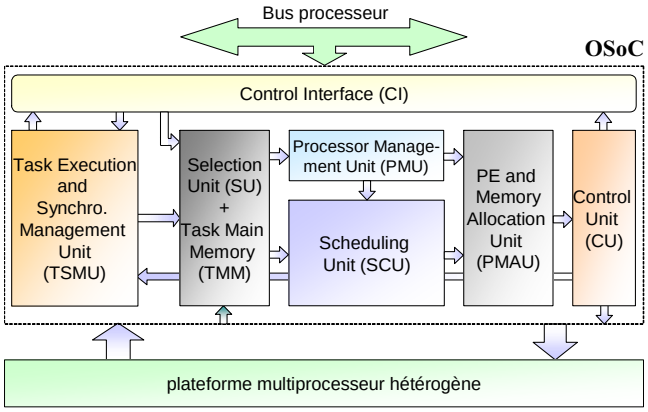

FIG. 4.1 – Détail de l'architecture OSoC

Le troisième désigné *Selection Unit* (SU) synchronise les demandes du système d'exploitation, du TSMU, des ressources de calcul et des ressources de mémorisation. Il contient notamment une mémoire (TMM : Task Main Memory) qui détient toutes les informations des tâches nécessaires à leur ordonnancement et à leur allocation. Celle-ci est configurée par le CI au cours de l'exécution en fonction de l'activation des tâches par le TSMU. La synchronisation des informations reçues par l'OSoC est importante car elle évite des prises de décisions inutiles. Par exemple, si une tâche a terminé son exécution, sa préemption devient superflue.

Le quatrième dénommé *Processor Managment Unit* (PMU) prend en compte les demandes du système d'exploitation. Il est activé avant tout nouvel ordonnancement et modifie au cours de l'exécution l'état des tâches d'une application en fonction des besoins de l'utilisateur. Par exemple, une demande de suspension d'une application a pour effet de forcer ses tâches dans un état d'attente.

Le cinquième nommé *Scheduling Unit* (SCU) ordonnance les tâches en fonction des demandes du système d'exploitation, des tâches actives et de l'état des ressources. Il modifie les états et les priorités des tâches et s'occupe de la sélection des tâches les plus prioritaires. Cet élément regroupe les principaux dispositifs utiles à la gestion des tâches et à la maîtrise de la consommation d'énergie.

Enfin, les derniers éléments appelés *PE and Memory Allocation Unit* (PMAU) et *Control Unit* (CU) assignent les tâches sur des ressources de calcul et retournent les informations d'ordonnancement à l'interface de contrôle et au TSMU. Cette unité d'allocation décide de la préemption et de la migration des tâches en fonction de la liste des tâches triées obtenue par le SCU.

L'ensemble des éléments constituant l'architecture de contrôle OSoC réalise toutes les fonctions nécessaires à l'exécution des tâches sur une architecture multiprocesseur. Pour la première fois, les étapes de sélection, d'ordonnancement et d'allocation sont accélérées ensembles matériellement. Ceci permet à l'OSoC de se passer complètement du système d'exploitation pour l'exécution des applications critiques et contribue à optimiser l'occupation des ressources de calcul en vue d'améliorer les performances. Néanmoins, ceci nécessite la mise en œuvre d'un nouveau modèle de programmation qui sera étudié dans la section suivante.

4.2 Modèle de programmation

Les accélérateurs matériels étudiés dans la littérature exploitent un modèle de programmation similaire à celui utilisé par le système d'exploitation. Au contraire, notre modèle d'exécution particulier et l'utilisation d'un *système d'exploitation matériel* nécessitent un *modèle de programmation* particulier. Un modèle de programmation définit toutes les étapes de la programmation d'un système. En pratique, plus celui-ci est proche des solutions couramment utilisées et plus il est facile d'utilisation. En outre, la compatibilité avec des modèles de programmation existants réduit les efforts de portage des applications déjà conçues pour des systèmes différents.

D'après notre modèle d'exécution, nous avons besoin d'extraire le contrôle et le parallélisme de tâches. Les dépendances de contrôle et de données doivent être explicitement décrites pour permettre à l'OSoC de n'exécuter que des tâches indépendantes. De plus, notre système d'exploitation n'est pas capable de modifier les relations entre les tâches au cours de l'exécution, donc tout défaut d'exécution doit être analysé et corrigé avant la compilation. Enfin, le modèle de programmation doit offrir toutes les informations nécessaires à l'OSoC pour l'ordonnancement des tâches, comme la durée d'exécution ou les contraintes de temps d'exécution dans le cas de tâches temps-réel.

4.2.1 Des réseaux de Petri...

La programmation de l'OSoC est liée à la description des applications et des dépendances de contrôle et de données. Ainsi, le TSMU impose au reste de l'architecture son modèle de programmation. Il se trouve que l'élément principal qui le compose se programme à partir de *réseaux de Petri* [209]. L'architecture utilisée, qui sera présentée plus en détail par la suite, dispose d'une structure matérielle capable d'exécuter ces graphes sans transformations majeures.

Un réseau de Petri est constitué d'états et de transitions. Un état est associé à une action particulière qui peut par exemple être une tâche. Un état en cours d'exécution est identifié par un unique jeton. Une transition est une condition logique permettant le passage du jeton vers les états suivants. Par exemple, dans une divergence en ET, la validation de la transition par un évènement extérieur a pour effet de transmettre en parallèle le jeton à tous les états suivants. Les réseaux de Petri ont la particularité d'être bien adaptés à la description du contrôle et offrent la possibilité de décrire avec efficacité le parallélisme d'exécution et de contrôle. Ainsi, ces graphes décrivent explicitement toutes les dépendances de contrôle et de données qui existent entre les tâches, les processus et les applications.

L'OSoC doit conserver un modèle de programmation proche de celui initialement utilisé. Par conséquent, les dépendances et les synchronisations entre les tâches seront représentées par la suite sous forme de graphes de tâches. Ce formalisme a l'avantage de décrire avec exactitude comment les tâches interagissent entre elles et de permettre une compréhension du comportement des applications, tout en facilitant leur analyse.

Ainsi, la séparation du contrôle et du calcul permet de disposer d'un côté de graphes de dépendance et de l'autre de tâches indépendantes. Par conséquent, l'exécution d'applications sur l'OSoC nécessite une génération de code pour chacune des tâches et une transformation des réseaux de Petri en un code interprétable par l'OSoC. Le code généré dépend du type de

ressource requis pour l'exécution. Le choix de la ressource la plus disposée pour exécuter la tâche doit être fait avant la compilation.

4.2.2 ...à une modélisation haut niveau

En fait, l'utilisation de réseaux de Petri pour la programmation n'est pas habituelle pour les développeurs de logiciel. Pour cette raison, il est souhaitable de mettre en œuvre un niveau d'abstraction supplémentaire pour une description des applications de plus haut niveau.

Néanmoins, ceci nécessite une étape de partitionnement et d'extraction du parallélisme, afin d'identifier les relations de dépendances et l'ensemble des tâches indépendantes. Ceci consiste à former pour chaque application le plus grand nombre possible de tâches dans chaque processus, tout en respectant la nature des dépendances liées au grain de calcul du système. Ce découpage peut se faire suivant les dépendances entre les données, ou suivant les fonctionnalités des différentes tâches. La taille minimale des tâches peut également être soumise à des contraintes particulières nécessaires au bon fonctionnement du système. Les étapes d'extraction peuvent ainsi donner lieu à une représentation différente sans changement de l'ossature du code.

Comme le montre la figure 4.2-a, la conversion d'un langage itératif en un ensemble composé d'un réseau de Petri et de tâches indépendantes est relativement naturelle. Ainsi, la dépendance de données qui lie les tâches $A1$ et $A2$ est représentée sous la forme de deux cellules reliées entre elles par une transition. Ici la transition est la fin de l'exécution de $A1$ car la donnée A est alors disponible. Par conséquent, l'exécution de la tâche $A2$ consiste juste à attendre que la cellule correspondante soit activée et à affecter la tâche indépendante pré-compilée *task_ A2* sur une ressource de calcul.

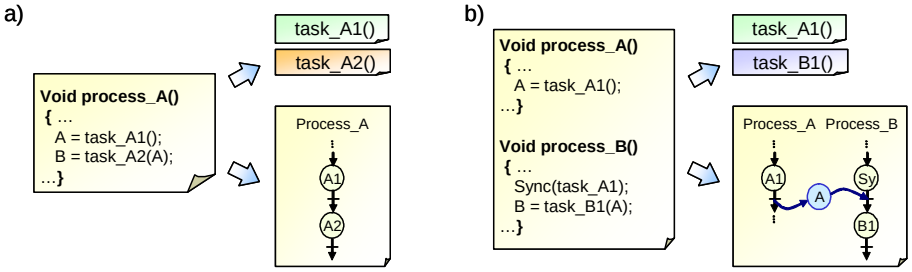

FIG. 4.2 – Modèle de programmation de l'OSoC avec des réseaux de Petri. On extrait d'un côté un réseau de Petri décrivant toutes les dépendances de contrôle et de données et de l'autre des tâches indépendantes. Les dépendances peuvent être décrites au sein d'un même processus (a) ou entre des processus indépendants (b).

Dans un réseau de Petri tous les mécanismes de contrôle peuvent être représentés. Il est possible de décrire des activations conditionnelles des tâches (*if*) ou des boucles d'exécution (*while* ou *for*) grâce à l'utilisation de divergences et de convergences en ET et en OU. Par ailleurs, des synchronisations et des dépendances sont envisageables entre des processus indépendants. Néanmoins, alors que les relations de contrôle et de données sont explicites et de nature séquentielle à l'intérieur des processus, des mécanismes de synchronisation particuliers sont nécessaires entre les processus. Ces dépendances doivent être représentées sous forme de sémaphores et non pas comme de simples liaisons (voir figure 4.2-b).

4.2.3 Synthèse

En fait, le modèle de programmation de l'architecture SCMP-LC peut être défini à différents niveaux d'abstraction (figure 4.3). Depuis le niveau fonctionnel qui ne décrit pas explicitement toutes les dépendances, au graphe de contrôle et de données qui représente tous les liens qui existent entre les processus, les tâches et les applications. Cependant, chaque nouveau niveau d'abstraction nécessite des outils logiciels particuliers pour arriver à une description sous forme de réseaux de Petri.

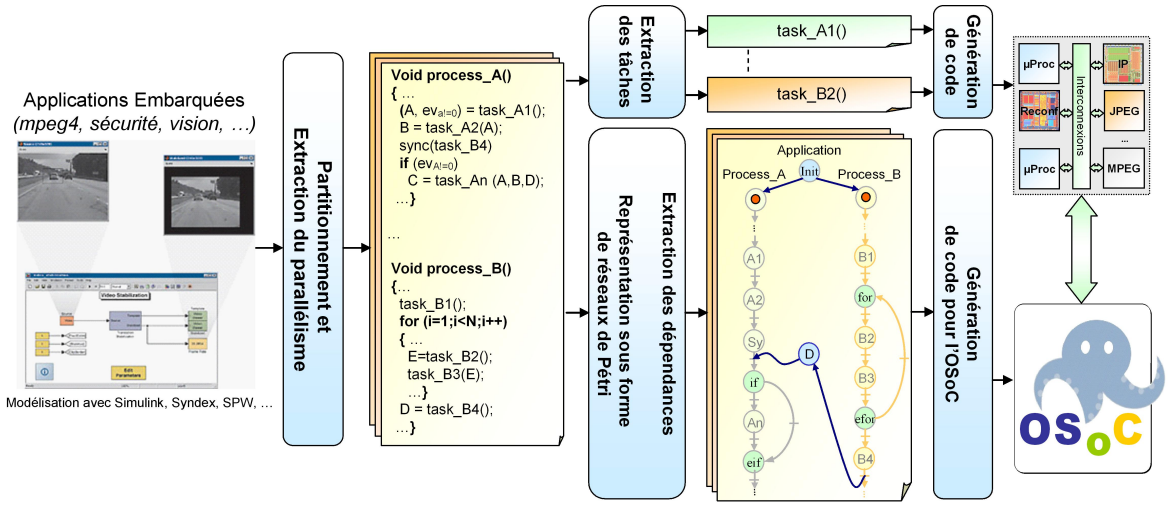

FIG. 4.3 – Les multiples niveaux de programmation de l'architecture SCMP-LC. Différents niveaux de description peuvent être utilisés : du niveau fonctionnel aux réseaux de Petri qui représentent toutes les dépendances de contrôle et de données entre les tâches. Les tâches extraites sont indépendantes et sont compilées pour seulement un type de ressource de calcul.

Le gain apporté par la mise en parallèle du calcul dépend principalement de l'étape de partitionnement. Plus l'extraction de tâches indépendantes est importante et plus l'application peut être parallélisée et accélérée. L'exécution en parallèle de plusieurs applications indépendantes dans les futurs systèmes embarqués est bien adaptée à ce modèle de programmation. Néanmoins, l'étape d'extraction est limitée par la complexité de mise en œuvre des moyens nécessaires à la parallélisation des tâches.

Dans l'état actuel de ces travaux de thèse, seul le dernier niveau de représentation est directement exploitable par notre architecture. Seuls des outils de compilation des tâches indépendantes et de transformation des graphes d'application sont utilisés.

4.3 Gestion des tâches et mise en œuvre du contrôle

Notre modèle d'exécution doit gérer les dépendances entre les tâches pour garantir l'intégrité et la cohérence des données. De ce point de vue, l'élément TSMU (Task Execution and Synchronisation Management Unit) a un rôle très important pour l'exécution de tâches indépendantes. Pour cela, une solution proche du RAC a été envisagée. Elle offre la possibilité de

synchroniser des tâches et de partager des ressources et contribue ainsi à réduire les surcoûts temporels présents dans les systèmes parallèles.

4.3.1 Présentation du RAC

Le RAC permet la mise en œuvre dynamique de réseaux de Petri [210]. Comme le montre la figure 4.4, cette architecture peut exécuter plusieurs applications en parallèle. Pour cela, cette structure matérielle asynchrone dispose d'un ensemble de cellules pour y implanter les différents états des graphes d'application et d'un réseau d'interconnexion complètement connecté pour autoriser une grande flexibilité. Au cours de l'exécution, chacun des graphes d'application est étendu en fonction de la propagation d'un jeton d'exécution et des ressources du RAC disponibles. Cette architecture est dîte auto-adaptable. Une partie de ce travail de thèse a consisté à définir et à mettre en œuvre ces mécanismes [211].

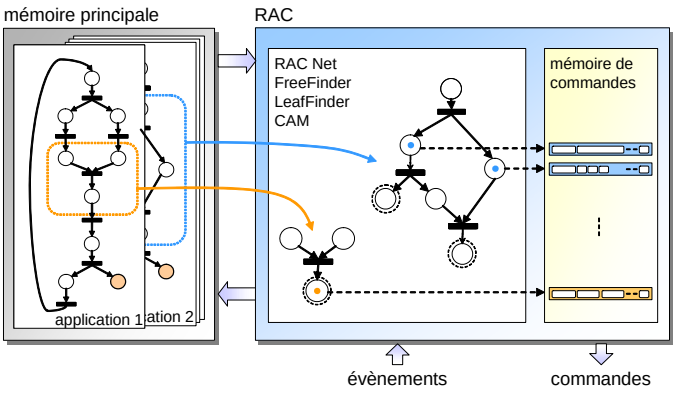

FIG. 4.4 – L'architecture du RAC. La réception des évènements conditionne la propagation des jetons d'exécution dans les différents graphes d'application. Les dernières cellules construites et représentées en pointillé sont des cellules feuilles. Elles permettent l'extension des graphes d'application. La présence d'un jeton d'exécution sélectionne la commande correspondante. Plusieurs flots de commande peuvent être gérés concurremment.

Initialement, la première cellule de l'application dispose d'un jeton d'exécution. La tâche associée à cette cellule est alors exécutée via l'envoi d'une commande aux ressources de calcul. Pour indiquer la fin de son exécution, un évènement est retourné au RAC et le jeton est propagé aux cellules suivantes de l'application. Ainsi, l'activation des tâches des applications a lieu suivant la propagation des jetons d'exécution et la géométrie de leur graphe. En parallèle, le graphe se construit de manière autonome en fonction des ressources disponibles. Tout d'abord, un dispositif identifie les dernières cellules des graphes appelées *cellules feuilles*. A partir de ces cellules, d'autres ressources dans le RAC sont réservées pour étendre les graphes d'application. Eventuellement, des cellules peuvent être libérées si les tâches correspondantes ont déjà été exécutées.

Le principe d'exécution des réseaux de Petri impose que les exécutions des tâches soient conditionnées par les tâches précédentes et par la réception d'un évènement extérieur. Ainsi, le RAC ne délivre à un instant donné que les demandes d'exécution de tâches indépendantes. Toutes les conditions sont donc réunies pour que celles-ci s'exécutent sans perturbation jusqu'à

l'établissement de leurs résultats finaux. De plus, les réseaux de Petri n'introduisent pas de contrainte lors de la description du parallélisme de contrôle puisque la propagation des jetons vers toutes les cellules suivantes se fait simultanément. Les demandes d'exécution des tâches peuvent donc être parallèles. Par ailleurs, il est possible de décrire des convergences ou des divergences multiples et de synchroniser des processus entre eux.

Toutes ces propriétés offrent au RAC, donc à l'OSoC, la possibilité de gérer efficacement un parallélisme de contrôle élevé et l'ordre d'exécution des tâches. Une part importante de la complexité de l'architecture SCMP-LC peut donc être aisément résolue en utilisant une telle architecture. Celle-ci souffre cependant d'un certain nombre de limitations. En effet, elle n'a pas initialement été conçue pour gérer des systèmes temps-réel et des structures de calcul similaires à l'architecture SCMP-LC. La sous-section suivante tente de les identifier et d'apporter une réponse adaptée à chacune de ces limitations.

4.3.2 Modifications du RAC

Pour le temps-réel

Le RAC doit avoir un comportement prédictible et permettre la maîtrise de ses temps d'exécution pour supporter une utilisation sous des contraintes temps-réel. Cependant, dans sa version actuelle [210], la construction des cellules intervient de manière aléatoire et a une durée indéfinie. Pour résoudre ce problème, il faut respecter quatres dispositions complémentaires.

Premièrement, la taille des divergences et des convergences doit être limitée pour majorer le temps de construction. Dans le cas contraire, il faut considérer, lors de chaque extension du graphe, que la cellule feuille ait la possibilité de créer des liens vers toutes les autres cellules. Le pire temps de construction serait donc beaucoup trop important.

Deuxièmement, les demandes d'exécution et de construction doivent être traitées dans l'ordre en fonction de leur date d'arrivée. Pour cela, il faut sélectionner toutes les demandes à un instant donné et empêcher la prise en compte de nouvelles demandes tant qu'elles n'ont pas toutes été traitées. Une mise à jour doit être prévue en cas d'initialisation ou de libération des cellules.

Troisièmement, pour limiter les constructions inutiles, il est préférable d'attendre que les cellules feuilles aient un jeton d'exécution. Dans ce cas, la construction des cellules suivantes a lieu pendant l'exécution de la tâche.

Enfin, il faut être capable d'effectuer des combinaisons d'évènements afin de limiter le nombre de cellules intermédiaires. Pour l'instant, un seul évènement est associé à une cellule. Pour prendre en compte un deuxième évènement lors de la propagation du jeton, il faut ajouter une cellule supplémentaire qui n'active aucune tâche. Avec un dispositif de type PAL (Programmable Array Logic), la construction de chaque cellule utile nécessite au plus la construction d'une cellule intermédiaire.

Une fois toutes ces solutions mises en œuvre, le temps de création d'une cellule (quel que soit le nombre de destruction) peut être borné. Le cas le plus défavorable lors de la construction d'une cellule est obtenu lorsqu'une divergence de taille maximale tente de se construire et échoue lors de la dernière connexion. Alors, il faut annuler les constructions réalisées et attendre la libération de suffisamment de cellules avant de recommencer. D'autre part, il faut prendre en compte le fait que d'autres constructions concurrentes peuvent avoir lieu.

Dans l'état actuel des développements du RAC, la construction d'une cellule nécessite 12 cycles et la libération 26 cycles (soit respectivement 60 et 130 ns à 200 MHz dans le cadre d'une réalisation en technologie CMOS 0,13 μm). Nous avons donc pu estimer le pire temps de propagation des jetons à environ 1,6 μs pour une divergence d'au plus 8 cellules. Pour un fonctionnement transparent du RAC, il faut donc que la durée d'exécution d'une tâche soit supérieure ou égale à ce temps. Dans ce cas, les mécanismes internes du RAC liés à l'exécution des tâches n'ont pas d'incidence majeure puisque l'exécution est concurrente à la construction de la tâche suivante. Il faut en effet seulement 6 ns à une cellule du RAC pour autoriser la propagation des jetons et faire une nouvelle demande de configuration.

Pour la configuration avant l'exécution

L'étude du principe d'exécution de l'architecture SCMP-LC a montré que l'exécution de chacune des tâches nécessite préalablement une configuration. Celle-ci prépare les mémoires partagées et configure le réseau d'interconnexion. Le cas échéant, elle pourrait configurer la ressource de calcul correspondante si des ressources reconfigurables sont utilisées. Toutes ces configurations induisent des coûts non négligeables sur les temps d'exécution. Par conséquent, la possibilité de configurer une tâche pendant l'exécution de ses tâches précédentes permettrait un gain de temps significatif.

L'intégration de cette nouvelle propriété nécessite de légères modifications de la structure actuelle du RAC. Il suffit de mettre en œuvre un mécanisme de *double jeton* pour autoriser la configuration des ressources avant l'exécution des tâches. La réception du *jeton d'exécution* requiert l'exécution de la tâche correspondante, tandis que le *jeton de configuration* envoie une requête de configuration au PMAU afin de préparer une mémoire libre à recevoir les instructions ou les configurations. Le jeton de configuration est propagé aux cellules suivantes lorsque les configurations des cellules précédentes sont terminées et qu'elles possèdent un jeton d'exécution. Le jeton d'exécution est quant à lui transmis lorsque les demandes d'exécution des tâches précédentes ont été validées. L'exécution d'une tâche n'est possible que si la cellule correspondante a été préalablement configurée et qu'elle possède un jeton d'exécution. Par ailleurs, la fin d'exécution d'une tâche a lieu lorsque tous les traitements et tous les transferts de données ont été effectués. La figure 4.5 illustre ce fonctionnement.

Au début de l'exécution d'un graphe d'application, la tâche initiale contient un jeton de configuration et d'exécution. Les demandes d'exécution et de configuration sont prises en compte par *niveau*. Ceci signifie que chaque ensemble de demandes est traité avant d'en acquérir un nouveau. Ceci ne garantit cependant pas que les demandes d'exécution arrivent après les configurations. En effet, si la durée de configuration est supérieure à la durée d'exécution des tâches précédentes, la tâche peut devoir attendre sa configuration. Les jetons de configuration et d'exécution doivent être pris en compte dès leur arrivée quel que soit l'état des cellules. Pour éviter des configurations inutiles ou prématurées, les configurations ne doivent pas avoir lieu en avance lors de convergence en ET et de divergence en OU.

Pour la création dynamique des applications

Le RAC est capable de construire dynamiquement de nouveaux graphes d'application, mais il ne peut pas en créer ou en supprimer au cours de l'exécution. Néanmoins, les modifications nécessaires sont relativement simples à mettre en œuvre. Tout d'abord, il faut intégrer une cellule supplémentaire capable de détruire les jetons de configuration et d'exécution pour permettre la fin de l'exécution des différents graphes présents dans le RAC.

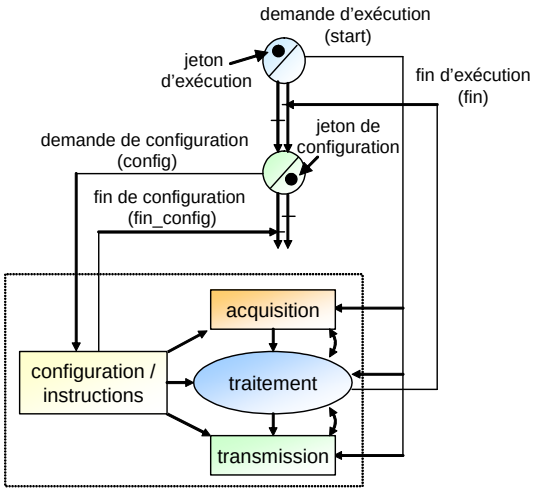

FIG. 4.5 – Gestion de l'exécution et propagation des jetons de configuration et d'exécution. La configuration est requise au début de l'exécution de la tâche précédente. Une cellule peut contenir deux jetons. Le jeton de configuration est propagé vers les tâches suivantes dès que le jeton d'exécution est reçu.

Ensuite, l'identification des tâches doit permettre de dissocier des tâches n'appartenant pas à la même application. En effet, on doit identifier les tâches à supprimer lors de la réception d'une demande de fin d'exécution d'une application. Pour cela, il faut mettre en œuvre un système d'identification des tâches permettant d'associer chaque tâche à une instance d'application.

L'utilisation du RAC existant n'est pas envisageable pour un système temps-réel. Néanmoins, la mise en œuvre de quelques modifications permet d'obtenir un élément de contrôle mieux adapté à nos besoins. En effet, il est capable de prendre en charge la gestion des dépendances entre les tâches, les synchronisations et les préséances d'exécution. Il est particulièrement bien adapté à notre modèle d'exécution. Nous allons maintenant détailler dans la section suivante sa mise en œuvre dans l'architecture OSoC.

4.3.3 Mise en œuvre de la sélection dynamique des tâches

Afin d'être utilisé dans l'OSoC et constituer l'élément TSMU, le RAC doit intégrer toutes les modifications précédemment proposées et ne pas être conservé dans son intégralité. En fait, la mémoire de commandes du RAC est remplacée par un dispositif permettant d'adapter sa structure aux besoins de l'OSoC (voir figure 4.6). L'OSoC requiert des informations sur les caractéristiques des tâches et le RAC ne délivre que des signaux d'état qui indiquent si un jeton d'exécution ou de configuration est présent.

Le RAC fournit des ensembles de signaux appelés *EXEC_ TASK* et *CONF_ TASK*. Le nombre de ces signaux est égal au nombre de cellules qui constituent le RAC. Le signal *EXEC_ TASK* correspond à la présence d'un jeton d'exécution et le signal *CONF_ TASK* à celle d'un jeton de configuration. Chacun de ces signaux pilote une ligne mémoire contenant des informations d'exécution ou de configuration. A titre d'exemple, ces données précisent

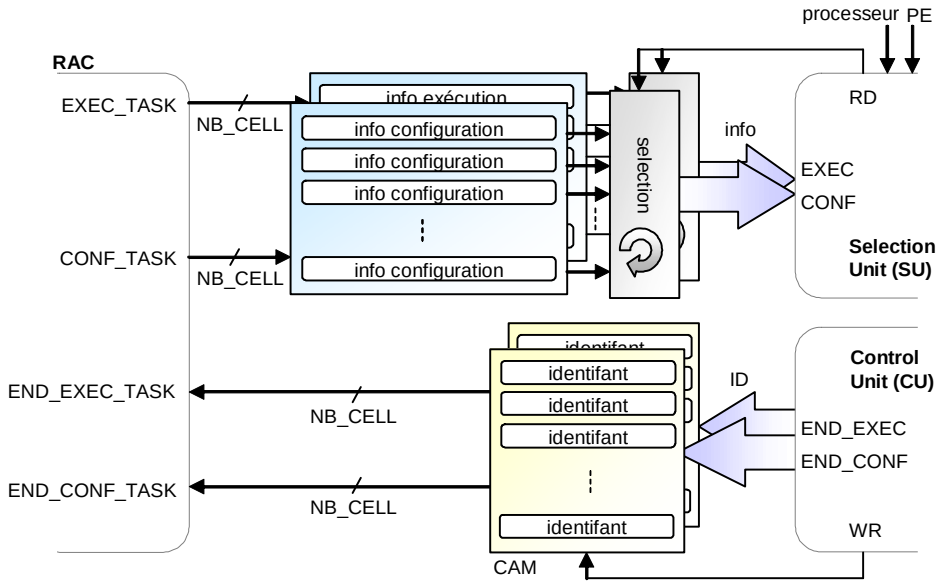

FIG. 4.6 – Intégration du RAC dans l'OSoC. L'élément TSMU est composé du RAC et de dispositifs de mémorisation et de sélection. Ces derniers fournissent aux autres éléments les informations nécessaires pour l'ordonnancement et l'allocation des tâches.

le WCET de la tâche, son échéance, sa périodicité ou le type de ressource associée à son exécution. Elles indiquent également des informations nécessaires à l'identification des tâches et des évènements qui sont associés à leur fin d'exécution.

Pour permettre l'exécution simultanée de plusieurs applications et de plusieurs instances d'une même application, chaque tâche ou évènement est associé à un identifiant particulier. Un mot de 16 bits est utilisé à cette fin. Il contient l'identifiant de l'application (4 bits), le numéro d'instance de l'application (4 bits) et l'identifiant de la tâche de l'application (8 bits). Ceci lui permet d'exécuter 256 applications en parallèle, ou encore 65 536 tâches différentes pour une taille limitée de l'OSoC. Une taille d'identifiant supérieure permettrait d'augmenter encore les possibilités de l'OSoC. Ainsi, l'ordonnancement, l'allocation et l'exécution des tâches utilisent toujours ce principe d'identification. Chaque demande de configuration ou d'exécution contient l'identifiant de la tâche.

A chaque nouvel ordonnancement, le signal *RD* est envoyé par le SU pour lire séquentiellement chaque nouvelle demande de configuration ou d'exécution. Une demande est valide lorsque le signal provenant de l'OSoC active la ligne mémoire correspondante. C'est l'occasion de configurer les autres éléments de l'OSoC avec les informations recueillies. A la réception d'une fin de configuration *END_CONF* ou d'une fin d'exécution *END_EXEC* par le SU, celui-ci transmet ces évènements via le CU au RAC. Chacun d'entre-eux est envoyé séquentiellement avec le signal *WR* en utilisant une mémoire CAM. Celle-ci permet de lier l'identifiant reçu à la cellule correspondante du RAC. La propagation des jetons de configuration et d'exécution a alors lieu et les prochaines tâches à configurer ou à exécuter sont activées.

4.3.4 Conclusion

Cette troisième section a détaillé la mise en œuvre de la sélection des tâches. Nous avons vu que l'utilisation du RAC, moyennant quelques modifications, parvient à répondre à une grande partie des contraintes liées à notre modèle d'exécution. Il gère :

- les dépendances entre les tâches ;
- les accès aux ressources partagées ;
- le parallélisme de contrôle ;
- les traitements non-déterministes ;
- et la cohérence et de l'intégrité des données.

De plus, l'utilisation d'un deuxième jeton pour les configurations devrait limiter la pénalité de chargement des mémoires partagées. Néanmoins, cette étape de sélection des tâches n'est pas suffisante pour respecter des contraintes temps-réel et optimiser l'occupation des ressources. En effet, si l'on dispose d'un nombre de ressources de calcul limité, il faut faire un choix entre les tâches à exécuter. Pour cela des étapes d'ordonnancement et d'allocation doivent être mises en œuvre. Elles seules contribuent au respect de nos contraintes d'exécution et à l'obtention de bonnes performances. La section suivante présente les mécanismes d'ordonnancement utilisés dans l'OSoC.

4.4 Ordonnancement des tâches

L'ordonnancement des tâches détermine, parmi un ensemble de tâches prêtes à être exécutées, celles qui doivent être exécutées en premier lieu. La décision peut reposer sur de nombreux critères. Pour cette raison la littérature propose de multiples solutions. Le choix de l'algorithme d'ordonnancement a des incidences directes sur les performances et doit donc faire l'objet d'une étude approfondie. En accord avec les deux premiers chapitres, nous nous intéresserons plus particulièrement aux techniques d'ordonnancement de tâches non-temps-réel, temps-réel ou limitant la consommation d'énergie sur de multiples ressources de calcul identiques. La gestion de l'hétérogénéité de notre architecture de calcul sera évoquée dans la section 4.5.

4.4.1 Ordonnancement non-temps-réel multiprocesseur

Le problème de l'ordonnancement de tâches sur de multiples ressources de calcul est connu pour être NP-complet [212]. Seules des heuristiques tentent d'approcher la solution optimale conduisant à un temps d'exécution minimal.

Les caractéristiques de l'architecture de calcul ont un impact important sur le choix de l'heuristique. Dans le cadre de notre étude, nous considérons uniquement des ressources de calcul identiques, des tâches indépendantes et des communications non bloquantes. Par ailleurs, on suppose que les durées des tâches et le nombre de ressources de calcul sont connus avant l'exécution. La possibilité d'exécuter des tâches sur un nombre variable de processeurs, encore appelées *tâches malléables*, n'est pas autorisée malgré le gain en performance qu'il est possible d'obtenir [213, 214, 215]. En effet, ceci suppose une nouvelle compilation de la tâche, ou d'avoir à disposition de multiples codes pré-compilés pour chacune des tâches. Le coût

induit par les ressources de mémorisation serait trop important. Tous ces choix sont donc en accord avec les propriétés de l'architecture SCMP-LC présentée précédemment.

Etat de l'art

Le problème de l'ordonnancement de *tâches non-malléables* pour ces modèles d'architecture a été largement étudié dans la littérature [216]. Les solutions peuvent être regroupées en quatre catégories : les algorithmes de *recherche aléatoire*, les algorithmes d'*agrégation*, les algorithmes de *duplication* et les algorithmes de *liste*.

Les algorithmes génétiques sont les algorithmes de recherche aléatoire les plus utilisés. Ils explorent l'espace des solutions possibles afin de retenir celle qui engendre un temps d'exécution minimal. Ce sont certainement les solutions qui obtiennent aujourd'hui les meilleurs résultats. Cependant, leur temps d'exécution et la complexité matérielle nécessaire pour les mettre en œuvre sont significativement plus importants que les autres approches d'ordonnancement [217, 218]. De plus, leur gain en performance ne justifie pas leur utilisation comparé à des algorithmes plus simples à mettre en œuvre, à savoir les algorithmes de liste [219, 220].

Les algorithmes d'agrégation regroupent des tâches entre elles en fonction de leur granularité et des communications. Ils déduisent ensuite le nombre optimal de PE nécessaires à l'exécution de chaque groupement [221]. Cette méthode donne de bons résultats. Cependant, la constitution d'agrégats se révèle difficile et coûteuse à mettre en œuvre.

De même, les algorithmes de duplication peuvent produire de très bons résultats mais induisent d'importants besoins en ressources matérielles [222]. Ces algorithmes dupliquent les tâches en favorisant l'exécution, sur les mêmes ressources de calcul, des tâches possédant des relations de dépendances de données. Ceci limite les communications et les temps d'attente, mais le nombre de ressources de calcul augmente avec le nombre de prédécesseurs possibles.

Enfin, la simplicité et la rapidité des algorithmes de liste en font des méthodes d'ordonnancement bien adaptées pour une mise en œuvre matérielle. En effet, même si ces heuritiques peuvent sembler moins performantes en simulation, le surcoût en temps nécessaire à l'ordonnancement est négligeable comparé aux autres solutions. Un ordonnancement de liste se décompose en deux parties distinctes. Dans un premier temps, une liste triée est construite selon une grandeur caractéristique. Puis, chaque tâche est ordonnancée suivant l'ordre imposé par la liste sur un ou plusieurs PE.

Les algorithmes de liste

Ainsi, contrairement aux autres solutions, les algorithmes de liste peuvent être aisément réalisés. Des comparaisons et éventuellement des permutations doivent être mises en œuvre pour établir une liste triée. En fait, le problème de l'ordonnancement de tâches est similaire à celui rencontré dans l'emballage optimal (*bin-packing*) largement étudié dans la littérature. Certaines des approches proposées supposent un nombre illimité de ressources de calcul, des exécutions sans contrainte de précédence, ou par exemple des durées d'exécution identiques pour toutes les tâches à ordonnancer [216, 223, 224, 225].

D'autres, plus proches de notre cas d'étude, atteignent de bonnes performances, mais nécessitent des algorithmes de complexité importante [226]. Par exemple, Jansen et al. considèrent l'ordonnancement de n tâches indépendantes afin de minimiser le temps total d'exécution [227]. Ce modèle d'approximation polynomial suppose que chaque tâche peut être exécutée avec un seul PE et optimise une fonction de coût induite par l'ensemble des tâches. Dans [228], Topcuoglu et al. proposent deux algorithmes : le Heterogeneous Earliest-Finish-Time (HEFT)

et le Critical-Path-On-a-Processor (CPOP). Dans l'algorithme HEFT, les priorités des tâches dépendent du temps restant à chaque tâche pour terminer leur exécution. L'algorithme CPOP minimise un chemin critique en fonction des coûts associés à chaque tâche, les coûts étant de nature quelconque. Ces approches nécessitent de connaître le temps courant d'exécution pour chacune des tâches, alors même que des tâches non-temps-réel doivent pouvoir s'affranchir d'une telle complexité. Par ailleurs, la réévaluation dynamique des critères d'ordonnancement induit un surcoût temporel et une augmentation de la consommation d'énergie due aux accès constants aux ressources de mémorisation.

Finalement, ces dernières années, ces heuristiques se sont de plus en plus complexifiées dans le but d'obtenir de meilleurs résultats, sans prendre en considération la faisabilité de leur mise en œuvre matérielle. Il est important de savoir se limiter en complexité et en optimalité, afin de proposer des heuristiques d'ordonnancement adaptées. Un simple tri de tâches suivant la durée d'exécution (*LTF : Longest Task First*), ou le nombre de ressources de calcul nécessaire à l'exécution des tâches (*LATF : LArgest Task First*) [229], peut se révéler intéressant.

D'autre part, de nombreuses heuristiques ont été proposées, mais peu de techniques d'allocation ont été évaluées. Pourtant, leur efficacité a un effet important sur l'ordonnancement des tâches [230]. La principale solution proposée consiste à paralléliser les tâches (tâches malléables). Cependant, une allocation dynamique peut aussi apporter une amélioration significative sur l'ordonnancement, tout en répartissant la complexité algorithmique. Une méthode d'allocation dynamique peut assigner des tâches en accord avec la disponibilité des ressources au cours de l'exécution.

L'algorithme LLD

Ainsi, ces travaux de thèse ont conduit à proposer un algorithme d'ordonnancement de liste appelé *Level-by-level and Largest-task-first scheduling with Dynamic-resource-occupation* (LLD) [231]. Si ce type d'ordonnancement est partiellement introduit dans [229], la principale innovation réside dans sa stratégie d'allocation qui améliore sensiblement les performances.

Dans un premier temps, cette heuristique trie de manière décroissante les tâches selon le nombre de PE nécessaire à leur exécution. Si l'on considère $\delta(T_i)$ comme étant le nombre de PE nécessaire à l'exécution de la tâche T_i et $\tau(T_i)$ sa durée d'exécution, alors le tri d'une liste C de n tâches indépendantes telles que $C = \{T_1, T_2, ...T_n\}$ est tel que $\delta(T_1) \geq \delta(T_2) ... \geq \delta(T_n)$. Par ailleurs, si $\delta(T_i) = \delta(T_j)$ et $\tau(T_i) \geq \tau(T_j)$, alors T_i est exécutée avant T_j.

Dans un deuxième temps, en accord avec la disponibilité des ressources, la tâche la plus prioritaire de la liste triée est allouée sur les ressources de calcul. Ce processus est répété jusqu'à ce que l'allocation de toutes les tâches de la liste ait été effectuée. La complexité de l'algorithme ainsi obtenu est du type $O(n \times P)$, où P est le nombre de ressources de calcul.

La figure 4.7 présente un exemple d'ordonnancement LLD (a), ainsi que le résultat obtenu en utilisant l'algorithme FCFS (First Come First Serve) qui ordonnance les tâches suivant leur date d'arrivée (b). Après l'exécution de la première tâche T_0, la liste C des tâches prêtes à être exécutées, est composée de $\{T_1, T_2, T_3\}$. Avec l'algorithme LLD, la tâche nécessitant le plus de ressources de calcul est prioritaire. Par conséquent, la tâche T_1 est ordonnancée puisque l'on dispose de suffisamment de ressources pour l'exécuter. Les tâches suivantes de la liste de tâches courante doivent alors attendre la fin de l'exécution de T_1.

Le manque de politique d'ordonnancement peut induire un ordonnancement moins efficace, comme le montre la figure 4.7 (b) avec un exemple d'ordonnancement FCFS. L'ajournement de l'exécution de la tâche T_1 empêche l'éligibilité des tâches suivantes et l'utilisation optimale

des ressources. Pour mieux s'en rendre compte, nous allons comparer différents algorithmes de liste avec notre heuristique LLD.

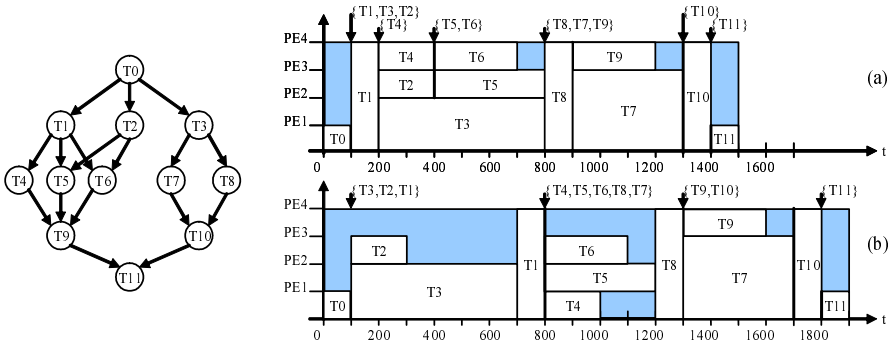

FIG. 4.7 – Ordonnancement de tâches non-temps-réel avec l'algorithme LLD. L'ordonnancement du graphe d'application sur 4 ressources de calcul identiques peut donner lieu à des résultats différents. La durée d'exécution obtenue avec un algorithme *FCFS* (b) est 30 % plus longue que celle obtenue avec un algorithme *LLD* (a).

Comparaison des algorithmes de liste

Cette étude comparative évalue les performances de différents algorithmes ayant une complexité algorithmique et de mise en œuvre matérielle équivalente à l'algorithme LLD [231]. Elle estime également l'efficacité de l'association d'une méthode d'allocation dynamique à différents algorithmes de liste. Ces derniers comportent l'extension *DRO* pour *Dynamic Resource Occupation*. Chacun des algorithmes comparés est présenté par la suite.

L'algorithme LTF favorise l'allocation des tâches dont les durées d'exécution sont les plus longues [215]. D'autre part, l'algorithme STF (Shortest Task First) consiste à trier les tâches suivant les durées d'exécution les plus courtes. Ensuite, l'algorithme *SLEAT* ordonnance et alloue itérativement toutes les tâches nécessitant plus de $\frac{P}{2}$ PE [232]. Ensuite, les tâches restantes sont triées suivant leur durée d'exécution, en créant deux groupes de $\frac{P}{2}$ PE contiguës. Enfin les tâches sont ordonnancées soit sur le premier groupe de PE, soit sur le deuxième, en favorisant le groupe de PE dont la durée d'exécution totale courante est la plus faible.

Les algorithmes proposés par Coffman et al. [233] sont au nombre de trois : le *Next-Fit-Decreasing-Height* (NFDH), le *First-Fit-Decreasing-Height* (FFDH) et le *First-Fit-Increasing-Height* (FFIH). Ils définissent des *niveaux* qui représentent temporellement des étapes possibles d'ordonnancement. L'algorithme NFDH alloue la prochaine tâche prête à être exécutée sur les ressources les plus à gauche sur le niveau courant. S'il n'y a pas suffisamment de ressources, le niveau est fermé et un nouveau est créé à la fin de l'exécution de la plus longue tâche du niveau courant précédent. Au contraire, l'algorithme FFDH ne ferme pas les niveaux. Enfin, l'algorithme FFIH propose de trier préalablement la liste des tâches à ordonnancer en favorisant les durées d'exécution les plus courtes. Ceci diminue les performances de l'ordonnancement, mais permet de réduire les temps de réponse moyens des tâches dans le but d'améliorer la réactivité du système.

Cependant, cette comparaison ne permet pas de garantir l'efficacité de notre solution, car il est impossible d'avoir une idée de ses performances par rapport à des algorithmes plus élaborés. Pour cette raison, nous les comparons également aux algorithmes *IDEAL* et *OPTIMAL*. L'algorithme IDEAL ordonnance les tâches en les compactant le plus possible, sans respecter

le nombre de ressources utilisées ou leur parallélisme. Même si cet algorithme n'a aucune signification réelle, il représente la limite haute que pourrait atteindre n'importe quelle heuristique. Au contraire, l'algorithme OPTIMAL n'est pas une heuristique. Il évalue toutes les possibilités d'ordonnancement et ne garde que la solution qui aboutit au temps d'exécution le plus court. Bien sûr, puisque le problème est NP-complet, cet algorithme nécessite un temps très long d'exécution. Le nombre de tâches qu'il est possible d'ordonnancer est limité. En pratique, nous n'avons pu évaluer cette solution que pour un nombre de PE inférieur ou égal à 8.

Afin d'évaluer la qualité de chacun de ces algorithmes, un nombre important de listes de tâches indépendantes a été généré aléatoirement. Chaque liste correspond à un niveau d'exécution créé par le *TSMU*. Nous considérons que $\tau(T_i) \in [1..100]$, que $\delta(T_i) \in [1..P]$ avec $P = \{4, 8, 16, 32\}$, et que le nombre de tâches indépendantes de chaque liste est de 30 (ceci correspond au nombre de tâches que peut renvoyer simultanément la structure du RAC). Une étude statistique basée sur une distribution uniforme garantit la généricité du comportement des différents algorithmes. Par ailleurs, le nombre de PE choisi est une puissance de deux afin de pouvoir effectuer la comparaison avec l'algorithme SLEAT. Enfin, nous définissons l'accélération $\lambda(H)$ d'une heuristique H, en comparaison avec une absence de politique d'ordonnancement (FCFS). Pour l'ordonnancement d'une liste C de tâches indépendantes, cette valeur est donnée par l'équation (4.1).

$$\lambda(H) = \frac{FCFS(C) - H(C)}{FCFS(C)} \tag{4.1}$$

Ainsi par exemple, l'accélération $\lambda(LLD)$ de l'exemple présenté figure 4.7 atteint 21% et le taux d'occupation des PE est de $86,7\%$ comparé à $68,4\%$ avec un ordonnancement FCFS.

Résultats de l'étude comparative

Comme le montrent les figures 4.8 (a,b), les qualités des solutions STF et LTF sont assez similaires et meilleures que le NFDH. Il est important de noter que l'algorithme LATF a de très bons résultats par rapport à ceux obtenus avec les heuristiques STF, LTF ou SLEAT. Il était attendu que le taux d'occupation des PE soit meilleur puisque le but de cet ordonnancement est de favoriser l'occupation des ressources. L'algorithme SLEAT atteint de bonnes performances mais celles-ci diminuent toutefois avec le nombre de PE. En fait, cette heuristique a un important inconvénient puisque chaque tâche qui utilise plus de $\frac{P}{2}$ PE, est séquentiellement exécutée sans prendre en compte les ressources libres. Par exemple, une tâche utilisant 9 PE sur une plateforme disposant de 16 PE alloue seulement 56% des ressources durant toute l'exécution de cette tâche.

Finalement, trois heuristiques se détachent des autres solutions et se rapprochent des résultats obtenus avec l'algorithme IDEAL. Ainsi, n'importe quel algorithme, dont les performances sont meilleures que ces heuristiques, apporterait seulement une amélioration moyenne de 7% sur le temps total d'exécution. Cependant, plus les résultats espérés doivent s'approcher de l'optimal et plus la complexité algorithmique et matérielle augmente. Ainsi, l'effort dépensé pour atteindre une faible amélioration serait perdu par la latence nécessaire pour générer une solution d'ordonnancement. L'algorithme FFDH atteint de bonnes performances comme [215] le laissait entrevoir. Ce qui est plus inattendu est le résultat obtenu en utilisant conjointement une allocation dynamique et l'algorithme LTF. En effet, avec finalement peu de complexité supplémentaire, l'accélération est augmentée de 10 % avec une allocation dynamique. Enfin, l'algorithme LLD proposé dans ce mémoire atteint les meilleurs résultats.

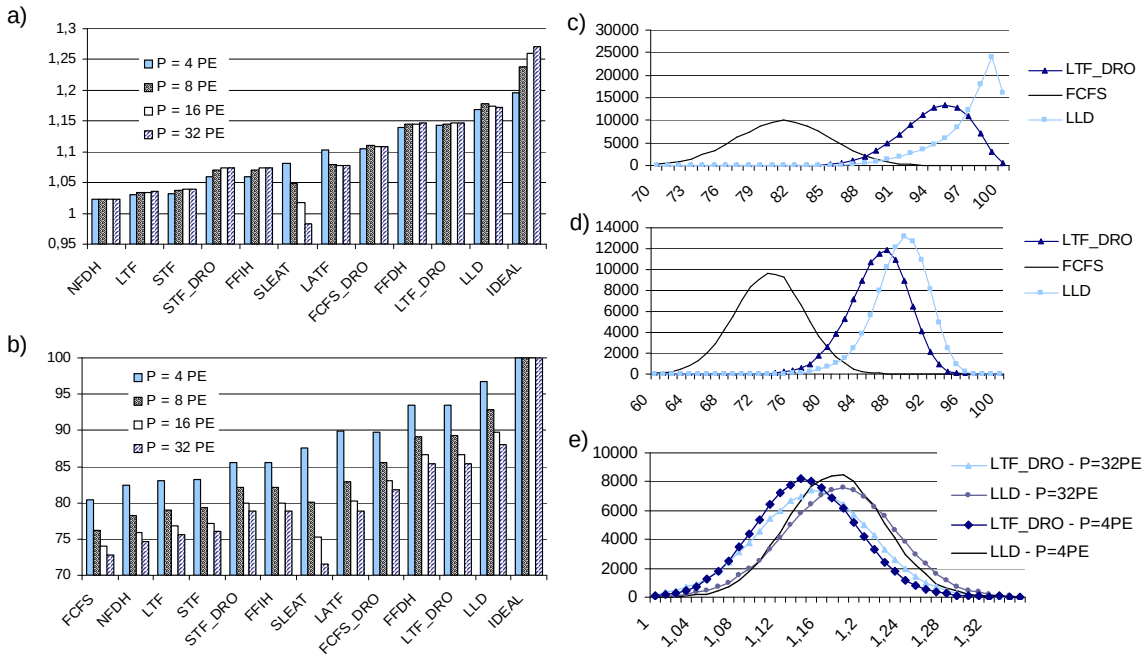

FIG. 4.8 – Comparaisons des algorithmes de liste. Ces résultats présentent : (a) l'accélération moyenne, (b) l'occupation moyenne des ressources de calcul, (c) la distribution de l'occupation moyenne sur 4 ressources de calcul, (d) la distribution de l'occupation moyenne sur 32 ressources de calcul et (e) la distribution de l'accélération moyenne pour 4 et 32 ressources de calcul. Le tirage aléatoire des listes de tâches à trier suit une loi uniforme.

Les figures 4.8 (c,d,e) représentent la fréquence des résultats obtenus. Elles mettent en évidence les très bonnes propriétés obtenues avec l'algorithme LLD et l'intérêt d'utiliser une technique d'allocation dynamique. En fait, seuls les algorithmes possédant une politique d'allocation élaborée atteignent des performances intéressantes. Si maintenant nous nous intéressons à une distribution exponentielle du nombre de ressources nécessaires avec une moyenne μ égale à $\frac{P}{4}$, les performances de l'algorithme LLD restent stables alors que celles des algorithmes LTF_DRO et FFDH augmentent (figure 4.9-a). Ces derniers ont donc encore de meilleurs résultats lorsque les tâches nécessitent moins de processeurs pour s'exécuter. De même, la figure 4.9-b atteste des très bonnes propriétés des algorithmes de liste malgré leur faible complexité, ici représentés par l'algorithme LLD. Par conséquent, les possibles améliorations apportées par n'importe quelle autre heuristique seraient d'un intérêt limité.

Tous ces résultats montrent que le choix du type de tri a un effet significatif sur la qualité de l'ordonnancement et les performances. Malheureusement, notre modèle d'exécution n'autorise pas l'exécution d'une tâche sur plusieurs ressources de calcul. En effet, les communications entre les ressources étant difficiles, l'exécution des tâches en seraient pénalisée. Par conséquent, nous avons choisi de mettre en œuvre dans l'OSoC l'algorithme LTF_DRO pour ses très bons résultats et sa grande simplicité, pour ordonnancer les tâches non-temps-réel. Celles-ci seront exécutées seulement lorsqu'une ressource de calcul sera disponible et qu'aucune tâche temps-réel ne sera en attente d'exécution. L'ordonnancement des tâches temps-réel fait quant à lui l'objet de la sous-section suivante.

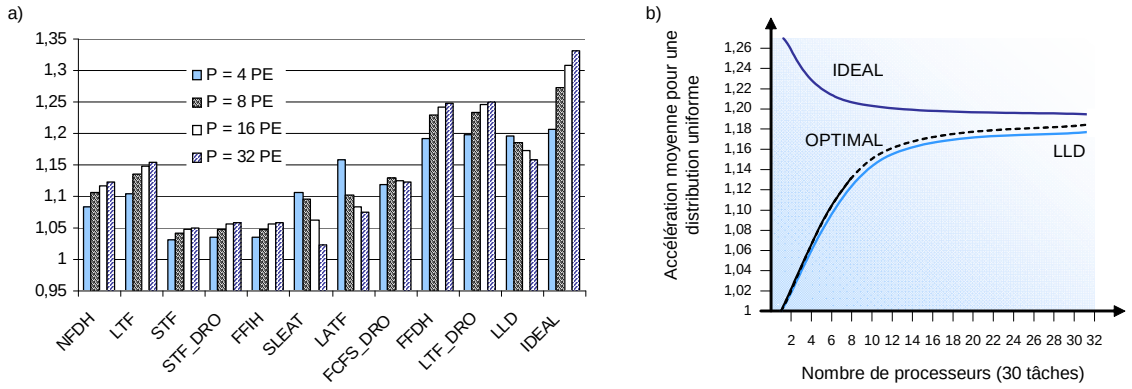

FIG. 4.9 – Comparaisons des algorithmes de liste et mise en évidence de leurs limitations. La figure (a) représente l'accélération moyenne pour une distribution exponentielle (μ=P/4) et la figure (b) l'accélération moyenne de l'algorithme LLD en comparaison avec les algorithmes IDEAL et OPTIMAL.

4.4.2 Ordonnancement temps-réel multiprocesseur

Contrairement à l'ordonnancement monoprocesseur temps-réel évoqué dans le second chapitre (sous-section 2.1.2), nous disposons à ce jour de peu de résultats sur l'ordonnancement multiprocesseur temps-réel.

Deux techniques d'ordonnancement peuvent être distinguées : l'ordonnancement *global* et *par partitionnement* [234, 120] (voir figure 4.10). Il est cependant possible de définir des solutions intermédiaires comme le *semi-partitionnement* [235]. Le lecteur pourra consulter [236] pour une taxinomie complète. Il faut souligner que dans la plupart des cas, ces deux approches sont *incomparables* [237, 50]. Autrement dit, il y a des ensembles de tâches périodiques ordonnançables avec une approche globale pour lesquels aucun partitionnement n'existe. L'inverse est également vérifiable. Ce résultat souligne l'intérêt d'étudier ces deux approches.

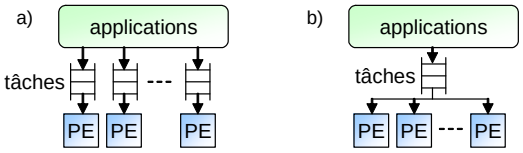

FIG. 4.10 – Ordonnancement temps-réel multiprocesseur. L'ordonnancement par partitionnement associe des ensembles de tâches à chaque processeur élémentaire (PE). Les migrations des tâches sont impossibles (a). Au contraire l'ordonnancement global dispose d'une unique file de tâches éligibles (b). Chaque tâche peut être exécutée sur n'importe quelle ressource de calcul. Au cours de l'exécution, la migration des tâches est possible.

Ordonnancement par partitionnement

L'ordonnancement par partitionnement repose sur la constitution de sous-ensembles disjoints de tâches [124, 123]. Chaque sous-ensemble est ensuite ordonnancé suivant une stratégie d'ordonnancement *locale* monoprocesseur, sur une ressource de calcul qui lui est propre. Les

tâches ne sont alors pas autorisées à migrer d'un processeur à l'autre, quelle que soit la charge de calcul de chacune des ressources. Généralement, ce partitionnement est réalisé hors-ligne.

L'approche hors-ligne semble être la solution au problème d'ordonnancement puisqu'elle permet de le transformer en un ensemble de sous-problèmes monoprocesseurs. En effet, on connait l'optimalité des algorithmes temps-réel dans un environnement monoprocesseur et il est même envisageable d'utiliser des politiques d'ordonnancement en ligne à contrainte temporelle [238].

Cependant, pour un nombre m de processeurs fixé, il est démontré dans [50] qu'il n'est pas possible de garantir un taux d'occupation supérieur à $\frac{m+1}{2}$ quel que soit l'heuristique ou l'algorithme utilisé. En effet, trouver un partitionnement optimal est équivalent au problème de *bin-packing* évoqué dans la sous-section précédente. Seules des heuristiques similaires au problème non-temps-réel peuvent être utilisées pour déterminer des sous-ensembles conduisant à minimiser la durée d'exécution globale. Seulement, avec des tâches temps-réel, il faut considérer le pire cas pour dimensionner le système. Par conséquent, ceci conduit à une sous-exploitation des ressources de calcul.

Par ailleurs, elle induit un ordonnancement des tâches statiques et la charge de calcul ne peut pas être répartie au cours de l'exécution. Par exemple, si un sous-ensemble se termine prématurément, le processeur devenu disponible ne pourra pas décharger les autres processeurs pour réduire la durée globale d'exécution. Pour cette raison, d'autres approches consistent à autoriser la migration des tâches entre les ressources de calcul.

Ordonnancement global

Ainsi, l'ordonnancement global applique une unique stratégie d'ordonnancement [239]. Elle consiste principalement à trier les tâches éligibles et à les allouer suivant leur niveau de priorité sur les ressources de calcul disponibles. Toutes les tâches peuvent être allouées sur tous les processeurs capables de les exécuter. Par ailleurs, si l'algorithme choisi le permet, son utilisation peut demander la préemption puis la migration d'une tâche au cours de l'exécution.

Même si cette approche n'a pas connaissance des exécutions futures pour prendre des décisions, l'allocation peut prendre en compte l'état courant des processeurs pour répartir la charge de calcul. La possibilité de préemption et de migration permet de modifier les choix effectués en fonction de l'arrivée de nouvelles tâches à ordonnancer. Ainsi, cette approche est de toute évidence plus générale que le partitionnement. Surtout, elle offre suffisamment de flexibilité pour adapter dynamiquement la charge de calcul entre toutes les ressources de calcul.

Les algorithmes d'ordonnancement global peuvent être basés sur des priorités statiques, des priorités dynamiques Event-Triggered *(ET, i.e. job-level dynamic)*, ou des priorités dynamiques Time-Triggered *(TT, i.e. fully-dynamic)*. Comme nous l'avons vu dans le second chapitre, l'approche statique ne peut pas convenir à nos besoins applicatifs. Par conséquent, nous nous intéresserons essentiellement aux deux approches dynamiques.

Ordonnancement dynamique ET L'ordonnancement dynamique sur évènement le plus utilisé est l'ordonnancement EDF. Il fait d'ailleurs l'objet de nombreuses études pour évaluer son utilisation sur des architectures multiprocesseurs. Les résultats les plus intéressants ont été obtenus par Carpenter et al. pour un ensemble de tâches indépendantes sur une structure

constituée de m ressources identiques [50]. Ils ont démontré que n'importe quel ordonnancement EDF conduit à un taux d'occupation des ressources compris entre $\frac{m}{2m-1}$ et $\frac{m+1}{2m}$.

D'autre part, Philips et al. ont montré que le surdimensionnement du système peut conduire à une solution ordonnançable [240]. Ainsi, un ensemble de tâches faisables sur m processeurs identiques est ordonnançable avec EDF si chacun des processeurs a une puissance de calcul $(2 - \frac{1}{m})$ fois plus grande que le système d'origine [241]. Mais cette technique *par augmentation* induit une sous-occupation des ressources de calcul et donc une rédution de l'efficacité transistor.

En fait, il n'existe pas d'ordonnancement optimal et cela quel que soit l'algorithme en ligne ET utilisé. Hong et al. ont montré que l'optimalité n'est pas possible à partir de deux ressources de calcul et de deux échéances distinctes [242]. Ceci peut néanmoins être nuancé si l'ordonnancement exploite des paramètres supplémentaires comme la date d'arrivée des tâches, ou si les priorités dépendent de l'écoulement du temps.

Ordonnancement dynamique TT Les ordonnancements par écoulement de temps les plus utilisés sont basés sur les algorithmes LLF et Pfair. Carpenter et al. ont également montré que ces algorithmes sont les seuls à pouvoir ordonnancer les tâches en utilisant toute la puissance de calcul disponible. N'importe quel ordonnancement avec un algorithme dynamique TT conduit à un taux d'occupation des ressources inférieur à m, soit le maximum possible.

Ainsi, le fait d'adjoindre des informations temporelles à l'ordonnancement permet d'envisager des algorithmes optimaux. C'est le cas des algorithmes Pfair [50, 243]. Ces algorithmes exécutent les tâches par partie en allouant une fenêtre d'exécution temporelle de taille identique à chacune des sous-tâches constituant la tâche. Il existe aujourd'hui trois algorithmes d'ordonnancement Pfair optimaux : PF [244], PD [245] et PD^2 [246]. Chacun d'eux applique une politique proche de EDF pour l'ordonnancement des sous-tâches. Malheureusement, même si cette stratégie est optimale, elle nécessite un nombre très important de préemption et conduirait en pratique à une sous-utilisation des ressources de calcul.

L'algorithme LLF peut engendrer potentiellement autant de préemptions mais celles-ci ne sont pas systématiques comme avec les algorithmes Pfair. Par conséquent, le nombre de préemptions est inférieur. Néanmoins cet algorithme n'est pas optimal, même si la durée des tâches, leur date d'activation, ou les échéances sont connues a priori [247]. Les figures 4.11 et 4.12 illustrent à travers deux exemples les problèmes d'optimalité de l'algorithme LLF comparé à EDF [248, 249]. Le dernier exemple montre d'ailleurs l'inexactitude des résultats de Leung [250] qui affirmaient que tout ensemble de tâches ordonnançable avec EDF l'est aussi avec LLF.

Contrairement aux algorithmes Pfair ou EDF, très peu de travaux ont porté sur les algorithmes LLF pour multiprocesseur. S*á*ez et al. ont cependant montré par simulation les très bonnes performances de l'algorithme LLF comparé à d'autres algorithmes dont le EDF [251]. Pourtant, aucun résultat général ne certifie l'intérêt d'utiliser un algorithme LLF.

Anomalies d'ordonnancement et synthèse

En fait, il n'existe pas de solutions parfaites. Il faut peut-être même douter d'en trouver une, tant les recherches dans ce domaine ont été nombreuses depuis plus de vingt ans. Les seules solutions optimales que sont les algorithmes Pfair sont des approches intéressantes en théorie, mais n'ont pas de sens en pratique. Il est néanmoins possible de trouver une solution

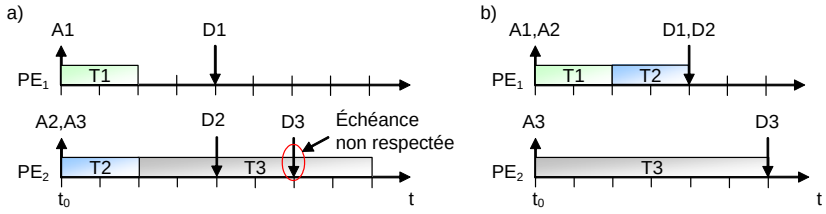

FIG. 4.11 – Problème avec un ordonnancement EDF pour multiprocesseur. Une flêche montante indique la date d'activation d'une tâche et une flêche descendante son échéance. Une tâche T_i est caractérisée par sa date d'activation A_i et son échéance D_i. L'ordonnancement EDF ne permet pas à la tâche T_3 de respecter son échéance (a), alors que l'ordonnancement LLF conduit à une utilisation optimale des ressources (b).

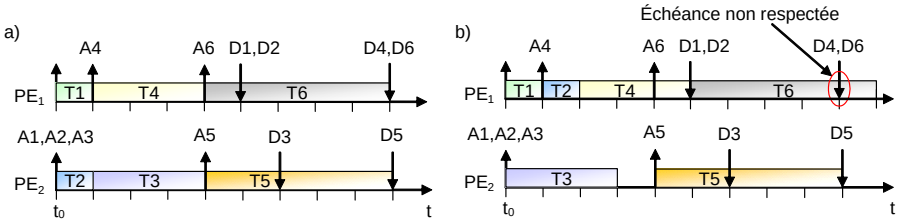

FIG. 4.12 – Problème avec un ordonnancement LLF pour multiprocesseur. L'ordonnancement EDF permet à toutes les tâches de respecter leurs échéances (a), tandis que le LLF ne permet pas à la tâche T_3 de s'exécuter avant son échéance (b).

pour que les préemptions soient négligeables et pour découper chaque tâche en sous-tâches de taille équivalente. Il reste donc les solutions d'ordonnancement EDF et LLF.

Avant de prendre une décision, il faut néanmoins nous intéresser aux *anomalies* d'ordonnancement qui peuvent apparaître en environnement multiprocesseur. Nous parlons d'anomalie lorsqu'un changement intuitivement positif dans un système ordonnançable conduit au non-respect des échéances. Ce terme s'applique également lorsqu'un ordonnancement peut conduire aléatoirement à des solutions ordonnançables ou non-ordonnançables pour un même système. Dans ce dernier cas, on parle également de *tie-break*.

Ainsi, il a été démontré dans [252] que le changement de priorité des tâches, l'accroissement du nombre de processeurs, la réduction de la durée d'exécution d'une tâche, l'augmentation de sa période d'activation, ou la diminution des contraintes de précédence, peuvent conduire à une augmentation de la durée d'exécution, voire à des dépassements des échéances. La considération de ces anomalies est très importante. Le retard d'activation d'une tâche, ou la réduction de sa durée d'exécution sont des évènements qui arrivent régulièrement au cours de l'exécution. Les figures 4.13 et 4.14 illustrent ces deux problèmes d'ordonnancement avec un algorithme EDF puis LLF. D'autre part, la figure 4.15 décrit un exemple d'aléa d'ordonnancement avec l'algorithme EDF. D'autres exemples présentés dans [253, 254] révèlent des problèmes d'ordonnancement rencontrés avec l'algorithme EDF. D'ailleurs, il est démontré dans [255] qu'il n'est pas possible de modifier l'algorithme EDF pour éviter ces aléas en conservant un ordonnancement sur évènement. Seul un algorithme pouvant exprimer l'urgence de l'exécution des tâches et modifier en permanence leur priorité, peut éviter ces aléas d'ordonnancement. C'est pourquoi l'algorithme LLF parvient à résoudre ces anomalies.

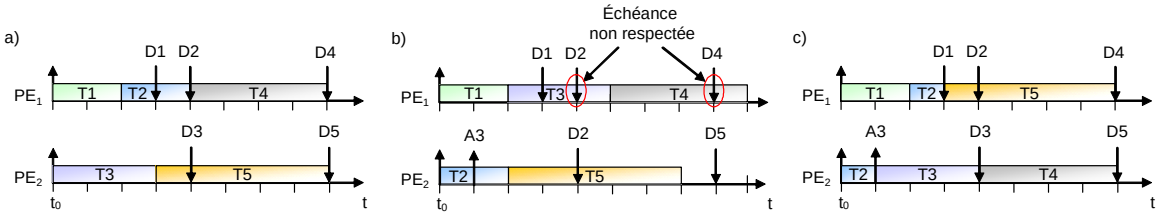

FIG. 4.13 – Anomalie d'ordonnancement lors d'un retard d'activation. Soit un ordonnancement faisable (a), le retard d'activation de la tâche T_3 avec un algorithme EDF engendre une solution non ordonnançable (b). Au contraire, un ordonnancement LLF s'en accomode très bien (c). L'anomalie n'est plus présente.

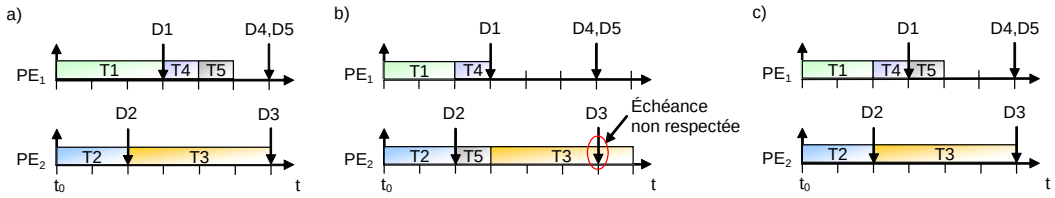

FIG. 4.14 – Anomalie d'ordonnancement lors d'une réduction de la durée d'exécution. Soit un ordonnancement faisable (a), la réduction de la durée d'exécution de T_1 ne permet plus d'obtenir une solution ordonnançable avec l'algorithme EDF (b). A l'inverse, l'algorithme LLF continue à respecter ses échéances (c).

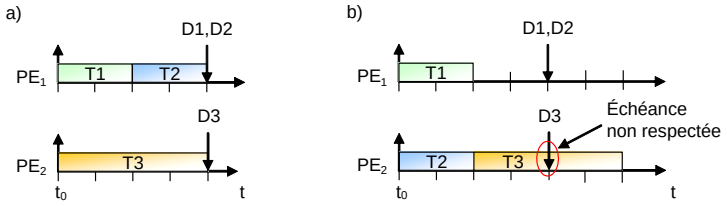

FIG. 4.15 – Anomalie d'ordonnancement avec un algorithme EDF. Sans changer l'algorithme d'ordonnancement utilisé, le résultat peut conduire à une solution ordonnançable (a) ou non-ordonnançable (b). En effet, à l'instant t_0 les 3 tâches ont la même échéance et ont donc le même niveau de priorité. Pourtant, ne pas exécuter T_3 en premier conduit à un défaut d'ordonnancement.

Par conséquent, l'algorithme LLF doit être retenu pour ordonnancer en ligne des tâches temps-réel sur de multiples ressources de calcul. Il ne conduit pas toujours à l'optimalité, mais il n'engendre pas d'anomalies comme l'algorithme EDF. Par ailleurs, même si les préemptions sont nombreuses avec un algorithme par écoulement de temps, elles restent moins pénalisantes qu'avec une solution Pfair. Enfin, même si très peu de travaux existent sur les ordonnancements LLF, nous savons que la limite du taux d'occupation de ses ressources de calcul est maximale. Nous choisissons donc sa réalisation et justifierons davantage notre choix lors de la présentation des résultats dans le prochain chapitre. En attendant, nous présentons dans la sous-section suivante l'algorithme qui a été mis en œuvre dans l'OSoC. Quelques modifications à l'algorithme LLF ont en effet dû être apportées pour réduire le nombre de préemptions.

L'algorithme ELLF

Avec les algorithmes LLF, la tâche dont la laxité est la plus faible est la tâche la plus prioritaire. Si deux tâches ont la même laxité, une des deux est choisie aléatoirement lors de l'allocation. Ceci n'a pas d'incidence sur le respect des échéances puisque si le système est correctement dimensionné, toutes les tâches auront le temps de s'exécuter. Néanmoins, au prochain ordonnancement, la tâche qui n'a pas été allouée devient plus prioritaire et demande la préemption de la tâche en cours d'exécution. Comme les priorités sont évaluées à chaque nouvel ordonnancement, les tâches se préemptent mutuellement jusqu'à la fin de leur exécution. Pourtant, les préemptions sont inutiles puisque si le système est correctement dimensionné toutes les tâches s'exécuteront avant leur échéance. C'est le phénomène de *trashing*. Pour un ensemble de n tâches exécutées sur un monoprocesseur, le nombre maximal de préemptions possibles est alors de $(R_{n-1}+1)$, où R_{n-1} est la durée totale d'exécution des $(n-1)$ premières tâches. Pour éviter ce problème de *trashing*, Hildebrandt et al. ont proposé dans [139] un algorithme appelé *Enhanced Least-Laxity-First (ELLF)*.

Comme le montre la figure 4.16, lorsque plusieurs tâches ont une laxité identique, la tâche dont l'échéance est la plus proche est choisie et toutes les autres sont exclues. En cas d'égalité, un choix arbitraire est effectué. Les tâches exclues sont relâchées à la fin de l'exécution de la tâche autorisée à s'exécuter. Ainsi, dans la situation où les tâches devaient inutilement se préempter mutuellement, elles s'exécutent successivement sans préemption.

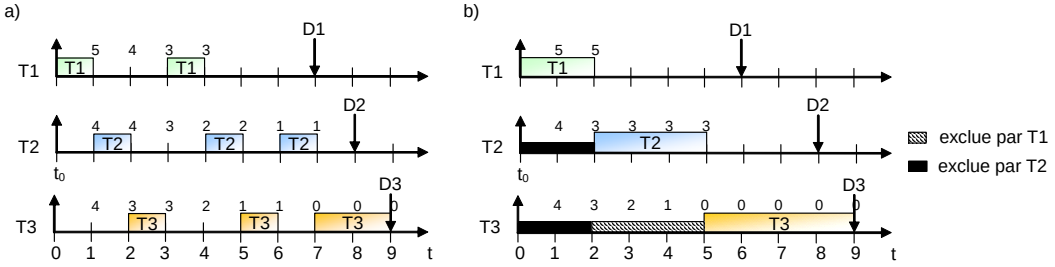

FIG. 4.16 – Phénomène de trashing avec l'algorithme LLF sur monoprocesseur. Dans cet exemple, toutes les tâches ont une laxité initiale identique et égale à 5. Les tâches T_2 et T_3 qui n'ont pas été allouées deviennent plus prioritaires à l'instant $t = 1$. La tâche T_2 demande alors la préemption de la tâche en cours d'exécution. Ainsi, les tâches se préemptent mutuellement jusqu'à la fin de leur exécution (a). Pour une durée d'exécution identique, l'algorithme ELLF évite ces préemptions inutiles (b).

Une tâche peut préempter une autre tâche si elle n'est pas exclue et si sa laxité est inférieure aux autres tâches exclues. Si la dernière condition n'est pas vérifiée, elle doit attendre même si sa laxité est inférieure à celle de la tâche en cours d'exécution. En fait, la tâche exclue continue à être évaluée comme les autres tâches, sauf qu'elle ne peut pas préempter la tâche qui l'a exclue. Grâce à ce dispositif, les préemptions non-nécessaires sont évitées. Alors, pour un ordonnancement de n tâches exécutées sur un unique processeur, le nombre maximal de préemptions devient égal à n quelle que soit la durée totale d'exécution.

Ainsi, l'algorithme mis en œuvre dans l'OSoC est l'algorithme ELLF. Cet algorithme TT permet de mieux maîtriser les temps d'exécution et d'exploiter efficacement la préemption et la migration des tâches. Enfin, même s'il engendre plus de préemptions que l'algorithme EDF, il s'adapte mieux aux conditions variables de l'exécution et des anomalies d'ordonnancement peuvent être évitées.

D'autres problèmes restent à résoudre dans l'ordonnancement multiprocesseur [256]. Tout d'abord, il faut pouvoir exploiter l'hétérogénéité des ressources de calcul. Mais le modèle des *processeurs uniformes* dont la durée d'exécution varie en fontion du processeur utilisé n'est pas adapté à nos besoins. En effet, nous considérons qu'il n'est pas envisageable de considérer qu'une tâche puisse s'exécuter indifféremment sur n'importe quelle ressource de calcul, alors que le code exécuté est différent. Nous étudierons dans la sous-section 4.5 l'approche qui a été retenue pour l'OSoC. Celle-ci exploite principalement les techniques de partitionnement hors-ligne matériel/logiciel.

Enfin, la gestion de la consommation d'énergie est devenue un problème majeur dans la conception des architectures. Comme nous l'avons vu dans le chapitre précédent, certains processeurs peuvent dynamiquement modifier leur fréquence de fonctionnement et leur consommation d'énergie. L'ordonnancement des tâches doit alors être en mesure d'activer ces modes de consommation tout en respectant les contraintes de temps-réel qui lui sont fixées. La sous-section suivante présente les différentes techniques utilisées et propose une solution en ligne adaptée à notre politique d'ordonnancement temps-réel.

4.4.3 Ordonnancement basse-consommation multiprocesseur

La consommation d'énergie des ressources de calcul et de mémorisation peut être adaptée en fonction des besoins applicatifs. Pour cela, il est possible d'activer différents modes de fonctionnement. Tout d'abord, un mode de repos permet une consommation d'énergie minimale en cas de non-activité de la ressource. Ensuite, certaines ressources de calcul peuvent dynamiquement modifier leur tension d'alimentation et leur fréquence de fonctionnement (DVFS). Ces modifications n'ont pas un rôle anodin sur le respect des contraintes temps-réel. Comme le montre la figure 4.17, la mise au repos et le réveil d'une tâche engendre une surconsommation d'énergie, mais également un surcoût temporel non-négligeable. De même, l'utilisation des modes DVFS des processeurs, ralentit l'exécution d'une tâche et nécessite un certain temps d'activation. Ainsi, l'adaptation dynamique de la consommation d'énergie modifie la durée d'exécution des tâches et peut empêcher le respect des échéances d'exécution.

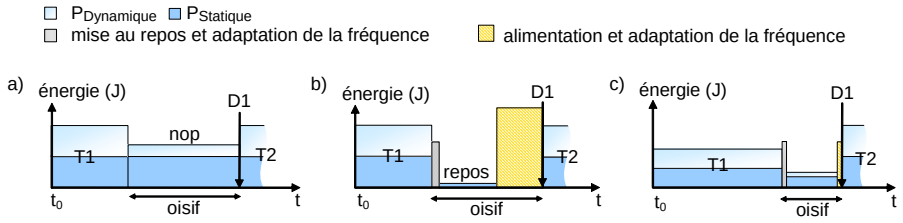

FIG. 4.17 – Réduction de la consommation d'énergie. La non-utilisation de mécanismes de réduction de la consommation d'énergie ne permet pas de diminuer la puissance dissipée lorsque la ressource n'est pas utilisée (a). La première solution consiste à activer le mode de repos (b). Ceci induit une augmentation de l'énergie consommée lors de la mise au repos et du réveil de la ressource. La deuxième solution utilise les modes DVFS en fonction des besoins applicatifs (c). La réduction de la vitesse d'exécution augmente la durée de la tâche. L'efficacité énergétique est accrue et le temps de non-utilisation est réduit.

Il est très difficile d'évaluer et de comparer l'utilisation de ces deux approches. D'une manière générale, les résultats obtenus dépendent de l'application visée et aucune solution ne peut

être optimale. Dans le cadre de ces travaux, nous choisirons une solution simple à mettre en œuvre et qui correspond aux algorithmes d'ordonnancement déjà choisis. Pour cette raison, nous ne nous intéresserons pas aux solutions hors-lignes, malgré les excellents résultats qu'il est possible d'obtenir. En fait, seules ces solutions peuvent s'approcher d'une consommation d'énergie optimale pour des applications déterministes. Ces approches consistent à ralentir le processeur, ou à optimiser le partitionnement logiciel/matériel des tâches, afin d'ajuster la durée d'exécution de l'application à son échéance [257, 258]. D'autres techniques regroupent les cycles non-utilisés sur une même ressource de calcul, pour augmenter le temps passé en mode de repos [259].

Etat de l'art des solutions en lignes

Ainsi, nous allons uniquement nous intéresser aux solutions en lignes. Elles permettent d'activer au cours de l'exécution les modes de repos et DVFS. Cet état de l'art sur les ordonnancements en lignes permettra d'introduire la solution choisie dans l'OSoC.

Mode de repos Les décisions en lignes de mise au repos des ressources de calcul peuvent dépendre du temps d'inactivité passé, de résultats de prédiction d'inactivité, ou enfin de calculs stochastiques [260].

La première approche active le mode de repos au bout d'un temps d'inactivité fixe [261], ou en fonction d'un temps déterminé à partir d'un historique d'exécution [262]. L'efficacité de ces solutions est difficile à discuter, mais elle a au moins l'avantage de la simplicité [263]. En fait, cette approche n'a de sens que si le temps moyen d'inactivité est constant et commun à l'ensemble des tâches à exécuter. Par ailleurs, si la décision prise s'avère inexacte, ceci peut avoir des effets négatifs sur la consommation d'énergie et la durée d'exécution [264].

La seconde approche évite d'attendre que le temps d'inactivité dépasse un certain seuil pour prendre une décision. Le mode de repos est activé dès que le mécanisme de prédiction prévoit un temps long d'inactivité [265]. Chung et al. proposent par exemple d'utiliser un arbre de décision capable d'adapter sa structure au cours de l'exécution [266]. Cependant, de la même façon, une mauvaise prédiction induit une surconsommation d'énergie et surtout peut empêcher une tâche de respecter ses contraintes temps-réel.

La dernière technique établit des relations stochastiques entre les tâches et les moments d'inactivité. Elles utilisent les dates d'activation des tâches et les propriétés des changements de mode, pour définir des chaînes de Markov discretisées. La probabilité des états futurs ainsi formés a la particularité de ne dépendre que de l'état présent [267, 268]. Le problème est qu'en pratique les conditions initiales changent. Ainsi, les prédictions peuvent s'avèrer tout aussi inexactes que les approches précédentes, même si la probabilité de réussite est supérieure.

Techniques DVFS L'apparition dans certains processeurs de mécanismes intégrés, permettant la gestion dynamique de sa consommation d'énergie, a fait émerger de nouvelles techniques d'ordonnancement basse-consommation. Deux techniques principales existent aujourd'hui : les algorithmes *inter-task DVS* et *intra-task DVS* [269].

Dans les systèmes temps-réel, la durée d'exécution des tâches est définie par le WCET. Cependant, une tâche peut utiliser de multiples chemins d'exécution différents et par conséquent la durée de son exécution peut varier de manière importante. Ainsi, lorsque le chemin d'exécution choisi n'est pas le chemin conduisant au pire temps d'exécution (WCEP : Worst

Case Execution Path), la tâche termine son exécution avant son WCET. La différence de temps est appelée *excédent* ou *slack time*. Dans ce cas, ces algorithmes exploitent ce surplus pour ralentir la vitesse du processeur.

La première approche consiste à ajuster la tension d'alimentation et la vitesse d'exécution d'une tâche en fonction de ses paramètres et de son chemin d'exécution. Une méthode appelée *Path-based* repose sur la mise à jour de la fréquence de fonctionnement dès que le chemin d'exécution diffère du WCEP. Les choix des modes DVFS sont définis hors-ligne par une analyse statique de l'exécution des tâches [270, 271]. Au contraire, d'autres solutions consistent à obtenir une estimation stochastique du mode DVFS avant l'exécution de la tâche [272]. Au cours de l'exécution, celui-ci est ensuite modifié en fonction du chemin effectivement suivi. La complexité nécessaire pour évaluer la nouvelle fréquence de fonctionnement ne permet pas une réévaluation permanente du mode à utiliser. Par conséquent, elle ne peut pas exploiter de manière optimale l'excédent et obtient des résultats inférieurs à la première méthode [269].

La dernière approche exploite le temps non-utilisé par la tâche précédente pour rallonger la durée d'exécution maximale des tâches suivantes (figure 4.18). Ainsi, la vitesse des processeurs peut être ralentie tout en respectant l'échéance d'exécution initiale. Pour cela, l'excédent peut être calculé en tenant compte de la différence entre le temps courant et la date d'arrivée de la tâche suivante [273]. Alors, il est donné à la prochaine tâche s'exécutant sur la même ressource de calcul, ou est partagé entre toutes les tâches suivantes [274]. De nombreuses solutions basées sur l'algorithme EDF choisissent de donner l'excédent à la tâche suivante la plus prioritaire avec de très bons résultats [275, 276].

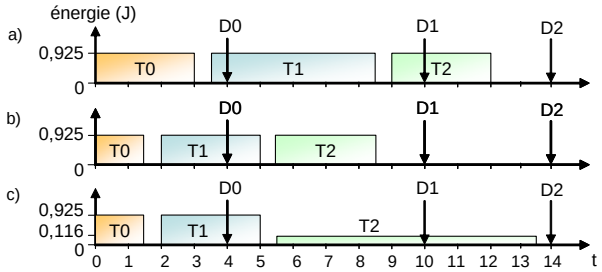

FIG. 4.18 – Répartition de l'excédent et réduction de la consommation d'énergie. La différence entre le temps d'exécution théorique (a) et réel (b) des tâches T_0 et T_1 permet d'obtenir un excédent et de le distribuer à la tâche T_2. La différence de consommation d'énergie entre l'utilisation (c) ou non (b) des modes DVFS du processeur XScale permet un gain en énergie égale à $(3 \times 0,925) - (8 \times 0,116)$, soit 1,84 J (26 %).

L'algorithme RSSR et utilisation des modes de repos

Nous allons maintenant nous intéresser à la solution choisie pour être mise en œuvre dans l'OSoC. En fait, ce dernier s'occupe de la gestion en ligne de la consommation d'énergie en exploitant les modes de repos et DVFS de ses ressources.

Mode de repos Initialement, toutes les ressources sont au repos. Au fur et à mesure des besoins applicatifs, les ressources de calcul ou de mémorisation se réveillent dès qu'une demande d'utilisation est reçue. Ainsi, le nombre de ressources activées dépend exactement du nombre de tâches à exécuter et du niveau de parallélisme nécessaire. Chaque ressource

repasse automatiquement dans son mode de repos dès qu'elle n'est plus utilisée. Ceci est le cas seulement à la fin de l'exécution de l'application (à la fin de l'exécution d'une tâche, la ressource n'est pas libérée). En fait, aucune des méthodes en lignes proposées dans la littérature n'a été utilisée pour la gestion des modes de repos. Cette gestion particulière permet de prendre en compte les temps de mise au repos et de réveil seulement une seule fois pendant toute la durée d'exécution de l'application.

Algorithme RSSR D'autre part, pour utiliser les mécanismes DVFS des ressources de calcul qui en disposent, nous avons mis en œuvre un algorithme appelé *Restricted Sharing Slack Reclamation (RSSR)*. Il s'apparente aux mécanismes de répartition de l'excédent présentés précédemment. Cependant, son adjonction avec l'algorithme ELLF et un fort parallélisme de tâches nécessite des aménagements particuliers.

Tout d'abord, le calcul du *slack time* est directement obtenu à partir des grandeurs utilisées pour définir la priorité des tâches. Dans la laxité, nous avons l'information du temps passé à exécuter la tâche. Il suffit donc de le soustraire au WCET connu au moment de l'ordonnancement. Ensuite, l'excédent est distribué aux tâches suivantes appartenant à la même application. Mais seule la tâche parmi toutes les suivantes, dont la durée d'exécution est la plus longue, est privilégiée pour le recevoir. En effet, comme l'a montré Zhu et al., il est plus intéressant de favoriser les tâches de longue durée car elles génèrent un excédent plus important [274]. Une étude plus approfondie pourrait éventuellement être effectuée, mais un algorithme trop complexe à mettre en œuvre serait trop pénalisant. D'autant plus que la fréquence de la distribution de l'excédent à de multiples successeurs reste limitée.

Lorsque l'excédent obtenu par la tâche est suffisant pour activer un mode DVFS, celui-ci est consommé et la laxité est mise à jour avec la nouvelle durée d'exécution obtenue. Le choix du mode DVFS passe par le calcul de la fréquence minimale de fonctionnement. Celle-ci doit permettre d'obtenir une augmentation de la durée d'exécution juste inférieure ou égale à l'excédent. Le mode DVFS permettant d'exploiter le plus de *slack time* est choisi pour réduire au maximum l'énergie dissipée. Dans le cas contraire, celui-ci est conservé et est propagé aux tâches suivantes à la fin de l'exécution de la tâche. L'algorithme RSSR peut être utilisé indifféremment pour des tâches périodiques ou apériodiques et respecte les précédences d'exécution.

Ces techniques de réduction de la consommation d'énergie complètent les algorithmes d'ordonnancement des tâches non-temps-réel et temps-réel présentés précédemment. La mise en œuvre de tous ces procédés d'ordonnancement dans l'OSoC fait l'objet de la sous-section suivante. Plus particulièremennt, elle décrit les éléments CMU et SCU qui permettent la gestion des tâches et leur ordonnancement pour l'architecture SCMP-LC.

4.4.4 Mise en œuvre de l'ordonnancement dans l'OSoC

Avant de détailler la mise en œuvre des mécanismes d'ordonnancement dans l'OSoC, nous allons préciser la manière dont sont calculées les priorités des tâches temps-réel. Contrairement aux systèmes temps-réel habituellement rencontrés, toutes les échéances sont dépendantes les unes des autres. Ceci permet d'adapter l'ordonnancement aux conditions réelles d'exécution.

Calcul des priorités

Les applications mobiles temps-réel possèdent une échéance d'exécution globale. Leurs tâches ne sont pas directement contraintes, mais elles doivent toutes s'exécuter avant une échéance commune. L'idée est de calculer en ligne l'échéance et la laxité de la première tâche à partir de la contrainte applicative. On en déduit par la suite l'échéance des autres tâches en fonction des chemins d'exécution choisis (figure 4.19). L'affectation statique d'une échéance empêche la considération des boucles d'exécution. En effet, dans ce cas, une tâche est appelée plusieurs fois et l'échéance qu'elle doit respecter est différente à chaque itération.

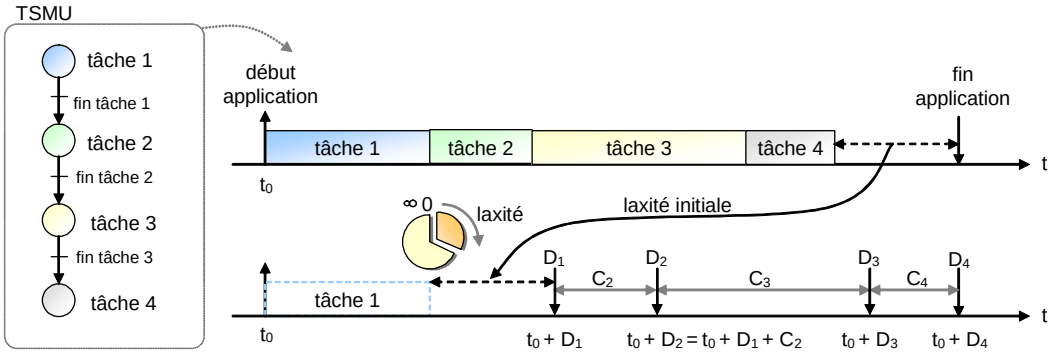

FIG. 4.19 – Initialisation du calcul de la laxité. Tout d'abord, la différence entre l'exécution totale des tâches et la date de fin d'application, est calculée. Elle correspond à la laxité admise pour l'ensemble de l'exécution des tâches de cette application. Ensuite, cette laxité est attribuée à la tâche 1. L'ensemble des échéances sont calculées en fonction des échéances des tâches précédentes et de leur durée d'exécution. Une tâche i est caractérisée par une durée d'exécution C_i et une échéance D_i.

Pour mieux comprendre l'intérêt de cette approche, nous allons reprendre l'application présentée figure 4.19. Comme le montre la figure 4.20, si la tâche 1 est préemptée par une autre tâche plus prioritaire, alors la laxité initiale est diminuée d'une durée égale à celle de la préemption. La propagation de cette laxité aux tâches suivantes permettra de prendre en compte cette interruption dans le calcul des priorités. Ceci est également vrai pour le retard d'exécution de la tâche 2 ou de la tâche 4, qui peut être dû à la durée d'ordonnancement, ou à la configuration de leur mémoire. L'urgence de la tâche 4 va pouvoir être modifiée en tenant compte de tous ces aléas d'exécution et l'échéance d'exécution de l'application sera respectée. Contrairement à toutes les solutions existantes, ceci permet de considérer des traitements non-déterministes et des conditions réelles d'exécution.

Principe de l'ordonnancement dans l'OSoC

Comme les précédentes sous-sections l'ont présenté, l'OSoC intègre plusieurs politiques d'ordonnancement : pour les tâches non-temps-réel, temps-réel et pour la consommation d'énergie. Les tâches non-temps-réel sont exécutées lorsqu'une ressource de calcul est disponible et qu'aucune tâche temps-réel est en attente d'exécution. L'association de ces algorithmes conduit à une solution complète et capable de s'adapter aux besoins des futures applications mobiles. La figure 4.21 tente de résumer le principe de l'ordonnancement dans l'OSoC qui sera détaillé par la suite.

Tout d'abord, l'élément responsable de l'ordonnancement appelé SCU récupère les informations envoyées par les ressources de calcul et l'unité de configuration. Les PE peuvent retourner

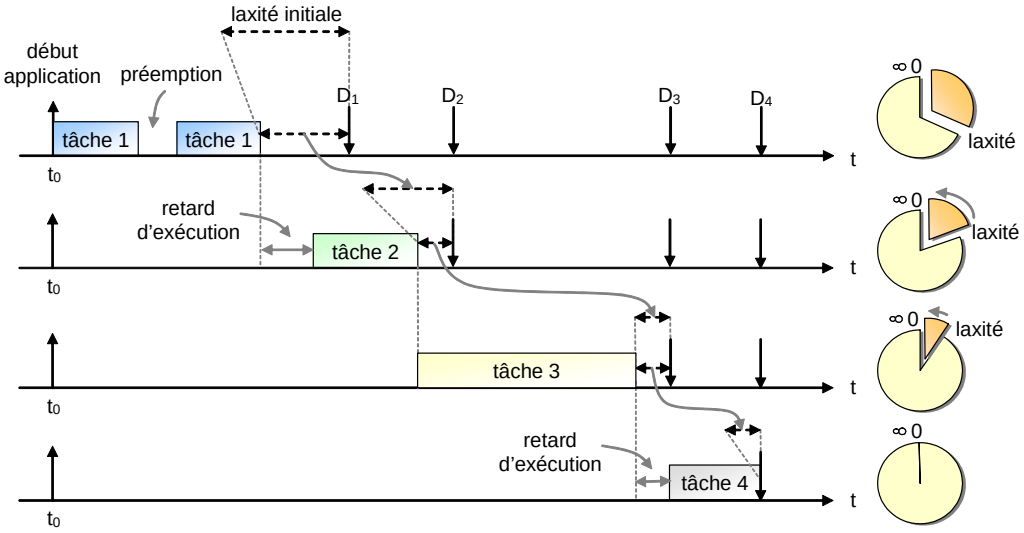

FIG. 4.20 – Calcul en ligne de la priorité des tâches. La propagation de la laxité permet de prendre en compte tous les évènements qui ralentissent l'exécution de l'application.

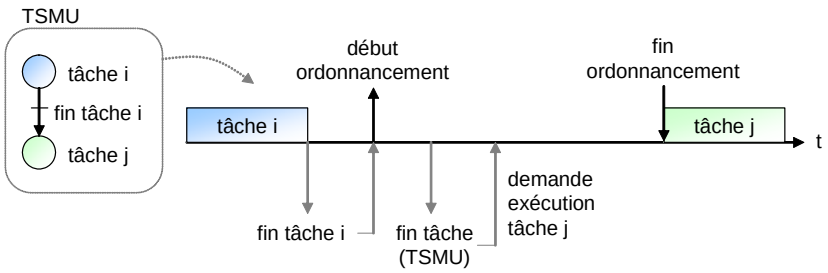

FIG. 4.21 – Principe de l'ordonnancement dans l'OSoC. Comme tout algorithme par écoulement du temps, un nouvel ordonnancement a lieu à chaque nouveau tick. La première étape consiste à prendre en compte les informations provenant des ressources de calcul et de mémorisation. Ensuite, elles sont retournées au TSMU, qui établit alors une nouvelle liste de demandes d'exécution et de configuration. Enfin, une liste triée de tâches éligibles est obtenue. Les tâches sont allouées sur les ressources de calcul, ou configurées par l'unité de configuration à la fin de l'ordonnancement.

des fins d'exécution de tâches ou d'applications, ou des erreurs d'exécution. La dernière tâche d'une application retourne un identifiant particulier qui permet par la suite, la libération des ressources de calcul et de mémorisation liées à cette application. Les erreurs sont liées à des problèmes éventuels d'identifiant de tâche, de configuration, ou d'exécution. De même, l'unité de configuration peut retourner des fins de configuration et une erreur lorsqu'une demande de configuration est envoyée par l'OSoC, alors que toutes les mémoires partagées sont occupées.

Ensuite, le SCU modifie l'état des tâches en fonction des évènements reçus et propage la laxité aux tâches suivantes lors d'une fin d'exécution comme nous venons de le présenter. Puis, les fins de configuration et d'exécution sont envoyées au TSMU via l'unité de contrôle afin de propager les jetons de configuration et d'exécution. Par ailleurs, les demandes de libération des mémoires et des ressources de calcul sont préparées si une fin d'application a été reçue.

Par la suite, le SCU reçoit les demandes de configuration et d'exécution provenant du TSMU, ainsi que les commandes envoyées par le système d'exploitation. Ces dernières peuvent demander la suspension, le réveil ou l'arrêt d'une application en cours d'exécution. Le SCU commence par modifier l'état des tâches en fonction des commandes reçues. Eventuellement, il prépare les demandes de libération des mémoires et des ressources de calcul si une demande d'arrêt d'application a été reçue. D'autre part, il modifie les états des tâches en fonction des demandes de configuration et d'exécution et initialise les grandeurs utilisées pour le calcul de la laxité.

Après quoi, le SCU met à jour les laxités des tâches en fonction de leurs états. Par exemple, une tâche temps-réel devant s'exécuter et non présente sur une ressource de calcul, voit sa laxité réduite d'une durée égale à un tick. Les éventuels dépassements d'échéance sont également détectés pour être ultérieurement signalés au système d'exploitation. Par ailleurs, le calcul de l'excédent est effectué et est propagé à la tâche suivante selon l'algorithme RSSR.

Enfin, le SCU réalise le tri de toutes les tâches actives dans l'OSoC. Selon les algorithmes étudiés précédemment, on retrouve dans l'ordre les tâches temps-réel ayant reçu une demande d'exécution, les tâches non-temps-réel ayant reçu une demande d'exécution, puis toutes les autres tâches actives. Pour effectuer le tri de ces dernières, une estimation de la laxité à partir des tâches précédentes est effectuée pour les tâches temps-réel. Ceci permettra par la suite de configurer les tâches selon un ordre de priorité.

A partir de la liste de tâches triées, les tâches éligibles sont désignées en fonction du nombre et du type des ressources disponibles. Le SCU envoie toutes les demandes d'exécution, de configuration, de suspension, ou de libération à l'élément en charge de l'allocation (PMAU). Les modes DVFS sont également choisis et associés aux demandes d'exécution. Les mécanismes d'allocation utilisés dans l'OSoC seront détaillés dans la section suivante.

Etats des tâches

La figure 4.22 représente les différents états occupés par les tâches actives dans l'OSoC et résume le principe de fonctionnement détaillé précédemment. Chaque tâche peut occuper 6 états différents : *sleep*, *wait*, *conf*, *conf_ready*, *ready* et *exec*.

Une tâche en mode *sleep* n'est pas active. Elle occupe cet état parce qu'elle a fini son exécution ou a été arrêtée par le système d'exploitation. L'état *wait* est utilisé pour les tâches dont le système d'exploitation a demandé la suspension. A leur réveil, les tâches retrouvent leur état initial. Une tâche en mode *conf* est en cours de configuration. Elle passe dans ce mode suite à une demande de configuration reçue par le TSMU. Si une demande d'exécution a été envoyée simultanément, l'état *conf_ready* est activé. La tâche attend alors la configuration de ses mémoires pour s'exécuter. L'état *ready* est occupé par les tâches dont la configuration est finie et qui attendent l'arrivée d'une demande d'exécution. Enfin, les tâches devant s'exécuter sont dans l'état *exec*. Seules ces tâches sont éligibles et susceptibles d'être allouées sur les ressources de calcul.

Ce diagramme d'état n'est néanmoins pas complet. Les tâches périodiques nécessitent l'introduction d'états supplémentaires pour exprimer une attente d'exécution sans intervention du système d'exploitation. Nous verrons par la suite comment les tâches périodiques sont exécutées par l'OSoC.

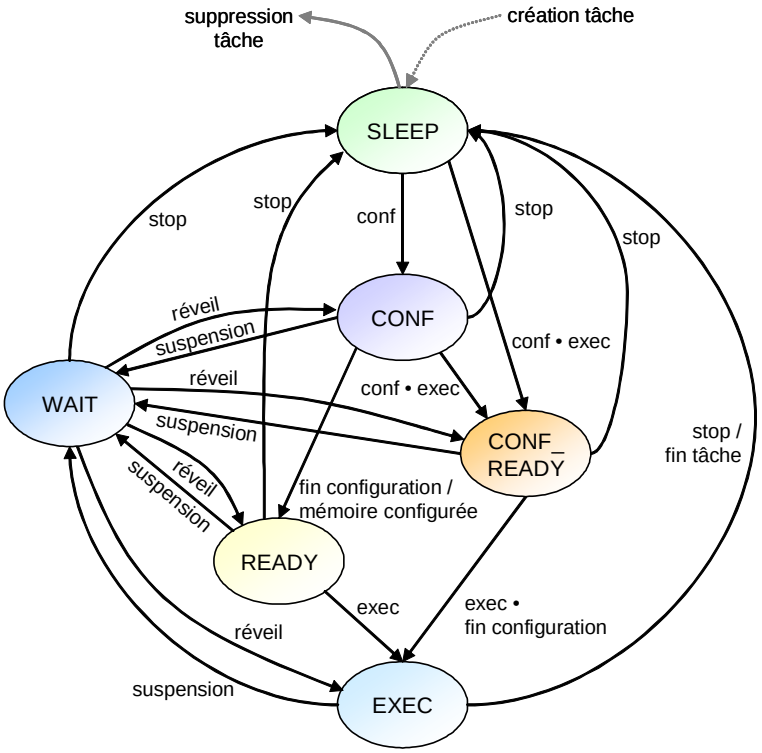

FIG. 4.22 – Diagramme d'états des tâches dans l'OSoC. Les demandes d'exécution sont notées *exec* et les demandes de configuration *conf*.

Les tâches périodiques

Comme toutes les autres applications, les tâches périodiques sont décrites sous forme de graphes (figure 4.23). Ceux-ci sont constitués de deux tâches ayant des modes de fonctionnement différents de ceux qui viennent d'être présentés. La première est nommée *periodic* et la seconde *wait*.

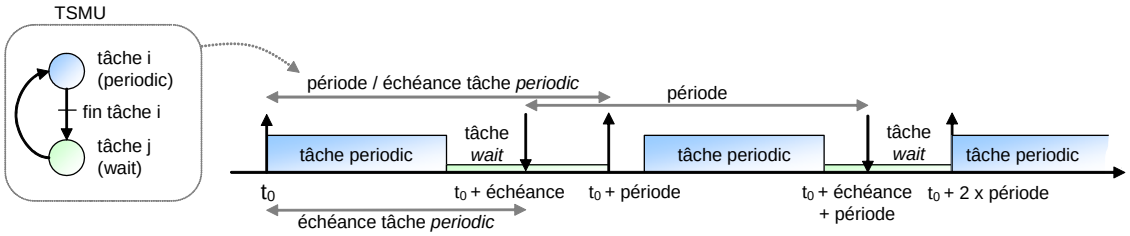

FIG. 4.23 – Ordonnancement des tâches périodiques et calcul des échéances. Une tâche périodique est représentée par deux sous-tâches. La tâche *wait* s'exécute dès que la tâche *periodic* se termine. Son échéance correspond à la période d'activation de la tâche *periodic*.

Le calcul de la laxité est singulier puisqu'il n'y a pas de propagation entre les tâches *wait* et *periodic*. Le début de la tâche *periodic* est similaire à un début d'exécution d'une application. En effet, si l'échéance précédente a été respectée, nous disposons toujours d'un temps constant pour exécuter notre tâche périodique. Il n'est donc pas nécessaire de prendre en compte les

aléas qui ont eu lieu lors de la période précédente.

Comme le montre la figure 4.23, nous considérons qu'une tâche périodique doit respecter une échéance d'exécution inférieure ou égale à sa période d'activation. Ainsi, il existe deux dates à respecter : l'échéance de la tâche *periodic* et sa période d'activation qui est également l'échéance de la tâche *wait*. Ces dates sont calculées à partir des valeurs précédentes et de la période d'activation. Il s'agit simplement d'une translation temporelle et le temps courant n'est toujours pas utilisé pour l'ordonnancement.

La tâche *periodic* s'exécute normalement et est ordonnancée comme toutes les autres tâches temps-réel par l'algorithme ELLF. A la fin de son exécution, la tâche *wait* est activée. En fait, elle n'est pas allouée sur une ressource de calcul. Elle est similaire à un compteur décrémenté par l'OSoC à chaque nouvel ordonnancement. Sa valeur initiale correspond à la laxité, ou encore à la différence entre la prochaine période d'activation et la date de fin d'exécution de la tâche *periodic*. Lorsque la laxité est nulle, la tâche *periodic* est à nouveau activée. La figure 4.24 décrit le diagramme d'état de la tâche *wait*. La demande de configuration n'entraîne pas une configuration de sa mémoire mais l'initialisation de sa laxité.

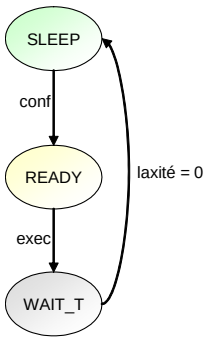

FIG. 4.24 – Diagramme d'état pour les tâches périodiques.

La suspension d'une tâche périodique a des incidences plus importantes que pour d'autres tâches. Il ne suffit pas de reprendre l'exécution de la tâche, il faut pouvoir respecter la périodicité de celle-ci. Pour cette raison, des mécanismes ont été mis en œuvre pour se synchroniser de nouveau avec la période d'activation initiale.

Cette quatrième section a permis de détailler l'ensemble des mécanismes d'ordonnancement intégrés dans l'OSoC. L'association de toutes les solutions présentées permet d'ordonnancer des tâches :

- non-temps-réel (selon la durée d'exécution) ;
- temps-réel (algorithme ELLF) ;
- périodiques et non-périodiques ;
- et d'adapter la consommation d'énergie de l'achitecture SCMP-LC en fonction des besoins applicatifs (modes de repos et algorithme RSSR).

Surtout, tous les procédés d'ordonnancement mis en œuvre dans l'OSoC permettent de prendre des décisions en ligne et d'avoir une vue globale de l'occupation des ressources. Par ailleurs, l'ordonnancement sur écoulement du temps offre une solution de contrôle entièrement déterministe et robuste aux anomalies d'ordonnancement habituellement rencontrées. Elle est particulièrement efficace pour prendre en compte tous les évènements qui pourraient conduire à des défauts d'ordonnancement.

Ainsi, nous avons étudié la sélection puis l'ordonnancement des tâches. L'ordonnancement transforme une liste de tâches actives, en une liste de tâches éligibles dont l'exécution est prioritaire. Il nous reste à préciser les mécanismes mis en œuvre dans le PMAU pour l'attribution des ressources de calcul aux tâches. Cette dernière étape, appelée *allocation*, joue un rôle particulièrement important lorsqu'elle est associée à un ordonnancement global. Elle décide de la préemption, de la migration et de la libération des ressources. La section suivante va donc permettre de détailler notre politique d'allocation et la gestion des ressources hétérogènes.

4.5 Allocation des tâches

L'allocation des tâches contribue fortement à l'amélioration du taux d'occupation des ressources et donc des performances du système. Le soin apporté à l'élaboration de la politique d'allocation est donc particulièrement important. Cependant, la littérature est inexistante, tant il est vrai que des solutions de contrôle globales en ligne n'ont jamais été mises en œuvre. C'est donc en effectuant de nombreuses simulations et mesures que l'élément PMAU a pu être réalisé.

Comme nous l'avons vu ultérieurement, l'OSoC doit permettre la migration et la préemption de tâches sur de multiples ressources de calcul hétérogènes. La gestion de l'hétérogénéité est un premier problème lors de l'allocation des tâches, alors même qu'elle n'avait aucune incidence sur la sélection et l'ordonnancement. Dans notre cas, elle est néanmoins gérée hors-ligne au moment de la compilation. Chaque tâche indépendante est compilée pour une ressource particulière qui sera choisie au moment de l'allocation pour son exécution. En fait, cela revient à utiliser des techniques d'ordonnancement et d'allocation globales pour des ressources homogènes, tandis qu'un partitionnement est effectué entre les tâches de nature identique.

La figure 4.25 illustre l'algorithme mis en œuvre dans l'OSoC et qui est appliqué à chacune des tâches éligibles fournies par le SCU. Tout d'abord, le PMAU vérifie que la tâche n'est pas déjà présente sur une ressource de calcul. Bien sûr, le type de ressource considéré dépend de celui de la tâche. Ceci permet d'éviter la préemption de tâches moins prioritaires, mais surtout d'utiliser une ressource au repos non nécessaire. Ensuite, si la tâche n'occupe aucun PE, il faut déterminer la meilleure ressource susceptible de recevoir la tâche. Pour limiter le nombre de préemptions, l'idée consiste à favoriser les ressources qui doivent être libérées, puis les ressources libres. Ensuite, si aucune solution n'existe, la tâche en cours d'exécution la moins prioritaire est préemptée.

La recherche de la tâche la moins prioritaire permet d'éviter de préempter une tâche qui doit être allouée par la suite. En effet, il est préférable d'éviter la migration d'une tâche si cela n'est pas nécessaire. Par ailleurs, un système temps-réel doit toujours favoriser les tâches les plus prioritaires. L'allocation mise en œuvre dans l'OSoC permet donc de maîtriser l'occupation des ressources de calcul disponibles. Pour cela, elle dispose de différents mécanismes permettant :

- la migration ;
- la préemption ;
- et la gestion des ressouces hétérogènes.

Surtout, elle contribue à réduire le nombre des préemptions et des migrations, même si l'architecture SCMP-LC minimise le coût de leur utilisation. Ceci évite une activité supplémentaire inutile et contribue donc à réduire la consommation d'énergie. De plus, le surcoût

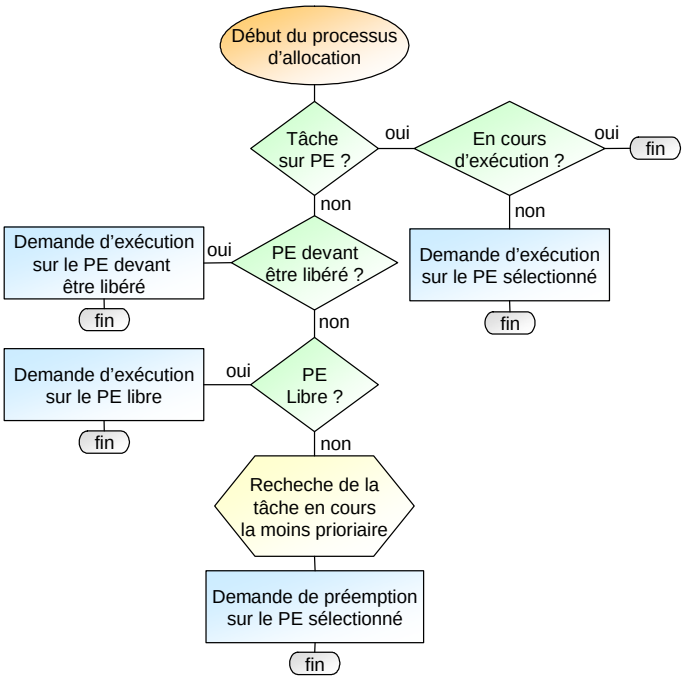

FIG. 4.25 – Algorithme d'allocation mis en œuvre dans l'OSoC. Le but de cet algorithme est d'allouer les tâches éligibles sur les ressources de calcul, en minimisant le nombre des préemptions et des migrations.

temporel induit n'étant pas nul, moins elles sont utilisées et moins les performances sont pénalisées.

4.6 Communications et interface

Pour terminer la description de l'OSoC, nous allons maintenant nous intéresser au lien de communication qui existe avec le système d'exploitation. Ce dernier peut envoyer des requêtes à l'OSoC, qui peut lui retourner des informations sur son état ou l'alerter en cas de dysfonctionnements (non respect des échéances, etc.). Pour cela, chacun dispose de registres partagés et d'un jeu d'interruptions (figure 4.26).

Quel que soit l'émetteur ou le récepteur, le principe d'envoi d'une requête est identique. Il consiste à réserver l'utilisation du moyen de communication et à mettre à jour le contenu de son registre partagé. L'envoi d'une interruption *ready* provoque la lecture de la requête envoyée. Une interruption d'acquittement *ack* finit par valider cet échange (figure 4.27).

Le système d'exploitation peut envoyer quatre commandes différentes à l'OSoC. Celles-ci permettent la création, l'arrêt, la mise en attente et le réveil dynamique d'applications. L'envoi de la commande est associé à l'identifiant de l'application et à son numéro d'instance. Ceci permet à l'OSoC de retrouver la tâche en mémoire, ainsi que d'appliquer la commande aux tâches concernées uniquement. Par exemple, la réception d'une demande d'arrêt d'une application entraîne la suppression du graphe en cours de construction dans le TSMU, l'arrêt des tâches en cours d'exécution, la libération des mémoires et enfin la suppression de toutes les

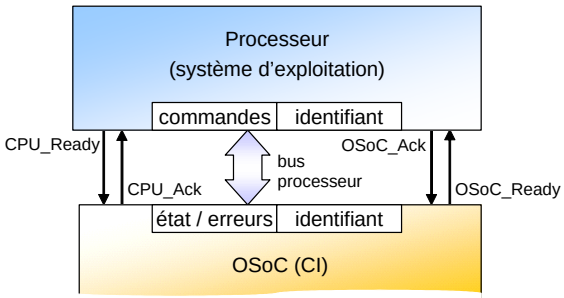

FIG. 4.26 – Interface de communication entre le système d'exploitation et l'OSoC. L'élément CI de l'OSoC communique avec le processeur via un système simple d'interruptions.

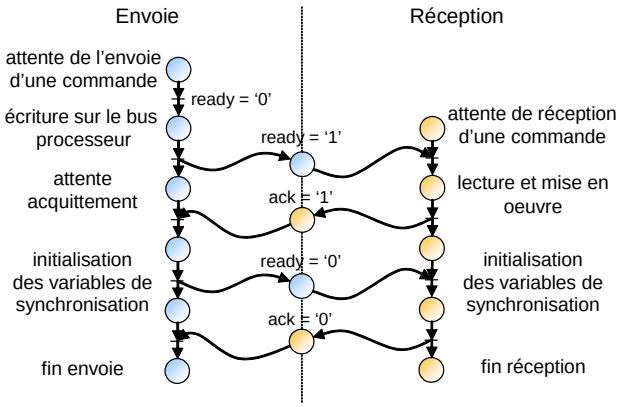

FIG. 4.27 – Interface de communication entre l'OS et l'OSoC

références liées à cette application dans l'OSoC. Par ailleurs, son utilisation permet l'exécution concurrente de multiples applications et de plusieurs instances d'une même application.

L'OSoC peut retourner trois types d'information à la fin de chaque nouvel ordonnancement, simultanément si nécessaire, au système d'exploitation. Il peut indiquer la fin de l'exécution d'une application, une erreur d'ordonnancement comprenant des défauts d'exécution ou de configuration, ou enfin une erreur relative au non-respect d'une échéance.

4.7 Synthèse

Les besoins des futures applications mobiles ont conduit à la proposition d'une architecture multiprocesseur asymétrique. Ces architectures comportaient a priori de nombreux défauts qu'il nous fallait surmonter pour que cette approche réponde à nos besoins. Elles souffraient d'une faible occupation de leurs ressources de calcul, de synchronisations et d'une programmation complexes, mais surtout aucune solution de contrôle efficace n'existait. Et cela alors même que la position centrale de notre contrôle est le facteur limitant de notre architecture.

La première étape a consisté à définir nos besoins en terme de calcul. Nous nous sommes aperçus qu'une meilleure occupation de nos ressources passait par des mécanismes de préemption et de migration efficace. Ainsi, un modèle d'exécution à mémoire partagée et distribuée

a donné lieu à une architecture appelée *SCMP-LC*. Il consiste à exécuter des tâches indépendantes sur des ressources de calcul hétérogènes.

En fait, toute la complexité a été transférée à l'architecture de contrôle, qui se voit dorénavant en charge des synchronisations, de la gestion de la cohérence et de l'intégrité des données, de la sélection des tâches, de l'ordonnancement et enfin de l'allocation. La réactivité du contrôle étant particulièrement importante dans notre architecture, il nous fallait alors trouver une solution innovante et capable de mettre en œuvre efficacement ces nombreux services. Alors même qu'une solution logicielle s'est révélée inexploitable, la conception d'une architecture matérielle a semblé être capable de répondre à toutes nos exigences.

Ce chapitre a donc présenté une architecture de contrôle offrant une solution complète et déterministe. Elle peut contrôler en ligne les systèmes multiprocesseurs hétérogènes embarqués. Ainsi, l'architecture *OSoC* est en mesure de gérer dynamiquement l'exécution concurrente de multiples applications, les dépendances de donnée et de contrôle, ainsi que l'ordonnancement de tâches temps-réel, non-temps-réel, périodiques et non-périodiques. De plus, elle permet de maîtriser la préemption et la migration des tâches, ainsi que la consommation d'énergie du système.

L'intégration de tous ces services dans une même architecture n'a cependant de sens que si la réactivité du contrôle pénalise faiblement le reste du système. L'examen des résultats obtenus constitue par conséquent une justification de l'intérêt de notre approche. Les résultats de synthèse et de simulation sont présentés dans le chapitre suivant.

Chapitre 5

Validation de l'architecture OSoC

Sommaire

Les chapitres précédents ont permis de présenter l'architecture *SCMP-LC* et notamment la partie en charge du contrôle. Cette dernière, appelée *OSoC*, intègre de nombreuses fonctionnalités permettant une gestion efficace de la cohérence, de l'intégrité des données, de la sélection des tâches, de l'ordonnancement et enfin de l'allocation.

La position centralisée du contrôle dans notre architecture multiprocesseur induit de fortes contraintes de performance et de réactivité. Les nombreux échanges avec les ressources de calcul et de mémorisation doivent être les plus courts possibles, pour ne pas pénaliser les performances du système. Notre architecture multiprocesseur asymétrique n'a de sens que si le contrôle est particulièrement réactif.

C'est pourquoi, l'examen des résultats obtenus est très important. Tout d'abord, nous présentons notre méthodologie de validation et de conception, ainsi que le modèle d'architecture

choisi pour nos simulations. Ensuite, nous mettons en évidence les avantages de nos architectures SCMP-LC et OSoC. Chacune des caractéristiques présentées dans les chapitres précédents est évaluée. Enfin, des comparaisons sont effectuées avec une solution logicielle similaire, le noyau temps-réel μCOS-II et d'autres solutions matérielles proposées dans la littérature.

5.1 Validations et implémentation matérielle

Ces travaux de thèse n'ont pas pour objectif de réaliser un circuit. Ils consistent simplement à proposer un nouveau paradigme d'architecture en vue de répondre aux besoins des futures applications mobiles. La vérification de son fonctionnement et de sa possible mise en œuvre matérielle, reste néanmoins incontournable pour garantir son intérêt. Ceci constitue la toute première étape avant sa fabrication et sa commercialisation.

La validation d'une architecture doit se faire en plusieurs étapes. La première consiste à vérifier l'intérêt et l'efficacité des fonctionnalités mises en œuvre : c'est la *validation fonctionnelle*. Elle permet d'affiner les solutions choisies, mais surtout de vérifier leur faisabilité. La seconde passe par la réalisation d'un modèle physique de l'architecture : c'est la *validation matérielle*. Elle offre la possibilité d'évaluer la complexité de l'architecture, ainsi que ses performances. La validation matérielle sera limitée à la synthèse logique dans le cadre de ce manuscrit. La sous-section suivante présente plus précisément notre méthodologie de conception et de validation.

5.1.1 La méthodologie de conception et de validation

La description des fonctionnalités de l'architecture doit être faite dans un langage de description de haut niveau. Ceci permet de valider rapidement l'architecture et de profiter d'un environnement de simulation efficace pour affiner les choix architecturaux. En effet, les temps de simulation peuvent avoir un impact important sur les temps de développement d'une architecture.

Pour cela, nous avons utilisé une bibliothèque de description matérielle appelée *SystemC* dans un environnement C/C++. Elle permet de profiter des mécanismes d'héritage ou de polymorphisme du C++ pour décrire aisément des ensembles hiérarchiques [277]. Par ailleurs, cette solution offre la possibilité de simuler conjointement des parties logicielles et matérielles et de décrire efficacement des bancs de test complexes. Ceci se révèle très utile dans notre cas puisque le système complet est composé d'éléments hétérogènes logiciels et matériels. De cette façon, les communications entre le processeur qui exécute le système d'exploitation et l'architecture SCMP-LC vont être modélisées facilement.

Cependant, l'utilisation de SystemC comporte quelques inconvénients [278]. Alors même qu'une description utilisant un haut niveau d'abstraction offre une vitesse de simulation importante, elle ne permet pas la détection d'erreurs matérielles. Seule une description au niveau transfert de registre (RTL : Register Transfer Level) offre ces informations. Il est bien sûr possible d'utiliser SystemC pour réaliser un modèle RTL de notre architecture, mais aucun outil de synthèse logique n'existe à partir du SystemC.

La méthode consiste alors à réaliser un modèle fonctionnel en SystemC qui décrit tous les protocoles de communication au niveau bit (figure 5.1). A ce moment, il est intéressant

de profiter des avantages du C++ pour effectuer une hiérarchisation des différents éléments de l'architecture. La description doit être uniquement fonctionnelle et être accélérée le plus possible pour réduire les temps de simulation. Une fois le modèle validé, il faut identifier les différents éléments dont on souhaite obtenir une description au niveau RTL. A ce niveau, le langage VHDL ou Verilog est utilisé en fonction des outils de synthèse disponibles.

La validation des fonctionnalités du modèle RTL de chacun des éléments se fait dans l'environnement SystemC créé précédemment. Grâce à cette méthode, on profite d'un environnement de simulation rapide et validé. La validation consiste simplement à retrouver le comportement initial du modèle fonctionnel décrit en SystemC. Une fois les fonctionnalités des éléments à nouveau validées au niveau RTL, il ne reste plus qu'à effectuer la synthèse logique. Celle-ci consiste à transformer le code VHDL en un ensemble de portes logiques élémentaires caractérisées en temps et en surface par un fondeur. La validation de cette dernière étape permet d'obtenir un élément validé matériellement.

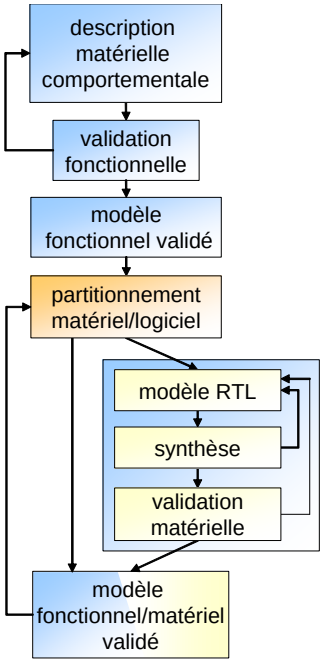

FIG. 5.1 – Méthodologie de conception et de validation. Tout d'abord, elle consiste à valider les fonctionnalités de l'architecture en réalisant un modèle fonctionnel. Ensuite, il faut identifier les éléments qui doivent être décrits au niveau RTL. La validation matérielle des différents éléments sélectionnés se fait à partir du modèle fonctionnel.

Le respect de cette méthodologie permet de toujours profiter d'un environnement de simulation efficace et de garantir le fonctionnement de l'architecture après la synthèse logique. En effet, les nouvelles fonctionnalités de co-simulation SystemC/VHDL des logiciels de CAO (Conception Assistée par Ordinateur) offrent la possibilité de conserver les mêmes vecteurs de test tout au long des étapes de conception et de validation. De plus, il est possible de raffiner le niveau de description, élément par élément, tout en conservant un environnement fonctionnel. Enfin, le modèle final combinant différents niveaux de description matérielle rend possible l'exécution d'applications complexes et l'évaluation des performances du système.

5.1.2 Validation fonctionnelle

La figure 5.2 représente le modèle fonctionnel de l'architecture SCMP-LC. Tous ces éléments ont été décrits à des niveaux d'abstraction différents. Le processeur et le système d'exploitation sont simulés par une console qui permet de demander l'exécution, la suspension ou l'arrêt d'une tâche au cours de la simulation. Elle reproduit le comportement non prédictible des demandes de l'utilisateur.

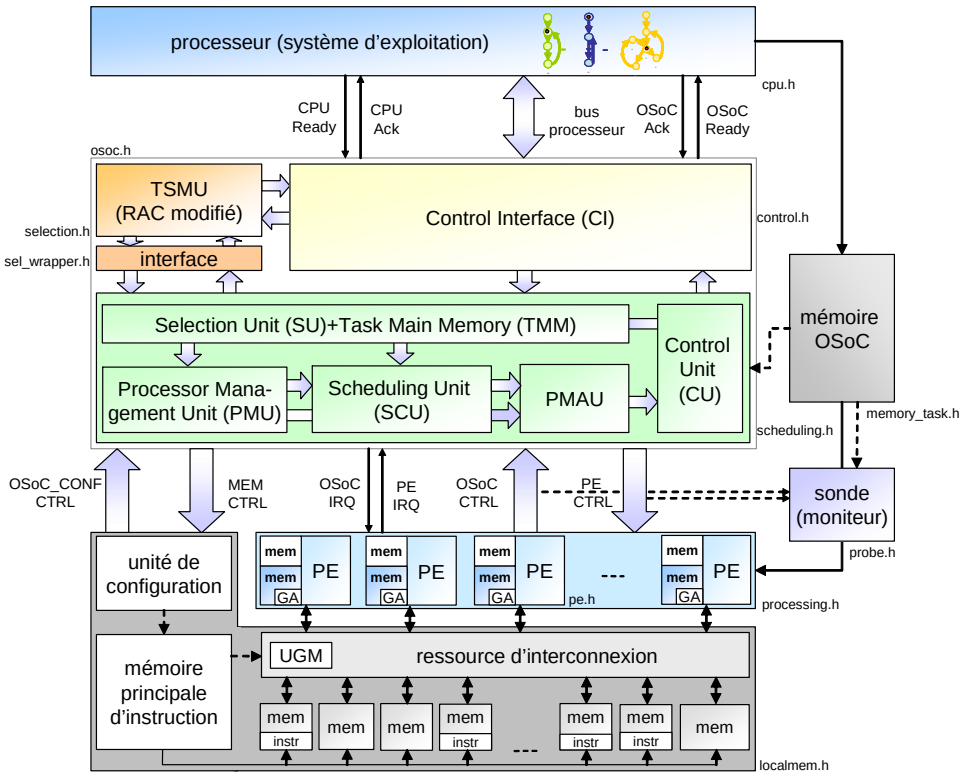

FIG. 5.2 – Modèle fonctionnel de l'architecture SCMP-LC. On retrouve tous les différents éléments de notre architecture, ainsi que leurs liens de communication. La *sonde* permet de faciliter la visualisation des signaux. Un programme développé en PERL intègre les informations fournies par la sonde directement dans les chronogrammes obtenus après simulation.

Les autres éléments extérieurs à l'OSoC ont également été modélisés à haut niveau afin de ne pas pénaliser les performances des simulations. Néanmoins, toutes les fonctionnalités nécessaires et ayant un impact sur le comportement de l'OSoC ont été décrits au niveau RTL.

Par exemple, une ressource de calcul est représentée principalement par un compteur. L'exécution d'une tâche est simulée par la décrémentation de celui-ci initialisé avec son WCET. La valeur du WCET est définie arbitrairement dans le cas d'applications de tests ou est obtenue après l'exécution de la tâche sur une plateforme d'évaluation. Dans le cadre de ces travaux de thèse, nous avons utilisé une carte Cogent CSB637 [279] disposant d'un cœur de processeur ARM920T [280]. Par ailleurs, un processeur Intel XScale [190] a été choisi pour valider notre gestion de la consommation d'énergie, en raison de la rapidité de ses changements de mode DVFS.

Par ailleurs, les ressources d'interconnexion et de mémorisation n'ont pas été modélisées. Nous avons supposé qu'un lien de communication existe toujours entre la ressource de calcul et ses mémoires. Nous considérons une latence du réseau d'interconnexion égale à 10 ns et un temps d'accès aux mémoires de 4 ns. De même, les instructions de la tâche sont toujours présentes avant que la tâche ne s'exécute. Il suffit donc de considérer les temps d'accès aux mémoires partagées via les ressources d'interconnexion dans le WCET des tâches. La durée de préemption d'une tâche a été fixée quant-à-elle à 1,034 μs. Elle correspond au temps nécessaire à la sauvegarde de 256 mots de 32 bits. Par ailleurs, nous avons supposé la taille maximale du code des tâches égale à 1 Ko. Par conséquent, la durée de configuration pour chacune des tâches est de 4 μs.

Dans l'OSoC, le TSMU intègre le RAC et toutes les modifications suggérées dans le chapitre précédent. En revanche, la construction dynamique ou les mécanismes de destruction n'ont pas été mis en œuvre. En effet, leur modélisation n'est pas nécessaire et cela pour deux raisons. Tout d'abord, nous étions en mesure de caractériser précisément son comportement grâce aux développements sur le RAC qui ont déjà été réalisés [210]. Ensuite, en disposant de son pire temps d'exécution, il suffit de considérer cette pénalité maximale lors de chaque accès. Ceci permet de s'affranchir d'une description de toutes ses fonctionnalités et de réduire les temps de simulation.

La complexité du modèle

Comme le montre le tableau 5.1, la complexité du modèle obtenu dépend des caractéristiques de l'OSoC. Nous avons choisi trois configurations différentes qui seront reprises pour la présentation des résultats suivants. Le nombre de tâches actives simultanément, qui correspond également à la taille du TSMU, peut être égal à 32 ou à 16. Ceci nous a semblé suffisant pour un premier modèle et conduira par la suite à des solutions matérielles de taille acceptable. Par ailleurs, les résultats que nous avons obtenus sur le RAC montrent que sa surface est proportionnelle au carré du nombre de cellules. Par conséquent, un nombre de cellules compris entre 8 et 30 est mieux adapté aux systèmes embarqués. Enfin, nous avons choisi de caractériser un OSoC pour 8 ou 16 ressources de calcul. Pour un nombre de tâches actives inférieur ou égal à 32, ceci nous semble justifié. Néanmoins, d'autres modèles seront étudiés par la suite pour mettre en évidence les propriétés de l'OSoC.

Caractéristiques du modèle fonctionnel	Modèle avec 32 tâches-16 PE	Modèle avec 32 tâches-8 PE	Modèle avec 16 tâches-8 PE
Nombre de processus séquentiels	26	18	18
Nombre de processus combinatoires	7	7	7
Nombre de signaux	1617	1369	889
Performance du modèle (cycles.s^{-1})	15679	15950	21381

TAB. 5.1 – Caractéristiques du modèle fonctionnel du système. Les performances ont été mesurées sur un processeur Intel Pentium 4 HT 3,2 GHz. Elles sont globalement comprises entre 12000 et 25000 cycles.s^{-1}. Les valeurs présentées sont indicatives et dépendent de la fréquence d'activation des processus et donc des applications. La taille du code utile est de 11000 lignes, tandis que le pas de simulation est de 1 ns. Ainsi, pour simuler 100 ms, il faut environ 1h45.

En SystemC, les performances du modèle sont directement dépendantes du nombre de processus et de signaux utilisés [281]. De même, la fréquence d'activation de ces processus, ou plus particulièrement de leurs ports d'entrée dans le cas de processus combinatoire, modi-

fie les temps de simulation. Néanmoins, les valeurs présentées dans le tableau 5.1 restent significatives. D'ailleurs, elles sont équivalentes à celles obtenues dans la littérature [282].

La programmation du modèle

La programmation des applications exécutées par notre modèle est effectuée à l'aide d'un langage assembleur. Ce dernier décrit les motifs élémentaires des réseaux de Petri, ou les dépendances de contrôles et de données entre les tâches. De plus, il offre toutes les primitives nécessaires pour caractériser l'application et chacune de ses tâches. Par exemple, il est possible de préciser l'échéance à respecter, ou le type de la ressource utilisé par la tâche.

Ensuite, une analyse du programme obtenu est effectuée par un outil réalisé sur la base de Lex&Yacc. Il assure la vérification de la cyclicité des graphes, de la validité de l'échéance par rapport à la durée des tâches à exécuter et fournit la laxité de la première tâche à exécuter. Enfin, il transforme le langage assembleur dans un code interprétable par notre modèle fonctionnel. Les outils développés pendant cette thèse sont uniquement destinés à simplifier la programmation de l'OSoC en attendant la disponibilité d'outils de programmation plus évolués.

Les exemples d'application

Pour effectuer la validation du fonctionnement de l'OSoC, nous avons implanté plusieurs applications différentes. Tout d'abord, nous avons réalisé des graphes d'application théoriques disposant de toutes les primitives élémentaires des réseaux de Petri. Ceci nous a permis de vérifier le fonctionnement de l'OSoC pour des géométries de graphe différentes, des temps de tâche minimaux, ou des fréquences d'activation variables. Ensuite, nous avons testé deux applications réelles : un encodeur MPEG4-AVC (Advanced Video Coder) et un décodeur MPEG2-AAC (Advanced Audio Coder).

Encodeur MPEG4-AVC Le MPEG4-AVC ou H.264 est une norme de compression de flux vidéo proposée par le Moving Picture Experts Group (MPEG) [283, 284]. Ce standard permet le codage d'une grande variété de formats et d'objets vidéos, en alliant une qualité de compression supérieure aux précédentes normes MPEG. Il offre une réduction importante du débit de transmission, tout en gardant une qualité visuelle équivalente. Ce gain en compression est nécessaire pour les communications bas débits qui ne dépassent pas les 48 Ko/s pour les dispositifs mobiles de troisième génération.

Dans cette thèse, nous avons considéré le profil *Baseline* avec une seule image de référence, le filtre de déblocage, la prédiction inter complète et la méthode CAVLC (Context Adaptative Variable Length Coding). Ce profil autorise un large spectre d'applications à faible latence et est donc particulièrement intéressant pour les systèmes embarqués.

La première étape dans le processus de compression est l'estimation et la compensation du mouvement (figure 5.3). L'atténuation de la redondance temporelle dans le signal vidéo représente une part importante de la compression obtenue. Plutôt que d'améliorer le codage du signal lui-même, la plupart des techniques efficaces essayent de réduire la taille du signal transmis en améliorant la prévision du mouvement. Ainsi, le codage d'une image peut être restreint à une simple translation par rapport à une image antérieure.

En pratique, comme la direction du mouvement est inconnue, de multiples prédictions composées d'une image de référence et d'une translation différentes sont effectuées. La prédiction conduisant à l'erreur minimale est alors conservée. Pour la détecter, un algorithme de mise

en correspondance des *macroblocs* (MB) est utilisé. Chaque image est divisée en macroblocs de 16×16 pixels. Dans l'algorithme MPEG4-AVC qui a été choisi, nous effectuons au total dix prédictions différentes en exploitant la redondance spatiale de l'image (intra), ou la redondance temporelle par rapport à l'image précédente (inter).

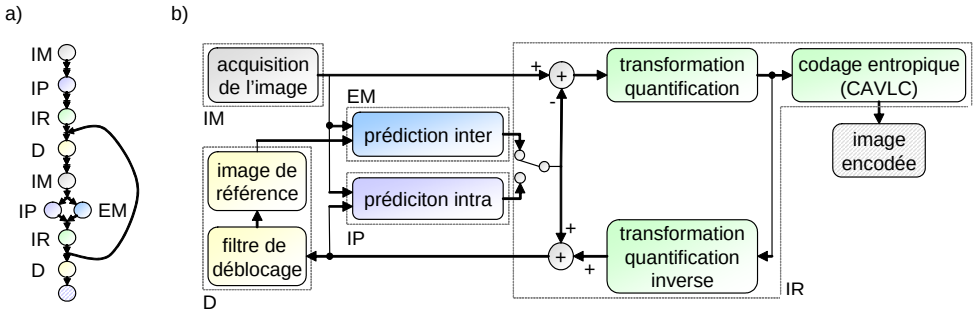

FIG. 5.3 – Diagramme d'un encodeur MPEG4-AVC. Le graphe d'application correspondant est constitué de 10 tâches indépendantes (a). Le découpage en macroblocs offre la possibilité de paralléliser les traitements et d'obtenir davantage de traitements concurrents. Le diagramme de l'encodeur est composé de 7 modules (b). Pour obtenir un codage optimal, il est possible de choisir entre une prédiction inter et intra.

La seconde étape consiste à réduire les redondances spatiales de l'image différentielle obtenue avec l'image prédite. Cette redondance exprime la similarité entre les pixels adjacents horizontaux ou verticaux. Une transformation mathématique appelée *transformée cosinus discrète (TCD)* permet de passer d'une représentation spatiale de l'information à une représentation fréquentielle. Il en résulte des coefficients qui représentent la valeur moyenne de l'intensité lumineuse ou de la couleur, ainsi que leurs variations. Ensuite, les coefficients de fréquences spatiales élevées sont minimisés lors de la quantification, au profit de ceux de fréquences basses. En effet, les changements rapides de l'intensité lumineuse sont imperceptibles pour l'œil humain et peuvent être éliminés.

La troisième étape est le codage entropique. Nous avons choisi de traiter la redondance des coefficients de fréquence obtenue par un codage à longueur variable appelé CAVLC. Celui-ci réduit le nombre moyen de bits utilisés pour représenter l'information vidéo compressée. Ce type de codage associe des coefficients à une série de codes de longueur variable. Les coefficients se répétant le plus souvent sont codés sur un nombre de bits inférieur à ceux dont la fréquence d'apparition est faible. Cette méthode est sans perte et ne détériore donc pas la qualité de l'image.

Enfin, la transformation et la quantification inverses sont réalisées. Puis, un *filtre de déblocage* est appliqué après décompression de l'image en cours pour créer une nouvelle image de référence. Il permet de réduire les artefacts qui apparaissent lors du traitement par blocs des images.

Le tableau 5.2 regroupe les temps d'exécution, sur un ARM920T pour chacune des tâches qui compose l'algorithme de compression MPEG4-AVC. En fait, l'ensemble de ces tâches couvre environ 99 % du temps nécessaire pour exécuter l'application. Le reste des fonctionnalités de l'algorithme n'a pas été pris en compte. Les résultats présentés sont obtenus pour l'exécution de 64 macroblocs sur un seul processeur ARM (sans contrôle). Pour obtenir un débit de 30 images par seconde, le temps maximal pour traiter une image est d'environ 33 ms.

Par conséquent, un seul processeur ARM ne peut pas atteindre ces performances pour le traitement d'une image CIF (352×288 pixels) qui est composée de 386 macroblocs. Cependant, cette application se prête facilement au parallélisme de tâches.

Par la suite, nous considèrerons que l'ensemble des traitements des macroblocs d'une même image sont concurrents et ne nécessitent pas de synchronisations supplémentaires. Ceci étant, la structure de l'architecture SCMP-LC partage l'ensemble de ses données et les macroblocs voisins utilisés dans des traitements concurrents pourraient être partagés.

Tâches	Nombre d'opérations par macrobloc	Nombre de cycles	Durée d'exécution (μs)
Acquisition de l'image (IM)	0	256	2
Prédiciton Intra (IP)	17799	83182	462
Filtre de déblocage (D)	7680	43207	240
Transformation, quantification et codage (IR)	11358	51517	286
Prédiction Inter (EM)	98059	220372	1224
Pour l'exécution de 64 macroblocs	248096	$25{,}86.10^6$	143683

TAB. 5.2 – Durée d'exécution des tâches de l'encodeur MPEG4-AVC avec un ARM920T cadencé à 180 MHz. Le nombre d'opérations par macrobloc ne prend pas en compte les chargements et les enregistrements des données.

Décodeur MPEG2-AAC Une autre application utilisée pour les validations de l'OSoC est le décodeur MPEG2-AAC [285]. Ce standard peut véhiculer jusqu'à 48 canaux audios à des fréquences d'échantillonnage inférieures aux normes précédentes, tout en gardant une qualité équivalente. Ces nombreux canaux permettent le codage multilingue, ainsi que l'utilisation d'effets sonores enveloppants. Surtout, il répond davantage aux besoins de compression que nécessitent les systèmes embarqués.

Nous considérons dans cette thèse le profil le plus complet (*Main profil*), sans la boucle de prédiction utilisée pour améliorer encore la compression du flux audio. Ce profil offre la meilleure qualité de compression disponible. Par ailleurs, nous utilisons un taux d'échantillonnage audio égal à 44,1 kHz et un débit de 128 kbps sur 2 canaux.

Comme le montre la figure 5.4, la première étape du décodeur MPEG2-AAC consiste à effectuer un décodage anti-bruit. Celui-ci permet de reconstruire l'ensemble des informations encodées précédemment, à partir d'une transformation inverse de Huffman. Nous obtenons alors les valeurs quantifiées, ainsi que les seuils de quantification sur l'échelle des fréquences. Par ailleurs, les *facteurs d'échelle* sont également retrouvés.

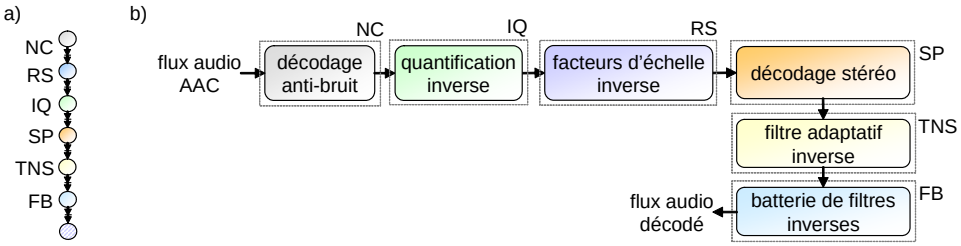

FIG. 5.4 – Diagramme d'un décodeur MPEG2-AAC (b). Le graphe d'application correspondant est constitué de 6 tâches indépendantes (a).

La seconde étape est la quantification inverse. A partir d'une transformation non-linéaire, elle retrouve la distribution approchée des intensités acoustiques en fonction de la fréquence, c'est-à-dire le *spectre* du signal.

La troisième étape applique inversement les facteurs d'échelle pour retrouver l'intensité du signal initial. En effet, lors de l'encodage, le signal a été amplifié pour certaines bandes de fréquences pour améliorer le rapport signal sur bruit.

La quatrième étape consiste à séparer le signal obtenu dans le domaine fréquentiel, en deux signaux distincts correspondants aux canaux encodés. Lors de la compression, un algorithme de jonction stéréo fusionne les canaux gauche et droit dans le domaine des basses fréquences, en tenant compte des redondances rencontrées dans chacun des canaux. En effet, à basse fréquence, le système auditif humain ne discerne quasiment plus la position stéréophonique des sons.

La cinquième étape restore l'enveloppe temporelle du signal. Pour améliorer la quantification du bruit dans le signal, un filtrage avait été effectué lors de sa compression. Celui-ci est encore appelé *Temporal Noise Shaping (TNS).*

Enfin, une batterie de filtres inverses permet de retrouver un signal audio proche de celui compressé. Tout d'abord, une transformation IMDCT (Inverse Modified Discrete Cosine Transform) vers le domaine temporel est effectuée. Cette transformation tient compte des propriétés psychoacoustiques de l'oreille humaine. Les nombreux filtres permettent de l'adapter en fonction de chaque sous-bande de fréquences, qui est perçue différemment par l'oreille.

Le tableau 5.3 indique les temps d'exécution de chacune de ces tâches sur un ARM920T. Les durées d'exécution ont été obtenues à partir des travaux de Liu et al. publiés dans [286]. Pour constituer des tâches de durées suffisamment importantes pour l'OSoC, nous avons choisi de traiter 16 trames simultanément par tâche. Ceci ne change pas le débit de décompression, mais nécessite des tampons en entrée et en sortie de la chaîne de traitement. Un seul ARM920T cadencé à 180 MHz peut traiter en temps-réel l'application. Cette application n'a donc pas besoin d'être parallélisée sur plusieurs ressources de calcul comme l'encodeur MPEG4-AVC.

Tâches	Nombre de cycles	Durée d'exécution (μs)
Décodage anti-bruit (NC)	11940	66,3 (18,9 %)
Quantification inverse (IQ)	3475	19,3 (5,5 %)
Facteurs d'échelle inverse (RS)	3032	16,8(4,8 %)
Décodage stéréo (SP)	821	4,6 (1,3 %)
Filtre adaptatif inverse (TNS)	505	2,8 (0,8 %)
Batterie de filtres inverse (FB)	43401	241 (68,7 %)
Pour l'exécution de 16 trames	63175	351 (100 %)

TAB. 5.3 – Durée d'exécution des tâches du décodeur MPEG2-AAC avec un ARM920T cadencé à 180 MHz. Le taux d'échantillonnage audio est égal à 44,1 kHz et le débit à 128 kbps sur 2 canaux.

Analyse des résultats La validation du fonctionnement de l'OSoC a été effectuée grâce à la visualisation des signaux internes de l'architecture après simulation. La figure 5.5 montre un exemple d'analyse des résultats. Ce chronogramme représente l'exécution de deux encodeurs MPEG4-AVC, ainsi que de deux décodeurs MPEG2-AAC. Les multiples applications s'exécutent de manière concurrente, en utilisant toutes les ressources disponibles.

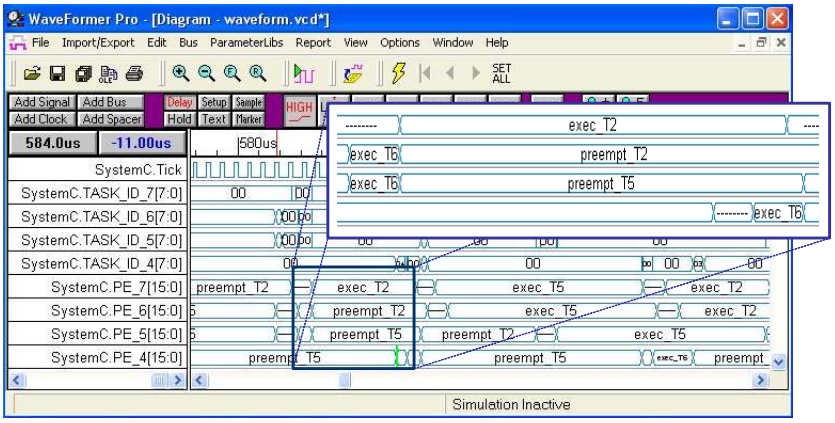

FIG. 5.5 – Exemple d'analyse des résultats. Ce chronogramme représente l'exécution de 4 applications concurrentes (2 encodeurs MPEG4-AVC et 2 décodeurs MPEG2-AAC). La tâche T6 est interrompue par la tâche T2, puis son exécution est reprise sur une ressource de calcul devenue disponible. La préemption et la migration de la tâche ont bien eu lieu. La deuxième tâche T6 préemptée par la tâche T5 appartient à une autre application.

L'ensemble des fonctionnalités présentées dans les deux chapitres précédents a été validé par ces simulations. Ces dernières ont également permis d'affiner les algorithmes d'ordonnancement et d'allocation mis en œuvre dans l'OSoC. La section suivante reprend chacune des mesures effectuées et analyse les résultats obtenus, en fonction de différents modèles d'architecture SCMP-LC.

5.1.3 Validation matérielle

Le modèle matériel de l'architecture OSoC a une structure identique au modèle fonctionnel. Seul le niveau de description des différents éléments est plus proche de la réalité physique du circuit. Toutes les fonctionnalités ont été décrites au niveau RTL, en tenant compte des caractéristiques physiques des composants utilisés. Pour cela, nous avons utilisé le langage VHDL et l'outil ModelSim de Mentor Graphics.

Nous n'avons pas réalisé tous les éléments qui constituent l'OSoC en VHDL. Le TSMU et le CI n'ont pas été décrits matériellement. En effet, la complexité du TSMU peut facilement être évaluée grâce aux précédents travaux de thèse de S. Chevobbe [210]. Par ailleurs, un circuit a été réalisé en technologie 130 nm par notre laboratoire. Enfin, l'élément *Control Interface* est de très faible complexité et n'a quasiment aucune incidence sur les caractéristiques de l'OSoC.

D'autre part, même si toutes les mémoires ont été décrites en VHDL, seules les mémoires inférieures à 256 octets ont été synthétisées. En effet, les mémoires de tailles supérieures sont réalisées avec des outils de conception particuliers. Ainsi, peu de validations ont pu être réalisées après la synthèse logique. Néanmoins, le modèle RTL est valide et une synthèse FPGA sur un Xilinx Virtex 4 LX100 a confirmé le fonctionnement de l'OSoC. De plus, les estimations de taille et de consommation d'énergie ont pu être réalisées en extrapolant les caractéristiques des modèles proposés par la société STMicroelectronics.

Détail de la structure interne de l'OSoC

Le modèle de l'architecture OSoC est principalement composé d'une machine d'états synchrone de 223 états distincts. En fait, la taille du code VHDL développé représente 13000 lignes utiles. Elle inclut également la description de 12 mémoires, 5 FIFO et 9 opérateurs (tri, modes DVFS, horloge...). Nous allons nous intéresser plus particulièrement à l'élément de tri et à une mémoire type utilisée dans l'OSoC. Les autres dispositifs sont relativement standards et ne méritent pas une attention particulière.

Tout d'abord, nous avons mis en œuvre un dispostif de tri systolique (figure 5.6). Il permet un tri très efficace au niveau matériel. En fait, alors même que la description d'un tri se révèle relativement simple au niveau logiciel, il reste très coûteux en accès mémoire. Le tri systolique permet le tri d'une liste de n éléments en seulement $2n$ cycles. Il consiste à propager vers l'étage suivant uniquement la tâche la plus prioritaire. Cette dernière est donc la première à sortir du dispositif (en $n+1$ cycles), tandis que la moins prioritaire est la dernière qui sera récupérée.

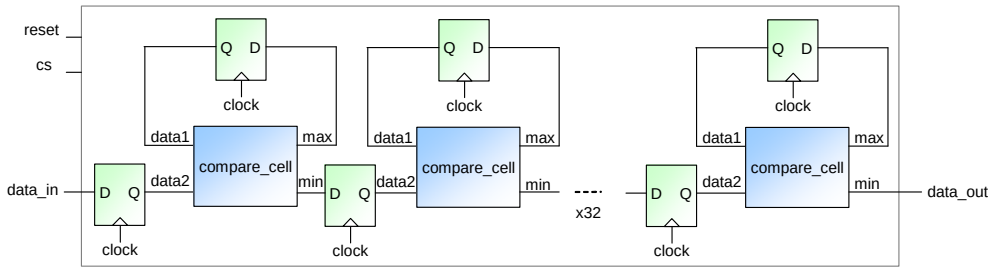

FIG. 5.6 – Elément de tri systolique mis en œuvre dans l'OSoC. Le bloc *compare_ cell* identifie la tâche la plus prioritaire (min) selon les algorithmes présentés dans les chapitres précédents. Si l'élément de tri est constitué de n blocs *compare_ cell*, le tri est effectué en $2n$ cycles.

Enfin, la plupart des mémoires utilisées dans l'OSoC est adressable par le contenu, comme présentées figure 5.7. En effet, pour accéder aux caractéristiques d'une tâche, nous disposons uniquement de son identifiant. Il faut donc pouvoir retrouver son adresse. Par ailleurs, la création et la suppression dynamique des tâches gérées par l'OSoC nécessitent de trouver une ligne libre pour effectuer une nouvelle écriture. Ces dispostifs s'ajoutant aux mémoires traditionnelles ont été parallélisés pour réduire leurs latences d'écriture et de lecture. D'autre part, alors même que les mémoires de données n'ont pas été synthétisées lorsqu'elles étaient de taille importante, toute la logique supplémentaire et ces dispositifs ont été considérés. Les résultats de synthèse ASIC et FPGA sont présentés dans les sous-sections suivantes.

Synthèse ASIC

La technologie utilisée pour la synthèse logique de l'OSoC est la HCMOS9 (CMOS 130 nm, low leakage) de STMicroelectronics. Elle a été effectuée avec l'outil *Design Compiler* de Synopsys. Les tableaux 5.4 et 5.5 regroupent l'ensemble des résultats de synthèse ASIC. Ces derniers ne prennent pas en compte le placement et le routage du circuit.

La durée d'exécution des éléments de l'OSoC met en évidence la prédominance de l'unité d'ordonnancement. Elle représente près de la moitié du nombre total de cycles. Le TSMU a finalement peu d'impact sur les performances de l'OSoC. Ceci est dû notamment au parallélisme qui a pu être exploité. En attendant les nouvelles demandes d'exécution et de

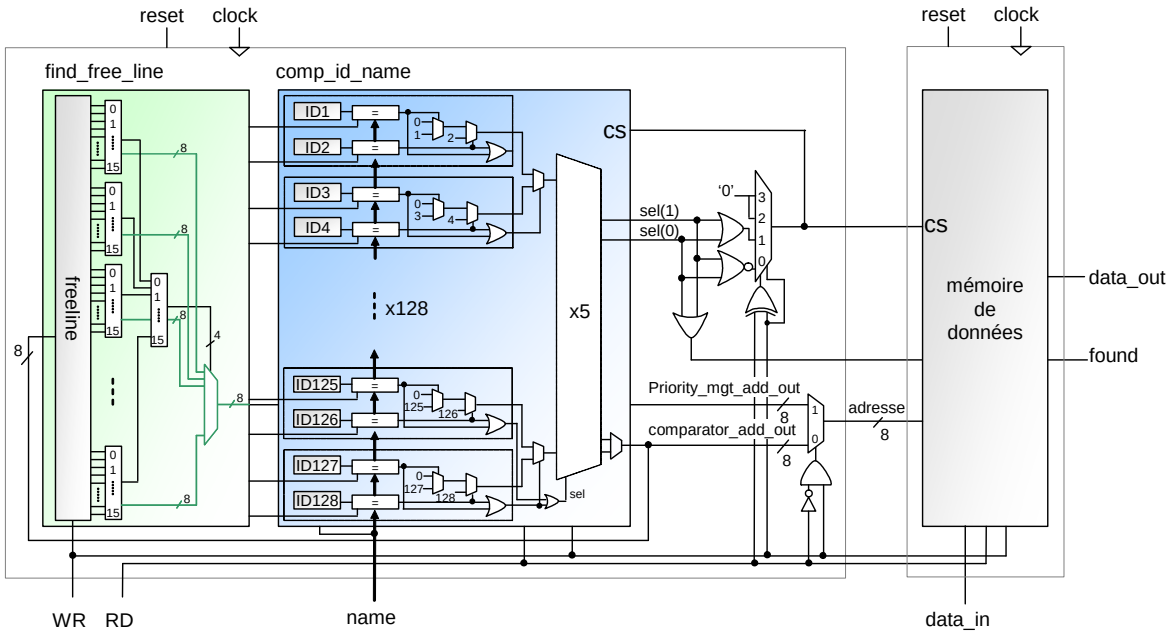

FIG. 5.7 – Exemple d'élément mémoire utilisé dans l'OSoC. Cette mémoire est adressable par le contenu. Le bloc *comp_id_name* compare l'identifiant *name* avec ceux présents dans chacune des lignes. Il retourne l'adresse de la ligne qui le contient, ainsi que deux bits *sel(0)* et *sel(1)* qui indiquent si il a été trouvé. Le bloc *find_free_line* retourne l'adresse de la première ligne mémoire libre. Elle est principalement constituée d'encodeurs de priorité. A partir de l'adresse obtenue, l'écriture ou la lecture a lieu normalement dans la mémoire de données.

Elements de l'OSoC	Nombre de cycles 32 tâches-16 PE	Nombre de cycles 32 tâches-8 PE	Nombre de cycles 16 tâches-8 PE
Scheduling Unit (SCU)	3302 (47,3 %)	3302 (57,7 %)	1654 (52,4 %)
PE and Memory Allocation Unit (PMAU)	1765 (25,3 %)	581 (10,1 %)	501 (15,9 %)
Selection Unit (SU + TMM)	1156 (16,5 %)	1124 (19,6 %)	580 (18,4 %)
Control Unit (CU)	340 (4,9 %)	300 (5,2 %)	172 (5,4 %)
Task Selection and Mgt Unit (TSMU)	322 (4,6 %)	322 (5,6 %)	200 (6,3 %)
Processor Management Unit (PMU)	96 (1,4 %)	96 (1,7 %)	48 (1,5 %)
Nombre total de cycles	6981	5725	3155
Temps total (μs)	34,905	28,625	15,775

TAB. 5.4 – Durée d'exécution des éléments de l'OSoC. Les durées sont exprimées en nombre de cycles et en pourcentage de la durée totale d'exécution de l'OSoC. La synthèse a été effectuée à 200 MHz.

configuration, l'unité d'ordonnancement met à jour le contenu de ses mémoires en fonction de l'évolution des tâches actives. Un parallélisme encore plus élevé pourrait être mis en œuvre si le TSMU avait une latence plus importante.

Le temps total est obtenu à partir de la fréquence maximale et égale à 200 MHz qui a été atteinte lors de la synthèse. Elle correspond au temps minimal à respecter entre deux ordonnancements. En pratique, nous choisirons donc pour les trois modèles de l'OSoC, des durées de *tick* égales à 35, 29 et 16 μs.

La répartition de la surface totale suit des règles relativement similaires. Toutefois, l'unité d'allocation PMAU a une faible complexité même si elle induit une durée d'exécution im-

Elements de l'OSoC	Surface en μm^2 32 tâches-16 PE	Surface en μm^2 32 tâches-8 PE	Surface en μm^2 16 tâches-8 PE
Scheduling Unit (SCU)	1927401 (46 %)	1950027 (47,5 %)	1108764 (47,5 %)
PE and Memory Allocation Unit (PMAU)	289451 (6,9 %)	208076 (5 %)	202645 (8,7 %)
Selection Unit (SU + TMM)	954326 (22,8 %)	923134 (22,5 %)	581071 (24,9 %)
Control Unit (CU)	35839 (0,8 %)	38743 (0,9 %)	40697 (1,7 %)
Task Selection and Mgt Unit (TSMU)	893398 (21,3 %)	893398 (21,8 %)	312173 (13,4 %)
Processor Management Unit (PMU)	86237 (2 %)	87019 (2,1 %)	86291 (3,6 %)
Surface totale	4189243	4102990	2334233

TAB. 5.5 – Surface des éléments de l'OSoC en μm^2 et % de la surface totale.

portante. La surface du TSMU est tirée des résultats obtenus lors de la synthèse du RAC. La mémoire d'application n'a pas été considérée. On peut envisager une surface relativement similaire pour le TSMU, car les modifications nécessaires n'impliquent pas l'intégration de dispositifs de taille importante. En fait, la taille du RAC dépend essentiellement de celle de son réseau d'interconnexion et donc du nombre de cellules utilisées. La taille d'une cellule, ou l'ajout d'un anneau de sélection n'a que peu d'incidence.

La surface totale de l'OSoC est comprise entre 2,3 mm^2 et 4,18 mm^2 avec respectivement 3,3 Ko et 5,7 Ko de mémoires. Comme le montre la figure 5.8, la surface occupée par ces mémoires représente près de 40 % de la surface totale. Par ailleurs, comme nous l'avons annoncé, le tri nécessite une surface importante (près de 1 mm^2). A titre d'exemple comparatif, pour une fréquence de fonctionnement et une technologie similaire, un cœur d'ARM9 occupe une surface d'environ 3,26 mm^2 avec sa mémoire cache de 2 Ko et 1,96 mm^2 sans mémoire.

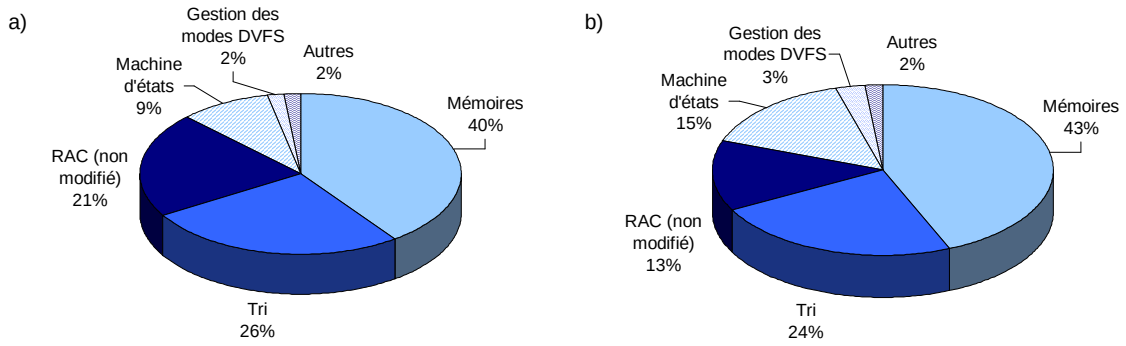

FIG. 5.8 – Répartition fonctionnelle de la surface de l'OSoC pour 32 tâches et 16 PE (a) et pour 16 tâches et 8 PE (b).

Pour évaluer la consommation d'énergie de l'architecture OSoC, nous avons utilisé l'outil Power Compiler de Synopsys. Nous avons obtenu une puissance totale dissipée égale à environ 160 mW avec 32 tâches et 110 mW avec 16 tâches (tableau 5.6). Par ailleurs, la puissance statique est respectivement d'environ 400 μW et 240 μW. Ces valeurs ne sont malheureusement pas très précises car nous n'avons pas pu associer à notre netlist une simulation après synthèse logique. Pourtant, la puissance dynamique dépend de l'activité du circuit et avec un taux d'activité de 50 %, elle représente 96 % de la puissance totale. Une étude plus fine permet de diminuer ces valeurs d'un facteur au moins égal à 2. Elle consiste à caractériser précisément l'activité des éléments de l'OSoC qui consomment le plus et d'en évaluer la consommation

d'énergie. A titre d'exemple, pour une fréquence de fonctionnement et une technologie similaire, la puissance totale dissipée par un cœur d'ARM9 est d'environ 96 mW avec sa mémoire cache.

Puissance consommée	32 tâches-16 PE	32 tâches-8 PE	16 tâches-8 PE
Statique (μW)	413	408	243
dynamique (mW)	154	163,5	110
Totale (mW)	154,4	163,8	110,3

TAB. 5.6 – Consommation d'énergie de l'OSoC. Ces chiffres ont été obtenus avec l'outil Power Compiler de Synopsys.

Synthèse FPGA

La synthèse FPGA permet une validation complémentaire et se révèle particulièrement utile lorsque nous ne disposons pas des modèles physiques de toutes nos mémoires. Elle consiste à configurer les ressources d'une structure reconfigurable, dans le but de réaliser un circuit fonctionnellement identique. Cependant, comme nous le verrons par la suite, la complexité de l'OSoC ne permet pas de considérer la solution FPGA comme une solution définitive.

Nous avons choisi le FPGA Xilinx Virtex 4 LX100 en raison de sa taille importante et de ses performances. Néanmoins, son nombre limité de ports nous a obligé à considérer un nombre de ressources de calcul inférieur. En effet, un OSoC de 32 tâches nécessite par exemple 1423 ports d'entrée/sortie. Nous avons choisi de paralléliser le plus possible les accès entre les ressources de calcul et l'OSoC. Ceci n'a pas d'incidence lorsque tous les éléments sont intégrés à l'intérieur d'un même circuit, mais est relativement dommageable lorsque l'OSoC est séparé de ses ressources de calcul. Les résultats de synthèse obtenus sont regroupés dans les tableaux 5.7 et 5.8.

Caractéristiques du FPGA	Eléments disponibles	Eléments utilisés 32 tâches-4 PE	Eléments utilisés 16 tâches-4 PE
Ports E/S	768	631 (82 %)	631 (82 %)
Nombre de slices	49152	33461 (68 %)	24424 (49 %)
Nombre de slices FF	98304	24033 (24 %)	14351 (14 %)
Nombre de LUT à 4 entrées	98304	59954 (60 %)	37903 (38 %)
Nombre de FIFO16/RAMB16s	240	24 (10 %)	21 (8 %)

TAB. 5.7 – Détail de l'occupation des ressources du FPGA. Le FPGA utilisé est un Xilinx Virtex 4 LX100. La quantité de ports étant limitée, le nombre de ressources de calcul a dû être réduit à 4.

Le FPGA est utilisé à environ 68 % avec le modèle le plus complexe synthétisé. Par ailleurs, nous avons pu utiliser les ressources spécialisées disponibles dans le FPGA pour réaliser nos mémoires et la plupart de celles adressables par le contenu. Cependant, il faut noter que le TSMU n'est ici pas considéré. Celui-ci utilisant une technologie asynchrone, la synthèse sur un FPGA synchrone engendre des difficultés supplémentaires. Ce travail ne s'inscrit pas directement dans ces travaux de thèse.

La fréquence de fonctionnement obtenue après synthèse est de 51,64 MHz. Comme le montre la figure 5.9, les performances de l'OSoC sont nettement inférieures lorsque celui-ci est synthétisé sur un FPGA. Elles restent malgré tout assez élevées pour une synthèse FPGA compte tenu de la complexité de notre architecture. Pourtant, nous verrons par la suite que la durée

Elements de l'OSoC	Nombre de cycles 32 tâches-4 PE	Nombre de cycles 16 tâches-4 PE
Scheduling Unit (SCU)	3302 (61,3 %)	1654 (59,6 %)
PE and Memory Allocation Unit (PMAU)	277 (5,1 %)	197 (7,1 %)
Selection Unit (SU + TMM)	1108 (20,6 %)	564 (20,3 %)
Control Unit (CU)	280 (5,2 %)	152 (5,5 %)
Task Selection and Management Unit (TSMU)	322 (5,9 %)	162 (5,8 %)
Processor Management Unit (PMU)	96 (1,8 %)	48 (1,7 %)
Nombre total de cycles	5385	2777
Temps total (μs)	104,28	53,78

TAB. 5.8 – Durée d'exécution des éléments de l'OSoC après synthèse FPGA. La synthèse a été effectuée à 51,64 MHz.

minimale du *tick* est déterminante pour les performances de l'architecture SCMP-LC. Néanmoins, les résultats de simulation obtenus après synthèse se sont révélés corrects et ont permis de valider matériellement l'architecture.

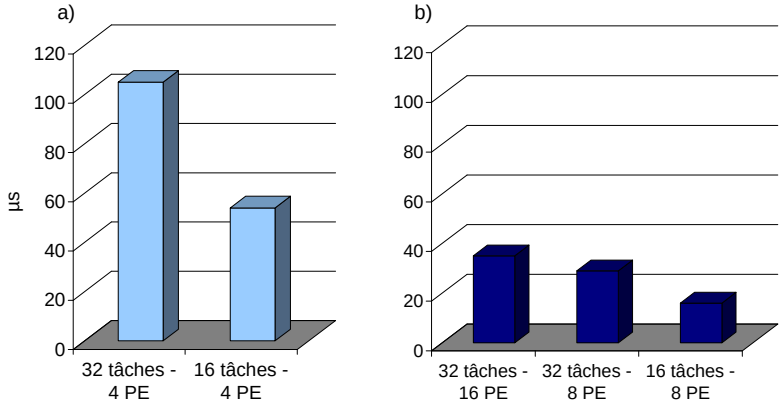

FIG. 5.9 – Bilan des temps d'exécution de l'OSoC après synthèse sur un FPGA Xilinx Virtex 4 LX100 (a). Malgré des caractéristiques dégradées, l'OSoC atteint des résultats nettement supérieurs en synthèse ASIC (b). Pour rappel, la technologie utilisée est la CMOS 130 nm à 1,2 V (low-leakage).

Améliorations possibles

Il est intéressant de se demander maintenant si les performances obtenues n'auraient pas pu être améliorées. Il est en effet possible de réduire le durée minimale du *tick* sans changer de manière significative la structure de l'OSoC. Cependant, ces modifications n'ont pas été apportées car elles engendrent une complexité supplémentaire non négligeable et ne sont pas nécessaires pour valider l'architecture. Les résultats présentés par la suite sont donc uniquement des estimations.

Toutes les recherches des tâches suivantes ou précédentes pour propager les laxités ou associer un évènement à une tâche, engendrent des temps d'exécution très importants. Typiquement, pour un OSoC de 32 tâches, chacune de ces opérations nécessite 256 cycles puisqu'une tâche peut avoir huit successeurs. Ceci représente tout de même environ 4 % du temps total

d'exécution. Une accélération peut être obtenue en multipliant les mémoires qui contiennent ces informations afin de paralléliser ces accès.

Les tableaux 5.9 et 5.10 regroupent l'ensemble des résultats qui seraient obtenus après ces modifications. Les éléments accélérés sont uniquement le SCU et le SU. En effet, le SU reçoit les fins d'exécution des ressources de calcul et doit retrouver l'évènement associé à la tâche correspondante. Ceci permet de modifier son état et d'activer la prochaine tâche à exécuter. Au contraire, le SCU s'occupe de la propagation des laxités vers les tâches suivantes.

Elements de l'OSoC	Nombre de cycles 32 tâches-16 PE	Nombre de cycles 32 tâches-8 PE	Nombre de cycles 16 tâches-8 PE
Scheduling Unit (SCU)	1344 (- 40,7 %)	1344 (- 40,7 %)	672 (- 40,6 %)
Selection Unit (SU + TMM)	560 (- 48,4 %)	560 (- 49,8 %)	280 (- 48,3 %)
Autres éléments	0 (- 0 %)	0 (- 0 %)	0 (- 0 %)
Nombre total de cycles gagnés	1904 (27,3 %)	1904 (33,2 %)	952 (30,2 %)
Temps total gagné (μs)	9,5	9,5	4,7
Temps total (μs)	25,385	19,105	11,015

TAB. 5.9 – Durée d'exécution des éléments de l'OSoC après l'exploitation de son parallélisme. En multipliant la mémoire permettant d'associer un évènement à une tâche et celle utilisée pour la propagation de la laxité, nous pouvons améliorer significativement les performances. Le pourcentage indiqué correspond au gain qui a été obtenu.

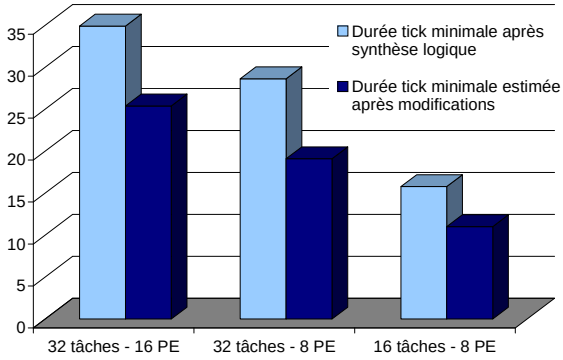

FIG. 5.10 – Comparaison des durées d'exécution de l'OSoC avec et sans modifications.

Le gain obtenu est d'environ 30 % sur le temps total d'exécution. Ceci représente une optimisation importante. Ainsi, le temps minimal entre deux ordonnancements est réduit à environ 11 μs pour un OSoC de 16 tâches. Ce gain représente près de 10 μs pour un OSoC à 32 tâches. La figure 5.10 compare ces résultats avec ceux obtenus après synthèse de l'OSoC. Le gain en performance reste significatif quel que soit le nombre de tâches.

Néanmoins, comme le montre le tableau 5.10, ces modifications engendrent une augmentation significative de la surface de l'OSoC. La taille augmente d'environ 170 %, pour atteindre plus de 11 mm^2 avec 32 tâches. Cette perspective n'est pas forcément inintéressante, cela dépend de nos besoins et sans doute aussi de la taille globale de l'architecture SCMP-LC. Le contrôle n'intervenant pas directement dans les calculs et l'obtention de résultats, la diminution de son influence augmente l'efficacité transistor et énergétique de l'architecture.

Elements de l'OSoC	Surface en μm^2 32 tâches-16 PE	Surface en μm^2 32 tâches-8 PE	Surface en μm^2 16 tâches-8 PE
Scheduling Unit (SCU)	5443466 (+ 182 %)	5444203 (+ 179 %)	3022249 (+ 172 %)
Selection Unit (SU + TMM)	4827475 (+ 406 %)	4773750 (+ 417 %)	2488522 (+ 328 %)
Autres éléments	1304925 (+ 0 %)	1227236 (+ 0 %)	641806 (+ 0 %)
Surface supplémentaire estimée (mm^2)	7,4 (+ 176 %)	7,3 (+ 179 %)	3,8 (+ 163 %)
Surface totale (mm^2)	11,6	11,5	6,2

TAB. 5.10 – Surface des éléments de l'OSoC après l'exploitation de son parallélisme. La surface de l'OSoC augmente alors de manière significative.

Dans ces travaux de thèse, il ne nous a pas semblé essentiel de réaliser ces modifications. Cependant, l'étude des performances de l'architecture SCMP-LC qui sera conduite par la suite prend en compte des durées de *tick* différentes et recouvre l'ensemble des solutions qui pourraient être obtenues après synthèse. Ceci permet de mieux se rendre compte de l'impact des performances de l'OSoC sur les performances globales et de l'intérêt d'améliorer encore le parallélisme de notre architecture.

5.1.4 Conclusion

Cette première section a présenté la méthodologie de conception et de validation qui a été mise en œuvre dans cette thèse. Toutes les fonctionnalités de l'OSoC ont pu être validées et la validation matérielle a confirmé la faisabilité d'une telle approche. La durée minimale entre deux ordonnancements est de 16 μs pour une surface de 2,3 mm^2. Nous avons également vu que des améliorations pouvaient apporter un gain significatif, même si ceci se fait au détriment de la surface occupée par l'OSoC.

Il nous faut maintenant étudier quel est l'impact de l'utilisation de l'OSoC sur les performances globales. Autrement dit, les résultats de synthèse obtenus sont-ils suffisants pour répondre aux besoins des futures applications mobiles ? La section suivante analyse les performances obtenues en fonction de la durée du *tick* et du nombre de ressources de calculs utilisées. Pour cela, les applications MPEG4-AVC et MPEG2-AAC seront implantées sur notre modèle d'architecture SCMP-LC.

5.2 Mise en œuvre d'applications et analyse des performances

5.2.1 Introduction

Les différents résultats de synthèse obtenus précédemment ne permettent pas d'évaluer les performances globales de l'architecture. Il est nécessaire de vérifier l'intérêt des solutions choisies à travers quelques exemples applicatifs. Pour cela, nous avons testé l'exécution de plusieurs applications en effectuant plus de 800 simulations. Ainsi, nous étudierons par la suite les performances de nos algorithmes ELLF et RSSR, ainsi que l'impact de la configuration avant l'exécution. Par ailleurs, nous évaluerons les performances de l'architecture SCMP-LC en fonction du parallélisme de l'application. Ce sera l'occasion d'analyser précisément quelle

est l'influence du contrôle sur le reste de l'architecture et si sa position centralisée n'a pas d'incidence significative sur les performances.

Pour généraliser nos résultats, nous ne nous attacherons pas seulement à des caractéristiques de l'OSoC identiques à celles obtenues après la synthèse logique. Ces différents OSoC permettent de mieux se rendre compte des éventuelles améliorations que l'on pourrait apporter, ou encore de l'importance d'obtenir une solution ayant une durée d'ordonnancement suffisamment faible.

5.2.2 Analyse des performances

L'algorithme ELLF

Nous avons vu dans le chapitre précédent que le choix d'un algorithme d'ordonnancement dynamique est le résultat de nombreux compromis. L'utilisation de l'algorithme ELLF ne conduit pas toujours à l'optimalité et engendre de nombreuses préemptions. Pourtant, contrairement à l'algorithme EDF, il évite les aléas d'ordonnancement et accorde la priorité des tâches en fonction de l'urgence de leur exécution. Ceci lui confère une meilleure adaptabilité aux conditions variables de l'exécution. Il reste malgré tout à évaluer l'impact des préemptions sur le temps total d'exécution. La figure 5.11 reprend quelques résultats obtenus et compare ces deux algorithmes.

Les figures 5.11-a et 5.11-d montrent un nombre de préemption globalement plus important pour l'algorithme ELLF. La différence est significative lorsqu'une application est exécutée avec un fort parallélisme sur une seule ressource de calcul. Toutefois, il faut noter que, à partir de 8 processeurs, l'application MPEG4-AVC inverse cette tendance et produit moins de préemptions que l'EDF. Avec un parallélisme de 32 tâches, ce dernier est identique entre les deux algorithmes. Ce nombre limité de préemptions est lié aux dépendances entre les tâches qui limitent la concurrence. Du reste, il est dû au mécanisme mis œuvre pour réduire les préemptions et éviter le phénomène de *trashing*.

Comme le montre la figure 5.11-b, malgré le nombre plus important de préemptions, la différence du taux d'occupation reste négligeable. D'ailleurs, bien souvent notre algorithme obtient de meilleurs résultats que l'algorithme EDF. La figure 5.11-e confirme ces résultats. Le gain obtenu atteint même près de 5 %. Ceci est dû aux mécanismes de préemption et de migration qui sont facilités dans l'architecture SCMP-LC. De plus, l'ordonnancement suivant la laxité engendre de très bons résultats et compense le plus souvent la pénalité de préemption. L'étude des temps d'exécution globaux dans les figures 5.11-c et 5.11-f confirme cette analyse.

L'influence des préemptions n'est finalement pas très importante. Ceci était prévisible puisqu'elles sont concurrentes entre elles, et que leur durée est inférieure au temps minimal entre deux ordonnancements. Ceci explique d'ailleurs pourquoi la durée du *tick* n'a pas d'incidence majeure sur les résultats obtenus. Néanmoins, il ne semble pas envisageable d'obtenir un *tick* inférieur à 2 μs pour la conception d'un OSoC. D'ailleurs, ça n'aurait sans doute aucun intérêt, à moins de ne plus considérer de préemptions ou de migrations.

Pour conclure, l'algorithme ELLF obtient des performances relativement correctes compte tenu de son potentiel de préemption élevé. Ce choix ne conduit donc pas à une dégradation des performances. Même si celles-ci sont quelquefois en retrait lorsque le parallélisme est élevé et les ressources sont limitées, elles sont bien supérieures à celles obtenues avec l'algorithme

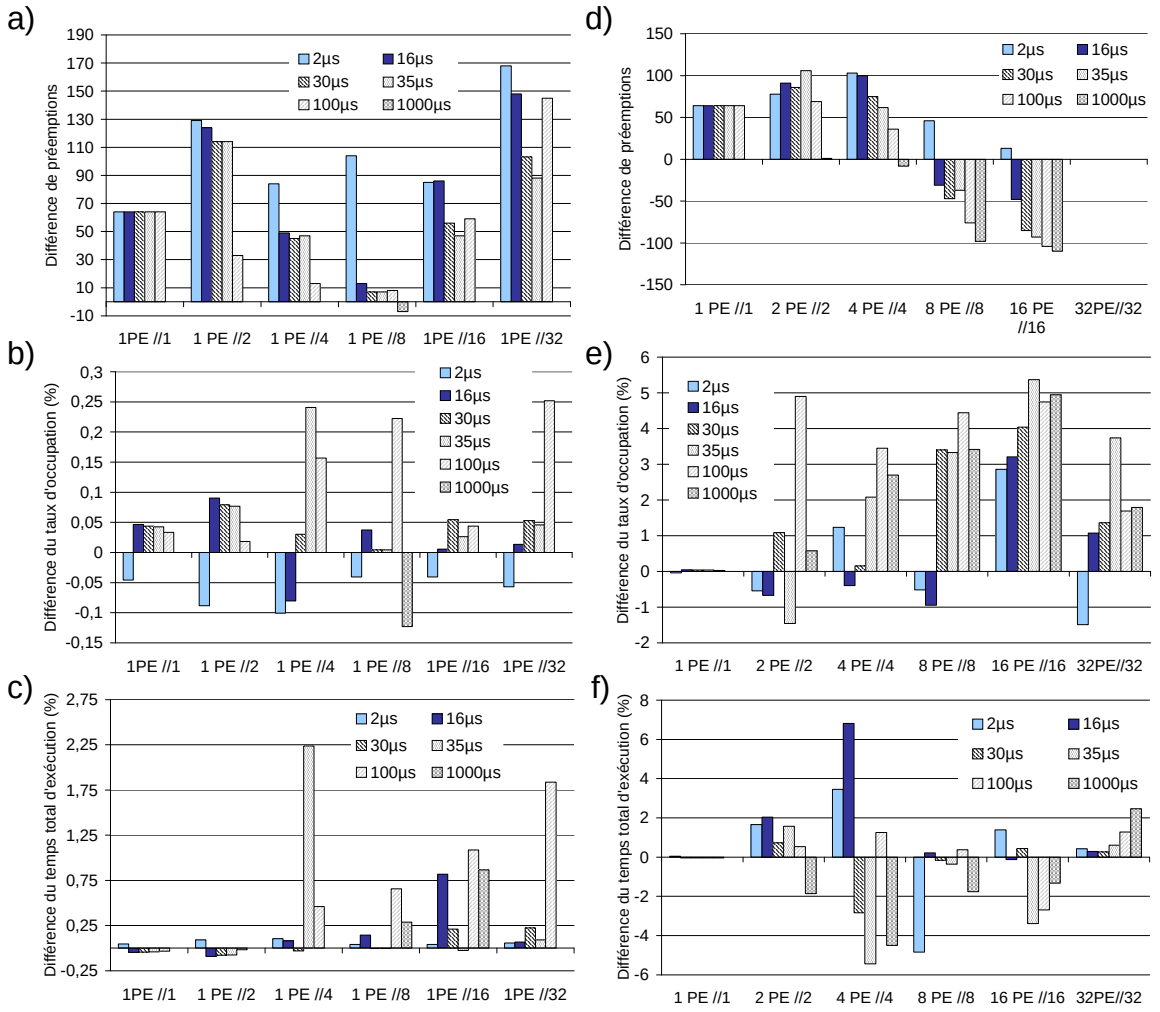

FIG. 5.11 – Etude et comparaison des algorithmes ELLF et EDF. Pour cela, nous avons implanté l'encodeur MPEG4-AVC pour des durées de *tick* variant de 2 à 1000 μs. Les figures $\{a, b, c\}$ représentent les résultats obtenus sur une seule ressource de calcul pour un parallélisme de l'application allant de 1 à 32 tâches. Au contraire, les figures $\{d, e, f\}$ considèrent un nombre de ressources égal au niveau du parallélisme. Les figures a et d étudient la différence du nombre de préemptions entre les algorithmes ELLF et EDF. Une valeur positive signifie que l'algorithme ELLF préempte plus souvent. Les figures b et e représentent la différence du taux d'occupation des ressources de calcul. Un gain positif montre un taux d'occupation supérieur pour l'algorithme ELLF. Enfin, les figures c et f illustrent la différence relative du temps d'exécution total par rapport à la durée obtenue avec l'algorithme EDF. Un pourcentage positif indique un temps d'exécution plus long avec l'algorithme ELLF.

EDF dans le cas contraire. Ces simulations démontrent donc l'intérêt de notre approche et de son intégration dans l'OSoC malgré une complexité supérieure.

L'algorithme RSSR

La validation de notre politique de réduction de la consommation d'énergie est plus difficile à évaluer. Tout d'abord, nous ne disposons pas de processeurs Intel XScale pour effectuer nos mesures. Ce processeur a été choisi pour sa gestion efficace de ses modes DVFS. C'est la

raison pour laquelle, il n'a pas été possible d'exécuter de vraies applications et d'en analyser la consommation d'énergie. Nous avons choisi une application de test composée de 17 tâches dont la durée d'exécution moyenne est égale à 540 μs. La consommation d'énergie des ressources de calcul et de mémorisation a été évaluée à partir de leurs documentations techniques.

La figure 5.12 reprend quelques résultats obtenus après simulation. L'utilisation des modes de repos réduit la consommation d'énergie de près de 30 %. D'autre part, le gain obtenu avec la gestion des modes DVFS s'accroît avec l'augmentation de l'écart qui existe entre le WCET et la durée d'exécution constatée. Ceci est tout à fait normal puisque plus le *slack time* est important et plus les modes DVFS activés engendrent une économie d'énergie. Pour une même durée entre deux ordonnancements, l'utilisation des modes DVFS apporte un gain supplémentaire allant de 2 à 30 %. Néanmoins, en pratique, lorsque le processeur n'exécute plus de code, il consomme moins d'énergie et ceci n'a pas été pris en compte dans ces simulations, faute d'informations constructeurs. Les réductions de la consommation d'énergie seraient donc légèrement inférieures à celles obtenues dans ces simulations.

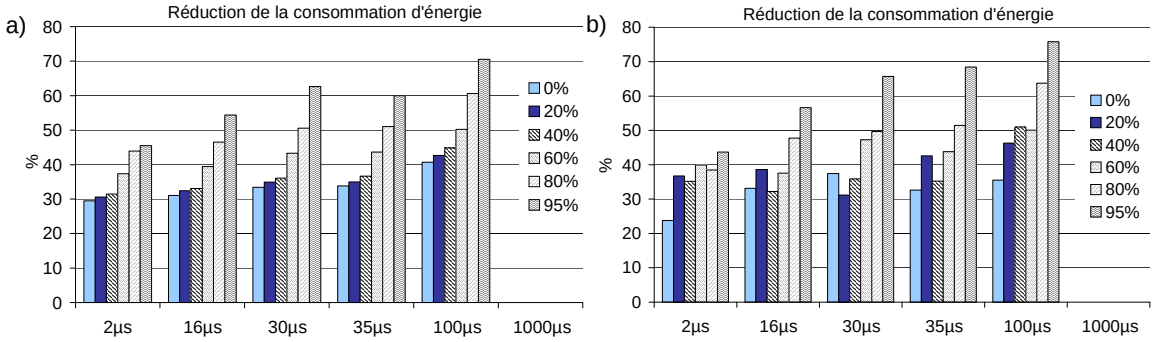

FIG. 5.12 – Evaluation de l'algorithme RSSR. Nous avons utilisé une application de test composée de 17 tâches d'une durée moyenne égale à 540 μs. Le parallélisme maximal de notre application est de 3 tâches. Chacune de ces tâches s'exécute sur un processeur Intel XScale PXA270. Ces simulations ont été effectuées en fonction de la durée du *tick* et de l'erreur sur le WCET. Par exemple, 20 % signifie que la durée d'exécution de la tâche est comprise entre 80 % et 100 % du WCET. L'erreur suit une loi uniforme. La figure a regroupe les résultats obtenus sur un seul processeur, tandis que la figure b considère 3 processeurs. Les figures $\{a, b\}$ représentent la différence entre la consommation d'énergie obtenue après simulation et celle que l'on aurait atteinte, sans gestion des modes DVFS et activation des modes de repos. En condition normale, nous avons considéré une puissance consommée égale à 925 mW par processeurs XScale, ainsi que 26,4 mW par mémoire distribuée et partagée (mémoires de 8 Ko à 200 MHz pour une tension d'alimentation de 1,2 V).

La durée entre deux ordonnancements a aussi une incidence qui n'était pas si évidente au départ. Plus celle-ci augmente et plus la gestion des modes DVFS apporte une réduction de la consommation d'énergie. Cependant, lorsque la durée du *tick* est trop élevée, le gain observé reste inférieur à 0,35 %. Pour une durée de *tick* importante, le temps d'inactivité du processeur est potentiellement long. En attendant une nouvelle demande d'exécution, il maintient sa fréquence d'exécution et sa tension d'alimentation. Par conséquent, les modes d'économie d'énergie sont actifs plus longtemps. Par contre, si la période d'inactivité devient trop longue, voire supérieure à la durée moyenne des tâches, elle conduit à une augmentation trop importante de la durée totale d'exécution. Alors, la réduction d'énergie répartie sur toute la durée d'exécution n'a que peu d'influence. L'association de l'algorithme RSSR et d'un *tick* trop grand n'a donc pas de sens.

La parallélisation de l'application sur plusieurs processeurs a un impact plutôt positif. En fait, le nombre d'activations de mode DVFS peut être multiplié par 4. L'application étant accélérée, la laxité globale de l'application est supérieure. Alors, il est possible de ralentir davantage la fréquence de fonctionnement tout en respectant les échéances des tâches. De plus, une échéance plus éloignée dans le temps offre davantage de possibilités d'activer un mode DVFS lorsque le *slack time* est suffisant.

Finalement, la réduction de la consommation d'énergie est assez conséquente. Elle peut aller de 30 % à près de 80 %. Cela alors même que la complexité matérielle nécessaire pour la mise en œuvre de notre politique de basse consommation est relativement faible. Elle représente seulement 2 % de la surface de l'architecture de l'OSoC.

La configuration avant l'exécution

Le mécanisme de double jetons mis en œuvre dans le TSMU offre la possibilité de configurer les tâches avant qu'elles ne soient éligibles. La configuration consiste à précharger les mémoires distribuées et partagées associées à chacune des tâches. Ainsi, il semble assez évident que ce dispositif de très faible complexité matérielle doit apporter une amélioration des performances globales du système. Pour s'en convaincre, nous avons étudié l'influence de la configuration sur deux applications différentes : l'encodeur MPEG4-AVC et le décodeur MPEG2-AAC. La figure 5.13 résume les résultats obtenus.

Les figures 5.13-a et 5.13-b montrent une amélioration importante du taux d'occupation et du temps total d'exécution pour l'application MPEG2-AAC. Celui-ci va de 10 % avec un *tick* de 2 μs à plus de 75 % avec un *tick* de 1 ms. Ainsi, l'augmentation de la durée entre deux ordonnancements augmente l'influence de la configuration. Ceci est normal puisque si la configuration n'est pas prête, il faut attendre plusieurs nouveaux ordonnancements pour demander l'exécution de la tâche. Cela dit, au vue de la faible complexité matérielle nécessaire pour mettre en œuvre les demandes de configuration, un gain de 10 % reste très intéressant.

Les gains obtenus avec l'application MPEG4-AVC sont nettement inférieurs. Ceci est dû au nombre important de convergences et de divergences en ET dans l'application. Elles empêchent l'OSoC d'anticiper les configurations. De plus, comme les configurations sont séquentielles, leur durée dépend du nombre de configurations simultanées. En fait, le pire temps de configuration est égal au nombre maximal de tâches actives multiplié par le temps de configuration d'une seule tâche. Ainsi, lorsque le parallélisme de l'application est important, il est tout à fait possible que la configuration ne soit pas finie avant la demande d'exécution. Cependant, une durée d'exécution suffisamment longue suffit pour négliger ce phénomène. D'autre part, les configurations des mémoires peuvent se faire en concurrence avec les demandes d'exécution ou de préemption. D'ailleurs, pour cette application le temps d'exécution global est tout de même réduit d'environ 10 % et ceci demeure relativement intéressant.

Ainsi, la configuration avant l'exécution améliore les performances du système, même si elle est limitée par le parallélisme ou la géométrie des graphes des applications. Sa très faible complexité matérielle en fait une solution particulièrement intéressante. Ce constat serait encore plus marqué avec des durées de configuration plus importantes.

La scalabilité et les performances

Les précédents résultats ont montré que les solutions mises en œuvre dans l'OSoC apportent globalement toutes un gain lors de l'exécution d'applications pourtant très différentes. Nous

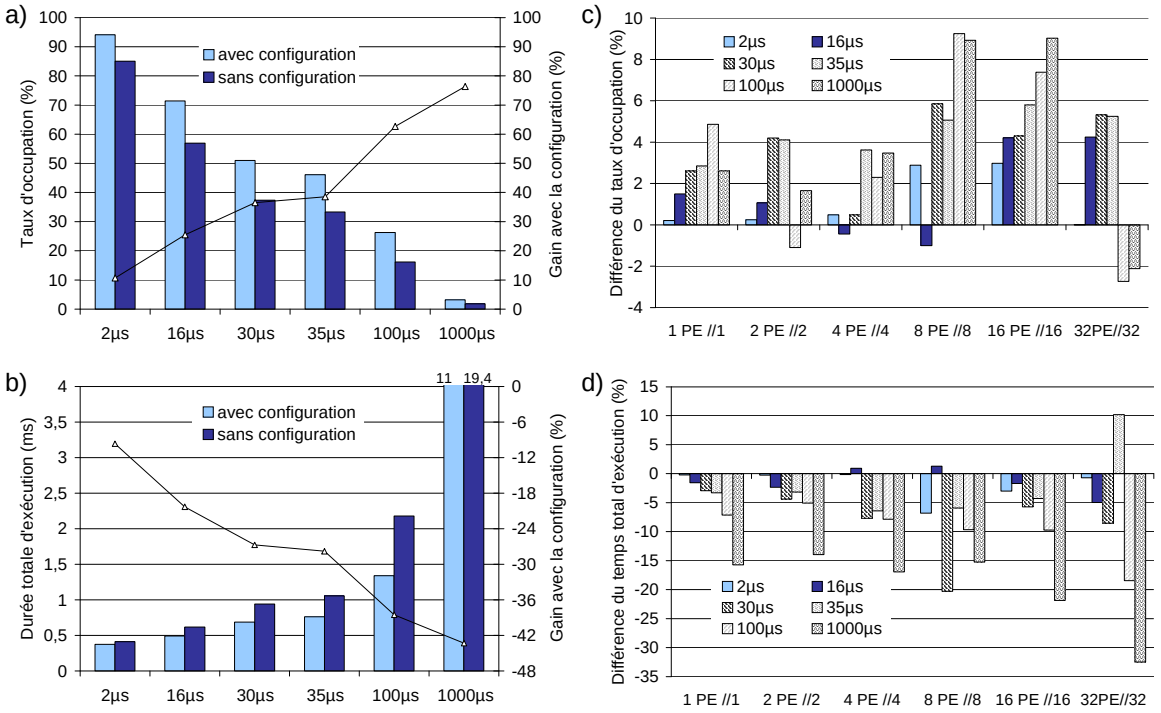

FIG. 5.13 – Etude de la configuration avant l'exécution des tâches. Pour cela, nous considérons le décodeur MPEG2-AAC (figures $\{a, b\}$) et l'encodeur MPEG4-AVC (figures $\{c, d\}$), pour des durées de *tick* allant de 2 à 1000 μs. Avec cette dernière application, le nombre de ressources de calcul est égal au niveau du parallélisme. Les figures a et c représentent la différence du taux d'occupation avec et sans la configuration avant l'exécution. Dans la figure c, un gain positif signifie que le taux d'occupation est supérieur avec la configuration. Enfin, les figures b et d montrent la différence entre les durées totales d'exécution. Dans la figure d, un gain négatif indique un temps d'exécution plus court avec la configuration.

allons maintenant étudier les performances de l'architecture SCMP-LC en fonction du parallélisme de l'application MPEG4-AVC. La gestion du parallélisme sur plusieurs processeurs reste complexe et nous devons enfin pouvoir évaluer l'influence de la migration des tâches et du contrôle sur l'occupation des ressources de calcul.

La figure 5.14 montre un taux d'occupation des ressources de calcul très important lorsque la durée entre deux ordonnancements est faible. En fait, lorsque celle-ci est inférieure à 20 μs, plus de 90 % du temps est utilisé pour effectuer des calculs. Par ailleurs, le taux d'occupation reste relativement constant quel que soit le parallélisme de l'application. Nous avons donc réussi à exploiter un haut niveau de parallélisme de tâches sans pénalités supplémentaires. Cependant, ces très bonnes performances ont tendance à se dégrader lorsque la durée du *tick* augmente. Pour garder un taux d'occupation élevé, il faut privilégier une durée inférieure à 40 μs.

En fait, le taux d'occupation des ressources de calcul dépend du rapport entre la durée de la tâche et de celle du *tick*. La figure 5.15 représente l'allure de la courbe obtenue avec une durée entre deux ordonnancements égale à 2 μs. La durée de la tâche doit être environ dix fois supérieure à celle du *tick* pour atteindre un taux d'occupation élevé. En fait, il faut privilégier un grain de calcul adapté à l'OSoC. L'utilisation de ressources de calcul très performantes doit être compensée par des tâches plus complexes et surtout plus longues à exécuter.

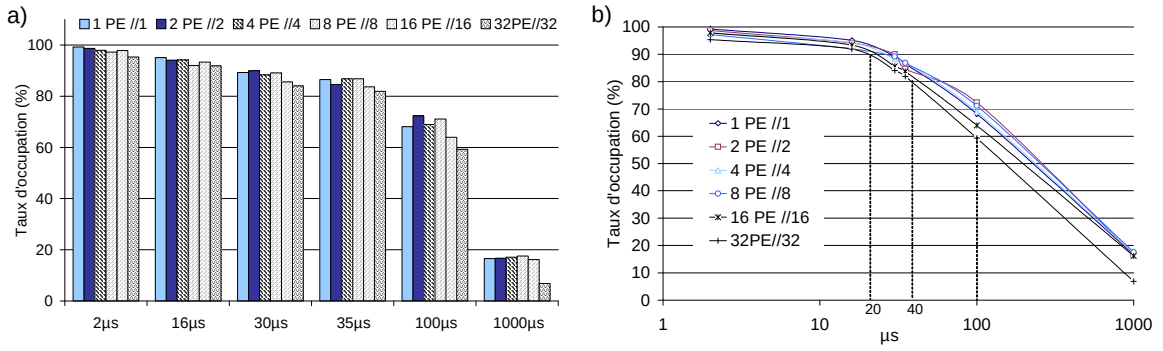

FIG. 5.14 – Taux d'occupation des ressources de calcul avec l'encodeur MPEG4-AVC. Le nombre de ressources de calcul est égal au parallélisme maximal de l'application. L'étude est effectuée pour différentes valeurs de *tick* allant de 2 à 1000 μs. Les figures *a* et *b* reprennent sous des formes différentes ces résultats. La figure *b* montre que pour un *tick* inférieur à 20 μs, le taux d'occupation est supérieur à 90 % lors de l'exécution de l'encodeur MPEG4-AVC.

L'étude de l'accélération obtenue en parallélisant l'encodeur MPEG4-AVC montre également de très bons résultats (figure 5.16). Quelle que soit la durée du *tick* considérée, l'accélération reste très forte et traduit une excellente exploitation de la puissance de calcul. Et cela alors même que l'application, avec un parallélisme de 32 tâches, est composée de plus de 400 tâches différentes. Malgré la légère dégradation subie à partir de huit ressources de calcul, le taux d'accélération atteint près de 25 avec 32 ARM 920T.

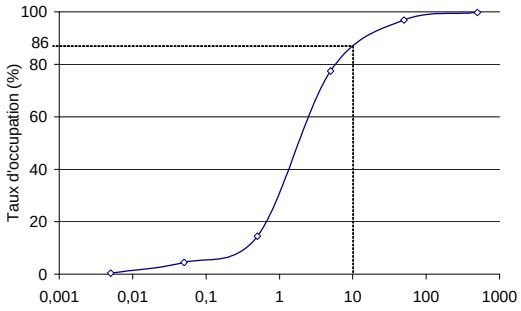

FIG. 5.15 – Occupation des ressources en fonction de la durée de la tâche et du temps entre deux ordonnancements. Cette courbe représente le taux d'occupation en fonction du rapport *durée tâche/durée tick*. Par exemple, pour un *tick* de 10 μs il faut que la durée moyenne des tâches de l'application soit supérieure à 100 μs pour obtenir un taux d'occupation égal à 86 %.

Cependant, la figure 5.17 met en évidence des performances insuffisantes pour répondre aux besoins de notre application. Nous avons besoin d'au moins 32 processeurs ARM 920T pour espérer traiter 30 images CIF (352 × 288 pixels) par seconde. Des ressources de calcul plus performantes et dotées d'une meilleure efficacité transistor auraient été mieux adaptées pour venir à bout des 2,5 GOPs de cette application. Par exemple, l'utilisation de processeurs VLIW ou MIPS 24KE aurait permis d'atteindre des performances supérieures. Cela dit, il faut un processeur Intel Pentium 4 HT cadencé à 3,2 GHz (Dell Precision 370) pour atteindre des performances similaires. L'exploitation des ressources de calcul est donc particulièrement

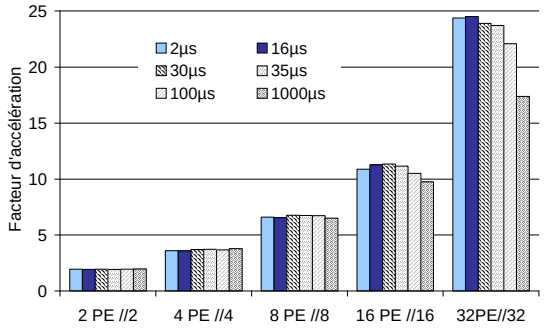

FIG. 5.16 – Accélération et exploitation du parallélisme avec l'encodeur MPEG4-AVC. Le gain obtenu est calculé par rapport au temps d'exécution sur une seule ressource de calcul pour une durée de *tick* similaire. Avec 32 processeurs, on atteint un gain d'environ 25.

performante dans l'architecture SCMP-LC, puisqu'à puissance de calcul crête équivalente nous disposerions de 72 ARM 920T (soit 12,8 GOPs). Ainsi, nous exploitons 43 % des performances crêtes des ARM 920T (sachant que le nombre d'instructions par cycle est de l'ordre de 0,5), alors que le Pentium 4 exploite seulement 20 % de ses capacités théoriques.

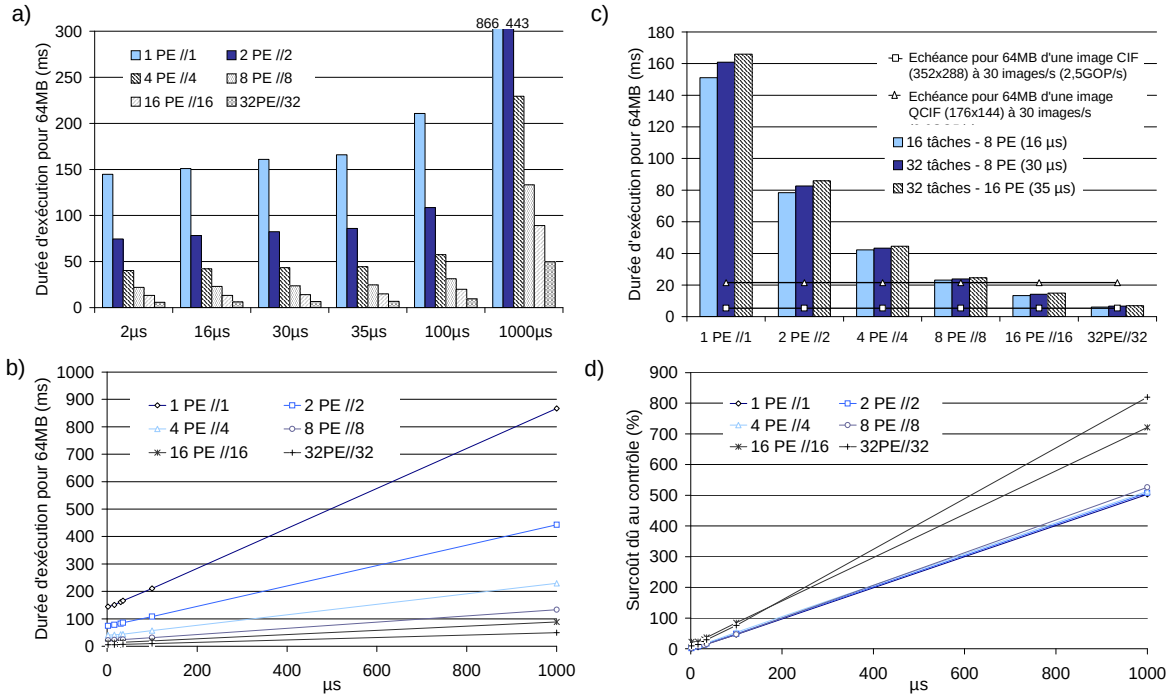

FIG. 5.17 – Evaluation du temps total d'exécution avec l'encodeur MPEG4-AVC. Ces résultats dépendent de la durée entre deux ordonnancements et du parallélisme de l'application. Le nombre de ressources de calcul est égal au niveau du parallélisme. Les figures *a* et *c* sont similaires. La figure *c* reprend uniquement les résultats pour des caractéristiques de l'OSoC identiques à celles obtenues après synthèse ASIC. La figure *b* représente l'évolution linéaire du temps d'exécution en fonction de la durée du *tick*. Enfin, la figure *d* montre le surcoût temporel induit par le contrôle en fonction de la durée du *tick*. Par exemple, un *tick* de 100 μs peut multiplier le temps d'exécution total par 2.

Les figures 5.17-b et 5.17-d confirment l'intérêt de privilégier la durée de *tick* la plus faible possible dans ce cadre applicatif. Dès l'utilisation d'un OSoC ayant un temps entre deux ordonnancements supérieur à 100 μs (ce qui est le cas avec un FPGA Xilinx Virtex 4 LX100), le temps passé dans le contrôle devient identique à celui passé à effectuer les calculs. L'efficacité transistor est alors considérablement réduite. D'ailleurs pour une durée de *tick* d'une milliseconde, le surcoût induit par l'OSoC dépasse 800 % du temps total d'exécution sans contrôle. Pour cette application et avec le grain de calcul choisi, un OSoC ayant une durée de *tick* aussi importante n'a aucun sens.

Nous allons maintenant nous intéresser plus en détail aux surcoûts temporels engendrés par l'OSoC. Si on fait un parallèle entre les figures 5.18-$\{a, c\}$ et 5.18-$\{b, d\}$, qui représentent le taux d'occupation des ressources de calcul et la part due au contrôle, on s'aperçoit que la somme des temps en attente de configuration ou utilisés par le contrôle, ne correspond pas au temps d'attente des ressources de calcul. Par exemple pour 32 ARM 920T et une durée de *tick* égale à 16 μs, près de 30 % du temps total est passé à attendre la configuration des ressources, alors que le taux d'occupation reste supérieur à 90 %. Ceci est dû au mécanisme de pré-chargement, mais surtout au parallélisme qui existe entre le calcul et le contrôle. L'exécution de tâches concurrentes continue pendant les configurations, les préemptions ou les synchronisations. La séparation du contrôle et du calcul permet donc un gain très important et compense largement les limitations de notre structure asymétrique.

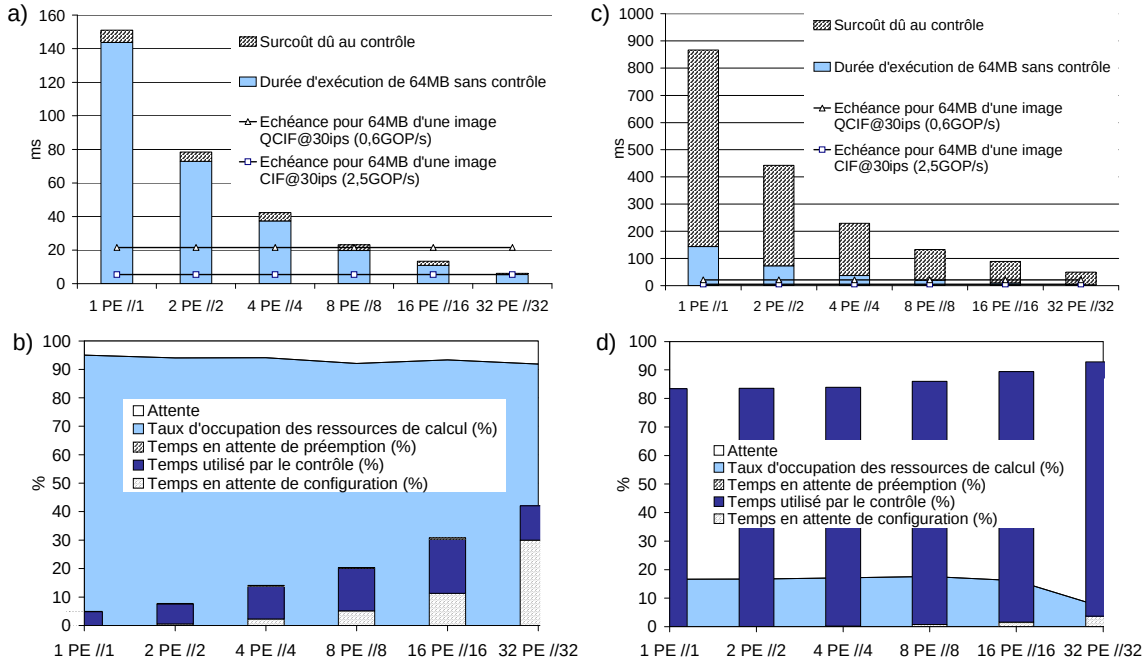

FIG. 5.18 – Evaluation du temps total d'exécution avec l'encodeur MPEG4-AVC et analyse du surcoût induit par le contrôle. Les figures *a* et *c* considèrent un OSoC ayant une durée entre deux ordonnancements égale respectivement à 16 et 1000 μs. Les figures *b* et *d* illustrent les coûts engendrés par l'OSoC, les configurations et les préemptions sur le temps total d'exécution respectivement pour 16 et 1000 μs.

Ensuite, nous retrouvons comme précédemment des temps de configuration plus importants lorsque le nombre de tâches augmente, et surtout une pénalité de l'OSoC très importante lorsque la durée entre deux ordonnancements est grande. Cependant, le surcoût temporel dû aux préemptions reste négligeable quelles que soient les caractéristiques de l'OSoC. Là encore, le parallélisme qui existe entre le contrôle et le calcul en est la cause principale.

En fait, le comportement de l'OSoC est totalement prédictible et déterministe. Il ne varie pas en fonction de la complexité des applications traitées ou du parallélisme. En revanche, les temps de préemption ou de configuration peuvent avoir une incidence sur la durée d'exécution. Ceux-ci peuvent malgré tout être pris en compte lors de l'évaluation des WCET.

5.2.3 Conclusion

Cette seconde section a permis d'analyser le comportement et les performances de l'architecture OSoC. Notre approche centralisée, le choix de l'algorithme ELLF et l'intérêt de tous les mécanismes mis en œuvre devaient être démontrés.

Tout d'abord, les différents algorithmes d'ordonnancement ou d'allocation des tâches se sont révélés très performants. Pour la première fois, l'algorithme ELLF a pu être mis en œuvre pratiquement et comparé à une autre solution. Les premières intuitions et l'étude des différentes solutions existantes se sont révélées concluantes.

Il est vrai que nous n'avons pas testé la gestion de l'hétérogénéité. L'utilisation de ressources hétérogènes permet de réduire les temps d'exécution des tâches, ou d'apporter plus de souplesse grâce à la reconfigurabilité. Cependant, les conditions les plus difficiles pour l'OSoC sont obtenues avec des plateformes homogènes autorisant un grand nombre de migrations.

L'implantation de graphes d'application complexes comme le MPEG4-AVC a mis en évidence une gestion efficace des dépendances de données et de contrôle, ainsi que des synchronisations. Mais surtout, malgré un haut niveau de parallélisme, nous n'avons pas réussi à mettre en défaut cette approche asymétrique. Ceci en particulier grâce à une séparation explicite du contrôle et du calcul et aux mécanismes performants de migration et de préemption des tâches.

Finalement, seule la durée minimale entre deux ordonnancements s'est révélée très importante. Plus précisément, le grain de calcul doit être correctement adapté aux performances de l'OSoC. Pour les exemples applicatifs choisis, les différentes versions de l'OSoC synthétisées se sont toutes distinguées par leur niveau de performance. L'intérêt de réaliser un OSoC ayant une durée de *tick* inférieure dépend uniquement des besoins applicatifs ou des performances des ressources de calcul.

Avant de comparer ces résultats avec les autres solutions actuellement commercialisées ou présentes dans la littérature, nous allons nous intéresser à une version logicielle de l'OSoC. La fabrication et la conception d'un circuit engendrent des coûts très importants. Par conséquent, il est important d'envisager toutes les alternatives possibles avant de conclure définitivement.

5.3 Comparaisons matériel/logiciel

Le deuxième chapitre a déjà mis en évidence les limitations des solutions de contrôle logicielles. Néanmoins, pour s'en convaincre nous avons décidé de réaliser un exécutif temps-réel

disposant des mêmes fonctionnalités que l'OSoC et d'en évaluer les performances. Ce travail a été réalisé par J-L. Leroy et a consisté à concevoir un noyau temps-réel basé sur l'algorithme ELLF pour des systèmes multiprocesseurs [287].

Pour cela, nous avons utilisé notre plateforme d'évaluation Cogent CSB637 et son processeur ARM 920T cadencé à 180 MHz. En fait, l'architecture SCMP-LC est composée d'un ARM 920T pour supporter l'exécution de notre noyau et remplacer l'OSoC. Les ressources de calcul sont remplacées par des décompteurs accessibles par le bus interne du processeur. L'exécution de la tâche est simulée par la décrémentation de la durée d'exécution initiale de la tâche obtenue sur l'ARM 920T seul. Enfin, comme dans le modèle fonctionnel de l'architecture, les autres ressources n'ont pas eu besoin d'être prises en compte.

5.3.1 Caractéristiques du noyau temps-réel

En fait, nous n'avons pas intégré l'ensemble des fonctionnalités de l'OSoC. Comme nous le verrons par la suite, la complexité du noyau est telle qu'il n'a pas été nécessaire de poursuivre nos développements pour démontrer son inefficacité. Par ailleurs, la mise en œuvre logicielle impose de modifier les accès aux données et de supprimer le parallélisme de l'architecture matérielle.

Contrairement à un noyau temps-réel classique, un nouvel ordonnancement a lieu périodiquement et le calcul des priorités est effectué au cours de l'exécution. La prise en compte des évènements et des interruptions se fait seulement au début de chaque nouveau *tick*. La durée minimale entre deux ordonnancements à considérer correspond au pire temps d'exécution du contrôle. En effet, tout nouvel ordonnancement doit avoir lieu après la fin des ordonnancements précédents pour maintenir la cohérence du système.

Tout d'abord, la sélection des tâches éligibles est effectuée en parcourant des listes chaînées de structure de données. Comme dans le TSMU, ces listes représentent les graphes d'application et tiennent compte des dépendances de contrôle et de données. Cependant, les liens matériels entre les tâches et le parallélisme n'ont pas de correspondance logicielle.

D'autre part, l'utilisation d'idendifiants pour accéder aux caractéristiques d'une tâche s'est révélée très pénalisante pour notre noyau. La solution matérielle qui consiste à concevoir des mémoires adressables par le contenu n'est pas réalisable en logiciel. En effet, tous les accès aux données sont séquentiels. Cette solution est néanmoins inévitable pour contrôler plusieurs tâches appartenant à des applications différentes et créer dynamiquement de nouvelles applications.

Ensuite, nous n'avons mis en œuvre que l'ordonnancement des tâches temps-réel non périodiques et le préchargement des mémoires. L'algorithme LTF et les mécanismes de gestion de la consommation d'énergie n'ont pas été réalisés. L'algorithme de tri utilisé est de type *quick sort* et n'offre pas de parallélisme comme le tri systolique. Par ailleurs, il est impossible de créer, suspendre ou interrompre dynamiquement une application. Même si les principaux mécanismes ont été réalisés pour en évaluer la complexité, la gestion dynamique ne concerne que les tâches d'une même application.

Enfin, les mécanismes d'allocation logiciels sont similaires à ceux présents dans l'OSoC. Ils permettent la migration et la préemption des tâches sur de multiples ressources de calcul. Par contre, nous n'avons pas considéré la gestion des ressources hétérogènes ou de mémorisation.

Par conséquent, le noyau temps-réel que nous avons réalisé intègre un peu moins de fonctionnalités que l'OSoC. Cependant, nous avons considéré les services qui nécessitent le plus d'accès mémoire et qui pénalisent le plus son exécution. Les résultats qui sont présentés dans la sous-section suivante sont donc proches de ceux d'un noyau temps-réel logiciel identique à l'OSoC.

5.3.2 Analyse des résultats et des performances

Le tableau 5.11 résume les durées minimales à considérer entre deux ordonnancements pour des configurations identiques à celles choisies pour la synthèse logique. Les durées d'exécution obtenues sont comprises entre 1,47 et 6,26 ms. A titre de comparaison, les temps d'exécution de l'OSoC après synthèse logique sont respectivement de 16 et 35 μs, soit un coefficient d'accélération compris entre 92 et 179.

Durée d'exécution	32 tâches-16 PE	32 tâches-8 PE	16 tâches-8 PE
Sélection	33417 (2,9 %)	33417 (5,1 %)	18063 (6,8 %)
Ordonnancement	643959 (57,1 %)	497646 (76,7 %)	186956 (70,4 %)
Allocation	449778 (39,9 %)	117412 (18,1 %)	60512 (22,8 %)
Total (cycles)	1127154	648475	265531
Total (ms)	6,26	3,60	1,47

TAB. 5.11 – Durée d'exécution de l'OSoC logiciel pour des caractéristiques similaires aux OSoC synthétisés. Les durées sont exprimées en nombre de cycles et en pourcentage de la durée totale d'exécution du noyau. Le processeur ARM 920T est cadencé à 180 MHz.

La répartition du temps d'exécution montre une nette prédominance pour l'ordonnancement des tâches. L'augmentation de la durée du *tick* semble être conditionnée en grande partie par l'étape d'ordonnancement.

Ceci est confirmé par la figure 5.19 qui montre les performances du noyau pour des nombres de tâches ou de ressources de calcul différents. Tout d'abord, on remarque que la solution mise en œuvre pour la sélection des tâches n'a pas une incidence très importante sur les temps d'exécution. Sa contribution reste toujours inférieure à 14 % et diminue avec le nombre de tâches et de ressources de calcul. Nous pouvons remarquer également que la solution logicielle est environ 100 fois moins performante que le RAC. En effet, à fréquence de fonctionnement équivalente, la sélection d'au plus 32 tâches actives nécessite 1,6 μs avec le RAC contre 167 μs avec l'OSoC logiciel. Sans compter que la création dynamique des tâches n'a pas été réalisée.

Ensuite, les durées d'ordonnancement ou d'allocation augmentent avec le nombre de PE mais surtout avec le nombre de tâches actives simultanément. Ceci est dû principalement au temps nécessaire pour propager les laxités et effectuer le tri entre les tâches, ainsi que pour rechercher une ressource de calcul libre ou exécutant une tâche moins prioritaire. Les structures de données sont de plus en plus importantes à manipuler et pénalisent les recherches des identifiants dans la liste des tâches actives. De plus, le nombre d'itérations des algorithmes mis en œuvre pour l'ordonnancement et l'allocation dépend directement du nombre de tâches ou de ressources de calcul. Ceci n'a cependant aucune incidence sur le déterminisme de notre noyau puisque ces grandeurs n'évoluent pas au cours de l'exécution.

Si on s'intéresse maintenant à la surface silicium nécessaire pour exécuter notre noyau, nous pouvons constater que cette solution occupe un espace 1,6 à 2,5 fois plus important

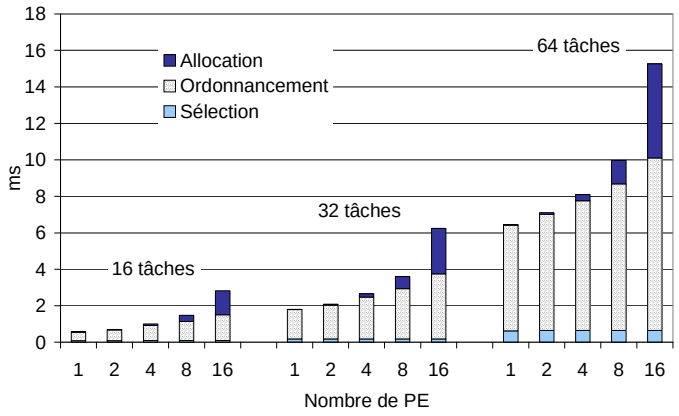

FIG. 5.19 – Durée d'exécution de l'OSoC logiciel en fonction du nombre de tâches et de ressources de calcul. Le temps d'ordonnancement augmente considérablement avec le nombre de tâches.

que les versions synthétisées (tableau 5.12). La figure 5.20 met en évidence l'augmentation importante de la surface engendrée par notre solution logicielle. Cependant, la taille du noyau dépend beaucoup du nombre de tâches et l'écart avec l'OSoC matériel se réduit à mesure que ce nombre augmente. D'autre part, la consommation d'énergie est inférieure à celle obtenue lors de la synthèse logique. Néanmoins, nous pouvons espérer obtenir des chiffres équivalents ou inférieurs en pratique (voir la sous-section 5.1.3). De plus, pour un OSoC de 32 tâches et 16 PE la consommation d'énergie pendant un *tick* est de 5,4 μJ, alors qu'un OSoC logiciel similaire consomme 638,5,μJ. Ceci représente une augmentation d'un facteur supérieur à 100.

Autres Caractéristiques	32 tâches-16 PE	32 tâches-8 PE	16 tâches-8 PE
Surface estimée (mm^2)	6,64 (74 Ko)	6,64 (74 Ko)	5,88 (43 Ko)
Puissance consommée (mW)	102	102	78

TAB. 5.12 – Surface et consommation de l'OSoC logiciel. Le processeur ARM 920T occupe une surface de 4,7 mm^2 et consomme 44 mW. Nous avons choisi une technologie 130 nm à 1,2 V et des mémoires cadencées à la fréquence du processeur. Les chiffres présentés sont des estimations.

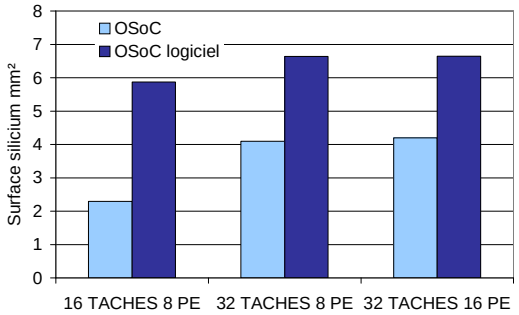

FIG. 5.20 – Comparaison de la surface silicium occupée par un OSoC matériel et logiciel

Pour mieux se rendre compte de la pénalité induite par notre noyau temps-réel, nous avons implanté l'application MPEG4-AVC et analysé les performances obtenues. La figure 5.21-a représente le temps total d'exécution pour différents OSoC logiciels. Le surcoût engendré par le contrôle est compris entre 200 et 3600 %. Par ailleurs, la figure 5.21-b met en évidence une très faible occupation des ressources de calcul due à une durée de *tick* élevée.

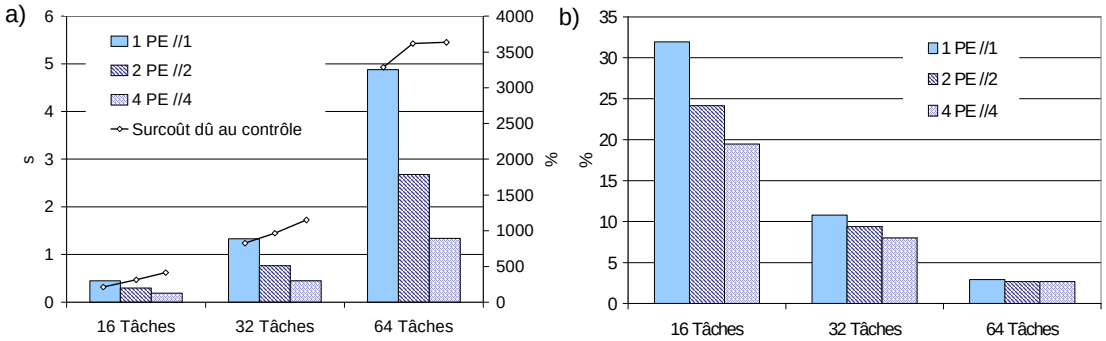

FIG. 5.21 – Evaluation du temps total d'exécution et de l'occupation des ressources de calcul avec l'encodeur MPEG4-AVC. La figure *a* montre le temps total d'exécution obtenu pour des versions de notre noyau constituée de 16, 32 et 64 tâches. Le nombre de ressources de calcul est égal au parallélisme maximal de l'application. La figure *b* donne le taux d'occupation des ressources de calcul dans des conditions similaires.

5.3.3 Conclusion

Notre noyau temps-réel ne souffre pas, comme les autres noyaux pour les systèmes embarqués, d'un problème de déterminisme. Son comportement est prédictible tout comme la version matérielle qui a été présentée précédemment. Par ailleurs, l'intégration de l'algorithme ELLF offre une grande robustesse pour l'ordonnancement de tâches sur de multiples ressources de calcul.

L'utilisation d'un algorithme par écoulement du temps dans un noyau temps-réel logiciel ne peut pas conduire à une solution de contrôle performante. La réévaluation continue des priorités induit de fortes contraintes qui sont difficilement surmontées par un processeur de calcul comme l'ARM 920T. Ainsi, les performances obtenues sont très inférieures à celles de l'OSoC matériel. Les pénalités sont beaucoup trop importantes pour que ce noyau puisse être utilisé dans les systèmes embarqués. Bien sûr, il est toujours possible d'augmenter la granularité des tâches. Mais il est plus difficile de disposer d'un fort parallélisme de tâches si l'on doit considérer des durées d'exécution supérieures à 30 ms.

Si maintenant un processeur de calcul plus performant était utilisé, que se passerait-il ? Déjà, on peut supposer que la surface et la puissance consommée seraient supérieures. En revanche, les temps d'exécution pourraient être considérablement réduits. Pour un OSoC constitué de 32 tâches, il faudrait un processeur cadencé à 15 GHz pour atteindre un taux d'occupation des ressources de calcul supérieur à 80 % pour l'encodeur MPEG4-AVC. Aujourd'hui, aucun processeur de calcul ne peut fonctionner à une fréquence aussi importante. Peut-être qu'en tirant parti du parallélisme d'instructions il serait possible de réduire la fréquence de fonc-

tionnement. Mais il ne faut pas oublier que l'OSoC logiciel réalisé n'intègre pas l'ensemble des services offerts par la version matérielle.

5.4 Comparaisons avec MicroCOS-II et l'état de l'art

Les sections précédentes ont permis de présenter nos résultats et d'évaluer les performances de l'architecture OSoC à l'aide de quelques applications. La comparaison avec une version équivalente logicielle a montré que l'approche matérielle était incontournable. Nous allons maintenant positionner ces travaux de thèse par rapport à l'état de l'art et tenter de comparer nos résultats. Comme nous le verrons par la suite, ceci constitue un exercice très difficile et contestable puisque notre architecture OSoC n'a aucun équivalent dans la littérature.

Tout d'abord, nous allons malgré tout profiter de cette étude pour présenter les défauts d'un noyau temps-réel appelé μCOS-II. Celui-ci est couramment utilisé et offre une taille réduite, de très bonnes performances et la possibilité de modifier facilement l'ensemble de ses services.

5.4.1 Comparaison avec MicroCOS-II

Le noyau μCOS-II est basé sur un ordonnancement dynamique sur 64 niveaux de priorité et cadencé par les évènements. La priorité d'une tâche peut être modifiée au cours de l'exécution par une tâche concurrente plus prioritaire. D'autre part, une interruption périodique est utilisée pour garder une trace du temps. Elle permet par exemple le réveil des tâches périodiques et l'incrémentation du compteur de temps utilisé par différents services du noyau.

la figure 5.22 met en évidence le comportement non-déterministe de μCOS-II. Certains services ont des durées d'exécution qui dépendent du nombre de tâches à exécuter. L'augmentation peut même être particulièrement élevée. Par exemple, la durée de prise en compte des interruptions peut atteindre 2477 cycles au lieu de 76 cycles, soit une augmentation supérieure à 3000 %. Le nombre de tâches actives ne peut malheureusement pas être connu à l'avance puisqu'il dépend des besoins de l'utilisateur et des applications qui s'exécutent simultanément.

Comme nous l'avons vu dans le second chapitre, l'utilisation d'un système d'exploitation engendre des problèmes de disponibilité et de déterminisme. Alors même que l'on pourrait s'attendre à une réactivité importante, celle-ci est limitée par le nombre de tâches actives et suspendues. Comme le montre la figure 5.22-c, avec 32 tâches actives, il faut attendre 2477 cycles pour prendre en compte l'interruption, puis 237 cycles supplémentaires pour la quitter. Au total, il faut donc attendre plus de 15 μs avec un ARM 920T entre deux prises en compte d'interruption.

La figure 5.23 montre les performances obtenues avec l'encodeur MPEG4-AVC. Celui-ci est exécuté sur la méme ressource qui supporte l'exécution du noyau temps-réel. Alors même que la pénalité du contrôle reste très faible lorsque μCOS-II est peu activé, il peut empêcher l'exécution des tâches lorsque la résolution du temps devient trop faible. Le processeur ne peut plus libérer de temps pour l'exécution des tâches. La création dynamique des tâches permet de limiter la pénalité du contrôle, mais il suffirait qu'une autre application vienne s'ajouter à l'application en cours pour se retrouver dans des conditions similaires.

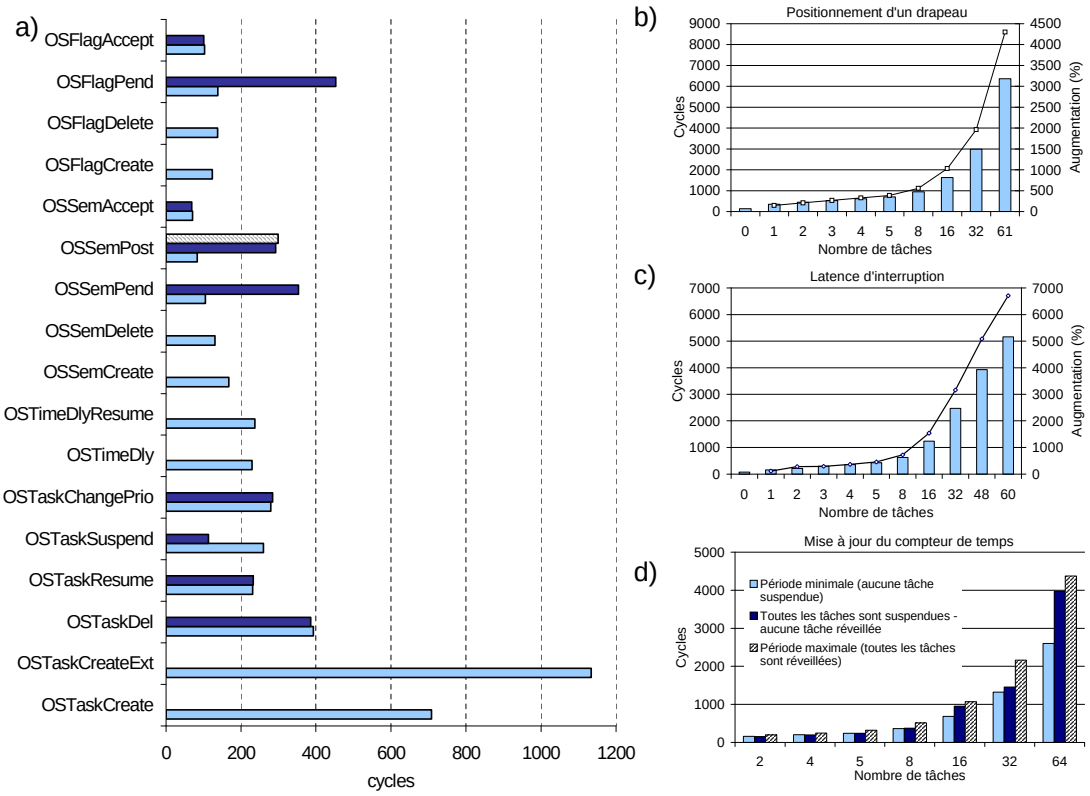

FIG. 5.22 – Evaluation des durées des services offerts par μCOS-II en fonction du nombre de tâches. Ces figures présentent les durées des principaux services offerts par μCOS-II (a), le temps nécessaire au positionnement d'un drapeau (b), la latence d'interruption (c) et la période minimale à respecter pour la mise à jour du compteur de temps (d). Les figures $\{b, c, d\}$ montrent l'évolution des comportements de ces différents services en fonction du nombre de tâches actives.

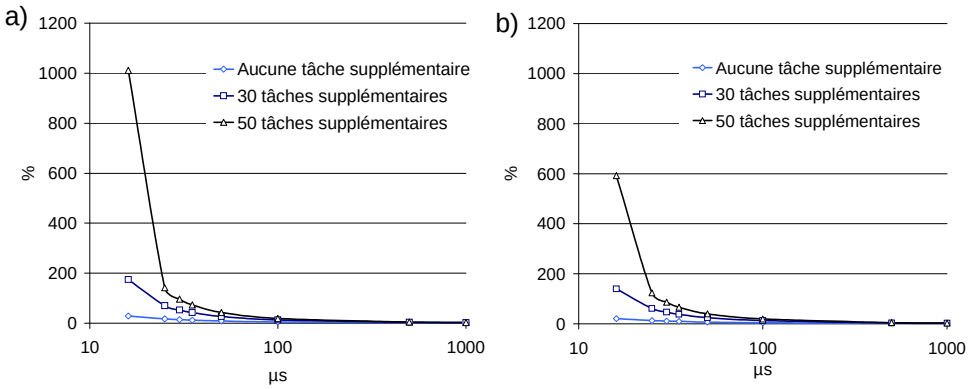

FIG. 5.23 – Surcoût dû au contrôle lors de l'exécution de l'encodeur MPEG4-AVC sur l'ARM 920T. L'ARM 920T est en charge de l'exécution de μCOS-II et des tâches. Les tâches supplémentaires ne s'exécutent jamais. Elles sont juste dans la liste des tâches à exécuter. Les tâches sont créées dans le noyau avant l'exécution de l'application (a), ou créées dynamiquement au cours de l'exécution (b). La solution dynamique est moins pénalisante puisqu'elle manipule moins de tâches simultanément.

En conclusion, le noyau temps-réel μCOS-II est plus réactif que l'OSoC. Néanmoins, il ne permet pas le contrôle en ligne de multiples ressources de calcul. Par ailleurs, l'OSoC autorise l'exécution de 65536 tâches (32 tâches actives simultanément) alors que μCOS-II est limité à 255 tâches. Pour exploiter efficacement le parallélisme de tâches, il est utile de disposer d'un nombre important de tâches pour décrire davantage d'applications. Par exemple, l'encodeur MPEG4-AVC doit être composé de 304 tâches pour une exécution optimale sur 32 ressources de calcul. De plus, l'utilisation de niveaux de priorité ne traduit pas l'urgence d'exécution et l'ordonnancement des tâches ne peut pas s'adapter aux besoins de l'utilisateur. D'autre part, la surface nécessaire pour exécuter μCOS-II est plus importante puisque l'ARM 920T occupe près de 4,7 mm^2. Enfin, son comportement non-déterministe oblige à se placer dans les conditions les plus défavorables et à considérer des WCET très éloignés des conditions réelles. Le taux d'occupation des ressources de calcul ne peut donc pas atteindre un niveau élevé malgré la réactivité du noyau.

5.4.2 Comparaison avec les autres accélérateurs matériels

Les autres accélérateurs matériels proposés dans la littérature résolvent la plupart de ces limitations. L'accélération des services ayant des comportements non-déterministes rend la solution de contrôle globalement déterministe. Par ailleurs, l'utilisation d'une architecture matérielle permet de mettre en œuvre des algorithmes d'ordonnancement plus élaborés ou qui nécessitent une fréquence d'activation élevée.

La figure 5.24 résume les fonctionnalités offertes par la plupart des accélérateurs de noyau temps-réel, tandis que le tableau 5.13 présente la taille et la réactivité de chacune de ces solutions. La comparaison est cependant très difficile puisque les technologies d'intégration ou les cibles reconfigurables ne sont pas les mêmes.

Accélérateurs temps-réel matériels	Ordonnancement	Nombre de tâches	Dépendances / synchronisations	Tâches non périodiques	Partage des ressources de calcul	Partage des ressources de mémorisation	Gestion dynamique des tâches	Migration des tâches
ATAC [137]	statique (64 niveaux)	32	non	non	non	non	non	non
ELLF [139]	dynamique (ELLF)	32	non	non	non	non	non	non
cs2 [138]	dynamique (EDF)	16	non	non	non	non	non	non
F-Timer [140]	SIFO (64 niveaux)	32	non	**oui**	non	non	non	non
RTM [141]	statique (64 niveaux)	64	non	**oui**	non	non	non	non
CHS [142]	statique/RM/EDF	32	non	**oui**	non	non	non	non
Sierra [143]	statique (8 niveaux)	16	**oui**	**oui**	non	non	non	non
STRON [144]	FIFO/round-robin (8 niveaux)	31	**oui**	**oui**	non	non	non	non
Spring [157]	dynamique (MDF/MESF)	32	non	non	**oui** (32)	non	non	non
SystemWeaver [158]	FIFO/round-robin/...	variable	**oui**	**oui**	**oui**	non	non	non
RTU 94 [161]	statique (8 niveaux)	64	**oui**	**oui**	**oui** (3)	non	non	non
OSoC	dynamique (ELLF)	variable	**oui**	**oui**	**oui**	**oui**	**oui**	**oui**

FIG. 5.24 – Comparaison des fonctionnalités de l'OSoC avec les autres accélérateurs matériels de noyau temps-réel

Accélérateurs temps-réel matériels	Taille	Réactivité
ATAC [137]	215 mm^2 (2 μm)	330 μs
ELLF [139]	3200 CLB (Xilinx XC40250XV)	850 ns
CS2 [138]	4576 logic elements (Altera 20K200E)	10 μs
F-Timer [140]	866 logic elements (Altera EPF10k20) + 6144 bits de mémoire	100 μs
RTM [141]	NC	2,2 μs
CHS [142]	0,24 mm^2 (0,35 μm) ou 421 logic elements et 564 registres (Altera Quartus II EP20K)	20 μs
Sierra [143]	532 slices (Xilinx XC2S100E-5)	2 μs (variable)
STRON [144]	40000 portes équivalentes (0,5 μm)	20 μs (variable)
Spring [157]	59,84 mm^2 (0,8 μm)	NC
SystemWeaver [158]	100000 portes équivalentes - 0,4 mm^2 (90 nm)	1 μs (variable)
RTU 94 [161]	250000 portes équivalentes	NC
OSoC	2,3 mm^2 (0,13 μm)	16 μs

TAB. 5.13 – Comparaison de la taille et de la réactivité de l'OSoC avec les autres accélérateurs matériels de noyau temps-réel

Tout d'abord, le choix de l'algorithme d'ordonnancement ne privilégie pas toujours une évolution dynamique des priorités. En fait, seules les solutions ELLF, Spring et OSoC intègrent des algorithmes d'ordonnancement qui nécessitent une forte accélération. Le nombre de tâches est limité dans presque toutes les solutions. L'utilisation de 32 tâches est la plus souvent privilégiée. Seules les solutions SystemWeaver et OSoC permettent l'exécution d'un nombre de tâches non limité avant l'exécution. Cependant, dans les deux cas les architectures doivent être modifiées si l'on souhaite augmenter le nombre de tâches actives simultanément.

Ensuite, la gestion des dépendances et des synchronisations se résume le plus souvent à la mise en œuvre de sémaphores qui peuvent être réservés ou libérés au cours de l'exécution. Aucune solution similaire à celle utilisée dans l'OSoC n'existe dans la littérature. Nous sommes les seuls à proposer une gestion aussi transparente des dépendances de contrôle et de données. Avant son exécution, une tâche n'a pas besoin d'empêcher les tâches suivantes de s'exécuter. La propagation des jetons impose l'ordre d'activation des tâches.

De même, les tâches non-périodiques sont bien souvent simplement des tâches qui se réveillent sur une interruption extérieure. On ne leur associe pas un ordonnancement particulier et ces tâches ne suivent pas les mêmes règles d'activation que les autres tâches temps-réel. Par ailleurs, les échéances d'exécution des différentes tâches ne sont pas du tout calculées en fonction de l'échéance globale de l'application. Elles sont attribuées individuellement comme si les tâches n'avaient pas de relations entre elles. Ceci empêche les tâches suivantes de prendre en compte des retards d'activation ou des interruptions d'exécution subies par les tâches en cours. Toutes les solutions matérielles existantes sont donc très différentes de l'approche inventée pendant cette thèse.

D'autre part, les seules approches qui supportent de multiples ressources de calcul sont Spring, SystemWeaver, RTU 94 et l'OSoC. Toutes, excepté l'OSoC, utilisent un unique bus comme moyen de communication entre le contrôle et les ressources de calcul. Il ne peut donc pas y avoir de parallélisme de contrôle et la réactivité peut souffrir d'une bande passante insuffisante.

Enfin, l'OSoC est la seule solution qui prend en charge les ressources de mémorisation, la gestion de la consommation d'énergie, le contrôle dynamique des tâches et leur migration.

Ceci améliore l'occupation des ressources de calcul et permet de s'adapter en ligne aux besoins de l'utilisateur. L'OSoC offre donc de plus nombreux services et demeure la seule solution capable de contrôler des systèmes multiprocesseurs hétérogènes de manière autonome.

Cependant, comme le montre le tableau 5.13, ceci a un coût important et l'OSoC est la solution matérielle la plus complexe mise en œuvre. Nos performances sont donc a priori inférieures mais la comparaison n'a de sens que si la même application est exécutée avec chacune des solutions proposées. Ceci n'étant pas possible, nous pouvons malgré tout affirmer que, lors de l'exécution de l'encodeur MPEG4-AVC par exemple, n'importe quelle solution plus performante que l'OSoC offre un gain inférieur à 10 % puisque notre taux d'occupation des ressources de calcul est supérieur à 90 %. Par ailleurs, en un seul *tick* nous exécutons l'intégralité des services supportés par l'OSoC alors que bien souvent les temps d'exécution s'ajoutent avec les autres solutions.

5.5 Synthèse

Ce cinquième chapitre a permis de justifier l'intérêt de notre approche pour contrôler des systèmes multiprocesseurs hétérogènes. La première section a détaillé notre méthodologie de conception et de validation. Puis, elle a introduit nos résultats après synthèse logique et FPGA. Un OSoC capable d'exécuter 32 tâches en parallèle et de contrôler 16 ressources de calcul peut atteindre une résolution inférieure à 35 μs. La surface totale est de 4,2 mm^2 pour une consommation d'énergie inférieure à 155 mW.

La section suivante a analysé l'impact de notre architecture de contrôle sur les performances globales du système. Pour cela, plusieurs applications différentes ont été implantées. L'étude des algorithmes ELLF et RSSR, ainsi que de l'influence de la configuration avant l'exécution, a mis en évidence l'intérêt des différents services mis en œuvre dans l'OSoC. Par ailleurs, l'examen du comportement de l'OSoC en fonction du parallélisme de l'application et du nombre de ressources de calcul, a montré un comportement particulièrement efficace pour les architectures synthétisées. Avec l'encodeur MPEG4-AVC, le taux d'occupation des ressources de calcul est supérieur à 80 % quel que soit le parallélisme de l'application. Nous avons également pu mettre en évidence l'intérêt de séparer le contrôle et le calcul. Le parallélisme obtenu permet de limiter les pénalités induites par le contrôle et de réduire significativement les problèmes liés à l'asymétrie de l'architecture SCMP-LC.

Enfin, les deux précédentes sections ont démontré qu'il n'était pas possible d'envisager une approche logicielle pour réaliser l'OSoC et qu'aucune solution équivalente n'existe dans la littérature. Un OSoC logiciel capable d'exécuter 32 tâches en parallèle et de contrôler 16 ressources de calcul a une résolution égale à 6,26 ms. La surface totale est de 6,64 mm^2 pour une consommation d'énergie de 102 mW. Les performances sont donc très inférieures puisqu'il y a un facteur égal à 180 entre les deux approches. De plus, l'approche logicielle occupe une surface silicium nettement plus importante.

Maintenant que nous avons validé notre approche et justifié son intérêt, nous pouvons étudier l'influence de son intégration dans une architecture multiprocesseur en reprenant les critères introduits dans le premier chapitre. Le tableau 5.5 compare, pour un nombre de transistors équivalent, l'architecture superscalaire Intel Pentium 4 Mobile cadencée à 2,2 GHz et deux plates-formes multiprocesseurs SCMP-LC.

Caractéristiques	Intel Northwood Pentium 4 Mobile	SCMP-LC avec 10 Philips Trimedia TM-1100 (32 mémoires de 8Ko)	SCMP-LC avec 32 MIPS 24KE (32 mémoires de 8Ko)
Fréquence processeur	2,2 GHz	133 MHz	625 MHz
Nombre de transistors	55 millions	$\approx$ 58 millions	$\approx$ 50 millions
Puissance consommée	35 W	$\approx$ 60 W	$\approx$ 11,6 W
Mémoire embarquée	8 Ko(L1)+512 Ko(L2)	10 Ko(L1)+256 Ko	32 Ko(L1)+256 Ko
Surface totale	146 mm^2	$\approx$ 175 mm^2	$\approx$ 120 mm^2
Surface pour le calcul	13 %	$\approx$ 90 %	$\approx$ 85 %
Technologie	0,13 μm	0,13 μm	0,13 μm
Performance crête	8,8 GOPs	53 GOPs	40 GOPs
Efficacité transistor	0,16 GOPs.MT^{-1}	0,91 GOPs.MT^{-1}	0,8 GOPs.MT^{-1}
Efficacité énergétique	0,25 MOPs.mW^{-1}	0,88 MOPs.mW^{-1}	3,45 MOPs.mW^{-1}

TAB. 5.14 – Caractéristiques de plates-formes SCMP-LC et comparaison avec le processeur Intel Pentium 4. Les grandeurs indiquées pour les architectures SCMP-LC sont approximatives et ne prennent pas en compte la surface des ressources d'interconnexion par exemple. Tout de même, les résultats présentés laissent envisager une efficacité transistor répondant aux contraintes des systèmes embarqués. L'efficacité énergétique reste un peu en retrait, mais ceci est dû aux ressources de calcul utilisées qui ne sont pas orientées basse-consommation. De plus, le procédé de réduction de la consommation d'énergie mis en œuvre dans l'OSoC n'est pas considéré. Un OSoC de 32 tâches a été choisi pour ces calculs (4,2 mm^2 - soit environ 2,4 millions de transistors).

La première architecture SCMP-LC privilégie la performance et atteint une performance crête de 53 GOPs, soit un gain de 500 %. L'efficacité transistor obtenue est excellente même si l'efficacité énergétique est encore un peu juste. La seconde architecture SCMP-LC utilise des ressources de calcul moins performantes mais plus efficaces. L'efficacité transistor est réduite mais reste très élevée. Au contraire, l'efficacité énergétique dépasse toutes les valeurs présentées dans le premier chapitre. L'utilisation ou la conception de ressources de calcul plus efficaces permettrait encore de l'augmenter et de réduire la consommation d'énergie de notre architecture.

Nous avons donc réussi à concevoir une architecture multiprocesseur hétérogène pour des systèmes embarqués pouvant répondre à de fortes contraintes en terme de performance et d'efficacité transistor ou énergétique. Ceci est rendu possible par l'utilisation d'une architecture capable de supporter le contrôle en ligne de multiples ressources de calculs et de mémorisations. Contrairement aux autres solutions de contrôle, l'architecture OSoC permet une utilisation efficace de ses ressources sous des contraintes de temps-réel et de consommation d'énergie. Elle repousse encore les limites des architectures et répond aux besoins des futures applications mobiles.

Conclusions et perspectives

Sommaire

Pour répondre aux besoins des futures applications mobiles, nous avons proposé dans cette thèse une architecture de contrôle capable de gérer en ligne de multiples ressources de calcul hétérogènes. Cette architecture, appelée OSoC, permet la gestion dynamique de multiples applications concurrentes, des dépendances de contrôle et de données, ainsi que l'ordonnancement de tâches temps-réel, non-temps-réel, périodiques et non-périodiques. De plus, elle maîtrise la préemption et la migration des tâches, et améliore le taux d'occupation des ressources de calcul et la consommation d'énergie.

Synthèse des contributions

La première partie de la thèse a validé les fonctionnalités dynamiques du RAC, dont les concepts architecturaux avaient été proposés par F. Blanc [288]. Cette architecture était initialement pensée pour contrôler plusieurs ressources de calcul reconfigurables dont le nombre dépendait du parallélisme d'instructions de l'application. La limitation du parallélisme d'instructions et les performances obtenues lors de la conception du RAC pendant la thèse de S. Chevobbe [210], nous ont motivés à exploiter le parallélisme de tâches et à augmenter la granularité des processus attachés à chacun des états des graphes d'application. Avec ces nouvelles considérations, le RAC est alors devenu une solution bien adaptée pour la gestion des dépendances de données et de contrôle, des synchronisations, des processus non-déterministes et des accès aux ressources partagées.

Par ailleurs, d'importants problèmes persistaient pour exploiter efficacement de multiples ressources de calcul hétérogènes, tout en conservant la cohérence et l'intégrité des données. De nombreux industriels et instituts de recherche mettaient en évidence des difficultés majeures liées aux synchronisations, à la programmation, à l'exploitation du parallélisme de tâches, ou par exemple aux réseaux d'interconnexion. Celles-ci les empêchaient alors de tirer profit des forts enjeux économiques associés à ces nouvelles architectures.

Pourtant, notre approche, consistant à séparer explicitement le contrôle du calcul et à disposer d'une ressource matérielle adaptée au contrôle, semblait intéressante pour répondre en partie à ces inconvénients. En effet, les solutions de contrôle logicielles utilisées possèdent une réactivité insuffisante et engendrent des systèmes dotés d'une faible efficacité énergétique et transistor. Ces travaux de thèse se sont donc attachés à transformer le RAC en une solution matérielle capable de contrôler efficacement de multiples ressources de calcul hétérogènes sous des contraintes de temps-réel et de consommation d'énergie.

Ainsi, ces travaux de thèse ont tout d'abord proposé une évolution du RAC pour maîtriser son comportement et ses temps d'exécution. Puis, ils ont défini une architecture asymétrique multiprocesseur, dénommée SCMP-LC, offrant des mécanismes de préemption et de migration efficaces, grâce à l'utilisation de mémoires physiquement distribuées et logiquement partagées. En effet, il nous fallait un modèle d'architecture multiprocesseur sans contrôle distribué et suffisamment souple pour permettre d'atteindre un taux d'occupation élevé des ressources de calcul. Enfin, nous avons élaboré et validé une architecture de contrôle, appelée OSoC, intégrant la plupart des services habituellement offerts par les micro-noyaux temps-réel.

Pour la première fois, l'architecture OSoC proposée permet la gestion matérielle en ligne des tâches et des applications, des dépendances de données et de contrôle, ainsi que le respect des précédences d'exécution entre les tâches. De plus, cette architecture met en œuvre une représentation nouvelle des échéances d'exécution permettant une gestion transparente des tâches périodiques et non-périodiques. L'innovation réside également dans l'utilisation et l'association de multiples algorithmes d'ordonnancement en ligne multiprocesseur temps-réel et non-temps-réel. Le changement de priorité suivant la laxité dans un environnement multiprocesseur constitue par ailleurs une approche originale, tout comme la propagation de la laxité entre les tâches de l'application. Enfin, ces travaux de thèse introduisent un modèle de programmation et d'exécution innovant de part la définition particulière des tâches utilisées. Les nombreuses fonctionnalités intégrées dans l'OSoC autorisent l'exécution de tâches indépendantes et simplifient les synchronisations et le partage des données.

Avec l'OSoC, l'architecture SCMP-LC atteint une bonne efficacité transistor et énergétique. Elle offre un taux d'occupation élevé grâce à ses mécanismes de préemption et de migration et à une forte réactivité du contrôle. Nous avons donc proposé une architecture multiprocesseur hétérogène capable de respecter de fortes contraintes en terme de surface, de performance, de consommation d'énergie et de temps-réel. Son fonctionnement en ligne et l'utilisation efficace de ses ressources de calcul permettent de répondre aux besoins des futures applications mobiles.

Ces travaux de thèse tiennent aujourd'hui une place importante dans les activités multi-cœurs du CEA LIST qui est un acteur majeur en matière de recherche, de développement, et d'innovation dans les systèmes à dominante logicielle. Ils sont valorisés dans le cadre de projets de recherche nationaux et européens (TeraOps, LoMoSA...) en collaboration avec les principaux partenaires industriels spécialisés dans la conception des systèmes embarqués. Notre approche innovante de la gestion du parallélisme offre de nouvelles perspectives et repousse encore les limites des architectures de calcul actuelles.

Néanmoins, des développements supplémentaires restent à effectuer avant d'espérer un transfert technologique vers les industriels. Dans la section suivante, nous présentons les limitations de notre approche ainsi que les perspectives de recherche et de développement.

Perspectives

A court terme

Même si ces travaux de thèse ont conduit à de nombreuses validations, celles-ci restent insuffisantes pour évaluer précisément les performances globales de notre architecture. Tout d'abord, il serait intéressant de comparer, à l'aide de quelques applications, l'architecture SCMP-LC avec d'autres architectures parallèles et de discuter davantage la conclusion énoncée dans le premier chapitre. Ensuite, il faudrait considérer l'encodeur MPEG4-AVC dans sa totalité et intégrer les synchronisations nécessaires à la parallélisation des tâches. Une preuve mathématique de l'algorithme d'ordonnancement temps-réel multiprocesseur permettrait également de s'assurer de son comportement et de vérifier nos présomptions. D'autre part, l'estimation de la consommation d'énergie de l'OSoC et du gain apporté par la gestion des modes de repos et DVFS n'est pas suffisamment précise pour conclure définitivement sur l'efficacité énergétique du système complet. Enfin, le réseau d'interconnexion entre les ressources de calcul et les mémoires partagées doit être davantage étudié. L'architecture SCMP-LC repose en grande partie sur le bon dimensionnement de ce réseau et il est donc nécessaire d'élaborer et de valider une solution adaptée pour concrétiser ces travaux de thèse. Pour cela, une étude est actuellement en cours et devrait se poursuivre dans le cadre d'une nouvelle thèse.

D'autres développements seraient à réaliser pour améliorer encore nos résultats. Tout d'abord, le comportement du RAC pourrait être synchronisé et simplifié afin de l'adapter à nos nouveaux besoins. Ensuite, le fonctionnement de l'OSoC repose sur une utilisation raisonnée de ses ressources. Les outils doivent être capables de garantir un comportement normal quelles que soient les conditions d'exécution. De plus, le nombre d'applications exécutées dans l'OSoC doit respecter le nombre maximal de tâches actives simultanément. Une solution plus robuste serait donc à mettre en œuvre pour éviter ces inconvénients. Enfin, le gain dû aux pré-configurations dépend actuellement de la géométrie des graphes d'application. Il serait intéressant d'anticiper davantage les configurations et d'étudier leur influence sur les préemptions et les performances du système. Par ailleurs, la structure de l'OSoC pourrait autoriser plus de parallélisme même si le respect de la cohérence d'exécution limite les améliorations possibles.

L'industrialisation de l'OSoC nécessite ensuite la disponibilité d'outils efficaces et faciles d'utilisation. Notre modèle de programmation n'est aujourd'hui pas suffisamment évolué pour réutiliser des applications existantes. Des travaux de recherche et de développement doivent être réalisés pour extraire le parallélisme des applications sans instrumentation excessive du code original. La réalisation d'un environnement de simulation est actuellement en cours de réalisation par K. Benchehida dans le cadre d'un contrat post-doctoral au sein du CEA LIST. Enfin, pour une meilleure utilisation de l'hétérogénéité des ressources de calcul, il pourrait être intéressant de disposer, pour une même tâche, de codes pré-compilés ciblant des ressources différentes. Le partitionnement logiciel/matériel serait alors effectué en ligne en fonction de la charge des ressources de calcul et des besoins de l'utilisateur. Pour éviter une utilisation excessive de la mémoire, il faudrait télécharger, au moment de la configuration, le code de la tâche à partir d'une mémoire de masse extérieure.

D'autres perspectives de recherche pourraient étendre les fonctionnalités de l'OSoC et introduire de nouveaux paradigmes d'architecture. Tout d'abord, l'élément de contrôle de l'OSoC pourrait faire l'objet de développements plus approfondis pour que l'architecture SCMP-LC

ait un comportement plus autonome par rapport au système d'exploitation. Par exemple, il pourrait s'occuper lui-même du chargement des applications et ainsi exploiter le parallélisme d'applications plus efficacement. Ensuite, l'intégration de mécanismes tolérants aux fautes permettrait de continuer l'exécution malgré la défaillance d'un processeur. La mise en œuvre de la migration et de la préemption des tâches constitue d'ailleurs une première étape vers de telles architectures. D'autre part, le niveau de couplage entre le système d'exploitation et l'OSoC pourrait être plus étroit pour faciliter la gestion des erreurs et réduire la granularité des tâches exécutées. Des travaux en cours ont défini une nouvelle architecture appelée *Tightly-Coupled SCalable Multi-Processor* (SCMP-TC) pour étudier ses avantages [191]. Enfin, la reconfigurabilité de l'OSoC pourrait être exploitée pour obtenir un contrôleur capable d'ajuster son comportement en fonction des ressources de calcul dont il dispose. En effet, on pourrait imaginer que le nombre et le type des ressources puissent varier en ligne. Alors, la reconfiguration des mécanismes d'allocation, par exemple, permettrait de s'adapter sans intervention extérieure aux changements de la structure de l'architecture. Ceci lui apporterait une meilleure tolérance aux fautes et une auto-adaptabilité importante nécessaire dans un environnement versatile.

A long terme

On peut maintenant s'intéresser à la pérennité des propositions faites dans ces travaux de thèse. Tout d'abord, le parallélisme de tâches atteindra certainement ses limites et réduira le nombre de processeurs utilisables dans notre architecture. Par ailleurs, les architectures devront certainement posséder une forte puissance de calcul mais n'auront sans doute pas de très nombreuses applications à exécuter en parallèle. Les dispositifs mobiles seront peut-être amenés à communiquer simultanément, suivant des normes de communication différentes, avec de multiples dispositifs électroniques ambiants. Mais, ils devront surtout exécuter, successivement, différentes applications critiques en fonction de leur environnement dans le cadre de la réalité augmentée par exemple.

Le nombre de processeurs à contrôler sera donc limité par le parallélisme des applications, mais avant tout par la consommation d'énergie, les moyens de communication utilisés et le nombre de mémoires. Pourtant, la quantité de transistors disponibles ne cessera pas d'augmenter et il faudra trouver une solution efficace pour les exploiter tout en maîtrisant la consommation d'énergie. La centralisation du contrôle n'est sans doute pas une solution à très long terme, mais une hiérarchisation des architectures OSoC permettrait de conserver localement ce modèle et de distribuer davantage le contrôle et la charge de calcul.

Enfin, les dispositifs électroniques seront de plus en plus pervasifs. Ils devront alors communiquer entre eux et s'auto-adapter en fonction des autres dispositifs distants et présents dans leur environnement. En cela, l'OSoC est une première étape vers la conception d'une architecture capable d'adapter dynamiquement sa structure et son comportement en fonction des données reçues, des aléas d'exécution ou de son environnement. L'effort maintenu tout au long de cette thèse pour disposer d'un contrôle en ligne prendrait alors tout son sens.

Glossaire

API ..... *Application Programming Interface*
ASIC ..... *Application Specific Integrated Circuit*
CAO ..... *Conception Assistée par Ordinateur*
CAVLC ..... *Context-Adaptive Variable Length Coding*
CDFG ..... *Control Data Flow Graph*
CI ..... *Control Interface*
CMP ..... *Chip Multiprocessor ou Chip Multiprocessing*
CMT ..... *Chip Multitâche ou Chip Multithreading*
CMU ..... *CPU Managment Unit*
CPOP ..... *Critical-Path-On-a-Processor*
CU ..... *Control Unit*
DMA ..... *Direct Memory Access*
DRO ..... *Dynamic Resource Occupation*
DVFS ..... *Dynamic Voltage Frequency Scaling*
DRO ..... *Dynamic Resource Occupation*
DSP ..... *Digital Signal Processing*
EDF ..... *Earliest Deadline First*
ELLF ..... *Enhanced Least Laxity First*
EPIC ..... *Explicitly Parallel Instruction Computing*
ET ..... *Event-Triggered*
FCFS ..... *First Come First Serve*
FFDH ..... *First-Fit-Decreasing-Height*
FFDH ..... *First-Fit-Increasing-Height*
FPGA ..... *Field Programmable Gate Array*
GA ..... *Générateur d'Adresse*
GPS ..... *Global Positioning System*
HAL ..... *Hardware Abstraction Layer*
HEFT ..... *Heterogeneous Earliest-Finish-Time*
HT ..... *Hyperthreading*
IMDCT ..... *Inverse Modified Discrete Cosine Transform*
IP ..... *Intellectual Property*
ILP ..... *Instruction-Level Parallelism*
IPC ..... *Inter-Process Communication*
LATF ..... *LArgest Task First*
LLD . *Level-by-level and Largest-task-first scheduling with Dynamic-resource-occupation*
LLF ..... *Least Laxity First*
LTF ..... *Longest Task First*

LUT ..... *Look-Up Table*
MB ..... *MacroBloc*
MIMD ..... *Multiple Instruction Multiple Data*
MISD ..... *Multiple Instruction Single Data*
MLCA ..... *Multilevel Computing Architecture*
MPEG ..... *Moving Picture Experts Group*
MPSoC ..... *Multi-Processor System on Chip*
NFDH ..... *Next-Fit-Decreasing-Height*
OSoC ..... *Operating System accelerator on Chip*
PAL ..... *Programmable Array Logic*
PDA ..... *Personal Digital Assistant*
PLL ..... *Phase-Locked Loop*
PMAU ..... *PE and Memory Allocation Unit*
PRAM ..... *Parallel Random Access Memory*
RAC ..... *Reconfigurable Adapted for the Control*
RISC ..... *Reduced Instruction Set Computer*
RSSR ..... *Restricted Sharing Slack Reclamation*
RTL ..... *Register Transfer Level*
RTPOS ..... *Real Time Processor Operating System*
RTU ..... *Real Time Unit*
SCMP-LC ..... *Loosely-Coupled SCalable Multi-Processors*
SCMP-TC ..... *Tightly-Coupled SCalable Multi-Processor*
SE ..... *Système d'exploitation*
SIMD ..... *Single Instruction Multiple Data*
SISD ..... *Single Instruction Single Data*
SMT ..... *Simultaneous Multithreading*
SMP ..... *Symetric Multiprocessor*
SCU ..... *Scheduling Unit*
SU ..... *Selection Unit*
SoCDDU ..... *System-on-Chip Deadlock Detection Unit*
SoCDMMU ..... *System-on-Chip Dynamic Memory Management Unit*
SoCLC ..... *System-on-Chip Lock Cache*
SPEAR ..... *Signal Processing Environments and ARchitectures*
STF ..... *Shortest Task First*
SWP ..... *Sub-Word Parallelism*
SynDEx ..... *Synchronized Distributed Execution*
TDMA ..... *Time Division Multiple Access*
TLP ..... *Thread-Level Parallelism*
TMM ..... *Task Main Memory*
TNS ..... *Temporal Noise Shaping*
TSMU ..... *Task Execution and Synchronization Unit*
TT ..... *Time-Triggered*
UF ..... *Unité Fonctionnelle*
UGM ..... *Unité de Gestion de la Mémoire*
UMA ..... *Uniform Access Memory*

UMTS ..... *Universal Mobile Telecommunications System*
VLIW ..... *Very Large Instruction Word*
VSP ..... *Variable Speed Processor*
WCEP ..... *Worst Case Execution Path*
WCET ..... *Worst Case Execution Time*
WiFi ..... *Wireless Fidelity*
WiMAX ..... *Worldwide Interoperability for Microwave Access*

Bibliographie

[1] Aad J. van der Steen. Overview of recent supercomputers. Technical report, NCF/Utrecht University, Ultrecht, The Netherlands, March 2005.

[2] A.J. Bernstein. Program Analysis for Parallel Processing. *IEEE Transactions on Electronic Computer*, 15(5) :757–762, 1966.

[3] D.W. Wall. Limits of instruction-level parallelism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, USA, April 1991.

[4] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, May 1991.

[5] John L. Hennesy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Number 1-55860-596-2. Morgan Kaufmann Publishers, third edition, 2003.

[6] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1987.

[7] J. Gustafson. Re-evaluating Amdahl's law. *Communications of the ACM*, 31(5) :532–533, May 1988.

[8] MIPS32 24KE Family of Synthesizable Processor Cores, http ://www.mips.com.

[9] R. David, S. Pillement, and O. Sentieys. *Low-Power Electronics design*, chapter 20 : Low Power Reconfigurable Processors. Number 0-8493-1941-2. CRC Press, 2004.

[10] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21 :948–960, September 1972.

[11] M.J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54(12) :1901–1909, December 1966.

[12] Intel. Embedded Intel486^{TM} Processor Hardware Reference Manual. Technical Report 273025-001, http ://www.intel.com/design/intarch/intel486/, July 1997.

[13] David Cormie. The ARM11 Microarchitecture. Technical report, ARM Ltd, 2002.

[14] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Number 0135904722. Prentice Hall PTR, 2nd edition edition, 1991.

[15] Texas Instrument Semiconductors. TMS320C5x User's Guide. Technical report, http ://www.ti.com, SPRU056D, 1998.

[16] C. Stephens, B. Cogswell, J. Heinlein, and G. Palmer. Instruction level profiling and evaluation of the IBM RS/6000. In *IEEE International Symposium on Computer Architecture (ISCA)*, Seattle, USA, May 1990.

[17] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13 :11–21, 1993.

[18] J.H. Edmondson and P.I. Rubinfeld. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 1995.

[19] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2) :28–40, April 1996.

[20] T. Horel and G. Lauterbach. UltraSPARC-III : Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3) :73–85, May 1999.

[21] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processor. In *IEEE International Symposium on Computer Architecture (ISCA)*, volume 23, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[22] E. S. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-30)*, pages 138–148, Research Triangle Park, USA, December 1997.

[23] Xavier Verians, Jean-Didier Legat, Jean-Jacques Quisquater, and Benoit M. Macq. A New Parallelism Management Scheme for Multiprocessor Systems. In *Austrian Center for Parallel Computation (ACPC)*, pages 246–256, Salzburg, Austria, February 1999.

[24] James S. Evans and Gregory L. Trimper. *Itanium Architecture for Programmers : Understanding 64-Bit Processors and EPIC Principles* . Number 0131013726. Prentice Hall PTR, 2003.

[25] Texas Instrument Semiconductors. TMS320C6x User's Guide. Technical report, http ://www.ti.com, 1999.

[26] F. Homewood and P. Faraboschi. ST200 : A VLIW Architecture for Media-Oriented Applications. In *Microprocessor Forum*, 2000.

[27] R. Sucher. Carmel : A configurable long instruction word DSP core. In *Microprocessor Forum*, October 1998.

[28] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Visser, E.J.D. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. TriMedia CPU64 Architecture. In *IEEE International Conference on Computer Design (ICCD)*, pages 586–592, Austin, Texas, October 1999.

[29] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading : Maximizing On-Chip Parallelism. In *IEEE International Symposium on Computer Architecture (ISCA)*, Santa Margherita Ligure, Italy, June 1995.

[30] Daniel William Towner. *The Uniform Heterogeneous Multi-threaded Processor Architecture*. PhD thesis, University of Bristol, June 2002.

[31] T. Ungerer, B. Robic, and J. Silc. Mutithreaded Processors. *The Computer Journal*, 45(3) :320–348, 2002.

[32] R.A. Iannuci, G.R. Gao, R. Halstead, and B. Smith. *Multithreaded Computer Architecture : A Summary of the State of the Art*. Number 0792394771. Kluwer Academic Publishers, 1994.

[33] Burton J. Smith. Architecture and application of the hep multiprocessor computer system. In *SPIE Real-Time Signal Processing IV*, volume 298, pages 241–248, 1981.

[34] R. Halstead and T. Fujita. MASA : a multithreaded processor architecture for parallel symbolic computing. In IEEE Computer Society Press, editor, *IEEE International Symposium on Computer Architecture (ISCA)*, Honolulu, USA, May 1988.

[35] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine : Architecture and Performance. In *IEEE International Symposium on Computer Architecture (ISCA)*, Santa Margherita Ligure, Italy, June 1995.

[36] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In International Conference on Supercomputing. In *ACM International Conference on Supercomputing (ICS)*, pages 1–6, Amsterdam, The Netherlands, June 1990.

[37] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle : An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13 :48–61, 1993.

[38] A. Mikschl and W. Damm. MSparc : A multithreaded sparc. In *IEEE/ACM/IFIP Euro-Par Conference*, volume 1123 of *Lecture Notes in Computer Science*, Lyon, France, August 1996. Springer.

[39] W. Grünwald and T. Ungerer. Towards extremely fast context switching in a block multithreaded processor. In IEEE Computer Society Press, editor, *IEEE Euromicro Conference*, Prague, Czech Republic, September 1996.

[40] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC Architecture : A Synthesis of Parallelism and Scalability. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, Austin, USA, November 2000.

[41] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, San Diego, USA, December 2003.

[42] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. Technical report, Intel Technology Journal Q1, 2001.

[43] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(2) :2, August 1997.

[44] M. N. Dorozhevets and Peter Wolcott. The el'brus-3 and MARS-M : Recent advances in russian high-performance computing. *Journal of Supercomputing*, 6 :5–48, 1992.

[45] J. Emer. Simultaneous multithreading : multiplying Alpha's performance. In *Microprocessor Forum*, San Jose, USA, October 1999.

[46] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip : a dualcore multithreaded processor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*, Portland, USA, December 2004.

[47] H.Oehring, U.Sigmund, and T.Ungerer. Performance of simultaneous multithreaded multimedia-enhanced processors for Mpeg-2 video decompression. *Journal of Systems Architecture*, 46 :1033–1046, 2000.

[48] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *ACM International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, USA, October 1996.

[49] Ahmed Amine Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Number 012385251X. Elsevier, 2005.

[50] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*, volume 30, chapter A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms, pages 1–30. Chapman and Hall/CRC, 2004.

[51] MPC8641D Dual-Core PowerPC Processor. Technical report, Freescale, 2005.

[52] Lance Hammond and Kunle Olukotun. Considerations in the Design of Hydra : A Multiprocessor-on-a-Chip Microarchitecture. Technical Report CSL-TR98 -749, Stanford University, February 1998.

[53] MIPS 34K MT, http ://www.mips.com.

[54] J.M. Tendler, S. Dodson, S. Fields, H. Lee, and B. Sinharoy. Power4 System Microarchitecture. Technical report, IBM, October 2001.

[55] UltraSPARC IV Processor Achitecture Overview, http ://www.sun.com.

[56] Luiz Andre Barroso et al. Piranha : A Scalable Architecture Based on Single-Chip Multiprocessing. In *IEEE International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, June 2000.

[57] OCTEON : Multi-core MIPS64 Processors, http :///www.cavium.com.

[58] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *IEEE International Symposium on Computer Architecture (ISCA)*, Jerusalem, Israel, June 1989.

[59] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony In Parallel Computing : From Dataflow To Multithreading. *Parallel and Distributed Computing Practices*, 1(1) :57–83, March 1998.

[60] J-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Pétrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits. COSY communication IP's. In *IEEE/ACM Design Automation Conference (DAC)*, Los Angeles, USA, June 2000.

[61] IBM. The CELL project at IBM Research. Technical report, http ://www.research.ibm.com/cell/, 2005.

[62] D. Pham et al. The Design and Implementation of a First-Generation Cell Processor. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2005.

[63] CT3400 Data Sheet Multiprocessor DSP. Technical report, Cradle Technologies.

[64] Wayne Wolf. The Future of Multiprocessor Systems-on-Chips. In *IEEE/ACM Design Automation Conference (DAC)*, San Diego, USA, June 2004.

[65] Texas Instrument OMAP Platform, http ://www.ti.com/omap.

[66] Nomadik - Open multimedia platform for next generation mobile devices. Technical report, STMicroelectronics, 2004.

[67] S. Dutta, R. Jensen, and A. Rieckmann. Viper : A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design and Test of Computers*, 18(5) :21–31, 2001.

[68] H-J. Stolberg, M. Berekovic, S. Moch, L. Friebe, M.B. Kulaczewski, S. Flugel, A. Dehnhardt, and P. Pirsch. HiBRID-SoC : A Multi-Core SoC Architecture for Multimedia Signal Processing. *Journal of VLSI Signal Processing*, 41(1) :9–20, 2005.

[69] J.A.J. Leijten, J.L. van Meerbergen, A.H. Timmerl, and J.A.G. Jess. PROPHID : A Heterogeneous Multi-Processor Architecture for Multimedia. In *IEEE International Conference on Computer Design (ICCD)*, Austin, USA, October 1997.

[70] Faraydon Karim, Alain Mellan, Anh Nguyen, Utku Aydonat, and Tarek S. Abdelrahman. A Multilevel Computing Architecture for Embedded Multimedia Applications. *IEEE Micro*, 24(3) :56–66, 2004.

[71] W. Cesrio, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, and M. Diaz-Nava. Component-Based Design Approach for Multicore SoCs. In *IEEE/ACM Design Automation Conference (DAC)*, New Orleans, USA, June 2000.

[72] R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes. Comparing the Energy Efficiency of CMP and SMT Architectures for MultimediaWorkloads. Technical Report UIUCDCS-R-2003-2325, University of Illinois at Urbana-Champaign, March 2003.

[73] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law Through EPI Throttling. In *IEEE International Symposium on Computer Architecture (ISCA)*, Madison, USA, June 2005.

[74] L. Spracklen and S. Abraham. Chip Multithreading : Opportunities and Challenges. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, San Francisco, USA, February 2005.

[75] Kevin Krewell. Sun Weaves Multithreaded Future. *Microprocessor Report*, April 2003.

[76] XLR Processor Product Overview . Technical report, Raza Microelectronics, Inc, 2005.

[77] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3) :32–45, May 2005.

[78] C. Lavarenne and Y. Sorel. Specification performance optimization and executive generation for real-time embedded multiprocessor application with SynDEx. In *Real-Time Embedded Processing for Space Applications in CNES international symposium*, Les Stes Maries de la Mer, France, November 1992.

[79] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized Rapid Prototyping for real-time embedded heterogeneous multiprocessors. In *IEEE/ACM/IFIPS International Workshop on Hardware/Software Co-design (CODES+ISSS)*, Rome, Italy, May 1999.

[80] E. Lenormand and G. Edelin. An industrial perspective : A pragmatic high end signal processing design environment at Thales. In *International Workshop on Computer Systems, Architectures, Modeling, and Simulation (SAMOS)*, volume 3133 of *Lecture Notes in Computer Science*, Samos, Greece, July 2003. Springer.

[81] F. Engel, G. Heiser, I. Kuz, S.M. Petters, and S. Ruocco. Operating systems on SoCs : a good idea ? In *IEEE Embedded Real-Time Systems Implementation Workshop (ERTSI)*, Lisbon, Portugal, December 2004.

[82] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design And Implementation* . Number 0131429388. Third edition, 2006.

[83] Inc Palm. Palm OS. Technical report, http ://www.palmos.com/, 2001.

[84] B. Mukherjee, K. Schwan, and P. Gopinath. A Survey of Multiprocessor Operating System Kernels. Technical Report GIT-CC-92/05, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, November 1993.

[85] I. Garcia, J.J. Merelo, J.D. Bruguera, and E.L. Zapata. Parallel quadrant interlocking factorization on hypercube computers. *Parallel Computing*, 15(1–3) :87–100, 1990.

[86] Andrew S. Tanenbaum. *Distributed Operating Systems*. Number 0132199084. 1994.

[87] Maurice J. Bach. *The Design of the UNIX Operating System*. Number 0132017997. May 1986.

[88] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, and S.J. Mullender. Experiences with the Amoeba distributed operating system. *Communications of ACM*, 33 :46–63, 1990.

[89] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4) :305–370, December 1998.

[90] Andrew S. Tanenbaum. *Modern Operating Systems*. Number 0130926418. 2nd edition, November 2001.

[91] R.N. Thompson and J.A. Wilkinson. The D825 Automatic Operating and Scheduling Program. In *AFIPS Summer Joint Computer Conference (SJCC)*, volume 23, pages 41–49, 1963.

[92] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA : The kernel of a multiprocessor operating system. *Communications of ACM*, 17 :337–345, 1974.

[93] J. A. Hawley and W. B. Meyer. *MUNIX, A Multiprocessing Version of UNIX*. PhD thesis, Naval Postgraduate School, Monterey, CA, 1975.

[94] M.J. Accetta, R.V. Baron, W.J. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian, and M. Young. Mach : A new kernel foundation for UNIX development. In *USENIX Summer Conference*, Atlanta, USA, 1986.

[95] E. Piel, P. Marquet, J. Souda, and J-L. Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. In *IEEE Euromicro Conference on Real-Time Systems (RTS)*, Catania, Italy, June 2004.

[96] Dan Hildebrand. An architectural overview of QNX. In *the USENIX Workshop on Microkernels and other Kernel Architectures*, Seattle, USA, April 1992.

[97] Jane W. S. Liu. *Real-Time Systems*. Number 0130996513. April 2000.

[98] D.B. Stewart, D.E. Schmitz, and P.K. Hhosla. The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6) :1282–1295, December 1992.

[99] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems : The MARS Approach. *IEEE Micro*, pages 25–40, 1989.

[100] T. Nakajima H. Tokuda and P. Rao. Real-Time Mach : Towards a Predictable Real-Time System. In *USENIX 1st Mach Workshop*, Burlington, USA, October 1990.

[101] B. Weinberg and C. Lundholm. Embedded Linux - Ready for Real-Time. Technical report, MontaVista Software White Paper, 2001.

[102] Enea OSE Systems. Technical report, http ://www.enea.com.

[103] VxWorks Reference Manual, 5.3.1. Technical report, http ://www.windriver.com.

[104] M. Akerholm and T. Samuelsson. Design and Benchmarking of Real-Time Multiprocessor Operating System Kernels. Master's thesis, The Department of Computer Science and Engineering, Mälardalen University, Sueden, June 2002.

[105] K. Ramamritham and J.A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of the IEEE*, 82(1) :55–67, January 1994.

[106] David Decotigny. Bibliographie d'introduction à l'ordonnancement dans les systèmes informatiques temps-réel. Technical report, INSA Rennes, 2002.

[107] Traian Pop. *Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems*. PhD thesis, Linköping university, Sueden, 2003.

[108] R. Yerraballi. Real-Time Operating Systems : An Ongoing Review. In *IEEE Work-In-Progress Sessions of the Real-Time Systems Symposium (RTSS)*, Orlando, USA, November 2000.

[109] Jean L. Labrosse. *Microc/OS II : The Real Time Kernel*. Number 1578201039. April 2002.

[110] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of ACM*, 20(1) :46–61, January 1973.

[111] G.C. Buttazo. *Hard Real-Time Computing Systems*. Number 0387231374. Kluwer Academic, 1997.

[112] J. Leung and M. Merril. A Note on Preemptive Scheduling of Periodic Real-Time Tasks. *Information Processing Letters*, 11(3) :115–118, November 1980.

[113] P. Martineau. *Ordonnancement en-ligne dans les systèmes informatiques temps-réel*. Ed 82-93, Ecole Centrale de Nantes, October 1994.

[114] J. Delacroix. *Un contrôleur d'ordonnancement temps-réel pour la stabilité de earliest deadline en surcharge : le régisseur*. PhD thesis, Conservatoire National des Arts et Métiers (CNAM), 1994.

[115] B. Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Carnegie Mellon University, August 1990.

[116] M. Caccamo, G. Lipari, and G. Buttazzo. Sharing resources among periodic and aperiodic tasks with dynamic deadlines. In *IEEE Real-Time Systems Symposium (RTSS)*, Phoenix, USA, December 1999.

[117] G. Lipari and S.K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *IEEE Real-Time Technology and Application Symposium (RTAS)*, Taipei, Taiwan, May 2001.

[118] M. Silly. The EDL Server for Scheduling Periodic and Soft Aperiodic Tasks with ResourceConstraints. *Journal of Time-Critical Computing Systems*, 17 :87–111, 1999.

[119] H. Ghetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-Time Task Under Precedence Constraints. *IEEE Real Time Systems*, 2 :181–194, 1990.

[120] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Butazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(28) :16–25, June 1995.

[121] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols : An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[122] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1) :67–100, 1991.

[123] E.G. Coffman and R. Graham. Optimal Scheduling for two-processor systems. *Acta Informatica*, 1(3), 1972.

[124] E.L. Lawler and C.U. Martel. Scheduling Periodically Occuring Tasks on Multiple Processors. *Information Processing Letters*, 12 :9–12, 1981.

[125] S.K. Sahni. Preemptive Scheduling with Due Dates. *Operational Research*, 27 :925–934, 1979.

[126] Benefits of Symbian OS for Palm Developers, http ://www.nokia.com. Technical report, Nokia, 2002.

[127] SavaJe OS, Platform Overview : J2ME Personal Basis Profile and MIDP 2.0 Edition, http ://www.savaje.com. Technical report, SavaJe Technologies, Inc, 2003.

[128] Palm OS Platform, http ://www.palmos.com. Technical report, Palm Inc., 2001.

[129] John Muray Reuter. Inside Windows CE. Technical report, Microsoft Programming Series. Microsoft Press, 1998.

[130] A.E. Eichenberger. Optimizing Compiler for a CELL Processor. In *IEEE/ACM Parallel Architecture and Compiler Techniques (PACT)*, Saint-Louis, USA, September 2005.

[131] A.E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1) :59–84, 2006.

[132] L. Dagum and R. Menon. OpenMP : An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1) :46–55, 1998.

[133] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr, and M. Zosel. *The High-Performance Fortran Handbook*. Number 0-262-61094-9. 1994.

[134] T. Li and L.K. John. Run-time Modeling and Estimation of Operating System Power Consumption. In *ACM International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, San Diego, USA, June 2003.

[135] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The Performance and Energy Consumption of Embedded Real-Time Operating Systems. *IEEE Transactions on Computers*, 52(11) :1454–1469, November 2003.

[136] M. Sindhwani, T.F. Oliver, D.L. Maskell, and T. Srikanthan. RTOS Acceleration Techniques - Review and Challenges. In *Real-Time Linux Workshop*, Singapore, Malaysia, November 2004.

[137] J. Roos. Designing a Real-Time Coprocessor for Ada Tasking. *IEEE Design and Test of Computers*, 8(1) :67–79, January 1991.

[138] A. Morton and W.M. Loucks. A Hardware/Software Kernel for System on Chip Designs. In *ACM/SIG Symposium on Applied Computing (SAC)*, Nicosia, Cyprus, March 2004.

[139] J. Hildebrandt, F. Golatowski, and D. Timmermann. Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems. In *IEEE EuroMicro Workshop on Real-Time Systems (EWRTS)*, York, England, June 1999.

[140] A. Parisoto, A. Jr. Souza, M. Pontremoli, C. Pereira, and A. Suzim. F-Timer : dedicated FPGA to real-time systems design support. In *IEEE EuroMicro Workshop on Real-Time Systems (EWRTS)*, Toledo, Spain, June 1997.

[141] P. Kohout, B. Ganesh, and B. Jacob. Hardware Support for Real-time Operating Systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Newport Beach, USA, October 2003.

[142] P. Kuacharoen, M.A. Shalan, and V.J. Mooney III. A Configurable Hardware Scheduler for Real-Time Systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, June 2003.

[143] RealFast. Sierra 16 - Operting System HW Accelerator - Users Reference Manual. Technical report, RealFast, May 2004.

[144] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai. VLSI Implementation of a Real-time Operating System. In *IEEE/SIGDA Asia and South Pacific Design Automation Conference (ASPDAC)*, Chiba, Japan, January 1997.

[145] Ken Sakamura. μITRON 3.0 Specification. Technical report, TRON Association, Tokyo, 1994.

[146] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai. Performance Evaluation of STRON : A Hardware Implementation of a Real-Time OS. *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, 82(11) :2375–2382, 1999.

[147] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware Implementation of a Real-time Operating System. In *IEEE TRON Project International Symposium*, Tokyo, Japan, November 1995.

[148] S. Isaacson and D. Wilde. The Task-Resource Matrix : Control for a Distributed Reconfigurable Multi-Processor Hardware RTOS. In *Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, June 2004.

[149] J. Lee, K.K. Ryu, and V.J. Mooney III. A Framework for Automatic Generation of Configuration Files for a CustomHardware/Software RTOS. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, June 2002.

[150] B.E.S. Akgul, J. Lee, and V. J. Mooney III. A system-on-a-chip lock cache with task preemption support. In *IEEE/ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Atlanta, USA, November 2001.

[151] B.E.S. Akgul, V.J. Mooney III, H. Thane, and P. Kuacharoen. Hardware Support for Priority Inheritance. In *IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[152] J. Lee and V.J. Mooney III. A novel deadlock avoidance algorithm and its hardware implementation. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISS)*, Stockhlom, Sweden, September 2004.

[153] M. Shalan and V.J. Mooney III. Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management. In *IEEE/ACM/IFIPS International Workshop on Hardware/Software Co-Design (CODES)*, Estes Park, USA, May 2002.

[154] M. Shalan and V.J. Mooney III. A dynamic memory management unit for embedded real-time system-on-a-chip. In *IEEE/ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, San Jose, USA, November 2000.

[155] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J.A. Stankovicand G. Wallace, and C. Weems. The Spring Scheduling Co-Processor : A Scheduling Accelerator. *IEEE Transactions on VLSI Systems*, 7(1) :38–47, March 1999.

[156] K. Ramamritham, J.A. Stankovic, and P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2) :184–194, April 1990.

[157] J.A. Stankovic and K. Ramamritham. The Spring Kernel : A New Paradigm for Real-Time Operating Systems. *ACM SIGOPS Operating Systems Review*, 23(3) :54–71, July 1989.

[158] Ignios SystemWeaver. Technical report, http ://www.ignios.com.

[159] J. Furunäs. Survey of methods of implementing IPC mechanisms. Technical report, Mälardalen University, Västeras, Sweden, 2001.

[160] J. Adomat, J. Furunäs, J. Stärner, and L. Lindh. RTU94 - Real Time Unit 1994. Technical report, Mälardalen University, Västeras, Sweden, 1994.

[161] J. Adomat, J. Furunäs, L. Lindh, and J. Stärner. Real-Time Kernel in Hardware RTU : A step towards deterministic and highperformance real-time systems. In *IEEE Euromicro Workshop on Real-Time Systems (EWRTS)*, L'Aquila, Italy, June 1996.

[162] J. Lee, V.J. Mooney III, A. Daleby, K. Ingström, and T. Klevin andL. Lindh. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In *IEEE/SIGDA Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2003.

[163] T. Samuelsson, M. Akerholm, P. Nygren, J. Stärner, and L. Lindh. A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software. In *International Workshop on Advanced Real-Time Operating System Services*, Porto, Portugal, July 2003.

[164] Raimo Haukilahti. Energy Characterization of a RTOS Hardware Accelerator for SoCs. In *Swedish System-on-Chip Conference*, Falkenberg, Sweden, March 2002.

[165] Faith E. Fich. *Synthesis of Parallel Algorithms*, chapter The complexity of computation on the Parallel Random Access Machine, pages 843–899. 1993.

[166] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6) :12–24, June 1990.

[167] Ramon Lawrence. A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors. Technical report, Department of Computer Science, University of Manitoba, May 1998.

[168] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, Phoenix, USA, December 1999.

[169] F. Mueller. Timing analysis for instruction caches. *IEEE Real-Time Systems*, 18(2) :217–247, May 2000.

[170] S. Rubini and D. Lavenier. Les architectures reconfigurables. *Calculateurs Parallèles*, 9(1) :9–27, 1997.

[171] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress 74*, Stockholm, Sweden, August 1974.

[172] K. Compton and S. Hauck. Reconfigurable Computing : A Survey of Systems and Software. *ACM Computing Surveys*, 34(2) :171–200, June 2002.

[173] R. Hartenstein. A Decade of Reconfigurable Computing : a Visionary Retrospective. In *IEEE/ACM Design Automation and Test in Europe (DATE)*, Munich, Germany, March 2001.

[174] Raphaël David. *Architecture reconfigurable dynamiquement pour applications mobiles.* PhD thesis, Université de Rennes 1, July 2003.

[175] Xilinx, http ://www.xilinx.com.

[176] Altera, http ://www.altera.com.

[177] J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk. Fast Communication Mechanisms in Coarse-grained Dynamically ReconfigurableArray Architectures. In *Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE)*, Las Vegas, USA, June 2000.

[178] H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys : An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5) :465–481, May 2000.

[179] T. Miyamori and K. Olukotun. REMARC : Reconfigurable Multimedia Array Coprocessor. In *ACM/SIGDA Field Programmable Gate Array (FPGA)*, Monterey, USA, February 1998.

[180] D. Cronquist. Architecture Design of Reconfigurable Pipelined Datapaths. In *IEEE Advanced Research in VLSI (ARVLSI)*, Atlanta, USA, March 1999.

[181] V. Baumgarte, G. Ehlers, F. May, A. Nückel, and M. Vorbach. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *Journal of Supercomputing*, 26 :167–184, 2003.

[182] V. David, C. Aussaguès, S. Louise, Ph. Hilsenkopf, B. Ortolo, and C. Hessler. The OASIS Based Qualified Display System. In *American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT)*, Columbus, USA, September 2004.

[183] G. Sassatelli, L. Torres, P. Benoit, T. Gil, C. Diou, G.Cambon, and J. Gali. Highly Scalable Dynamically Reconfigurable Systollic Ring-Architecture for DSP Applications. In *IEEE/ACM International conference on Design Automation and Test in Europe (DATE)*, Paris, France, March 2002.

[184] G. Flavius. System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors. In *The Power-Aware Computing Systems 2000 Workshop at ASPLOS'00*, Cambridge, USA, November 2000.

[185] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Tasks on Variable Voltage Processors. In *IEEE Real-Time System Symposium (RTSS)*, Madrid, Spain, December 1998.

[186] B. Brock and K. Rajamani. Dynamic Power Management for Embedded Systems. In *IEEE International SOC Conference*, Portland, USA, September 2003.

[187] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low Power Microprocessor. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, Rome, Italy, July 2001.

[188] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. In *IEEE International Symposium on Low-Power Electronic Design (ISPLED)*, Rapallo, Italy, July 2000.

[189] Marc Fleischmann. LongRun Power Management. Technical report, Transmeta Corporation, January 2001.

[190] Intel XScale Technology, http ://www.intel.com/design/intelxscale.

[191] R. David, V. David, N. Ventroux, and T. Collette. Procédé et système de calcul intensif multitâche et multiflot en temps réel. Brevet en cours de dépôt, July 2005.

[192] J-M. Philippe. *Intégration des réseaux sur silicium : optimisation des performances des couches physiques et liaison.* PhD thesis, Université de Rennes 1, November 2005.

[193] Lyle Adams. Overview of the CoreFRAME Architecture. Technical report, PalmChip Corporation.

[194] AMBA On-Chip Bus Specification (rev. 2.0). Technical report, ARM Limited, 1999.

[195] L. Benini and G. De Micheli. Networks on Chips : A New SoC Paradigm. *IEEE Computer*, 35(1) :70–78, January 2002.

[196] W.J. Dally and B. Towles. Route packets, not wires : On-chip interconnection networks. In *IEEE/ACM Design Automation Conference (DAC)*, Las Vegas, USA, June 2001.

[197] J. Liang, S. Swaminathan, and R. Tessier. aSoC : A scalable, Single-Chip Communications Architecture. In *IEEE Conference on Parallel Architectures and Compilation Technique (PACT)*, Philadelphia, USA, October 2000.

[198] D. Wiklund and D. Liu. SoCBUS : Switched network on chip for hard real time embedded systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003.

[199] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2) :62–76, February 1993.

[200] A. Rädeulescu, J. Dielissen, S.G. Pestana, O.P. Hangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1) :4–17, January 2005.

[201] M. Millberg, E. Nilsson, R. Third, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *IEEE/ACM International Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004.

[202] F. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems on Chips. *IEEE Micro*, 22(5) :36–45, September 2002.

[203] P. Crowley, M.A. Franklin, H. Hadimioglu, and P. Onufryk. *Network Processor Design : Issues and Practice*, volume 1, chapter Chapter1 : Network Processor Design : An Introduction to Design Issues. 2003.

[204] P. Guerrier. *Un réseau d'interconnexion pour systèmes intégrés*. PhD thesis, Université de Paris VI, 2000.

[205] P. Vivet, F. Clermidy, and D. Lattard. FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2006.

[206] SiliconBackplane Network Reference. Technical report, SONICS, 2002.

[207] T. Collette, H. Essafi, K. Kaiser, and D. Juvin. SYMPATIX : a SIMD Computer Performing the Low and Intermediate levels of Image Processing. In *International Parallel Architectures and Languages Europe Conference (PARLE)*, volume 605 of *Lecture Notes in Computer Science*, pages 147–161, Paris, France, June 1992. Springer.

[208] S. Chevobbe, N. Ventroux, F. Blanc, and T. Collette. RAMPASS : Reconfigurable and Advanced Multi-processing Architecture for Future Silicon Systems. In *International Workshop on Computer Systems, Architectures, Modeling, and Simulation (SAMOS)*, volume 3133 of *Lecture Notes in Computer Science*, pages 20–29, Samos, Greece, July 2003. Springer.

[209] T. Murata. Petri nets : Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[210] Stéphane Chevobbe. *Unité de commande pour systèmes parallèles : contrôleur basé sur la mise en œuvre dynamique de réseaux de Pétri*. PhD thesis, CEA LIST / IRISA, October 2005.

[211] N. Ventroux, S. Chevobbe, F. Blanc, and T. Collette. An Auto-Adaptative Reconfigurable Architecture for the Control. In *Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC)*, volume 3189 of *Lecture Notes in Computer Science*, pages 72–87, Beijing, China, September 2004. Springer.

[212] M.R. Garey et D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Number 0-7167-1045-5. W.H. Freeman, 1979.

[213] Pierre-François Dutot and D. Trystram. Scheduling on hierarchical clusters using Malleable Tasks. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Crete, Greece, July 2001.

[214] K. Jansen and L. Porkolab. Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks. *Algorithmica*, 32(3) :507–520, 2002.

[215] J. Turek, J. L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, San Diego, USA, June 1992.

[216] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4) :406–471, December 1999.

[217] L. Wang, H.J. Siegel, V.P. Roychowdhury, and A.A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1) :1–15, November 1997.

[218] F. Xu. Integration, Simulation and Implementation of a hardware-based Genetic Optimizer to Adjust Smart Antenna Receiver. Master's thesis, Friedrich-Alexander-University Erlangen-Nuremberg and Franhaufer IIS Erlangen, August 2003.

[219] T.D. Braun et al. A Comparison Study of Static Mapping Heuristics for a Class of Meta-taskson Heterogeneous Computing Systems. In *IEEE Heterogeneous Computing Workshop (HCW)*, San Juan, Puerto Rico, April 1999.

[220] A. Auyeung, I. Gondra, and H.K. Dai. Multi-Heuristic List Scheduling Genetic Algorithm for Task Scheduling. In *ACM Symposium on Applied Computing (SAC)*, Melbourne, USA, March 2003.

[221] J. Liou and M. A. Palis. A Comparison of General Approaches to Multiprocessor Scheduling. In *IEEE International Parallel Processing Symposium (IPPS)*, Geneva, Switzerland, April 1997.

[222] S. Ranaweera and D. Agrawal. A Scalable Task Duplication Based Scheduling Algorithm for Heterogeneous Systems. In *International Conference on Parallel Processing (ICPP)*, Toronto, Canada, August 2000.

[223] C. Y. Lee, L. Lei, and M. Pinedo. Current trends in deterministic scheduling. *Annals of Operations Research*, 70 :1–41, 1997.

[224] J. L. Baer. A Survey of Some Theoretical Aspects of Multiprocessing. *ACM Computing Surveys*, 5(1) :31–80, March 1973.

[225] J. Blazewicz, M. Drabowski, and Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, 35(5) :389–393, May 1986.

[226] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 Algorithm for Two-Dimensional Packing. *Journal of Algorithms*, 2(4) :348–368, June 1981.

[227] K. Jansen and L. Porkolab. Improved Approximation Schemes for Scheduling Unrelated Parallel Machines. In *ACM Symposium on Theory of Computing (STOC)*, Atlanta, USA, May 1999.

[228] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for HeterogeneousComputing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :260–274, March 2002.

[229] Keqin Li and Yi Pan. Probabilistic Analysis of Scheduling Precedence Constrained Parallel Taskson Multicomputers with Contiguous Processor Allocation. *IEEE Transactions on Computer*, 49(10) :1021–1030, October 2000.

[230] T. Ibaraki and N. Katoh. *Resource allocation Problems : Algorithmic Approaches*. Number 0-262-09027-9. The MIT Press, Cambridge, Massachusetts, 1988.

[231] N. Ventroux, F. Blanc, and D. Lavenier. A Low Complex Scheduling Algorithm for Multi-Processor System-on-Chip. In *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, Innsbrück, Austria, February 2005.

[232] D. Sleator. A 2.5 Times Optimal Algorithm for Packing in Two Dimensions. *Information Processing Letters*, 10(1) :37–40, February 1980.

[233] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Computing*, 9(4) :808–826, November 1980.

[234] Joël Goossens. *Systèmes temps réel*, volume 2, chapter Ordonnancement temps réel multiprocesseur. Nicolas Navet, 2006.

[235] S. Funk, S. Baruah, and J. Goossens. Heuristics for restricting EDF migration uniform multiprocessors. In *IEEE Work-in-progress Session in Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, December 2004.

[236] J. Leung. *Handbook of scheduling : Algorithms, Models, and Performance Analysis*. Number 1584883979. Chapman Hall/CRC Press, 2004.

[237] J.Y. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4) :237–250, December 1982.

[238] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *IEEE Euromicro Conference on Real-Time Systems (ERTS)*, Stockholm, Sweden, June 2000.

[239] S. Funk, J. Goossens, and S. Baruah. On-line Scheduling On Uniform Multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[240] C.A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal Time-Critical Scheduling via Resource Augmentation. In *ACM Symposium on Theory of Computing (STC)*, El Paso, USA, May 1997.

[241] J. Goossens, S. Funk, and S. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, 25 :187–205, 2003.

[242] K.S. Hong and J.Y-T. Leung. On-Line Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, 41(10) :1326–1331, 1992.

[243] A.Srinivasan and J.H. Anderson. Optimal Rate-based scheduling on Multiprocessors. In *ACM Symposium on Theory of Computing (STOC)*, Montréal, Canada, May 2002.

[244] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *IEEE International Parallel Processing Symposium (IPPS)*, Santa Barbara, USA, April 1995.

[245] S. Baruah, J. Gehrke, and C.G. Plaxton. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15 :600–625, 1996.

[246] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *IEEE Euromicro Conference on Real-Time Systems (ERTS)*, Stockholm, Sweden, June 2000.

[247] M.L. Dertouzos and A.K-L. Mok. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, December 1989.

[248] Emmanuel Grolleau. *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Pétri en environnement monoprocesseur et multiprocesseur*. PhD thesis, Université de Poitiers, November 1999.

[249] B. Kalyanasundaram, K.R. Pruhs, and E. Torng. Errata : A New Algorithm for Scheduling Periodic, Real-Time Tasks. *Algorithmica*, 28(3) :269–270, 2000.

[250] J. Leung. A New Algorithm for Scheduling Periodic, Real-Time Tasks. *Algorithmica*, 4 :209–219, 1989.

[251] S. Sáez, J. Vila, and A. Crespo. Dynamic Scheduling Solutions for Real-Time Multiprocessor Systems. *Control Engineering Practice*, 5(7) :1007–1013, July 1997.

[252] R. Graham. *Computer and Job Shop Scheduling*, chapter Bounds on the Performance of Scheduling Algorithms. Number 0471163198. John Wiley and Sons, 1976.

[253] J. Goossens, S. Funk, and S. Baruah. EDF scheduling on multiprocessors : some (perhaps) counterintuitive observations. In *IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Tokyo, Japan, March 2002.

[254] B. Anderson and J. Jonsson. Preemptive Multiprocessor Scheduling Anomalies. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, USA, April 2002.

[255] J. Goossens, R. Devillers, and S. Funk. Tie-breaking for EDF on multiprocessor platforms. In *IEEE Work-in-progress Session in Real-Time Systems Symposium (RTSS)*, Austin, USA, December 2002.

[256] J. Goossens and P.l Richard. Overview of real-time scheduling problems. In *International Workshop on Project Management and Scheduling (CPMS)*, Nancy, France, April 2004.

[257] P. Guitton-Ouhamou, C. Belleudy, and M. Auguin. Dynamic Voltage Scaling : implementations during the scheduling step of a codesign tool. In *Sophia Antipolis Microelectronics Forum (SAME)*, Sophia-Antipolis, France, October 2003.

[258] D. Wu, B.M. Al-Hashimi, and P. Eles. Scheduling and Mapping of Conditional Task Graphs for the Synthesis of Low Power Embedded Systems. In *IEEE/ACM Design Automation and Test in Europe (DATE)*, Munich, Germany, March 2003.

[259] P.H. Chou, J. Liu, D. Li, and N. Bagherzadeh. IMPACCT : Methodology and tools for Power-Aware Embedded Systems. *Kluwer International Journal, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems*, 7(3) :233–270, October 2002.

[260] L-C. Weng, W. Wang, and B. Liu. A Survey of Dynamic Power Optimization Techniques. In *IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC)*, Alberta, Canada, July 2003.

[261] E. Macii. Dynamic power management of electronic systems. *IEEE Design and Test of Computers*, 18(2) :6–9, March 1998.

[262] D. Ramanathan and R. Gupta. System level online power management algorithms. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, Paris, France, March 2000.

[263] Y-H. Lu and G.D. Micheli. Comparing system-level power management. *IEEE Design and Test of Computers*, 18(2) :10–19, March 2001.

[264] C-H. Hwang and A.C-H. Wu. A predictive system shutdown method for energy saving of event-driven computation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, USA, November 1997.

[265] M.B. Srivastava, A.P. Chandrakasan, and R.W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1) :42–55, March 1996.

[266] E-Y. Chung, L. Benini, and G.D. Micheli. Dynamic power management using adaptive learning tree. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, USA, November 1999.

[267] G.A. Paleologo, L. Benini, A. Bogliolo, and G.D. Micheli. Policy Optimization for Dynamic Power Management. In *IEEE/ACM Design Automation Conference (DAC)*, San Francico, USA, June 1998.

[268] Q. Qiu and M. Pedram. Dynamic Power Management Based on Continuous-Time Markov Decision Processes. In *IEEE/ACM Design Automation Conference (DAC)*, New Orleans, USA, June 1999.

[269] W. Kim, D. Shin, H-S. Yun, J. Kim, and S.L. Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Jose, USA, September 2002.

[270] S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *IEEE/ACM Design Automation Conference (DAC)*, Los Angeles, USA, June 2000.

[271] D. Shin, J. Kim, and S. Lee. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design and Test of Computers*, 18(2) :20–30, March 2001.

[272] F. Gruian. Hard Real-Time Scheduling Using Stochastic Data and DVS Processors. In *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, Huntington Beach, USA, August 2001.

[273] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, USA, November 2000.

[274] D. Zhu, R. Melhem, and B. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(7) :686–700, 2003.

[275] H. Aydin, R. Melhem, D. Mosse, and P.M. Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[276] W. Kim, J. Kim, and S.L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, Paris, France, March 2002.

[277] L. Charest, E.M. Aboulhamid, and A. Tsikhanovich. Designing with SystemC : Multi-paradigm modeling and simulation performance evaluation. In *International HDL Conference*, San Jose, USA, March 2002.

[278] A. Fin, F. Fummi, M. Martignano, and M. Signoretto. SystemC : A Homogenous Environment to Test Embedded Systems. In *IEEE/ACM/IFIPS International Symposium on Hardware/software Codesign (CODES+ISSS)*, Copenhagen, Denmark, April 2001.

[279] http ://cogcomp.com. Technical report, Cogent Computer Systems, Inc.

[280] David Seal. *ARM Architecture Reference Manual*. Number 0201737191. Addison-Wesley Professional, 2nd edition edition, 2000.

[281] L. Charest, E.M. Aboulhamid, C. Pilkington, and P. Paulin. SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor. In *IEEE/ACM Design Automation and Test in Europe (DATE) Designers' Forum*, Paris, France, March 2002.

[282] D.G. Pérez, G. Mouchard, and O. Temam. A Fast SystemC Engine. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004.

[283] S. Bauer, J. Kneip, T. Mlasko, B. Schmale, J. Vollmer, A. Hutter, and M. Berekovic. The MPEG-4 multimedia coding standard : Algorithms, architectures and applications. *VLSI Signal Processing Systems*, 23(1) :7–26, October 1999.

[284] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7) :560–576, July 2003.

[285] M. Bosi, K. Brandenburg, S. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, M. Dietz, J. Herre, G. Davidson, and Yoshiaki Oikawa. ISO/IEC MPEG-2 Advanced Audio Coding. *Journal of the Audio Engineering Society*, 45(10) :789–814, October 1997.

[286] C-N. Liu and T-H. Tsai. SoC Platform Based Design of MPEG-2/4 AAC Audio Decoder. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Kobe, Japan, May 2005.

[287] Jean-Loup Leroy. Validation d'un noyau temps-réel matériel pour les systèmes multiprocesseurs hétérogènes embarqués. rapport de stage de fin d'étude d'ingénieur esiee, CEA LIST - Laboratoire Calculateurs Embarqués et Image, 2006.

[288] F. Blanc. *Etude d'un nouveau concept de calculateur reconfigurable : architecture et outils*. PhD thesis, Université de Cergy-Pontoise, CEA LIST, December 2002.

Publications personnelles

Conférences internationales

[1] **N. Ventroux**, F. Blanc, and D. Lavenier. A Low Complex Scheduling Algorithm for Multi-Processor System-on-Chip. In 23^{rd} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'05)*, Innsbrück, Austria, February 2005.

[2] **N. Ventroux**, S. Chevobbe, F. Blanc, and T. Collette. An Auto-Adaptative Reconfigurable Architecture for the Control. In 9^{th} *Asia-Pacific Computer Systems Architecture Conference (ACSAC'04)*, Springer-Verlag LNCS 3189, pages 72-87, Beijing, China, September 2004.

[3] **S. Chevobbe**, N. Ventroux, F. Blanc, and T. Collette. RAMPASS : Reconfigurable And Advanced Multi-Processing Architecture for future Silicon Systems. In 3^{rd} *International Workshop on Computer Systems, Architectures, Modeling and Simulation (SAMOS'03)*, Springer-Verlag LNCS 3133, pages 20-29, Samos, Greece, July 2003.

[4] **N. Ventroux**, J.F. Nezan, M. Raulet, and O. Déforges. Rapid prototyping for an optimized MPEG-4 decoder implementation over a parallel heterogeneous architecture. In *IEEE International Conference on Accoustics, Speech and Signal Processing (ICASSP)*, Hong-Kong, April 2003.

Brevets

[5] **R. David**, V. David, N. Ventroux, and T. Collette. Procédé et système de calcul intensif multitâche et multiflot en temps réel. *en cours de dépôt*, July 2005.

[6] **N. Ventroux**, S. Chevobbe, F. Blanc, and T. Collette. Procédé d'ordonnancement de traitement de tâches et dispositif pour mettre en œuvre le procédé. *N° 04 51743*, July 2004.

Rapports de recherche internes

[7] **N. Ventroux**. RAMPASS : état d'avancement (mise à jour). *Rapport interne*, CEA LIST, February 2004.

[8] **N. Ventroux**. RAMPASS : état d'avancement. *Rapport interne*, CEA LIST, October 2003.

Résumé

Les futurs systèmes embarqués auront besoin d'intégrer de plus en plus de services. Mais surtout, ils devront adapter dynamiquement leur structure à leur environnement et aux besoins des utilisateurs. L'augmentation des performances de ces dispositifs a longtemps été induite par l'amélioration des techniques d'extraction et d'exploitation du parallélisme d'instructions. Mais la complexité de leur mise en œuvre et la limitation de ce parallélisme, engendrent des architectures peu efficaces. Leurs performances restent insuffisantes aux regards de nos besoins applicatifs.

Une solution pour continuer à améliorer les performances consiste à exploiter le parallélisme de tâches et à multiplier les ressources de calcul intégrées sur une même puce. Néanmoins, l'étude de l'ensemble des solutions matérielles existantes montre qu'aucune d'entre-elles n'est en mesure de répondre à nos besoins particuliers. En effet, notre architecture doit respecter des exigences liées aux systèmes embarqués et donc disposer d'une bonne efficacité énergétique et transistor.

Ainsi, nous proposons dans cette thèse une architecture multiprocesseur asymétrique appelée *SCMP-LC*. Elle est caractérisée par une séparation explicite du contrôle et du calcul. Par ailleurs, elle offre des mécanismes de préemption et de migration efficaces, grâce à l'utilisation de mémoires physiquement distribuées et logiquement partagées. Notre modèle d'exécution consiste à exécuter des tâches indépendantes sur des ressources de calcul hétérogènes.

Dans cette architecture, la partie dédiée au contrôle centralise dorénavant toute la gestion de l'exécution des tâches. Par conséquent, le déterminisme et la réactivité du contrôle ont un rôle déterminant sur les performances du système. L'étude des solutions de contrôle pour les architectures multiprocesseurs montre que seule une approche matérielle est en mesure de répondre à toutes ces exigences.

C'est pourquoi nous avons élaboré une architecture dénommée *OSoC*, capable de supporter la plupart des services habituellement offerts par un système d'exploitation temps-réel. Elle permet la gestion dynamique de multiples applications concurrentes, des dépendances de données et de contrôle, ainsi que l'ordonnancement de tâches temps-réel, non-temps-réel, périodiques et non-périodiques. De plus, elle maîtrise la préemption et la migration des tâches, ainsi que la consommation d'énergie du système.

Les résultats de synthèse et de simulation confirment l'intérêt de notre solution matérielle. Le temps entre chaque nouvel ordonnancement est de 16 μs pour une surface de 2,3 mm^2 (technologie ST 130 nm). Ceci permet d'atteindre un taux d'occupation des ressources de calcul supérieur à 85 % lors de l'exécution d'un encodeur MPEG-4 AVC. La comparaison avec le noyau temps-réel μCOS-II et une approche équivalente logicielle montrent un gain significatif en terme de surface, de consommation d'énergie et de performance.

mots-clés : architecture des ordinateurs, parallélisme de tâches, multiprocesseur, contrôle, systèmes d'exploitation, systèmes en ligne, temps-réel, hétérogénéité