THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par *Jonathan Barre*
Le *15/12/2008*

Titre : *Architectures*
multi-flots simultanés pour le temps-réel strict

JURY
Jean-Paul Bodeveix, Professeur Toulouse 3 (Président du jury)
Pascal Sainrat, Professeur Toulouse 3 (Directeur de thèse)
Nathalie Drach-Temam, Professeur Paris 6 (Rapporteur)
Frédéric Pétrot, Professeur TIMA Grenoble (Rapporteur)

Ecole doctorale : *EDMITT*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse*
Directeur(s) de Thèse : *Pascal Sainrat*
Rapporteurs : *Nathalie Drach-Temam, Frédéric Pétrot*

Table des matières

1 Introduction

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Autrement dit le système ne doit pas seulement délivrer des résultats exacts, il doit également les délivrer dans des délais imposés.

Les systèmes informatiques temps réel sont présents dans de nombreux secteurs d'activités : dans l'industrie de production par exemple, au travers des systèmes de contrôle de procédé (usines, centrales nucléaires), dans les salles de marché au travers du traitement des données boursières en temps réel, dans l'aéronautique au travers des systèmes de pilotage embarqués (avions, satellites), ou encore dans le secteur de la nouvelle économie au travers du besoin, toujours croissant, du traitement et de l'acheminement de l'information (vidéo, données, pilotage à distance, réalité virtuelle, etc.).

Lorsque les contraintes temporelles peuvent ne pas être respectées ponctuellement, on parle de temps-réel souple. Par opposition, dans les systèmes temps-réel stricts, une contrainte transgressée peut entraîner une défaillance du système. Pour garantir le respect des contraintes temporelles, il est nécessaire que les différents services et algorithmes utilisés s'exécutent en temps borné. Un système d'exploitation temps réel

doit ainsi être conçu de manière à ce que les services qu'il propose répondent en un temps borné ; les différents enchaînements possibles des traitements doivent assurer que chacun d'eux ne dépasse pas ses limites temporelles (ou alors rarement si le système est temps-réel souple).

Dans un système temps-réel strict, la grande importance du respect des contraintes temporelles oblige à avoir une estimation sûre des temps d'exécution de manière à éviter tout dépassement d'échéance. C'est pourquoi beaucoup de recherches ont été faites sur le calcul de temps d'exécution pire cas de manière à pouvoir présenter des temps totalement sûrs quelque soit l'exécution de la tâche concernée.

Pour le calcul de WCET, les particularités de l'architecture d'exécution (par exemple, caches, prédiction de branchement, pipeline) doivent être prises en compte pour obtenir un WCET correct. On observe que plus des éléments architecturaux augmentent les performances, plus leur prise en compte dans le calcul du WCET est complexe à cause du manque de prédictibilité de tels éléments. Or, devant la complexité grandissante des systèmes temps-réel, la nécessité d'architectures haute performance se justifie de plus en plus.

Dans ce recueil, nous nous préoccupons du calcul de WCET pour des programmes exécutés sur des systèmes temps-réel strict. Notre travail consistera à faciliter le calcul de WCET sur une architecture *Simultaneous MultiThreading* (SMT). C'est une architecture parallèle qui permet l'exécution simultanée de plusieurs fils d'exécution sur un même cœur mais, comme les autres architectures parallèles, la complexité liée à la co-exécution de plusieurs tâches rend le calcul de WCET très difficilement faisable.

Notre but est de développer de nouveaux mécanismes de prise en compte des particularités des architectures SMT pour le calcul du WCET et également adapter ces architectures pour les rendre plus prédictibles. Nous pensons en effet que des modifications architecturales liées à de nouvelles méthodes de modélisation vont nous permettre d'arriver à nos fins.

Dans ce document, nous commençons d'abord par parler du calcul de WCET. Nous décrivons différentes méthodes existantes de calcul de WCET, nous détaillons en particulier les différentes étapes de ce calcul. Nous concluons ensuite en comparant les

différentes méthodes et en expliquant pourquoi nous décidons de nous servir d'une certaine méthode (la méthode appelée IPET).

Dans la section 3, nous présentons les caractéristiques des architectures SMT ainsi que des données sur les performances. Viennent ensuite une série d'exemples de processeurs SMT du commerce, leur microarchitecture est présentée et les performances sont également décrites. On insiste ensuite sur les points des architectures SMT qui posent problème à la modélisation pour le calcul de WCET, en particulier les sources d'imprédictibilité, telles les politiques de distribution des ressources internes et leurs politiques d'ordonnancement associées. En particulier, nous exhibons le fait qu'il est préférable d'adapter les politiques de distribution et d'ordonnancement pour rendre l'architecture SMT plus prévisible et faciliter le calcul de WCET.

La section 4 décrit notre architecture SMT prédictible permettant l'exécution déterministe de plusieurs threads critiques en parallèle sans interférences entre les threads. Sur cette architecture, prévue pour exécuter jusqu'à 4 threads en parallèle, des threads non critiques peuvent s'exécuter si il y a moins de 4 threads critiques. Nous devons assurer une prédictibilité maximum pour les threads critiques, sans oublier de garantir des performances correctes pour les non critiques. Nous décrivons ensuite une nouvelle méthode de prise en compte des partages de ressources internes dans un processeur (par exemple les places dans les files de stockage et les unités fonctionnelles) permettant de calculer des WCET plus précis sur notre architecture.

La méthodologie de simulation est ensuite présentée dans la section suivante. On y détaille le pipeline de notre architecture ainsi que les données sur les files de stockage internes et les unités fonctionnelles par exemple. Nous décrivons ensuite notre simulateur basé sur l'outil GLISS [64] ainsi que les programmes de test utilisés pour nos expérimentations.

Les performances sont décrites dans la section 6. On vérifie d'abord que l'exécution des threads critiques ne dépend pas de la nature des threads co-exécutés et est donc prédictible, le temps d'exécution dépend seulement du nombre de threads critiques co-exécutés. On évalue également les performances des threads non critiques qui doivent être acceptables. On observe ensuite l'évolution des performances quand des paramètres de l'architecture sont modifiés (quantités des ressources internes, largeur

du pipeline). Un exemple simple de partage des caches assurant prédictibilité pour les threads critiques est présenté. Des WCET sont ensuite calculés pour cette architecture en fonction du nombre de thread critiques co-exécutés, seul élément dont dépend l'exécution des threads critiques.

On passe ensuite à la section 7, où nous faisons le bilan de notre architecture prédictible. Nous énumérons ensuite les perspectives d'utilisation de notre architecture, en particulier pour les algorithmes d'ordonnancement de tâches temps-réel, pour finir par la conclusion de notre travail dans la dernière section.

Contribution

Ainsi, nous avons déterminé dans le chapitre 3 les problèmes des SMT actuels et notre contribution est exposée dans les chapitres 4, 6 et 7.

Elle consiste en une architecture SMT prédictible acceptant plusieurs threads critiques, ce qui n'a jamais été proposé jusqu'à présent.

2 Calcul de temps d'exécution pire-cas

2.1 Introduction

Les systèmes temps-réel regroupent les applications dont l'exécution correcte ne dépend pas seulement de l'exactitude des résultats mais également du temps d'exécution, une telle application devant s'efforcer de respecter des échéances données.

On peut classer ces systèmes en deux familles suivant leur tolérance quant au respect des échéances. Si certaines échéances peuvent être ratées sans compromettre le bon fonctionnement du système, même s'il est légèrement dégradé, on parle de *temps-réel souple*. Si le respect des échéances est impératif à la bonne marche du système, on se trouve dans un cadre *temps-réel strict*.

Le temps réel strict ne tolère aucun dépassement des contraintes spécifiées, souvent de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques. Des exemples classiques de systèmes temps-réel souple sont les services de diffusion de media (son, vidéo) par internet ou autre réseau. Si une vidéo à 25 images par seconde

doit être transmise, il faudrait dans le cas idéal que l'on reçoive bien 25 images chaque seconde. Malheureusement, sur un réseau partagé comme internet, des retards voire des pertes peuvent survenir. On se retrouve donc avec des « sauts » dans la lecture, ce qui entraine une gêne pour l'utilisateur mais n'est pas considéré comme critique car la lecture et la compréhension du contenu sont toujours possibles. Dans ce cas le non respect des échéances entraine une perte de qualité mais n'empêche pas le bon fonctionnement du système.

Les systèmes temps-réel strict, quant à eux, ont justement de très bonnes raisons d'être stricts. Il s'agit principalement de systèmes commandant des processus à criticité élevée qui doivent faire la bonne action en un temps bien déterminé. On peut citer par exemple les systèmes d'antiblocage de freins (ABS) ou de commande d'airbag présents sur les automobiles récentes, les systèmes actionnant les commandes de vol d'un avion ou le contrôle d'une centrale nucléaire.

De tels systèmes sont dits critiques car la moindre échéance non respectée (gouverne d'un avion tournant trop tard, réaction à la fusion d'un réacteur nucléaire trop tardive) peut entrainer des pertes très importantes, tant en vies humaines que matérielles. Mais les systèmes sont également considérés comme critiques par les industriels dès lors qu'en cas de défaillance, ils font perdre énormément d'argent, ce qui peut aller dans certains cas jusqu'à la fermeture de l'entreprise. Ainsi, la perte d'un satellite a un coût exorbitant, une panne électrique due à un disjoncteur défectueux peut entrainer la fermeture d'une usine durant plusieurs jours, …

Dans un système temps-réel, il y a souvent plusieurs tâches qui doivent toutes pouvoir s'exécuter en respectant leurs contraintes. Chacune des tâches a une date de début, une échéance et souvent une période car les tâches sont cycliques. Il existe de nombreux algorithmes d'ordonnancement temps-réel dont la fiabilité est prouvée. Pour garantir le respect des contraintes propres à chaque tâche, en particulier l'échéance finale, il suffit de pouvoir déterminer avec exactitude le temps d'exécution de la tâche en question, la date de début étant donnée par le système.

Sur une architecture donnée, la durée d'exécution d'un programme dépendant fortement des données en entrée du programme, il faut arriver à trouver le plus long temps d'exécution de la tâche dans le contexte donné ou, à défaut, une borne supérieure

de tous les temps d'exécution possibles en fonction des entrées. Ce temps d'exécution sert ensuite de temps de référence de la tâche pour l'ordonnancement.

Dans la suite, ce plus long temps d'exécution sera appelé *WCET réel* (Worst Case Execution Time, temps d'exécution pire cas) par opposition au *WCET estimé* ou *WCET* obtenu par calcul. Il est bien entendu nécessaire que le WCET estimé soit supérieur ou égal au WCET réel, une sous-estimation conduirait à des ordonnancements invalides avec des tâches ne disposant pas d'assez de temps pour s'exécuter. A l'inverse une surestimation trop forte gonflerait artificiellement le temps de référence de la tâche et l'ordonnanceur pourrait ne pas arriver à agencer les tâches.

Nous allons maintenant décrire les étapes du calcul du WCET d'une tâche et les problèmes pouvant survenir en prenant en compte une architecture SMT.

2.2 Le calcul de WCET

On utilise actuellement deux catégories de méthodes pour calculer des WCET. Les méthodes dynamiques essayent de déterminer le WCET à partir de mesures des temps d'exécution, tandis que les méthodes statiques calculent les temps de morceaux de code, tentent de trouver le pire « scénario » d'exécution de ces morceaux et calculent un WCET à partir de ces deux ensembles d'informations.

2.2.1 Les méthodes dynamiques

Ces techniques ont pour but de trouver le WCET réel. La méthode la plus simple conceptuellement consiste à exécuter un programme avec toutes les configurations d'entrées possibles et à prendre le plus long temps d'exécution obtenu qui est alors, de façon sûre et certaine, le WCET réel du programme considéré.

Le problème est qu'il est bien entendu difficile de pouvoir tester tous les jeux de données possibles en un temps raisonnable, leur nombre pouvant être très grand. Il faut donc sélectionner un ou plusieurs jeu(x) de test parmi tous les jeux de données possibles. Pour cela il y a deux possibilités : soit le programmeur arrive à avoir une idée des données pouvant conduire au WCET (à condition que le fonctionnement du programme soit facilement compréhensible et relativement simple) et il prépare un jeu de test dont le pire temps d'exécution sera alors considéré comme le WCET, soit on

dispose d'un outil capable de générer des jeux de test qui couvrent tous les chemins possibles [28][61].

Une autre technique, l'exécution symbolique, permet d'éviter de chercher des jeux de test adéquats. On initialise les variables dépendant des valeurs en entrée du programme à une valeur « inconnue » et la sémantique des opérations est modifiée pour prendre en compte ces valeurs inconnues. Une addition, par exemple, renvoie une valeur inconnue comme résultat dès que l'un des deux opérandes est inconnu ; si les deux opérandes sont connus, elle renvoie bien entendu le résultat réel. Ainsi ces valeurs inconnues se propagent et, lorsque la condition d'un branchement conditionnel dépend d'une valeur inconnue, les deux branches possibles (branchement pris ou non pris) sont explorées. Au fur et à mesure de l'exécution, le nombre de chemins à explorer peut devenir très grand. Pour essayer d'éliminer certains chemins impossibles, on peut prendre en compte des annotations concernant, par exemple, les bornes de boucles (intervalle des nombres d'itérations), les tests des branchements conditionnels (pronostic sur la direction), ou les paramètres d'un sous-programme (valeurs attendues). L'exécution symbolique ne peut être réalisée directement sur la machine cible du programme à cause des valeurs inconnues. On doit donc la réaliser par simulation logicielle [4][38][39].

L'inconvénient de ces méthodes est que le nombre de chemins à explorer est généralement très grand. La réduction de ce nombre de chemins par des annotations de l'utilisateur pose le problème des risques d'erreurs liées au facteur humain. Nous allons décrire maintenant une autre classe de méthodes permettant de contourner cet écueil : les méthodes statiques.

2.2.2 Les méthodes statiques

Avec les méthodes dynamiques, le WCET est déterminé à partir de l'exécution du programme entier, au contraire des méthodes statiques qui analysent de petits bouts de programmes (« briques élémentaires » faciles à mesurer) et reconstituent le temps d'exécution du programmes complet à partir de l'agencement de ces petits bouts de codes (généralement des blocs de base, c'est-à-dire des séquences d'instructions avec point d'entrée et de sortie uniques) représenté sous forme d'un graphe.

Dans la section suivante, nous allons décrire les différentes étapes du calcul avec de telles méthodes. La première étape est l'analyse de flot où sont déterminés, entre autres, les chemins impossibles et les bornes de boucles à partir du code source ou exécutable. Il faut ensuite prendre en compte les particularités de l'architecture du processeur pour le calcul des temps d'exécution des blocs de base : c'est ce qu'on appelle analyse bas niveau. La dernière étape consiste à reconstituer le WCET du programme complet.

2.3 Analyse de flot

2.3.1 Méthodes de représentation du code

Deux manières de représenter le code d'un programme sont utilisées dans le calcul de WCET : l'arbre syntaxique et le graphe de flot de contrôle (CFG, control flow graph).

L'arbre syntaxique représente la structure syntaxique du programme. Un nœud est étiqueté par la structure algorithmique correspondante : séquence, boucle, structure conditionnelle ("if-then" ou "if-then-else"). Chacun des nœuds est parent des parties de code le constituant : test et corps de boucle par exemple. Les feuilles de l'arbre syntaxique sont constituées de blocs d'instructions ayant un seul point d'entrée et un seul point de sortie. L'arbre syntaxique se basant sur les structures algorithmiques, il est construit à partir du code source du programme. Il est donc en général impossible de retrouver l'arbre syntaxique à partir de l'exécutable seul.

Le CFG, quant à lui, se construit à partir du code bas-niveau du programme, code machine ou code assembleur. Chaque nœud du graphe représente une séquence d'instructions sans branchement et telle qu'aucun branchement du programme ne peut amener à une instruction de la séquence autre que la première. On appelle « bloc de base » une séquence d'instructions ayant un seul point d'entrée et un seul point de sortie. Le graphe représente les liens entre les blocs de base dans le programme. Lorsqu'un bloc se termine par une instruction de branchement conditionnelle, alors deux arêtes permettent d'en sortir, une pour chacune des deux directions possibles

2.3.2 Informations de flot supplémentaires

Représenter le code du programme et mesurer le temps d'exécution de chaque bloc d'instructions n'est pas suffisant pour calculer le WCET d'un programme. Si aucune information au sujet du nombre d'itérations d'une boucle n'est fournie, alors le temps d'exécution pire-cas calculé est infini. Pour garantir un WCET fini et surtout réaliste, les informations supplémentaires suivantes doivent ou peuvent être fournies :

- la borne supérieure du nombre d'itérations de chaque boucle du programme ;
- le nombre d'exécutions de chaque branche d'une structure conditionnelle si la condition de test varie de manière analysable au cours du temps.
- une liste de chemins infaisables. Par exemple, deux tests de branchements conditionnels impliquant une même variable peuvent s'exclure l'un l'autre.

Nous avons décrit ci-dessus les graphes à partir desquels le calcul de WCET est réalisé. Dans la suite de cette partie, nous présentons les différents éléments de l'architecture d'un ordinateur qui ont un impact sur le temps d'exécution et comment ils peuvent être modélisés pour le calcul de WCET.

2.4 L'analyse de bas niveau

L'analyse de bas niveau a pour but de calculer le temps d'exécution de parties de chemins, typiquement de blocs de base.

Tant que les processeurs cibles étaient très simples, le temps d'exécution d'un bloc de base pouvait être calculé de façon triviale en ajoutant les durées des instructions qui le composaient indiquées par le constructeur. Depuis, les choses se sont un peu compliquées car de nouveaux mécanismes ont été introduits pour augmenter les performances: les mémoires caches (d'instructions et de données), la prédiction de branchement, l'exécution pipelinée, dans le désordre et enfin spéculative..

De manière générale, pour prendre en compte l'impact de l'utilisation de ces éléments sur le temps d'exécution, la méthode la plus simple est de considérer le pire des scénarios. Or, comme nous l'avons précisé dans les sections précédentes, il n'est pas bénéfique d'obtenir un WCET trop surestimé. Pour cette raison, des travaux ont été

menés pour affiner l'impact de chaque élément ou mécanisme de l'architecture sur le temps d'exécution.

Nous allons maintenant décrire ces travaux sur les principales structures. Nous verrons d'abord la prise en compte du pipeline, puis les caches et enfin la prédiction de branchement.

2.4.1 Le pipeline d'exécution

Avec l'arrivée de processeurs comportant un pipeline il a fallu prendre en compte le recouvrement entre les différentes phases de traitement des instructions. Des techniques basées sur une représentation de l'état du pipeline par une table de réservation ont alors été proposées [35]. Toutefois, cette représentation n'est pas suffisante pour exprimer l'exécution d'une séquence d'instructions dans un pipeline superscalaire, a fortiori s'il exécute les instructions dans le désordre. Or ce type d'architecture est parfois utilisé, y compris pour des applications critiques, dans des systèmes embarqués gourmands en puissance de calcul.

Par ailleurs, les architectures avancées, telles que celles que nous venons d'évoquer, font que le temps d'exécution d'un bloc de base dépend de l'état du pipeline (disponibilité des ressources) au début de l'exécution, c'est-à-dire de ce qui a été exécuté avant le bloc.

Lorsque l'on considère un processeur ayant les bonnes propriétés (exécution dans l'ordre, pas d'unité fonctionnelle à latence longue, etc), il peut être suffisant de n'examiner que les blocs de base prédécesseurs directs du bloc considéré et de calculer un coût d'exécution du bloc pour chacun de ses prédécesseurs possibles (le coût d'exécution diffère du temps d'exécution en ce qu'il tient compte du recouvrement entre blocs successifs dans le pipeline). Malheureusement, ce n'est plus vrai dès que l'on considère une architecture plus complexe : un bloc de base peut subir un effet temporel de la part d'un bloc de base distant (c'est-à-dire exécuté bien avant lui).

Ce phénomène a été mis en évidence par Jakob Engblom [18] et est référencé sous le terme d' « effet long ».L'auteur a également démontré que la distance entre un bloc subissant un effet et celui qui le lui inflige peut être infiniment grande. Ainsi, pour être certain de calculer toutes les valeurs possibles du coût d'exécution d'un bloc de base, il

faudrait explorer tous les chemins possibles menant à ce bloc, ce qui, bien évidemment, va à l'encontre des objectifs de l'analyse statique.

C'est pourquoi, deux types d'approches ont été proposés pour estimer le coût pire cas d'un bloc sans énumérer tous les chemins préfixes possibles. La première solution consiste à évaluer, par interprétation abstraite, tous les états possibles du pipeline au début de l'exécution d'un bloc de base (outil commercial AbsInt [63]). Pour chacun de ces états, on peut calculer un coût d'exécution du bloc et la valeur maximale parmi tous les coûts possibles est retenue pour calculer le WCET de l'application. L'autre technique consiste à exprimer l'exécution d'un bloc par un graphe d'exécution dans lequel chaque nœud correspond à une étape du traitement d'une instruction du bloc (typiquement son passage par un étage du pipeline) et les arcs définissent les contraintes de précédences entre nœuds. Ce type de représentation a été proposé par Li *et al.* [34] avec une technique d'analyse consistant à calculer les dates relatives auxquelles les différents nœuds peuvent être exécutés en posant des hypothèses pessimistes sur l'historique (contexte d'exécution pire cas).

Un autre algorithme de résolution permettant d'affiner les coûts estimés en considérant l'impact des dates de libération de chacune des ressources nécessaires au bloc a été proposé récemment [50]. Les ressources considérées incluent les ressources physiques (unités fonctionnelles, étages du pipeline, entrées dans des files d'instructions, …) mais aussi les ressources logiques que sont les valeurs des registres dont dépendent les instructions du bloc de base. Un graphe d'exécution est construit pour représenter l'exécution d'une séquence constituée d'un bloc de base dont on peut calculer le temps d'exécution et un de ses prédécesseurs dans le CFG. Comme illustré sur la Figure 1, la date de démarrage et de terminaison de chaque nœud est calculée en fonction des dates auxquelles ces différentes ressources sont disponibles. Sur ce schéma, huit ressources sont considérées : les ressources iw_0 à iw_3 représentent des entrées de la fenêtre d'instructions, ALU et MEM sont associées à des unités fonctionnelles, CM représente un étage du pipeline et r10 est la valeur d'un registre produite par une instruction antérieure à la séquence.

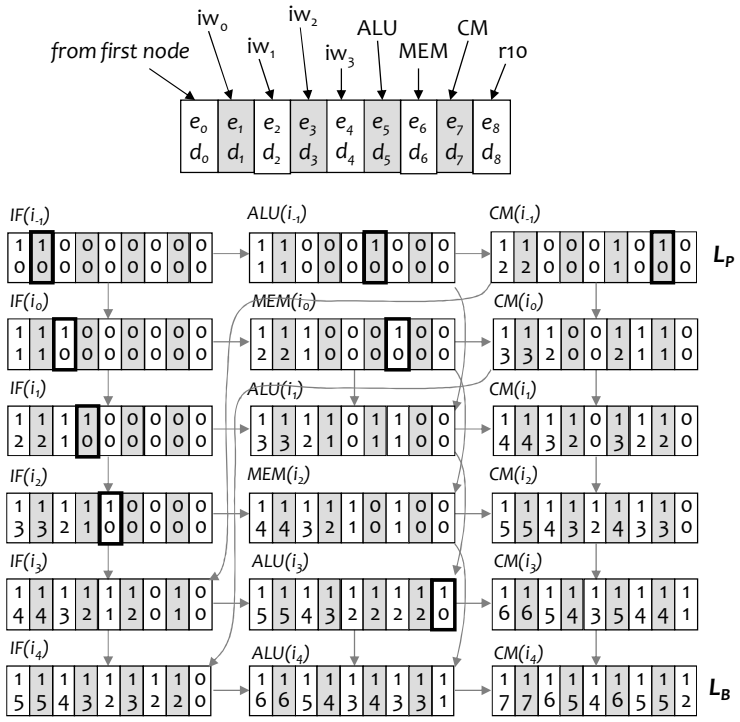

Figure 1. Calcul du coût d'un bloc de base à partir d'un graphe d'exécution

A partir des dates de terminaison des nœuds, on peut borner le coût du bloc de base, c'est-à-dire le temps entre la terminaison du dernier nœud du bloc et le dernier nœud du bloc précédent. Comme montré dans [50], on peut écrire :

$$\Delta(L_B, L_P) \leq \max_{r \in \mathcal{R}}(\delta^r(L_B, L_P))$$

avec :

$$\delta^r(L_B, L_P) = e^r_{L_B}.\left(d^r_{L_B} - e^r_{L_P}.d^r_{L_P} - \left(1 - e^r_{L_P}\right).\left(d^\lambda_{L_P} + \alpha_r\right)\right)$$

où :

- e^r_n indique si la date de démarrage du nœud *n* dépend de la date de libération de la ressource *r*
- d^r_n est le délai minimum entre le démarrage du nœud *N* et la libération de la ressource *r* (lorsque $e^r_n == 1$)
- L_P et L_B sont les derniers nœuds du bloc de base considéré et de son prédécesseur dans la séquence analysée
- $\mathcal{R}$ est l'ensemble des ressources

- λ représente la dernière ressource utilisée dans le pipeline (le dernier étage du pipeline)
- $\alpha_r$est le délai minimum entre la libération de la ressource r et la libération de la dernière ressource λ

Un calcul spécifique permet de prendre en compte l'exécution non ordonnée en estimant le délai maximum qui peut être engendré pour un nœud par des nœuds utilisant la même unité fonctionnelle sans que l'ordre entre les nœuds ne soit spécifié.

C'est cette méthode que nous avons utilisée pour calculer les WCET rapportés dans ce document.

2.4.2 Les caches

Les caches d'instructions et de données sont les éléments dont l'utilisation a le plus d'impact sur le temps d'exécution. Lorsque l'information recherchée (instruction ou donnée) n'est pas contenue dans un niveau de cache, elle doit être chargée depuis le niveau supérieur, voire la mémoire principale. Ce temps de chargement peut être important et il est donc nécessaire de modéliser les accès au cache pour calculer l'impact sur le WCET. Pour cette modélisation, deux principales méthodes sont utilisées.

La première méthode classe les accès au cache d'instructions de la manière suivante :

- une référence mémoire qui sera toujours un succès (resp. un échec) dans le cache est classée "always hit" (resp. "always miss").
- s'il n'est pas possible de se prononcer sur le succès ou l'échec d'un accès au cache, il est classé "conflict".
- selon que l'on distingue (ou non) la première occurrence d'une instruction (correspondant à la première itération d'une boucle), une classe supplémentaire peut être ajoutée pour spécifier que le premier accès est un échec mais les suivants sont des succès (ou l'inverse).

Une telle classification peut être obtenue par simulation statique [24][42][53] ou interprétation abstraite [1][54].

Le deuxième type de méthode consiste à décrire l'évolution du contenu du cache. Li *et al.* [33] ont introduit les CCG (Cache Conflict Graph). Pour expliquer le

fonctionnement de ces graphes, il est nécessaire de comprendre que les instructions d'un même bloc de base n'accèdent pas toutes à la même ligne de cache. On divise d'abord les instructions d'un bloc de base en "l-blocks" regroupant les instructions du bloc situées dans une même ligne de cache. Deux *l-blocks* de deux blocs de base différents peuvent très bien accéder à une même ligne de cache. Un CCG est généré pour chaque ligne de cache. Il permet de décrire l'ordre d'exécution entre les l-blocks qui accèdent à cette ligne. Ce graphe est ensuite décrit par un programme linéaire en nombre entiers. La résolution de ce système permet d'obtenir le nombre maximum d'échecs d'accès au cache.

Les adresses des instructions étant statiques, il est plus aisé de modéliser le comportement du cache d'instructions. En effet, avec le cache de données, l'adresse d'une donnée peut changer d'une exécution d'une instruction à la suivante car cette adresse peut être calculée par d'autres instructions du programme (quand on parcourt une liste chaînée par exemple). Il est donc nécessaire d'analyser les adresses avant de modéliser les accès. Cela peut se faire par analyse de flot, par interprétation abstraite [19] ou par simulation statique [27]. Une fois les adresses déterminées (lorsque c'est possible), le cache de données peut être modélisé par les CCG. Les CCG sont de forme différente de ceux générés pour un cache d'instructions car deux états sont générés pour chaque donnée : elle peut être lue ou écrite en mémoire [33].

Pour contourner la difficulté de modélisation des caches, des stratégies de gel de cache ont été proposées. Ce mécanisme permet de fixer le contenu du cache, interdisant à toute ligne déjà allouée pour une adresse donnée d'être ré-attribuée à une adresse différente (élimine les conflits dans le cache). Le cache peut être gelé dynamiquement pendant l'exécution de quelques instructions [58] ou tout au long de l'exécution d'une tâche [46][47]. Le choix des instructions (ou données) à charger avant le gel est effectué par un algorithme tenant compte des propriétés du programme et de son code. Le principal défaut de cette méthode est l'augmentation du temps d'exécution dû au fait que les instructions ou les données qui ne sont pas dans le cache sont chargées depuis la mémoire principale à chaque fois car elles ne peuvent pas profiter du cache à cause du gel. Par contre, elle rend les latences mémoire déterministes et le WCET obtenu n'est pas surestimé.

2.4.3 La prédiction de branchement

L’effet de la prédiction de branchement est souvent inclus dans le temps d'exécution des blocs [19]. C'est le cas par exemple des méthodes qui utilisent l'interprétation abstraite pour modéliser le pipeline. Dans ce cas, l'état du pipeline est calculé en début et fin de bloc. Cela permet de voir l’influence que peut avoir un branchement sur les blocs suivants, ainsi que les effets des blocs précédents sur le bloc courant. Si le temps d'exécution des blocs est calculé au pire-cas, le branchement est toujours pris en compte comme mal prédit. Considérer uniquement des mauvaises prédictions n’est néanmoins pas optimal et augmente le WCET de façon inutile. Des méthodes plus élaborées analysent et classent les branchements (en fonction du type de boucle ou de structure conditionnelle) et permettent de quantifier plus finement le nombre de mauvaises prédictions [8].

2.5 Méthodes de calcul

Dans cette partie, nous décrivons comment les méthodes d'analyse statique permettent de calculer le WCET. Ces méthodes se basent sur tout ce que nous avons vu plus haut, à savoir la représentation du code, les informations sur le flot, l'analyse bas niveau et les temps d'exécution des blocs d'instructions.

Après avoir présenté les méthodes statiques, nous expliquons comment le temps d'exécution pire-cas peut être calculé par des méthodes qui combinent calcul statique et dynamique.

2.5.1 Méthodes purement statiques

Deux méthodes ("tree-based" et "path-based") calculent le temps d'exécution pire-cas par lecture du graphe représentant le code du programme (respectivement l'arbre syntaxique et le graphe de flot de contrôle).

Le parcours de l'arbre syntaxique s'effectue des feuilles à la racine [48]. A chaque nœud est associée une fonction qui calcule le temps d'exécution pire-cas des sous-arbres qui le composent. A un nœud "séquence" est associée la somme des temps d'exécution des sous-arbres. Pour un branchement conditionnel de type "if-then-else", les temps d'exécution des deux branches (then et else) sont calculés et comparés : le temps

d'exécution maximum est additionné au temps de résolution du test de la condition pour constituer le temps d'exécution de la structure. Enfin, pour une boucle, le nombre d'itérations maximum est utilisé : le temps d'exécution du corps de boucle est comptabilisé *n* fois où *n* représente la borne supérieure du nombre d'itérations. Le test de boucle est, quant à lui, comptabilisé *n*+1 fois car il est exécuté à la sortie de boucle. Le résultat obtenu par cette méthode est un WCET associé à chaque structure algorithmique (nœud de l'arbre). Cet arbre dont les nœuds sont valués par leur WCET est nommé "timing-tree".

La méthode "path-based" utilise la représentation du code du programme sous forme de graphe de flot de contrôle [24][53]. Comme nous l'avons précisé, à chaque bloc de base est associé un temps d'exécution pire-cas. En considérant le graphe de contrôle comme un graphe valué par les temps d'exécution des blocs de base, une recherche du plus long chemin est possible. Les informations supplémentaires telles que le nombre d'itérations maximum des boucles sont alors nécessaires pour limiter le nombre d'occurrences d'un sommet ou d'un arc le long du chemin le plus long. Les résultats obtenus ne correspondant pas toujours à un chemin d'exécution possible, il est nécessaire de les vérifier. Lorsque le chemin trouvé est impossible, on relance de nouveau la recherche en excluant ce chemin.

La méthode IPET (Implicit Path Enumeration Technique) se base sur le graphe de flot de contrôle à partir duquel on peut extraire des chemins d'exécution. Lorsque l'on définit ces chemins d'instructions en spécifiant la liste des blocs exécutés dans l'ordre, on obtient des chemins dits explicites. La méthode IPET, quant à elle, manipule des chemins d'exécution implicites décrits seulement par le nombre d'occurrences de chaque bloc et arête du graphe. Un chemin implicite ne contient aucune information sur l'ordre d'exécution des blocs de base.

Le but de la méthode IPET est de déterminer le chemin implicite le plus long [32]. On construit un système d'équations linéaires en nombres entiers représentant les contraintes qui lient les nombres d'occurrences des blocs et arêtes du graphe. Chaque bloc peut s'exécuter autant de fois que les arêtes qui permettent d'y accéder et autant de fois que les arêtes qui en sortent. Le nombre d'exécutions des blocs qui constituent le corps de boucle est limité par la borne du nombre d'itérations de la boucle.

Le temps d'exécution du programme étant égal à la somme des temps d'exécution des blocs de base pondérés par leur nombre d'exécutions, le WCET est obtenu en maximisant cette expression tout en respectant les contraintes linéaires. Le résultat du calcul est alors le chemin implicite pire-cas exprimé par la liste des nombres d'exécutions des blocs et arêtes et le WCET est le temps d'exécution correspondant à ce chemin pire-cas.

2.5.2 Méthodes intégrant des mesures

Certains travaux utilisent des mesures plus approfondies pour affiner le calcul par méthode statique. Dans [29], les auteurs s'appuient sur l'arbre syntaxique pour effectuer des mesures de temps d'exécution des sous-arbres utilisées pour calculer les bornes du WCET. Dans [60], le calcul est principalement basé sur des mesures du temps d'exécution des programmes. L'analyse statique de partie de code est utilisée comme support pour calculer le prochain chemin à tester.

Dans [5][6], les probabilités sont utilisées pour affiner le WCET estimé. La première étape de cette méthode consiste à définir des profils d'exécution des blocs de base par de nombreuses mesures. Ces profils sont ensuite combinés en utilisant les probabilités pour obtenir un profil de chemin. L'étape suivante permet de prendre en compte les dépendances entre les profils d'exécution des blocs de base. Un WCET probabiliste est finalement calculé. Le but de cette méthode n'est pas d'obtenir une borne supérieure du WCET, mais d'amener le principe de marges de calcul appliqué par certains industriels. En effet, pour être certain de prendre en compte un WCET suffisamment large, une certaine marge est souvent ajoutée au WCET estimé. En incluant des probabilités dans le calcul de WCET ces marges sont affinées. Le WCET obtenu n'est pas garanti mais il peut être utilisé dans des systèmes temps-réel souples où les échéances peuvent parfois être dépassées.

Les deux dernières méthodes que nous évoquons dans cette section intègrent davantage de mesures en modifiant la granularité de l'étude. En effet, l'élément de base de l'analyse n'est plus le bloc de base. Dans [62], des segments de programme sont constitués d'une séquence d'instructions qui ne dépendent pas des valeurs des données en entrée du programme. Pour calculer le WCET global, la méthode IPET est utilisée en se basant sur les segments à la place des blocs de base. Dans [44], on prend en compte

des ensembles d'instructions différents : les "blocs de mesure" sont définis en fonction du nombre de chemins à explorer qu'ils contiennent. Le but est de pouvoir mesurer de façon déterministe le temps d'exécution de ces blocs. Les blocs de mesure sont spécifiques au matériel et le WCET global combine ces mesures de manière spécifique à chaque processeur.

Toutes ces méthodes ont pour but de limiter la surestimation du calcul du WCET par méthode statique. Surestimation qui est due à la difficulté de modéliser, par analyse statique, les comportements dynamiques des mécanismes de l'architecture du processeur.

2.6 Conclusion

Nous avons vu l'importance du calcul de WCET pour les applications temps-réel strict. Ce qui est prépondérant dans le calcul de WCET c'est de pouvoir fournir des WCET sûrs qui ne soient pas trop surestimés et dont le calcul ne demande pas des quantités démesurées de ressources ou de temps.

Nous avons d'abord décrit les méthodes dynamiques qui essayent de trouver, à partir de l'exécution réelle ou symbolique du programme, le temps d'exécution pire cas en parcourant des jeux de données essayant d'être les plus exhaustifs possibles. Ces méthodes nécessitent beaucoup de temps de calcul, ce temps peut être réduit en affinant les jeux de test grâce des annotations. Malheureusement, cela augmente le risque d'erreur humaine due à des annotations erronées.

Viennent ensuite les méthodes statiques, la particularité de ces méthodes est qu'elles ne demandent de considérer l'exécution que de petits bouts de code, les blocs de base. Tout le calcul du WCET se fait ensuite de manière formelle sans avoir besoin d'exécuter le programme, cela permet ainsi de calculer un WCET pour n'importe quelle architecture sans même disposer de machine de l'architecture en question.

Ces méthodes comportent 3 étapes. D'abord l'analyse de flot, où des informations sont recueillies à propos du flot d'exécution (bornes de boucles, conditions de tests). Les particularités de l'architecture cible (où doit s'exécuter le programme dont on cherche le WCET) sont ensuite prises en compte lors de l'analyse bas niveau, les principaux éléments pris en compte sont les latences des instructions, le pipeline, les

caches et la prédiction de branchement, à la fin de cette étape on obtient les temps d'exécution des blocs. A partir de ces temps de base, la troisième étape permet de calculer le WCET.

Pour cette troisième étape, plusieurs méthodes sont possibles. La méthode "tree-based" se base uniquement sur le code source et non le code exécutable, cela peut poser des problèmes suivant les méthodes de compilation. La méthode "path-based" cherche le plus long chemin d'exécution, malheureusement rien ne permet de garantir qu'on trouve un chemin valide, il faut vérifier après coup et éventuellement recommencer l'opération si le chemin trouvé est impossible. La méthode IPET, quant à elle, permet de ne manipuler que des chemins possibles, au contraire de la méthode "path-based", et travaille sur les blocs de base et non le code source comme la méthode "tree-based". Ces avantages font que la méthode IPET est couramment utilisée pour le calcul de WCET. C'est d'ailleurs cette méthode que nous utilisons par la suite pour calculer les WCET de nos programmes.

3 Les processeurs multi-flots simultanés et le temps-réel strict

Dans cette partie, nous allons décrire les origines et les caractéristiques de l'architecture à multi-flots simultanés (*Simultaneous Multithreading*, on parlera par la suite de processeurs ou d'architecture *SMT*). Les processeurs SMT ont besoin de mécanismes spécifiques pour permettre l'exécution simultanée de plusieurs fils d'exécution (programme ou thread logiciel, que nous appellerons par la suite *thread*). Ces mécanismes consistent principalement à partager ou dupliquer les ressources du cœur.

Nous décrirons ensuite les enjeux et techniques associées aux applications temps réel strict en considérant l'exécution de tâches temps-réel strict sur une architecture SMT. Nous passerons en revue les problèmes pouvant apparaître sur une telle configuration. Nous verrons que ces problèmes sont principalement liés aux politiques de partage et d'accès aux ressources du cœur SMT, ces politiques pouvant être améliorées pour assurer une certaine prévisibilité de l'exécution.

3.1 Les processeurs SMT

Cette section nous permettra de présenter l'architecture SMT et ses caractéristiques. Des exemples de processeurs SMT existants seront donnés en guise d'illustration. Nous préciserons également le cadre de notre recherche en indiquant les points importants de l'architecture SMT que nous traiterons dans notre étude.

3.1.1 L'architecture SMT

L'exécution SMT a été introduite par Tullsen *et al.* [57] dans le but de maximiser l'utilisation des ressources internes d'un cœur d'exécution en permettant le traitement au même cycle d'instructions provenant de *threads* différents.

Elle a été imaginée pour essayer de combiner les avantages de deux types d'architecture, l'architecture superscalaire et l'architecture multithread. Pour illustrer ceci, nous nous baserons sur la Figure 2 qui présente des exemples simplifiés de séquences d'exécution pour les architectures superscalaire, multithread et SMT. Chaque ligne représente l'occupation des unités d'exécution pour un cycle donné (chaque case correspondant à une unité). Si le processeur trouve une instruction pour occuper une unité, la case en question est colorée, sinon, l'unité est libre et la case est vide.

Comme illustré sur la Figure 2, on distingue ainsi facilement deux types de sous-utilisations de ressources [17]. La *sous-utilisation horizontale* se produit quand, au cours d'un cycle, certaines unités sont alloués mais pas toutes. Cette situation est caractéristique d'un degré relativement faible de parallélisme d'instructions dans le *thread* considéré. On observe une *sous-utilisation verticale* quand toutes les unités sont inoccupées pendant un ou plusieurs cycles consécutifs, une telle situation se produisant la plupart du temps quand une opération à latence longue (accès mémoire typiquement) bloque l'émission des instructions en amont du même *thread*.

Dans un processeur superscalaire, plusieurs instructions d'un même thread peuvent être traitées en même temps à chaque étage du pipeline (tant que les dépendances fonctionnelles entre instructions sont respectées), le processeur s'efforçant d'en trouver le plus possible à émettre en parallèle. Le nombre maximum d'instructions pouvant être émises lors d'un même cycle est appelé le degré de superscalarité de l'architecture.

Malheureusement, sur ce type d'architecture, les deux types de sous-utilisation décrits ci-dessus peuvent se produire fréquemment.

En partant de ce constat, Tullsen *et al.* ont introduit la technique du multi-flots simultanés (Simultaneous Multithreading, que nous abrégerons par SMT dans la suite) où plusieurs *threads* sont exécutés en même temps [57]. Ce que nous appelons *thread* est soit un thread logiciel d'un programme, soit un programme mono-thread.

Pour limiter la sous-utilisation horizontale, l'idée est d'améliorer le point fort du superscalaire, à savoir l'émission simultanée de plusieurs instructions. Pour augmenter la capacité d'émission des instructions à chaque étage, on s'affranchit de la limitation du parallélisme d'instructions intra-thread observée sur un superscalaire en ajoutant la possibilité de lancer des instructions d'autres *threads* qui s'exécuteront ainsi en parallèle dans le processeur.

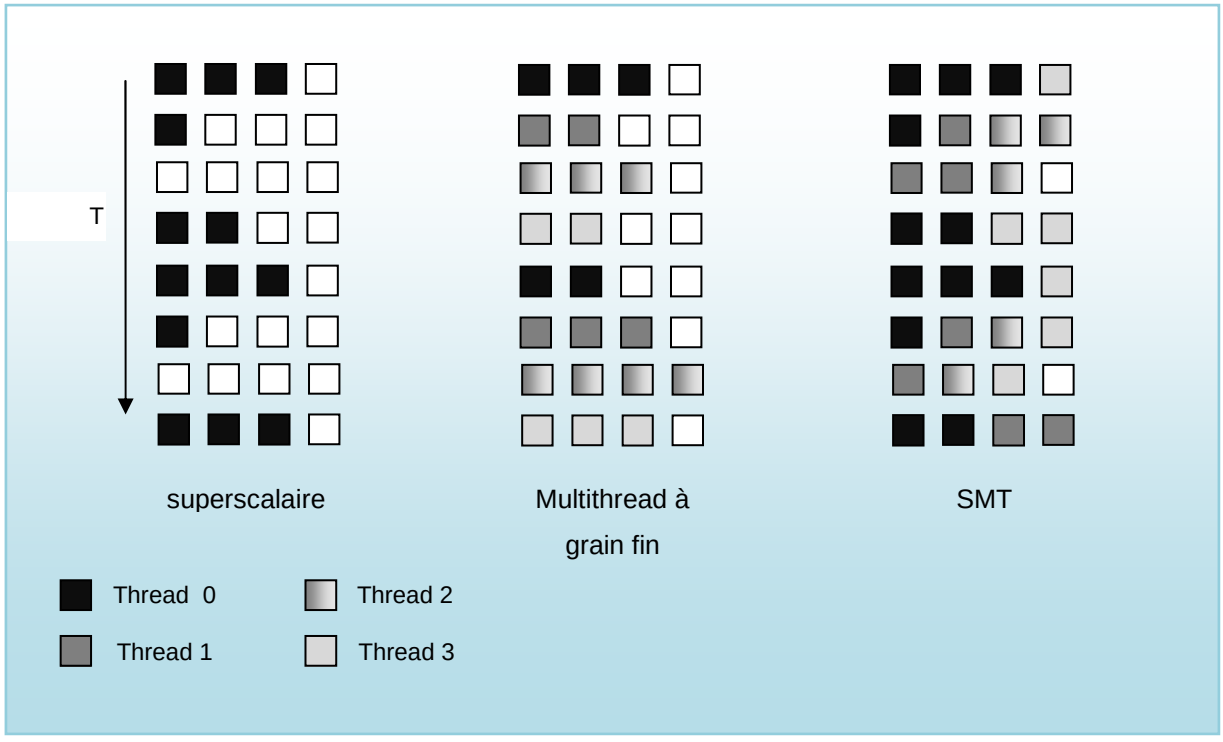

Figure 2. Partitionnement des unités d'exécution sur différentes architectures

L'exécution d'instructions de différents *threads* permet aussi de limiter l'impact des instructions à longues latences, la sous-utilisation verticale étant réduite en exécutant des instructions de *threads* non bloqués. On peut donc ainsi profiter du parallélisme de thread comme sur une architecture multithread (non simultanés).

L’abstraction la plus courante est qu’un processeur SMT de degré *n* (*n* threads simultanés) peut être vu comme un ensemble de *n* processeurs logiques, chacun avec son propre état architectural. Pour implémenter une architecture SMT, la solution la plus simple est de modifier une architecture superscalaire en dupliquant, augmentant et/ou partageant les ressources internes du processeur de manière à pouvoir suivre plusieurs flots d’exécution en permanence. C’est cette solution quia été considérée dans un premier temps car le SMT a été conçu dans une optique d’optimisation de l’architecture superscalaire. Ainsi, d’après Marr *et al.* [40], la première implémentation du Pentium 4 Hyper-Threading d’Intel (processeur SMT d’ordre 2) a augmenté la taille du circuit et les besoins énergétiques de moins de 5% pour pouvoir traiter deux threads simultanés.

Dans la Figure 3, on peut voir une illustration des changements entre un pipeline superscalaire et son équivalent SMT au niveau de la répartition des principales ressources internes. On peut noter que les différences sont minimes. Les modifications essentielles sont la duplication du compteur d’instructions (PC) pour chaque thread, l’ajout de registres physiques pour pouvoir gérer en théorie *n* fois plus d’instructions simultanément (pour *n* threads). Les « tables » du processeur (renommage, logique de prédiction de branchement, TLB, caches, etc) doivent aussi être agrandies mais surtout on doit pouvoir y faire la distinction entre les instructions de threads différents, soit en dédiant une table à chaque thread, soit en rajoutant des mécanismes d’identification (typiquement une étiquette pour l’identifiant du thread) dans une table partagée.

Concernant les caches d’instructions et de données, ils sont toujours partagés dynamiquement (pas de quota ni de limite pour un thread) dans les implémentations commerciales, ce qui peut entrainer des défauts récurrents si deux threads accèdent à tour de rôle au même bloc de cache, chacun engendrant ainsi un défaut pour l’autre. La pression serait surtout accrue sur le cache d’instruction si des instructions de plusieurs threads étaient lues à chaque cycle mais, dans la pratique, on ne lit des instructions que d’un seul thread. Il semble donc judicieux d’augmenter la taille des caches, voire de changer leur géométrie, pour limiter les interférences entre threads.

Les files d’instructions peuvent avoir plusieurs politiques de partage suivant l’implémentation, le choix des instructions à traiter se faisant en utilisant diverses

politiques d'ordonnancement, elles-aussi dépendant de l'implémentation. Ces politiques de partage et d'ordonnancement seront décrites dans la prochaine section.

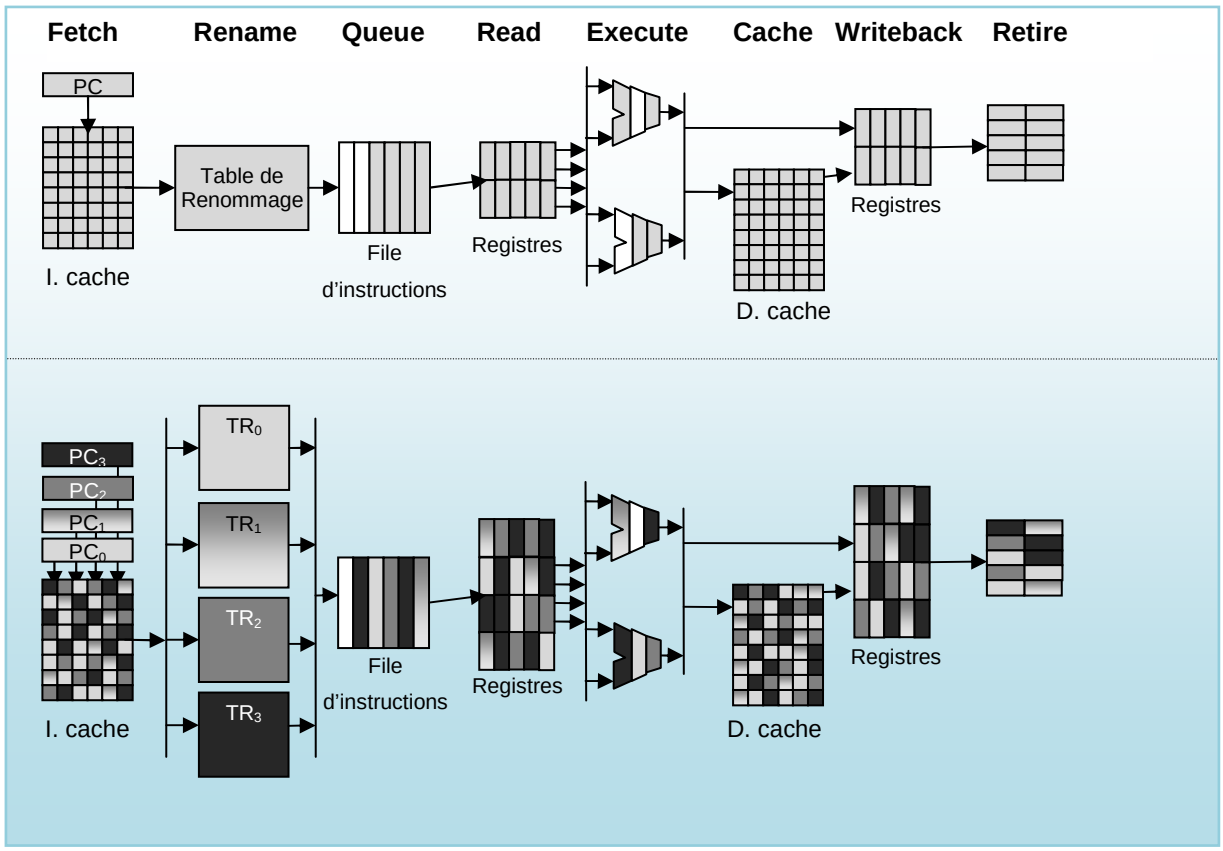

Figure 3. Comparaison entre un pipeline superscalaire et un pipeline SMT

3.1.1.1 Politiques de partage

La première de ces politiques, le *partage dynamique*, n'impose aucune limitation. Dès qu'une ressource (plus spécifiquement une entrée dans une file d'instruction) est libre, n'importe quelle instruction de n'importe quel thread peut concourir pour l'utiliser, sans aucune limitation quant au nombre d'entrées qu'un thread peut occuper. Cette politique est celle qui a été retenue en premier dans les travaux de recherche en raison de ses très bonnes performances globales moyennes. En contrepartie, son manque flagrant d'équité peut être pénalisant dans certains cas. En effet, des famines peuvent survenir quand un ou plusieurs threads « monopolisent » la ressource considérée (généralement ce sont des threads avec fort parallélisme d'instructions et donc fort IPC[1]

[1] IPC = Instructions Par Cycle. Par ce terme, on désigne le débit d'exécution du processeur.

potentiel). Les threads restants se retrouvent ainsi privés dans une large mesure de l'accès à la ressource, ce qui conduit à leur ralentissement voire à leur blocage.

Un aménagement du partage dynamique le rendant plus équitable est le *partage dynamique avec seuil*. La compétition entre threads est maintenant régulée par un nombre maximum d'entrées (seuil) que chaque thread peut réserver. Ce seuil ne peut en aucun cas être dépassé et est généralement supérieur à la part qui serait réservée en cas de partitionnement strict afin de maintenir de bonnes performances, les threads « gourmands » pouvant toujours utiliser des parts relativement grosses. Les risques de famines sont diminués (en particulier un thread ne peut plus monopoliser toute la ressource) mais sont encore présents.

La dernière politique, la plus stricte, est le *partitionnement*. Ici, la ressource est partitionnée en *n* parts égales (pour un SMT de degré *n*), chaque thread a une part réservée et aucun thread ne peut empiéter sur la part d'un autre. Les performances globales sont moins bonnes qu'avec les partages précédents car là où, avec le partage dynamique, un thread peut espérer avoir accès à toute la ressource et ainsi potentiellement augmenter sa vitesse d'exécution, ici il ne dispose que de *n* fois moins d'entrées et la réalisation de son IPC maximum est compromise. On peut par contre dire que le comportement de chaque thread vis-à-vis de cette ressource est indépendant du comportement des autres, comme s'il était isolé.

3.1.1.2 Politiques d'ordonnancement

Pour sélectionner les threads et en particulier les instructions à traiter dans une file, on est obligé de mettre en œuvre des politiques d'ordonnancement. Le choix d'une politique est souvent étroitement lié au type de partage de la ressource.

En cas de partage dynamique avec ou sans seuil, la politique la plus souvent mentionnée consiste à faire progresser les instructions les plus « vieilles » en priorité (en considérant leur ordre de chargement), le nombre d'instructions étant limité par la largeur de l'étage et l'occupation de la partie aval du pipeline.

Pour un partage statique, plusieurs politiques peuvent être envisagées. Il y a d'abord le très simple *Round-Robin* (RR) : chaque thread se voit accorder une opportunité à son tour de manière circulaire. Il faut distinguer le round-robin optimisé

(O-RR), qui saute le tour d'un thread qui n'a pas d'instruction prête ou qui est inactif, du round-robin strict (S-RR) qui sélectionne toujours aveuglément le prochain thread qui peut éventuellement être inactif ou n'avoir aucune instruction prête. L'algorithme RR étant insensible à ce qui se passe dans le processeur (avec une petite concession en ce qui concerne O-RR), il est réputé avoir des performances modérées par rapport à d'autres politiques se souciant du contexte.

Ces politiques basées sur l'état du processeur et donc des threads ont été considérées dans de nombreux travaux de recherche. Elles sont principalement destinées à contrôler le partitionnement de la bande passante de lecture des instructions entre plusieurs threads (dans le monde académique, on considère souvent des cœurs SMT d'ordre 4 ou 8, devant charger des instructions de plusieurs threads à chaque cycle pour avoir de bonnes performances). Chaque thread se voit assigner une priorité qui est réévaluée à chaque cycle.

Un bon exemple est la technique ICOUNT [10][12][13][56]. Elle est l'une des plus anciennes et est encore considérée comme étalon pour évaluer une nouvelle politique grâce à ses performances et à sa relative simplicité. Avec cette politique, les priorités sont calculées en fonction du nombre d'instructions dans les étages de pré-exécution du pipeline, c'est-à-dire le nombre d'instructions qui ne sont pas encore prêtes à s'exécuter et qui risquent de rester longtemps en attente. Plus un thread a d'instructions dans ces étages, plus il est considéré comme potentiellement bloquant ou lent, et sa priorité est donc faible. Quant aux threads réussissant à faire avancer leurs instructions rapidement et occupant peu ces étages, ils se voient attribuer une priorité élevée. On lit des instructions pour les threads les plus prioritaires (en effet, cette politique permet de lire des instructions pour plusieurs threads en même temps, à condition que le matériel le supporte).

D'autres politiques existent [13][57] et sont toutes basées sur un ensemble de compteurs divers pour la mise à jour des priorités des threads. Par exemple la politique BRCOUNT [57] considère le nombre de branchements dans les mêmes étages pré-émission alors que la politique DCRA [13] est fondée sur l'usage des ressources.

Une dernière politique possible, que nous qualifierons de *parallèle*, partitionne la bande passante entre les threads en sélectionnant, si possible, le même nombre

d'instructions pour chaque thread. Cela signifie tous les threads peuvent progresser de la même manière à chaque cycle. A notre connaissance, cette politique n'est implémentée que pour la sélection des instructions à retirer dans le POWER5 d'IBM [52].

3.1.1.3 Evaluation des performances

Nous allons maintenant parler des performances de l'architecture SMT. Pour cela, nous commenterons les résultats de Tulsenn *et al.* publiés dans [57] qui donnent un aperçu des performances théoriques que l'on peut attendre d'une telle architecture.

L'architecture simulée est dérivée d'un superscalaire de degré 8 avec partage dynamique des ressources et sans limitation sur le nombre d'instructions pouvant être envoyées à l'exécution. Plusieurs threads peuvent progresser en même temps dans la limite du degré de superscalarité. Les auteurs pensent que ce modèle est le moins réaliste en termes de complexité matérielle mais qu'il permet d'analyser le plein potentiel du SMT. D'autres configurations plus réalistes sont testées ; dans celles-ci on limite le nombre d'instructions pouvant être émises par thread.

On observe d'abord que l'utilisation du processeur augmente notablement avec le nombre de threads. Ceci est dû au partage dynamique des ressources entre un nombre de threads de plus en plus grand, la plupart de ces ressources restant inutilisées sur un superscalaire (c'est-à-dire le cas avec un seul thread sur la figure). Les effets négatifs sont la non équité entre les threads, certains à fort IPC se retrouvant favorisés par rapport à d'autres plus lents, et également la pression sur le cache qui est bien sûr accrue à cause du manque de localité dû au partage entre plusieurs threads, comme montré dans [30].

Du point de vue d'un thread, le taux de défauts du cache d'instructions passe de 1% avec un thread à 14% avec 8 threads. Les auteurs ont également mesuré l'impact de différentes techniques de partage des caches : leurs performances convergent avec l'augmentation du nombre de threads, des caches partitionnés entrainant de moins bons résultats avec peu de threads que des caches partagés dynamiquement. Ce désavantage pour le partitionnement pour l'exécution de peu de threads est cependant logique car les auteurs partitionnent toujours le cache en 8 parties même si moins de 8 threads s'exécutent.

Pour résumer, avec le modèle donné, une architecture SMT correctement configurée peut exécuter 4 fois plus d'instructions par cycle qu'un superscalaire mono-thread avec la même largeur d'émission (8 threads simultanés pour 8 instructions par cycle dans l'article). En clair, ces résultats laissent à penser que le SMT peut être une solution intéressante pour maximiser l'utilisation d'un processeur sans pour autant augmenter dramatiquement la taille du circuit. Dans la section suivante, nous allons décrire des processeurs haute performance du commerce.

3.1.2 Les principaux processeurs SMT du commerce

3.1.2.1 L'Alpha 21464 de Compaq

Ce processeur devait être le tout premier processeur SMT, mais il a été abandonné en 2001 avant d'avoir pu voir le jour, alors que sa conception était terminée. Peu d'informations sont disponibles à propos de ce processeur mais les principales données intéressantes sont quand même disponibles dans [15][45].

Il permet l'exécution de 4 threads simultanés. A chaque cycle, des instructions sont lues pour un thread (le mécanisme de choix du thread n'est pas mentionné) et reçoivent un identifiant de thread permettant de les repérer tout au long du pipeline. Ce pipeline semble similaire à celui de la Figure 3, tant au niveau des étages principaux, qu'au niveau des partages. Seuls les compteurs de programme et les tables de renommage semblent dupliqués, les autres ressources sont partagées dynamiquement.

Les instructions sont émises dans le désordre et le choix des instructions à émettre ne dépend pas des threads mais uniquement l'état (prêt ou non) des instructions. Pour permettre la gestion de l'exécution SMT, la surface du circuit a augmenté de seulement 6% environ par rapport à une version superscalaire. En ce qui concerne les registres, chaque thread se voit attribuer 32 registres entiers et 32 flottants. Ce qui fait donc 256 registres physiques nécessaires pour maintenir l'état architectural des 4 threads. Avec les 256 registres de renommage additionnels permettant de supporter jusqu'à 256 instructions en cours d'exécution, on a ainsi un total de 512 registres.

Dans [14], on trouve quelques résultats de simulation que Compaq aurait obtenus en simulant « leur SMT » (on suppose qu'il s'agit du futur 21464). En exécutant simultanément des groupes de 4 programmes, on trouve que le débit d'instructions est

supérieur en moyenne de 125% au cas où les programmes sont exécutés de manière séquentielle. Sur des applications multithreadées (composées de plusieurs threads logiciels), l'augmentation est plus modérée, à hauteur de 75% (avec des pointes jusqu'à 150%).

3.1.2.2 Le Pentium 4 Hyper-Threading d'Intel

Chez Intel, la technologie SMT a été appelée Hyper-Threading (HT). L'Hyper Threading a été introduit sur les processeurs pour serveurs de la gamme Xeon début 2002 et vers la fin de l'année est apparu le Pentium 4 HT destiné au grand public [40].

Dans ce processeur, deux threads peuvent s'exécuter simultanément. Chaque processeur logique maintient une copie de son état architectural (registres logiques, registres de contrôles, logique de traitement d'interruption) ainsi qu'un exemplaire des ressources suivantes : ITLB, pile d'adresses de retour, historique des branchements, table de renommage ; le reste est partagé entre les deux processeurs logiques. Les files sont partagées principalement de manière statique avec un arbitrage O-RR, également utilisé pour déterminer le thread dont on doit lire les instructions.

Ce mode de partage garantit une bonne isolation des threads et évite que le comportement de l'un ne perturbe trop l'autre. Chacun des deux processeurs logiques peut être désactivé par le système (exécution d'une instruction HALT sur le processeur logique à arrêter), le processeur passant alors en mode mono-thread (le thread restant dispose alors de toutes les ressources disponibles).

Dans [40], quelques résultats expérimentaux sont donnés pour des charges de travail orientées serveur. On a, en moyenne, un gain de 20% en performance si on active l'Hyper-Threading (par rapport à une exécution séquentielle), avec une augmentation de 65% par rapport à la précédente génération de processeur (qui n'avait pas d'HT). Ces mesures sont corroborées dans [55] et [7], les auteurs estimant qu'elles correspondent bien aux promesses de la recherche. En effet, en considérant le Pentium 4 HT comme un SMT à 2 threads de largeur 3, les resultats obtenus sont bien en accord avec le modèle décrit dans [57].

3.1.2.3 Le POWER5 d'IBM

Le POWER5 d'IBM [25][52] est un processeur haut de gamme pour serveurs, évolution du POWER4. Sa principale nouveauté est l'introduction de l'exécution SMT. Il comporte deux cœurs, chacun est SMT d'ordre 2 et peut fonctionner aussi bien en mode SMT qu'en mode ST (Single Thread) comme le Pentium 4 HT.

Des groupes de huit instructions appartenant à un même thread, choisi par une politique S-RR, sont lus simultanément. Le suivi des instructions dans le pipeline se fait par groupes d'instructions formés au dispatch, d'au plus 5 instructions consécutives d'un même thread, chaque groupe occupant une entrée dans la Global Completion Table (GCT, équivalent du tampon de réordonnancement). Quand toutes les instructions d'un groupe sont terminées, on peut retirer le groupe : la logique de retrait fonctionne suivant la politique parallèle mentionnée plus haut, un groupe de chaque thread pouvant être retiré à chaque cycle.

La plupart des ressources de stockage de la partie désordonnée du pipeline, en particulier les files d'émission, sont partagées dynamiquement, d'autres étant dupliquées (ce qui équivaut à un partage statique). Contrairement au Pentium 4 HT, le but principal n'est pas de garantir une bonne isolation des threads, mais d'assurer de bonnes performances, d'où l'utilisation du partage dynamique.

Pour essayer de minimiser les problèmes liés à un tel mode de distribution de ressources, des améliorations sont prévues pour garantir une certaine qualité de service et éviter de trop grandes inégalités entre les threads. L'équilibrage dynamique de ressources (*dynamic resource balancing*) a pour but de garantir l'équité : si un thread occupe trop d'entrées de la GCT, on peut le freiner en « ralentissant » son décodage (on lui donne moins de cycles de décodage). La « vitesse » du décodage peut aussi être contrôlée par une priorité ajustable (*ajustable thread priority*) modifiable par logiciel et qui va de 0 (thread inactif) à 7 (mode *Single Thread* ou ST, un thread actif avec une telle priorité se retrouvera seul à s'exécuter, l'autre thread étant stoppé jusqu'au prochain changement de priorité). Le thread avec la plus grande priorité se voit attribuer plus de cycles de décodage que les autres. Si tous les threads ont une priorité de 0 ou 1, le processeur rentre alors en mode économie d'énergie en ralentissant le décodage pour tout le monde. A partir de tout ça, on voit que la politique d'ordonnancement en sortie

des deux files d'instructions (une pour chaque thread) est relativement complexe et pourrait s'apparenter aux politiques de type ICOUNT.

Des résultats concernant les performances de ce processeur SMT peuvent être trouvés dans [41]. Huit applications de traitement lourd de données sont testées et les auteurs déterminent un gain SMT en comparant l'exécution d'un thread en mode ST au même thread exécuté en double en mode SMT. Ces gains varient suivant les programmes de 11% à 41% environ, avec une moyenne de 23%. Cette architecture a récemment évolué pour donner le POWER6 décrit dans la section suivante.

3.1.2.4 Le POWER6 d'IBM

Le POWER6 est actuellement le dernier processeur haut de gamme d'IBM sorti courant 2007 [31]. C'est toujours un double cœur SMT d'ordre 2, mais avec une philosophie différente concernant les performances visées.

La logique de chaque étage a été simplifiée pour minimiser les délais et la consommation, permettant ainsi une importante montée en fréquence souhaitée par les concepteurs. A la place d'une exécution désordonnée et spéculative, la conception s'est concentrée sur la pré-lecture de données (*data prefetch*). L'exécution désordonnée ne subsiste partiellement que pour les instructions flottantes et la prédiction de branchement est réduite à sa plus simple expression (une table d'historiques 2 bits à 16k entrées).

Les autres principales nouveautés sont l'introduction d'unités de calcul flottant en base 10 et la possibilité de détecter et récupérer des erreurs dans l'état du processeur. En ce qui concerne le mode SMT, peu de changements à part l'associativité accrue des caches L1 et une plus grande capacité au niveau L2. On note aussi une logique de décodage dupliquée, de même que la GCT.

Peu d'informations sur ce processeur SMT sont disponibles : l'accent semble avoir été porté sur la nouvelle architecture interne censée apporter des gains importants en performance grâce à une facile montée en fréquence. Pour le moment il n'y a pas de données disponibles concernant ses performances et il est encore trop tôt pour pouvoir porter un jugement.

3.1.2.5 L'Atom d'Intel

Ce processeur est dédié à l'informatique embarquée, il est en effet caractérisé par une petite taille et une consommation très basse (comparé à un processeur traditionnel). Il marque également le retour de la technologie Hyper-Threading (implémentation du SMT d'ordre 2 par Intel) qui avait été délaissée après les derniers Pentium 4. Une description de son architecture est donnée dans [23].

Il dispose d'une architecture 64 bits x86 avec les mêmes extensions que les processeurs Intel grand public telles la virtualisation et les instructions SSE3 destinées au multimédia. Il est ainsi capable d'exécuter des systèmes d'exploitation lourds tels Windows Vista ou Linux. Bien entendu, des compromis ont dû être faits au niveau des performances pour réduire la consommation.

Les ingénieurs Intel décidèrent de concevoir une nouvelle architecture, et non pas en modifier une existante, en prenant comme objectif premier une faible consommation, les performances étant secondaires à l'inverse des autres processeurs de la marque. Le fait d'inverser les priorités entre performances et consommation affecte toute l'architecture.

L'architecture comprend 2 pipelines ordonnés, chacun pouvant émettre une instruction à la fois (ce qui est équivalent à un pipeline d'ordre 2). On voit donc que l'exécution désordonnée n'a pas été retenue, ceci pour des raisons de consommation d'énergie. En effet, la logique permettant de suivre les instructions et de les remettre dans l'ordre a été jugée trop consommatrice d'énergie. Il a été fait de même avec la prédiction de branchement, qui elle aussi a été simplifiée, ainsi que la transformation des instructions x86 CISC en micro-opérations RISC qui n'a été gardée que pour quelques cas particuliers (instructions combinant des opérations multiples par exemple).

L'Atom supporte l'Hyper-Threading (implémentation SMT d'Intel déjà vue sur le Pentium 4 Hyper-Threading), c'est à dire que c'est un processeur SMT d'ordre 2. Les ressources sont principalement partitionnées, du moins celles dont le partage est mentionné, comme la table d'envoi des instructions (*instruction-dispatch table*), le TLB ainsi que le tampon de pré-lecture du cache d'instruction (*prefetch buffer*).

Aucune information n'est donnée sur les mécanismes d'ordonnancement, comme avec les précédentes implémentations du SMT, on peut supposer que l'accès aux étages autres que l'exécution se fait en utilisant un Round Robin (strict ou optimisé suivant les cas). On peut aussi bien entendu penser que l'étage d'exécution peut traiter 2 instructions de threads différents à chaque cycle, ce qui est la base du SMT.

En ce qui concerne les résultats, des tests faits par Intel et révélés aux auteurs sous clause de confidentialité ont montré que l'activation du SMT augmente les performances de l'Atom de 36% à 47%, ces tests comprenaient des programmes des suites SPEC et EEMBC (suite adaptée à l'embarqué). La consommation, quant à elle, augmente de 17% à 19% sur ces mêmes programmes, ce que les auteurs jugent tout à fait convenable comparé au gain de performances. Pour ceux qui seraient d'avis contraire, Intel propose également des Atom sans Hyper-Threading. On peut remarquer en dernier lieu que, comme observé avec le Pentium 4, la surface du circuit augmente de façon modérée avec environ 8% de logique en plus.

Pour résumer on peut dire que l'Atom est le premier processeur SMT hautes performances destiné aux systèmes embarqués nécessitant une faible consommation et de bonnes performances. D'après les données filtrées par Intel il semblerait que ce double objectif soit atteint. Néanmoins, peu d'informations fiables sont disponibles pour le moment (excepté une foison d'articles sur certains sites internet) car l'architecture est toute récente à l'heure où ces lignes sont écrites. Pour se faire une meilleure idée, des tests indépendants seraient également les bienvenus.

3.1.3 Bilan

Tous les processeurs SMT commercialisés ne supportent que deux threads simultanés. Cela permet de se passer d'ordonnancements trop compliqués et inutiles pour seulement deux threads. Concernant les ressources partitionnées (ou dupliquées) qui sont omniprésentes sur les parties ordonnées des pipelines de ces processeurs (voire sur tout le pipeline comme sur le Pentium 4 HT), on voit que l'arbitrage est toujours un Round Robin, sauf en sortie des files d'instructions du POWER5 où l'on utilise un mécanisme permettant de surveiller le partage dynamique de la partie désordonnée du pipeline.

3.2 SMT et prévisibilité temporelle

Dans cette partie, nous allons décrire les problèmes pouvant survenir sur une architecture SMT dans un cadre temps-réel strict, problèmes liés aux spécificités architecturales des processeurs SMT. Nous montrons d'abord les difficultés causées par l'entrelacement des threads, ensuite nous décrivons les avantages et inconvénients des différentes stratégies de partage de ressources avec leurs politiques d'ordonnancement associées.

3.2.1 L'entrelacement des threads

Dans un processeur SMT, plusieurs threads sont exécutés en parallèle. Pour calculer le WCET d'un des threads co-exécutés, plusieurs méthodes sont envisageables. Leur faisabilité est ensuite discutée.

Crowley et Baer [14] étendent une approche largement utilisée pour le calcul du WCET (la technique IPET, présentée dans le chapitre 1) à l'analyse d'un processeur SMT. Ceci conduit à exprimer tous les entrelacements possibles entre les threads dans le cadre de la formulation ILP (programmation linéaire en nombres entiers) du calcul de WCET. Naturellement, la taille du système ILP généré croît exponentiellement avec la taille des threads et, à moins de considérer des tâches très simples (avec peu de contrôle de flot, engendrant ainsi peu de combinaisons possibles d'états), le problème ne peut être résolu en un temps raisonnable.

Cette constatation est la source de notre travail sur les architectures SMT prévisibles : nous pensons que l'entrelacement des threads doit être contrôlé à l'exécution (c'est-à-dire par le matériel) pour le rendre efficace et déterministe afin qu'il puisse être pris en compte pour l'estimation du WCET de tâches ayant des échéances strictes.

D'autres travaux ont pour but de rendre le comportement temporel des threads plus prévisible à travers des stratégies appropriées d'ordonnancement au niveau système. Lo *et al.* [36] explorent des algorithmes d'ordonnancement de tâches temps-réel sur une architecture SMT. Toutefois, ils ne prennent pas en compte les interférences possibles entre les threads à l'intérieur du pipeline et leurs effets sur le WCET des threads.

Dans [26], Kato *et al.* introduisent la notion de temps d'exécution multi-cas (*Multi-Case Execution Time* ou MCET), calculé à partir des différents WCETs obtenus lorsque le thread considéré s'exécute en présence de différents threads concurrents. Dans cette approche, les auteurs ne considèrent que des tâches périodiques et sans dépendances, chacune pouvant être préemptée à tout moment, l'ensemble des tâches est défini au départ et immuable. Le MCET est calculé en relevant les temps d'exécution sur une hyper-période et en faisant la moyenne des WCETs de chaque occurrence de la tâche considérée. Les MCETs servent ensuite de références pour des ordonnancements de type EDF. Nous pensons que le nombre de valeurs possibles d'un WCET peut être considérable dès que les threads concurrents ont un grand nombre de chemins d'exécution possibles (le nombre d'entrelacements des chemins peut alors être énorme). C'est pourquoi nous croyons que ces résultats requièrent une architecture prévisible pour être applicables.

Dans [16], le but est de préserver autant que possible les performances de certains threads prioritaires, tout en permettant aux autres threads de progresser. L'objectif est proche de celui permettant à un thread temps-réel de s'exécuter sans "interférences" de la part des autres threads (qui seraient alors non temps-réel) sauf qu'il n'assure pas de prévisibilité temporelle et n'est ainsi pas approprié à un contexte temps-réel strict.

Des mécanismes architecturaux ont été proposés par Carzola *et al.* [10][11][12] pour garantir une qualité de service pour un ensemble de threads. Cette solution cible principalement les systèmes temps-réels souples et dépasse le cadre de notre travail. Nous la citons quand même pour montrer que la recherche concernant les systèmes temps-réel à architecture SMT est loin d'être anecdotique, elle concerne en grande partie le temps-réel souple et nous pensons que les investigations concernant le temps-réel strict doivent être plus poussées.

3.2.2 Les politiques de distribution de ressources et d'ordonnancement

Les processeurs SMT exécutent plusieurs threads en même temps pour améliorer l'utilisation des ressources matérielles (principalement les unités fonctionnelles) [57]. Les threads concurrents partagent des ressources communes : files d'instructions, unités fonctionnelles, mais également les caches de données et d'instructions et les tables de

prédiction de branchements. Dans cette section, nous évoquons d'abord le problème des caches puis nous nous concentrons sur les ressources du pipeline.

En ce qui concerne les caches, le partage entre plusieurs threads pose bien entendu des problèmes. En effet, que ce soit dans la recherche ou dans les implémentations commerciales, les caches de données ou d'instructions sont toujours partagés dynamiquement. Aucun système de quota n'est prévu, l'allocation se fait en répondant strictement à la demande. Ainsi, si une ligne est allouée par un thread *a*, et qu'un thread *b* provoque un transfert mémoire engendrant fortuitement un accès à la même ligne, celle-ci lui sera allouée, engendrant un défaut lors du prochain accès de *a*.

A cause du nombre d'entrelacements possibles, cette situation de "vol de ligne de cache" d'un thread ne peut être prévue même avec la connaissance du comportement des autres threads susceptibles d'être co-exécutés. Dans un cadre temps-réel strict, si on veut pouvoir assurer un total déterminisme pour un thread donné indépendamment des autres, on ne peut garder cette configuration basique des caches.

Des tentatives de partages du cache ont été étudiées [51][37]. Typiquement, il s'agit d'allouer à chaque thread une partie du cache. La taille de ces parties pouvant être modifiée au cours de l'exécution suivant le comportement et les besoins des threads. Pour notre cadre temps-réel strict, ces méthodes cherchant à maximiser les performances globales ne conviennent pas. En effet, la quantité de cache allouée à un thread critique donné peut dépendre des autres threads co-exécutés, on ne pourrait alors déterminer le comportement du thread critique considéré de façon déterministe et indépendamment des autres threads. Il nous faut donc trouver un autre système de partage de cache pour notre objectif de prédictibilité maximum.

Pour les ressources du pipeline, deux catégories doivent être distinguées : les ressources de stockage (files et tampons d'instructions) gardent les instructions pendant un certain temps, généralement pour plusieurs cycles, tandis que les ressources de bande passante (ex : unités fonctionnelles ou étage de retrait) sont typiquement réallouées à chaque cycle [49].

Le partage des ressources de stockage est contrôlé à la fois en termes d'espace et de temps : il y a plusieurs politiques possibles pour distribuer les entrées de ressources parmi les threads actifs (politiques de distribution) et pour choisir les instructions qui

quitteront la ressource à chaque cycle (politiques d'ordonnancement).Ces politiques ont déjà été passées en revue section 3.1.1.1 et 3.1.1.2. Le partage spatial n'a pas de sens pour les ressources de bande passante car elles sont réallouées à chaque cycle. En fonction des stratégies de distribution et d'ordonnancement, le partage des ressources peut être une source majeure d'indéterminisme pour le comportement temporel d'un thread.

Dans cette section, nous reprenons les politiques de distribution et d'ordonnancement les plus courantes (que ce soit dans la recherche ou dans des projets industriels) et nous soulignons les problèmes de prévisibilité qui peuvent se poser quand on considère un thread temps-réel strict s'exécutant avec des threads arbitraires sur un processeur SMT.

3.2.2.1 Politiques de distribution de ressources

Le procédé le plus flexible pour distribuer les entrées d'une ressource (ex : une file d'instructions) entre les threads est la politique de *partage dynamique* selon laquelle n'importe quelle instruction de n'importe quel thread peut prétendre à n'importe quelle entrée libre.

On a déjà vu que le risque de famine est non négligeable avec cette distribution, le moment où une telle famine peut se produire pour un thread dépendant des comportements respectifs des threads concurrents. Si on considère un thread temps-réel, on ne peut pas garantir qu'une de ses instructions nécessitant une entrée dans une ressource partagée dynamiquement l'obtiendra immédiatement, et le temps que cette instruction peut avoir à attendre avant d'être admise par la ressource ne peut pas être borné.

Ainsi, avec une distribution dynamique, le WCET d'un thread ne peut pas être estimé à cause de la grande variabilité des délais d'accès à des ressources partagées. Cette politique n'est donc pas appropriée pour des applications temps-réel strict.

La politique de *distribution dynamique avec seuil* a été conçue pour minimiser les risques de famine, maintenant un seul thread ne peut plus monopoliser toutes les entrées. Malheureusement, dans le contexte des applications temps-réel strict, cette politique est également de nature non prévisible : on ne peut pas déterminer si un thread

pourra avoir autant d'entrées de ressource que le seuil puisque certaines entrées peuvent être utilisées par d'autres threads. Aussi, le thread en question pourrait être retardé pour l'obtention d'une entrée dans une ressource de stockage, même s'il n'a pas atteint le seuil, et ce délai ne peut être borné. Ainsi, il n'est pas possible de dériver une estimation précise de WCET pour un thread temps-réel.

La distribution des ressources peut aussi être *statique* : chaque ressource est partitionnée et chaque thread a un accès privé à une des partitions. En dépit d'une baisse des performances, cette politique de partage est naturellement la plus adéquate pour un système temps réel strict car elle est totalement déterministe. En effet, chaque thread se voit accorder une partition fixe de chaque ressource qu'il ne peut dépasser et qui ne peut pas être utilisée par un autre thread. Ainsi le comportement d'un thread en ce qui concerne les ressources partagées statiquement ne dépend pas des threads environnants.

3.2.2.2 Politiques d'ordonnancement

En plus de leur politique de distribution, les ressources partagées sont aussi contrôlées par une politique d'ordonnancement qui joue le rôle d'arbitre entre les threads pour le choix des instructions qui peuvent quitter la ressource et avancer dans le pipeline. Nous avons déjà décrit ces différentes politiques et nous allons maintenant passer en revue leurs particularités liées au temps-réel strict.

Le premier de ces algorithmes est le très simple *round robin* (RR) qui peut se décliner de deux manières, *round robin strict* (S-RR) et *round robin optimisé* (O-RR). En ce qui concerne les applications temps-réel strict et le calcul du WCET, S-RR est une politique très prévisible : il est très facile de déterminer quand les instructions d'un thread seront choisies puisqu'on est sûr que le thread sera sélectionné tous les *n* cycles si le processeur est conçu pour gérer jusqu'à *n* threads actifs. L'algorithme O-RR, qui saute le tour d'un thread inactif ou sans instruction prête, altère légèrement la prévisibilité mais le délai entre deux sélections d'un thread peut toujours être borné.

En ce qui concerne les politiques basées sur l'état du processeur et donc des threads (*icount* et ses dérivés, de même que la politique "les plus vieilles instructions d'abord"), leur comportement est fortement indéterministe. En effet, un ordonnancement basé sur des priorités dynamiques de threads rend le comportement temporel d'un thread beaucoup trop dépendant des autres threads actifs. Aussi, la politique *icount* et ses

dérivées ne peuvent pas être utilisés dans le cadre du calcul de WCET pour un thread temps-réel strict.

La dernière politique que nous avons décrite, celle que nous qualifions de *parallèle*, est, comme la politique S-RR, totalement déterministe. En effet le partitionnement de la bande passante entre les threads en sélectionnant, si possible, le même nombre d'instructions pour chaque thread permet de garantir que tous les threads peuvent progresser de la même manière à chaque cycle, sans interférences sur le comportement d'un thread de la part des autres threads présents.

3.3 Conclusion

Dans cette section, nous avons décrit l'architecture SMT ainsi que quelques exemples de processeurs réels. Nous avons ensuite mis en lumière les difficultés engendrées par une telle architecture pour assurer l'exécution de tâches temps-réel strict et le calcul de WCET. Nous avons noté qu'un déterminisme minimum est nécessaire pour parvenir à nos fins, ce qui n'est pas possible sur une architecture SMT de base non adaptée au temps-réel.

Notre objectif est donc de concevoir une architecture SMT qui rend possible l'analyse du WCET de tâches temps-réel. Concrètement, nous désirons une architecture où l'exécution de n'importe quel thread temps-réel est totalement déterministe et ne dépend pas des threads concurrents. Pour cela, nous décidons d'axer nos travaux sur les politiques de partage et d'ordonnancement car nous sommes convaincus que des politiques judicieusement conçues offriraient le déterminisme attendu. Selon ce que nous avons expliqué dans ce chapitre, il parait évident que nous choisirons une distribution statique des ressources et que nous exclurons les politiques dynamiques (avec ou sans seuil) ainsi que celles basées sur l'état du processeur.

4 Vers une architecture multi-flots simultanés prévisible

Dans ce chapitre, nous proposons une architecture SMT adaptée au calcul de WCET. Cette architecture doit être capable d'exécuter un ou plusieurs thread(s) ayant des contraintes temps-réel strictes en parallèle avec des threads moins critiques. Le comportement temporel du thread temps-réel strict doit être prévisible par analyse statique pour que l'on puisse prouver qu'il satisfait toujours ses échéances.

Le cœur de la solution réside dans le choix et la conception des politiques de contrôle du partage des ressources entre les threads concurrents (distribution et ordonnancement).

4.1 Architecture de base

Nous allons tout d'abord décrire l'architecture SMT qui sert de base à notre travail.

Le pipeline, présenté sur la Figure 4 comporte cinq étages et trois files/tampons d'instructions. L'étage d'exécution comporte plusieurs unités fonctionnelles. L'ensemble de ces ressources (étages, unités fonctionnelles, files et tampons d'instructions) est partagé par les threads co-exécutés.

Dans l'étage de lecture des instructions (IF), les instructions d'un thread sont lues en mémoire et rangées dans la file de lecture des instructions (FQ). Là, elles attendent d'être sélectionnées pour le décodage (étage ID), après quoi elles entrent dans la file de décodage (DQ). Après le renommage (étage RN), elles sont rangées dans le tampon de ré-ordonnancement (ROB). L'étage EX (exécution) sélectionne à partir du ROB des instructions dont les opérandes sont prêts et pour lesquelles une unité fonctionnelle est disponible. Des instructions appartenant au même thread peuvent être exécutées *dans le désordre* pour améliorer le parallélisme d'instructions. Les instructions terminées sont finalement enlevées du ROB, dans l'ordre du programme, par l'étage CM (retrait) et quittent le pipeline.

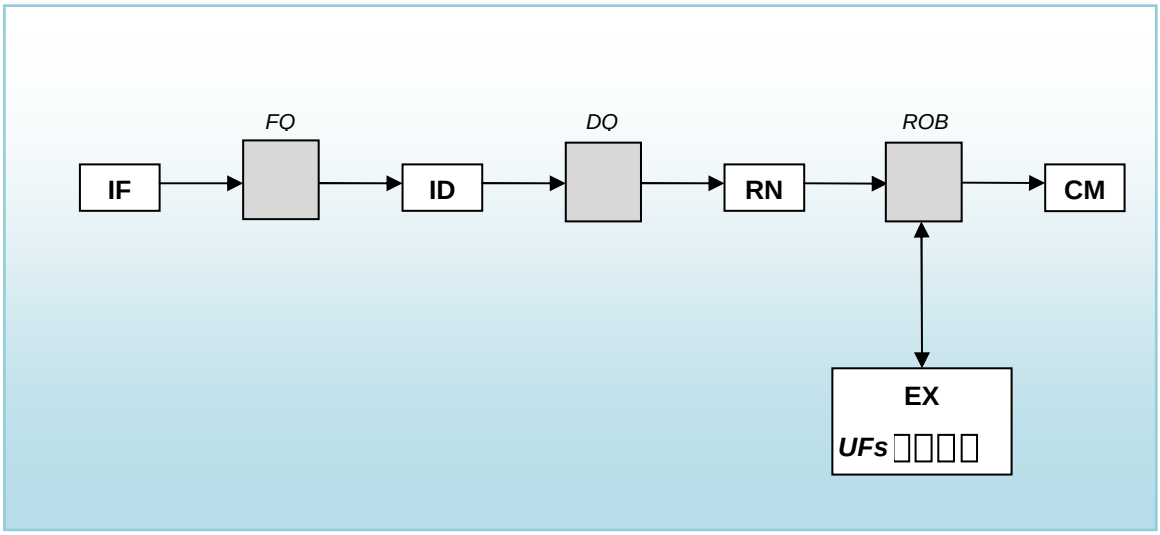

Figure 4. Architecture SMT de base (non prévisible)

4.2 Une première étape : support d'un thread critique

Pour cette première étape, notre objectif est de concevoir une architecture SMT qui rend possible l'analyse du WCET d'un *unique* thread temps-réel strict qui peut s'exécuter avec d'autres threads non critiques. Le thread temps-réel doit impérativement s'exécuter sans interférences avec les autres threads pour présenter un comportement temporel analysable. Dans le même temps, un certain niveau de performances doit être maintenu, au moins pour les autres threads, afin que les avantages de l'exécution SMT ne soient pas annulés par nos modifications de l'architecture.

4.2.1 Politique de distribution de ressources

Pour obtenir la prévisibilité temporelle d'un thread critique, chaque ressource de stockage (files d'instructions et de décodage, tampon de ré-ordonnancement) est partitionnée statiquement. Comme indiqué dans la section précédente, seul le partitionnement statique peut rendre le comportement temporel d'un thread temps-réel analysable. Ceci est illustré dans la Figure 5 où l'on considère, à titre d'exemple, un processeur pouvant supporter deux threads actifs : les files de lecture, de décodage et de réordonnancement sont distribuées statiquement en deux partitions, une pour chaque thread.

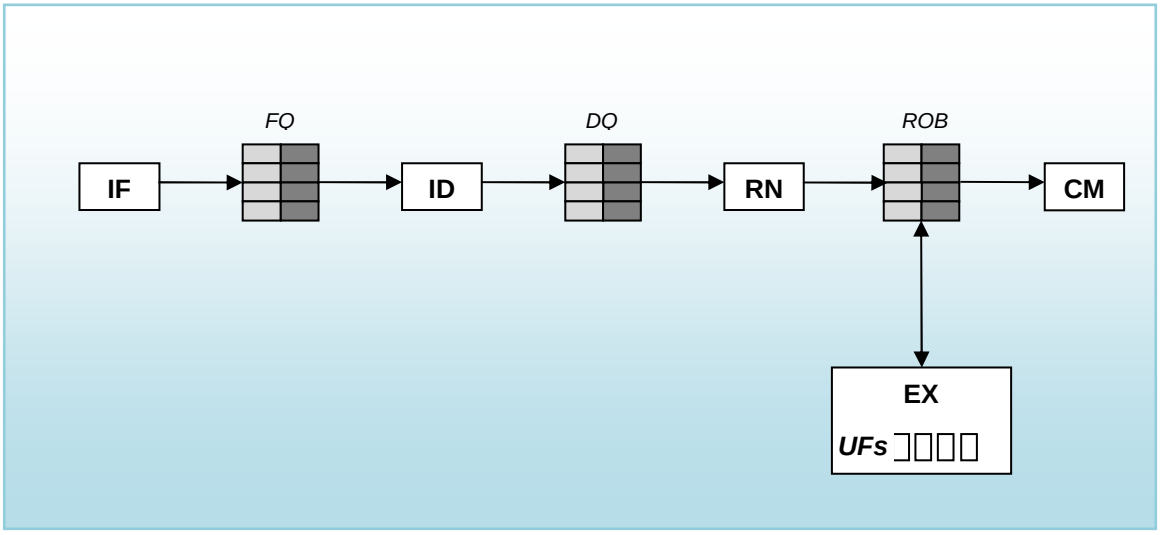

Figure 5. Un pipeline SMT temporellement prédictible (support de deux threads dont *un* critique)

4.2.2 Ordonnancement des threads

A chaque cycle, l'étage IF lit une séquence d'instructions pour un seul des threads, sélectionné par une politique S-RR dont nous avons dit qu'elle était prévisible. Cette politique est totalement prédictible et relativement simple à mettre en œuvre car on ne lit qu'à partir d'un thread, contrairement à la politique parallèle par exemple. Elle permet également à tous les threads (critiques ou non) d'avoir les mêmes opportunités pour la lecture des instructions. Comme indiqué précédemment, la file de lecture est partitionnée statiquement. Nous devons maintenant spécifier l'algorithme implémenté pour sélectionner les instructions à décoder parmi celles présentes dans la file de lecture. La Figure 6 rappelle les ordonnancements utilisés pour gérer un thread critique.

Pour que le thread temps-réel strict (que nous désignerons par *hrt-t*, ou *hard real-time thread*, par la suite) avance de manière indépendante des threads concurrents, nous proposons un mécanisme à priorités partielles fixes, que nous appelons *Most-Critical-First* (MCF), où les instructions du thread *hrt-t* sont sélectionnées en premier lieu. Si le nombre d'instructions du thread *hrt-t* dans la file de lecture est inférieur à la bande passante de décodage, des instructions des threads non critiques sont également sélectionnées selon un algorithme O-RR. Après le décodage, les instructions entrent dans la file de décodage qui est partitionnée. Elles sont sélectionnées pour le renommage à l'aide de la stratégie prévisible MCF décrite ci-dessus. Une fois renommées, les instructions sont insérées dans le tampon de ré-ordonnancement partitionné statiquement où elles attendent leurs opérandes sources. Quand une instruction est prête, elle devient éligible pour l'envoi à une unité fonctionnelle.

En plus de l'ordonnancement entre instructions d'un même thread (ex : instructions les plus vieilles sélectionnées en premier), la politique d'ordonnancement des threads doit aussi choisir entre les threads qui ont des instructions prêtes. En ce qui concerne le thread *hrt-t*, le temps d'attente d'une de ses instructions prêtes ne doit pas dépendre des threads concurrents. Pour cela, on complète la politique MCF par un mécanisme que nous appelons *Replay*. Quand une instruction du thread *hrt-t* est prête à s'exécuter, trois situations peuvent être observées :

- si l'unité fonctionnelle est libre, l'instruction peut y être envoyée immédiatement :
- si l'unité fonctionnelle requise est déjà allouée à une instruction appartenant également au thread *hrt-t*, l'exécution est retardée. Néanmoins, cette situation reste analysable statiquement car elle ne dépend que du comportement propre du thread.
- si l'unité demandée est utilisée par une instruction d'un thread non critique, cette instruction est éjectée de l'unité fonctionnelle et marquée prête dans le tampon de ré-ordonnancement (c'est le mécanisme de *Replay*). Cela signifie qu'elle devra être réémise plus tard. L'unité fonctionnelle est alors immédiatement attribuée au thread *hrt-t*. Grâce à l'ordonnancement MCF, cette situation ne peut arriver que lorsqu'un thread non critique exécute une opération à latence supérieure à un cycle sur une unité fonctionnelle non pipelinée (dans notre exemple de cœur, cela ne concerne que les divisions).

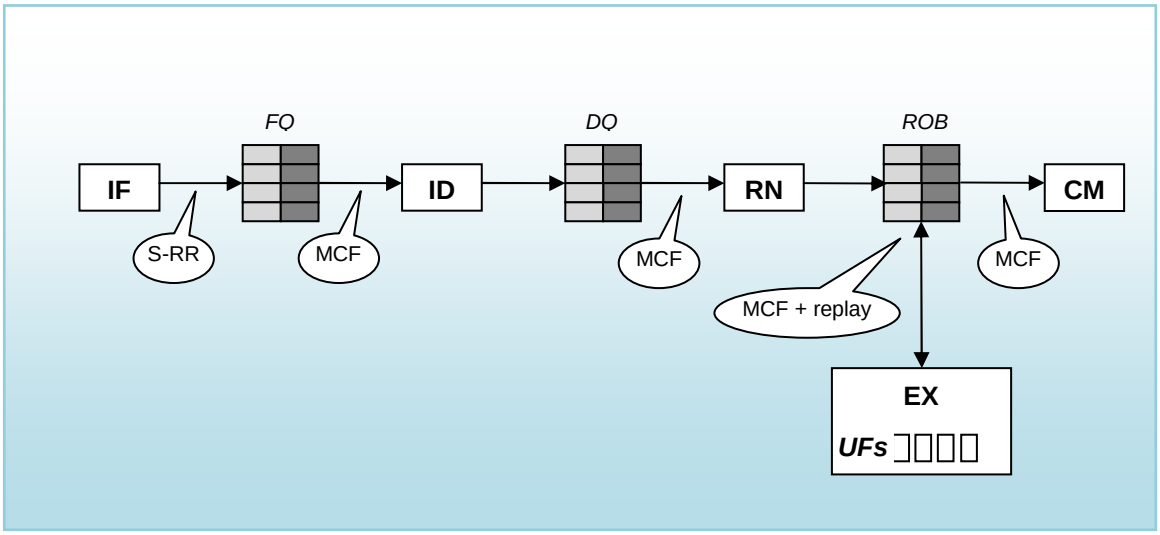

Figure 6. Un pipeline SMT temporellement prédictible avec ses politiques d'ordonnancement (support de 2 threads dont 1 critique).

Une alternative à ce mécanisme Replay serait de laisser les instructions non critiques s'exécuter sans les éjecter. Dans ce cas, les interférences des threads non critiques sur le thread critique sont alors prises en compte dans le calcul du WCET. Ce qui se fait en surestimant au pire cas le temps passé par une instruction critique à attendre et s'exécuter sur une unité fonctionnelle.

Finalement, les instructions terminées sont sélectionnées pour le retrait par le mécanisme MCF qui assure prévisibilité temporelle au thread temps-réel strict.

4.3 Support de *plusieurs* threads critiques

Nous allons maintenant décrire une architecture SMT qui permet d'exécuter *plusieurs* threads temps-réel strict en parallèle avec d'autres threads non critiques. La principale difficulté est que le comportement temporel de chaque thread temps-réel strict doit être prédictible par analyse statique afin qu'il puisse être ordonnancé de manière valide pour respecter ses échéances. Comme pour notre architecture à un seul thread critique, la prédictibilité temporelle est assurée par l'implémentation de politiques spécifiques pour la distribution et l'ordonnancement des ressources internes du processeur, sans oublier de garantir également une certaine part de bande passante pour les éventuels threads non critiques.

4.3.1 Politique de distribution des ressources

Pour assurer une prédictibilité temporelle, nous choisissons encore de partitionner chaque ressource de stockage. En effet, seul le partitionnement statique peut permettre d'assurer une prédictibilité temporelle pour chaque thread critique. Sur la Figure 5, on peut voir ce partitionnement illustré pour 2 threads : chaque file de stockage est partagée en deux parts égales, une pour chaque thread.

4.3.2 Politique d'ordonnancement des threads

Comme précédemment, le but est de favoriser la progression des threads critiques en leur donnant la priorité sur les threads non critiques pour éviter d'éventuelles perturbations qui pourraient entrainer une situation non déterministe. La Figure 7 récapitule les nouvelles politiques d'ordonnancement présentées ci-après.

Lorsque l'on a plusieurs threads critiques, la solution est moins triviale. En effet, chacun d'entre eux doit être considéré comme prioritaire, ce qui dépasse les capacités de notre algorithme MCF. Une solution envisageable serait d'accorder la priorité aux threads critiques, chacun son tour. Toutefois, ceci diminuerait fortement les avantages du modèle d'exécution SMT lorsque l'on a peu de threads non critiques pour « combler les vides », un seul thread étant « actif » à chaque cycle. C'est pourquoi nous nous sommes plutôt orientés vers un partitionnement de la bande passante que nous allons décrire ci-dessous.

A chaque cycle, l'étage IF lit des instructions d'un thread choisi suivant une politique Round-Robin Strict (S-RR). Pour les autres étages les instructions sont choisies par une politique de partitionnement de bande passante que nous appellerons *Bandwidth Partitioning* (BP). Elle consiste, comme son nom le laisse deviner, à attribuer une part égale de la bande passante à chaque thread critique Dans un cœur supportant *nc* threads critiques simultanés, pour une ressource qui est capable de gérer *n* instructions par cycle, la politique BP garantit une bande passante *b* à chaque thread critique avec :

- *b* égal à *n* / *nc* instructions par cycle si $n \geq nc$
- *b* égal à une instruction tous les *p* cycles, avec $p=nc / n$, sinon

Dans le Tableau 1, on trouvera des exemples numériques pour illustrer ces différentes possibilités. Naturellement, dès qu'un thread critique n'utilise pas toute sa part de bande passante, les threads non critiques peuvent utiliser les places libres suivant un ordonnancement Round-Robin.

	Nombre de threads critiques		
Bande passante totale de la ressource	***1***	***2***	***4***
8 inst. par cycle	8 inst. par cycle	4 inst. par cycle	2 inst. par cycle
4 inst. par cycle	4 inst. par cycle	2 inst. par cycle	1 inst. par cycle
2 inst. par cycle	2 inst. par cycle	1 inst. par cycle	1 inst. tous les 2 cycles
1 inst. par cycle	1 inst. par cycle	1 inst. tous les 2 cycles	1 inst. tous les 4 cycles
1 inst. tous les ℓ cycles	1 inst. tous les ℓ cycles	1 inst. tous les *2*ℓ cycles	1 inst. tous les *4*ℓ cycles

Tableau 1. Partitionnement des bandes passantes de ressource

La politique BP doit être complétée pour traiter le problème de l'occupation des unités fonctionnelles (UF) non pipelinées. Ici, on ne peut pas se permettre d'éjecter une instruction occupant une UF demandée par un thread critique (mécanisme *Replay*). En effet, l'instruction à éjecter peut appartenir à un autre thread critique et le fait de l'éjecter conduirait, certes, à une situation déterministe pour le thread demandeur de l'UF, mais s'avèrerait catastrophique pour le thread éjecté s'il est lui-aussi critique.

Une solution, comme pour notre architecture à un thread temps-réel est de tenir compte des latences d'accès pire cas aux unités fonctionnelles non pipelinées dans le calcul de WCET. En général, les unités non pipelinées sont peut utilisées donc le pessimisme introduit par cette méthode devrait rester faible. Par la suite, nous ne considérons que les unités fonctionnelles pipelinées. Ainsi, grâce à la politique BP partitionnant la bande passante et les unités fonctionnelles, si un thread critique veut accéder à une unité fonctionnelle qui lui est accordée par BP, on est sûr qu'il ne peut pas subir d'interférence de la part des autres instructions occupant l'unité. En effet, les autres instructions sont rentrées dans l'unité à un cycle antérieur et, comme l'unité est pipelinée, elles ne produisent aucune gêne pour une instruction désirant accéder à l'unité en question.

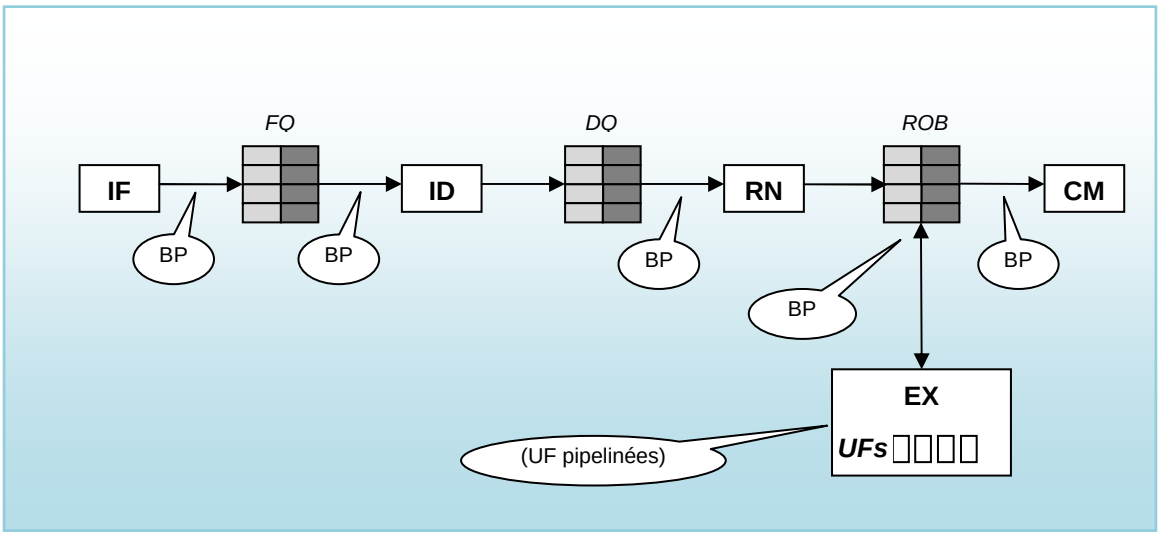

Figure 7. Un pipeline SMT temporellement prédictible avec ses politiques d'ordonnancement (support de 2 threads dont 1 ou 2 critiques).

La garantie d'une certaine quantité bien définie de la bande passante disponible pour un thread critique constitue l'atout principal pour rendre son comportement indépendant des autres threads co-exécutés, et donc déterministe.

On remarque que si on considère un seul thread critique, l'algorithme BP devient équivalent à l'algorithme MCF concernant le comportement du thread critique. Sur une architecture à unités fonctionnelles pipelinées, les threads non critiques se comportent également de la même manière, que ce soit sur l'architecture à un thread critique ou sur la deuxième architecture configurée pour n'exécuter qu'un seul thread critique. On peut donc ne considérer que la deuxième architecture (ce que nous faisons dans nos expérimentations), la première architecture ayant quand même été traitée dans [2] et [3].

4.3.3 Modes d'exécution

Ce que nous proposons est une architecture adaptée au calcul de WCET qui dispose de plusieurs modes d'exécution, chacun correspondant à un certain nombre de threads critiques supportés. Par exemple, un cœur à 4 threads dispose de quatre modes correspondant, respectivement, à zéro, un, deux ou quatre threads critiques (le partitionnement de ressources en trois serait plus compliqué à mettre en œuvre).

Par la suite, nous noterons respectivement ces modes m_0, m_1, m_2 et m_4. Le changement entre deux modes nécessite de reconfigurer le partitionnement des ressources (que ce soit les ressources de stockage ou de bande passante, selon la

terminologie expliquée dans le chapitre d'introduction). Pour des raisons de simplicité, nous considérons que cela ne peut se faire que quand le pipeline est vide (aucun thread actif).

Le temps d'exécution pire cas d'un thread dépend du mode d'exécution du cœur au moment de son lancement. Nous pensons qu'un ordonnancement de tâches intelligent pourrait prendre en compte les différents WCET (un pour chaque mode d'exécution) de chaque tâche pour séquencer les threads de manière efficace, en initiant un changement de mode lorsque nécessaire.

4.4 Calcul de WCET pour l'architecture SMT prévisible

La méthode d'analyse de coûts de blocs de base fondée sur des graphes d'exécution paramétrés (que nous avons brièvement présentée dans le chapitre 2, paragraphe 2.4.1, et qui a été exposée plus en détail dans [50]) est tout à fait adaptée à notre architecture prévisible. En effet, chaque thread critique étant exécuté de manière tout à fait isolée des autres threads, on peut construire un graphe tenant compte des restrictions et calculer ainsi le coût de chaque bloc de base sans rien savoir des threads potentiellement co-ordonnancés.

4.5 Conclusion

Nous avons décrit deux types d'architecture SMT prédictible. Notre première architecture permet d'exécuter un thread critique seulement avec éventuellement d'autres threads non critiques. La deuxième autorise l'exécution de 1, 2 ou 4 threads critiques (en considérant 4 threads co-exécutables), le nombre de threads critiques est prévu pour être modifiable par le système quand le pipeline est vide.

Sur ces deux architectures, le calcul de WCET pour chaque thread critique sera effectué grâce à la méthode des graphes d'exécution paramétrés. La prédictibilité de nos architectures rendant les calculs aussi aisés que dans un cadre où un seul programme s'exécute.

On peut noter que la deuxième architecture configurée en mode m_1 est équivalente à la première. La différence étant que la première architecture est optimisée pour un

thread temps-réel grâce au mécanisme Replay qui n’est pas utilisable si plusieurs threads critiques sont présents. Dans la suite, nous ne considérerons que l’architecture prévue pour plusieurs threads, les résultats pour un seul thread critique étant les mêmes sur les deux architectures.

5 Méthodologie d’analyse des performances

Nous allons maintenant parler de la méthodologie nous permettant d’étudier les performances de notre architecture prévisible (celle permettant l’exécution de plusieurs threads critiques). Nous décrivons d’abord les différentes configurations architecturales que nous testons, pour l’architecture de base (non prédictible) et notre architecture prévisible. Seront ensuite présentés les programmes de test et l’environnement de simulation.

5.1 Configuration de l’architecture

Dans nos expérimentations, nous considérons deux architectures : (a) le cœur *de base*, qui implémente un ordonnancement de type O-RR pour toutes les ressources excepté pour l’envoi aux unités fonctionnelles où les instructions les plus vieilles sont sélectionnées d’abord, quel que soit le thread auquel elles appartiennent ; (b) l’architecture *adaptée au WCET* que nous avons conçue pour être prévisible temporellement et qui implémente notre politique BP. Même avec l’architecture de base (non prédictible), les ressources de stockage sont toujours partitionnées en 4, permettant

ainsi l'exécution de 4 threads et la comparaison avec l'architecture prédictible (les autres caractéristiques restant identiques).

Nous avons dérivé notre architecture SMT prévisible d'un cœur superscalaire de degré 4 (chaque étage a une largeur de 4 instructions), nous faisons également des mesures avec un pipeline de largeur 8. Le nombre de threads simultanés a été fixé à 4. Nous avons considéré un prédicteur de branchements parfait (oracle).

Du Tableau 2 au Tableau 5 on donne les principales caractéristiques du pipeline.

Les caches de données et d'instructions sont toujours parfaits (succès systématiques avec latence de 1 cycle), la prédiction de branchement est elle aussi supposée parfaite. Nous n'avons pas, dans cette partie, modélisé les caches, en particulier les interférences entre threads. Nous prenons donc des hypothèses optimistes sur le nombre maximum d'accès en cours, n'importe quel thread peut accéder au cache à n'importe quel moment sans subir d'autre pénalité que la latence des caches, on utilise donc un cache non bloquant à nombre de requêtes simultanées illimité. Nous sommes sûrs ainsi qu'aucun thread ne subira d'interférences de la part des autres au niveau du cache. La lecture d'instruction est toujours alignée sur les frontières de lignes de cache (on considère des lignes de 8 instructions), c'est-à-dire que, pendant un cycle, on ne peut lire que depuis une seule ligne de cache à la fois, la lecture s'arrêtant à chaque instruction de saut ou frontière de ligne de cache.

Les unités fonctionnelles sont toutes pipelinées, ce qui évite la prise en compte de délais pessimistes pour le calcul de WCET. Les files sont partitionnées statiquement en 4 car on peut gérer jusqu'à quatre threads simultanés, l'exécution peut se faire suivant quatre modes différents, correspondant à 0, 1, 2 ou 4 threads critiques. Les partitions (de bande passante et de stockage) allouées à un thread critique (modes m_1, m_2 et m_4) sont données du Tableau 2 au Tableau 5. Il faut noter que le processeur ne peut changer de mode que lorsque les quatre « emplacements » pour thread sont libres (quand aucun thread ne s'exécute).

Pour chaque largeur du pipeline, on teste trois configurations différentes en faisant varier les quantités d'unités fonctionnelles. On note que chaque configuration est désignée par un mnémonique rappelant la largeur du pipeline et la disposition des unités fonctionnelles.

Paramètre	*Total*	*Ressources allouées à chaque thread critique*		
		mode m_1	*mode m_2*	*mode m_4*
Nombre maximum de threads	4			
Bande passante lecture (fetch)	8 inst./cycle	8 inst. tous les 4 cycles		
Bande passante autres étages (DI, RN, CM)	4 inst./cycle	4 inst./cycle	2 inst./cycle	1 inst./cycle
Taille file de lecture (FQ)	16 entrées	16 (à partager netre les threads exécutés)		
Taille file de décodage (DQ)	16 entrées	16 (à partager netre les threads exécutés)		
Taille du ROB	256 entrées	256 (à partager netre les threads exécutés)		

Tableau 2. Caractéristiques générales de l'architecture (pipeline 4 voies)

Config.	*Paramètre*	*Nombre*	*Ressources allouées à chaque thread critique*		
			mode m_1	*mode m_2*	*mode m_4*
UF4-2421	UFs (*latence*)				
	memoire (*1 cycle*)	2	4	2	1
	ALU entière (*1 cycle*)	4	2	1	1 cycle sur 2
	FALU (*3 cycles*)	2	1	1 cycle sur 2	1 cycle sur 4
	diviseur (*15 cycles*)	1	2	1	1 cycle sur 2
UF4-1111	UFs (*latence*)				
	memoire (*1 cycle*)	1	1	1 cycle sur 2	1 cycle sur 4
	ALU entière (*1 cycle*)	1	1	1 cycle sur 2	1 cycle sur 4
	FALU (*3 cycles*)	1	1	1 cycle sur 2	1 cycle sur 4
	diviseur (*15 cycles*)	1	1	1 cycle sur 2	1 cycle sur 4
UF4-4444	UFs (*latence*)				
	memoire (*1 cycle*)	4	4	2	1
	ALU entière (*1 cycle*)	4	4	2	1
	FALU (*3 cycles*)	4	4	2	1
	diviseur (*15 cycles*)	4	4	2	1

Tableau 3. Configuration des unités fonctionnelles (pipeline 4 voies)

Paramètre	*Total*	*Ressources allouées à chaque thread critique*		
		mode m_1	*mode m_2*	*mode m_4*
Nombre maximum de threads	4			
Bande passante lecture (fetch)	8 inst./cycle	8 inst. tous les 4 cycles		
Bande passante autres étages (DI, RN, CM)	8 inst./cycle	8 inst./cycle	4 inst./cycle	2 inst./cycle
Taille file de lecture (FQ)	32 entrées	32 (à partager netre les threads exécutés)		
Taille file de décodage (DQ)	32 entrées	32 (à partager netre les threads exécutés)		
Taille du ROB	256 entrées	256 (à partager netre les threads exécutés)		

Tableau 4. Caractéristiques générales de l'architecture (pipeline 8 voies)

Config.	*Paramètre*	*Nombre*	*Ressources allouées à chaque thread critique*		
			mode m_1	*mode m_2*	*mode m_4*
UF8-2822	UFs (*latence*)				
	memoire (*1 cycle*)	2	2	1	1 cycle sur 2
	ALU entière (*1 cycle*)	8	8	4	2
	FALU (*3 cycles*)	2	2	1	1 cycle sur 2
	diviseur (*15 cycles*)	2	2	1	1 cycle sur 2
UF8-1211	UFs (*latence*)				
	memoire (*1 cycle*)	1	1	1 cycle sur 2	1 cycle sur 4
	ALU entière (*1 cycle*)	2	2	1	1 cycle sur 2
	FALU (*3 cycles*)	1	1	1 cycle sur 2	1 cycle sur 4
	diviseur (*15 cycles*)	1	1	1 cycle sur 2	1 cycle sur 4
UF8-4844	UFs (*latence*)				
	memoire (*1 cycle*)	4	4	2	1
	ALU entière (*1 cycle*)	8	8	4	2
	FALU (*3 cycles*)	4	4	2	1
	diviseur (*15 cycles*)	4	4	2	1

Tableau 5. Configuration des unités fonctionnelles (pipeline 8 voies)

5.2 Programmes de test

Un jeu de test est composé de 2 ou 4 threads compilés dans un même code binaire (ceci parce que notre simulateur ne peut gérer qu'un seul espace d'adressage). Le code source de chaque thread est encapsulé dans un appel de fonction et le compteur de

programme correspondant est initialisé au point d'entrée de la fonction. Pour maximiser les interférences possibles entre threads (et donc nous situer dans un contexte défavorable), nous avons considéré des threads concurrents exécutant la même fonction. Les différentes fonctions utilisées au cours des tests sont listées dans le Tableau 6, leur code source provient de la suite SNU-RT [65] et des programmes de test maintenus par l'équipe d'analyse de WCET de l'université de Mälardalen [67], des collections de tâches relativement simples exécutées couramment sur des systèmes embarqués temps-réel strict.

La particularité des tâches de cette suite est de ne pas présenter d'appels système. En effet, le temps d'exécution d'un appel système est souvent difficile à borner (un accès à une interface réseau par exemple) car il dépend de bien plus de facteurs que l'architecture du processeur. C'est pour cette raison qu'on ne considère pas les appels système pour l'étude des temps d'exécution et en particulier du WCET de tâches critiques.

Fonction	***Description***
fir	Filtre FIR avec générateur de nombres gaussiens
ludcmp	Décomposition LU
lms	Amélioration adaptative de signal LMS
fft1k	FFT (transformée de Fourier) sur des tableaux de 1000 nombres complexes
edn	Calculs à l'aide de filtres FIR
jfdctint	Transformée cosinus discrètes sur des blocs 8x8
minver	Inversion d'une matrice en nombres flottants
nsichneu	Simulation d'un réseau de Petri étendu

Tableau 6. Fonctions de test

Pour avoir des résultats significatifs, nous avons légèrement modifié certains programmes de la suite maintenue par l'université de Mälardalen en augmentant la taille des données. Les nombres d'instructions indiqués ci-dessous correspondent aux nombres d'instructions exécutées pendant le déroulement du programme (une même instruction du code exécutable pouvant être exécutées plusieurs fois).

- **fir** : ce programme, qui réaliste un filtrage FIR, comporte deux boucles consécutives qui lancent des calculs sur une série de nombres. Chaque boucle itère 200 fois pour un total de 325 408 instructions exécutées.

- **ludcmp** : ce programme réalise une décomposition LU de matrices de taille 50×50 (avec 562 200 instructions exécutées)
- **lms** : amélioration adaptative de signal LMS, le signal d'entrée étant une sinusoïde avec du bruit blanc rajouté. Ce programme exécute 225 903 instructions
- **fft1k** : c'est le programme le plus long et il compte 704 680 instructions exécutées. Il réalise un calcul de transformée de Fourier rapide.
- **edn** : ce programme effectue des calculs basés sur des filtrages FIR. Nous avons modifié le code original en englobant les appels de fonctions de calcul dans une boucle à 6 itérations (on refait donc les mêmes calculs plusieurs fois d'affilée) avec 304 464 instructions au total.
- **jfdctint** : ce programme fait des transformations cosinus discrètes sur des blocs de 8 pixels de large pour le codage d'ne image jpeg, le calcul est répété 300 fois, ce qui fait 380 477instructions exécutées.
- **minver** : ce programme inverse une matrice carrée de dimension 30×30, avec332 093 instructions exécutées.
- **nsichneu** : il calcule 120 états successifs d'un réseau de Petri avec 341 647 instructions exécutées.

Tous les exécutables ont été compilés avec `gcc` et l'option `-O` (équivalent de –O1)qui enlève la plupart des accès inutiles à la mémoire tout en conservant la structure algorithmique du code (ce qui permet d'effectuer l'analyse de flot sur le code source).

5.3 Simulation

Nous avons mené les expérimentations rapportées à l'aide d'un simulateur de niveau cycle développé dans le cadre de la plateforme OTAWA [8]. OTAWA est destinée au calcul de WCET et comprend de nombreux utilitaires dont un simulateur de niveau cycle construit sur SystemC. Le simulateur modélise des architectures génériques de processeurs où chaque étage de pipeline est vu comme un composant qui lit des instructions à partir d'une file d'entrée, les traite en appliquant un ensemble d'opérations prédéfinies, et les insère dans une file de sortie, ou bien traite des instructions présentes dans un tampon. Chaque file est paramétrée par une politique de distribution et une politique d'ordonnancement. Ce simulateur temporel est piloté par un

simulateur fonctionnel généré automatiquement à partir de la description d'un jeu d'instructions (dans ce mémoire, on considère le jeu PowerPC) grâce à notre outil GLISS [64].

Au début de ce travail, le simulateur existant ne modélisait que des architectures mono-thread. Il a donc fallu apporter des modifications importantes pour lui permettre de gérer l'architecture multi-flots que nous proposons. Le choix de développer nous-mêmes un simulateur pour ce type d'architecture plutôt que d'utiliser un outil existant, comme smt-sim [57] a été dicté par la nécessité d'avoir un modèle de simulation cohérent avec le modèle utilisé pour le calcul de WCET. Dans OTAWA, c'est la même description de l'architecture qui est utilisée pour générer ces deux modèles.

Les modifications apportées sont les suivantes. Tout d'abord, l'état architectural géré par le simulateur fonctionnel a dû être dupliqué pour que chaque thread dispose de son jeu de registres logiques privés. La mémoire est partagée entre les threads, comme sur les processeurs SMT réels. Les modules SystemC qui composent le simulateur structurel ainsi que les signaux de communication entre ces modules ont été modifiés pour permettre à chaque étage de traiter des instructions de différents threads à un cycle donné. Il a fallu également implémenter les politiques d'ordonnancement et de partage des ressources de stockage.

Le simulateur exécute les tâches de manière cyclique (toute tâche qui se termine est relancée immédiatement), ceci afin de maintenir la pression sur les threads non critiques tout au long de leur exécution. Les temps donnés comme résultats correspondent à la fin de la première exécution de la tâche considérée.

5.4 Calcul de WCET

OTAWA offre toutes les fonctionnalités nécessaires à un calcul de WCET par analyse statique. Tout d'abord, un chargeur de code objet permet de lire des codes binaires (plusieurs jeux d'instructions, dont celui du PowerPC, sont supportés) et d'en construire une représentation sous forme de graphe de flot de contrôle. Il est ensuite possible de lancer un processeur de code (parmi ceux qui sont disponibles) qui évalue le coût pire-cas de chaque bloc de base et annote le CFG avec ces valeurs. Dans ce travail,

nous avons utilisé un processeur de code qui implémente la méthode basée sur les graphes d'exécution paramétrés [50] : il recense toutes les séquences de blocs de base à analyser, construit les graphes d'exécution correspondants et calcule une borne supérieure du coût de chaque bloc.

OTAWA dispose également d'un module qui implémente la méthode IPET de calcul de WCET : il peut lire des informations relatives au flot de contrôle (bornes de boucles, …) et construire le programme linéaire nécessaire à la recherche du temps d'exécution pire cas. Toutefois, nous n'avons pas utilisé ce module pour produire les résultats présentés dans ce mémoire. En effet, nous avons souhaité évaluer la surestimation liée uniquement au calcul des coûts de blocs et nous affranchir de la surestimation due à des informations insuffisantes, ou insuffisamment exploitées, sur le contrôle de flot (bornes de boucles surévaluées ou prise en compte de manière peu précise, comme c'est parfois le cas en présence de boucles imbriquées). Aussi, les surestimations présentées dans ce document ont été calculées de la manière suivante :

- nous avons tout d'abord simulé l'exécution de chaque application et enregistré le nombre d'exécution (x_i) de chaque bloc de base et son coût d'exécution observé maximum (c_i^o)
- nous avons ensuite calculé de manière statique le coût d'exécution pire cas de chaque bloc (c_i^e)
- la surestimation du WCET du programme a été calculée par : $x_i.(c_i^e - c_i^o)$. Il s'agit de la moyenne des surestimations des blocs de base pondérées par leur nombre d'exécutions.

5.5 Conclusion

Nous venons de décrire notre plateforme de test qui nous permettra de mesurer les temps d'exécution de nos programmes. A partir de ces temps, nous allons vérifier si les threads temps-réel strict ont bien un comportement prédictible. En théorie leur exécution devrait se faire sans dépendre de la nature des autres threads, critiques ou non.

Nous pouvons également mesurer l'impact que ces mécanismes de prédictibilité ont sur les threads non critiques, en effet nous ne voulons pas pénaliser ceux-ci de

manière disproportionnée pour assurer un comportement prédictible pour nos threads critiques. Les nombreuses configurations architecturales testées nous permettront de suivre l'évolution des performances en fonction des quantités d'unités fonctionnelles et des tailles des files.

Enfin, pour juger de l'intérêt de notre architecture prévisible, les WCET des threads critiques peuvent également être calculés.

6 Performances d'une architecture SMT prévisible

Après avoir présenté nos architectures prédictibles et notre méthodologie de simulation, nous allons maintenant passer aux mesures permettant d'évaluer les performances de nos architectures.

Pour rappel, comme notre architecture pour un seul thread critique est équivalente à notre architecture pour plusieurs threads critiques où un seul thread critique s'exécute, nous ne testons que cette dernière architecture.

Après avoir donné les performances pour les threads critiques et non critiques, les WCET des threads critiques sont calculés et commentés.

6.1 Temps d'exécution de référence

Nous avons tout d'abord mesuré le temps d'exécution des programmes test sur l'architecture de base non prévisible, dont les caractéristiques sont celles indiquées dans le chapitre précédent. Dans cette architecture, les files d'instructions sont partitionnées statiquement et, dans chaque étage du pipeline, les threads sont ordonnancés selon un

algorithme Round Robin O-RR (lorsqu'un thread n'est pas prêt, c'est le thread suivant qui est traité).

Ces temps, que nous désignerons par la suite comme *temps d'exécution de référence*, sont donnés dans le Tableau 7 (pour quatre threads co-exécutés, le temps indiqué correspondant au dernier thread terminé) et le Tableau 8 (un thread exécuté seul). Ils serviront à évaluer le coût de la prévisibilité pour notre architecture. La signification des différentes configurations est donnée dans le Tableau 5 page 68.

	Pipeline 4 voies			***Pipeline 8 voies***		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	309 679	322 889	309 343	233 594	265 275	228 476
fft1k	768 511	1 219 587	744 595	670 715	1 219 527	568 699
fir	356 315	434 479	356 315	314 423	423 763	314 423
jfdctint	382 102	384 886	382 102	216 791	313 967	216 791
lms	250 135	313 599	250 135	223 811	313 207	223 811
ludcmp	607 133	606 585	607 133	504 619	514 567	504 619
minver	340 855	346 008	340 855	271 981	281 343	270 575
nsichneu	433 947	478 023	433 947	433 939	477 995	433 939

Tableau 7. Temps d'exécution de référence (en nombre de cycles) de 4 threads co-exécutés

	Pipeline 4 voies			***Pipeline 8 voies***		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	227 788	256 520	227 788	227 716	228 460	227 716
fft1k	566 436	568676	566 436	566 436	568 668	566 436
fir	314 420	314 420	314 420	314 420	314 420	314 420
jfdctint	216 788	312 268	216 788	216 788	216 788	216 788
lms	223 808	223 808	223 808	223 808	223 808	223 808
ludcmp	504 616	504 616	504 616	504 616	504 616	504 616
minver	270 572	270 572	270 572	270 572	270 572	270 572
nsichneu	433 936	433 936	433 936	433 936	433 936	433 936

Tableau 8. Temps d'exécution de référence (en nombre de cycles) d'un thread exécuté seul

6.2 Temps d'exécution observés sur l'architecture prévisible

L'architecture prévisible met en œuvre des politiques d'ordonnancement destinées à permettre d'estimer de manière à la fois fiable et précise le temps d'exécution des threads critiques.

Nous avons mesuré les temps d'exécution de threads critiques et non critiques sur cette architecture. Dans tous les cas, nous avons considéré l'exécution concurrente de quatre threads, le nombre de threads critiques variant selon le mode d'exécution (m1, m2 ou m4). A chaque fois, c'est le temps d'exécution du dernier des threads critiques ou non critiques qui est indiqué. On rappelle que, comme les tâches sont exécutées de manière cyclique (une tâche terminée est immédiatement relacée, de sorte qu'il y a en permanence quatre tâches actives), les temps correspondent à la première exécution de chaque tâche.

6.2.1 Temps d'exécution des threads critiques

Le Tableau 9 donne le temps d'exécution d'un thread critique exécuté en mode m1. On peut remarquer que les temps d'exécution ne varient pas beaucoup entre les différentes configurations. En effet, la lecture d'instruction est l'élément le plus contraignant de l'architecture. Le fait de lire des instructions pour le thread critique seulement tous les 4 cycles ne permet pas des performances optimales, mais, comme évoqué dans les sections précédentes, ce fonctionnement est nécessaire pour assurer la prédictibilité du thread critique.

On observe qu'à chaque fois la largeur du pipeline et les quantités d'unités fonctionnelles sont suffisantes pour permettre au thread critique de s'exécuter assez rapidement.

Les variations des temps d'exécution sont faibles mais on peut quand même constater que plus une configuration est « large » (en considérant la largeur du pipeline et le nombre d'unités fonctionnelles), plus l'exécution se fait rapidement. Ainsi la configuration UF4-1111 est toujours plus lente que la configuration UF4-2421 qui elle-même est toujours plus lente que la configuration UF-4444 (plus lente signifie pour nous que les temps d'exécution sont inférieurs ou égaux, ils sont d'ailleurs souvent égaux). Pour un pipeline de largeur 8, les configurations sont dans cet ordre-ci (de la plus lente à la plus rapide) : UF8-1211, UF8-2822 et UF8-4844.

Le fait le plus intéressant à constater est que ces temps sont égaux aux temps de référence présentés dans le Tableau 8. Ainsi, un thread critique s'exécute dans notre architecture prévisible sans subir d'interférences d'autres threads de la même manière que sur une architecture standard sur laquelle il s'exécuterait seul. Ceci confirme notre

hypothèse de départ : notre architecture permet d'assurer pour le thread critique un comportement prédictible et totalement indépendant des autres threads co-exécutés.

	Pipeline 4 voies			*Pipeline 8 voies*		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	227 788	256 520	227 788	227 716	228 460	227 716
fft1k	566 436	568 676	566 436	566 436	568 668	566 436
fir	314 420	314 420	314 420	314 420	314 420	314 420
jfdctint	216 788	312 268	216 788	216 788	216 788	216 788
lms	223 808	223 808	223 808	223 808	223 808	223 808
ludcmp	504 616	504 616	504 616	504 616	504 616	504 616
minver	270 572	270 572	270 572	270 572	270 572	270 572
nsichneu	433 936	433 936	433 936	433 936	433 936	433 936

Tableau 9. Temps d'exécution (en nombre de cycles) d'un thread critique exécuté sur l'architecture prévisible en mode m1

	Pipeline 4 voies			*Pipeline 8 voies*		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	0,74%	87,02%	0,27%	0,19%	19,93%	0,00%
fft1k	0,40%	28,37%	0,00%	0,39%	17,94%	0,00%
fir	0,00%	11,61%	0,00%	0,00%	0,00%	0,00%
jfdctint	0,00%	96,92%	0,00%	0,00%	0,55%	0,00%
lms	0,00%	15,96%	0,00%	0,00%	0,00%	0,00%
ludcmp	0,00%	44,98%	0,00%	0,00%	0,00%	0,00%
minver	0,00%	71,78%	0,00%	0,00%	0,52%	0,00%
nsichneu	0,00%	2,36%	0,00%	0,00%	0,00%	0,00%

Tableau 10. Augmentation du temps d'exécution d'un thread critique exécuté sur l'architecture prévisible en mode m2 (par rapport au mode m1)

	Pipeline 4 voies			*Pipeline 8 voies*		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	58,68%	374,64%	51,35%	8,10%	112,18%	0,71%
fft1k	67,86%	395,61%	43,76%	18,41%	114,58%	0,40%
fir	28,75%	313,88%	28,75%	0,00%	61,37%	0,00%
jfdctint	93,77%	387,27%	81,06%	8,29%	200,24%	0,00%
lms	30,31%	303,60%	30,31%	0,00%	49,37%	0,00%
ludcmp	39,93%	345,58%	39,93%	0,00%	51,23%	0,00%
minver	49,30%	390,82%	49,30%	0,52%	73,91%	0,00%
nsichneu	4,87%	214,86%	4,87%	0,00%	10,16%	0,00%

Tableau 11. Augmentation du temps d'exécution d'un thread critique exécuté sur l'architecture prévisible en mode m4 (par rapport au mode m1)

Le Tableau 10 et le Tableau 11 présentent les résultats obtenus pour les modes d'exécution m_2 et m_4.

Pour le mode m_2, on remarque que, sauf avec la configuration UF4-1111 et les programmes edn et fft1k en configuration UF8-1211, les threads critiques s'exécutent aussi vite (moins de 1% de différence) qu'en mode m_1. Avec la configuration UF4-1111, on se retrouve dans une situation où il n'y a plus suffisamment de ressources pour que les 2 threads critiques puissent s'exécuter à leur maximum (44,9% d'augmentation en moyenne).

Les deux programmes edn et fft1k sont plus sensibles que les autres aux différences de configuration. Leurs temps d'exécution augmentent en effet de manière plus sensible que ceux des autres programmes. Ces augmentations sont tout de même minimes, sauf avec la configuration UF8-1211 pour laquelle on peut noter des augmentations entre 15 et 20%.

En conclusion, le mode m_2 permet aux deux threads critiques de s'exécuter quasiment à la même vitesse qu'en mode m_1 si les ressources sont suffisantes. Des pertes de performances sont quand même constatées si les programmes exécutés disposent d'un certain degré de parallélisme d'instructions.

Le mode m_4 montre des baisses de performances notables pour toutes les configurations avec pipeline 4 voies. Les temps sur la configuration UF4-2421 (resp. UF4-1111, UF4-4444) augmentent en moyenne de 46,7% (resp. 340,8%, 41,2%). Cela

montre que cette largeur de pipeline n'est pas suffisante pour permettre d'exécuter quatre threads critiques à leur plein potentiel. Néanmoins, les configurations UF4-2421 et UF4-4444 ont des pertes pouvant être considérées comme acceptables, surtout si on ne passe pas la majorité du temps en mode m_4. De telles pertes sont le prix à payer pour profiter d'un parallélisme d'ordre 4 sur une architecture SMT.

Les configurations à pipeline 8 voies montrent des résultats plus intéressants. En effet, excepté les threads edn, fft1k et jfdctint, sur les configurations UF8-2822 et UF8-4844 les temps augmentent très peu (moins de 1% très souvent). Sur la configuration UF8-1211, on observe une augmentation moyenne de 84,1%. Cette configuration ne diffère de la configuration UF4-1111 que par la présence d'une ALU entière supplémentaire, ainsi que par la largeur du pipeline. Malgré la très forte similitude entre ces deux configurations, on voit que le simple fait d'augmenter la largeur du pipeline permet d'augmenter beaucoup moins les temps d'exécution (l'ajout seul d'une ALU en plus ne permet pas de baisser autant les temps d'exécution).

En résumé pour le mode m_4, les configurations à pipeline 4 voies montrent des augmentations modérées aux alentours de 40% (la configuration UF-1111 n'est pas considérée dans cette affirmation à cause de ses basses performances qui diminuent son intérêt). Les configurations à pipeline 8 voies sont elles bien plus intéressantes, hormis la configuration UF8-1211 qui montre une augmentation d'environ 80%, toutes les configurations permettent à la plupart des threads de s'exécuter aussi vite qu'en mode m_1. Sur ces configurations, les threads avec un certain degré de parallélisme souffrent quand même d'une pénalité (18,4% au plus dans nos tests).

6.2.2 Temps d'exécution des threads non critiques

Le coût de la prévisibilité pour les threads non critiques est donné dans le Tableau 12 et le Tableau 13. Ce coût correspond à l'augmentation du temps d'exécution par rapport au temps de référence (Tableau 7).

Dans le mode m_1 (m_4 n'est pas considéré car aucun thread non critique s'y exécute), on remarque que, à part avec les configurations à peu d'unités fonctionnelles (UF4-1111 et UF8-1211), les temps d'exécution restent modérément élevés comparés aux temps de référence (pénalité maximum de 20% en moyenne pour les configurations

UF4-*). On constate également que plus le pipeline est large, plus les threads non critiques arrivent à s'exécuter de la même manière que sur l'architecture de base non prédictible (pénalité moyenne hors UF8-1211 d'environ 6%).

En mode m_2, la situation est similaire mais les ordres de grandeur changent, en effet les pénalités augmentent logiquement car un deuxième thread critique se rajoute. Les configurations UF4-* (sans UF4-1111) donnent des pénalités entre 70 et 75% en moyenne, on trouve aussi entre 5 et 8% pour les configurations UF8-*.

Concernant les configurations UF4-1111 et UF8-1211, entre le peu de ressources disponibles et le fait que deux threads soient critiques et non plus un seul, on voit que les threads non critiques ont beaucoup de mal à s'exécuter, avec des accroissements du temps d'exécution pouvant dépasser les 5 000%. Certaines cases sont vides et représentent deux types de comportement non observés en mode m_1 : certains threads non critiques se retrouvent bloqués au bout d'un certain temps et ne progressent plus du tout (jfdctint et ludcmp), d'autres progressent encore mais de façon très lente (pouvant descendre jusqu'à seulement une dizaine d'instructions tous les 10 millions de cycles) comme edn, minver et nsichneu.

	Pipeline 4 voies			***Pipeline 8 voies***		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	72,07%	1873,31%	72,40%	47,39%	63,35%	42,89%
fft1k	10,64%	43,88%	13,91%	3,00%	43,78%	0,17%
fir	8,00%	13,06%	8,00%	0,00%	12,11%	0,00%
jfdctint	43,86%	2987,87%	44,00%	0,00%	17,62%	0,14%
lms	6,16%	1,77%	6,16%	0,00%	0,50%	0,00%
ludcmp	3,84%	3,64%	3,84%	0,00%	0,61%	0,00%
minver	14,50%	15,31%	14,50%	0,08%	2,00%	0,00%
nsichneu	0,47%	3,52%	0,47%	0,00%	3,51%	0,00%

Tableau 12. Coût de la prévisibilité pour un thread *non* critique exécuté sur l'architecture prévisible en mode m1

	Pipeline 4 voies			*Pipeline 8 voies*		
	UF4-2421	**UF4-1111**	**UF4-4444**	**UF8-2822**	**UF8-1211**	**UF8-4844**
edn	113,09%		108,62%	53,64%	5960,78%	42,95%
fft1k	49,86%	502,73%	36,46%	6,46%	317,07%	0,60%
fir	17,88%	151,50%	17,88%	0,00%	105,55%	0,00%
jfdctint	346,92%		346,92%	0,01%	244,47%	3,32%
lms	14,36%	181,73%	14,36%	0,00%	23,75%	0,00%
ludcmp	20,75%		20,75%	0,00%	2,16%	0,00%
minver	31,38%		31,38%	0,37%	5,09%	0,00%
nsichneu	0,00%		0,00%	0,00%	11,32%	0,00%

Tableau 13. Coût de la prévisibilité pour un thread *non* critique exécuté sur l'architecture prévisible en mode m2

6.3 Impact de la taille des files d'instructions sur les temps d'exécution

La taille des files d'instructions qui, rappelons-le, sont partitionnées de manière statique entre les threads, a probablement un impact sur les performances de l'architecture. Dans cette section, nous allons examiner les temps d'exécution mesurés en considérant différentes tailles de files, et ce pour les configurations d'unités fonctionnelles intermédiaires : UF4-2421 pour un pipeline à 4 voies et UF8-2822 pour un pipeline à 8 voies.

Par défaut, dans notre configuration de base avec pipeline 4 voies, la file de lecture (FQ) comporte 2^4 entrées (16), de même que la file de décodage (DQ). Le ROB contient, quant à lui, 2^8 entrées (256). Avec un pipeline de largeur 8, on passe à 2^5 entrées pour FQ et DQ, le ROB reste inchangé.

6.3.1 Files de taille infinie

Nous allons d'abord faire des expérimentations en accordant à chaque thread des files de tailles infinies, nous voulons ainsi avoir une idée du taux potentiel maximum d'utilisation des files pour chaque benchmark.

Les mesures obtenues sont résumées ci-après.

L'occupation de FQ et DQ par un thread critique est toujours inférieur à la largeur du pipeline (4 ou 8) divisée par le nombre de threads critiques (1, 2 ou 4). On remarque

également que chaque thread non critique occupe au maximum autant d'entrées que la largeur du pipeline (4 ou 8).

Quant à l'utilisation du ROB, si on ne considère pas edn, fft1k et jfdctint, celle-ci peut varier de moins de 10 jusqu'à plus de 90 et ne dépend pas de la criticité des threads. Pour edn, fft1k et jfdctint, l'utilisation est plus importante et plus variable, elle peut aller de 18 (minimum pour nos mesures) à plus de 100 000 et est souvent supérieur à 10 000.

Excepté ces trois derniers benchmarks, tous les autres ont un temps très voisin, voire identique, à celui obtenu avec des files bornées.

Les temps d'exécution en mode m_4 (temps du dernier thread) pour la configuration UF8-2822, ainsi que la différence par rapport aux temps obtenus avec des files bornées (comparaison avec le temps du dernier thread) sont données dans le Tableau 14. On note que seul edn, fft1k et jfdctint bénéficient de l'utilisation de files très grandes, dans des proportions toutefois modérées comparé aux quantités de ressources mises à disposition.

	Temps d'exécution	***Gain***
edn	219 127	10,99%
fft1k	566 439	15,55%
fir	314 423	0,00%
jfdctint	216 791	7,66%
lms	223 811	0,00%
ludcmp	504 619	0,00%
minver	270 575	0,52%
nsichneu	433 939	0,00%

Tableau 14. Gains en temps d'exécution avec files infinies, configuration UF8-2822, mode m4

Au vu des résultats concernant les files FQ et DQ, nous remarquons que, hors cas particuliers peu fréquents observés en bornant la taille des files (à 5 000 dans nos expériences), chaque thread occupe au maximum autant d'entrées que la largeur du pipeline, ceci quelque soit sa nature (critique ou non). C'est pourquoi nous avons choisi dès le début de ce mémoire de fixer la taille des files FQ et DQ à 16 entrées pour un pipeline de largeur 4 prévu pour 4 threads maximum. Pour un pipeline 8 voies, toujours avec 4 threads, nous avons choisi 32 entrées (8 entrées pour chaque thread).

6.3.2 Le Reorder Buffer

Nous nous intéressons maintenant au ROB. Comme vu précédemment, l'utilisation du ROB peut varier de 18 (minimum mesuré) à 180 357 (maximum mesuré). Nous rappelons que la taille par défaut est de 256 entrées.

Nous allons examiner les performances obtenues en faisant varier la taille du ROB de 32 à 1024. Les résultats sont présentés du Tableau 15 au Tableau 22, les pourcentages indiquent toujours le coût de la prévisibilité pour les threads critiques (temps comparés avec le temps d'un thread exécuté tout seul sur la même configuration). Pour les configurations avec ROB de 256 entrées qui servent de référence, nous rappelons les coûts que l'on peut trouver aux sous-sections précédentes.

On remarque que les gains apportés par un ROB de plus de 256 entrées sont très marginaux : un tel ROB ne permet pas de faire baisser significativement le coût de la prédictibilité. De même, on peut constater qu'un ROB de 128 entrées ne présente pas de différence notable, excepté en mode m_4 où le coût augmente parfois de façon un peu plus visible. Les tailles de ROB inférieures, quant à elles, présentent des dégradations de prédictibilité pouvant frôler les 80% pour des threads critiques. On peut même assister à des blocages ou des ralentissements importants des threads non critiques avec de tels ROB, ceci ne se produisant plus seulement avec des configurations pauvres en ressources.

On voit que, en général, pour un programme donné, la pénalité baisse avec l'augmentation de la taille du ROB ou de la largeur du pipeline. Pour les threads critiques elle est bien entendu plus élevée en mode m_4 qu'en mode m_2. Les threads non critiques ont eux des pénalités qui augmentent logiquement en passant du mode m_1 au mode m_2.

On peut noter des disparités entre les threads si on regarde la sensibilité à la quantité de ressources mise à disposition. Le programme nsichneu, par exemple, s'exécute toujours à la même vitesse quelque soit la taille du ROB. A l'inverse, edn diminue toujours son temps d'exécution quand les ressources augmentent, même si les différences sont relativement faibles. On peut citer des cas intermédiaires avec une augmentation suivie d'un palier, tel fft1k qui augmente rapidement ses performances pour se stabiliser à partir d'une taille de ROB de 128 entrées.

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	2,82%	1,07%	0,98%	0,74%	0,62%	0,38%
fft1k	7,08%	1,38%	0,77%	0,40%	0,39%	0,38%
fir	3,97%	0,03%	0,00%	0,00%	0,00%	0,00%
jfdctint	9,63%	8,26%	0,00%	0,00%	0,00%	0,00%
lms	3,97%	0,11%	0,00%	0,00%	0,00%	0,00%
ludcmp	3,83%	0,00%	0,00%	0,00%	0,00%	0,00%
minver	1,56%	0,40%	0,00%	0,00%	0,00%	0,00%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 15. Résultats pour la configuration UF4-2421, mode m2

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	57,60%	59,19%	59,31%	58,68%	58,05%	57,69%
fft1k	74,96%	86,31%	69,07%	67,86%	67,84%	67,81%
fir	45,56%	39,62%	29,49%	28,75%	28,75%	28,75%
jfdctint	79,66%	105,37%	102,63%	93,77%	92,64%	92,57%
lms	38,44%	37,76%	31,07%	30,31%	30,31%	30,31%
ludcmp	39,18%	39,93%	39,93%	39,93%	39,93%	39,93%
minver	49,30%	49,30%	49,30%	49,30%	49,30%	49,30%
nsichneu	4,87%	4,87%	4,87%	4,87%	4,87%	4,87%

Tableau 16. Résultats pour la configuration UF4-2421, mode m4

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	0,66%	0,76%	0,46%	0,19%	0,11%	0,00%
fft1k	5,93%	0,79%	0,40%	0,39%	0,39%	0,37%
fir	2,90%	0,00%	0,00%	0,00%	0,00%	0,00%
jfdctint	2,18%	1,24%	0,00%	0,00%	0,00%	0,00%
lms	2,20%	0,00%	0,00%	0,00%	0,00%	0,00%
ludcmp	3,60%	0,00%	0,00%	0,00%	0,00%	0,00%
minver	0,97%	0,00%	0,00%	0,00%	0,00%	0,00%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 17. Résultats pour la configuration UF8-2822, mode m2

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	8,81%	9,12%	8,36%	8,10%	7,61%	7,46%
fft1k	40,94%	42,98%	21,92%	18,41%	18,40%	18,37%
fir	24,57%	13,40%	1,01%	0,00%	0,00%	0,00%
jfdctint	26,13%	30,44%	16,60%	8,29%	0,54%	0,51%
lms	16,87%	9,24%	1,07%	0,00%	0,00%	0,00%
ludcmp	11,16%	0,00%	0,00%	0,00%	0,00%	0,00%
minver	4,48%	1,24%	0,53%	0,52%	0,50%	0,47%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 18. Résultats pour la configuration UF8-2822, mode m4

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	83,05%	72,75%	77,00%	72,07%	66,41%	64,15%
fft1k	très lent	48,39%	10,92%	10,64%	10,62%	10,61%
fir	104,98%	10,56%	8,02%	8,00%	8,00%	8,00%
jfdctint	7476,60%	108,49%	44,12%	43,86%	43,86%	43,86%
lms	95,65%	8,09%	6,48%	6,16%	6,16%	6,16%
ludcmp	81,83%	3,89%	3,84%	3,84%	3,84%	3,84%
minver	23,83%	14,57%	14,50%	14,50%	14,50%	14,50%
nsichneu	0,47%	0,47%	0,47%	0,47%	0,47%	0,47%

Tableau 19. Résultats pour la configuration UF4-2421, mode m1 (threads non critiques)

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	144,17%	113,41%	114,10%	113,09%	104,41%	113,53%
fft1k	très lent	313,02%	50,41%	49,86%	49,92%	50,00%
fir	409,46%	47,12%	17,89%	17,88%	17,88%	17,88%
jfdctint	15 319,24%	1 046,57%	346,92%	346,92%	346,92%	346,92%
lms	198,52%	34,84%	14,97%	14,36%	14,36%	14,36%
ludcmp	131,45%	21,09%	20,75%	20,75%	20,75%	20,75%
minver	528,05%	34,12%	31,38%	31,38%	31,38%	31,38%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 20. Résultats pour la configuration UF4-2421, mode m2 (threads non critiques)

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	63,39%	52,30%	50,78%	47,39%	44,32%	37,92%
fft1k	très lent	13,64%	3,03%	3,00%	3,01%	3,03%
fir	79,97%	-0,08%	0,00%	0,00%	0,00%	0,00%
jfdctint	bloqué	39,65%	0,00%	0,00%	0,00%	0,00%
lms	63,32%	-0,18%	0,00%	0,00%	0,00%	0,00%
ludcmp	69,32%	0,00%	0,00%	0,00%	0,00%	0,00%
minver	21,27%	0,04%	0,08%	0,08%	0,07%	0,08%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 21. Résultats pour la configuration UF8-2822, mode m1 (threads non critiques)

	Taille du ROB					
	32	**64**	**128**	**256**	**512**	**1024**
edn	80,65%	55,07%	55,66%	53,64%	47,06%	40,11%
fft1k	très lent	27,71%	6,54%	6,46%	6,46%	6,48%
fir	211,82%	2,98%	0,01%	0,00%	0,00%	0,00%
jfdctint	30 912,58%	1 584,18%	0,04%	0,01%	0,01%	0,02%
lms	128,36%	1,57%	0,00%	0,00%	0,00%	0,00%
ludcmp	71,30%	0,00%	0,00%	0,00%	0,00%	0,00%
minver	11,96%	0,44%	0,40%	0,37%	0,37%	0,38%
nsichneu	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%

Tableau 22. Résultats pour la configuration UF8-2822, mode m2 (threads non critiques)

6.4 Prise en compte des caches d'instructions et de données

Que ce soit en considérant un ou plusieurs threads critiques, nous n'avons jamais modélisé les caches jusqu'à présent. En effet, leur prise en compte pour le calcul du WCET n'est pas aisée et le partage dynamique entre tous les threads est totalement imprédictible.

Pour essayer de remédier à l'imprédictibilité du partage dynamique, nous allons essayer la même technique que pour les files de stockage, à savoir, le partage statique, appliqué aux caches.

Le cache est modélisé et simulé, ses caractéristiques sont données dans le Tableau 23. Chaque niveau de cache est partitionné en fonction du nombre de threads exécutés

(critiques ou pas). Une implémentation concrète pourrait se faire en ayant à tous les niveaux une associativité d'ensemble d'ordre 4 (pour 4 threads).

niveau	*nb lignes * taille de ligne - pénalité*	
	Instructions	***Données***
L1	128 * 8 octets – 1 cycle	128 * 8 o – 1 cyc.
L2	512 * 8 o – 10 cyc.	
L3	4 096 * 8 o – 70 cyc.	

Tableau 23. Caractéristiques du cache

Dans un cache réel se pose la question du nombre d'accès concurrents possibles. Pour la prévisibilité de notre architecture, il faut qu'il soit possible que tous les threads critiques puissent faire une demande sans être gênés par les autres threads (critiques ou pas). Dans nos expérimentations, nous supposons que le nombre maximum d'accès concurrents est suffisant pour ne pas handicaper les threads critiques.

Nous gardons donc les mêmes hypothèses optimistes sur le nombre maximum d'accès en cours que pour nos expérimentations avec des caches parfaits. On considère donc toujours un cache non bloquant à nombre de requêtes simultanées illimité. Nous garantissons encore ainsi qu'aucun thread ne subira d'interférences de la part des autres au niveau du cache.

Pour nos expérimentations, nous considérons toujours l'exécution de 4 threads en parallèle.

Dans le Tableau 24, on donne les temps observés sur une architecture standard (non prédictible et avec une configuration d'unités fonctionnelles UF4-2421) avec un ordonnancement Round-Robin pour les threads. La première colonne indique les valeurs pour un thread exécuté « tout seul » (avec trois autres threads « fantômes » qui ne font strictement rien si ce n'est réserver des partitions de ressources). La deuxième colonne donne le temps d'exécution du dernier thread terminé quand quatre copies du même thread sont co-exécutées.

Ces temps serviront plus loin de référence pour juger les performances des threads critiques.

Fonction	*Un thread tout seul*	*Dernier de 4 threads*
edn	230 516	311 764
fft1k	578 632	779 225
fir	316 000	357 783
jfdctint	217 352	383 263
lms	226 140	252 501
ludcmp	523 200	623 381
minver	285 264	384 390
nsichneu	456 336	456 347

Tableau 24. Temps d'exécution sur un cœur standard

Les résultats obtenus pour les modes d'exécution m_1, m_2 et m_4 sont donnés dans le Tableau 25 et dans le Tableau 26.

Chaque valeur du Tableau 25 concerne le temps d'exécution du thread critique qui s'est achevé en dernier tandis que les valeurs du Tableau 26 concernent les threads *non* critiques co-ordonnancés. Les pourcentages indiquent toujours l'augmentation du temps d'exécution comparé à celui d'un thread exécuté tout seul sur le cœur standard pour les threads critiques. Les threads non critiques sont encore comparés au dernier à s'exécuter parmi 4 (sur l'architecture non prédictible).

On peut constater les mêmes comportements qu'avec notre architecture pour plusieurs *hrt-t* sans cache.

Dans le mode m_1, comme prévu, les threads critiques s'exécutent aussi vite que s'ils étaient tout seuls car ils sont toujours prioritaires lors de l'accès à une ressource. Leurs temps d'exécution sont tous inférieurs à ce qui est obtenu sans les caches, ce qui montre bien l'avantage de l'utilisation des caches. On peut observer que la pénalité pour les threads non critiques reste modérée (20,6% en moyenne contre 19,9% sans caches).

La diminution de performance pour deux threads critiques s'exécutant en mode m_2 est faible, ils s'exécutent à peu près aussi vite que si ils étaient tout seuls car la quantité de ressource est suffisante. On peut quand même noter une très légère augmentation du coût de la prédictibilité (on passe de 0,1% en moyenne sans caches à 0,3%). La pénalité pour les threads non critiques reste, en moyenne, au même niveau que sans caches (si on ne tient pas compte de jfdctint qui est un cas extrême, on passe de 35% à 33%).

En mode m_4, la quantité de ressources allouées à chaque thread critique est inférieure à ses besoins et donc les temps d'exécution sont plus longs (de 45% en moyenne contre 47% sans les caches).

	Mode d'exécution		
	m_1	**m_2**	**m_4**
edn	-	0,78%	57,80%
fft1k	-	0,71%	65,46%
fir	-	0,00%	28,58%
jfdctint	-	0,15%	93,64%
lms	-	0,07%	30,12%
ludcmp	-	0,52%	36,37%
minver	-	0,40%	46,57%
nsichneu	-	0,00%	4,62%

Tableau 25. Coût de la prévisibilité pour les threads critiques

	Mode d'exécution		
	m_1	m_2	m_4
edn	71,89%	113,32%	-
fft1k	12,29%	31,43%	-
fir	7,63%	17,75%	-
jfdctint	44,23%	330,08%	-
lms	6,40%	10,15%	-
ludcmp	6,12%	20,91%	-
minver	15,71%	34,31%	-
nsichneu	0,47%	0,00%	-

Tableau 26. Coût de la prévisibilité pour les threads *non* critiques

On voit donc que l'utilisation de caches de données et d'instructions (correctement partagés) est possible avec notre architecture prédictible. On peut donc bénéficier des performances des caches tout en profitant d'une exécution prédictible pour des threads critiques.

6.5 Temps d'exécution pire cas (WCET) d'un thread critique

Nous avons modélisé l'architecture proposée ci-dessus (sans prise en compte des caches) pour évaluer les temps d'exécution pire cas (WCET) en utilisant l'approche décrite dans [50].

Cela consiste à représenter l'exécution d'un bloc de base comme un graphe d'exécution exprimant les dépendances entre les instructions traversant le pipeline. Ensuite le moment auquel chaque nœud du graphe peut être exécuté est calculé comme une fonction des moments auxquels les ressources sont libérées par les instructions précédentes. Le coup d'exécution d'un bloc peut être borné en analysant les temps d'exécution respectifs du bloc de base et du dernier nœud du bloc précédent. Plus d'informations peuvent être trouvées dans l'article d'origine.

Naturellement, les coûts d'exécution estimés de cette manière sont surestimés à cause des hypothèses pessimistes qui sont faites sur le contexte dans lequel un bloc de base s'exécute (c'est-à-dire l'état du pipeline, qui dépend du chemin exécuté avant le bloc). En fait le coup d'un bloc est évalué en considérant tous les états possibles du pipeline alors que seulement quelques uns de ces états peuvent se rencontrer dans une exécution réelle. Néanmoins la méthode est simple et permet de donner des WCET *sûrs*.

Les WCET trouvés pour nos programmes de test en considérant notre architecture prédictible sont donnés dans le Tableau 27. Les pourcentages montrent la différence entre les WCET estimés et les temps observés (voir Tableau 9). La surestimation peut sembler élevée mais elle est du même ordre que ce qui est souvent observé avec des outils d'estimation de WCET pour processeur mono-thread (voir les résultats du *2006 WCET tool challenge* [22]).

	Mode d'exécution		
	m_1	m_2	m_4
edn	+19,2 %	+25,2 %	+33,1 %
fft1k	+ 38,2 %	+ 63,8 %	+ 69,8 %
fir	+ 44,3 %	+ 59,2 %	+ 63,5 %
jfdctint	+ 29,1 %	+ 34,2 %	+ 33,1 %
lms	+ 53,1 %	+ 66,8 %	+ 70,6 %
ludcmp	+ 22,7 %	+ 41,8 %	+ 53,8 %
minver	+ 33,6 %	+ 38,4 %	+ 38,8 %
nsichneu	+38,9 %	+52,3 %	+65,8 %

Tableau 27. Temps d'exécution pire cas (WCET) d'un thread critique en mode m1

6.6 Conclusion

6.6.1 Contributions et bilan

Nous avons proposé une architecture SMT prédictible qui supporte plusieurs threads critiques. Grace à l'implémentation de politiques prédictibles pour contrôler le

partage des ressources parmi les threads, il est possible de borner le temps d'exécution des threads qui ont des échéances strictes. Les ressources de stockage sont partitionnées statiquement et l'ordonnancement bas-niveau se fait en utilisant une stratégie *Bandwidth Partitioning* (partitionnement de bande passante) qui donne la priorité aux threads temps-réels pour l'accès aux parts privées de ressources.

L'architecture supporte quatre modes d'exécution qui permettent respectivement une exécution prédictible pour zéro, un, deux ou quatre threads critiques (les autres threads étant non critiques).

Les résultats expérimentaux montrent que deux threads critiques peuvent être supportés avec une perte modérée de performance, les threads critiques s'exécutent presque comme si ils étaient tout seuls dans le pipeline et les autres threads voient leur temps d'exécution augmenter de 20% à 40% (c'est le prix à payer pour la prédictibilité). L'exécution de quatre threads en parallèle dégrade naturellement les performances respectives de ceux-ci à cause du côté strict de la politique d'allocation de ressources nécessaire pour assurer le déterminisme. Toutefois le coût peut être acceptable dès qu'un parallélisme de threads est nécessaire. A partir d'un exemple simple, nous avons donné un aperçu de la manière dont un ordonnanceur temps-réel hors-ligne pouvait tirer avantage des modes d'exécutions variés supportés par l'architecture. En fonction des temps respectifs des tâches à ordonnancer, l'ordonnanceur peut sélectionner le mode d'exécution approprié pour assurer le respect des échéances des tâches

6.6.2 Possibilités de l'architecture

Dans cette section, nous comptons montrer comment les multiples modes d'exécution de notre architecture (liés au nombre de *hrt-t* dont on garantit le comportement prédictible) pourraient être utilisés pour ordonnancer des tâches temps-réel strict.

Nous illustrerons ceci en considérant l'ensemble de tâches PapaBench [43]. PapaBench est une collection de programmes de test conçue pour être utilisée dans la recherche académique sur l'estimation de WCET. Cette collection est basée sur le logiciel Paparazzi [66] développé pour contrôler des drones (ou UAV, *unmanned aerial vehicle*). Nous allons maintenant considérer un sous-ensemble des tâches PapaBench v.2 listées dans le Tableau 28. Les valeurs indiquées sont leurs WCET

estimés (pipeline de largeur 4, configuration 2 des unités fonctionnelles) sur notre architecture prédictible.

		WCET suivant le mode d'exécution		
N°	***Nom de la tâche***	***m_1***	***m_2***	***m_4***
t7	stabilization	217	223	297
t9	parse_gps_msg	107	127	222
t10	send_gps_pos	949	962	1160
t11	send_radIR	397	398	456
t12	send_takeOff	184	186	222
t13	navigation_update	886	941	1205
t14	course_run	139	148	171
t15	send_nav_values	513	513	599
t16	altitude_control	68	69	87
t17	climb_control	190	198	250

Tableau 28. Tâches de PapaBench

Toutes ces tâches sont répétées avec la même fréquence, exceptée pour la tâche t7 qui doit être lancée à une fréquence cinq fois supérieure. Les dépendances entre tâches sont indiquées dans la Figure 8.

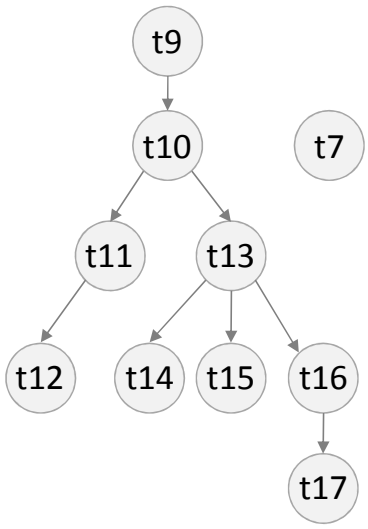

Figure 8. Dépendances entre les tâches de PapaBench

Comme toutes les tâches ont des échéances strictes, elles doivent être exécutées dans un mode prédictible (m_1, m_2 ou m_4). Si la longueur d'un cycle est trop courte, comme dans la Figure 9, il n'est pas possible d'ordonnancer toutes les tâches en mode m_1. Le mode m_2 devrait être sélectionné au début du cycle puisque le parallélisme à

l'intérieur de l'ensemble de tâches est limité par les dépendances (on peut noter que deux emplacements libres peuvent être alloués à des threads non critiques pouvant être préemptés). Néanmoins le temps restant après que la tâche t13 finisse est trop court pour exécuter les dernières tâches par paires. Un passage en mode m_4 rend possible l'ordonnancement d'au plus quatre tâches en parallèle avec une prédictibilité temporelle garantie.

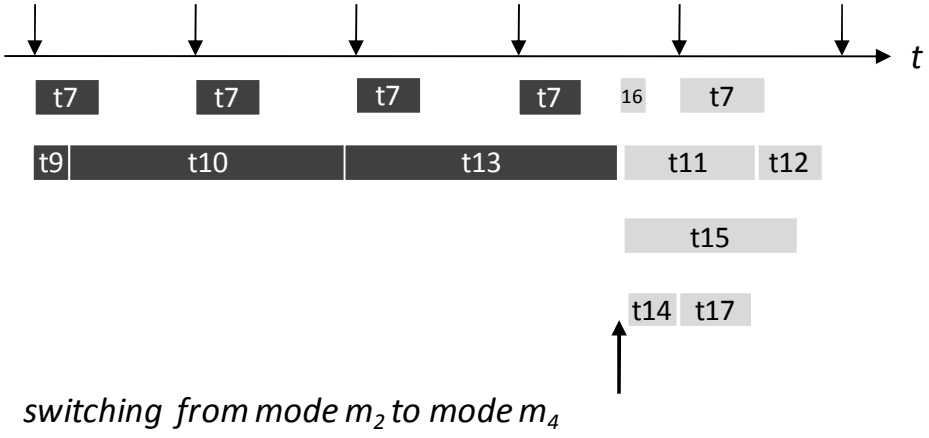

Figure 9. Un ordonnancement possible des tâches PapaBench

L'exemple simple ci-dessus montre comment les modes d'exécution prédictible pourraient être utilisés pour produire des ordonnancements respectant les échéances. Nous avons généré l'ordonnancement manuellement mais nous pensons que ces modes d'exécution pourraient être exploités efficacement par un ordonnanceur de tâches temps-réel hors-ligne. L'idée que nous proposons se rapproche de cas similaires d'étude pour concevoir des techniques d'ordonnancement temps-réel pour les processeurs bénéficiant du *Dynamic Voltage Scaling* [21].

6.6.3 Perspectives

Nous pensons que notre architecture peut être améliorée. En effet, nous ne traitons pas par exemple la prédiction de branchement. Le partage de la logique de prédiction entre les threads peut rendre difficile la prise en compte de cet élément pour le calcul de WCET sur notre architecture. Ce partage rendrait encore plus complexe une situation déjà compliquée à traiter dans le cas monothread [8].

Selon nous, la gestion des caches peut être perfectionnée. Un simple partage à part égales entre tous les threads, quelque soit leur degré de criticité, semble un peu trop simpliste. D'autres stratégies plus élaborées doivent être étudiées pour pouvoir améliorer les performances tout en assurant la prédictibilité nécessaire au calcul de WCET.

7 Conclusion

Dans ce travail, nous avons conçu une architecture SMT permettant l'exécution déterministe d'un seul ou de plusieurs threads critiques. Pour permettre ceci, nous avons implémenté de nouvelles politiques d'ordonnancement des ressources processeur. En effet, nous avons remarqué que les politiques existantes étaient imprédictibles concernant le comportement d'un thread particulier.

Nous avons donc implémenté une politique de partage de bande passante, que nous appelons BP (*Bandwidth Partitioning*). Pour la distribution des ressources nous avons retenu une politique existante qui est le partage statique, où chaque thread se voit attribuer une part égale de ressources, l'utilisation de chaque partition étant interdite aux autres threads.

Avec cette nouvelle politique BP, la bande passante de chaque étage et l'accès à chaque unité fonctionnelle est partagée de façon équitable entre chaque thread critique. Les threads non critiques pouvant profiter de la bande non utilisée par les threads critiques, ce qui leur permet de progresser eux aussi.

Nous avons observé que l'exécution d'un thread critique donné ne dépend pas des autres threads co-exécutés mais uniquement du nombre de threads critiques et de la

"place" du thread dans le processeur (ou *slot* en anglais). Cela nous permet, pour nos calculs de WCET, de ramener à un cas similaire à un cas monothread.

Nous avons utilisé également une nouvelle modélisation des conflits de ressources internes à un processeur qui nous sert pour l'étape d'analyse de bas niveau pour calculer de manière plus fine les temps des blocs de base. Ce qui permet à la méthode IPET de nous fournir des WCET modéremment surestimés

Après avoir testé diverses modifications sur les quantités de ressources internes, nous avons noté que la rigidité de nos mécanismes de prédictibilité se traduit par une faible évolution des performances. Ce point, couplé à des pertes modérées pour les threads non critiques, constitue le prix à payer pour bénéficier d'une exécution parallèle prédictible pour des threads critiques.

Nous avons abordé le problème du partage des caches sur les architectures SMT. Une solution basique a été étudiée, le partage statique des caches. On observe les mêmes propriétés sur l'exécution des threads critiques que sur notre architecture sans caches. Des WCET sont calculés, toujours avec la même méthode, et donnent des surestimations acceptables.

Notre architecture peut fonctionner suivant différents modes, chacun correspondant à un nombre de threads critiques parmi les 4 threads pouvant s'exécuter. Le changement de mode se faisant quand le processeur est vide (aucun thread en exécution). Ces différents modes de fonctionnement peuvent être utilisés pour l'ordonnancement de tâches temps-réel sur notre architecture, les changements de mode permettant d'adapter la répartition de la bande passante à la charge critique du processeur.

Il reste néanmoins certains aspects architecturaux ignorés par cette étude, tel par exemple la gestion du partage de la logique de prédiction de branchement qui n'a pas été abordé. La gestion du partage du cache peut aussi être améliorée en examinant d'autres méthodes peut-être plus performantes tout en restant prédictible.

8 Bibliographie

[1] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm. “Cache behaviour prediction by abstract Interpretation”. *Static Analysis Symposium (SAS'96)*, 1996.

[2] Jonathan Barre, Christine Rochange, Pascal Sainrat. “Une architecture SMT pour le temps-réel strict”, *Symposium sur les Architectures Nouvelles de Machines (SympA 2008)*.

[3] Jonathan Barre, Christine Rochange, Pascal Sainrat. “A Predictable Simultaneous Multithreading Scheme for Hard Real-Time”. *International Conference on Architecture of Computing Systems, 2008*, LNCS 4934, p. 161-172, février 2008 (Best Paper).

[4] G. Bernat, A. Burns. “An approach to symbolic worst-case execution time analysis”. *25th Workshop on Real-Time Programming*, 2000.

[5] G. Bernat, A. Colin, and S.M. Petters. “pwcet : a tool for probabilistic worst case execution time analysis of real-time systems”. *Technical Report YCS353*, University of York, Department of Computer Science, april 2003

[6] G. Bernat, A. Colin, and S.M. Petters. “Wcet Analysis of Probabilistic Hard Real-Time Systems”. In Proceedings of the *23th IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, december 2002.

[7] J. R. Bulpin, I. A. Pratt. “Multiprogramming Performance of the Pentium 4 with Hyper-Threading”. *Workshop on Duplicating, Deconstruction and Debunking (ISCA'04)*, 2004.

[8] Claire Burguière. “Modélisation de la prédiction de branchement pour le calcul de temps d’exécution pire-cas”. *Thèse de doctorat, Université Paul Sabatier*, juin 2008.

[9] H. Cassé et P. Sainrat. “OTAWA, a framework for experimenting WCET computations”. *3rd European Congress on Embedded Real-Time Software*, 2006. Voir aussi : `http://www.otawa.fr`

[10] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero. “Predictable Performance in SMT Processors”. *ACM Conference on Computing Frontiers*, 2004.

[11] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero. “Architectural Support for Real-Time Task Scheduling in SMT Processors”. *Int’l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2005

[12] F. Cazorla, A. Ramirez, M. Valero, P. Knijnenburg, R. Sakellariou, E. Fernández. “QoS for High-Performance SMT Processors in Embedded Systems”. *IEEE Micro, 24(4)*, 2004.

[13] F. Cazorla, A. Ramirez, M. Valero, E. Fernández. “Dynamically Controlled Resource Allocation in SMT Processors”. *37th International Symposium on Microarchitecture*, 2004.

[14] P. Crowley, J.-L. Baer. “Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors”. *HPCA-9 Workshop on Network Processors*, 2003.

[15] K. Diefendorff. “Compaq Chooses SMT for Alpha”. *Microprocessor Report, vol. 13, no 16*, 6/12/1999

[16] G. Dorai, D. Yeung, S. Choi. “Optimizing SMT Processors for High Single-Thread Performance”. *Journal of Instruction-Level Parallelism, vol. 5*, 2003.

[17] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm et D. M. Tullsen. “Simultaneous Multithreading : A Platform for Next-Generation Processors”. *IEEE Micro, vol. 17, no. 5*, pp. 12-19, Sept./Oct. 1997.

[18] J. Engblom. “Processor Pipelines and Static Worst-Case Execution Time Analysis”. *PhD thesis, Université d’Uppsala*, 2002.

[19] C. Ferdinand. “Worst-case execution time prediction by static program analysis”. *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Workshop 2, 2004.

[20] C. Ferdinand, R. Wilhem. “On Predicting Data Cache Behavior for Real-Time Systems”. *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, 1998.

[21] F. Gruian, “Hard real-time scheduling for low-energy using stochastic data and DVS processors”, *International Symposium on Low Power Electronics and Design (ISPLED)*, 2001.

[22] J. Gustafsson, “The WCET Tool Challenge 2006”, *2nd International Symposium on Leveraging Applications of Formal Methods*, 2006.

[23] Tom R. Halfhill. “Intel’s Tiny Atom”. *Microprocessor Report*, 7 avril 2008

[24] C. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, M.G. Harmon. “Bounding pipeline and instruction cache performance”. *IEEE Transactions on Computers, 48(1)*, janvier 1999.

[25] R. Kalla, B. Sinharoy, J. M. Tendler. “IBM Power5 Chip: a Dual-Core Multithreaded Processor”. *IEEE Micro, vol. 24, no 2*, Mar/Avr 2004.

[26] S. Kato, H. Kobayashi, N. Yamasaki. “U-Link: Bounding Execution Time of Real-Time Tasks with Multi-Case Execution Time on SMT Processors”. *11th Int’l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2005.

[27] S.-K. Kim, S.L. Min, R. Ha. “Efficient worst case timing analysis of data caching”. *2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, 1996.

[28] R. Kirner, P. Puschner,I. Wenzel. “Measurement-based worst-case execution time analysis using automatic test-data generation”. *4th Euromicro International Workshop on WCET Analysis*, 2004.

[29] B. Kirner, I. Wenzel, B. Rieder, P. Pushner. “Using measurements as a complement to static worst-case execution time analysis”. *Conference on Intelligent Systems at the Service of Mankind*, décembre 2005.

[30] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon, Woo-Jong Hahn, “Effects of Multithreading on Cache Performance”, *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 176-184, Fevrier 1999

[31] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, M. T. Vaden. “IBM POWER6 Microarchitecture”. *IBM Journal of Research and development, vol. 51, no 6,* Nov. 2007.

[32] Y.-T. Li, S. Malik. “Performance analysis of embedded software using implicit path enumeration”. *ACM/SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'95)*, 1995.

[33] Y.-T. S. Li, S. Malik, A. Wolfe. “Cache Modeling for Real-Time Software : Beyond Direct Mapped Instruction Cache”. *17th IEEE Real-Time Systems Symposium (RTSS'96)*, 1996.

[34] X. Li, A. Roychoudhury, T. Mitra. “Modeling out-of-order processors for WCET analysis”. *Real-Time Systems, 34(3)*, 2006.

[35] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, C. S. Kim. “An accurate worst case timing analysis technique for RISC processors”. *Real-Time Systems Symposium*, 1994.

[36] S.-W. Lo, K.-Y. Lam, T.-W. Kuo. “Real-time task scheduling for SMT systems”. *11th Int’l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2005.

[37] S. Lopez, S. Dropsho, D. H. Albonesi, O. Garnica, J. Lanchares. “Rate-Driven Control of Resizable Caches for Highly Threaded SMT Processors”. *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*

[38] T. Lundqvist, P. Stenström. “Timing anomalies in dynamically scheduled microprocessors”. *20th IEEE Real-Time Systems Symposium (RTSS'99)*, 1999

[39] T. Lundqvist, P. Stenström, “An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution “, *Real-Time Systems, 17(2-3)*, 1999.

[40] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller et M. Upton. “Hyper-Threading Technology Architecture and Microarchitecture”. *Intel Technology Journal, vol. 06, no 01*, 14/02/2002.

[41] H. M. Mathis, A. E. Mericas, J.D. McCalpin, R. J. Eickemeyer, S.R. Kunkel. “Characterization of Simultaneous Multithreading efficiency in POWER5”. *IBM Journal of Research and development, vol. 49, no 4/5*, Juil./Sept. 2005.

[42] F. Mueller. “Timing Analysis for Instruction Caches”. *Journal of Real-Time Systems, 18(2-3)*, mai 2000.

[43] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, M. De Michiel, “PapaBench : A Free Real-Time Benchmark”, *6th Workshop on Worst-Case Execution Time Analysis*, 2006.

[44] S. M. Petters. “Worst-Case Execution Time Estimation for Advanced Processor Architectures”. *PhD thesis, Technische Universität München*, septembre 2002.

[45] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, M. J. Smith. “Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading”. *International Solid-State Circuits Conference*, 2002.

[46] I. Puaut, A. Arnaud, D. Decotigny. “Performance analysis of static cache locking in multitasking hard real-time”. *Technical Report PI 1568, IRISA*, oct. 2003.

[47] I. Puaut, D. Decotigny. “Low-complexity algorithms for static cache-locking in multitasking hard real-time systems”. *23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

[48] P. Pushner, C. Koza. “Calculating the Maximum Execution Time of Real-Time Programs”. *Journal of Real-Time Systems, volume 1*, 1989.

[49] S. Raasch, S. Reinhardt. “The Impact of Resource Partitioning on SMT Processors”. *12th Int’l Conf. on Parallel Architectures and Compilation Techniques*, 2003.

[50] C. Rochange, P. Sainrat. “A Context-Parameterized Model for Static Analysis of Execution Times”. *Transactions on HiPEAC, 2(3),* Springer, 2007.

[51] A. Settle, J. Kihm, A. Janiszewski, D. Connors. “Architectural Support for Enhanced SMT Job Scheduling”. *13th International Conference on Parallel Architecture and Compilation Techniques (PACT’04)*

[52] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, J. B. Joyner. “POWER5 system microarchitecture”. *IBM Journal of Research and development, vol. 49, no 4/5*, Juil./Sept. 2005.

[53] F. Stappert, P. Altenbernd. “Complete worst-case execution time analysis of straightline hard real-time programs”. *Euromicro Journal of Systems Architecture, 46(4)*, 2000.

[54] H. Theiling, C. Ferdinand, R. Wilhelm. “Fast and Precise WCET Prediction by Separated Cache and Path Analyses”. *Journal of Real-Time Systems, 18(2/3)*, mai 2000.

[55] N. Tuck, D. M. Tullsen. “Initial Observations of the Simultaneous Multithreading Pentium 4 Processor”. *International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[56] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm. “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”. *23rd Int’l Symposium on Computer Architecture*, 1996.

[57] D. Tullsen, S. Eggers et H. Levy. “Simultaneous Multithreading : Maximizing On-Chip Parallelism”. *22nd Int’l Symposium on Computer Architecture*, 1995.

[58] X. Vera, B. Lisper, J. Xue. “Data cache locking for higher program predictability”. ACM/SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03), 2003.

[59] I.Wenzel, R.Kirner, B.Rieder, P.Puschner. “Measurement-based worst-case execution time analysis. *IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems(SEUS 2005)*, 2005.

[60] N.Williams, B.Marre, and P.Mouy. “Interleaving static and dynamic analyses to generate path tests for c functions”, *3rd Workshop on System Testing and Validation (SV'04)*, 2004.

[61] N. Williams, B. Marre, P. Mouy, M. Roger. “PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis”. *5th European Dependable Computing Conference*, 2005.

[62] F. Wolf, J. Staschulat, R. Ernst. “Associative Caches in Formal Software Timing Analysis”. *39th ACM/IEEE Design Automation Conference (DAC 2002)*, 2002.

[63] AbsInt : http://absint.com

[64] http://www.irit.fr/Gliss

[65] http://archi.snu.ac.kr/realtime/benchmark

[66] http://paparazzi.enac.fr

[67] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[68] http://www.simplescalar.com

AUTHOR: Barre Jonathan

TITLE: Simultaneous Multithreading Architectures for hard Real-Time

ABSTRACT:

In critical systems, applications must satisfy hard timing constraints, each task must execute in a maximum predefinite time. Any unrespected constraint may compromise the stability of the whole system and generate disastrous effects. Such a system is called hard real-time system.

To be able to assign a constraint to a task, you must be able to determinate the maximum time this task will execute, independently from the input data of the task. This maximum time you search is called the WCET (Worst Case Execution Time), it is obtained by a calculation process where we need to modelise the structures of the processor architecture. The architecture mechanisms increasing performance (caches, branch prediction) are often a lot undeterministic and thus are difficult to modelise.

That’s why we usually prefer using relatively simple architectures for a hard real-time system, or simplifying recent high-performance architecture.

In this work, we will rather adapt, using small modifications, one of those high-performance but little predictible architecture to respect hard timing constraints and make simpler WCET calculation. We choose the Simultaneous Multithreading architecture where several programs can run at the same time sharing the resources of one core only.

KEY WORDS: hard real-time, WCET, SMT architectures, task scheduling

AUTEUR : Barre Jonathan

TITRE : Architectures multi-flots simultanés pour le temps-réel strict

DIRECTEUR DE THESE : Sainrat Pascal

LIEU ET DATE DE SOUTENANCE : IRIT, université P. Sabatier, 15/12/2008

RESUME en français :

Dans les systèmes critiques, les applications doivent satisfaire des contraintes temporelles strictes, chaque tâche devant s'exécuter en un temps maximum prédéfini ; le non-respect d'une seule échéance peut compromettre toute la stabilité du système et engendrer des effets désastreux. Un tel système est appelé système temps-réel strict.

Pour pouvoir assigner une échéance à une tâche, il faut être capable de déterminer le temps maximum que mettra cette tâche à s'exécuter, ceci indépendamment des données en entrée de la tâche. Ce temps maximum recherché s'appelle le WCET (Worst Case Execution Time, temps d'exécution pire cas), il est souvent déterminé à l'issue d'un processus de calcul nécessitant une modélisation des structures de l'architecture du processeur. Les mécanismes architecturaux qui augmentent les performances d'un processeur (prédiction de branchement, cache) induisent souvent un fort taux d'indéterminisme qui rend la modélisation difficile.

C'est pourquoi il est souvent préférable d'utiliser des architectures relativement simples pour un système temps-réel strict, ou de simplifier des architectures hautes performances récentes.

Notre optique est plutôt d'essayer d'adapter, par de légères modifications, une de ces architectures performantes mais peu prédictibles pour un respect de contraintes temps-réel strict et un calcul de WCET facilité. L'architecture que nous choisissons est l'architecture Multi-Flots Simultanés (Simultaneous Multihtreading, SMT), ou plusieurs programmes peuvent s'exécuter simultanément en partageant les ressources d'un seul cœur d'exécution.

MOTS-CLES : temps-réel strict, WCET, architectures SMT, ordonnancement de tâches

---

DISCIPLINE ADMINISTRATIVE : Informatique

INTITULE ET ADRESSE DE L'U.F.R. OU DU LABORATOIRE :
IRIT, Université Paul Sabatier, 118 Route de Narbonne, 31062 TOULOUSE CEDEX 9