Les architectures parallèles sur puce

Synthèse des architectures multitâches pour les systèmes embarqués

Nicolas Ventroux — **Raphaël David**

CEA,LIST
Laboratoire Calcul Embarqué
Point Courrier 94
91191 Gif-sur-Yvette Cedex
nicolas.ventroux@cea.fr

RÉSUMÉ. *Les systèmes embarqués sont aujourd'hui omniprésents et nécessitent des puissances de calcul toujours plus importantes. Les dispositifs électroniques embarqués envahissent notre vie quotidienne et des milliers d'applications ou de fonctionnalités numériques nous accompagnent en permanence. Les architectures de calcul doivent alors être capables d'atteindre un très haut niveau de performance sous de fortes contraintes d'intégration et de consommation d'énergie. Une solution très efficace pour les systèmes embarqués consiste à multiplier le nombre de ressources de calcul sur une même puce. Ce sont les architectures multiprocesseurs. Elles permettent de traiter de multiples tâches en parallèle sur plusieurs ressources de calcul, voire de favoriser l'exécution concurrente de multiples applications. Cet article synthétise l'état de l'art des solutions existantes.*

ABSTRACT. *Embedded systems are today pervasives and need more and more important computation efficiency. Embedded electronic devices overrun our daily life and million of applications and digital functions are permanently around us. In the future, we will see the utilization boom of these services, which will need very highly performing computing architectures under integration and energy consumption constraints. A solution that can be efficiently exploited in embedded systems consists in multiplying the number of computing resources on the same die. These are multiprocessor architectures. They offer the possibility to execute multiple tasks in parallel onto several computing resources and to eventually favour concurrent execution of multiple applications. This article is a survey on existing solutions.*

MOTS-CLÉS : *architectures multiprocesseurs, MPSoC, systèmes embarqués*

KEYWORDS: *multiprocessor architectures, MPSoC, embedded systems*

RSTI - TSI. Volume 29 – n° 3/2010, pages 345 à 378

1. Introduction

Les systèmes embarqués requièrent de plus en plus de traitements intensifs et doivent pouvoir s'adapter à une évolution rapide des applications. Par ailleurs, ils doivent maîtriser leur consommation d'énergie, leur coût de développement et de réalisation et garantir un niveau de robustesse élevé. Dans le but d'apporter des solutions flexibles et facilement programmables répondant à ces contraintes, les architectes n'ont eu de cesse de proposer de nouvelles architectures parallèles.

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 aujourd'hui 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 sont nées. Elles consistent, en fait, à paralléliser le traitement des instructions ou à accroître le nombre de ressources pour exécuter plus d'instructions simultanément.

La complexité des applications est de plus en plus importante et les dispositifs électroniques embarqués doivent aujourd'hui supporter l'exécution de multiples tâches en parallèle. Comme nous le verrons par la suite, la parallélisation au niveau tâche des applications est nécessaire dans le but d'atteindre des performances encore supérieures. Mais, elle est également présente de manière naturelle au sein même des applications pour des raisons de réactivité du système, de concurrence, voire de programmabilité.

La solution la plus simple pour exécuter de multiples tâches, même indépendantes, consiste à utiliser un processeur monotâche et l'ensemble des techniques mises en œuvre jusqu'alors pour accélérer le traitement d'un unique flot d'instructions. L'exécution des tâches est alors séquentielle et l'ordonnancement est laissé à la charge du système d'exploitation. On peut alors parler de virtualisation de l'exécution multitâche car l'utilisateur a l'impression que ces tâches s'exécutent en parallèle puisqu'elles évoluent dans le temps de manière pseudo-concurrente. Pour cette virtualisation, deux techniques d'accélération des processeurs monotâches sont largement utilisées. La première appelée *parallélisme temporel* consiste à réduire le temps d'exécution en découpant le traitement d'une instruction en plusieurs étapes successives. C'est l'exécution *pipeline* (Hennessy *et al.*, 2003; Ramamoorthy *et al.*, 1977).

La seconde solution nommée *parallélisme spatial* repose sur la multiplication des ressources de calcul. 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. C'est le cas par exemple des architectures SIMD (Tanenbaum, 2006; Hord, 1982) ou VLIW (Fisher *et al.*, 2005; Rau *et al.*, 1989). Mais la simplicité de leur structure de contrôle impose de dépendre fortement de la capacité des outils à exploiter l'ensemble des ressources de calcul disponibles.

Au contraire, lorsque l'architecture s'occupe dynamiquement de tous ces aléas d'exécution, le parallélisme est exploité de manière implicite. Ces architectures sont du type superscalaire (Taha *et al.*, 2008; Stephens *et al.*, 1990) (voir figure 1). Les avantages de cette approche sont la simplicité de la description du parallélisme et sa gestion dynamique durant l'exécution. Néanmoins, la complexité des mécanismes de spéculation, d'exécution non ordonnée ou de prédiction de branchement engendre des mises en œuvre inefficaces du point de vue énergétique et de la quantité de transistors.

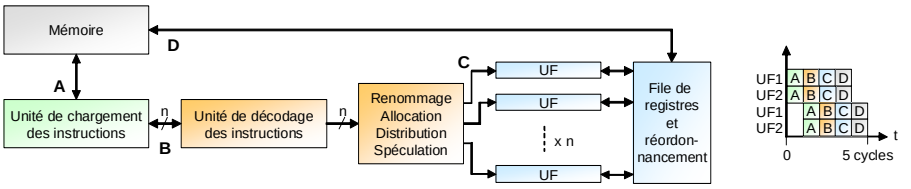

Figure 1. *Architecture superscalaire et principe d'exécution. Une architecture superscalaire contient plusieurs pipelines en parallèle. Il est possible d'exécuter plusieurs instructions simultanément. Des unités de renommage ou de spéculation par exemple permettent de maximiser l'occupation des unités fonctionnelles (UF) et ainsi le parallélisme d'instructions*

L'utilisation des structures de calcul classiques pour l'exécution de multiples tâches en parallèle n'est plus envisageable tant les besoins de performance sont élevés. Leur utilisation conduit inexorablement à des structures ou des outils de plus en plus complexes pour peu de gain en performance addtionnel. C'est pourquoi il faut intégrer des mécanismes de contrôle supplémentaires et concevoir des architectures multitâches conçues pour ces besoins particuliers.

Cet article introduit tout d'abord la notion de parallélisme et présente les moyens architecturaux qu'il est possible de mettre en œuvre pour son exploitation. Il rappelle les limitations du parallélisme d'instructions et introduit le parallélisme de tâches. Ensuite, il apporte une nouvelle classification des architectures parallèles exploitant le parallélisme de tâches 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 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 enrichi des données disponibles dans la littérature permet de comprendre pourquoi elles apportent un gain de performance et en quoi elles sont limitées. L'objectif de cet article est d'étudier les architectures multitâches pour les systèmes embarqués et d'évaluer leurs domaines d'application possibles. Les solutions couramment utilisées par les supercalculateurs et autres machines de calcul parallèle (machines vectorielles ou parallèles (van der Steen, 2005)) ne sont pas abordées dans cet article, puisqu'elles ne peuvent pas être intégrées dans les systèmes embarqués. Nous nous intéressons essentiellement aux solutions intégrées sur une même puce.

2. Définition du parallélisme et limitations

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 utilisés 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 (Bernstein, 1966). On dit également que les traitements sont *indépendants sur le séquencement* si l'exécution d'un des traitements n'est pas conditionnée 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 (Wall, 1991; Butler *et al.*, 1991). 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 (Hennessy *et al.*, 2003). 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 zones de stockage qui contiennent les dernières valeurs des registres utilisés par le programme. Les dépendances de ressources sont alors distinguées des autres dépendances. Enfin, des techniques de compilation sont employées pour limiter le coût de l'extraction du parallélisme au cours de l'exécution. Elles permettent de supprimer des dépendances de contrôle et de données, ainsi que d'extraire, par exemple, le parallélisme dans les cœurs de boucle (Moudgill *et al.*, 1993).

En fait, deux traitements 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. Pour permettre l'exécution concurrente de deux traitements non indépendants devant partager une ressource commune, il faut mettre en place des mécanismes garantissant un accès exclusif à cette ressource. Par ailleurs, lorsque deux traitements doivent se succéder, il faut garantir un accès ordonné aux 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, p le temps d'exécution de la partie P et n le nombre de processeurs, la loi d'Amdahl (Amdahl, 1987) 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 (Gustafson, 1988) 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.

La durée d'exécution d'un programme peut ainsi être réduite 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 pouvant exécuter plusieurs flots d'instructions simultanément. Ces architectures sont appelées *architectures multitâches*.

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 des moyens de comparaison qui nous seront utiles pour comparer ces différentes solutions. Ensuite, nous présenterons l'ensemble des architectures exploitant le parallélisme de tâches et nous discuterons des domaines d'utilisation possibles.

3. Classification des architectures multitâches et définition des besoins

Comme l'attestent les tableaux 1 à 6, un nombre très important d'architectures multiprocesseurs ou multitâches existe. Le terme *IP* utilisé signifie *Intellectual Property* et désigne des éléments matériels adjoints aux processeurs pour réaliser certains traitements spécifiques. Pour simplifier leur comparaison, quelques métriques sont utilisées et présentées par la suite.

3.1. *Métriques d'évaluation de performance*

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. Ce paramètre influence directement les coûts de production et de développement du circuit. Une bonne utilité des transistors employés est primordiale lors de la conception d'une architecture pour les systèmes embarqués. C'est pourquoi nous définissons l'*efficacité transistor* comme premier critère. Il représente le nombre de milliards d'opérations par seconde et par millions de transistors $(GOPS/MT)$.

La consommation d'énergie est également un moyen de comparaison incontournable. Les systèmes embarqués doivent tous limiter leur consommation d'énergie pour des raisons économiques ou d'autonomie. Une autre grandeur caractéristique est donc le nombre de milliards d'opérations par seconde et par Watt $(GOPS/W)$.

Ces deux critères sont particulièrement importants à prendre en compte lors de la conception d'architectures et en particulier dans le contexte des systèmes embarqués. Ainsi, il est très important de ne pas atteindre de hautes performances au détriment de la complexité ou de la consommation d'énergie par exemple. Par ailleurs, une bonne mise en œuvre d'une architecture multiprocesseur réside dans sa capacité à multiplier le nombre de cœurs de calcul simples pour atteindre des performances plus élevées.

Néanmoins, de nombreux autres critères doivent être pris en compte même si ceux-ci sont difficilement quantifiables. En effet, une évaluation précise des efficacités transistors et énergétiques repose sur la capacité de l'architecte à estimer les performances efficaces de l'architecture. Si le parallélisme de données est insuffisant, certaines architectures deviennent inadaptées pour les systèmes embarqués de part leurs faibles efficacités transistor et énergétique qui en résultent.

nom	nb proc	détails processeurs	nb IP	mémoire
Processeurs multitâches avec exécution successive				
MIPS 34K	1	RISC + 2 threads	0	(32KBI + 32KBD)
Ubicom MASI	1	RISC + 8 threads	0	(256KBI + 64KBD)
Processeurs multitâches avec exécution bloquée				
Infineon Tricore 2	1	superscalaire 3 voies + 2 threads	0	160KB
Processeurs multitâches avec exécution simultanée				
Intel Pentium 4 HT	1	superscalaire et SIMD 3 voies + 2 threads	0	128KB(L1) + 2MB(L2)
Compaq Alpha 21464	1	superscalaire 8 voies + 4 threads	0	(64KBI + 64KBD) + 3MB(L2)

Tableau 1. *Processeurs multitâches*

nom	techno. (nm)	fréqu. (MHz)	nb trans. (million)	perf. crêtes (GOPS)	puiss. (W)	eff. trans. (GOPS/MT)	eff. énerg. (GOPS/W)
Processeurs multitâches avec exécution successive							
MIPS 34K	65	700	∼ 1,5	1,3	0,2	∼ 0,86	10,85
Ubicom MASI	130	250	∼ 12,5	0,25	nc	∼ 0,02	nc
Processeurs multitâches avec exécution bloquée							
Infineon Tricore 2	130	500	∼ 1,5	0,75	0,25	∼ 1,3	3
Processeurs multitâches avec exécution simultanée							
Intel Pentium 4 HT	90	3800	169	11,4	115	0,07	0,1
Compaq Alpha 21464	130	1800	250	∼ 14	150	∼ 0,06	0,1

Tableau 2. *Processeurs multitâches*

nom	nb proc	détails processeurs	nb IP	réseau interco.	mémoire
Multiprocesseurs symétriques avec mémoire distribuée					
Freescale 8641D	2	superscalaire 3 voies (PowerPC e600)	0	multibus	2x (32KB + 1MB (L2))
Intel 80 Core	80	2 FPMAC + proc. VLIW 96 bits	0	maillé	80 x (2KBD + 3KBI)
Tilera Tile64	64	VLIW 3 voies	0	maillé	64 x 80KB
Hitachi SH-X3 Multicore	4	superscalaire 4 voies	3	bus	4x (32KBI + 32KBD + 24KBI + 128 KBD)
Ambric Am2045	45	360 RISC 32 bit	0	hiérarchique	13KB par brique
Intel Larrabee	48	Pentium-like + 4 threads + vector. 16-way	0	anneau	48x ((32KBI + 32KBD) + 256KB (L2))
Intelasys S-24A	24	RISC 18 bit	0	maillé	4x 64KB
Picochip PC102	240	240 VLIW 3 voies (16 bit)	14	bus	240x 768B + 64x 8KB + 4x 64KB
Raytheon Monarch	6	RISC (5 étages)	1	maillé	6x 2MB + 31x (4x1KB)
Multiprocesseurs symétriques avec mémoire partagée					
Azul Sys. Vega2	48	RISC 64 bit	0	hiérarchique	48x (16KBI + 16KBD) + 6x 1MB (L2)
Freescale Muli-core	32	superscalaire 2 voies (PowerPC e500-mc)	5	multibus	nc
Stanford Univ. Hydra	4	superscalaire 4 voies (R10000)	0	bus	4x (8KBI + 8KBD) + 128KBD (L2)
Broadcom BMC1480	4	superscalaire 4 voies (MIPS64)	0	bus	4x (32KBI + 32KBD) + 1MB (L2)
Cavium Net. CN5860	16	superscalaire 4 voies (MIPS64)	0	bus	16x (32 KBI + 16KBD) + 2MB (L2)
ARM ARM11 MPCore	4	ARMv6 SIMD	0	bus	4x (16KBI + 16KBD)
PA Semi. 1682M	2	superscalaire 3 voies (PowerPC-like)	0	crossbar	2x (64KBD + 64KBI) + 2MB (L2)
TI TMS320VC5441	4	DSP c54x	0	bus hiérar-chique	4x (32KBI + 64KBD + 64 KBD) + 256KB (L2)
AMD Barcelona	4	AMD64 superscalaire 3 voies 64 bit	0	bus	2x (64KBI + 64KBD + 1MB (L2)) + 2MB (L3)
Compaq Piranha	8	superscalaire 4 voies (Alpha 21364)	0	maillé	8x (8KB + 128KB (L2))
IBM Power 4	2	superscalaire 4 voies	0	bus	2x (64KBI + 64KBD) + 1,44MB (L2)

Tableau 3. *Multiprocesseurs symétriques*

nom	techno. (nm)	fréqu. (MHz)	nb trans. (million)	perf. crêtes (GOPS)	puiss. (W)	eff. trans. (GOPS/MT)	eff. énerg. (GOPS/W)
Multiprocesseurs symétriques avec mémoire distribuée							
Freescale 8641D	90	1500	∼ 150	7	15-25	> 0,05	0,28
Intel 80 Core	65	4000	100	1280	181	12,8	7,1
Tilera Tile64	90	900	nc	192	0,17-0,3	nc	640
Hitachi SH-X3	90	600	nc	16,8	1,5	nc	11,2
Ambric Am2045	130	333	117	1080	3-14	9,2	77
Intel Larrabee	45	1700-2500	nc	1500	> 150	nc	nc
Intelasys S-24A	nc	1000	nc	24	0,15	nc	160
Picochip PC102	130	160	200	76	4	0,38	19
Raytheon Monarch	90	333	nc	256	42-85	nc	3-6
Multiprocesseurs symétriques avec mémoire partagée							
Azul Sys. Vega2	90	nc	812	nc	237	nc	nc
Freescale Muli-core	45	1500	nc	60	nc	nc	nc
Stanford Univ. Hydra	250	250	∼ 20	1	30	0,05	0,03
Broadcom BMC1480	130	1000	∼ 13	8	23	0,6	0,34
Cavium Net. CN5860	90	1000	∼ 50	32	40	0,6	0,8
ARM ARM11 MPCore	90	620	∼ 2,5	2,5	0,266	1	9,4
PA Semi. 1682M	65	2000	200	16	5-13	0,08	1,2-3,2
TI TMS320VC5441	90	133	> 15	1	0,32	∼ 0,06	3,1
AMD Barcelona	65	2,5	450	30	65	0,06	0,46
Compaq Piranha	180	500	nc	4	nc	nc	nc
IBM Power 4	180	1300	170	2,6	125	0,01	0,02

Tableau 4. *Multiprocesseurs symétriques*

nom	nb proc	détails processeurs	nb IP	réseau interco.	mémoire
Multiprocesseurs asymétriques homogènes avec mémoire distribuée					
CEA LIST SCMP	8	RISC 32 bit (MIPS32)	0	multibus	8x (8KBD + 8KBI) + 16x 64KBD + 512KB
NXP Xetal II	320	40x 8 SIMD	0	segmenté	112 KBI + 40x8x 2KBD + 30KB (in) + 30KB (out) + registres 320 pixels
IBM CELL	9	PowerPC 64 bit SMT 2 voies + 8 SIMD	0	anneau	8x 256KB (SPE) + (32KB + 512KB (L2)) (PPE)
ClearSpeed CSX600	96	RISC 32 bit	0	nc	96x (32 mots + 6KB) + 128 KB
Stream Proc. SP16	82	80 VLIW 16 voies + 2 RISC 32 bit (MIPS32)	0	segmenté	16x (16KB + 302 mots) + 128KB + 96KBI + 2x (16KBI + 16KBD)
Intel IXP2800	17	32-bit RISC (Intel XScale) + 16 RISC multitâches	2	bus	(32KBI + 32KBD) + 16x 2KB
Multiprocesseurs asymétriques homogènes avec mémoire partagée					
Craddle Tech. 3616	24	8 RISC + 16 DSP (SMID + MAC +FPU)	0	bus	2x (32KBI + 128KBD + 8x 192 mots)
Univ. Texas TRIPS	2	SMT 4 voies	0	maillé	2x (8KBD + 16KBI) + 16x 64KB (L2)
Plurality Hypercore	64	RISC 32 bit	0	nc	nc
Multiprocesseurs asymétriques hétérogènes avec mémoire partagée					
HiBRID-SoC	3	DSP + RISC 32 bit + VLIW	0	bus	(16KBI + 4KBD) + (8KBI + 1KBD) + (4KBI + 4KBD)
Stretch Inc. S6000	1	VLIW 2 voies	4	bus	(32KBI + 32KBD + 64KB + 32 registres 128 bits)
TI OMAP 3430	1	superscalaire 2 voies (ARM Cortex-A8)	70	bus	(32KBI + 16KBD) + 2MB (L2)
Mobileye EyeQ2	2	SMT 4 voies (MIPS64 34K)	8	bus	2x (32KBI + 32KBD + 40KB)
Multiprocesseurs asymétriques hétérogènes avec mémoire distribuée					
ST Micro. Nomadik	3	RISC (ARM926) + 2 DSP/VLIW	5	bus	(32KBI + 16KBD) + 2x (48KB)
LSI Domino	3	2x RISC + DSP	1	bus	nc
NXP Nexperia	2	RISC (MIPS) + VLIW 5 voies	50	bus	(32KBI + 16KBD) + (16KBI + 32KBD)
Multiprocesseurs symétriques homogènes multitâches					
Sun UltraSparc-T3	16	superscalaire 2 voies + SMT 2 threads	0	crossbar	4x (32KBI + 2x32KBD) + 2MB (L2)
RMI XLR732	8	SMT 4 threads	0	crossbar	8x (32KBI + 32KBD) + 2MB (L2)
IBM Power6	2	supercalaire 7 voies + SMT 2 threads	0	crossbar	2x 64KB + 2x 4MB (L2) + 32MB (L3)

Tableau 5. *Multiprocesseurs asymétriques et symétriques multitâches*

nom	techno. (nm)	fréqu. (MHz)	nb trans. (million)	perf. crêtes (GOPS)	puiss. (W)	eff. trans. (GOPS/MT)	eff. énerg. (GOPS/W)
Multiprocesseurs asymétriques homogènes avec mémoire distribuée							
CEA LIST SCMP	65	500	nc	4	nc	0,42	nc
NXP Xetal II	90	84-150	63	107	0,6	35,6	178
IBM CELL	65	6000	234	512	< 80	2,2	6,4
ClearSpeed CSX600	130	250	nc	25	10	nc	2,5
Stream Proc. SP16	130	800	34	160	10	4,7	16
Intel IXP2800	90	1400	82	25,2	32	0,3	0,78
Multiprocesseurs asymétriques homogènes avec mémoire partagée							
Craddle Tech. 3616	130	375	nc	96	4,5	nc	21,4
Univ. Texas TRIPS	130	500	170	16	nc	0,09	nc
Plurality Hypercore	90	150	nc	nc	nc	nc	nc
Multiprocesseurs asymétriques hétérogènes avec mémoire partagée							
HiBRID-SoC	180	145	25	3,2	1,76	0,23	1,8
Stretch Inc. S6000	130	300	>4	>12	0,5	~ 3	~ 24
TI OMAP 3430	65	> 550	nc	nc	nc	nc	nc
Mobileye EyeQ2	90	> 166	nc	nc	~ 3	nc	nc
Multiprocesseurs asymétriques hétérogènes avec mémoire distribuée							
ST Micro. Nomadik	130	350	nc	1500	nc	nc	nc
LSI Domino	130	~ 150	nc	45,3	nc	nc	nc
NXP Nexperia	180	150	35	nc	4,5	nc	nc
Multiprocesseurs symétriques homogènes multitâches							
Sun UltraSparc-T3	65	2300	410	73,6	250	0,18	0,29
RMI XLR732	90	1200	333	9,6	49	0,03	0,2
IBM Power6	65	5000	790	20	100	0,06	0,2

Tableau 6. *Multiprocesseurs asymétriques et symétriques multitâches*

Ainsi, la complexité de la programmation ou de la mise en œuvre des synchronisations, la capacité de migration des tâches entre les ressources de calcul, les entrées-sorties, la régularité de l'architecture, son déterminisme, sa prédictibilité, sa hiérarchie mémoire ou son réseau d'interconnexion sont autant de critères à prendre en compte.

3.2. *Classification des architectures multitâches*

La première classification des architectures parallèles a été proposée par (Flynn, 1972). 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 les différences et les caractéristiques majeures qui existent entre les architectures multitâches. Ainsi, toutes les architectures multitâches sont de type MIMD alors qu'en pratique elles peuvent être très différentes. Nous choisirons donc par la suite une autre classification.

Cette classification propose deux grandes catégories : les *processeurs multitâches* et les *multiprocesseurs*. Les processeurs multitâches regroupent les architectures qui sont capables d'exécuter plusieurs tâches simultanément sur une seule unité de traitement. Au contraire, les multiprocesseurs ou MPSoC (*Multi-Processor System on Chip*) sont des architectures constituées de multiples processeurs monotâches ou multitâches. Les architectures multiprocesseurs constituées de processeurs monotâches peuvent être symétriques ou asymétriques. Cette distinction permet de mettre en évidence deux types de modèle d'exécution distinct. Enfin, les architectures multiprocesseurs asymétriques ont été regroupées en deux grandes catégories : homogènes et hétérogènes. Les solutions hétérogènes addressent des domaines d'application spécifiques et ne peuvent donc pas être comparées aisément avec les solutions homogènes. Les architectures multiprocesseurs considérées dans cet article sont toutes intégrées sur une même puce. Les sections suivantes présentent dans l'ordre ces deux grandes familles d'architectures parallèles. Les critères introduits précédement seront utilisés

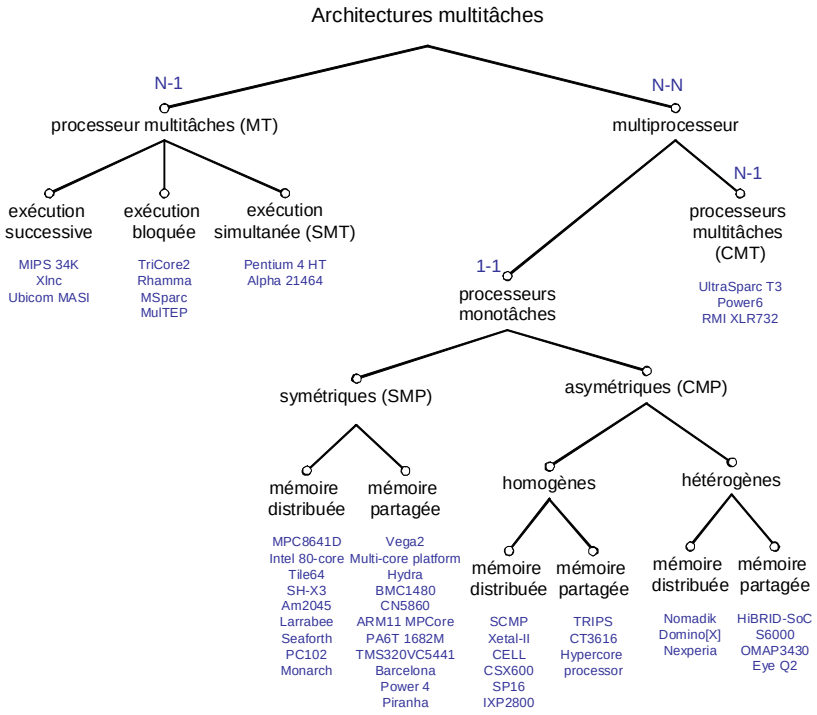

Figure 2. *Classification des architectures multitâches sur une même puce (les architectures multiprocesseurs sont détaillées dans les tableaux de 1 à 6)*

comme éléments de comparaison et permettront de discuter l'intérêt de chacune des solutions étudiées.

3.3. *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 (Tullsen *et al.*, 1995). Ceci est principalement dû aux mécanismes de spéculation qui n'ont pas un comportement optimal, au parallélisme d'instructions qui est limité et à la hiérarchie mémoire. Le principe des processeurs multitâches est donc d'occuper ses unités de calcul par d'autres tâches. Ces architectures exécutent alors plusieurs flots d'instructions indépendants sur plusieurs flots de données.

De manière générale, un processeur multitâche garde le principe d'exécution des architectures superscalaires ou VLIW. Néanmoins, certains étages du pipeline se complexifient pour exécuter plusieurs flots de données et d'instructions différents. Ainsi,

comme le montre la figure 3, plusieurs instructions sont sélectionnées en parallèle à des emplacements mémoires différents à l'aide de multiples compteurs de programme. Pour pouvoir accéder à plusieurs flots indépendants et soutenir l'alimentation en instructions du pipeline, 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. L'efficacité de ces architectures est largement influencée par l'efficacité des communications entre les tâches. Ainsi, 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.

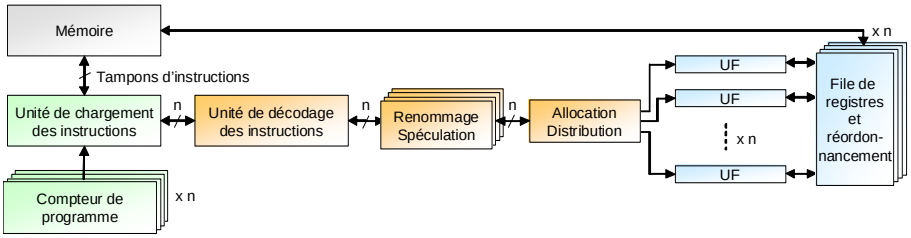

Figure 3. *Processeur multitâche et principe d'exécution. Un processeur multitâche est capable d'accéder à plusieurs flots d'instructions différents, à chaque cycle d'horloge, afin de maximiser l'occupation des unités fonctionnelles (UF)*

Il existe de nombreux processeurs multitâches dans la littérature (Towner, 2002; Ungerer *et al.*, 2002; Iannuci *et al.*, 1994). Ils peuvent être soit basés sur du parallélisme de tâches explicite ou implicite. L'exécution implicite de multiples tâches consiste a créer dynamiquement des tâches par des mécanismes de spéculation à partir de programmes monotâches. Celle-ci engendre des architectures très complexes, destinées aux supercalculateurs, qui ne seront pas traitées dans cet article. Au contraire, l'exécution explicite de tâches consiste à simplement exécuter de multiples tâches sur des ressources de calcul partagées. Les manières d'exécuter ces tâches en parallèle peuvent être regroupées en trois grandes catégories. La première technique appelée *exécution successive* ou *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*. Quelques exemples de processeurs multitâches sont décrits dans les tableaux 1 et 2.

3.3.1. *Exécution successive*

Les processeurs multitâches à exécution successive changent la tâche à exécuter après chaque cycle d'exécution (figure 4-a). Les architectures MIPS 34K (MIPS, n.d.) et Ubicom MASI (Ubicom, n.d.) 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. La durée 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 (voir section 2). Toutes les caractéristiques et limitations inhérentes au parallélisme d'instructions s'appliquent pour les processeurs multitâches même de manière plus limitée. Par ailleurs, 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.

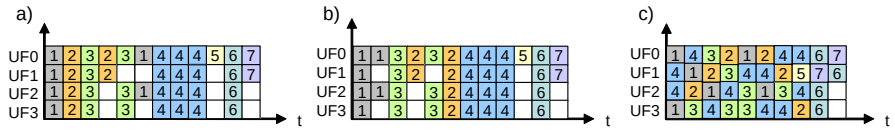

Figure 4. *Exemple d'exécution dans un processeur multitâche. Pour cet exemple, un ensemble de tâches indépendantes* $\{1, 2, 3, 4, 5, 6, 7\}$ *est choisi. A chaque cycle d'horloge, plusieurs instructions sont envoyées aux unités d'exécution (UFi). Les différentes stratégies d'exécution portent sur le choix des flots d'instructions sélectionnés. 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* 3 *permet l'exécution de la tâche* 2*. Enfin, l'exécution simultanée exécute indifféremment les tâches en fonction de la disponibilité des ressources de calcul (c)*

3.3.2. *Exécution bloquée*

L'approche avec exécution bloquée consiste à exécuter chaque tâche jusqu'à ce qu'elle rencontre une situation bloquante (figure 4-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. Les processeurs multitâches ont l'avantage de ne pas avoir à multiplier la taille de la file de

registres par le nombre de tâches pouvant être exécutées en parallèle. 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. Les blocages disparaissent alors pour la plupart et n'engendrent plus de changement de contexte si l'architecture dispose de plusieurs files de registres. Au contraire, les architectures avec changement de tâche dynamique souffrent d'une pénalité supplémentaire 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. Par exemple, le processeur Infineon TriCore 2 (Infineon,) permute entre deux tâches lors de défauts de cache et sur l'occurence d'autres signaux de contrôle gérés dynamiquement.

3.3.3. *Exécution simultanée*

Contrairement aux deux autres techniques d'exécution qui sont utilisées avec des processeurs pipeline ou VLIW, l'exécution simultanée est associée uniquement 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 (Lo *et al.*, 1997). 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 4-c). Les pénalités qui interviennent dans l'exécution de chacune des tâches sont réduites considérablement par l'exécution simultanée d'autres instructions. Les processeurs Compaq Alpha 21464 (Preston *et al.*, 2002) et Intel Pentium 4 HT (Intel, 2008) sont quelques exemples d'architectures SMT.

Les avantages des architectures SMT 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, les 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 ou de la file de registres doit permettre l'exécution simultanée de plusieurs tâches.

3.3.4. *Synthèse des processeurs multitâches*

Depuis peu, certains processeurs multitâches conçus pour les systèmes embarqués apparaissent afin de répondre aux besoins des nouvelles applications. La solution à exécution successive, choisie par MIPS (MIPS, n.d.) par exemple, permet de passer très rapidement d'une tâche à une autre et est donc particulièrement adaptée pour la préemption de tâches. Dans un contexte très dynamique et fortement dépendant des

données, cette solution peut être intéressante. De plus, la multiplication de la file de registres est peu pénalisante puisque l'augmentation de surface reste inférieure à 12 %. De même, pour l'exécution bloquée, les mécanismes mis en œuvre dans le processeur TriCore 2 (Infineon,) engendre un coût silicium inférieur à 17 %. L'avantage de cette dernière solution réside dans sa simplicité et est à préférer lorsque l'exécution est maîtrisée ou si l'on dispose d'outils de compilation performants capables de minimiser les temps de préemption. Par ailleurs, le déterminisme ou la prédictibilité de l'exécution sur de telles structures, avec exécution successive ou bloquée, peuvent devenir un problème majeur qui impose la non-utilisation de ces processeurs. En effet, si l'application traitée est dynamique, le nombre de préemption peut être variable, et même si l'ordre d'exécution des instructions est maîtrisé, ceci peut avoir un impact important sur le déterminisme global du système.

Contrairement aux processeurs multitâches à exécution successive ou bloquée qui peuvent être avantageusement intégrés dans des systèmes embarqués, les processeurs SMT sont destinés aux supercalculateurs ou à être intégrés dans des systèmes peu contraints par les coûts d'intégration ou la consommation d'énergie. Comme le montre le tableau 2, leurs efficacités transistor et énergétique restent très faibles comparées aux solutions avec exécution successive et bloquée. Le cas du processeur Ubicom MASI est un peu particulier car il supporte un nombre important de contextes d'exécution et ceci contribue à réduire son efficacité transistor. Ces architectures souffrent toutes d'une forte complexité et engendrent une importante consommation d'énergie. Par ailleurs, l'exécution des tâches est très peu déterministe et l'estimation du temps d'exécution d'une tâche reste difficile, voire impossible, dans un contexte temps-réel.

Les processeurs multitâches améliorent l'occupation des unités fonctionnelles des architectures monotâches (Olukotun *et al.*, 1996). Ils sont capables d'exécuter plusieurs instructions issues de différentes tâches simultanément. Quelle que soit la façon dont les tâches sont exécutées, ces architectures offrent une solution au problème d'exploitation de programmes multitâches. Néanmoins, le nombre de tâches qu'il est possible d'exécuter en parallèle reste limité. Il est donc intéressant d'étudier des solutions qui consistent à regrouper sur un même circuit plusieurs processeurs monotâches ou multitâches.

3.4. *Les multiprocesseurs*

Les architectures multiprocesseurs consistent à intégrer de multiples processeurs de calcul sur une même puce (Jerraya *et al.*, 2005). La figure 5 rappelle la classification présentée en section 3. A performance égale, ces solutions peuvent conserver des fréquences de fonctionnement et des tensions 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 170 processeurs MIPS 34K, soit un gain théorique en performance de 16. La première famille de solutions mul-

tiprocesseurs étudiée repose sur la mise en œuvre de plusieurs cœurs de processeurs monotâches indépendants autour d'un réseau partagé.

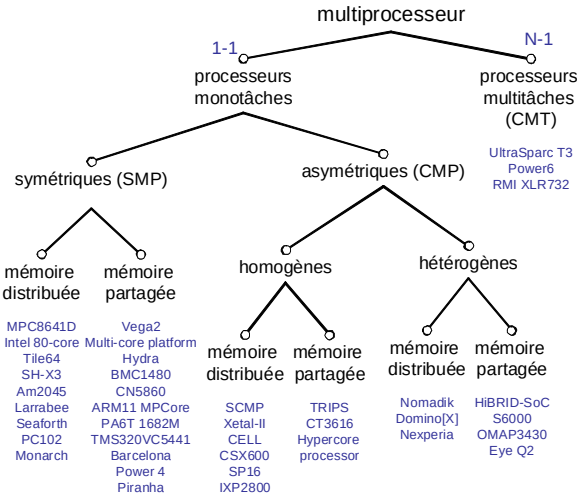

Figure 5. *Rappel de la classification des architectures multiprocesseurs*

3.4.1. *Avec des processeurs monotâches*

3.4.1.1. Symétriques

L'association de plusieurs processeurs indépendants sur un réseau partagé conduit à des solutions appelées *multiprocesseurs symétriques* (SMP : *Symmetric Multiprocessor*). Ces architectures allouent statiquement ou dynamiquement des tâches sur les différents processeurs qui composent l'architecture. Par exemple, dans la figure 6, 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 (Carpenter *et al.*, 2004). Ainsi, 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.

Mémoire distribuée La première solution consiste à distribuer la mémoire entre tous les processeurs, comme le représente la figure 6-a. Les architectures MPC8641D

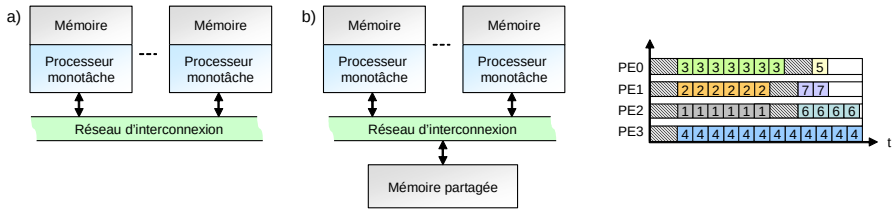

Figure 6. *Architecture multiprocesseur symétrique constituée de multiples processeurs monotâches. Soit un ensemble de tâches indépendantes* $\{1, 2, 3, 4, 5, 6, 7\}$*, chaque processeur élémentaire (PE) se voit attribuer un groupe de tâches à exécuter. L'ensemble des données peut être réparti dans des mémoires distribuées (a), ou être regroupé dans une mémoire partagée (b)*

de Freescale (Freescale, 2007), 80-Core d'Intel (Vangali *et al.*, 2007), SH-X3 d'Hitachi (Yoshida *et al.*, 2007), Monarch de Raytheon (Vahey *et al.*, 2006), Tile64 de Tilera (Tilera Corporation,) ou Larrabee d'Intel (Seiler *et al.*, 2008) 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 termes de performance.

Les communications par passage de message consistent à envoyer explicitement des messages aux processeurs distants. Cette technique était très utilisée dans des supercalculateurs organisés en grille. Elle a été transférée au niveau d'une puce lors de l'arrivée sur le marché de solutions multiprocesseurs à mémoires distribuées. Par ailleurs, la taille de la mémoire est d'autant plus grande que le nombre de processeurs est important. Ceci conduit donc à des temps d'accès trop importants qui imposent d'autres techniques de communication. Plus généralement, les systèmes par passage de messages sont plus aisés à concevoir matériellement même s'ils nécessitent un effort plus important de développement qu'avec une mémoire partagée. La mise en œuvre des communications repose en effet sur le programmeur et il est par conséquent responsable de la bonne coordination des synchronisations entre les différentes tâches.

L'utilisation de messages pour faire communiquer plusieurs éléments de calcul entre eux comporte d'autres inconvénients. Tout d'abord, le support est relativement lent et surtout est partagé entre toutes les ressources de calcul. Ensuite, l'empaquetage, la demande d'accès au moyen de communication, le transfert du message, l'attente d'un acquittement, sont autant d'étapes qui induisent des délais importants lors de chaque communication. Ces synchronisations sérialisent le flot de contrôle et chaque nouvelle dépendance oblige la tâche à interrompre son exécution. Les performances

globales de l'architecture sont alors réduites. Pour limiter l'impact des communications, une solution consiste à regrouper les informations au sein d'un même paquet pour augmenter les débits mais l'atomicité des messages reste un élément très pénalisant pour cette technique de communication.

Une solution pour améliorer l'efficacité des systèmes distribués consiste à distribuer logiquement la mémoire entre tous les processeurs. Cette dernière solution exploite massivement la localité des données. Les synchronisations sont alors limitées, voire interdites, durant toute l'exécution de la tâche. Dans le premier cas, les envois et les réceptions de données sont réalisés avec des proches voisins uniquement. Lorsque ce modèle d'exécution est synchronisé par les données, il est appelé *streaming* (Rixner, 2001). Dans le dernier cas, les données produites ne sont consommées qu'à la fin de la tâche productrice de ces données. Plus généralement, le CEA LIST propose une solution synchronisée par un contrôleur matériel centralisé capable de supporter le mode *streaming* et d'autres modes de fonctionnement nécessitant des synchronisations plus globales (Ventroux, 2006). Ce dernier modèle d'exécution est basé sur une mémoire pouvant être logiquement partagée ou distribuée selon le mode d'exécution choisi.

L'exécution streaming est basée sur une suite d'exécution de tâches synchronisées par les données entrantes (exécution flot de données synchrone) et organisées sous la forme d'un graphe de dépendances. Les architectures S-24A d'Intelasys (Intellasys,), PC102 de Picochip (Duller *et al.*, 2005) ou encore Am2045 d'Ambric (Halfhill, 2006) sont basées sur ce principe. Chaque tâche est une boucle interne prenant des données en entrée précédemment produites et retournent des données en sortie sauvegardées dans un espace partagé avec d'autres processeurs. Un processeur exploitant le streaming partage dans le temps ses ressources matérielles. Le traitement d'une tâche est réparti sur toutes les ressources disponibles. Au contraire le multiplexage spatial associe un traitement particulier à une ressource. Chaque ressource effectue toujours le même calcul sur des données différentes et les données se propagent de ressource en ressource jusqu'à l'obtention du résultat final. Ce mode s'apparente à un pipeline synchronisé sur les données. L'avantage de ce modèle d'exécution est qu'il favorise grandement la localité des données et rend prédictible le nombre des accès mémoires. Malgré de fortes contraintes lors de l'exécution des tâches, il impose finalement au développeur un modèle d'exécution et de programmation simple qui met en jeu de manière explicite les communications.

Mémoire partagée 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 multiprocesseurs Power 4 d'IBM (Tendler *et al.*, 2001), Piranha de Compaq (Barroso *et al.*, 2000), Hydra Chip de l'université de Stanford (Hammond *et al.*, 2000), Vega2 de

Azul Systems (Azul Systems,), Multi-Core de Freescale (Freescale, 2008), BMC1480 de Broadcom (Broadcom, 2008), MPCore d'ARM (ARM, 2008), 1682M de P.A. Semiconductors (P.A. Semiconductors,), TMS320VC5441 de Texas Instrument (Texas Instrument, b), Barcelona d'AMD (Dorsey *et al.*, 2007) ou Octeon CN5860 de la société Cavium Networks (Cavium Networks,). 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 inconsistance des données. Les communications par mémoire partagée peuvent être très efficaces dans les architectures multiprocesseurs dès lors que les cœurs de processeur et les mémoires sont intégrés sur la même puce. Il existe de nombreuses façons de mettre en œuvre ces communications. Chaque niveau de mémoire dans la hiérarchie peut être utilisé.

Une architecture avec les mémoires de niveaux 1 et 2 privés est intéressante du point de vue architectural au vu de sa simplicité (figure 7-a). Une telle architecture présente néanmoins deux inconvénients majeurs. Tout d'abord, toutes les communications distantes doivent passer par les deux niveaux de mémoire et sont caractérisées par des latences supplémentaires. Ensuite, comme dans tout système physiquement distribué, l'occupation des mémoires ne peut pas être optimale puisque l'espace mémoire n'est pas partagé en fonction des besoins de chaque tâche. Dans le cas de mémoires caches, pour améliorer l'utilisation de cette structure de communication, une méthode appelée Cooperative Caching a été proposée (Chang *et al.*, 2006). Elle permet à tous les processeurs d'écrire dans les mémoires caches L2 privées d'autres processeurs. Pour cela des techniques de transfert de cache à cache, de duplication ou de remplacement de données inactives ont été mises en œuvre.

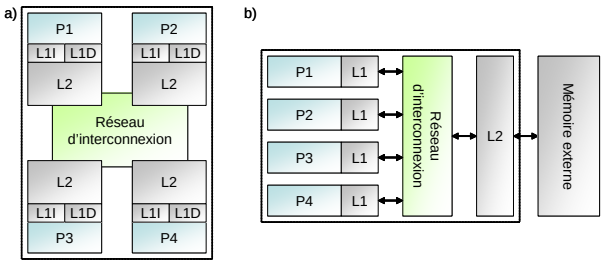

Figure 7. *Illustration d'une architecture avec les mémoires L1 et L2 privées (a) et d'une architecture avec la mémoire L2 partagée uniquement (b)*

Le partage de la mémoire de niveau 2 uniquement est la solution la plus utilisée dans les architectures multiprocesseurs commerciales de type SMP (figure 7-b). Elle présente l'avantage d'être une solution très rapide pour permettre à plusieurs processeurs de communiquer. Elle est néanmoins basée sur l'idée que la plupart des accès

mémoires se font avec la mémoire privée de niveau 1 et ainsi, des accès un peu plus lent à la mémoire partagée de niveau 2 n'ont pas trop d'incidences sur les performances. C'est d'ailleurs une hypothèse similaire qui conduit à accepter l'utilisation d'une mémoire L2 de taille relativement importante. Par ailleurs, les communications avec des éléments matériels extérieurs sont très pénalisantes et doivent être réduites le plus possible pour garantir de bonnes performances. Enfin, le partage d'une mémoire améliore son taux d'utilisation et donc l'efficacité transistor de l'architecture. L'inconvénient majeur est quant à lui une conséquence des hypothèses faîtes précédemment. L'accès à la mémoire partagée L2 est potentiellement pénalisant. Il faut prendre en compte la concurrence entre les différents processeurs, la taille importante de la mémoire partagée et la localité des données et des instructions.

Le partage de toute la hiérarchie pourrait quant à lui sembler être la solution la plus intéressante pour réduire les durées des communications. Cependant, cette solution n'est pratiquement jamais retenue car la fréquence de fonctionnement des processeurs dépend très largement du temps d'accès de leur mémoire de niveau 1 (Hennessy *et al.*, 2003). Bien sûr, il est toujours possible d'envisager cette solution dès lors que les pénalités dues au partage de la mémoire de niveau 1 peuvent être prises en compte au moment de la compilation. Un exemple d'architecture est le processeur multitâche à exécution bloquée MAJC-5200 de Sun (Tremblay *et al.*, 2000).

D'une manière générale, l'utilisation des mémoires partagées pour permettre aux tâches de communiquer simplifie la programmation de l'architecture mais réduit ses performances. En effet, la mise en œuvre de ces mécanismes induit des délais supplémentaires. Les mécanismes de protection indispensables pour garantir l'intégrité des données ainsi que la taille plus importante des mémoires ralentissent l'ensemble des accès mémoires. Par ailleurs, dans le cas de mémoires caches, des défauts de cache supplémentaires interviennent, voire dans le cas plus général, des conflits d'écriture ou de lecture conduisant parfois à des refus d'accès à l'espace mémoire pendant des durées non prédictibles. Enfin, chaque communication nécessite la traversée des différents niveaux de mémoire et des réseaux d'interconnexion les reliant.

Gestion de la cohérence 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. Pour les architectures SMP à mémoire distribuée, la synchronisation des tâches doit être logicielle afin de maintenir la cohérence entre tous les processeurs. En ce qui concerne les architectures SMP à mémoire partagée, les mécanismes mis en œuvre pour la gestion de la cohérence dépendent du caractère caché ou non des mémoires L1 et L2. Si les mémoires ne sont pas cachées, la solution est également logicielle et les synhronisations doivent être effectuées par l'utilisateur ou les outils de programmation. Au contraire, lorsque la hiérarchie mémoire est cachée, toute la complexité est reportée au niveau matérielle

(Eggers *et al.*, 1989). Plusieurs solutions sont alors possibles : par répertoire centralisé (*directory-based*) ou par espionnage (*snooping*) (Hennessy *et al.*, 2003).

Un répertoire centralisé consiste à maintenir dans un tableau partagé l'état courant des blocs mémoires répartis entre tous les processeurs (Censier *et al.*, 1978; Dubois *et al.*, 1982). Cette solution, largement utilisée dans les architectures à haute performance, engendre cependant un surcoût temporel important, pénalisant fortement les performances de l'architecture. Ce coût étant principalement dû à la latence d'accès au répertoire, une solution consistant à distribuer le répertoire dans les routeurs du réseau a été proposée (Hennessy *et al.*, 1999; Eisley *et al.*, 2006). Malheureusement, cette approche pose des problèmes de coût matériel importants.

Les techniques d'espionnage engendrent moins de pénalités temporelles grâce à la distribution du support matériel dans l'architecture. 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. Seuls changent les protocoles qui informent les autres processeurs d'une modification locale de leurs données. La solution adoptée peut soit invalider les données mises à jour dans les autres caches, où la donnée est également présente, pour forcer la mise à jour du cache lors d'une prochaine lecture de cette donnée (*Multiple Write Back*) ; soit mettre à jour tous les caches possédant la donnée lors de chaque écriture (*Multiple Write Through*). La première solution est la plus souvent retenue car la seconde a un impact très négatif sur la bande passante. Cependant, un modèle de programmation qui supporte une mise à jour différée (*cohérence faible*) peut réduire son incidence sur les performances. Par ailleurs, à mesure que le nombre de processeur augmente, les supports de communication deviennent de plus en plus complexe pour autoriser une bande passante plus importante. Ces techniques d'espionnage s'appliquent alors difficilement et engendrent une consommation d'énergie excessive.

De manière générale, ces mécanismes ont un coût d'intégration élevé et induisent des baisses de performance significatives dues, 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.

Synthèse Bien que ces solutions paraissent simples à mettre en œuvre, elles introduisent des problèmes d'accès aux données et des problèmes de cohérence. Par ailleurs, la maîtrise des synchronisations et des communications impacte grandement les performances de l'architecture. Tout cela peut être résolu par les outils de compilation ou le logiciel embarqué, 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. Le mode streaming est quant à lui très efficace lors des traitements flots de données et se prête bien aux calculs intensifs.

Le tableau 4 montre bien les différences importantes d'efficacité transistor et énergétique des solutions avec mémoire partagée comparées aux solutions avec mémoire distribuée. Ceci pour plusieurs raisons. Tout d'abord, les multiprocesseurs symétriques à mémoire distribuée sont majoritairement composés de processeurs destinés aux systèmes embarqués. Ensuite, la régularité des structures symétriques permettent d'atteindre des densités d'intégration plus importantes. Enfin, la plupart des solutions avec mémoire partagée possèdent des mécanismes de gestion de cohérence de cache qui pénalisent leur efficacité transistor et énergétique.

Pratiquement, les multiprocesseurs symétriques sont intéressants lorsque l'effort principal de développement est logiciel. Ils peuvent cependant engendrer des performances moindres dues aux différentes couches logicielles utiles au fonctionnement de l'architecture. Plusieurs niveaux d'interfaces logicielles (API : *Application Programming Interface, ou HAL : Hardware Abstraction Layer*) sont en effet souvent utilisés pour faciliter la programmation de ces architectures. Ces solutions peuvent également être envisagées lorsque l'application peut être facilement partitionnée, l'application n'est pas dominée par des traitements conditionnels, et les durées d'exécution des tâches dépendent peu des données ou du contexte d'utilisation. Enfin, il faudra favoriser une solution à mémoire partagée si des volumes de données importants sont à traités, tandis qu'une solution distribuée est bien adaptée aux traitements flots de données.

3.4.1.2. Asymétriques

Dans tous les modèles d'architecture étudiés jusqu'alors, les processus dominés par des traitements conditionnels sont exécutés avec les mêmes ressources que celles qui exécutent les calculs critiques ou intensifs. Autrement dit, des processeurs possédant des unités d'exécution spécialisées pour le traitement intensif sont utilisés pour exécuter des traitements comportant principalement des instructions de contrôle. 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 8, ce modèle a des similitudes avec les architectures superscalaires ou VLIW. 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 et l'état complet du système. 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, sans craindre une sous-utilisation trop importante lors des phases de synchronisation.

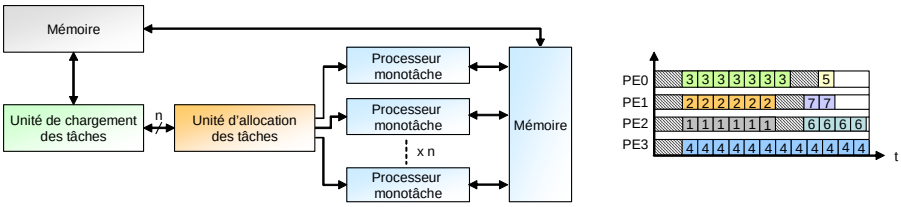

Figure 8. *Architecture multiprocesseur asymétrique constituée de multiples processeurs monotâches. Soit un ensemble de tâches indépendantes* $\{1, 2, 3, 4, 5, 6, 7\}$*, l'allocation des tâches est effectuée en-ligne par une unité de contrôle spécialisée. Si les processeurs élémentaires sont homogènes, les tâches peuvent s'exécuter sur n'importe quelle ressource de calcul*

Homogènes Les premières solutions CMP disposent de processeurs de calcul identiques. Elles sont dîtes *homogènes*. C'est par exemple le cas des multiprocesseurs CELL d'IBM (Pham *et al.*, 2006), Xetal-II de NXP (Abbo *et al.*, 2007), IXP2800 d'Intel (Intel,), CSX600 de ClearSpeed (ClearSpeed,) ou SP16 de Stream Processors (Khailany *et al.*, 2007) qui utilisent une mémoire distribuée. C'est aussi le cas des multiprocesseurs CT3616 de Cradle Technologie (Cradle Technology,), TRIPS de l'université du Texas (Sethumadhavan *et al.*, 2006) ou Hypercore de Plurality (Plurality,) qui sont eux basés sur une mémoire partagée. L'architecture SCMP du CEA LIST (Ventroux, 2006) a une gestion particulière de son espace mémoire puisque celui-ci est logiquement distribué pendant l'exécution des tâches et devient logiquement partagé à la fin de la production d'une donnée. Les avantages et inconvénients de la hiérarchie mémoire utilisée ont été présentés dans la section 3.4.1.1.

Hétérogènes A l'inverse, les multiprocesseurs asymétriques hétérogènes sont des solutions optimisées pour des domaines de traitements particuliers (Wolf, 2004). Les multiprocesseurs asymétriques hétérogènes OMAP 3430 de Texas Instrument (Texas Instrument, a), HiBRID-SoC de l'université de Hannover (Stolberg *et al.*, 2005), EyeQ2 de Mobileye (Mobileye,) ou S6000 de Stretch Inc. (Stretch Inc., 2008) utilisent une mémoire distribuée, tandis que Domino[X] de LSI (LSI,), Nomadik de ST Microelectronics (STMicroelectronics, 2004) ou Nexperia de NXP (Dutta *et al.*, 2001) sont basés sur une mémoire partagée. Ces architectures sont destinées aux applications de télécommunication mobile (OMAP 3430), de traitements multimedias (HiBRID-SoC, S6000, Domino[X], Nomadik, Nexperia), ou encore pour les systèmes de vision destinés à l'automobile (EyeQ2). Les unités matérielles optimisées qu'ils possèdent leur permettent de répondre à un besoin applicatif précis. Les performances sont alors améliorées ainsi que l'efficacité énergétique. L'allocation des tâches sur ces

ressources est souvent statique. De même, l'absence d'homogénéité empêche des algorithmes d'allocation de répartir efficacement la charge de calcul entre plusieurs ressources. 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.

Synthèse Une solution très hétérogène offre de meilleures performances puisqu'elle comporte des unités de calcul dédiées à des domaines applicatifs particuliers. Par contre, la spécificité de certains de ces opérateurs peut les rendre inopérants dans le cadre d'autres traitements, réduisant ainsi drastiquement les performances globales de l'architecture. D'un autre côté, une solution homogène offre davantage de flexibilité et peut exécuter plusieurs types d'applications.

Malheureusement, le manque d'information ne permet pas de conclure aisément sur l'approche permettant d'atteindre une meilleure efficacité énergétique et transistor. Le tableau 6 montre en général de bonnes prédispositions aux structures asymétriques avec un léger avantage aux solutions homogènes avec mémoire distribuée. Cependant, l'efficacité transistor et énergétique des solutions hétérogènes devraient être supérieure puisque ces structures répondent à un besoin spécifique et sont donc dimensionnées en conséquence. L'utilisation de structures accélératrices contribue également à augmenter sensiblement l'efficacité de ces architectures.

3.4.2. *Avec des processeurs multitâches*

Le dernier modèle d'architecture que nous allons étudier propose un compromis en associant les solutions SMP et les processeurs multitâches (figure 9). Il est appelé Chip Multithreading (CMT) (Spracklen L. and Abraham S., 2005). Le multiprocesseur UltraSparc T3 de Sun Microsystems (Tremblay *et al.*, 2008), le Power6 d'IBM (Friedrich *et al.*, 2007), ainsi que le RMI XLR 732 de la société Raze Microelectronics (Raza Microelectronics,) utilisent ce modèle d'exécution.

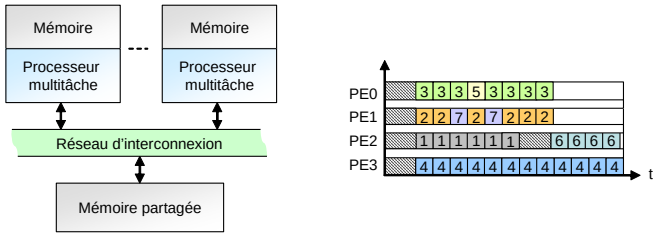

Figure 9. *Architecture multiprocesseur constituée de multiples processeurs multitâches. Soit un ensemble de tâches indépendantes* $\{1, 2, 3, 4, 5, 6, 7\}$*, chaque processeur élémentaire peut exécuter plusieurs tâches simultanément*

L'avantage de ces solutions réside dans une meilleure occupation des ressources de calcul qu'un modèle SMP. Il permet d'augmenter encore le parallélisme de tâches

et d'accélérer l'exécution des tâches allouées sur les ressources de calcul. 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. La multiplicité des contextes locaux aux tâches nécessitent des mécanismes complexes de migration. Enfin, comme le montre le tableau 6, l'efficacité énergétique et transitor des solutions SMP existantes reste très faible. Ceci est principalement dû au type de processeur utilisé et pourrait être réduit en utilisant des processeurs davantage destinés aux systèmes embarqués.

4. Synthèse

Les approches SMP et CMP minimisent 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 est équivalente à un processeur SMT pouvant exécuter 12 instructions en parallèle. Avec ce potentiel de parallélisme supplémentaire, le multiprocesseur obtient de meilleures performances (Tullsen *et al.*, 1995; Ungerer *et al.*, 2002). En effet, les processeurs multitâches sont limités en nombre d'instructions à chaque cycle d'exécution par la bande passante de la mémoire d'instructions. 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 multiprocesseurs. 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 multiprocesseurs consomment moins d'énergie que les processeurs multitâches (Sasanka *et al.*, 2003; Annavaram *et al.*, 2005). Du point de vue du déterminisme et de la prédictibilité, ces deux approches sont bien adaptées et conviennent parfaitement aux besoins temps-réel des systèmes embarqués. Bien sûr, une étude complémentaire sur les synchronisations des différents processus concurrents est nécessaire pour éviter les interblocages.

Néanmoins, il est vrai que les multiprocesseurs 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 potentiellement mieux aux besoins applicatifs. De plus, la latence engendrée par le contrôle ou le réseau d'interconnexion pénalise la réactivité des architectures multiprocesseurs et limite l'occupation des ressources de calcul.

Enfin, pour exploiter correctement les multiprocesseurs, le parallélisme de tâches doit être explicite. Ceci nécessite une programmation particulière des applications.

Le choix entre une approche symétrique (SMP) ou asymétrique (CMP) est plus délicat. La conception d'une architecture asymétrique est plus complexe mais offre des possibilités plus intéressantes pour exploiter le parallélisme de tâches. En effet, ces architectures supportent mieux la parallélisation de traitements intensifs à l'intérieur d'une même tâche. Elles améliorent l'efficacité transistor et énergétique en utilisant des ressources adaptées aux traitements à effectuer, voire spécifiques pour des traitements intensifs. Surtout, elles sont capables de tenir compte de la variabilité des applications dynamiques dont les durées d'exécution dépendent fortement des données. L'occupation effective des ressources de calcul peut alors être optimisée.

Par ailleurs, de façon générale une solution asymétrique, c'est-à-dire avec un contrôleur unique ou hiérarchique, offre une meilleure efficacité transistor. En effet, l'asymétrie de la structure autorise l'usage de ressources optimisées pour le contrôle ou le calcul. Le défaut majeur de l'asymétrie réside dans la mise à l'échelle de l'architecture. Un unique contrôleur limite le nombre de ressources qu'il est possible de piloter sans pénaliser fortement les performances globales. Par exemple, l'architecture *Hypercore Processor* de Plurality est limitée à 256 processeurs (Plurality,). Un autre avantage de l'asymétrie est de simplifier la programmation de l'architecture. En effet, ceci simplifie généralement la gestion des synchronisations et des communications et offre un moyen de mise au point simplifié. Ceci grâce à l'utilisation d'un contrôleur unique et centralisé.

Enfin, le modèle CMT consiste à multiplier le nombre de processeurs multitâches sur une même puce. Les solutions actuellement disponibles sur le marché sont principalement constituées de processeurs multitâches à exécution simultanée. Ces solutions conduisent à des multiprocesseurs très complexes avec une forte consommation d'énergie et un coût silicium important. Elles possèdent également les caractéristiques propres aux types de ressources multitâches utilisées concernant le déterminisme et la prédictibilité. Ces architectures sont donc à éviter dans un système devant répondre à de fortes contraintes temps-réel. Néanmoins, ce sont des solutions capable de répondre au problème du parallélisme de tâches et peuvent être utilisées efficacement dans des supercalculateurs ou pour le calcul intensif. D'autres solutions basées sur des processeurs multitâches plus simples peuvent cependant être envisagées pour les systèmes embarqués. Elles cumulent alors les avantages des processeurs multitâches aux solutions multiprocesseurs présentées précédemment.

5. Conclusion

La complexité et le parallélisme de tâches des applications étant de plus en plus importants, les systèmes embarqués doivent aujourd'hui supporter l'exécution de multiples tâches en parallèle.

Dans cet article, 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, 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 capables d'exécuter de multiples tâches en parallèle a été exploré. L'étude de l'ensemble de ces solutions a permis de comprendre pourquoi elles apportent un gain de performance et en quoi elles sont limitées. Ainsi, il nous a été possible d'associer à chacune des architectures multitâches un domaine d'utilisation possible.

Nous avons vu que de manière générale, les processeurs multitâches améliorent l'occupation des unités fonctionnelles des architectures monotâches. Certaines solutions, à exécution bloquée ou successive, restent intéressantes pour les systèmes embarqués. L'augmentation de la surface silicium reste mesurée et peut conduire à une augmentation conséquente des performances efficaces des processeurs. Ces solutions peuvent se montrer encore plus intéressantes dans un environnement SMP puisque les conflits d'accès mémoires ou les attentes dues aux synchronisations sont importants. Les temps d'attente peuvent être habilement utilisés pour exécuter concurremment d'autres tâches en attente d'exécution.

Au contraire, les processeurs multitâches à exécution simultanée ou les multiprocesseurs constitués de ces ressources de calcul, restent complexes et destinés aux supercalculateurs. L'utilisation d'une structure superscalaire à exécution non ordonnée est trop orientée vers la performance pour tenir compte des contraintes fortes liées aux systèmes embarqués.

L'analyse des architectures multiprocesseurs symétriques ou asymétriques constituées de processeurs monotâches est plus difficile. Elles dépendent principalement du type de processeur ou par exemple de réseau d'interconnexion utilisé. Les solutions mettant en œuvre des processeurs de calcul complexes visent des applications généralistes, tandis que les architectures asymétriques hétérogènes tentent de répondre à un besoin applicatif spécifique. Si le besoin applicatif est connu et fixé, une solution hétérogène disposant d'unités de calcul spécialisées semble être le meilleur choix. L'homogénéité des ressources de calcul conduit à de moins bonnes performances mais permet de supporter de multiples domaines applicatifs. Son utilisation autorise également la migration des tâches et permet d'optimiser l'exécution d'applications dynamiques. Enfin, l'asymétrie de l'architecture contribue à améliorer encore cela en offrant une vue globale sur l'état du système.

L'étude de chacune de ces solutions pour exploiter le parallélisme de tâches n'a pas fait apparaître une solution unique. Chaque architecture est à choisir en adéquation avec le type d'applications exécutées, la consommation d'énergie ou la surface silicium engendrée.

6. Bibliographie

Abbo A., Kleihorst R., Choudhary V., Sevat L., Wielage P., Mouy S., Heijligers M., « XETAL-II : A 107 GOPS, 600mW Massively-Parallel Processor for Video Scene Analysis », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, 2007.

Amdahl G., « Validity of the single processor approach to achieving large-scale computing capabilities », *AFIPS Conference*, p. 483-485, 1987.

Annavaram M., Grochowski E., Shen J., « Mitigating Amdahl's Law Through EPI Throttling », *IEEE International Symposium on Computer Architecture (ISCA)*, Madison, USA, June, 2005.

ARM, « ARM11 MPCore », http ://www.arm.com.

Azul Systems, « VEGA 2 : 48-Way Multicore Chip », http ://www.azulsystems.com.

Barroso L., Gharachorloo K., McNamara R., Nowatzyk A., Qadeer S., Sano B., Smith S., Stets R., Verghese B., « Piranha : a scalable architecture based on single-chip multiprocessing », *IEEE International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, 2000.

Bernstein A., « Program Analysis for Parallel Processing », *IEEE Transactions on Electronic Computer*, vol. 15, n° 5, p. 757-762, 1966.

Broadcom, « BCM1480 Quad-Core 64-bit MIPS® Processor », http ://www.broadcom.com.

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

Carpenter J., Funk S., Holman P., Anderson A. S. J., Baruah S., *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*, vol. 30, Chapman and Hall/CRC, chapter A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms, p. 1-30, 2004.

Cavium Networks, « OCTEON Plus CN58XX Multi-Core MIPS64 Based SoC Processors », http ://www.cavium.com.

Censier L., Feautrier P., « A New Solution to Coherence Problems in Multicache Systems », *IEEE Transactions on Computers*, vol. 27, n° 12, p. 1112-1118, December, 1978.

Chang J., Sohi G. S., « Cooperative Caching for Chip Multiprocessors », *IEEE International Symposium on Computer Architecture (ISCA)*, Boston, USA, 2006.

ClearSpeed, « The CSX600 architecture », http ://www.clearspeed.com.

Cradle Technology, « The Cradle 3616 », http ://www.cradle.com.

Dorsey J., Searles S., Ciraula M., Bujanos S. J. N., Wu D., Braganza M., Meyers S., Fang E., Kumar R., « Integrated Quad-Core Opteron Processor », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, USA, 2007.

Dubois M., Briggs F., « Effects of Cache Coherency in Multiiprocessors », *IEEE Transactions on Computers*, vol. 31, n° 11, p. 1083-1099, November, 1982.

Duller A., Towner D., Panesar G., Gray A., Robbins W., « PicoArray technology : the tool's story », *IEEE Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2005.

Dutta S., Jensen R., Rieckmann A., « Viper : A multiprocessor SOC for advanced set-top box and digital TV systems », *IEEE Journal of Design & Test of Computers*, vol. 18, n° 5, p. 21-31, 2001.

Eggers S., Katz R., « Evaluating the Performance of Four Snooping Cache Coherency Protocols », *IEEE International Symposium on Computer Architecture (ISCA)*, Jerusalem, Israel, June, 1989.

Eisley N., Peh L.-S., Shang L., « In-Network Cache Coherence », *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, USA, December, 2006.

Fisher J., Faraboschi P., Young C., *Embedded Computing : A VLIW Approach to Architecture, Compilers, and Tools*, Morgan Kaufmann Publishers, 2005.

Flynn M., « Very High Speed Computing Systems », *Proceedings of the IEEE*, vol. 54(2), p. 1901-1909, 1972.

Freescale, MPC8641D Integrated Host Processor Family Reference Manual, Technical report, May, 2007.

Freescale, « Freescale's Multi-core Platform », http ://www.freescale.com.

Friedrich J., McCredie B., James N., Huott B., Curran B., Fluhr E., Mittal G., Chan E., Chan Y., Sam D. P. C., Hung L., Clark L., Ripley J., Taylor S., Dilullo J., Lanzerotti M., « Design of the Power6 Microprocessor », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, USA, 2007.

Gustafson J., « Re-evaluating Amdahl's law », *Communications of the ACM*, vol. 31, n° 5, p. 532-533, May, 1988.

Halfhill T. R., « Ambric's new parallel processor : Globally Asynchronous Architecture Eases Parallel Programming », *Microprocessor Report*, 2006.

Hammond L., Hubbert B., Siu M., Prabhu M., Chen M., Olukotun K., « The Stanford Hydra CMP », *IEEE Micro Magazine*p. 71-84, 2000.

Hennessy J., Heinrich M., Gupta A., « Cache-Coherent Distributed Shared Memory : Perspectives on Its Development and Future Challenges », *Proceedings of the IEEE*, vol. 87, n° 3, p. 418-429, March, 1999.

Hennessy J., Patterson D., *Architecture - A quantitative approach, Third Edition*, ISBN : 1558605967, Morgan Kauffmann Publishers, 2003.

Hord R. M., *The Illiac-IV, The First Supercomputer*, ISBN : 0914894714, Computer Science Press, May, 1982.

Iannuci R., Gao G., Halstead R., Smith B., *Multithreaded Computer Architecture : A Summary of the State of the Art*, n° 0792394771, Kluwer Academic Publishers, 1994.

Infineon, « TriCore 2 », http ://www.infineon.com.

Intel, « Intel IXP2800 Network Processor », http ://www.intel.com.

Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. December, 2008.

Intellasys, « Seaforth : Scalable Embedded Array processing », http ://www.intellasys.net.

Jerraya A., Wolf W., *Multiprocessor Systems-on-Chips*, n° 012385251X, Elsevier, 2005.

Khailany B., Williams T., Lin J., Long E., Rygh M., Tovey D., Dally W., « A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, USA, 2007.

Lo J., Eggers S., Emer J., Levy H., Stamm R., Tullsen D., « Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading », *ACM Transactions on Computer Systems*, vol. 15, n° 2, p. 2, August, 1997.

LSI, « DOMINO[X] Pro », http ://www.lsi.com.

MIPS, « MIPS 34K MT », http ://www.mips.com.

Mobileye, « EyeQ2 », http ://www.mobileye.com.

Moudgill M., Pingali K., Vassiliadis S., « Register Renaming and Dynamic Speculation : An Alternative Approach », *International Symposium on Microarchitecture (MICRO-26)*, Austin, Texas, December, 1993.

Olukotun K., Nayfeh B., Hammond L., Wilson K., Chang K., « The Case for a Single-Chip Multiprocessor », *ACM International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, USA, October, 1996.

P.A. Semiconductors, « PA6T-1682M », http ://www.pasemi.com.

Pham D., Aipperspach T., Boerstler D., Bolliger M., Chaudhry R., Cox D., Harvey P., Harvey P., Hofstee H., Johns C., Kahle J., Kameyama A., Keaty J., Masubuchi Y., Pham M., Pille J., Posluszny S., Riley M., Stasiak D., Suzuoki M., Takahashi O., Warnock J., Weitzel S., Wendel D., Yazawa K., « Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor », *IEEE Journal of Solid-State Circuits*, vol. 41(1), p. 179-196, 2006.

Plurality, « The Hypercore Processor », http ://www.plurality.com.

Preston R., Badeau R., Bailey D., Bell S., Biro L., Bowhill W., Dever D., Felix S., Gammack R., Germini V., Gowan M., Gronowski P., Jackson D., Mehta S., Morton S., Pickholtz J., Reilly M., Smith M., « Design of an 8-wide superscalar RISC microprocessor with Simultaneous Multithreading », *IEEE International Solid-State Circuits Conference*, San Francisco, USA, February, 2002.

Ramamoorthy C., Li H., « Pipeline architecture », *ACM Computing Survey*, vol. 9, n° 1, p. 61-102, March, 1977.

Rau B., Yen D., Yen W., Towle R., « The Cydra 5 departemental supercomputer : Design philosophies, decisions, and trade-offs », *IEEE Computers*, vol. 22, n° 1, p. 12-34, January, 1989.

Raza Microelectronics, « XLR Family of Thread Processors », http ://www.razamicroelectronics.com.

Rixner S., *« A Bandwidth-efficient Architecture for a Streaming Media Processor »*, Master's thesis, Massachusetts Institute of Technology, 2001.

Sasanka R., Adve S., Chen Y.-K., Debes E., Comparing the Energy Efficiency of CMP and SMT Architectures for Multimedia Workloads, Technical Report n° UIUCDCS-R-2003-2325, University of Illinois at Urbana-Champaign, March, 2003.

Seiler L., Carmean D., Sprangle E., Forsyth T., Abrash M., Dubey P., Junkins S., Lake A., Sugerman J., Cavin R., Espasa R., Grochowski E., Juan T., Hanrahan P., « Larrabee : A Many-Core x86 Architecture for Visual Commputing », *ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, Los Angeles, USA, 2008.

Sethumadhavan S., McDonald R., Desikan R., Burger D., Keckler S., « Design and Implementation of the TRIPS Primary Memory System », *IEEE International Conference on Computer Design (ICCD)*, San Jose, USA, 2006.

Spracklen L. and Abraham S., « Chip Multithreading : Opportunities and Challenges », *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, San Francisco, USA, February, 2005.

Stephens C., Cogswell B., Heinlein J., Palmer G., « Instruction level profiling and evaluation of the IBM RS/6000 », *IEEE International Symposium on Computer Architecture (ISCA)*, Seattle, USA, May, 1990.

STMicroelectronics, Nomadik - Open multimedia platform for next generation mobile devices, Technical report, 2004.

Stolberg H.-J., Berekovic M., Moch S., Friebe L., Kulaczewski M., Flugel S., Dehnhardt A., Pirsch P., « HiBRID-SoC : A Multi-Core SoC Architecture for Multimedia Signal Processing », *Journal of VLSI Signal Processing*, vol. 41, n° 1, p. 9-20, 2005.

Stretch Inc., « S6000 Family Software Configurable Processor », http ://www.stretchinc.com.

Taha T., Wills D. S., « An Instruction Throughput Model of Superscalar Processors », *IEEE Transactions on Computers*, vol. 57, n° 3, p. 389-403, March, 2008.

Tanenbaum A. S., *Structured Computer Organization*, 5 edn, Prentice Hall, 2006.

Tendler J., Dodson S., Fields S., Hung L., Sinharoy B., Power4 System Microarchitecture, Technical report, IBM Server Group, 2001.

Texas Instrument, « High-Performance OMAP Platform : OMAP3430 », http ://www.ti.com.

Texas Instrument, « TMS320VC5541 », http ://www.ti.com.

Tilera Corporation, « The Tile64 architecture », http ://www.tilera.com.

Towner D. W., The Uniform Heterogeneous Multi-threaded Processor Architecture, PhD thesis, University of Bristol, June, 2002.

Tremblay M., Chan J., Chaudhry S., Conigliaro A., Tse S., « The MAJC Architecture : A Synthesis of Parallelism and Scalability », *IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, Austin, USA, 2000.

Tremblay M., Chaudhry S., « A Third-Generation 65nm 16-core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, USA, 2008.

Tullsen D., Eggers S., Levy H., « Simultaneous Multithreading : Maximizing On-Chip Parallelism », *IEEE International Symposium on Computer Architecture (ISCA)*, Santa Margherita Ligure, Italy, June, 1995.

Ubicom, « MASI Processor », http ://www.ubicom.com.

Ungerer T., Robic B., Silc J., « Mutithreaded Processors », *The Computer Journal*, vol. 45, n° 3, p. 320-348, 2002.

Vahey M., Granacki J., Lewins L., Davidoff D., Draper J., Steele C., Groves G., Kramer M., LaCoss J., Prager K., Kulp J., Channell C., « MONARCH : A First Generation Polymorphic Computing Processor », *High performance Embedded Computing*, Lexington, USA, 2006.

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

Vangali S., Howard J., Ruhl G., Dighe S., Wilson H., Tschanz J., Finan D., Iyer P., Singh A., Jacob T., Jain S., Venkataraman S., Y. Hoskote a. N. B., « An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, USA, 2007.

Ventroux N., *« Contrôle en ligne des systèmes multiprocesseurs hétérgoènes embrqués : élaboration et validation d'une architecture »*, Master's thesis, Université de Rennes 1 / CEA LIST, 2006.

Wall D., « Limits of instruction-level parallelism », *ACM International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, USA, 1991.

Wolf W., « The Future of Multiprocessor System-on-Chips », *IEEE/ACM Design Automation Conference (DAC)*, San Diego, USA, June, 2004.

Yoshida Y., Kamei T., K.Hayase, Shibahara S., Nishii O., Hattori T., Hasegawa A., Takada M., Irie N., Uchiyama K., Odaka T., Takada K., Kimura K., Kasahara H., « A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption », *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco Marriott, USA, 2007.

Article reçu le 30 mars 2009
Accepté après révisions le 21 janvier 2010

Nicolas Ventroux *est ingénieur chercheur au laboratoire Calcul Embarqué du CEA LIST. Il est titulaire d'un diplôme d'ingénieur INSA Rennes option électronique et informatique industrielle (2003) et d'un doctorat en électronique de l'université de Rennes 1 (2006). Ses travaux de recherche portent sur la modélisation et la conception d'architectures multiprocesseurs, sur les architectures pour les systèmes de vision, sur la fiabilité et la reconfigurabilité des architectures. Plus particulièrement, il a proposé une solution de contrôle en ligne matérielle de multiples ressources de calcul sous des contraintes de consommation d'énergie et de temps-réel.*

Raphaël David *a reçu en 2003 le titre de docteur en traitement du signal et télécommunications de l'Université de Rennes I (France), pour avoir conçu le processeur reconfigurable DART. Il est aujourd'hui ingénieur de recherche au CEA LIST. Il a intégré cet organisme de recherche suite à un séjour post-doctoral l'ayant conduit à étudier l'efficacité énergétique des architectures reconfigurables. Ses travaux de recherche portent aujourd'hui sur l'exploration de nouveaux modèles d'exécution pour les architectures parallèles et il est en charge de la thématique* architectures multicœurs *au sein du Laboratoire Calcul Embarqué. Ses thématiques de recherche couvrent également le domaine des processeurs reconfigurables et de la conception faible consommation.*