Calcul de majorants sûrs de temps d'exécution au pire pour des tâches d'applications temps-réels critiques, pour des systèmes disposants de caches mémoire

Stéphane Louise

N° D'ORDRE : 6802

UNIVERSITÉ PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée

Pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

PAR

Stéphane LOUISE

Sujet : Calcul de majorants sûrs de temps d'exécution au pire pour des tâches d'application Temps-réel critique pour des sytèmes disposants de caches mémoire

Soutenue le 21 janvier 2002 devant la Commission d'examen

Monsieur	Vincent DAVID
Monsieur	Alain MÉRIGOT
Madame	Isabelle PUAUT
Monsieur	Pascal SAINRAT
Monsieur	Per STENSTRÖM
Monsieur	Guy VIDAL-NAQUET

REMERCIEMENTS. Un grand merci au gens du LLSP à Saclay, en particulier à Jean, Vincent et Christophe pour les conseils, les corrections et les idées ; à Arnaud, Manu et Guillaume pour les conversations, mais aussi à tout les autres pour leur accueil. Un grand merci également à Guy Vidal-Nacquet et Nathalie Drach-Temam pour leur rôle primordial dans la constitution du jury. Enfin, merci à l'équipe de développement de LyXpour leur outil précieux qui a largement simplifié l'écriture de ce mémoire.

Ce mémoire de thèse est dédié à mes parents qui m'ont toujours soutenu malgré les trop nombreuses années d'incertitudes et qui ont eux aussi (et encore une fois) contribué à l'amélioration othographique de ce mémoire.

Table des matières

Table des figures

Liste des tableaux

Résumé

Ce mémoire présente une nouvelle approche pour le calcul de temps d'exécution au pire (*WCET*) de tâche temps-réel critique, en particulier en ce qui concerne les aléas dus aux caches mémoire. Le point général est fait sur la problématique et l'état de l'art en la matière, mais l'accent est mis sur la théorie elle-même et son formalisme, d'abord dans le cadre monotâche puis dans le cadre multitâche. La méthode utilisée repose sur une technique d'interprétation abstraite, comme la plupart des autres méthodes de calcul de *WCET*, mais le formalisme est dans une approche probabiliste (bien que déterministe dans le cadre monotâche) de par l'utilisation de chaînes de Markov. La généralisation au cadre multitâche utilise les propriétés probabilistes pour faire une évaluation pessimiste d'un WCET et d'un écart type au pire, grâce à une modification astucieuse du propagateur dans ce cadre. Des premières évaluations du modèle, codées à la main à partir des résultats de compilation d'applications assez simples montrent des résultats prometteurs quant à l'application du modèle sur des programmes réels en vraie grandeur.

Abstract

This report is a presentation of a new approach for Worst Case Execution Time (WCET) computation for hard real-time systems, especially for cache memories hazard issues. A general overview of issues and state of the art in this matter is drawn, but the main point is the theory in itself and its formalism, first, in single task execution, then in multitasking environment. The method being used relies on abstract interpretation, like many other methods of WCET computation, but its formalism is a probabilistic approach (although it is deterministic in the single task field) with the use of Markov chains. Generalization to the multitask field make use of this probabilistic base to compute a pessimistic evaluation of the WCET and of its standard deviation, thanks to a clever modification of the propagator. First evaluation of the model results, hand-coded from the compilation output of rather simple tasks, shows that results are promising in order to apply it to real-life applications.

Introduction

La programmation de systèmes temps-réel est l'art d'obtenir un comportement temporel déterministe pour les programmes. Des modèles élaborés, comme OASIS[**6**] permettent, en cela, de faciliter grandement la tâche du programmeur qui peut se concentrer sur la conception en fournissant juste des contraintes temporelles au compilateur. Cependant pour vérifier que le système pourra supporter la tâche prévue, il est nécessaire de savoir obtenir un majorant des temps d'exécution dans le pire cas des programmes exécutés par le système en question. Pourtant, les améliorations successives aux architectures de processeurs faites depuis plus de dix ans ne vont pas dans le sens d'un déterminisme supérieur du temps d'exécution d'une instruction prise au hasard, bien au contraire. La vitesse d'exécution moyenne augmente très rapidement, mais il est de plus en plus difficile de dire quels sont les effets réels sur un temps d'exécution dans le pire cas qui est pourtant la donnée essentielle qui intéresse un développeur d'application temps-réel. Il semble donc y avoir contradiction entre l'évolution des processeurs et les besoins de ces systèmes. En particulier les caches mémoire dont les contenus peuvent varier de façon parfois difficile à prévoir alliée à des conséquences pouvant s'avérer importantes sur les temps d'exécution également, est le système qui pose le plus de problème.

C'est pourquoi, depuis le début des années 90, un nombre important d'équipes de recherche se focalisent sur la tâche délicate de la prédiction du comportement des caches et de leurs incidences sur la prédiction des

temps d'exécution. Il existe un intérêt économique important à ce genre de résultats car ils permettent d'envisager, à terme, d'utiliser de vrais systèmes temps-réel (dits *temps-réel critiques*) sur des systèmes qui peuvent aller jusqu'au grand public (on peut penser à l'industrie automobile, par exemple). Une résolution exacte, dans le cas général est cependant hors de question car il s'agit d'un problème NP-complet variant avec le nombre d'instructions potentiellement exécutées, nombre généralement énorme, même pour un programme simple. Le but est donc d'utiliser les propriétés spécifiques de ces programmes particuliers pour réduire un peu la complexité du problème et, de réduire dans le même temps la difficulté en cherchant simplement un majorant sûr de ces temps d'exécutions, mais qui surestime aussi faiblement que possible les temps effectifs, ou même d'en fournir une évaluation avec un moyen de majorer les probabilités de sous-évaluation.

La présente thèse introduit donc une nouvelle façon d'arriver à ce résultat et s'inscrit dans la recherche fondamentale autour du projet OASIS développé au CEA-Saclay dans le laboratoire des Logiciels pour la Sûreté des Programmes (LLSP).

Ce mémoire s'articule de la façon suivante : dans un premier temps, il s'agira d'exposer la problématique dans ses aspects les plus importants autour de trois chapitres : le premier qui expose les grande idées et les exigences du temps-réel critique, le deuxième qui s'intéresse au modèle OASIS qui forme le cadre à cette thèse en tant que modèle de conception des systèmes temps-réels critiques et un troisième chapitre qui lui introduit un état de l'art des architectures de processeurs en mettant particulièrement l'accent sur les problèmes que posent ces architectures modernes vis à vis de la prédiction des temps d'exécution des programmes. La deuxième partie établie un état de l'art des calculs de temps d'exécution au pire au travers de

morceaux choisis particulièrement caractéristiques. Cela s'articule en deux chapitres, le premier d'entre eux s'appliquant à donner un aperçu des modèles possibles et le second détaillant quelques points importants vis à vis des caches. La troisième partie de l'exposé concerne le travail de thèse proprement dit et se focalise sur le formalisme théorique qui a été développé. Un premier chapitre s'intéresse au cadre d'une exécution séquentielle et développe les bases même du formalisme et de la modélisation de l'interaction entre le cœur d'exécution d'un processeur, de la (ou des) mémoire(s) cache(s) et de la mémoire principale auquel il accède au cours de l'exécution d'un segment de programme. Le deuxième chapitre de cette partie s'attache à généraliser le formalisme précédent au cadre multitâche. Enfin un chapitre est consacré à expliquer comment passer de l'estimation des échecs de cache(s) que l'on obtient dans le modèle à une estimation des temps d'exécutions au pire. La dernière partie fournira enfin des résultats numériques obtenus par simulation, avant de conclure.

Première partie

Introduction au Problème ; Temps d’exécution au pire

CHAPITRE 1

Temps-réel critique, cadre général

1.1. Introduction : programmes temps-réel

Un programme temps-réel est un programme, *a priori,* semblable à tout autre programme, si ce n'est que le paramètre temps doit être pris en compte et qu'il ne possède pas de terminaison. Il doit être capable dans un temps imparti et borné de traiter les données qui lui sont fournies. Les exemples les plus classiques sont l'acquisition de données, surveillance, traitements de données, reconnaissance de formes (au sens général), les systèmes de contrôle-commande... Une définition généralement admise en informatique est qu'un système temps réel est un système pour lequel non seulement les résultats de calculs sont valides pour le système considéré - donc des algorithmes corrects - mais également la date à laquelle est produit le résultat est importante. Un résultat qui arrive trop tard (ou trop tôt) peu donc être aussi invalide qu'un résultat faux algorithmiquement parlant. Un exemple banal serait un petit robot auto-piloté suivant une ligne tracée au sol. Si la ligne tourne et que le temps de traitement des données sur l'acquisition de la ligne est trop long (intervalle dépendant de la vitesse de l'engin et de la courbure de la ligne), le robot aura dépassé la ligne avant d'avoir pu tourner, et sera donc perdu (car sorti de la ligne de guidage). Il faut donc être capable de traiter toutes les données importantes pour le fonctionnement d'un système temps réel dans un temps donné calculable. Il y a trois grands modèles d'implémentation de systèmes temps-réels que l'on va maintenant développer.

Modèle de programmation en boucle. Le concept en est simple. Il s'agit de faire succéder dans une boucle globale la totalité des acquisitions et des traitements de manière séquentielle, entrelacés de façon à respecter les contraintes temps-réel. Ce type de programmation permet un comportement vérifiable de l'application au niveau de l'ordonnancement (car elle est strictement linéaire, donc les points de blocages possibles sont facilement identifiables) mais n'est vraiment applicable qu'à des projets de petite taille, sans quoi la charge de programmation et de vérification devient trop lourde à gérer (surtout au niveau de la vérification des contraintes temporelles).

Modèle de programmation dirigée par évènements (Event Triggered - ET). C'est la démarche la plus utilisée, car c'est la plus aisée à mettre en oeuvre, même s'il existe des variantes plus ou moins complexes. Dans sa forme la plus simple, chaque évènement reçu génère une interruption de priorité fixe qui est gérée à l'aide d'une routine adaptée par le microprocesseur. La mise en oeuvre est donc très simple puisqu'il s'agit simplement d'utiliser les mécanismes de gestion d'interruptions des microprocesseurs. On peut donc se passer totalement de programmes d'ordonnancement de tâche puisque cela est géré au niveau matériel par le microprocesseur. Malheureusement, au niveau de la sûreté de fonctionnement, dans le cadre du temps-réel critique, cela peut poser des problèmes vu qu'un évènement particulier se répétant à un rythme inadapté pour le système hôte peut provoquer une série d'interruption de haut niveau, et donc non préemptibles par d'autres tâches, ce qui peut contraindre le système à violer les échéances de ces tâches de priorités inférieures. Il existe bien entendu des modèles plus abouti que cette version très simple de la méthode ET. De manière générale, la vérification des contraintes temporelles est un des points délicat de l'approche ET, bien que cela ne soit pas le seul problème potentiel.

Modèle de programmation dirigée par le temps (Time Triggered - TT). Cela signifie que le système est synchronisé par une ou plusieurs horloges qui indiquent à l'application à quels moments il est possible de changer de tâche. A ce titre, l'acquisition de données sur les capteurs est une tâche comme une autre, donc régie par une commutation TT, plutôt que par l'arrivée d'un évènement sur un capteur. En cela l'approche TT[**15**] s'oppose à la démarche de direction par évènements (ET) qui est plus répandue à l'heure actuelle dans le domaine du temps réel. L'avantage de TT est de permettre un agencement centralisé et sûr des tâches. Cela suppose, bien sûr, la présence d'un système d'ordonnancement et de commutation de tâches. Son avantage étant de pouvoir gérer de manière fine le cas échéant les problèmes de conflits de tâches que l'on avait évoqué dans le cadre d'ET à priorité fixe. C'est une démarche assez difficile à mettre en oeuvre et donc assez peu utilisée sauf dans le cas des systèmes temps-réel critiques dont on va parler à présent.

1.2. Temps-réel critique

Le cadre du temps réel critique nécessite, de plus, une gestion rigoureuse du temps, car tout dépassement des délais impartis peut avoir des conséquences graves sur l'environnement. Cela peut être morts ou blessures de personnes, dangers sur l'intégrité du système piloté, risques de pollutions, catastrophes économiques ou humaines, etc…

Généralement il s'agit de gros systèmes comme ceux embarqués sur des fusées, des satellites, des avions, ou des centrales électriques, mais il est probable qu'à l'avenir, des impératifs de sécurité comparables à la norme ISO/CEI 880 en vigueur dans le domaine du nucléaire viennent se greffer à des systèmes embarqués plus courants comme les systèmes d'asservissements des véhicules routiers, pour ne citer que cet exemple. De ce fait,

la possibilité de savoir gérer de façon sûre les architectures actuelles voire futures devient un besoin de plus en plus impérieux.

Une autre définition que l'on trouve parfois pour le concept de temps-réel critique est celui d'un système qui est en faute dès lors qu'une échéance quelconque est dépassée. Un système temps-réel non critique serait-alors, par opposition, un système pour lequel une certaine proportion de dépassements d'échéances serait tolérée.

De ce fait, le temps-réel critique implique des propriétés fortes pour le logiciel :

Vivacité du système: aucune tâche ne peut être bloquée indéfiniment.

Ponctualité des tâches: les tâches respectent leurs échéances.

Sûreté de fonctionnement: une tâche mal programmée, ou défaillante (panne) ne doit pas avoir d'influence sur les autres.

Pour respecter de telles exigences, il est nécessaire que l'on puisse, d'une manière ou d'une autre, faire les choses suivantes sur le système et les programmes mis en œuvre :

- obtenir un dimensionnement correct de l'architecture cible, c'est à dire être sûr que la machine cible dispose de suffisamment de ressources matérielles pour effectuer les tâches demandées (et implicitement être capable d'évaluer correctement ces ressources) ;
- connaître la charge du système dans le pire cas d'exécution de l'application temps réel, c'est à dire être capable de prévoir de façon sure que l'on dispose de suffisamment de ressources processeurs pour l'exécution de toutes les tâches tout en respectant les échéances temporelles, quelque soit les circonstances ;

- éviter de surdimensionner le système mis en place, pour des problèmes économiques évidents, car cela augmente le coût du système matériel mais également complique la tâche de démonstration de sûreté de fonctionnement ;
- être en mesure de démontrer la sûreté du système complet intégré à son environnement.

La problématique de cette thèse renvoie au deuxième point exposé : trouver un majorant strict, non trivial, pour des systèmes qui tirent partie des architectures actuelles, et en particulier de la présence de caches mémoire. Cela permettra de faire des systèmes mieux dimensionnés, donc plus sûrs.

CHAPITRE 2

Modèle OASIS

Le modèle OASIS[**6**] repose sur une approche multitâche coopérative dont la commutation est assurée par une ou plusieurs horloges, le plaçant donc dans une approche TT[1]. Un noyaux central est chargé de la conformité de la commutation des tâches par rapport au modèle. Chaque tâche, également appelé agent, possède des échéances connues pour chaque traitement. L'ordonnancement des tâches dans le cadre monoprocesseur est actuellement réalisé par l'algorithme EDF, mais le modèle peut être adapté à d'autres méthodes d'ordonnancements (sous contrainte de prédictibilité de la tenue en charge, bien entendu). Le but principal du modèle OASIS est de faciliter le développement et la mise en œuvre de systèmes temps-réels critiques. Nous allons en voir les principaux éléments.

2.1. Tâches-Agents

Le modèle OASIS repose sur une approche basée objet qui encapsule chaque tâche et ses données associées. La structure ainsi construite est appelée *Agent*. Chaque agent a l'unique privilège de pouvoir écrire les données qui lui appartiennent, opérant ainsi un cloisonnement des tâches entre elles, ce qui participe à l'objectif de sûreté du système. Ces agents ont cependant la possibilité de communiquer de deux façons qui sont d'une part, l'envoi de messages et d'autre part, la lecture d'un buffer de variables temporelles, contenant des valeurs passées des variables considérées.

Les hypothèses faites sur les tâches ainsi définies sont simples :

[1] Time Triggered, voir section 1.1

– le nombre de tâches est fixé à l'avance par le programmeur ;

– les algorithmes itératifs ou récursifs ont des bornes finies connues.

Ces hypothèses sont peu restrictives dans le cadre des programmes temps-réel. Ceci étant établi, une application OASIS devient un ensemble de tâches (agents) qui interagissent. Ces hypothèses formulées et vérifiées par le compilateur pour l'application développée permettent d'estimer la charge maximale (qui est finie, bornée) pour le système cible, moyennant la connaissance de majorants de temps d'exécution des différentes tâches[**2**], ce qui est le but de cette thèse. Moyennant cela, la *ponctualité* du système est assurée, et vérifiable analytiquement dans l'approche OASIS.

2.2. Temps réel et sûreté dans le modèle OASIS

L'aspect temps-réel en lui-même est assuré par une architecture dirigée par le temps (TT, tel qu'on l'a vu au paragraphe 1.1). A chaque tâche OASIS ω est associée une horloge H_ω qui sert de base à l'architecture *Time Tiggered*. L'application globale Ω a de même une horloge H_Ω qui est l'union de toutes les horloges d'agents H_ω que l'on peut, a posteriori, donner en fonction de l'horloge du système TT ainsi défini.

Un métalangage est utilisé dans le modèle OASIS afin de n'écrire que des programmes conformes au modèle. Dans le cas du C, le métalangage associé est le ΨC. Il n'existe qu'une seule instruction bloquante dans ce modèle, il s'agit de l'instruction ADV (*advance*) qui permet de définir la prochaine date d'activation au plus tôt de la suite de la tâche (mais aussi la date d'échéance au plus tard du traitement courant). Cette absence d'autre instruction totalement bloquante permet de s'assurer a priori de la *vivacité* du système.

Les conséquences des choix faits dans OASIS permettent ainsi d'avoir :

– un ordonnancement toujours possible des tâches ;

– un comportement déterministe de l'application ;
– une démonstration automatique possible du respect des échéances et adaptation du système à la charge prévue sous réserve de connaître un majorant des temps d'exécution des différentes tâches.

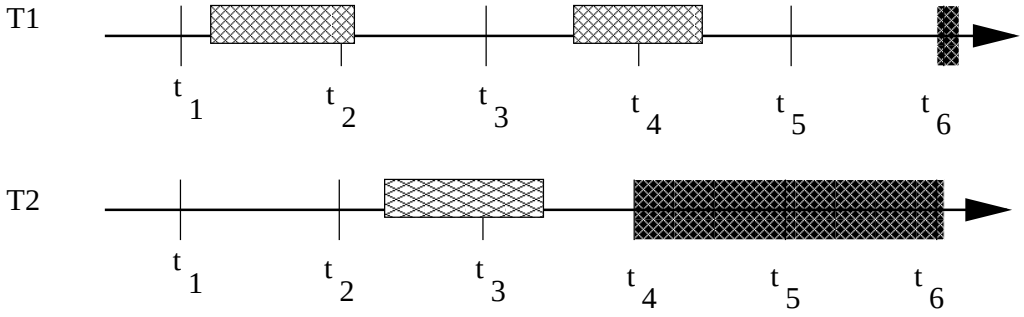
Ce dernier point est le problème qui nous intéresse plus particulièrement.

On doit également se souvenir qu'OASIS est typiquement multitâches (multiagents), et de ce fait les temps d'exécution sont finalement influencés par l'interaction entre tâches, et la commutation de tâches.

Enfin, on note qu'un des aspect de la sûreté de fonctionnement - mais non lié aux problèmes temporels - est aussi assurée de manière matérielle par l'unité de gestion de la mémoire (MMU) des processeurs, rendant impossible tout accès d'une tâche donnée à des segments mémoires non autorisés. Ainsi, dans la mesure où l'on sait démontrer la sûreté du micronoyau chargé de la commutation de tâches, on peut s'assurer de la sûreté de fonctionnement de l'application. Dans l'approche OASIS, le minimum de choses sont requises pour la démonstration de la sûreté des systèmes temps-réel critiques implémentés en suivant le modèle.

Le modèle OASIS permet donc de s'assurer dès la conception du système de la vivacité du système, de la ponctualité des tâches grâce à une adaptation du système à la charge prévue ainsi que la sûreté du système au point de vu logiciel grâce à l'utilisation d'un mécanisme matériel de protection des tâches entre-elles. Ce modèle permet d'assurer au programmeur de concevoir un système temporellement conforme aux exigences des systèmes temps-réel critiques, telles qu'on les a vu au paragraphe 1.2. Il ne lui reste plus qu'à s'assurer de l'aspect algorithmique du projet.

FIG. 2.3.1. *ordonnancement EDF de deux tâches T_1 et T_2.*



2.3. Ordonnancement EDF

C'est l'ordonnancement actuellement implémenté dans OASIS sur un processeur. A noter qu'il s'agit bien d'une implémentation possible, et que cela n'est pas une exigence du modèle OASIS. Dans un tel ordonnancement, on fait en sorte que la ressource processeur est donnée à la tâche prête dont l'échéance est la plus proche dans le temps (*Earliest Dealine First* : EDF). On sait qu'un tel algorithme d'ordonnancement des tâches est un des plus efficaces existants[**18**]. Avec un tel ordonnancement, on assure la *ponctualité* des tâches tant que le système est adapté, donc non surchargé.

A titre d'illustration du principe d'EDF, supposons que l'on ait deux traitements T_1 et T_2, le premier étant en cours d'exécution et le second ne pouvant être exécuté avant la date t_2, T_1 a pour échéance t_6 et T_2 a pour échéance t_4 (voir fig-2.3.1). En t_1, seul T_1 peut-être traité, donc il s'agit de la tâche en cours d'exécution. A la date t_2, T_1 et T_2 sont potentiellement à exécuter, il faut donc opérer un choix. Comme T_2 a $t_4<t_6$ comme échéance (plus proche échéance), c'est T_2 qui doit être élue par l'algorithme d'ordonnancement. Lorsque T_2 est finie, l'algorithme choisi de reprendre l'exécution de T_1 qui est encore à finir.

L'ordonnancement EDF n'est pas le seul que l'on puisse implémenter dans le modèle OASIS mais c'est un candidat de choix au vu de ses propriétés. Il faut noter également que la nécessité de déterminisme dans les applications temps-réel critique impliquent de faire également un choix déterministe en cas d'égalité des échéances. C'est donc une légère contrainte supplémentaire qu'il faut prendre en compte par rapport à un ordonnancement EDF ordinaire. D'autres méthodes ordonancement, et en particulier les extensions possibles au cadre multiprocesseur avec réseau de communication sont traités dans [**2**].

CHAPITRE 3

Temps d'exécution et architecture des processeurs

Introduction

Depuis la fin des années 70, de plus en plus d'améliorations architecturales destinées à augmenter l'efficacité d'exécution du code machine par les microprocesseurs ont fait leur apparition.

Chacune pose des problèmes spécifiques quant à la prédiction des temps d'exécution.

On va, dans ce chapitre, s'intéresser plus particulièrement aux points suivants :

- le pipeline ;
- les architectures superscalaires ;
- la prédiction de branchement ;
- l'unité de Gestion de la Mémoire, MMU ;
- la hiérarchie mémoire.

On donnera également quelques idées concernant les tendances récentes ou prévisibles à moyen terme dans le domaine de l'architecture des processeurs.

3.1. Le Pipeline

3.1.1. Principe. L'idée est de découper l'exécution globale d'une instruction en processus plus élémentaires, donc plus rapides et plus simples ce qui permet d'augmenter la cadence d'horloge du processeur.

Classiquement sur un processeur pipeliné d'ordre 4 (exemple classique), pour la séquence d'instructions simples 1, 2, 3, 4, 5, 6, on peut décrire l'état du pipeline aux instants successifs :

	déroulement du programme $\mapsto$					
t	FET-4	DEC-3	EX-2	STO-1		
t+1		FET-5	DEC-4	EX-3	STO-2	
t+2			FET-6	DEC-5	EX-4	STO-3

FET : chargement de l'instruction dans le coeur du processeur.

DEC : décodage de l'instruction/chargement des données nécessaires à l'exécution.

EX : exécution élémentaire de l'instruction.

STO : terminaison de l'instruction/mise à jour des données.

Permet d'exécuter une instruction par cycle avec une vitesse d'horloge multipliée par 4, dans cet exemple simple. En effet, si l'on regarde l'exécution de l'instruction 4, par exemple, elle rentre dans le pipeline (chargement) au temps t, elle est exécutée à proprement parler au temps t+2 et elle n'est réellement terminée qu'à la fin du cycle en t+3. Une instruction particulière met donc autant de temps pour s'exécuter totalement, mais 4 instructions peuvent être exécutées dans cette chaîne. On compare classiquement le pipeline aux chaînes de montages de véhicules automobiles, par exemple. Chaque étage du pipeline correspond à une partie de la complétion d'une instruction, comme chaque robot de la chaîne de montage complète un petit bout du véhicule final. Cependant contrairement à une chaîne de montage classique, pour le pipeline, la complétion d'une instruction donnée peut dépendre d'une instruction précédente, car il ne s'agit pas d'exécuter une suite

d'instructions indépendantes mais un programme qui est le résultat d'interactions entre des instructions. Dans ce pipeline, on superpose l'exécution de 4 instructions, avec tout les problèmes qui peuvent en découler et que l'on va détailler à présent.

3.1.2. Les inconvénients du pipeline : influence sur les temps de transit des instructions. Il y a deux grandes classes de problèmes qui peuvent provoquer des aléas dans les temps d'exécution constatés pour une instruction donnée :

- la dépendance de données entre instructions présentes dans le pipeline peut entraîner des retards ("bulles") dans l'exécution d'instructions ;
- les ruptures de séquence d'instructions correspondant au flot de contrôle de l'application entraînent des retards qui deviennent difficiles à calculer dans les architectures les plus récentes.

Tout cela fait que le temps d'exécution d'une instruction donnée n'est pas déterminé à l'avance mais dépend à la fois du contexte local à l'instruction considérée au sein du programme et du contexte d'exécution.

Exemples :

	ADD R2,R1,R8	LD R1,(R23)	
FET4	DEC3	EX2	STO1

Dépendance de donnée (ici de type Lecture après Écriture - RaW[1]) : Avant de pouvoir être exécutée, l'instruction en cours de décodage a besoin de la valeur du registre R1 qui ne sera connue qu'après la fin de l'exécution de l'instruction courante d'où une suspension de pipeline pour attendre la mise à jour de R1. On doit noter qu'il existe d'autres types de dépendances de données, comme les dépendances de type Écriture après Lecture (WaR[2])

[1] Read after Write
[2] Write after Read

mais contrairement à la dépendance RaW que l'on qualifie de dépendance de données vraie, elles sont appelées fausses dépendances de données car elles peuvent être supprimées en changeant les registres utilisés sans changer l'algorithme ni le résultat. Dans les processeurs modernes, il existe des mécanismes de renommage de registres qui permettent d'éviter en partie ces fausses dépendances de données en effectuant la séparation entre les noms ou numéros de registres donnés dans le code machine (registres logiques) et les registres effectivement utilisés lors de l'exécution de l'instruction (registres physiques) plus nombreux. En quelque sorte, cela revient à une simulation matérielle d'une architecture sur une implémentation en fait plus puissante. Cela permet pour un coût architectural raisonnable à l'heure actuelle, de faire progresser les performances des processeurs sans changer l'architecture et donc en conservant la compatibilité binaire. Il va de soi que ce genre d'astuces matérielles ne va pas non plus dans le sens de l'augmentation des préditibilités des temps d'exécution des instructions car dans un programme complexe, il devient délicat de savoir à quels moments on risque de saturer les mécanismes de renommage et l'incidence exacte d'un tel évènement sur le nombre de cycles de retard qui en découlent.

		BSR (Addr)	
FET4	DEC3	EX2	STO1

Cas du branchement : si on attend le niveau d'exécution pour effectuer le branchement, les instructions 3 et 4 doivent être invalidées : perte de 2 cycles. Là encore, les architectures les plus modernes utilisent des prédicteurs de branchement dont certains ont une excellente efficacité statistique pour charger et exécuter spéculativement le bon code aussi souvent que possible. On en discutera plus particulièrement au paragraphe 3.3.

Il faut bien noter que l'influence du pipeline lors des commutations de tâches se limite au plus à quelques cycles de retard. Ce n'est pas l'effet prépondérant pour cet aspect.

3.2. Les architectures superscalaires

L'idée à la base des architectures superscalaires est simple : pour augmenter la vitesse de traitement d'un processeur, pourquoi ne pas exécuter plusieurs instructions à la fois. Les architectures superscalaires se sont généralisées depuis le début des années 90. Aujourd'hui, les processeurs courant peuvent exécuter jusqu'à quatre instructions par cycle, voire même six pour le processeur Power 3 d'IBM. On peut même noter que certains DSP à architecture VLIW peuvent aller jusqu'à 8 instructions par cycle, mais on ne parlera de VLIW que lorsqu'on abordera les perspectives d'évolutions des processeurs (voir le paragraphe 3.7.1). Les architectes parlent désormais aussi de futur avec des processeurs superscalaires d'ordre 8 voire plus, mais cela ne se fera pas sans de nouvelles révolutions architecturales majeures (et probablement de nouveaux aléas sur les temps d'exécutions).

exemple : cas d'un processeur superscalaire d'ordre 2, pipeliné d'ordre 4

FET7	DEC5	EX3	STO1
FET8	DEC6	EX4	STO2

Cela aura en fait le même genre de conséquence que le pipeline seul, du fait de l'exécution globale simultanée d'un nombre accru d'instructions.

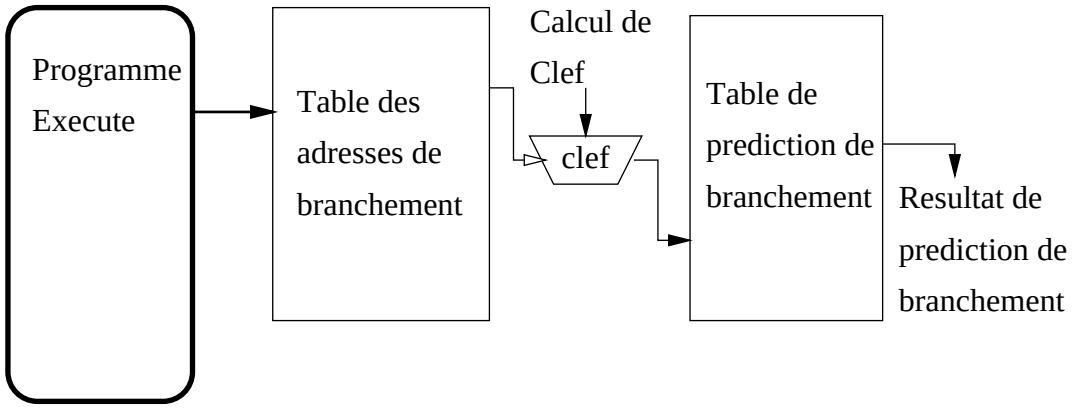
Un point à remarquer pour le cas des processeurs superscalaires est que, de nos jours, les instructions ne sont plus exécutées dans l'ordre dans lequel elles apparaissent dans le programme, afin de gagner du temps. Néanmoins,

on les force quand même à se terminer dans l'ordre de séquence du programme afin d'en respecter la cohérence et de pouvoir réagir de manière déterministe à une éventuelle exception. Il faut aussi remarquer que l'aspect superscalaire influence essentiellement le temps d'exécution d'un code séquentiel mais qu'une commutation de tâche n'a pas une influence fondamentalement différente du cas d'un pipeline ordinaire, et qui reste du même ordre de grandeur. Ce ne sera donc pas non plus l'effet prépondérant lors des commutations de tâches.

3.3. La prédiction de branchements

Comme on l'a vu lors de la discussion sur les pipelines, les aléas du flot de contrôle sont une source de suspension du pipeline. C'est pourquoi, surtout depuis l'arrivée des processeurs superscalaires, il est intéressant de minimiser les suspensions dues à ces aléas. La première idée fut de faire de la prédiction statique de branchement, car on sait que statistiquement un branchement vers l'avant est majoritairement non pris et un branchement vers l'arrière très souvent pris (le plus souvent il s'agit de boucles). Mais il est plus efficace de faire une prédiction dynamique, d'où l'idée de référencer les instructions de branchements et leur comportement afin de pouvoir prédire à l'avance les résultats de ceux-ci. C'est ce que font les prédicteurs de branchements. Les meilleurs prédicteurs utilisés actuellement ont une efficacité de prédiction moyenne supérieure à 90%. Pour une boucle donnée, par exemple, il est exceptionnel qu'il y ait plus de 3 fausses prédictions. Tout ces prédicteurs reposent sur un principe similaire. Une clef est calculée en fonction de divers paramètres (qui déterminent en grande partie l'efficacité de la prédiction) qui sert à connaître une prédiction référencée par la clef, dans une table. La précision de la prédiction dépend en général de la manière de calculer la clef et de la taille de la table.

FIG. 3.3.1. *Prédicteur de branchement : schéma de principe*



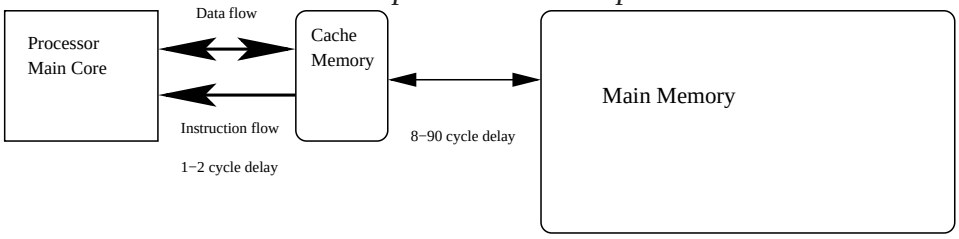
Cette table sera fatalement modifiée à la suite d'une commutation de tâche, car la nouvelle tâche ne va pas manquer d'en modifier le contenu. On peut penser que pour le type de programmes qui nous intéresse que l'impact devrait rester limité, par rapport à celui sur les caches mémoire. On peut voir un synopsis d'une prédiction de branchement en figure 3.3.1 page 41. On pourra obtenir une vue à peu près complète des prédicteurs de branchements actuels en se référant à [**14**]. Les premières méthodes de prédiction dynamique de branchement faisait appel à des schémas relativement simples, en particulier à des compteurs saturants d'abord à 1 bit (0 signifiant branchement non pris et 1 branchement pris, souvent initialisés par défaut en fonction du type de branchement avant ou arrière), puis les compteurs saturants deux bits (de 00 fortement non pris à 11 fortement pris), avant d'arriver à des euristiques plus complexes utilisant des fonctions de hachage dépendante de l'histoire d'exécution. Il faut néanmoins noter que l'évaluation des temps d'exécutions de séquences de codes avec des prédicteurs de branchements perfectionnés tel les derniers que l'on a cité peut s'avérer complexe.

3.4. Unité de gestion de mémoire (MMU)

Il s'agit d'une unité matérielle qui opère des traductions d'adresses entre des adresses logiques utilisées par un programme et des adresses réelles (ou physiques) qui sont les adresses de stockage effectives dans la mémoire physique de l'ordinateur et aussi la segmentation et la protection des zones d'adressage mémoire. Leur besoin s'est fait sentir avec les systèmes d'exploitation multitâches et multi-utilisateurs, pour des raisons de sûreté de fonctionnement et pour la gestion du swap disque. Depuis le milieu des années 80, ces systèmes sont intégrés aux microprocesseurs destinés aux ordinateurs. Mais dans le domaine des processeurs destinés aux systèmes embarqués grand public, ce n'est pas encore généralisé (car une partie des applications sont encore monotâches - on peut penser à la gamme de processeur Dragonball/Fireball de Motorola, par exemple). Cette unité permet, entre autres choses, de faire exécuter un programme de manière indépendante de son adresse physique de chargement, de façon à lui faire croire qu'il est toujours chargé à la même adresse, qu'il est seul en mémoire ou de lui interdire d'écrire des données en dehors de l'espace mémoire qui lui est alloué. Une autre utilisation, particulièrement appréciée pour la sûreté de fonctionnement des systèmes, est de prévenir qu'une tâche ou un programme particulier ne puisse écrire, ou même lire dans certains cas, des informations dans des zones de la mémoire qui ne doivent pas lui être accessibles (en OASIS, à la fois l'écriture et la lecture sont interdites hors des zones autorisées).

Ainsi, les unités de gestion de mémoire sont extrêmement utiles au niveau de la sûreté de fonctionnement des applications puisqu'elles permettent d'éviter qu'une faute spécifique à une tâche particulière n'ait de conséquences sur les autres tâches.

FIG. 3.5.1. *Principe d'un cache de premier niveau*



Cependant, comme la correspondance avec la mémoire physique n'est pas immédiate, il est nécessaire d'opérer des conversions d'adresses qui ne sont pas toujours instantanées (présence d'un petit cache local). De ce fait cette unité peut aussi générer des aléas sur le temps d'exécution. Les effets sont toutefois marginaux par rapport à ceux dus au cache mémoire proprement dit (au plus un facteur 10^{-3}), du fait de la taille des pages. Il sont de plus corrélés à ceux de la hiérarchie mémoire que l'on va exposer à présent.

3.5. La hiérarchie mémoire

3.5.1. Principe et architecture. Depuis le milieu des années 80, le cœur d'exécution des microprocesseurs dépasse en vitesse la mémoire principale qui est souvent de la RAM dynamique (DRAM) pour des raisons de coût. Comme les programmes ont des propriétés générales qui les amènent à utiliser à des intervalles de temps rapprochés les mêmes données ou instructions (ce qu'on appelle cohérence temporelle des accès mémoires) ou des données ou des instructions proches de celles actuellement utilisées (ce qu'on appelle cohérence spatiale des accès mémoires), il est rapidement apparut qu'il serait avantageux de mettre une petite mémoire rapide entre le cœur d'exécution du microprocesseur et la mémoire principale, comme l'illustre la figure 3.5.1 . En stoquant dans cette petite mémoire rapide appelée «cache» les accès mémoires les plus récents et ce qui en est proche

au niveau des adresses (en stoquant en une seule fois plusieurs adresses consécutives dans une structure appelée ligne ou bloc de cache), cela permet d'obtenir pour un surcoût modeste la vitesse d'une mémoire rapide sur la majorité des accès. C'est le concept général de hiérarchie mémoire d'avoir en partant du processeur et en allant vers les couches extérieures du système des mémoires de plus en plus lentes mais également de plus en plus grandes, par une organisation en *niveaux* de caches. Depuis plus de dix ans, l'utilisation de caches de niveau 1 de petites tailles (quelques Ko typiquement) est intégrée à toutes les architectures. Il n'est pas rare de voir trois niveaux de cache pour les stations de travail actuelles. Le cache de premier niveau (dit L1 - pour Level 1 en anglais) dispose cependant d'une architecture plus originale du fait qu'il y a généralement deux caches de premier niveau (architecture dites Harvard) :

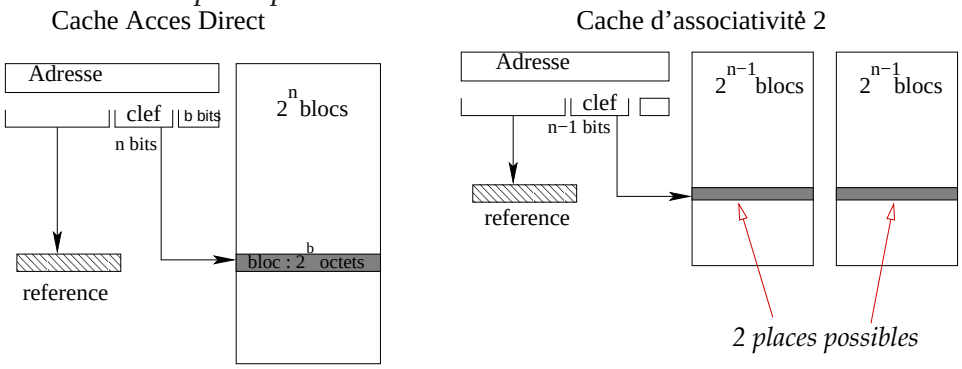
- **le cache d'instructions** : le processeur va y chercher les instructions à exécuter (donc en lecture seule) ;
- **le cache de données** : Le processeur va y chercher en priorité les données lues ou écrites par les instructions exécutées (donc lecture/écriture).

L'avantage de séparer les deux est de pouvoir accéder en parallèle aux instructions et aux données, au niveau du coeur du processeur, d'où un plus faible nombre d'accès simultanés à un cache donné.

Lorsque la donnée ou l'instruction recherchée n'est pas dans le cache on dit qu'il y a échec de cache ; il y a alors un accès à la mémoire centrale, et stockage dans le cache (sauf exception).

La correspondance entre une adresse dans le cache et une adresse dans la mémoire centrale n'est pas immédiate puisque le cache est beaucoup plus petit. Les différentes méthodes existantes sont basées sur deux modèles :

FIG. 3.5.2. *Caches à accès direct et associativité 2 ; schémas de principe*



– cache à correspondance directe(Direct Mapped Cache) : on utilise une partie des bits de poids faibles de l'adresse en mémoire pour calculer la position dans le cache (voir figure 3.5.2). La ligne est directement indexée par ces bits de poids faibles et la présence ou l'absence de la donnée ou de l'instruction dans le cache est déterminée par la comparaison des bits de poids forts référencés lors du stockage de la ligne avec ceux de l'adresse recherchée. Il s'agit de l'architecture la plus simple et la plus rapide mais c'est celle qui engendre le plus de conflits lors des accès ;
– cache associatif par ensembles (Set Associative Cache) : on utilise encore une partie des bits de poids faibles de l'adresse à référencer dans le cache, mais cette fois, il y a plusieurs positions possibles indépendantes pour stoker la donnée ou l'instruction (voir figure 3.5.2), donc un ensemble de positions. On décide habituellement de remplacer la donnée la moins récemment utilisée (politique dite LRU[3]), qui est une des méthodes les plus efficaces lorsqu'on a pas de connaissance *a priori* sur le comportement du programme, et qui est de plus assez

[3] Least Recently Used replacement policy c'est à dire «moins récemment utilisé»

facile à mettre en œuvre, architecturalement parlant. S'il y a deux positions possibles équivalentes, on dit que le cache a une associativité de deux. S'il y en a 4 on dit que le cache a une associativité de 4, et ainsi de suite. S'il y a autant de positions possibles que de lignes de cache (donc la position de stockage dans le cache ne dépend pas de l'adresse mais seulement des accès qui ont eu lieu lors de l'exécution), on dit que le cache est totalement associatif. A l'heure actuelle, l'associativité des caches utilisés dans les microprocesseurs est entre 2 et 8 la plupart du temps ;

Depuis la fin des années 80, il est devenu impensable de fonctionner sans caches à cause des énormes différences entre les horloges des processeurs et les latences des RAM dynamiques. Dans les machines à usage scientifique, la présence d'un cache intermédiaire en vitesse entre les caches au niveau du processeur (dits caches de niveau 1 ou caches L1) et la mémoire centrale s'est rapidement imposée ; ce sont les caches de niveau 2 ou caches L2. On voit même des caches de niveau 2 "*on chip*"[4] pour une partie des processeurs grand public, et des caches de niveau 3 (L3) pour les processeurs hautes performances pour le calcul scientifique ou la gestion de bases de données. Notons au passage que contrairement aux caches de niveau 1, les caches de niveaux supérieurs ont une architecture unifiée (donc instructions et données y sont stockées au même titre). Cela est possible du fait que les accès y sont beaucoup moins fréquents que pour le niveau 1. De même, la taille des lignes de cache dépend du niveau du cache considéré. Pour plus de détails, on pourra se référer à [**14**].

3.5.2. Difficultés de la hiérarchie mémoire. On observe de manière évidente dans les architectures disposant d'une hiérarchie mémoire :

[4]C'est à dire sur la même pastille de silicium que le reste du processeur.

– une dépendance complexe avec les flots de contrôle et les flots de données du contenu des caches et donc des latences d'exécution des tâches (en effet, le contenu des caches dépend en grande partie de l'historique des accès mémoires) ;
– que tout changement de contexte important (par exemple une commutation entre tâches) peut changer le contenu des caches de manière sensible, voire totale (pour les mêmes raisons).

Cela amplifie le non déterminisme du temps d'exécution pour une instruction donnée. On peut quantifier l'impact des caches à environ 80% des cycles d'attente, voire plus sur les architectures RISC à haute fréquence d'horloge qui sont toutes optimisées pour fonctionner avec les caches.

Il ne faut tout de même pas oublier que, malgré les désagréments éventuellement importants causés par l'indéterminisme des temps de traitement, cela reste toujours beaucoup plus rapide que de travailler sans cache. On sait en effet que dès 1990, les processeurs subissent beaucoup plus de cycles d'attente pour des problèmes d'accès à la mémoire que pour des problèmes dus au pipeline[**14**] (et cela reste toujours vrai, même si la complexité des processus d'exécution n'a cessé d'augmenter ces dernières années).

Un des aspects particuliers des caches par rapport aux autres unités architecturales que l'on a abordées est que la commutation de tâches apporte des délais éventuellement assez importants par rapport à une exécution sans commutations. C'est ce que l'on va détailler à présent.

3.6. Problèmes spécifiques du multitâche sur les architectures modernes

C'est cet aspect des programmes temps réel qui posent le plus de problèmes. En effet, le fait de commuter d'une tâche à une autre crée des effets non linéaires quant aux temps d'exécution des instructions, en particulier au

niveau des caches mémoire, ce qui est la source de nombreuses interrogations et la problématique essentielle de cette thèse. On a les faits suivants :

- les instants de commutation de tâche, au niveau du code machine ne sont pas déterministes et, sauf programme très régulier, éventuellement très variables ;
- cela bouleverse de façon importante le contenu des caches ;
- les latences dues à la remise à jour du contenu des caches après une commutation de tâche sont en général très grands devant les autres latences dues à d'autres élément architecturaux (pipeline, MMU, *etc*...).

Pour toutes ces raisons, les aspects multitâches augmentent l'indéterminisme du temps d'exécution pour une tâche donnée. Le problème est donc de trouver des majorants sous ces contraintes.

A noter, tout de même, que si pour les pipelines, on peut facilement majorer les coûts de commutations de tâches à un nombre de cycles processeurs de l'ordre de la profondeur du pipeline en question, cette majoration est nettement plus délicate pour les caches. C'est pourquoi cette thèse est essentiellement concentrée sur les problématiques de caches qui sont beaucoup moins régulières.

3.7. Architectures futures des microprocesseurs

On va également discuter de l'évolution pressentie des architectures des microprocesseurs afin de donner quelques indices *a priori* sur les impacts éventuels que représenteraient ces nouvelles évolutions.

3.7.1. Approche VLIW. C'est la voie mise en avant par l'alliance Hewlet-Packard avec Intel pour la nouvelle génération d'architecture Intel connue sous le nom générique de IA-64 et dont le premier représentant est connu

sous le nom commercial d'*Itanium*. Le concept de VLIW (Very Large Instruction Word), est en fait connu depuis longtemps dans le cadre d'un certain nombre de processeurs dédiés au traitement du signal (DSP[5]) dont les applications embarquées et temps-réel font encore grande consommation (même si les processeurs génériques basse consommation commencent petit à petit à l'emporter, de par leur puissance de calcul de plus en plus grande et leurs capacités de plus en plus étendues). Dans cette voie les instructions sont regroupées entre elles et alignées de façon à ce qu'un tri préliminaire soit fait en fonction du type d'instruction. C'est donc le compilateur, dans cette approche, qui est chargé d'optimiser l'ordonnancement des instructions. Les pipelines sont bien séparés en fonction du type d'instruction exécuté. Dans l'approche Intel/HP, il s'agit d'un concept un peu plus avancé que l'approche VLIW standard et qui vise à reporter une plus grande partie du travail (en particulier l'ordonnancement des instructions et la résolution des dépendances) sur le compilateur. Cette approche a pour nom générique EPIC[6]. Un autre concept fort associé à ce modèle d'architecture, est la présence systématique d'instructions conditionnelles, ce qui permet d'éviter une partie des ruptures de séquence d'instruction dûs au flot de contrôle, car l'ordonnancement statique des instructions limite en partie les possibilités de prédictions de branchements. Avec une telle approche, le compilateur possède toutes les clefs du contrôle des pipelines et donc de la gestion fine des timings à ce niveau. Par contre, l'impact du changement de tâche sur ce type d'architecture ne devrait pas être fondamentalement différent de ce que l'on a dans les architectures actuelles car, en dehors de la génération de code, les problèmes sont les mêmes que pour un processeurs superscalaire classique (pipeline, caches, *etc*...).

[5] Digital Signal Processor

[6] Explicitly Parallel Instruction Computing

3.7.2. Cache de traces. Le cache de traces est un cache d'instructions particulier dans lequel les instructions sont stockées dans l'ordre effectif de leur exécution par le processeur (c'est à dire en suivant les flots de contrôle) plutôt que dans l'ordre dans lequel elles sont stockées en mémoire. Intel a implémenté un cache de trace dans la dernière version prévue de l'implémentation de IA-32, le Pentium 4. Comme il s'agit d'un cache, l'influence de la commutation de tâches sera donc très sensible. Le principe général est donc celui d'une fusion du cache d'instruction avec une sorte de prédiction de branchement, ce qui rend *a priori* les évaluations de temps très complexes. L'avantage d'une telle architecture est de pouvoir obtenir le chargement d'un grand nombre d'instructions en parallèle, et donc d'augmenter le parallélisme potentiellement détecté. Les limites d'une telle approche sont connues mais reste intéressantes pour un chargement d'une dizaine d'instructions et donc des traces de cette taille[**21**] (sauf utilisation de prédictions de données qui peut augmenter le parallélisme potentiel et donc le nombre d'instruction qu'il est intéressant de charger simultanément comme on va le voir). Néanmoins, nous ne traiterons pas d'avantage du cache de trace dans ce présent document.

3.7.3. Prédiction de données. De même qu'il existe des prédicteurs de branchements pour anticiper les aléas du flot de contrôle, les architectes pensent implémenter des prédicteurs de données pour contourner les aléas de données. Cela devrait s'avérer assez efficace pour les données constantes et celles en progressions linéaires, comme les indices de boucles, par exemple. Leur impact devrait être plus délicat à évaluer que les problèmes de prédiction de branchement, mais comme il s'agit d'une technologie qui n'aboutira pas avant plusieurs années, on ne l'évoquera pas d'avantage, non plus, dans ce mémoire.

3.7.4. Processeurs asynchrones. Il s'agit de processeurs qui n'ont plus d'horloge pour cadencer de rythme des opérations. De ce fait, le calcul se poursuit au niveau suivant dès qu'il a fini d'être validé au niveau courant de l'exécution. Cela nécessite l'introduction de systèmes de synchronisations spécifiques à l'intérieur même du processeur, augmentant du même coup le nombre de transistors du processeur, par comparaison avec une architecture synchrone. Cependant cela n'augmente pas la consommation du processeur, car elle s'adapte automatiquement aux besoins de la charge en calculs. C'est une nouvelle voie qui commence à être ouverte, en direction plus particulièrement des marchés de l'informatique embarquée. Pour l'instant il est difficile de dire s'il y a réellement un mouvement qui aboutira à un produit commercial. Mais cela intéresse particulièrement les constructeurs de la gamme de processeurs ARM ainsi que ceux d'IBM dans une variante à "horloge locale".

En ce qui concerne les temps d'exécution, on peut tangentiellement se ramener à ceux d'une architecture synchrone (éventuellement moyennant de légère surestimation de quelques pour cent) du fait même que l'on peut se ramener tangentiellement à un processeur synchrone en forçant une synchronisation des horloges locales (pire cas). Donc cela ne devrait pas poser de problèmes spécifiques.

3.7.5. Processeurs virtuels. J'entends par processeurs virtuels des processeurs spécialisés dans l'émulation du code d'autres processeurs. L'exemple le plus frappant est la gamme de processeur Crusoe de Transmeta. Le processeur dispose de son propre coeur d'exécution et de son propre langage mais il exécute du code spécifique de traduction du code d'une autre architecture (IA32 est mis en avant pour Crusoe, mais le problème serait le

même pour une machine virtuelle Java, par exemple). Dans le cas du processeur Crusoe, par exemple, le processeur dispose d'un équivalent cache d'instructions qui est en fait un cache de traduction. Cette méthode de transformation est de plus cachée entièrement et ce genre de processeur pose deux problèmes dans le cadre d'applications de sûreté :

- Sauf à avoir une coopération étroite avec la société qui édite ce genre de processeur, il est impossible de savoir réellement comment une instruction donnée est traduite au niveau du coeur d'exécution, car il faut disposer du code de traduction qui est une partie essentielle de ce genre de processeur et est donc couverte par des droits de copie, par exemple.
- Une fois cette première difficulté résolue, il faut encore être capable d'en déterminer les temps d'exécution, ce qui est une tâche beaucoup plus délicate que pour des processeurs standards, du fait de l'interaction entre le logiciel de traduction, son exécution et l'exécution du code traduit.

On peut donc penser que sauf motivation économique très forte, les difficultés intrinsèques au concept devraient écarter toute tentative d'utiliser ce genre de processeurs tels quels dans des applications critiques. Il faut noter que ce n'est pas le langage utilisé qui pose un problème (on pourrait éventuellement envisager d'utiliser un langage dérivé de Java sur un système critique, à la condition impérieuse d'être capable de maîtriser l'intégralité du code final qui s'exécute de manière native).

Conclusions sur la problématique

Nous avons fait, dans cette première partie, une reconnaissance rapide des raisons qui amènent à vouloir déterminer des temps d'exécution au pire

pour les application temps-réel critique et les difficultés que pouvaient engendrer l'architecture des microprocesseurs à cet égard. Nous avons dégagé aussi le fait que sur les architectures actuelles les caches mémoire sont les sources d'irrégularités temporelles les plus critiques. C'est donc sur ce problème particulier que nous allons poursuivre cet exposé, en commençant par une introduction à l'état de l'art en la matière.

Deuxième partie

Un aperçu de l’état de l’art

CHAPITRE 4

État de l'art pour les majorations de WCET : les modèles

L'évaluation des temps d'exécution au pire (WCET[1]) pour les processeurs pourvus de cache est un problème qui intéresse la communauté des systèmes temps-réels depuis que les microprocesseurs intègrent couramment des caches. Depuis un peu plus de dix ans, des articles ont commencés à paraître sur ce sujet particulier. On va donc faire dans ce chapitre un tour rapide de ce qui est fait dans le domaine.

4.1. Approche pragmatique

C'est ce que font la plupart des ingénieurs dans le domaine du temps-réel jusqu'à récemment.

Il s'agit de faire une mesure expérimentale des temps d'exécution avec des jeux de données d'entrée «bien choisis» et avec des invalidations de caches rajoutées à la main dans le code et «judicieusement placées».

Le problème le plus évident avec cette technique est qu'elle interdit toute notion claire de démonstration quant à la validité des choix effectués et des résultats obtenus. Dans le cadre de la sûreté de fonctionnement des applications temps-réel, il est nécessaire de suivre une piste plus systématique.

Une autre méthode triviale est de faire des tests en invalidant les caches, mais cela revient à sur-dimensionner le système de manière drastique, et donc la difficulté de conception et les prix de revient. C'est pour cette raison que depuis une dizaine d'années les équipes de recherche du domaine

[1] Worst Case Execution time

du temps-réel critique cherchent des méthodes sûres pour donner des majorants de temps d'exécution en tenant compte des aléas dus aux nouveautés architecturales des processeurs, et en particulier des caches.

4.2. Approche prédictive par interprétation abstraite

4.2.1. Idées générales. Il s'agit de la principale démarche mise en avant, dans la plupart des articles récents sur le sujet[**1**][**10**][**11**][**25**], ou des méthodes basées sur des principes similaires [**20**].

4.2.1.1. *Principes de base de l'interprétation abstraite.* L'article fondateur de l'interprétation abstraite de Cousot et Cousot date de 1977 [**5**] et décrit comment transformer un état concret sur un domaine abstrait pour calculer un état abstrait qui donne des résultats intéressant pour l'état réel. Un exemple simple est la multiplication de deux réels, que l'on va détailler à présent :

Supposons donc que l'ensemble des réels $\mathbb{R}$ soit l'ensemble concret d'origine. On défini un ensemble abstrait $\{-, 0, +\}$ dont les éléments correspondent pour $\mathbb{R}$ à, respectivement, $\mathbb{R}^*_-$ (ensemble des réels strictement négatifs), le singleton $\{0\}$ et $\mathbb{R}^*_+$ (ensemble des réels strictement positifs). Si l'on fait correspondre tout élément de $\mathbb{R}^*_-$ à $(-)$ et tout élément de $\mathbb{R}^*_+$ à $(+)$, on obtient les règles suivantes pour l'interprétation abstraite de l'opération de multiplication dans $\mathbb{R}$:

$$\begin{cases} \bullet \quad (+) \times (+) = (+) \\ \bullet \quad (-) \times (-) = (+) \\ \bullet \quad (-) \times (+) = (-) \\ \bullet \quad (+) \times (0) = (0) \\ \bullet \quad (-) \times (0) = (0) \end{cases}$$

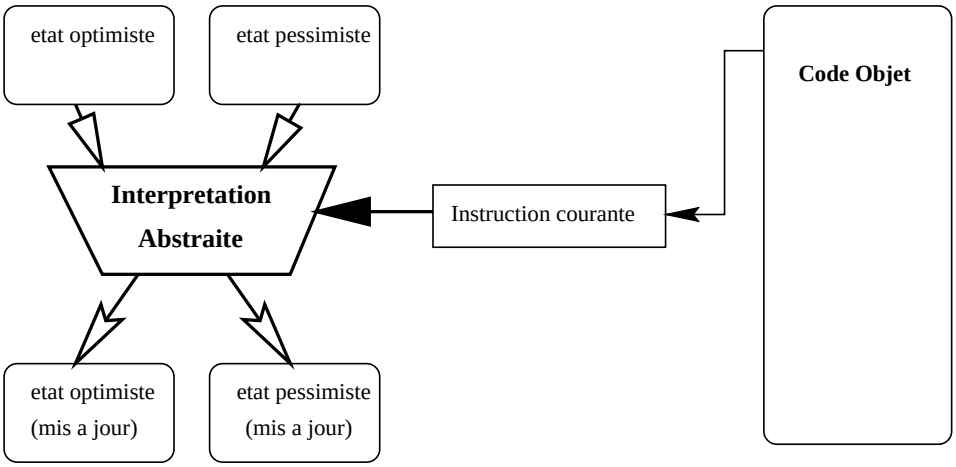
ainsi que les relations que l'on peut déduire par commutativité.

L'idée générale est donc de déduire une propriété de l'état concret résultant d'une opération en appliquant une opération similaire aux états abstraits initiaux. Dans l'exemple exposé ici, on peut citer par exemple que le produit d'un réel positif par un réel négatif est un réel négatif. La propriété mise en avant est la position du réel par rapport à zéro (ce que traduit l'espace abstrait associé), et l'opération considérée est la multiplication de deux réels. Le choix de l'espace d'abstraction conditionne en partie les propriétés que l'on va pouvoir déduire sur l'état concret et dans une certaine mesure influence les modèles d'opérations à appliquer sur l'espace abstrait ainsi défini.

4.2.1.2. *Application au cadre de l'évaluation des temps d'exécution au pire (WCET).* Le principe est de faire une exécution symbolique du code, instruction après instruction (correspondant à des opérations à définir), afin de calculer des états abstraits du système matériel considéré. Dans les publications que l'on avait pu voir jusqu'à présent, il s'agissait de déterminer des états optimistes et pessimistes des caches par cette méthode en utilisant des règles algébriques simples.

A chaque étape, on dispose donc d'un état optimiste et d'un état pessimiste des caches (en général initialement vides pour les deux cas). On utilise l'instruction courante (la plupart du temps, seules les instructions agissant sur le flot de contrôle sont prises en compte, car cette méthode est rarement généralisée au cache de données du fait de la complexité des accès à celui-ci) dans le parcours du code objet pour engendrer un nouvel état optimiste et un nouvel état pessimiste pour chaque cache, le processus étant schématisé en figure 4.2.1 . On défini une sémantique abstraite permettant de passer d'un état donné du cache à un état après exécution de l'instruction. Cette sémantique dépend de différents paramètres, comme l'architecture du cache,

FIG. 4.2.1. *modèle d'analyse par interprétation abstraite*



le degré de prise en compte des aléas du flot de contrôle[2], de la politique de remplacement des lignes de cache, *etc*...

L'exécution d'un code vis à vis de l'influence sur l'état des caches peut être résumée à deux grands types d'opérations :

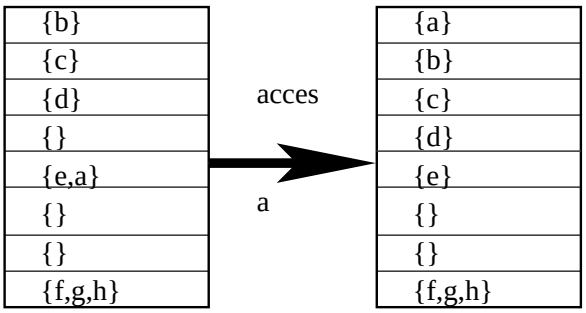
Les accès mémoire: qui correspondent dans le mécanisme LRU au chargement du bloc considéré dans le cache, en tête d'historique pour l'ensemble auquel il appartient et à l'obsolescence de tout les autres dans l'historique pour les blocs qui étaient mieux placés dans l'historique avant que l'accès n'ait lieu. Il faut noter que dans le cas d'un accès de données, un tel mécanisme peut être compliqué par le fait que le flot de données[3] n'est pas forcement déterministe, *a priori* (aléas de flot de données) ;

Les fusions de chemins d'exécution: qui correspondent à la manière de résoudre de manière statique les aléas du flot de contrôle du

[2]Le flot de contrôle est le concept qui fait référence à la succession des instruction dans l'ordre d'exécution d'un élément de programme donné, dans un contexte d'exécution donné.

[3]Le flot de données est en fait un concept qui fait référence à la succession des accès de données -autant par leur position en mémoire que par leur ordonancement temporel- lors de l'exécution d'un élément de programme.

FIG. 4.2.2. *Must analysis* pour un accès mémoire sur *a*

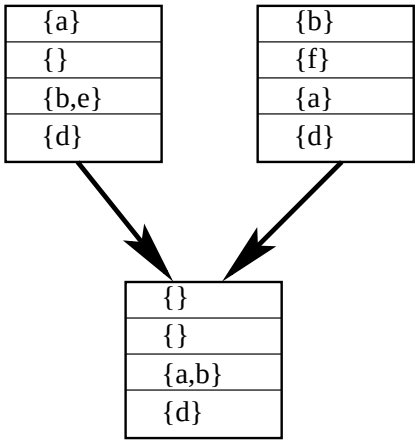


fait qu'il n'est pas toujours possible de déterminer *a priori* quel branche d'une condition sera effectivement exécutée, par exemple (aléas de flot de contrôle).

Souvent, la façon de calculer l'état pessimiste du cache est appelée *Must Analysis* dans la littérature, que l'on pourrait traduire par analyse impérative ou analyse au pire tandis que la façon de déterminer l'état optimiste des caches est appelée *May Analysis*, que l'on traduira par analyse potentielle ou analyse au mieux et que l'on va maintenant détailler.

4.2.2. Must analysis - calcul d'un historique d'accès au pire. Pour la *must analysis*, on considère pour chaque référence possible une collection de A ensembles (pour un cache d'associativité A) que l'on appellera c_h pour h entre 1 et A. Les c_h correspondent au fait que les références qu'il contiennent ont pour âge maximum h au sens du mécanisme LRU. Ainsi, lors de l'accès à une référence r, celle-ci est placée dans l'ensemble c_1 et les références précédemment sont transférées dans c_2 (sauf r si elle s'y trouvait déjà) et ainsi de suite, jusqu'à atteindre un c_h dans lequel se trouvait déjà r s'il y en a ou jusqu'à c_A sinon (les anciennes références de c_A étant bien évidemment perdues). Une illustration du principe est faite en figure 4.2.2 . L'autre cas où l'on met à jour ces ensembles correspond au cas de la réunion de flot de contrôle, par exemple, lorsque l'on sort d'une alternative *if*...

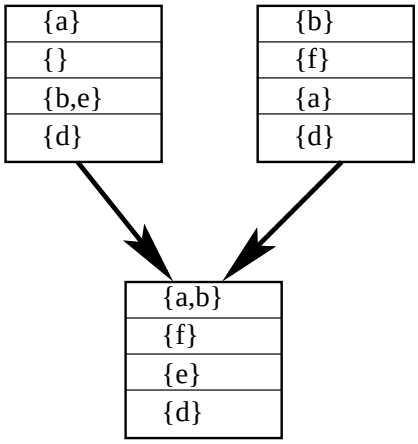
FIG. 4.2.3. Must analysis *lors de la reconvergence de flots de contrôle*



then... *else* pour revenir à la branche principale. On prend alors l'intersection pessimiste des ensembles issus des deux branches, c'est à dire que l'on prend dans la comparaison de la collection des ensembles les références qui sont communes aux deux branches et on les place dans les ensembles d'indice le plus élevé rencontré à chaque fois (figure 4.2.4) . On peut noter que, par construction, la *must analysis* conduit à des ensembles possédant au plus A références par ensemble (avec A l'associativité du cache et donc la taille de l'ensemble), c'est à dire que le cardinal de l'union des ensembles de la collection obtenue par *must analysis* est inférieure ou égale à A.

4.2.3. May analysis - calcul d'un historique d'accès au mieux. On peut également introduire la *may analysis* dont le principe est le même à ceci près que lors de reconvergences du flot de contrôle, on prend l'union optimiste des deux collections correspondantes. Ainsi on prend toutes les références présentes, même si elles ne sont rencontrées que dans une seule des deux voies et on les place dans l'ensemble résultant qui correspond à l'indice i le plus faible pour chaque référence. Sur un accès, comme en figure 4.2.2, on a exactement le même résultat, par contre sur la fusion de flot de contrôle, on aboutit au résultat présenté en figure 4.2.4 . On note

FIG. 4.2.4. May analysis *lors de la reconvergence de flots de contrôle*



qu'on a alors un minimum de A références pour un ensemble à A positions (cache d'associativité A), ce que l'on peut exprimer comme le fait que le cardinal de l'union des ensembles de la collection obtenue par *may analysis* est supérieure ou égale à A.

4.2.4. Obtention de temps d'exécution au pire. Lorsque l'on a fini de parcourir le code objet, on détermine l'accélération due au cache en calculant le nombre d'échecs de cache évités en utilisant comme base les états pessimistes du cache afin d'obtenir une accélération dans le pire des cas, et donc un temps d'exécution dans le pire des cas($WCET$[4]).

Il faut noter que dans [**1**], la prise en compte des aléas est minimale (pas de vrais aléas du flot de contrôle, prise en compte des données seulement pour les adresses que l'on sait déterminer clairement lors du parcours du code, sans l'exécuter réellement). Il existe des versions plus élaborées de la prédiction par interprétation abstraite[**10**] dans lesquelles on tient compte des boucles, des fonctions et procédures, éventuellement récursives, mais

[4] Worst Case Execution Time soit effectivement temps d'exécution dans le pire cas.

reste attaché à la seule description du comportement du cache d'instructions. Le cas du cache de données est abordé dans [**27**], mais on aura l'occasion de revenir en détails plus tard sur le sujet.

4.2.5. Mise en oeuvre de la méthode. Cette partie est inspirée essentiellement d'un rapport technique du groupe de l'université d'Uppsala [**8**]. L'avantage de présenter la méthode du groupe d'Uppsala étant qu'ils ont un but industriel, donc une volonté d'utilisation sur du code réel, destiné à être intégré à un projet d'outils de développement.

Les hypothèses de base pour l'analyse de temps d'exécution dans le pire cas(WCET), est que l'exécution du programme est ininterrompue et aucune interférence d'activité extérieure comme des accès DMA ou de rafraîchissement de DRAM ne sont pris en compte dans ce calcul. L'obtention d'un WCET se passe en trois étapes.

Il s'agit dans un premier temps d'obtenir un code objet ainsi que des informations sur le nombre maximum d'itération pour chaque boucle, les fonctions appelées, bref ce qu'on peut rassembler comme informations sûres concernant le flot de contrôle. Il est à noter que cela peut exiger la modification du code du compilateur utilisé pour avoir des informations fines au niveau de l'optimisation du code et sa correspondance avec le code source. En effet, si certaines informations de contrôle sont aisément extraites du code source, ce n'est pas la même chose sur du code final optimisé - l'optimisation du code engendré par un compilateur pouvant conduire à des transformations radicales du flot de contrôle initial (ex : pipeline logiciel, développement de boucles, inclusion de code de fonctions, *etc*...)[**14**]. Mais, par ailleurs, l'information sur le temps d'exécution ne peut être accessible qu'à partir du code objet engendré final qui est le seul à avoir un sens vis à vis de l'exécution par un processeur.

Ensuite, on découpe le programme en blocs fondamentaux dont on détermine, pour chaque, un temps d'exécution. Ces blocs fondamentaux sont issus de l'analyse du flot de contrôle faite à l'étape précédente. La détermination des temps d'exécution pour chaque bloc se fait au niveau du code exécutable engendré par la compilation. Il s'agit donc d'une analyse de bas niveau qui doit tenir compte de l'architecture complète du processeur cible, en particulier les pipelines, les caches, les différentes unités d'exécutions, les prédicteurs de branchement, la MMU, les renomages de registres, *etc*... et de manière générale toutes les structures de l'architecture du microprocesseur cible qui peuvent influer sur le temps de transit d'une instruction donnée dans le coeur d'exécution d'un microprocesseur. Il ne faut pas perdre de vue que toutes ces structures font que le temps total d'exécution d'une instruction donnée dépend fortement de son contexte. Dans l'approche faite par l'équipe d'Uppsala, ce temps est obtenu par l'utilisation d'un simulateur que fournit le fabriquant du microprocesseur. On donne un scénario d'exécution en entrée et à chaque fois que des informations sont manquantes, on choisit le cas le pire pour sa détermination (à ce niveau là, on peut utiliser l'interprétation abstraite). Ainsi, on obtient un WCET pour chaque bloc élémentaire du flot de contrôle. En revanche, le cas du cache de donnée, par exemple, n'est pris en compte que dans l'aspect de localité des accès à deux données successives.

Enfin, ces blocs sont combinés, en accord avec le flot de contrôle, pour pouvoir déterminer le chemin qui donnera le temps d'exécution au pire (WCET). Ce temps n'est évidemment pas une simple somme des temps calculés sur les blocs de base du fait de la présence des pipelines et des caches. Le deuxième point délicat est de faire une analyse correcte des chemins d'exécution possibles, et d'écarter les chemins impossibles, le nombre

maximum d'itérations des boucles, *etc*... C'est là qu'il est important d'avoir des informations précises que l'on ne peut trouver simplement, en fait, que dans le code source, et donc l'utilité de pouvoir retracer l'équivalence entre le code source et le code objet.

4.2.6. Les résultats de la méthode. Cette méthode est assez efficace, et on peut arriver à l'implémenter sans rencontrer de grandes difficultés intrinsèques. Plusieurs groupes de recherche ont d'ailleurs implémenter des outils plus ou moins finalisés permettant de trouver des majorants de temps d'exécution au pire sur des modèles plus ou moins réalistes. Ces méthodes sont plus ou moins avancés mais ont des résultats assez proches. Les points intéressants en fin de compte pour leur application sont que :

- on obtient des majorants stricts ;
- les derniers résultats donnent une surestimation de l'ordre de 20% par rapport aux résultats expérimentaux (bien que la surestimation effective varie fortement selon le type de programme, entre 5% et 40%, environ), bien meilleurs qu'avec toutes les méthodes sûres que l'on pouvait utiliser précédemment, parfois moins lorsque les pipe-lines sont pris en compte (mais il s'agit dans tout les cas de programmes de tests relativement réguliers) ;
- c'est une méthode sûre.

L'inconvénient majeur de cette méthode est que cela marche uniquement pour du code séquentiel pur, et les résultats donnés sont faits la plupart du temps sur des programmes de tests relativement réguliers. De plus, un certain nombre d'hypothèses doivent être faites sur les états initiaux, en particulier, pour pouvoir faire les prédictions d'échecs de cache. De plus, dans un cadre multitâche, il faut aussi tenir compte des commutations de tâche

qui vont introduire des échecs de caches supplémentaires et donc augmenter le temps d'exécution au pire, cependant le passage au cadre multitâche est loin d'être trivial. On donnera quelques éléments bibliographiques à ce sujet au paragraphe 5.2.2.

4.3. Approche Probabiliste

Le but est donc de chercher des lois statistiques sur les accès aux caches, par exemple, en construisant un modèle théorique qui permet d'effectuer des calculs de probabilités.

C'est une approche assez peu répandue dans la littérature. Elle est d'avantage utilisée dans le domaine de l'architecture, bien qu'elle y soit également marginale. En fait c'est plus une méthode théorique de justification a posteriori de choix architecturaux qui sont souvent faits le plus souvent a priori sur des bases expérimentales (par simulation, sur des modèles d'architectures à implémenter). Les architectes préfèrent de loin la simulation aux calculs statistiques et probabiliste, en général.

On peut cependant trouver de bons articles sur des modèles d'accès en mémoire éventuellement intéressants que nous détaillerons au prochain chapitre.

CHAPITRE 5

Modèles d'accès aux caches - État de l'art, suite

5.1. Approche naïve

On peut trouver des lois naïves quant au taux d'échec pour un cache de taille donné. En faisant une hypothèse simple d'accès aléatoire aux données (modèle IRM[1][**3, 22, 12**]), on aboutit au résultat suivant :

THEORÈME 5.1.1. *Pour une taille de cache S donnée le rapport R entre le nombre de référence au cache et le nombre d'échec d'accès au cache est donné par :*

$$R = K\sqrt{S}$$

où K est un paramètre qui ne dépend pas de S (mais éventuellement de la structure du cache et du programme exécuté et des données manipulées).

Cela est particulièrement intéressant, pour un programme donné au départ, ayant connaissance du rapport R_1 pour une taille de cache S_1, pouvoir en déduire facilement R_2, pour une taille de cache S_2, sur laquelle on n'a pas d'information a priori :

$$\frac{R_2}{R_1} = \sqrt{\frac{S_2}{S_1}}$$

Il faut noter que si ce résultat est souvent assez bon pour les accès de bases de données vis à vis des caches disques, elle est beaucoup trop simple pour le genre de problèmes qui nous intéressent. Il suffit de constater, par

[1] Independant Reference Model

exemple, que le résultat est indépendant de l'architecture du cache (associativité, taille des blocs, *etc* ...) pour constater que c'est un modèle très grossier. Pour les cas qui nous intéressent du comportement d'un cache vis à vis de l'exécution d'un élément de programme, on ne peut pas, en général considéré que les accès sont aléatoires et indépendants.

5.2. Cache d'instructions

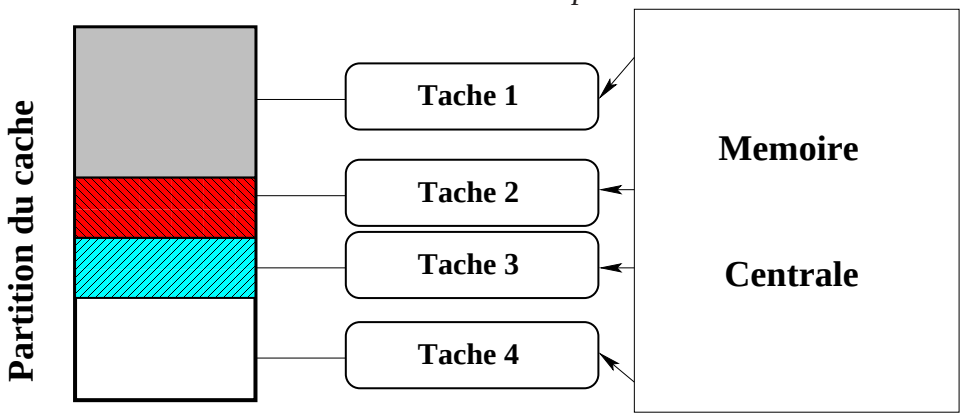
5.2.1. En exécution séquentielle. On peut avoir une bonne approche de ce cas grâce à l'analyse par interprétation abstraite et des méthodes dérivés, avec un minimum d'approximations, comme on l'a vu au paragraphe 4.2. En ce qui concerne les caches, il s'agit de chercher à voir comment le cache est géré par la tâche elle-même, c'est à dire dans quelle mesure une donnée ou une instruction chargée dans le cache est éventuellement réutilisée par la suite, ou éjectée du cache avant réutilisation. On appellera ce phénomène interférence intra-tâche.

5.2.2. En exécution multitâche. En exécution multitâche, il existe une source d'invalidation de cache supplémentaire. Il s'agit des invalidations de caches qui ont lieux lorsque des données ou des instructions normalement dans le cache lors d'une exécution linéaire sont éjectées des caches à cause d'une exécution d'une autre tâche entre le moment ou elles sont normalement chargées dans le cache et le moment de leur réutilisation. On appellera de tels effets sur les caches, et donc en définitive sur le *WCET*[2], interférences inter-tâches. Cela s'oppose évidemment à des interférences intra-tâche.

Il existe une méthode classique pour prévenir les interférences inter-tâches. Il s'agit de la méthode de "partition de cache" qui consiste à faire en

[2] Worst Case Execution Time

FIG. 5.2.1. *Une illustration de partition de cache*



sorte que chaque tâche a exécuter ne puisse se mettre dans le cache qu'en des endroits où les autres ne se logent pas. Cela revient en fait à faire en sorte que les bits de poids faible des adresses auxquels accèdent la tâche soient dans un segment donné des clefs de référencement des lignes de cache, de façon à opérer une partition des ces clefs sur les différentes tâches. Une illustration du principe est faite en figure 5.2.1. Il va de soit que c'est une technique beaucoup plus simple à mettre en oeuvre pour les données que pour les instructions car il est plus à la portée du programmeur de gérer l'agencement des données que l'agencement d'un programme. C'est aussi une technique plus délicate à mettre en oeuvre lorsque l'associativité du cache augmente ou que le nombre de tâches augmentent. C'est pourquoi, il est en général nécessaire de gérer les effets d'interférences inter-tâches. On trouve dans [**16**](largement inspiré par un travail précédent de Busquet-Mataix et Al.[**4**]), des éléments de réflections intéressants pour la prise en compte de cet effet. Il s'agit de trouver les points d'une tâche donnée qui donnent les plus grandes pénalités sur le cache d'instructions, en utilisant l'hypothèse simplificatrice la plus radicale, c'est à dire que la commutation de tâche revient à invalider le cache d'instructions (seuls les effets pour le cache d'instructions sont pris en compte dans cet article). Ces calculs ont

été effectués dans l'hypothèse simple d'un système «multitâche à priorités fixes et statiques», qui est un modèle assez simple voire simpliste, mais qui ne correspond pas au modèle OASIS. Il n'est tenu aucunement compte du problème du cache de données qui est un problème complexe du fait de l'interaction entre flot de contrôle et flot de données.

Le modèle de base est l'approche du temps de réponse dans le pire cas. Dans ce modèle on utilise l'équation de récurrence suivante pour déterminer le temps de réponse dans le pire cas R_i de la tâche T_i :

$$R_i = C_i + \sum_{j \in H(i)} [\frac{R_i}{\tau_j}] C_j$$

où

H(i): est l'ensemble des tâches T_j de priorité supérieure à la tâche T_i.

C_x: est le temps propre d'exécution de la tâche x.

τ_i: est la période de la tâche T_i.

On voit ainsi que le terme $\sum_j [\frac{R_i}{\tau_j}] C_j$ est le temps total dû aux préemptions des tâches de priorité supérieure à la tâche T_i. Il s'agit là d'un modèle simpliste ne prenant pas en compte les interactions inter-tâches qui viennent de la présence des caches. C'est pourquoi il faut élaborer le modèle de base pour en tenir compte, en écrivant :

$$R_i = C_i + \sum_{j \in H(i)} [\frac{R_i}{\tau_j}] \times (C_j + \gamma_j)$$

$$= C_i + \sum_{j \in H(i)} [\frac{R_i}{\tau_j}] C_j + \Gamma_i(R_i)$$

où γ_j est le coût de préemption qu'une tâche de la priorité de T_j peut imposer à celle de priorité inférieure. Par conséquent, on en déduit $\Gamma_i(R_i)$ qui est le coût total de préemption de la tâche T_i par toute les autres tâches de priorités supérieures, pendant R_i, son temps apparent d'exécution. La difficulté est de déterminer ce facteur, bien sûr. En particulier, telle qu'elle est écrite en dernier, on voit que cette expression dépend de l'évaluation faite : il s'agit de trouver un point fixe. Les hypothèses faites sur le système permettent de s'assurer qu'une solution existe (il n'y aura pas de divergence). On peut donc utiliser une récurrence simple sur chaque R_i pour trouver le point fixe (méthode classique de résolution de point fixe d'une suite de type $u_{n+1} = f(u_n)$) :

$$R_i^{k+1} = C_i + \sum_{j \in H(i)} [\frac{R_i^k}{\tau_j}] C_j + \Gamma_i(R_i^k)$$

Et l'on s'arrête simplement lorsque $R_i^{k+1} = R_i^k$ pour un i donné. Mais il faut tout de même trouver une méthode pour évaluer au moins un majorant de $\Gamma_i(R_i^k)$.

Pour cela, le groupe coréen utilise une méthode proche de l'analyse abstraite pour déterminer le nombre de blocs de cache utiles. Il s'agit, en chaque point d'exécution de la tâche de déterminer le nombre de bloc de

cache utiles, c'est à dire qui seront réutiliser dans le futur, et ainsi de faire un classement des points où il est le plus coûteux pour la tâche d'être interrompue, permettant ainsi de majorer le nombre d'échecs de cache surnuméraires induit au maximum par une, puis deux, puis trois préemptions de la tâche, et ainsi de suite. Cela nécessite bien-sûr d'analyser les tâches dans l'ordre normal de leur exécution, comme dans l'analyse abstraite classique, mais aussi dans l'ordre inverse, afin de pouvoir déterminer les blocs de cache utiles. Cette analyse permet de connaître le coût maximal en fonction du nombre de préemptions. Il s'agit ensuite de déterminer un scénario de préemption au pire à partir des données obtenues. Cela n'est pas toujours faisable de manière exacte, mais on peut toujours trouver un jeu de contraintes linéaires se basant sur les hypothèses données (comme les périodes des différentes tâches, ou le fait que la tâche de plus grande priorité ne peut-être préemptée, ...) de façon à ce qu'après résolution du système, on ait un majorant sûr mais peu surestimé de $\Gamma_i(R_i^k)$. Cette méthode donne de bons résultats, dans le cadre du modèle précis dans lequel il s'inscrit. En particulier sur les temps dus aux interférences inter-tâches, malgré l'hypothèse simpliste d'invalidation totale du cache lors d'une commutation de tâche. Elle est cependant extrêmement coûteuse en temps de calcul et exige d'être adaptée si l'on veut sortir du cadre du «multitâche à priorité fixe», et cela ne peut aller que dans le sens d'une lourdeur accrue des calculs.

5.3. Cache de données

5.3.1. Modèle général. Le cas général de l'accès aux données est délicats à traiter car s'il n'est pas aléatoire, il est beaucoup moins régulier que l'accès aux instructions, du fait en partie que le flot de contrôle a également une incidence sur le flot de données, au final. Néanmoins, si l'on regarde simplement le cas de boucles imbriquées à bornes dépendantes des indices

FIG. 5.3.1. *Boucles imbriquées, modèle de base*

```
Do j1=0,N1-1
    Do j2=0,N2-1
        Do j2=0,N2-1
            ...
        Enddo
    Enddo
Enddo
```

de boucles, rien que ce cas possède une complexité propre déjà très grande. Heureusement, c'est un cas a priori rare dans le contexte qui nous intéresse du temps réel critique où les bornes de boucles doivent être contrôlées avec rigueur.

5.3.2. Cas particulier des boucles imbriquées monotones croissantes à bornes fixes. Ce cas a été traité en détail par des articles récents dans le domaine de l'architecture[**13**] (cet article étant le plus complet en date puisqu'il traite également du cas des caches associatifs, contrairement aux articles précédents, comme l'article originel de Temam et Al de 1994[**24**]).

Le point de départ de cette méthode, pour un ensemble de boucles imbriquées monotones croissantes à bornes fixes est de les transformer pour obtenir la forme donnée en figure 5.3.1. On peut classer les échecs de cache de manière générale en trois catégories :

- Les échecs forcés qui ont lieux lors du premier accès à l'élément référencé.
- Les échecs de capacité lorsque le cache ne peut contenir physiquement la totalité de l'ensemble d'éléments avec lesquels travaille le programme

– Les échecs de conflits dus au fait qu'au moins deux éléments de l'ensemble de travail ont le même espace cible dans le cache (le nombre dépendant de l'associativité du cache).

On peut regrouper ces deux derniers types d'échecs en temps qu'échecs d'interférences. Ils dépendent de l'architecture du cache et ont donc une nature extrinsèque (alors que les échecs forcés sont intrinsèques à l'utilisation de caches).

Le modèle d'accès aux tableaux de données considère une forme générale conforme au cas suivant pris pour un tableau X donné :

$$X(\alpha_1 j_{\gamma_1} + \beta_1, \ldots, \alpha_m j_{\gamma_m} + \beta_m)$$

Où les α, β et γ sont des constantes données.

Ce qui correspond à un accès en mémoire à une adresse donnée par l'expression suivante :

$$B + A_1 j_1 + \ldots + A_n j_n$$

Où B et les A_i sont des constantes.

A partir de là, l'article donne une méthode pour calculer une *empreinte des accès* (ou *empreinte mémoire*) sur le cache de données en fonction de son architecture (associativité, taille de blocs, ...), de l'adresse de base B et du vecteur $\overrightarrow{A}$ (défini par les A_i), par un processus récursif. Enfin, les diverses sources d'interférences sont prises en compte pour déterminer, au final, le nombre d'échecs de cache lors de l'exécution des boucles imbriquées. Il faut noter que les interférences externes (dues à des références n'appartenant pas au même groupe de translation, c'est à dire ceux dont les vecteurs $\overrightarrow{A}$ diffèrent) étant complexes sont assez mal gérées par ce modèle.

5.3.3. Remarques générales sur l'état de l'art. Il faut noter que sur des processeurs modernes, un échec de cache de données pour un programme correctement optimisé peut ne pas provoquer de retard à l'exécution (cas des processeurs à exécution non ordonné), si le compilateur arrive à séparer suffisamment l'accès à la mémoire et son utilisation par les unités de calculs (par exemple par l'utilisation de déroulement de boucles ou de pipeline logiciel, ... confère [**14**]). Pour des processeurs moins optimisés, tout échec de cache provoque des cycles d'attente.

C'est la raison pour laquelle, il n'est pas possible d'opérer ce genre de calculs de taux d'échec de façon naïve sur le code source, mais forcement et uniquement sur le code objet final. Le nombre d'échecs des caches peut-être calculé de façon indépendante de l'architecture cible, mais pas les délais associés pour les raisons que j'ai précédemment exposées. Un premier effort de prise en compte du cache de données pour l'évaluation du WCET a été fourni dans [**27**]. Il a l'avantage de donner quelques pistes pour du travail sur un code final réel et optimisé.

Cependant, il n'existe, à notre connaissance, pas de modèle d'accès au cache de données prenant en compte des aspects multitâches, ou de préemptions, de manière plus générale. Il est d'ailleurs difficile d'étendre le modèle précédemment exposé dans ce cadre. Bien sûr, on peut toujours utiliser l'hypothèse d'invalidation totale du cache de données lors d'une préemption, ce qui de toute façon est une méthode sûre.

5.4. Conclusion provisoire sur les modèles d'accès aux caches

Comme on l'a vu au cours de ce chapitre et du précédent, les caches constitue une source majeure des aléas de temps d'exécution des applications. Dans le cadre des applications temps-réel critique, il est impératif d'être capable de les maîtriser. Les modèles aujourd'hui disponible

montrent de bons résultats sur du code séquentiel simple, en partant de l'hypothèse de caches invalides. Par contre, on est loin d'avoir un modèle applicable réellement dans le cadre d'un système multitâche.

On va à présent exposer le modèle théorique développé au cours de cette thèse, afin de voir en quoi il améliore les résultats de la littérature.

Troisième partie

Le modèle et la base théorique développée

CHAPITRE 6

Interprétation abstraite à base de chaînes de Markov

6.1. Principe général

La méthode que l'on a développée est également une méthode d'interprétation abstraite mais beaucoup plus orientée vers le domaine des probabilités que les méthodes algébriques que l'on voit généralement dans la littérature. Cela rend le calcul sensiblement plus lourd en terme de temps de calcul mais ouvre des perspectives tout à fait inédites comme ce chapitre et le suivant vont le montrer. Le modèle reste cependant très accessible puisque qu'il repose sur un formalisme dérivé des chaînes de Markov qui est un des plus simple formalisme markoviens. On va décrire dans ce chapitre la modélisation, le formalisme de base, la mise en œuvre de la méthode et quelques résultats dans le cadre simple des systèmes monotâches, avec une vue particulière sur les systèmes bouclés qui constituent notre cible principale.

6.2. Les espaces d'états

6.2.1. Partition mémoire. Comme on l'a vu, le cache regroupe un certain nombre d'adresses consécutives de la mémoire principale dans une ligne de cache, ou bloc. Considérons une division de la mémoire en blocs tels qu'ils pourraient être transférés dans le cache considéré (cela dépend de l'architecture du cache, et en particulier du niveau de cache en question). On peut numéroter les blocs ainsi définis en mémoire par l'adresse de leur

contenu divisée par la taille du bloc en octet. On nommera *référence mémoire* (en abrégé cela sera simplement *référence*) le nombre ainsi obtenu. Par exemple, l'adresse 136 pour une taille de bloc de 8 octets correspond au bloc de référence (mémoire) 17 et l'adresse 252 au bloc de référence (mémoire) 31.

6.2.2. Espace d'état de probabilité. Cette partition de la mémoire étant faite, on associe à chaque référence i un vecteur d'état de probabilité dans un espace vectoriel $\mathcal{P}_i$ de dimension $A+1$ pour un cache d'associativité A et que l'on définit de la façon suivante :

$$V \in \mathcal{P}_i,\ V = \begin{pmatrix} p_1 \\ \vdots \\ p_A \\ p_\emptyset \end{pmatrix} \ avec \begin{cases} p_1\ probabilité\ de\ prochain\ accès\ en\ historique\ 1 \\ \vdots \\ p_A\ probabilité\ de\ prochain\ accès\ en\ historique\ A \\ p_\emptyset\ probabilité\ de\ prochain\ accès\ en\ échec\ de\ cache \end{cases}$$

les accès en historique n, $n \in \{1, 2, \ldots A\}$ étant à comprendre au sens LRU du terme.

6.2.3. Espace d'état de nombre d'accès et espace d'état réduit correspondant. Pour obtenir une description complète de l'état pour le bloc mémoire considéré, il faut également connaître les nombres d'accès correspondant. On définit alors un deuxième espace vectoriel $\mathcal{N}_i$ de dimension $A+1$ pour le vecteur d'état de nombre d'accès :

$$W \in \mathcal{N}_i,\ W = \begin{pmatrix} n_1 \\ \vdots \\ n_A \\ n_\emptyset \end{pmatrix} \ avec \begin{cases} n_1\ nombre\ total\ d'accès\ en\ historique\ 1 \\ \vdots \\ n_A\ nombre\ total\ d'accès\ en\ historique\ A \\ n_\emptyset\ nombre\ total\ d'échecs \end{cases}$$

A noter qu'avoir un tel nombre de variables peut se révéler comme étant un détail inutile pour le modèle. Si on le souhaite, il est donc possible d'utiliser une version très simplifiée de l'espace $\mathcal{N}_i$ de dimension 2 seulement, en réduisant le vecteur V au couple $(n_h, n_\emptyset)$ avec $n_h = \sum_{k=1}^{A} n_k$ le nombre d'accès en hit dans le cache et $n_\emptyset$ toujours le nombre d'échecs ou éventuellement le couple $(n, n_\emptyset)$ avec $n = \sum_{k=1}^{A} n_k + n_\emptyset$ le nombre total d'accès mémoire sur le bloc considéré. On appellera $\mathcal{N}_i^*$ l'espace vectoriel réduit ainsi défini. On note que si on opère le premier choix pour définir $\mathcal{N}_i^*$, alors, pour un cache d'associativité $A = 1$ (à correspondance directe), on a naturellement $\mathcal{N}_i = \mathcal{N}_i^*$.

6.2.4. Espace d'état global. On peut définir un vecteur d'état global et un espace d'état global pour la référence mémoire *i* considérée en effectuant le produit cartésien des deux espaces précédemment définis. On notera $\mathcal{E}_i = \mathcal{N}_i \otimes \mathcal{P}_i$ l'espace résultant (et respectivement $\mathcal{E}_i^* = \mathcal{N}_i^* \otimes \mathcal{P}_i$ l'espace d'état réduit correspondant). Comme le passage de $\mathcal{N}_i$ à $\mathcal{N}_i^*$ est trivial, on se contentera le plus souvent pour la suite de l'exposé de montrer les relations sur $\mathcal{N}_i$ et $\mathcal{E}_i$, mais il faut garder en mémoire le fait qu'utiliser des espaces non réduits n'est pas forcement utile en pratique.

Les hypothèses requises pour qu'un tel modèle puisse fonctionner sont que l'associativité du cache soit une constante, de même que la taille de bloc, que le mécanisme de remplacement dans le cache est LRU et que les données ou les instructions dans chaque bloc soient stoquées de façon séquentielle. Ces hypothèses sont généralement vérifiées sur la plupart des processeurs actuels[1]. A noter que la constance de l'associativité et de la taille de bloc ne sont pas nécessaires strictement parlant, si on accepte une modification des opérateurs d'accès élémentaires et de concurrence. De

[1]Seul le Pentium 4 d'Intel avec son cache de trace est une exception notable à ce cadre, mais il faudra plusieurs années avant de savoir si cela se généralisera.

même, la politique de remplacement peut être aisemment changée moyennant une adaptation de l'opérateur d'accès que l'on introduira à la section 6.4.3. La prise en compte de caches non séquentiels comme les caches de trace n'a pas été étudiée par contre. Vraissemblablement, cela nécessiterait une adaptation plus importante de la théorie.

6.2.5. Espace d'état du système. Pour la mémoire totale et pas simplement une référence isolée, l'état global peut donc être défini comme le produit cartésien des états des références mémoire. Si $\mathcal{J}$ est l'ensemble des références utilisées par le programme et si l'indice i désigne une référence donnée, alors on a :

$$\Upsilon = \bigotimes_{i \in \mathcal{J}} \mathcal{E}_i$$

l'espace d'état de la mémoire. Il faut noter que pour une architecture de type Harvard, il est avantageux de définir une partition entre l'espace Υ_D des données et l'espace Υ_I des instructions, si le code auto-modifiable est interdit, ce qui est en principe le cas dans les applications que l'on cherche à traiter. On a alors deux espaces indépendants, sans interactions ce qui simplifie un peu les calculs.

$$\begin{cases} \Upsilon_D = \bigotimes_{i \in J_D} \mathcal{E}_i \ \textit{avec}\ J_D\ \textit{ensemble des références de données} \\ \Upsilon_I = \bigotimes_{i \in J_I} \mathcal{E}_i \ \textit{avec}\ J_I\ \textit{ensemble des références d'instructions} \end{cases}$$

On peut généraliser cela à tout type de partition de cache. Pour la suite, on pourra même en cas d'une existence de partition de cache se concentrer sur l'une d'elles seulement, puisque par construction, les partitions de cache n'ont pas d'interactions entre elles.

On notera υ_p le produit cartésien des espaces d'état de probabilité, à chaque fois que l'on en aura besoin.

$$\upsilon_p = \bigotimes_{i \in \mathcal{J}} \mathcal{P}_i$$

6.3. Modèles à base de chaînes de Markov

Les chaînes de Markov sont parmis les plus simples exemples de processus markoviens[**23**][**17**]. On considère $s_t,\ t \in \mathbb{N}$ une coordonnée discrète d'évolution du système (par exemple cela peut être un temps discrétisé sous forme d'une succession de dates) et un ensemble discret d'état possible pour le système considéré que l'on note

$$\{y(n),\ n \in \{1, 2, \ldots, M\}\}$$

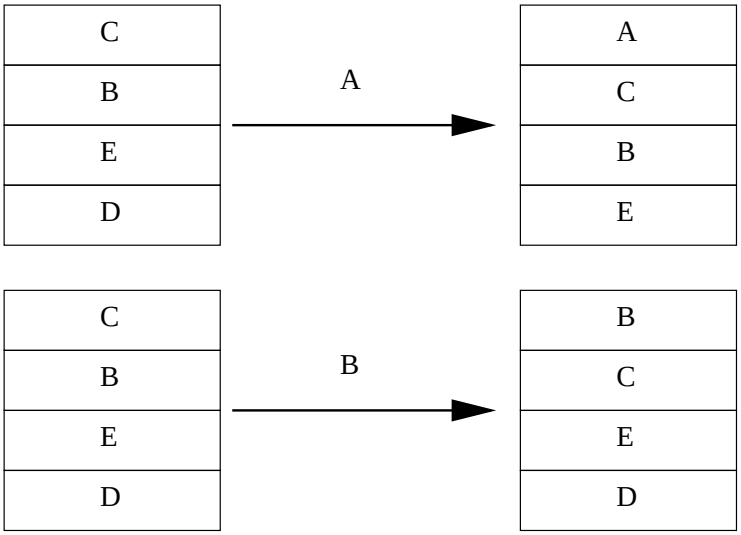
Pour une chaîne de Markov, on peut écrire la probabilité pour le système d'être dans l'état $y(n)$ en s_{t+1} (noté $p(n, t+1)$) en fonction des probabilités des états du système en s_t (noté $p(m, t)$), sous la forme suivante :

$$\forall t,\ \ p(n, t+1) = \sum_{m=1}^{M} p(m, t) P_{1|1}(m, t \,|\, n, t+1)$$

où le facteur numérique $P_{1|1}(m, t \,|\, n, t+1)$ peut être interprété comme une probabilité conditionnelle. On peut facilement synthétiser cette relation en utilisant une matrice $Q(t) \in \mathcal{M}(M)$ comme opérateur d'évolution entre le vecteur d'état de probabilité en s_t, soit $|p(t)\rangle \in \mathcal{P}_i$, et le vecteur d'état de probabilité en s_{t+1}, soit $|p(t+1)\rangle$, et on peut écrire :

$$|p(t+1)\rangle = Q(t)\ |p(t)\rangle$$

FIG. 6.4.1. *Illustration de deux accès sur un ensemble de cache d'associativité 4*



On constate donc qu'une chaîne de Markov traduit une évolution linéaire pour les probabilités des différents états possibles entre deux coordonnées successives. On va maintenant voir comment utiliser ce genre de modélisation dans le problème qui nous intéresse.

6.4. Opérateur d'accès en mémoire

L'accès à une référence mémoire sera modélisé par un opérateur agissant sur l'espace Υ (éventuellement Υ_D ou Υ_I si cela s'applique). Cependant on va d'abord définir les opérateurs de base correspondant qui s'appliquent au niveau de l'espace d'état de la référence, $\mathcal{E}_i$. Le schéma général d'un accès mémoire quant à ses effets sur un ensemble du cache est illustré en figure 6.4.1 . Au niveau de l'action possible sur l'espace $\mathcal{E}_i$, on peut discerner deux cas bien distincts. Le premier correspond à la référence d'accès elle-même (A dans la première figure et B dans la seconde), l'opérateur associé sera appelé opérateur d'accès élémentaire et a pour effet de placer la référence considérée en tête de son ensemble au niveau des historiques ainsi

que d'ajouter un accès avec une mise à jour des nombres d'accès en historique et le nombre d'échecs ; le second correspond à toutes les références qui sont avant l'accès considéré dans l'historique LRU (toutes les référence de l'ensemble du cache dans la première figure, et la référence C dans la seconde) et pour laquelle l'opérateur associé, que l'on appellera opérateur de concurrence, a pour effet de faire reculer d'un rang les probabilités d'accès dans l'historique. On prendra soin de noter qu'il est inutile, à ce niveau, de distinguer une instruction de chargement pour utilisation (un chargement dans un registre pour une architecture *load/store* si on s'occupe des données) d'une instruction de préchargement de bloc de cache telle qu'on en trouve sur certains processeurs RISC. La différence se fera seulement au niveau des répercussions sur les temps d'exécution lors du calcul de l'évaluation du WCET.

6.4.1. Opérateur d'accès élémentaire. Supposons que l'on rencontre une nouvelle instruction qui nécessite d'accéder à une référence mémoire i. Pour la partie flot de contrôle, cela sera par exemple l'arrivée sur un nouveau bloc d'instructions due à l'incrémentation du compteur de programme, ou à la suite d'une instruction de branchement. Pour le flot de donnée, cela sera, sur une architecture *load/store*, une instruction de chargement ou de sauvegarde en mémoire. L'effet, au niveau du cache est le suivant : si le bloc est dans le cache, il est mis en tête de son ensemble dans l'historique ; s'il n'est pas dans le cache (échec donc), le bloc est transféré dans le cache depuis la mémoire principale et mis en tête de son ensemble. Au niveau du vecteur d'état de nombre d'accès, il suffit d'ajouter le vecteur de probabilité (augmentation globale de une unité du nombre d'accès et le vecteur probabilité fournit les répartitions effectives avant accès) ; quant au vecteur d'état de probabilité, on a donc passage à 1 de la probabilité de prochain accès en

historique 1, d'où le système d'équation suivant :

$$\begin{cases} |n(t+1)\rangle = |n(t)\rangle + |p(t)\rangle \\ |p(t+1)\rangle = \top |p(t)\rangle \end{cases}, |n\rangle \in \mathcal{N}_i \; et \; |p\rangle \in \mathcal{P}_i$$

et où on a l'opérateur $\top = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$

Au niveau du vecteur d'état, on peut écrire cela de manière plus synthétique, mathématiquement, en introduisant l'opérateur linéaire $a^\dagger$:

$$|s_t\rangle, |s_{t+1}\rangle \in \mathcal{E}_i, \; |s_{t+1}\rangle = a^\dagger |s_t\rangle$$

et l'opérateur d'accès élémentaire $a^\dagger$ est défini comme suit :

$$a^\dagger = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ 0 & \top \end{pmatrix}$$

et lorsqu'un besoin de représentation graphique se fait sentir, nous utiliserons la suivante pour cette opération :

On note qu'il suffit d'effectuer le produit à gauche par $a^\dagger$ pour obtenir la mise à jour du vecteur d'état pour la référence considérée après l'accès.

6.4.2. Opérateur de concurrence d'accès. On considère cette fois lors de l'accès, une référence j qui appartient au même ensemble du point de vu du cache que la référence i à laquelle on accède mais qui est dans le cache et mieux placée dans l'historique. Cela signifie que le présent accès va décaler son historique d'un rang puisque c'est la référence à laquelle on accède qui va prendre la tête de l'historique. On en déduit que la probabilité d'accès

de cette référence concurrente en tête d'historique passe à 0, que sa probabilité d'accès en deuxième place dans l'historique devient la probabilité précédente d'accès en tête d'historique, le décalage se reportant de place en place sur tout l'historique, la probabilité d'échec étant, quant à elle, accumulative. On traduit cela au niveau du vecteur d'état de probabilité sous la forme suivante :

$$|p(t+1)\rangle = \lambda\,|p(t)\rangle \;\; avec \;\; \lambda = \begin{pmatrix} 0 & 0 & & \cdots & 0 \\ 1 & 0 & & & \vdots \\ 0 & 1 & \ddots & & \\ \vdots & & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 1 \end{pmatrix}$$

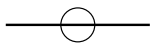
Évidemment, comme il n'y a pas d'accès effectif sur cette référence, les nombres d'accès restent constants. On peut donc écrire également l'opération d'évolution du vecteur d'état global comme ceci :

$$|s_t\rangle\,, |s_{t+1}\rangle \in \mathcal{E}_j,\;\; |s_{t+1}\rangle = b\,|s_t\rangle$$

et l'opérateur de concurrence d'accès b est défini comme suit :

$$b = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \lambda \end{pmatrix}$$

Lorsqu'on aura besoin d'une représentation graphique pour cet opérateur, nous utiliserons celle-ci :



Comme précédemment, on note qu'il suffit d'effectuer le produit à gauche par l'opérateur pour obtenir la mise à jour du vecteur d'état concerné.

6.4.3. L'opérateur d'accès complet et son approximation. En faisant la synthèse de ce que l'on vient de voir, on aboutit à l'expression de l'opérateur d'accès sur l'espace Υ (ou un sous espace pour une architecture Havard ou une partition de cache) sous la forme suivante dans laquelle les indices désignent les références aux quelles s'appliquent les opérateurs de base :

$$\mathcal{A}_i = a_i^\dagger \underset{j\in\kappa_i^n}{\otimes} b_j \underset{k\notin\kappa_i^n, k\neq i}{\otimes} \mathbb{1}_k$$

où il faut comprendre la notation de sorte que $a_i^\dagger$ s'applique à l'espace $\mathcal{E}_i$ de Υ, où les b_j s'appliquent aux espaces $\mathcal{E}_j$, et où κ_i^n est l'ensemble des références du même ensemble que i qui sont avant celle-ci dans l'historique des références de l'ensemble. Une propriété évidente est que l'on a $\kappa_i^n \subset K_i^n$ avec

$$K_i^n = \{ j \in \mathcal{J} / \; j \equiv i\,[n] \wedge j \neq i \}$$

et

$$n = \frac{taille\,de\,cache\,en\,blocs}{A}$$

car K_i^n est l'ensemble de tous les blocs référence mémoire qui *peuvent* entrer en concurence avec la référence i. En général il n'est, en effet, pas possible de trouver l'ensemble κ_i^n car il peut dépendre de paramètres soit extérieurs soit qui ne sont connus que dynamiquement, au cours de l'exécution du code. C'est pourquoi, on est amené à utiliser une approximation pessimiste de κ_i^n qui soit la plus proche possible de la réalité mais qui ne dépend pas d'aléas pour le code analysé. Si on appelle Γ_i^n cet ensemble, on aura la relation $\kappa_i^n \subset \Gamma_i^n \subset K_i^n$, l'inclusion étant à comprendre au sens large dans les deux cas. En définitif, on va donc appliquer un opérateur *effectif* sur

l'espace d'état qui sera donné par l'expression suivante :

$$\mathcal{A}_i^* = a_i^\dagger \underset{j\in\Gamma_i^n}{\otimes} b_j \underset{k\notin\Gamma_i^n, k\neq i}{\otimes} 1_k$$

Le calcul de l'ensemble Γ_i^n doit découler d'une expression pessimiste de l'histoire des accès au vu du but que l'on cherche à atteindre. C'est une technique qui a déjà été abordée dans la littérature[**11**][**25**] et qui est généralement appelée technique *must analysis* comme nous l'avons vu à la section 4.2.3. En reprenant les même notation qu'alors, l'ensemble Γ_i^n que l'on cherche est alors l'union des ensembles de la collection que l'on détermine par *must analysis* en chaque point de l'exécution de la tâche considérée. Donc simplement, en notation mathématique :

$$\Gamma_i^n = \bigcup_{h=1}^{A} c_h(i[n])$$

6.5. Traitement des aléas du flot de contrôle

6.5.1. Aléas de flot de contrôle. Les aléas de flot de contrôle rendent compte du non déterminisme de la suite d'instructions effectivement exécutées au cours du traitement d'une tâche donnée. Un exemple simple est celui des alternatives *if (C)... then (T1)... else (T2)...* (si... alors... sinon...) qui traduisent un traitement conditionnel. Le traitement *T1* est exécuté si la condition *(C)* est réalisée. Dans le cas contraire, c'est le traitement *T2* qui est exécuté. On est en présence d'un cas réel d'aléas de flot de contrôle s'il est impossible de déterminer de manière statique laquelle des deux alternatives sera exécutée (la condition *(C)* dépend d'éléments externes à la tâche considérée).

Dans un tel cas, on est en présence d'une indétermination du flot de contrôle et il faut trouver un moyen d'en rendre compte au niveau du calcul d'un WCET. Il faut noter que la difficulté ne vient pas de l'exécution du corps principal de la tâche suivi de *T1*, ni de l'exécution du corps principal de la tâche suivi de *T2*, mais bien du fait qu'à la fin du traitement de l'alternative, on ne sait pas laquelle des deux alternatives a été exécutée. C'est donc un problème de recombinaison de chemins d'exécutions multiples, concurrents.

6.5.2. Utilisation d'une opération linéaire.

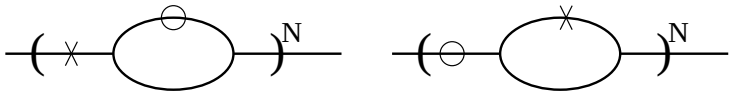
6.5.2.1. *Principe général.* C'est la méthode la plus simple qui consiste en fait à moyenner entre toutes les branches du flot de contrôle quant à ses répercussions sur les probabilités et le nombre d'échecs. L'avantage de cette méthode est qu'elle est linéaire et donc simple et rapide à mettre en oeuvre. Par exemple si p_1 est un sous-chemin d'exécution possible et p_2 un autre, que l'on a O_1 et O_2 respectivement pour les opérateurs issus de chaque chemin et $|s_1\rangle$ et $|s_2\rangle$ et si on note O_{12} et $|s_{12}\rangle$ l'opérateur et le vecteur d'état résultant lors de la fusion des chemins (ou de la fin de la tâche étudiée), on peut alors écrire :

$$\begin{cases} O_{12} = \frac{1}{2}(O_1 + O_2) \\ |s_{12}\rangle = \frac{1}{2}(|s_1\rangle + |s_2\rangle) \end{cases}$$

L'inconvénient principal de cette méthode est que l'on s'éloigne, *a priori*, du pire cas possible, et il faut, en conséquence, calculer un écart maximum possible ou un écart type en parallèle à cette méthode pour avoir un résultat valide. C'est un problème que l'on va exposer sur un exemple simple que l'on résoudra de bout en bout.

6.5.2.2. *Calcul d'un écart maximum sur le nombre moyen d'échecs de cache.* Pour un vecteur d'état donné on peut toujours avoir un échec de

FIG. 6.5.1. *boucle à N itérations avec un accès mémoire dans le corps de la boucle et un accès concurrent conditionnel*



cache, sauf dans le cas où on a la composante de probabilité d'échec de cache telle que $p_\emptyset = 0$ qui est un cas déterministe. Sauf dans ce dernier cas, la distance de $p_\emptyset$ à 1 est l'erreur commise sur la mesure de la variable «nombre d'échecs de cache». Si pour chaque étape, on garde la trace de cette distance, on peut définir un écart maximum qui serait simplement la somme de toutes ces valeurs. Il suffit à chaque accès de rajouter cette distance pour les $p_\emptyset$ non nuls à la somme déjà en cours.

On peut, pour mieux expliciter son fonctionnement, montrer l'utilisation de cette méthode sur un cas simple. On va se contenter d'un cache non associatif ($A = 1$), et tester la méthode sur le programme qui se traduit par le couple de diagrammes de la figure 6.5.1.

Il est immédiat de constater qu'à chaque étape i de l'itération de la boucle, on a pour le premier diagramme l'opérateur suivant :

$$A = [\frac{1}{2}(\mathbb{1} + b)\, a^\dagger]^i = \frac{1}{2} \begin{pmatrix} 2 & 0 & i+1 & i-1 \\ 0 & 2 & i-1 & i+1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

et de la même façon, pour le second diagramme :

$$A' = [\frac{1}{2}(\mathbb{1} + a^\dagger) b]^i = \frac{1}{2} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & i & i \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

En faisant l'hypothèse d'un démarrage à froid pour l'état initial des caches (c'est à dire que les vecteurs d'état valent tous $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$), on a les vecteurs d'état suivants à chaque itération :

– Vecteur d'état avant le premier accès de la boucle pour le diagramme 1 :

$$|s_{1\times}\rangle = \frac{1}{2}\begin{pmatrix} i-1 \\ i+1 \\ 1 \\ 1 \end{pmatrix}$$

c'est un accès, on aura $\frac{1}{2}$ d'écart maximal à chaque fois, sauf au premier, d'où

$$\Delta_i^1 = \frac{i-1}{2}$$

– Vecteur d'état après le premier accès de la boucle pour le diagramme 2 :

$$|s_{2l}\rangle = \frac{1}{2}\begin{pmatrix} 0 \\ i \\ 0 \\ 2 \end{pmatrix}$$

– Vecteur d'état après l'accès de la condition pour le diagramme 2 :

$$|s_{2i}\rangle = \frac{1}{2}\begin{pmatrix} 0 \\ i+2 \\ 2 \\ 0 \end{pmatrix}$$

ces deux états superposés en sortie de boucle nous donnent

$$\Delta_i^2 = \frac{i}{2}$$

de la même manière que précédemment.

Pour les contributions à l'écart maximum, il faut le comptabiliser à chaque accès mémoire, et chaque fusion de chemins dans le flot d'instructions. Cela conduit aux valeurs suivantes pour cet exemple, à la fin du programme ($i = N$) :

$$\begin{cases} m_1 &= \frac{N+1}{2} \pm \frac{N-1}{2} \\ m_2 &= \frac{N}{2} \pm \frac{N}{2} \end{cases}$$

ce qui correspond bien aux résultats rigoureux, sur cet exemple particulier.

6.5.3. Utilisation d'une méthode non linéaire avec obtention directe d'un majorant du WCET.

6.5.3.1. *Principe général.* Plutôt que d'utiliser une variable annexe pour obtenir une évaluation du cas d'exécution au pire, on peut imaginer directement rendre compte du cas au pire dans le vecteur d'état ou dans l'opérateur d'évolution après fusion des chemins. Pour le vecteur d'état, il s'agit simplement de choisir comme vecteur d'état après la reconvergence de deux chemins d'exécution celui issu d'un des chemin qui maximise le nombre d'échecs de cache ($n_\emptyset$ dans le vecteur) où à défaut celui qui maximise les probabilités d'échecs de prochain accès (la valeur $p_\emptyset$ du vecteur). En ce qui concerne l'opérateur d'évolution, il s'agit d'en créer en général un nouveau dont chaque colonne est choisie entre les colonnes correspondantes des opérateurs avant recombinaison selon le même principe que le cas du vecteur d'état. Cette opération n'est, bien-sûr, pas linéaire mais elle est plus systématique dans sa mise en œuvre, et plus proche de ce qui se fait dans

les autres travaux d'évaluations de WCET par des méthodes d'interprétation abstraite. Le seul léger inconvénient de la méthode étant la perte de la linéarité, c'est plutôt sur cette méthode que l'on s'appuiera dans la suite pour la reconvergence de flots de contrôle. A noter tout de même que, sauf si l'on possède des hypothèses fortes sur le comportement de la suite de l'exécution d'une tâche, cette recombinaison pessimiste doit être faite de la même manière pour tous les espaces élémentaires $\mathcal{E}_i$, même si cela conduit à effectuer des combinaisons qui ne sont pas compatibles vis à vis des aléas du flot de contrôle car on recherche le cas au pire et que chacune des deux voie peut y mener, *a priori*. On va maintenant voir comment cela se passe sur un exemple simple, en utilisant cette méthode.

6.5.3.2. *Mise en oeuvre sur un exemple simple.* Comme précédemment, on va traiter avec cette méthode l'exemple de la figure 6.5.1. On aboutit simplement à $A_{nl} = (ba^\dagger)^N$ sur le premier diagramme, qui maximise le nombre d'échecs. En postulant un cache invalidé au départ $|s_i\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ on en déduit aisément le nombre d'échecs de cache

$$m_1 = \langle m | \, (ba^\dagger)^N \, |s_i\rangle = N$$

de la même manière que précédemment avec

$$\langle m | = \begin{pmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} = \left(\delta_{A+1,\,j}\right)$$

le projecteur d'échecs de cache.

Pour le second diagramme, on va associer l'opérateur $A' = (a^\dagger b)^N$ qui maximise la probabilité d'échecs. Avec la même hypothèse de cache invalide au départ, on en déduit le nombre d'échecs associés

$$m_2 = \langle m | \, (a^\dagger b)^N \, | s_{nl} \rangle = N$$

ce qui donne encore une fois le même résultat qu'au paragraphe 6.5.2.2 sur cet exemple simple. Dans les deux cas, on aboutit au résultat au pire exact.

6.6. Traitement des aléas du flot de données

6.6.1. Aléas de flots de données. Il existe des cas où il est totalement impossible de connaître *a priori* l'endroit exact où un accès de donnée va avoir lieu. Un cas typique est celui des buffers ou des tables de look-up. Dans ces cas là, tout ce que l'on peut connaître est une zone où l'accès peut avoir lieu et la taille de l'accès dans cette zone. La position exacte de l'accès dépend, elle, de paramètres non contrôlables, lorsque ce cas s'applique, comme l'histoire de l'exécution ou de données extérieures à la tâche.

6.6.2. Traitement de ces aléas. Dans le cadre actuel de la théorie, peu de travail a été fait dans cette direction, la priorité se situant à d'autre niveau. S'agissant de déterminer un cas au pire, on se contente pour l'instant de faire comme si l'accès avait lieu sur la totalité de l'espace mémoire concerné, au niveau des concurrences d'accès (mais pas des accès eux-même). Ainsi, l'opérateur utilisé dans le cadre actuel, lors de l'accès sur une zone mémoire $\mathbb{A}$ est :

$$M_i^* = \bigotimes_{j \in \Gamma^n_{\mathbb{A}}} b_j \bigotimes_{k \notin \Gamma^n_{\mathbb{A}}} 1_k$$

où $\Gamma^n_{\mathbb{A}}$ est une notation condensée pour $\bigcup_{i\in\mathbb{A}} \Gamma^n_i$. On doit cependant noter que c'est une détermination très pessimiste, que l'on peut probablement améliorer.

Avec cette opération, on peut modéliser l'essemble des comportements de la mémoire vis à vis des caches quelque soit ce qui peut arriver lors de l'exécution d'un programme.

6.7. Utilisation des opérateurs

6.7.1. Définition : Chemin d'exécution complet. On appelle chemin d'exécution complet toute succession d'instructions dont le flot de contrôle associé est tel que si l'exécution passe par la première instruction, elle passera obligatoirement par la dernière instruction de ce chemin.

6.7.2. Opérateurs d'évolution général. On a vu, lors de ce développement théorique comment passer de proche en proche d'un vecteur d'état à un autre au cours de l'exécution symbolique de la tâche, en tenant compte de tout ce qui pouvait se produire au niveau de l'incidence de l'exécution sur l'état de la mémoire tel qu'on l'a conçu. Pour une branche d'exécution donnée, on peut associer un vecteur d'état avant exécution de la branche, soit $|s_0\rangle$. A chaque étape de l'exécution, on peut donc avoir un des cas suivant :

- accès mémoire déterminé : on va avoir un opérateur $A^*_{t+1,i}$ tel que $|s_{t+1}\rangle = A^*_{t+1,i}|s_t\rangle$;
- *pour les accès de données uniquement* ; accès avec aléas de flot de données : on peut trouver un opérateur $M^*_{t+1,t}$, comme on l'a vu au

paragraphe 6.6.2, tel que l'on ait $|s_{t+1}\rangle = M^*_{t+1,t}|s_t\rangle$. Avec la propriété précédente, une récurrence simple montre que pour toute succession purement séquentielle d'instructions, on peut trouver un opérateur $O_{k0} = O_{k,k-1}\cdots O_{2,1}O_{1,0}$ tel que $|s_k\rangle = O_{k0}|s_0\rangle$;

– recombinaison de deux chemins d'exécution : soit $\left|s_f\right\rangle$ le vecteur d'état avant le traitement conditionnel. Si les deux chemins sont des chemins d'exécution purement séquentiels, alors l'application du résultat précédent permet d'écrire que la recombinaison peut s'écrire en terme de deux vecteurs d'états $|s_e\rangle = O_{ef}\left|s_f\right\rangle$ et $|s'_e\rangle = O'_{ef}\left|s_f\right\rangle$ et donc qu'il existe un opérateur O^*_{ef} comme on l'a vu à la section 6.5 tel que le vecteur d'état après recombinaison peut s'écrire $O^*_{ef}\left|s_f\right\rangle$. Si les branches des alternatives d'exécution ne sont pas purement séquentielle, une récurrence simple sur ce résultat (étant bien entendu que l'on finit toujours par tomber de proche en proche sur un chemin d'exécution purement séquentiel qui est au pire réduit à une instruction unique) permet de généraliser le théorème à tout chemin d'exécution complet.

D'où le théorème suivant :

THEORÈME 6.7.1. *Pour tout chemin d'exécution complet $\mathcal{P}$, on peut trouver un opérateur $O_{\mathcal{P}} \in \mathcal{L}(\Upsilon)$ tel que pour tout état initial $\left|s_i^{\mathcal{P}}\right\rangle \in \Upsilon$ avant exécution dudit chemin, l'état final $\left|s_f^{\mathcal{P}}\right\rangle \in \Upsilon$ puisse s'écrire :*

$$\left|s_f^{\mathcal{P}}\right\rangle = O_{\mathcal{P}}\left|s_i^{\mathcal{P}}\right\rangle$$

A noter que l'on peut exprimer un résultat similaire en se restreignant à l'espace des probabilités υ_p

THEORÈME 6.7.2. *Pour tout chemin d'exécution complet $\mathcal{P}$, on peut trouver un opérateur $Q_{\mathcal{P}} \in \mathcal{L}(\upsilon_p)$ tel que pour tout état initial $\left|p_i^{\mathcal{P}}\right\rangle \in \upsilon_p$*

avant exécution dudit chemin, l'état final $\left|p_f^{\mathcal{P}}\right\rangle \in \upsilon_p$ *puisse s'écrire :*

$$\left|p_f^{\mathcal{P}}\right\rangle = Q_{\mathcal{P}} \left|p_i^{\mathcal{P}}\right\rangle$$

Ce qui met très bien en évidence le cadre des chaînes de Markov pour la théorie.

6.7.3. Propriétés de combinaison. Les théorèmes précédents permettent donc de déterminer des opérateurs généraux qui sont associés à tout chemin d'exécution complet. Il est donc possible de combiner ces opérateurs une fois calculés. Par exemple, pour deux chemins d'exécution complets $\mathcal{P}_1$ et $\mathcal{P}_2$ et leurs opérateurs associés respectifs O_1 et O_2, on obtient simplement un opérateur associé pour l'exécution successive de $\mathcal{P}_1$ et $\mathcal{P}_2$ avec le produit de O_2 et O_1 :

$$O_{12} = O_2 O_1$$

On doit tout de même noter que le produit pur ne permet pas d'obtenir l'opérateur le moins pessimiste si l'on n'a pas pris en compte les restrictions d'historiques dues au fait que $\mathcal{P}_1$ s'est exécuté avant $\mathcal{P}_2$, et les informations que cela amène au niveau de l'évaluation des opérateurs d'accès et des recombinaisons de flots de contrôle. Pour des segments assez importants et un nombre de combinaisons d'opérateurs négligeable devant le nombre d'échecs de cache évalué, cela n'a, cependant, qu'un impact très limité.

De même, pour l'exécution concurrente de $\mathcal{P}_1$ et $\mathcal{P}_2$ (exécution soit de l'un soit de l'autre comme aléas du flot de contrôle), l'opérateur associé sera l'opérateur recombiné de O_1 et O_2 au sens où on l'a introduit au paragraphe 6.5.

Bien entendu, les résultats sont exactement les mêmes en ce qui concerne l'espace de probabilité.

6.7.4. Calcul de l'état stationnaire. On se place au niveau de l'espace de probabilité υ_p et on considère l'opérateur de probabilité associé à une tâche complète T soit Q_T. Une tâche temps réel est, en général, amenée à se répéter à intervalles réguliers, sans limitation dans le temps. Pour un tel système, on est donc dans le cas classique d'un système de chaîne de Markov indépendante du temps. On sait qu'alors, le système trouvera un état d'équilibre et que le vecteur d'état de probabilité associé est l'unique vecteur propre de Q_T associé à la valeur propre 1 (il existe et est unique[**23**]). En termes mathématiques :

$$\exists! \, |p_e\rangle \in \upsilon_p \,/\, |p_e\rangle = Q_T \, |p_e\rangle$$

Le vecteur $|p_e\rangle$ est le vecteur d'équilibre pour la tâche qui boucle indéfiniment.

On en déduit aisément le nombre d'échecs de cache à l'équilibre pour la tâche considérée :

$$m_T = \langle m| \, O_T(|0\rangle \otimes |p_e\rangle)$$

où O_T est, comme le suggère la notation, l'opérateur global (dans $\mathcal{L}(\Upsilon)$) associé à la tâche T.

Comme on le voit, cette méthode permet de se passer de l'hypothèse de caches invalides en début de tâche dans un tel système ce qui permet d'améliorer les résultats de cette méthode par rapport aux autres méthodes d'interprétation abstraite.

6.8. Synthèse sur le cas séquentiel

Le cadre théorique dégagé ici pour le cas séquentiel se distingue de plusieurs manières des approches ordinaires de calculs de WCET par interprétation abstraite :

- l'espace abstrait auquel on s'intéresse n'est pas lié aux états du cache mais aux états de la mémoire vis à vis du cache ;
- le modèle repose sur un espace de probabilité, très différent des collections d'ensembles des théories généralement rencontrées ;
- si l'on garde globalement un modèle d'interprétation abstraite, en revanche on se base sur un système de type markovien pour mettre à jour les états.

Cela conduit, indubitablement, à des calculs plus lourds que les autres modèles d'interprétation abstraite. On obtient cependant comme compensation de pouvoir se passer de toutes hypothèses sur les états initiaux considérés. Le fait d'utiliser un espace de probabilités permet, comme on va le voir à présent, d'élargir le champ de la théorie au cadre multitâche.

CHAPITRE 7

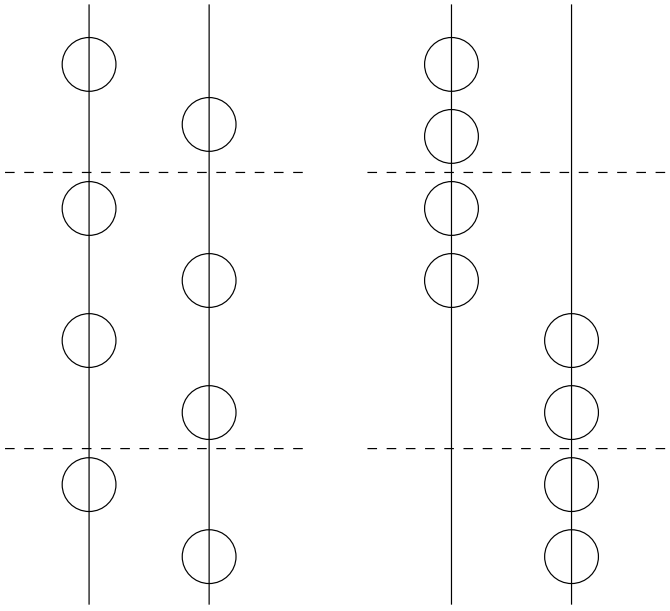
Passage au cadre multitâche

7.1. Problématique

On a vu au paragraphe 5.2.2 la difficulté qu'il pouvait y avoir à généraliser une méthode d'évaluation de WCET au cadre multitâches. Nous allons plus particulièrement quantifier la difficulté associée en calculant l'entropie du système considéré.

Considérons par exemple la succession des accès en *hit* de deux références distinctes au cours de l'exécution du programme. En figure 7.1.1 on a représenté le cas de quatre accès en hit dans deux situations extrèmes et opposées, à savoir le cas d'une corrélation forte et le cas d'une corrélation faible . En pointillés on a représenté deux endroits possibles de préemption

FIG. 7.1.1. *Deux schémas distincts d'accès à deux références de corrélation opposée*



qui sont des endroits au pire. On constate que dans le premier cas (forte corrélation), on va obtenir $2 \times 2 = 4$ échecs de cache supplémentaires (2 échecs pour chaque préemption). Dans le second cas (faible corrélation), on aura cette fois $1 \times 2 + 1 = 3$ échecs de cache supplémentaire (1 échec par préemption plus un, dû à un effet d'interface). Tangentiellement parlant, il est assez simple de constater que l'on aura, si n_{max} est le nombre maximum de préemption et N le nombre de références mises en jeu, un nombre d'échecs supplémentaires $m_{sup} = N \times n_{max}$ pour le cas totalement corrélé et $m_{sup} = n_{max} + N - 1$ au pire pour le cas complètement décorrélé. Le problème vient, bien-sûr, du calcul de la corrélation puisque c'est de ce calcul que dépend l'évaluation d'un nombre maximum d'échecs supplémentaires.

Supposons que l'on ait une corrélation de m pour n références, la détermination de cette corrélation nécessite de considérer les combinaisons de m parmis n soit C_n^m possibilités. Comme m est inconnu, il faut le faire varier entre 1 et n. On en déduit que le nombre de combinaisons totales à tester est

$$\sum_{m=1}^{n} C_n^m = 2^n$$

et donc pour N références totales, on a une complexion totale $W = 2^N$, on en déduit immédiatement l'entropie du système en choisissant $k = \frac{1}{\ln 2}$ comme constante de normalisation (qui est un choix habituel en théorie de l'information)

$$S = N$$

Ainsi le coût d'un calcul exact de corrélation pour un tel système est exponentiel en fonction du nombre de référence (sauf s'il existe une régularité forte des accès, ce qui est rarement le cas sur les accès de données). Il est donc hors de question dans le cas général de chercher à obtenir un résultat exact pour le coût de préemption.

Ceci étant entendu, nous allons chercher à exploiter à présent le fait que le modèle que l'on a construit possède une base probabiliste pour estimer les majorants de temps d'exécution dans le domaine étendu de l'exécution multitâche.

7.2. Idée générale

Le fil conducteur est de tirer avantage de l'aspect probabiliste des chaînes de Markov. Dans le cadre monotâche, tout les calculs que nous avons réalisés sont déterministes strictement, et on va donc relacher cette contrainte via une modification du propagateur par défaut. De façon plus explicite, ce que l'on cherche à faire est de simplement modifier les opérateurs $a^\dagger$ et b de façon à ce qu'ils rendent compte d'une possibilité de préemption. Ainsi, en remplaçant $a^\dagger$ par $a^\dagger\Pi$ et b par Πb, directement sans changer la théorie, on cherche à obtenir une estimation réaliste du nombre d'échecs de cache pour la tâche lorsqu'il y a préemptions par d'autres. Ainsi pour une référence mémoire auquel on accède régulièrement dans le cache pour le cas sans préemption, on souhaite que l'application des nouveaux opérateurs fasse apparaître un nombre d'échecs supplémentaires qui soit de l'ordre du nombre de préemptions possibles qui puissent entraîner l'évincement du cache de la référence en question. Pour obtenir un majorant à partir de cette estimation, il est alors nécessaire d'évaluer un écart type qui dépendra en général de la répartition effective des accès et de l'évolution *a priori* de la probabilité de préemption. La majoration sera faite au final en ajoutant à l'estimation des échecs un certain nombre de fois l'écart type trouvé en fonction de la sûreté que l'on veut avoir pour le résultat.

Pour la suite de l'exposé, voici les notations qui seront utilisées :

n_{max}**:** nombre maximum de préemptions attendues lors de l'exécution de la tâche considérée ;

N**:** nombre d'instructions minimales exécutées par la tâche (ou minimum pris par une forme quelconque de coordonnée d'avancement dans l'exécution d'une tâche, à la fin de l'exécution de la tâche) ;

$\mathcal{A}$**:** nombre d'accès à la référence étudié lors de l'exécution de la tâche.

p**:** probabilité de préemption. On supposera cette probabilité constante au cours de l'exécution de la tâche, en première approximation.

On doit considérer que n_{max}, N, et $\mathcal{A}$ sont des données connues du problème, sans lesquels il est impossible de trouver un résultat raisonnable. Par contre p pourra être calculé dans le modèle comme on le verra.

Ce que l'on cherchera à trouver ici sont des opérateurs de préemption Π. En particulier, lors d'une progression de Δs dans le programme, on peut poser que l'opérateur Π s'écrive $\pi^{\Delta s}$, pour le cas d'une probabilité de préemption uniforme. Dans le cas où cette probabilité n'est pas uniforme, on pourra tout de même écrire l'opérateur de préemption comme un produit d'opérateurs élémentaires $\Pi = \prod_i \pi_i$ où le produit s'entend comme un produit à gauche. Le résultat d'estimation ainsi que l'écart type s'en ressentiraient en conséquence. Mais on peut aisément contourner ce genre de problème.

7.3. Calcul de l'opérateur de préemption

L'hypothèse que l'on fait est que l'on choisi s, l'avancement dans le programme de façon à ce que la probabilité de préemption soit une constante vis à vis de s. Comme on le verra c'est une hypothèse qu'il serra aisé de dépasser par la suite. A chaque étape de l'exécution de la tâche l'opérateur d'avancement dans le programme, ou propagateur, peut s'écrire de la façon

suivante :

$$\pi = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph \end{pmatrix} \textit{ avec } \aleph = \begin{pmatrix} 1-p & & & 0 \\ & \ddots & & \vdots \\ 0 & & 1-p & 0 \\ p & \cdots & p & 1 \end{pmatrix}$$

Ce que l'on peut encore écrire comme :

$$\aleph = (1-p) + p\perp \textit{ avec } \perp = \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ 1 & \cdots & 1 \end{pmatrix}$$

et qui signifie simplement que l'on a la probabilité p que la ligne de cache soit évincée à cause de la préemption et par conséquent la probabilité $1-p$ que rien de particulier ne se passe sur une variation élémentaire de s. Un calcul immédiat conduit au résultat suivant, dans le cas de p constant :

$$\pi^m = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph^m \end{pmatrix} \textit{ et } \aleph^m = (1-p)^m + [1-(1-p)^m]\perp$$

qui permet d'obtenir une expression du propagateur sur Δs quelconque. On peut facilement généraliser le résultat précédent au cas où la préemption ne provoque pas forcement un échec au prochain accès à la référence concernée. Supposons que l'on puisse déterminer que la préemption de la tâche fasse glisser la référence auquel on s'intéresse de i positions dans l'historique au sens LRU (ce que l'on peut appeler l'*empreinte de préemption*). Alors on aboutit aux résultats suivants :

$$\pi_i = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i \end{pmatrix} \textit{ avec } \aleph_i = (1-p) + p\lambda^i$$

λ étant l'opérateur de concurrence d'accès sur l'espace d'état de probabilité (section 6.4.2) et

$$\pi_i^m = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i^m \end{pmatrix} \text{ avec } \aleph_i^m = \left[(1-p) + p\lambda^i\right]^m$$

On voudrait à présent obtenir une expression de p en fonction des autres paramètres. Considérons pour cela un programme totalement hypothétique dont la seule action est d'accéder à une référence unique donnée. Si on considère la phase d'initialisation passée, un tel programme ne générera aucun échec de cache si la préemption n'est pas prise en compte. Ainsi, la seule source d'échecs de cache vient des effets de la préemption, en calculant le nombre d'échecs dus à la préemption et en le comparant à n_{max} on obtiendra ainsi l'expression de p en fonction des autres paramètres pertinents du modèle. Voyons ce que donnent les calculs à présent.

On commence par définir l'opérateur ξ :

$$\xi = \pi a^\dagger = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ 0 & \tau_p \end{pmatrix} \text{ avec } \tau_p = (1-p)\top + p\bot$$

comme on le voit, pour simplifier, on utilise pour ce calcul le cas où la préemption provoque une sortie de la référence considérée du cache (utilisation de $\bot$ plutôt que λ^i car $\forall \nu \geq A,\ \lambda^\nu = \bot$). La séquence du programme étudié va donc être modélisée par les puissances de l'opérateur ξ. τ_p étant idempotent, on en déduit immédiatement que :

$$\xi^k = \begin{pmatrix} \mathbb{1} & \mathbb{1} + k\tau_p \\ 0 & \tau_p \end{pmatrix}$$

et donc pour $k = N$ qui correspond à l'exécution complète de la tâche, on doit obtenir n_{max} échecs de cache. Ainsi :

$$Np = n_{max} \;\Leftrightarrow\; p = \frac{n_{max}}{N}$$

On peut aussi se demander ce qu'il se passe lorsque les accès à la référence ne sont pas aussi regroupés et que $\Delta s \neq 1$ donc $\xi_m = \pi^m a^\dagger$ est l'opérateur de base. On constate alors que l'on a les mêmes calculs à condition de remplacer $1-p$ par $(1-p)^m$ et p par $[1-(1-p)^m]$. A la limite $k = N$ on obtient le résultat suivant :

$$p = 1 - \left(1 - \frac{n_{max}}{N}\right)^{\frac{1}{m}}$$

A noter que si on choisit le cas d'accès uniformément répartis, on aboutit à $m = \frac{N}{A}$ et à la probabilité correspondante. Néanmoins, le but étant un calcul au pire, on peut si on le souhaite se contenter de l'expression précédente de p qui est suffisante dans la plupart des cas, et est de toute façon pessimiste.

THEORÈME 7.3.1. *L'opérateur π_i est l'opérateur de préemption fondamental, ou propagateur pour la progression dans l'exécution du programme pour une référence donnée. On peut l'exprimer sous la forme :*

$$\pi_i = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i \end{pmatrix} \text{ avec } \aleph_i = (1-p) + p\lambda^i$$

où i est la profondeur de l'empreinte mémoire pour les tâche pouvant entrainer une prémption, λ est l'opérateur de concurrence d'accès sur l'espace d'état de probabilité et où la probabilité de préemption p a l'expression suivante :

$$p = \frac{n_{max}}{N}$$

7.4. Calcul de l'écart-type associé

On va mener ce calcul important en deux étapes. Tout d'abord effectuer le calcul pour une référence donnée puis, dans un deuxième temps, le faire pour un ensemble de références correspondant à l'exécution de la tâche. Il s'agira dans ce dernier cas de prendre en compte la plus ou moins grande cohérence des accès entre eux comme nous le verrons. Le cas échéant nous calculerons un majorant de l'écart type si l'expression exacte est difficile à manier.

7.4.1. Cas d'une référence seule. On procède par comparaison avec le cas de l'accès d'une référence à chaque instruction. On a parcouru Δs dans l'exécution de la tâche entre le précédent accès et l'accès actuel. On suppose que la seule source d'éviction de la référence mémoire considérée est due aux éventuelles préemptions (sinon, il est inutile de compter cet accès pour la référence). La probabilité d'échecs associée est notée $p_{\Delta s}$.

Le déroulement en accès permanent du programme, devrait conduire à $\Delta s \times p$ échecs de cache alors que l'on en voit que $p_{\Delta s}$. On en déduit l'écart type total pour la référence, dans le cas où $n_{max} \geq \mathcal{A}$:

$$\sigma^2 = < m^2 > - < m >^2 = n_{max}^2 - < \sum_{i \in (accès ref)} p >^2$$

et si on généralise cela au cas de n références, on aboutit au résultat suivant :

$$\sigma^2 = n^2 \times n_{max}^2 - < \sum_{j \in \mathcal{J}} p_{i,j} >^2$$

qui correspond une séquentialisation artificielle des accès et donc à une forte surestimation de l'écart type dans le cas où les accès ne sont pas fortement corrélés. L'ordre de grandeur typique de cette estimation est en effet $n \times n_{max}$. On peut aussi faire une hypothèse d'accès totalement décorrélés entre

les références mémoires et l'écart type associé sera alors de la forme :

$$\sigma^2 = n \times n_{max}^2 - \sum_{j \in \mathcal{J}} < \sum_i p >^2$$

mais qui est cette fois sous-estimée dans le cas général. Il faut donc trouver un moyen de faire une évaluation moins évasive de l'écart type dans le cas de références multiples.

7.4.2. Cas de références multiples. En général les accès à des références distinctes n'ont pas lieu en même temps lors de l'exécution d'une tâche. De ce fait la préemption n'est pas forcement dans une situation ou elle peut provoquer un nombre toujours maximum d'échecs de cache. Il s'agit donc de tenir compte de la plus ou moins grande cohérence des accès entre chaque référence pour réduire d'autant l'écart type. Intuitivement, un système décorrélé aura un écart type dont l'ordre de grandeur sera proportionnel à la racine carré du nombre d'accès, tandis qu'un système très corrélé aura plutôt un écart type proportionnel au nombre d'accès (il suffit de prolonger les calculs effectués au paragraphe 7.1 pour s'en rendre compte).

Si on regarde ce qui se passe au niveau d'une référence, on constate que chaque accès s'accompagne d'une décroissance exponentielle de la probabilité de *hit*, se que l'on peut encore exprimer comme une propagation évanescente des *hits* au cours de l'exécution d'une tâche dans un cadre multitâche. On va utiliser cela pour calculer un majorant de la corrélation et de l'écart type.

A la suite immédiate d'un accès, on obtient sur la référence accédée une probabilité de *hit* de 1. Si on somme toutes les probabilités de *hit* à ce moment là, on obtient un majorant du nombre d'échecs supplémentaires que peut provoquer une préemption effective au moment considéré. Donc

si on nomme t_a l'instant d'accès à une référence on a

$$\delta m(t_a) = \sum_{i \in \mathcal{I}} \left(1 - p_{\emptyset,i}(t_a)\right)$$

Et en choisissant a_1, a_2, $\ldots a_{n_{max}}$ les accès qui maximisent $\delta m(t_a)$, on en déduit un écart maximum majoré :

$$\Delta m = \sum_{j \in \{1,\cdots n_{max}\}} \delta m(t_{a_j})$$

et un majorant de l'écart type :

$$\sigma^{*2} = \sum_{j \in \{1,\cdots n_{max}\}} \delta m^2(t_{a_j})$$

On peut noter que $\sigma^* \sqrt{n_{max}}$ fournit l'ordre de grandeur de l'écart maximum à l'évaluation proposée, au point de vu des échecs de cache. On utilisera donc généralement $\bar{m}(n_{max}) + \sigma^* \sqrt{n_{max}}$ comme majorant du nombre d'échecs de cache et en déduire un *WCET*.

CHAPITRE 8

Passage des échecs de cache au temps d'exécution au pire

On peut distinguer deux cas dans le cadre du passage des évaluations d'échecs de cache au temps d'exécution. Pour les processeurs les plus anciens, les échecs de cache vont bloquer le pipeline. Pour les processeurs modernes, les échecs de caches ne bloquent pas forcement le pipeline, le pipeline se bloque cependant si le jeu des dépendances des instructions ou des données fait en sorte qu'aucune instruction en cours d'exécution ne peut continuer sans avoir les résultats des accès mémoires.

8.1. Échecs de cache bloquants pour le pipeline

On peut dans ce cas précis simplement compter que chaque échec de cache va produire un délai égal au nombre maximum de cycles que peut nécessiter un accès mémoire principale. Le temps d'exécution de la tâche considérée sera donc simplement le temps d'exécution de celle-ci seule avec un cache parfait augmenté de la quantité correspondant au nombre d'échecs de cache multiplié par le délai maximum dû à un accès à la mémoire principal. La correspondance entre l'écart type en nombre d'échecs de cache σ_m et l'écart type en temps d'exécution σ_T est donc une simple relation linéaire.

8.2. Échecs de cache non bloquants

C'est dans ce cas précis qu'il faut passer par un simulateur du processeur afin de savoir à quel moment les dépendances de données ou d'instruction dues à l'échec de caches vont engendrer une attente de résolution de l'échec.

Il s'agit souvent d'évaluer ce temps au plus tôt, afin de pouvoir calculer un temps de retard dans le pire des cas. Ce temps est bien sûr majoré par le pire temps de rapatriement depuis la mémoire centrale (on a alors l'équivalence avec des caches bloquants).

Il est a noter que pour les chargement spéculatifs, il n'y aura pas de dépendance des instructions suivantes avec l'accès à la mémoire centrale et donc pas de délais associés même en cas d'échecs. Les délais ne seront possibles que lorsqu'un rapatriement depuis la mémoire centrale doit être utilisé suffisamment peu de temps après par le cœur du processeur.

De ce fait, les calculs de temps avec une grande précision exigent un investisment dans le développement d'un tel simulateur de dépendances. Ce problème est cependant commun à toutes les approches de calculs de WCET pour les caches, mais aussi le pipeline. Il n'a donc rien de particulier à celle-ci, mais constitue un passage obligé assez délicat. Le développement théorique qui a été réalisé au cours de cette thèse n'a pas permis de libérer suffisemment de temps pour mettre en œuvre un tel simulateur. Il a donc fallu se contenter de modéliser manuellement des programmes suffisamment simples pour que cela reste du ressort humain afin de réaliser des tests et d'évaluer l'efficacité de notre modèle. C'est ce que nous allons voir à présent.

Quatrième partie

Résultats et conclusion

CHAPITRE 9

Résultats obtenus par le modèle

9.1. Les benchs utilisés

Pour l'évaluation, on utilise des programmes simples car la simulation est créée à la main à partir des résultats de compilation des programmes en question à partir de GCC sur plateforme PowerPC (version 2.95.3 avec une directive d'optimisation "-O3"). Les programmes utilisés pour tester le modèle sont les suivants :

- produit de deux matrices 50×50 en flottants double précision ;
- somme de deux matrices 38×38 en flottants double précision ;
- transformation de Fourier rapide (FFT) 512 points, en flottants double précision ;
- normalisation d'un vecteur 6000 points, en flottants double précision ;
- tri à bulle de 4200 nombres (de façon à tester un programme dont le graphe de contrôle est non trivial).

Ces programmes restent assez simples du fait que les simulations sont encore codées à la main. Pour évaluer un cas plus proche d'une application réelle, il faudrait réaliser des outils supplémentaires, comme nous l'avons vu. Les tests seront faits dans le cadre du régime permanent de tâches qui bouclent, ce qui permet de les mettre dans une réelle perspective de système temps-réel.

9.2. Les résultats

Dans le tableau 1, on a comparé des résultats d'une simulation à ceux du modèle que l'on a développé.

Les résultats obtenus sont tout de même très bons compte tenu de l'option systématiquement pessimiste de l'évaluation. On constate en particulier que cette méthode est très bonne sur le code séquentiel, comme on pouvait s'y attendre. De plus, il est probable que l'on pourrait améliorer les résultats de la normalisation de vecteur en affinant l'implémentation. Nous n'avons pas noté σ^* pour le cas sans préemption car il est rigoureusement nul (le cadre est alors déterministe).

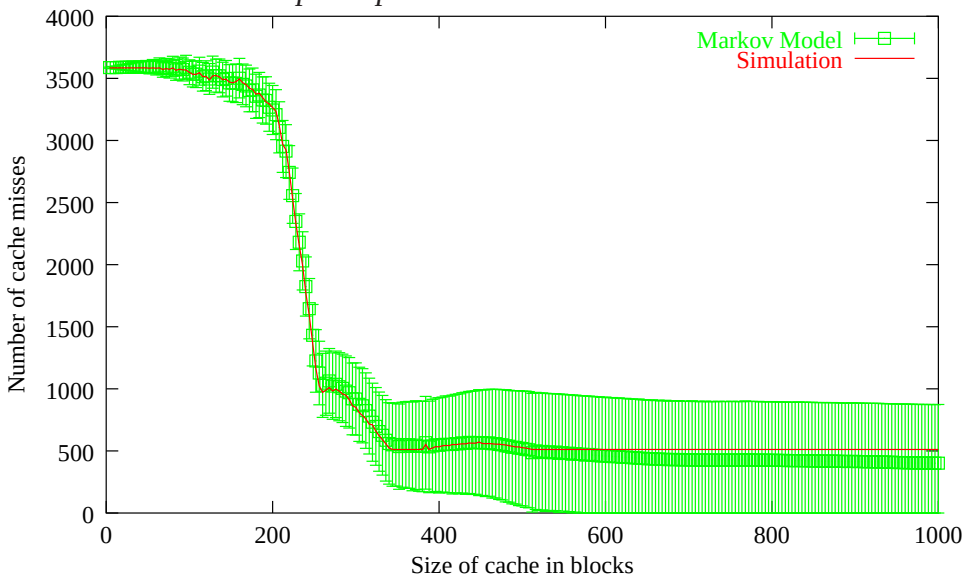
En ce qui concerne le cas avec préemption, on s'est limité dans le tableau à trois préemptions, mais il est aussi simple et rapide d'obtenir des résultats pour un nombre de préemptions supérieur, et les résultats constatés gardent la même tendance, comme on le verra au paragraphe 9.3.3, donc il n'est pas utile d'ajouter un grand nombre de colonnes qui apporterait peu d'informations. De manière générale, on constate que l'on reste toujours très largement en dessous de $\sigma^*\sqrt{n_{max}}$, ce qui était attendu mais que cet écart type majoré reste raisonnable en magnitude malgré tout.

Pour avoir une idée plus précise de la façon dont évolue l'évaluation lorsque la taille du cache varie, on peut se référer à la figure 9.2.1 où on voit les nombres d'échecs de cache évalués, avec un écart type et les résultats de simulation sur le même graphique, pour le programme de FFT, dans le cas d'une seule préemption. On constate que pour un tel programme qui fait un appel important à la mémoire cache, les résultats de simulation restent très proche de l'évaluation directe et que l'écart majoré est très important vis à vis du comportement réel du cache. Néanmoins sur des programmes faisant moins appel au cache (du style de l'addition matricielle) ce résultat aurait

TAB. 1. *Comparaison pour les échecs de cache de la simulation m_{max} avec les résultats du modèle ($\bar{m}$ et σ^* lorsque cela s'applique).*

nombre de préemptions		0		1			2			3		
		m_{max}	$\bar{m}$	m_{max}	$\bar{m}$	σ^*	m_{max}	$\bar{m}$	σ^*	m_{max}	$\bar{m}$	σ^*
Produit	4Ko	**32565**	32565	**32629**	32606	129	**32656**	32647	182	**32706**	32688	223
matriciel 50×50	16Ko	**1250**	1250	**1889**	1875	970	**1941**	2490	1309	**2016**	3094	1538
Addition	4Ko	**1083**	1083	**1086**	1106	125	**1089**	1129	171	**1092**	1151	203
matricielle 38×38	16Ko	**295**	295	**1084**	750	804	**1089**	1063	929	**1092**	1283	963
FFT	4Ko	**3520**	3520	**3523**	3550	129	**3526**	3579	182	**3529**	3608	223
512 points	16Ko	**0**	0	**512**	402	473	**768**	692	622	**1024**	927	715
Normalisation	4Ko	**2744**	2808	**2872**	2844	127	**3000**	2880	177	**3003**	2914	214
Vectorielle 6000pts	16Ko	**952**	1464	**1976**	1949	913	**3000**	2339	1160	**3003**	2662	1286
Tri à bulle	4Ko	**2166140**	2166200	**2166272**	2166240	129	**2166400**	2166280	182	**2166528**	2166320	223
4200 nombres	16Ko	**7020**	7532	**8048**	8111	1025	**8968**	8679	1449	**9884**	9236	1775

FIG. 9.2.1. *Évaluation du nombre d'échecs de cache en fonction de la taille de cache pour le programme de FFT avec une seule préemption*



été moins net, même si, par construction, la simulation serait restée toujours à l'intérieur de l'intervalle d'erreur obtenu par le modèle.

9.3. Évaluation du WCET sur les exemples

9.3.1. Hypothèses de travail. Pour fixer les idées, les calculs de WCET sont faits à partir d'un modèle d'architecture super-scalaire type PowerPC. On suppose que l'on dispose de 2 unités de calcul entier, une unité *load/store*, une unité de calcul flottant et une unité de branchement. Le chargement est effectué dans l'ordre de même que la terminaison des instructions. Le dispatch et l'exécution sont supposées pouvoir être réalisés dans le désordre, tant que l'on respecte les dépendances de données. Pour simplifier, on suppose que l'on dispose de suffisamment de registres physiques pour qu'il n'y ait jamais d'arrêt du pipeline pour des problèmes de renomage ; de même la mauvaise prédiction de branchement sera moyennée lorsqu'il existe de

réels aléas de flot de contrôle. Ces hypothèses auront tendance à donner une évaluation qui augmente la part des caches sur le temps d'exécution.

On choisit également une hypothèse de cache non bloquants mais on simplifiera parfois certaines hypothèses pour rendre les calculs plus simples, si on peut s'assurer que l'ordre de grandeur des résultats en est faiblement influencé. Cette simplification légère est rendue nécessaire par le fait que toutes les évaluations sont faites à la main, toujours à partir du code objet généré par le compilateur. On choisit également un temps de rapatriement depuis la mémoire centrale de 8 cycles processeurs, ce qui semble raisonnable au vu des architectures des systèmes temps-réels critiques.

9.3.2. Les résultats sur les exemples. L'évaluation du WCET par la méthode sera faite par l'hypothèse de $m_{maj} = \bar{m}(n_{max}) + \sqrt{n_{max}}\sigma^*$, comme on l'a vu au paragraphe 7.4.2. Le WCET exact est obtenu par simulation et Δt est l'écart entre l'évaluation et cette valeur exacte. On calculera également la surestimation en pourcentage du temps d'exécution, afin d'avoir des éléments de comparaison avec d'autres méthodes de la littérature. Tous ces résultats sont synthétisés dans le tableau 2.

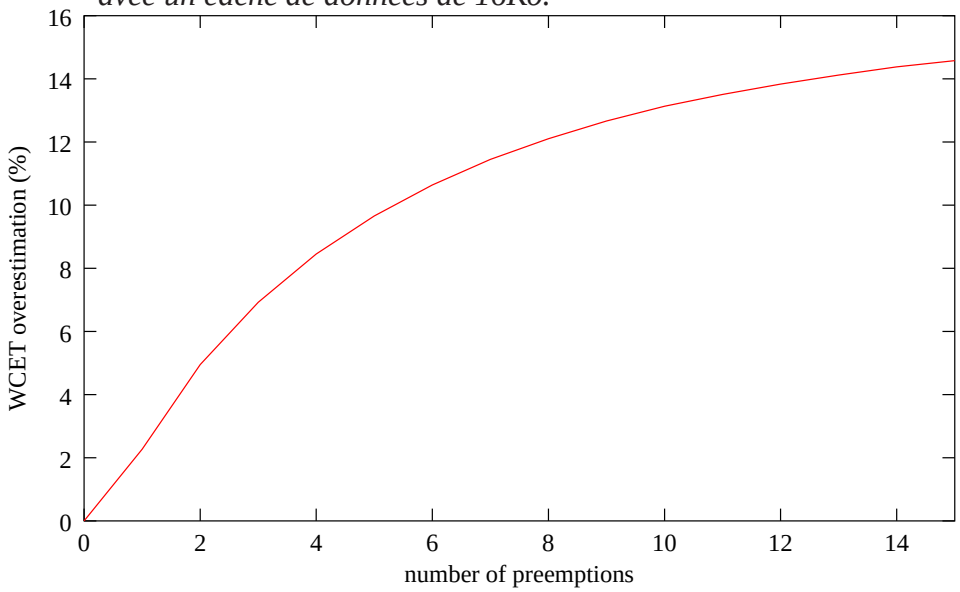
Comme on le voit les résultats sont très bons, car tout à fait comparables et même meilleurs que les autres méthodes d'interprétation abstraite dans le domaine pour le cas sans préemption et des résultats intéressants et inédits pour le cas avec préemption.

9.3.3. Évolution de l'évaluation du WCET avec le nombre de préemption. Cette évaluation a été réalisée sur le cas de la transformée de Fourier rapide (FFT) pour un nombre de préemptions variants de 0 à 15 (au delà, tout les accès en cohérence temporelle sont des échecs de cache dans le pire cas, et donc on peut envisager des techniques annexes de réduction des incertitudes). Les résultat sont montrés en figure 9.3.1 où l'on peut voir

TAB. 2. WCET *en nombre de cycles, surestimation en nombre de cycles et en pourcentage*

nombre de préemptions		0			1			2			3		
		WCET	Δt	%	WCET	Δt	%	WCET	Δt	%	WCET	Δt	%
Produit	4Ko	**827316**	0	0,00	**825769**	625	0,08	**827852**	1465	0,18	**828147**	2173	0,26
matriciel 50×50	16Ko	**641307**	0	0,00	**645572**	5258	0,81	**646634**	14161	2,19	**647076**	22077	3,41
Addition	4Ko	**13486**	0	0,00	**13507**	280	2,07	**13528**	287	2,12	**13549**	476	3,51
matricielle 38×38	16Ko	**7085**	0	0,00	**13493**	812	6,02	**13528**	1918	14,2	**13549**	2296	16,9
FFT	4Ko	**142354**	0	0,00	**142377**	1170	0,82	**142400**	2328	1,63	**142422**	3489	2,45
512 points	16Ko	**115954**	0	0,00	**119794**	2723	2,27	**121714**	6027	4,95	**123634**	8561	6,92
Normalisation	4Ko	**92010**	186	0,20	**92381**	287	0,31	**92752**	378	0,41	**92761**	620	0,67
Vectorielle 6000pts	16Ko	**86812**	1485	1,71	**89782**	2569	2,86	**92752**	2840	3,06	**92761**	4285	4,62
Tri à bulle	4Ko	**189434652**	360	0,00	**189435444**	582	0,00	**189436212**	824	0,00	**189436980**	1069	0,00
4200 nombres	16Ko	**176479932**	3072	0,00	**176486100**	6528	0,00	**176491620**	10561	0,00	**176497116**	14558	0,00

FIG. 9.3.1. *Variation de la surestimation du* WCET *en fonction du nombre de préemption pour le cas de la FFT, avec un cache de données de 16Ko.*



la variation de la surestimation en fonction du nombre de préemptions, en pourcentage. Comme on le voit, la surestimation ne dépasse pas 15% même pour un nombre élevé de préemptions, ce qui est un très bon résultat, sachant que le cas d'étude présenté ici est le plus défavorable de nos programmes d'exemple qui fasse une réelle réutilisation des données (le cas de l'addition matricielle est lui caractéristique d'une application qui ne réutilise pas ses données et où seul le phénomène de cohérence spaciale intervient, donc d'une application où l'intérêt de l'utilisation du cache peut se poser). L'hypothèse de cache toujours invalide dans ce cadre de la FFT conduirait, par comparaison, à des surestimations allant jusqu'à 75% dans ce cas (en particulier à nombre de préemption faible où les résultats de surestimation par la méthode sont inférieurs à 10%). Le gain par rapport à une non utilisation des caches est donc important, même dans un cas où la méthode n'offre pas ses meilleurs résultats.

9.4. Coût calculatoire de la méthode

Le modèle que l'on utilise est fondé sur une algèbre d'opérateurs. Pour les processeurs actuellement utilisés, dont l'associativité A est typiquement 2, 4 ou 8 pour les caches L1, les opérateurs correspondants seront des matrices de $\mathcal{M}(6)$, $\mathcal{M}(10)$ ou $\mathcal{M}(18)$. Les matrices étant très creuses, il est simple de se réduire à une complexité de $O\left((A+1)^2\right)$ pour les produits d'opérateurs et $O(A+1)$ pour les calculs de vecteurs ce qui permet de garder des calculs relativement abordables. Si $\mathcal{S}$ est le nombre total maximal d'accès mémoire, et $l = Card\, \mathcal{J}$, le nombre total d'opérations matricielles à réaliser est $\mathcal{S}\frac{l}{n}$ qui peut être un nombre assez important tout de même. Avec la bibliothèque de modélisation qui est pourtant loin d'être totalement optimisée, on dépassait rarement les 10 minutes de calculs pour quelques centaines de millions de produits matriciels, sur des stations assez anciennes. En optimisant la bibliothèque de calcul pour utiliser les unités vectorielles que l'on rencontre souvent sur les machines actuelles, il semble raisonnable de penser qu'on ne dépasserait pas l'heure de calcul même pour des applications de grande taille avec quelques milliards d'accès mémoire. De plus, comme on l'a vu au paragraphe 6.7.3, il est assez simple de paralléliser les calculs d'opérateurs généraux, en plus de la parallélisation des calculs de produits matriciels, ce qui permet encore de gagner quelques ordres de grandeurs sur les temps de calcul. Enfin, si pour une raison ou pour une autre, les calculs paraissent encore trop longs, on peut se contenter de calculer les vecteurs d'états, ce qui permet de tendre à des coût calculatoires proche des autres méthodes dans le domaine, moyennant, il est vrai, une dégradation des résultats obtenus.

9.5. Comparaison avec les autres méthodes

En ce qui concerne les publications récentes pour l'évaluation dans le cadre monotâche, on peut citer plus particulièrement [**11, 19, 26, 25, 20**]. Les résultats obtenus sur des *benchs* semblables sont dans tout les cas les mêmes, ou comparables. Ceci n'est pas très étonnant du fait qu'il s'agit dans pratiquement tout les cas d'interprétation abstraite. Les seules limites sur la bonne prédiction sont les limites normales dues à l'analyse statique qui ne permet pas de connaître certaines informations qui sont spécifiques à une exécution en cours (*run time parameters*). Une différence fondamentale, cependant est que l'on s'affranchit ici de l'hypothèse de démarrage avec des caches totalement invalidés que les autres méthodes sont obligées de faire pour effectuer des calculs. De ce fait les résultats du modèle sont mieux adaptés à un système temps-réel où, par essence, les tâches sont en constante ré-exécution.

Pour le cadre multitâche, par contre, il n'existe aucun équivalent dans un cadre comparable et surtout aussi large, du fait que les hypothèses que l'on a faites sur les préemptions sont minimales. C'est également la première fois qu'un formalisme permet d'unifier l'approche monotâche et l'approche multitâche des calculs de *WCET* liés au caches. Les résultats obtenus au niveau des surestimations des temps sur des programmes caractéristiques sont de l'ordre de quelques pour-cents et donc très bons, le tout au prix d'un calcul dont le coût reste tolérable.

La démarche est donc prometteuse même si on ne peut pas dire encore ce que cela donnera sur des programmes en vrai grandeur. De plus, le traitement automatique des programmes exigera de déterminer des contraintes diverses comme les bornes de boucles, ou des chemins d'exécutions mutuellement exclusifs qui ne sont pas simples à mettre en œuvre, mais dont la

difficulté n'est pas moindre sur les autres approches et dont certains aspects ont déjà été traités dans la littérature. Il s'agit donc de difficultés classiques de passage d'un cadre encore assez formel à un cadre applicatif pur et relève pour la plus grande partie plus de l'ingénierie que de la recherche. Bien que cela soit également intéressant, cela ne semble pas être du cadre d'une thèse.

9.6. Extension à plusieurs niveaux de caches

Dans le cadre d'une architecture mémoire à plusieurs niveaux de caches, il existe plusieurs techniques de mise à jour des caches de niveaux supérieurs (les plus externes) en fonction de ceux de niveau moins élevé. Les deux politiques les plus courantes sur des systèmes mono-processeurs sont "*write through*" pour laquelle chaque écriture en mémoire est reportée à travers tous les niveaux à chaque fois et "*copy back*" pour laquelle l'écriture en mémoire est différée autant que possible, c'est à dire jusqu'à l'instant où la référence mémoire est expulsée du cache du fait d'autres accès concurrents. Dans le premier cas, on privilégie la cohérence de l'état mémoire et l'étalement des accès bus dans le temps au détriment d'une charge moyenne élevée sur celui-ci ; dans le second, on privilégie au contraire une charge aussi faible que possible sur le bus, mais au prix d'une difficulté à connaître l'état de la mémoire à un instant précis, et surtout une probable concentration des accès bus sur des intervalles de temps assez courts avec des risques importants de saturation du bus, entraînant des aléas temporels supplémentaires qui sont de plus très difficiles à prévoir car largement dépendants de paramètres connus seulement au moment de l'exécution (*run time*).

La politique *copy back* est surtout intéressante pour des variables temporaires qui sont, par essence, souvent ré-écrites et dont le contenu n'est pertinent qu'à l'instant considéré de son utilisation mais pas après. En fait,

il s'agit alors de suppléer au manque de registres physiques de l'architecture cible par le stockage de données temporaires en mémoire. C'est un schéma normal dans une architecture CISC, mais en principe exceptionnelle dans une architecture RISC car un de ses paradigmes est d'utiliser un nombre de registres physiques élevé[1] pour limiter les accès en mémoire. Compte tenu de l'obsolescence des architectures CISC et de la difficulté de calculer des temps d'accès mémoire avec la politique *write-back*, il est probable que la politique *write-through* doive être favorisée dans le cadre des applications temps-réel critiques. Il est tout de même possible d'utiliser des techniques d'interprétation abstraite pour obtenir des majorants d'occupation du bus mémoire avec cette politique[**9**], mais cela complique le modèle. A noter également que cela complique énormément les évaluations dans le cas où il existe de multiples niveaux de caches. Il reste tout de même un avantage important du modèle *copy back* qui est de pouvoir regrouper les réécritures de blocs en mémoire, en mode rafale (utilisation avantageuse de la localité des accès). Cet avantage peut quand même être tempéré par l'utilisation d'astuces de programmation comme dépliement de boucles et regroupement des accès. On peut aussi imaginer que des méthodes comme le *buffered write-through* qui sont déjà mis en œuvre pour les caches disques le soit aussi pour les caches processeurs, la croissance du nombre de transistors disponibles pour les futures architectures aidant.

La politique *write-through*, quant à elle, rend triviale l'évaluation des échecs de caches à chaque niveau pour le cas de multiples niveaux de caches, car il suffit alors d'appliquer la méthode en parallèle sur tous les niveaux de caches présents, indépendamment. Seule l'évaluation des temps

[1] typiquement 32 registres généraux entiers sur les architectures 32 bits actuelles et nettement plus sur IA-64

en eux même va devoir s'occuper de la présence ou de l'absence des données considérées pour obtenir les temps d'accès proprement dits. C'est donc très probablement la solution à adopter dans ce cas précis, sauf exception ou avancée significative dans le modèle.

9.7. A propos de multi-processeurs SMP

Il y a de nombreuses façons de mettre en œuvre un système à plusieurs processeurs. Une technique assez courante aujourd'hui est la mise en œuvre dite SMP[2] qui est telle que plusieurs processeurs semblables partagent le même bus mémoire de manière équivalente, de sorte qu'une unique mémoire centrale puisse être partagée par tous les processeurs en présence. Dans ce cadre, il existe des processus complexes de synchronisation des caches qu'il est inutile d'exposer ici mais que l'on pourra trouver dans [**14**]. De ce fait, il est difficile de connaître l'état réel de la mémoire à un instant précis, surtout en cas d'accès concurrents. Il faut noter cependant, qu'à moins d'avoir des programmes symptomatiques, le cas des accès concurrents, surtout dans un domaine aussi sensible que les applications temps-réel critique, doit mettre en œuvre des mécanismes de coordination et donc de sérialisation. L'intérêt du programmeur est, bien-entendu de limiter ces phénomènes qui sont une limitation au parallélisme intrinsèque. Le modèle de programmation temps-réel doit donc fournir par lui même une partie du mécanisme de cohérence mémoire, en particulier des caches dans cet environnement particulier. Le plus simple est, dans le modèle que l'on a exposé ici, de mettre les zones d'échanges de données dans des zones mémoires non cachable (l'intérêt de les positionner dans le cache étant souvent minime, car ce sont le plus souvent des données périssables et, en conséquence, rarement relues). Le modèle s'applique alors sans adaptation particulière.

[2] *Symetric Multi-Processing* soit système multiprocesseurs symétrique

Conclusions et perspectives

Synthèse

Le but de cette thèse était de défricher une voie d'approche globale pour le traitement des aléas de temps d'exécution des tâches d'application temps-réel, dus aux caches. Dans l'exposé que l'on vient de faire, on a pu voir le contexte qui avait amené à cette problématique et en particulier l'importance des problèmes de caches mémoire sur le déterminisme des temps d'exécution, en particulier dans le cadre multitâche. L'état de l'art nous a conduit à mettre en avant les modèles d'interprétation abstraite dans le but de résoudre ce problème, mais aussi de voir l'insuffisance des approches faites jusqu'alors.

Malgré une modélisation sur des bases semblables d'interprétation abstraite, cette thèse a permis de dégager une nouvelle voie pour l'évaluation de temps d'exécution au pire pour les systèmes disposant de caches mémoire. Au contraire des modèles précédents, l'approche s'oriente vers un formalisme de type probabiliste à base de chaînes de Markov. Il est a noter que dans le cadre monotâche, le modèle reste parfaitement déterministe et fournit un majorant sûr du temps d'exécution au pire pour la tâche considérée qui est très semblable à celui que fournirait une autre approche reposant sur l'interprétation abstraite. Mais le résultat est tout de même plus précis car il est simple de s'affranchir de toute hypothèse sur les états initiaux des caches, au prix, il est vrai, de calculs sensiblement plus lourds.

Dans le cadre multitâche, on utilise de façon astucieuse le fait que le formalisme est probabiliste pour étendre directement les résultats monotâche vers le multitâche. Il a suffit pour cela de modifier le propagateur du vecteur d'état lors de l'avancement du programme. On pourrait comparer cela à la propagation d'une onde dans un milieu dissipatif. Le cadre monotâche est alors comparable à une propagation sans dissipation, et le cadre multitâche à une propagation dans un milieu dispersif. Le but étant d'évaluer un cas d'exécution au pire, il faut également calculer un écart maximum, comme on l'a vu.

On a pu montrer de façon théorique les avantages importants que l'on peut obtenir en procédant ainsi, et on a pu voir une évaluation sur des cas concrets, mais simples du fait de la modélisation manuelle de la méthode. Les résultats obtenus sont d'ores et déjà encourageants. Il est évident qu'il faut maintenant aller plus loin et développer les outils qui permettront un traitement automatique de l'évaluation des WCET afin d'obtenir des résultats plus substantiels, sur des cas réels, ou proches de cas réels.

Extensions possibles

Le modèle que l'on vient de développer se généralise sans difficultés à n'importe quel produit cartésien d'automates, d'autant plus simplement que les automates en question sont référencés par adresse ou groupe d'adresses séquentielles. Ainsi, une généralisation possible du formalisme se situe au niveau des prédicteurs de branchements qui sont uniquement référencés par adresses, par exemple les prédicteurs 1 bit (pris, non pris) et deux bits, que l'on a évoqués à la section 3.3. Le formalisme mis en œuvre ici, est donc généralisable à un certain nombre d'autres aléas architecturaux, et même probablement à d'autres domaines avec bonheur. En définitive c'est tout un

champ d'exploration de l'interprétation abstraite que cette thèse a permis de défricher.

Annexe 1

Modélisation “à la main” d’un programme pour le modèle

On a vu que le modèle avait été testé sur des programmes modélisés “à la main” à partir du code généré par GCC. Nous allons prendre l’exemple du tri à bulle pour voir ce que cela signifie.

Nous avons pris un code classique de tri à bulle tel qu’on peut le voir sur l’algorithme .0.1.

ALGORITHME .0.1. *Tri à bulle*

```
#define false 0
#define true !false
#define lim 4200

void main()
{
int i,k,swap=true;
double temp,sa[lim];

    for(i=0; swap&&(i<lim-1); i++)
    {
        swap=false;
        for(k=0;k<lim-i-1;k++)
            if(sa[k]>sa[k+1])
            {
                temp=sa[k];
```

```
                sa[k]=sa[k+1];
                sa[k+1]=temp;
                swap=true;
            }
    }
}
```

La compilation par GCC 2.95.3 avec l’option d’optimisation “-O3” donne ce que l’on voit en algorithme .0.2.

ALGORITHME .0.2. *Code assembleur généré*

```
main :
    mr 12,1
    lis 0,0xffff
    ori 0,0,17520
    stwux 1,1,0
    li 9,0
.L6 :
    li 6,0
    subfic 0,9,4199
    cmpw 0,6,0
    addi 5,9,1
    li 8,0
    bc 4,0,.L5
    mr 7,0
    addi 11,1,8
.L10 :
    addi 0,8,1
    slwi 9,8,3
```

```
    slwi 10,0,3
    lfdx 13,11,9
    lfdx 0,11,10
    mr 8,0
    cmpw 7,8,7
    fcmpu 0,13,0
    bc 4,1,.L9
    stfdx 0,11,9
    li 6,1
    stfdx 13,11,10
.L9 :
    bc 12,28,.L10
.L5 :
    mr 9,5
    cmpwi 7,9,5998
    cror 31,30,28
    mfcr 0
    rlwinm 0,0,0,1
    and. 11,6,0
    bc 4,2,.L6
    lwz 11,0(1)
    mr 1,11
    blr
```

Comme on le voit, la structure de boucle n’est pas transformée lors de la compilation ce qui permettra d’appliquer le même squelette d’exécution au programme de simulation du modèle que le programme C d’origine. Les accès mémoire sont soulignés. C’est eux qu’il va falloir faire entrer dans

le modèle. Il pour la structure du modèle, il faut applanir les aléas de flot de contrôle pour les remplacer par leur modélisation dans le cadre de la théorie. D’où le programme suivant qui correspond à l’implémentation du modèle sur le programme de tri à bulle.

ALGORITHME .0.3. *Modélisation du tri à bulle*

```
#include <iostream.h>
#include <stdlib.h>
#include "cache-op.h"

const int N=4200;
int TAILLE=(N+3)/4;

int main(int argc,char **argv)
{
      int i,j,k,n,m,m2,l,loop;
      int pos,pos_,brk=0,done=1;
      long wbuf;

      if(argc>1) brk=atoi(argv[1]);
      else brk=0;
      CacheOp Cmem(TAILLE,4,8+(15+10*N)*N,brk);
      bool stop=false;

      for(k=0;k<2;k++)
      {
            if(k==0) n=128;
            else n=1024; /* cache memory set */
```

```
Cmem.SetSize(n);
/* operating bubble sort for size n of CACHE */
m=0;
m2=0;
l=1;
cout<<n<<" ";

for(i=0; i<N-1; i++)
{
      for(j=0;j<N-i-1;j++)
      {
            // ld f10,sa[j]
            pos=j*4;
            pos_=pos+4;
            Cmem.PAccess(pos,brk*(9+(15+10*N)*i+10*j));
            // ld f13,sa[j+1]
            Cmem.PAccess(pos_,brk*(10+(15+10*N)*i+10*j));
            // control flow forked here
            Cmem.fork();
            // st f0,sa[j+1]
            Cmem.PAccess(pos_,brk*(14+(15+10*N)*i+10*j));
            // st f13,sa[j]
            Cmem.PAccess(pos,brk*(16+(15+10*N)*i+10*j));
            // alternate treatment
            Cmem.alt();
            // end of fork
            Cmem.unfork();
```

```
                }
            }

            // output results
            cout<<Cmem.nLoopMiss()<<" "<<Cmem.var()<<endl;
        }
}
```

Annexe 2 : Traductions anglaises

Introduction

Programming real-time systems is the art of guaranteeing a deterministic temporal comportment of programs. High level models like OASIS[**6**] allows to greatly simplify the programmer task who can then concentrate on program design, just giving the compilation tools some time constraints. Nonetheless, in order to guarantee that the system will be able to handle the tasks load, it is a necessity to obtain some upper bounds (and tighter is better) on Worst Case Execution Time (WCET) for each task the system may execute. Anyway, successive enhancements in processor architectures done over the last twenty years tends to worsen execution time determinism. Average execution speed grows quickly, but it is often harder to predict the real effect on worst case execution time. This is, however, what is the main interest for real-time programmer. Thus, some contradiction seems to appear between processors evolution and real-time systems needs. Especially, cache memories whose content can vary in a way very hard to predict and whose incidence on execution time can be important, is one of the harder issue.

That is why, since the beginning of the 90s, a number of research teams are looking for a way to predict cache comportment and compute related WCET. There is indeed an economic interest for such a kind of result since they would enable to use cache memories in hard real-time systems, so their costs can drop, allowing their use in widely spread systems like those met in motor industry. An exact resolution of the issue is out of the question,

since it is NP-complete varying with the number of instructions that can be executed. The actual goal is then to use real-time programs properties to reduce difficulties and to compute not exact WCET but a tight upper bound.

This thesis introduce a new method to obtain such a kind of results. Its main context is basic research around the OASIS project and formalism developed in CEA-Saclay in the *Laboratoire des Logiciels pour la Sûreté des Programmes* (LLSP - laboratory of softwares for program safety).

This report has the following organization: in a first time, essential aspects of issues are exposed in three chapters: the first developing main ideas and problems of hard real-time systems, the second is a short introduction to the OASIS model as a model for conception of hard real-time systems and the third chapter is a quick state of the art regarding processor architecture and especially associated time issues. The second part draw a state of the art for WCET computation models with chosen part of todays main theories in two chapters, the first one being an overview of possible models and the second one showing some important points regarding cache memories. The third part is the main thesis work and shows the theorical formalism. A first chapter concentrate on the model for sequential execution and shows the bases of the formalism and modeling of interaction between processor main core, cache memories and the main memory. The second chapter is a generalization of the model to the multitask field. A last chapter explains how to compute WCET from cache misses evaluations. Then the last part shows numerical results obtained by simulation before concluding.

Note for English translation readers

Only the third and the last part are translated in the following, since this shows the actual developments done in my thesis work. I should nonetheless describe what can be found in the first two parts.

First part:

chapter 1: defines real-time and some basic concepts of real-time programming models like loop programming, Event Triggered (ET) models and Time Triggered (TT) models[**15**]. Basic demands of hard real-time systems are described.

chapter 2: describe the OASIS model. This is a TT based model, people would refer to [**7**] for further information.

chapter 3: this chapter shows how modern processor architecture has an incidence on execution times. Some discussions are done for pipeline, superscalar architecture, branch predictors, MMUs and cache memories. Some quick remarks are also done for possible future architectural breakthroughs like VLIW/EPIC, trace caches, data prediction, asynchronous processors and virtual/emulated processors.

Second part:

first chapter: is a state of the art of cache related WCET with a focus on abstract interpretation models. A quick introduction to abstract interpretation is done[**5**] then some important works are introduced [**1, 8, 10, 11, 25, 27**].

second chapter: cache related models are discussed here. At first, a quick view of IRM model [**3, 12, 22**] is shown and that is it inadequate for our problem. Then some results for cache memories in multitask environment are shown [**16, 4**]. At last, the concept of memory footprint is introduced [**13, 24**].

The model developed in this work is related to abstract interpretation models. It shares some concepts, like may analysis but otherwise is a totally

new method for computing WCET. Its originality is to use probability formalism, although it obtain deterministic results.

CHAPTER A

Abstract interpretation model with Markov chains basis

A.1. General overview

The method being developed has also abstract interpretation as a basis but is much more inspired by probabilities than algebraic methods generally seen in papers. Thus, the computation becomes longer in time but a brand new field is open thanks to it, as shall be seen in this chapter and the next one. Nonetheless, the model still remains very accessible since it relies on Markov chains, one of the simplest Markovian formalism. This chapter shall introduce the model, the basic formalism, the implementation of the method and some results in the simple field of mono-task systems with a special interest for ever looping systems.

A.2. State space

A.2.1. Memory partition. As already seen, cache memory joins a set of successive addresses in a block. So let's divide the main memory in bloc-size elements, as they could be transfered to the cache (depending on its architecture and its associativity). Then let's number these blocs in main memory - thus numbers of each bloc in memory should be the address of any byte in it divided by the bloc size in bytes. We shall call these numbers *memory references* in the following (or *references* for short). For example, address 136 for a bloc size of 8 bytes is in the 17th memory reference and address 252 in the 31st memory reference.

A.2.2. Probability state space. The memory partition being made, let's associate for each reference i a probability state vector in a $A+1$ dimensions vector space $\mathcal{P}_i$, where A is cache associativity. This vector is defined as follows:

$$V \in \mathcal{P}_i,\ V = \begin{pmatrix} p_1 \\ \vdots \\ p_A \\ p_\emptyset \end{pmatrix} \ with \begin{cases} p_1\ probability\ next\ access\ is\ 1st\ for\ LRU\ history \\ \vdots \\ p_A\ probability\ next\ access\ is\ A-th\ for\ LRU\ history \\ p_\emptyset\ probability\ next\ access\ is\ cache\ miss \end{cases}$$

A.2.3. Number of accesses state space and reduced space associated. In order to have a complete description of the state for the reference being studied, it is also a necessity to know the associated number of accesses. Let's define for that purpose, a second $A+1$ dimensions vector space $\mathcal{N}_i$ for the number of accesses state space:

$$W \in \mathcal{N}_i,\ W = \begin{pmatrix} n_1 \\ \vdots \\ n_A \\ n_\emptyset \end{pmatrix} \ with \begin{cases} n_1\ total\ number\ of\ accesses\ as\ 1st\ in\ history \\ \vdots \\ n_A\ total\ number\ of\ accesses\ as\ A-th\ in\ history \\ n_\emptyset\ total\ number\ of\ misses \end{cases}$$

It should be noted that such an amount of variables can be a useless detail for the model. If need be, this space can be reduced to a two-dimensional space only, by using either $(n_h, n_\emptyset)$ with $n_h = \sum_{k=1}^{A} n_k$ the number of hits in the cache or $(n, n_\emptyset)$ with $n = \sum_{k=1}^{A} n_k + n_\emptyset$ the total number of accesses for this reference. The reduced space thus defined shall be called $\mathcal{N}_i^*$. It should also be noted that for the former choice for the reduced space and for a direct-mapped cache ($A = 1$), then $\mathcal{N}_i = \mathcal{N}_i^*$.

A.2.4. Global state space. Using the Cartesian product of both the previously defined spaces, it is now possible to define a global state space

for the memory reference i. Let's call $\mathcal{E}_i = \mathcal{N}_i \times \mathcal{P}_i$ (respectively $\mathcal{E}_i^* = \mathcal{N}_i^* \times \mathcal{P}_i$ the associated reduced space). Since the transformation from $\mathcal{N}_i$ to $\mathcal{N}_i^*$ is straightforward, we should concentrate on $\mathcal{N}_i$ and $\mathcal{E}_i$ in the following, but it should be reminded that using non reduced spaces would barely happens in practice.

Hypotheses required for this model are that cache associativity is a constant, and so is block size, that replacement policy is LRU[1] and that data or instructions are stored sequentially in the cache. These hypotheses are generally met in modern microprocessors. No constancy of associativity, of bloc size or a change in replacement policy can be taken into account by a change of the access operators discussed in section A.4. Nonetheless, taking into account non sequential caches like trace caches had not been studied. It should probably induce quite a large adaptation of the theory.

A.2.5. System state space. For the whole memory of the system and not only an isolated reference, the state vector can be defined as the Cartesian product of state vectors for each reference. If $\mathcal{I}$ is the set of all the references actually used by the task, the system state space is then:

$$\Upsilon = \bigtimes_{i \in \mathcal{I}} \mathcal{E}_i$$

For a Havard architecture it is useful to partition this space off between Υ_D the system state space for data and Υ_I the system state space for instructions, as long as auto-modifying code is forbidden as this is generally the case when such a cache architecture is used. Two independent space exist then, without interactions between them and this can save some amount of

[1] Least Recently Used replacement policy

computation:

$$\begin{cases} \Upsilon_D = \underset{i\in J_D}{\times} \mathcal{E}_i \ with\, J_D\, set\, of\, data\, references \\ \Upsilon_I = \underset{i\in J_I}{\times} \mathcal{E}_i \ with\, J_I\, set\, of\, instruction\, references \end{cases}$$

This can be generalized for any cache partition. In the following, as all partitions are independent, it is possible to concentrate on only one of them and its associated space without loosing generality. The space to be focused on shall be called Υ and $\mathcal{J}$ shall be the set of references associated.

A system probability space can also be defined, and shall be denoted υ_p:

$$\upsilon_p = \underset{i\in\mathcal{J}}{\times} \mathcal{P}_i$$

A.3. Markov chains based model

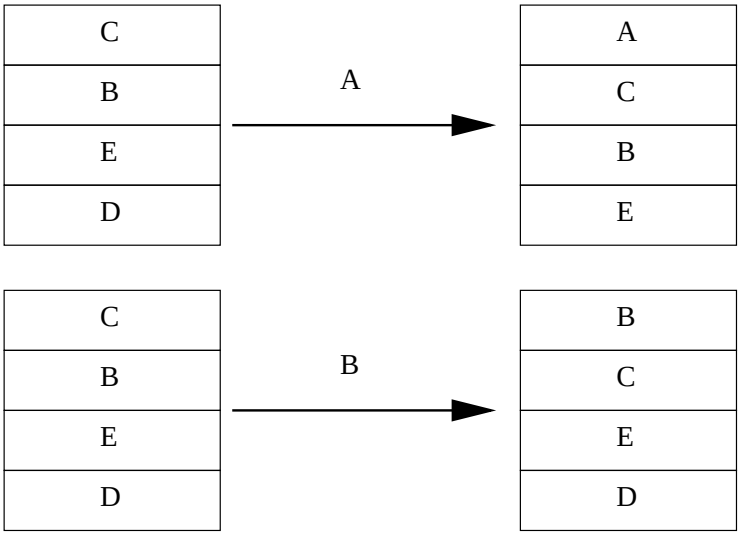
Markov chains are among the simplest Markovian processes[**23**][**17**]. Let s_t, $t \in \mathbb{N}$ be a discrete coordinate for system evolution (for example it can be some kind of discrete time seen as a succession of dates) and a set of discrete possible states for the system being studied:

$$\{y(n),\, n \in \{1, 2, \ldots, M\}\}$$

For a Markov chain, one can write the probability for the system to be in state $y(n)$ in s_{t+1} (and denoted $p(n, t+1)$) as a function of the probabilities of states for the system in s_t (denoted $p(m, t)$) :

$$\forall t,\ p(n, t+1) = \sum_{m=1}^{M} p(m, t) P_{1|1}(m, t \,|\, n, t+1)$$

FIGURE A.4.1. *accesses for a cache set and 4-way set associative cache*



where $P_{1|1}(m,t\,|\,n,t+1)$ can be seen as a conditional probability. A matrix $Q(t) \in \mathbb{R}^{M\times M}$ can also be used to synthetize the notation:

$$|p(t+1)\rangle = Q(t)\,|p(t)\rangle$$

Thus, a Markov chain translate a linear evolution of probability between two successive states. Now, let's see how to apply this formalism to the problem.

A.4. Operator for memory access

Access to a memory reference shall be modelized by an operator acting on Υ space (possibly Υ_D or Υ_I if it applies). Nonetheless, basic operators shall be first introduced acting only on reference state space $\mathcal{E}_i$. The effect of a memory access on a cache is illustrated in figure A.4.1 . For space $\mathcal{E}_i$ two different cases can be observed. The first one is access on the reference itself (A for first figure and B for the second), and associated operator shall be called elementary access operator and should put the reference at the top of its set for history and add an access and so update historical accesses and

misses number ; the second one is for references stored before the acceded reference in LRU history (all references in the cache for first figure and C for the second one) and the associated operator ; it shall be called concurrency operator, and shall update position in history, thus associated probabilities. There is no use at this point to distinguish between pre-load instructions sometime found on some RISC processors from real load for immediate use. This shall be distinguished only for WCET computation itself, and not for this miss computation.

A.4.1. Elementary access operator. Let's suppose that a new instruction is encountered that necessitate an access to memory reference i. For control flow part, it can be reaching a new block boundary in execution or following a branch instruction. For data flow part, it can be any load or store instruction for RISC architecture. The effect on the cache is: if the block is in the cache, it is put on top of its set for LRU history ; if it is not in the cache (miss) it is loaded form main memory then put on top of its set (as for the hit case). For the access number vector what is needed is only to add the probability vector to it. For the probability state vector the new p_1 must be set to 1 whereas all other must be set to 0. Thus the following set of equations:

$$\begin{cases} |n(t+1)\rangle = |n(t)\rangle + |p(t)\rangle \\ |p(t+1)\rangle = \top |p(t)\rangle \end{cases}, |n\rangle \in \mathcal{N}_i \text{ and } |p\rangle \in \mathcal{P}_i$$

with operator $\top = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$

This can be synthetized by the introduction of the operator $a^\dagger$ to write:

$$|s_t\rangle, |s_{t+1}\rangle \in \mathcal{E}_i,\ |s_{t+1}\rangle = a^\dagger |s_t\rangle$$

elementary access operator $a^\dagger$ being defined as follows:

$$a^\dagger = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ 0 & \top \end{pmatrix}$$

Whenever graphical notation shall be needed the following shall be used:

A left product by $a^\dagger$ is enough to modelize access to reference i on state vector.

A.4.2. Access concurrency operator. Let's consider, for an access on reference i, the effect for reference j that shares the same set in the cache but at a better place for LRU history. It means that the access shall alter its position in history as reference i shall take the first place. Thus if j was is first position, it shall go to the second; if it was in second, it shall go to the third, and so on, until the miss position is reached which is accumulative. It is easy to modelize that for probability state vector like this:

$$|p(t+1)\rangle = \lambda |p(t)\rangle \ \textit{with}\ \lambda = \begin{pmatrix} 0 & 0 & & \cdots & 0 \\ 1 & 0 & & & \vdots \\ 0 & 1 & \ddots & & \\ \vdots & & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 1 \end{pmatrix}$$

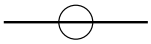
As no access is done on j, number of accesses shall remain constant. So an access concurrency operator b can be defined:

$$|s_t\rangle, |s_{t+1}\rangle \in \mathcal{E}_j,\ |s_{t+1}\rangle = b |s_t\rangle$$

with b being defined as:

$$b = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \lambda \end{pmatrix}$$

Whenever a graphic representation shall be needed the following shall be used:



For the elementary access operator, a left product by b is enought to modelize concurrency on j by i for the state vector.

A.4.3. Complete access operator and its approximation. Let's synthesize what have been seen on access. On the Υ space (or a subspace for any cache partition), the access operator on reference i has the following form:

$$\mathcal{A}_i = a_i^\dagger \underset{j\in\kappa_i^n}{\times} b_j \underset{k\notin\kappa_i^n, k\neq i}{\times} \mathbb{1}_k$$

where the notation $a_i^\dagger$ means it applies to the $\mathcal{E}_i$ space in Υ, b_j apply to $\mathcal{E}_j$ spaces and where κ_i^n is the set of reference of the same cache set as i that are before it in LRU history. It is obvious that $\kappa_i^n \subset K_i^n$ with:

$$K_i^n = \{ j \in \mathcal{J} / \; j \equiv i \, mod[n] \wedge j \neq i \}$$

and

$$n = \frac{size\,of\,cache\,in\,blocks}{A}$$

as K_i^n is the set of all references that *may* be on concurrency with i. Indeed, it is not possible, generally speaking, to find the κ_i^n set as it depends on dynamicly or externally set parameters, and execution history which is not unique, for statical analysis. That's why one is brought to use a pessimistic approximation of this set as close as possible to it but that does not depend

on any hazard of code execution. If this set is denoted Γ_i^n then obviously the relation $\kappa_i^n \subset \Gamma_i^n \subset K_i^n$ shall be true. Thus, an *effective* operator shall be applied for access modeling:

$$\mathcal{A}_i^* = a_i^\dagger \underset{j\in\Gamma_i^n}{\times} b_j \underset{k\notin\Gamma_i^n, k\neq i}{\times} 1_k$$

Computation of Γ_i^n set is a consequence of a pessimistic evaluation of program history of accesses. This kind of computation is widely seen in papers[**11**][**25**] and is generally called *may analysis*. It has already been seen in section 4.2.3. Using the same notations as then, the Γ_i^n set is the union of the collection determined by may analysis at each point in execution. In mathematical notation it can be written as follows:

$$\Gamma_i^n = \bigcup_{h=1}^{A} c_h(i[n])$$

A.5. Control flow hazards modeling

A.5.1. Control flow hazard. Control flow hazards are the expression of non determinism in the succession of instructions actually executed for the task. A simple example is *if (C)... then (T1)... else (T2)...* alternatives for conditional treatments. Treatment *T1* is done only if condition *(C)* is true. If not, then Treatment *T2* is done instead. This is a real control flow hazard if it is statically impossible to know which of both alternative will be executed (condition *(C)* depends on external elements).

For such a case, a mean to take into account this control flow indetermination in WCET computation is needed. It should be noted that the problem

is neither to find out what happens in the core execution followed by *T1* execution nor core execution followed by *T2* execution but from the fact that after the execution of conditional treatment one does not know which alternative have been executed. So the problem is the recombination of multiple concurrent execution paths.

A.5.2. Using a linear operation.

A.5.2.1. *General principle.* This is the simplest method. It consists in making the mean of state vectors coming from each possible execution path. The most obvious advantages of this method is its simplicity and quickness. For example if p_1 is one possible execution path and p_2 an other, if O_1 and O_2 are the respective associated operators and if $|s_1\rangle$ and $|s_2\rangle$ are the associated state vectors, then the resulting operator and state vector for path recombination operation are:

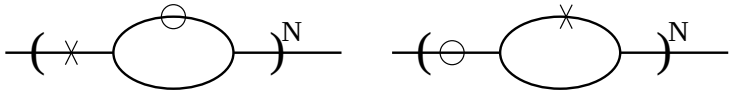
$$\begin{cases} O_{12} = \frac{1}{2}(O_1 + O_2) \\ |s_{12}\rangle = \frac{1}{2}(|s_1\rangle + |s_2\rangle) \end{cases}$$

The main drawback of this method is that it won't show the worst case, and it is a necessity to add a new parameter that gives a maximum deviation for the worst case. Let's show how it works on a simple example.

A.5.2.2. *Computation of a maximum deviation from mean case.* For a given state vector, it is always possible to get a cache miss unless $p_\emptyset = 0$ which is a deterministic case. So, apart from this late case, the distance from $p_\emptyset$ to 1 it the maximum deviation from number of cache misses for this access. Summing these deviations for each access enables to compute a global maximum deviation.

Let's try this on a simple case. So let's suppose the little program showed on figure A.5.1 for a non associative cache ($A = 1$). It is straightfor-

FIGURE A.5.1. *N iterations loop with an access in loop main core and some concurrent access in conditional alternative*



ward to find out that for each iteration i of the loop, one has for first diagram the following operator:

$$A = [\frac{1}{2}(\mathbb{1}+b)\,a^{\dagger}]^{i} = \frac{1}{2}\begin{pmatrix} 2 & 0 & i+1 & i-1 \\ 0 & 2 & i-1 & i+1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

and for second diagram:

$$A' = [\frac{1}{2}(\mathbb{1}+a^{\dagger})b]^{i} = \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & i & i \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Supposing that initial state for the cache is always miss (state vectors are all $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$), the following state vectors can be found:

- State vector for first access in the loop and first diagram:

$$|s_{1\times}\rangle = \frac{1}{2}\begin{pmatrix} i-1 \\ i+1 \\ 1 \\ 1 \end{pmatrix}$$

the maximum deviation shall be $\frac{1}{2}$ apart from first case, so:

$$\Delta_i^1 = \frac{i-1}{2}$$

- State vector for first access in the loop for second diagram:

$$|s_{2l}\rangle = \frac{1}{2}\begin{pmatrix} 0 \\ i \\ 0 \\ 2 \end{pmatrix}$$

- State vector after alternative for second diagram:

$$|s_{2i}\rangle = \frac{1}{2}\begin{pmatrix} 0 \\ i+2 \\ 2 \\ 0 \end{pmatrix}$$

recombination of those two vectors gives:

$$\Delta_i^2 = \frac{i}{2}$$

the same way.

For maximum deviation, summing up all these, give at last for $i = N$:

$$\begin{cases} m_1 &= \frac{N+1}{2} \pm \frac{N-1}{2} \\ m_2 &= \frac{N}{2} \pm \frac{N}{2} \end{cases}$$

which is the exact result on this simple example.

A.5.3. Using a non linear operation that gives the worst case.

A.5.3.1. *General principle.* Instead of using a side parameter to compute a worst case deviation, the idea is a direct computation of a pessimistic

or worst case recombinated vector or operator after path recombination. For state vector, this leads only to choose among all possible vectors, for each reference, the one that maximize either miss number ($n_\emptyset$) or if miss numbers are equal the one that maximize miss probability ($p_\emptyset$) for next access. For evolution operators, a new one have to be forged using the worst case columns of each operators, and using the same rules as for state vectors. This operation is not linear but does not need any other parameter to give the worst case. This is also closer to what is done in other abstract interpretation methods. The only drawback is, indeed, a loose of the linearity but is not vital. So, this will be the operation that should be actually applied for path recombinations. It should be noted that unless some external strong hypotheses are known on the task comportment, this will lead a priori to combine worst case from incompatible paths. Let's show how it works on the simple example formerly introduced.

A.5.3.2. *A simple example.* Once again, let's use the example program from figure A.5.1. Using above rules, it is easy to find out operator $A_{nl} = (ba^\dagger)^N$ for first diagram, that maximize the number of misses. Supposing invalid cache as the task starts $|s_i\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$, the number of misses can be deduced for first reference:

$$m_1 = \langle m | (ba^\dagger)^N | s_{nl} \rangle = N$$

with

$$\langle m | = \begin{pmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} = \left(\delta_{A+1,j}\right)$$

the cache misses projector.

For the second diagram, the associated operator shall be $A' = (a^\dagger b)^N$ that maximize miss probability. Thus the number of misses:

$$m_2 = \langle m | (a^\dagger b)^N | s_{nl} \rangle = N$$

Once again the exact numbers are found which was expected for this simple example.

A.6. Taking care of data flow hazards

A.6.1. Data flow hazards. For some kind of data accesses, it is impossible to tell *a priori* where the access shall take place exactly. Some typical cases are buffers and look-up tables. What can only be known about these accesses is an actual zone where accesses may be. Exact location depend on external parameters only known at run time.

A.6.2. Taking care of such hazards. At present, few work has been done in this scope. The goal being a worst case computation, a really pessimistic operation is actually applied, applying concurrency operators to all possibly relevant spaces. Thus, the operator used for an data flow hazard access on the memory location $\mathbb{A}$ is:

$$M_i^* = \underset{j \in \Gamma_{\mathbb{A}}^n}{\times} b_j \underset{k \notin \Gamma_{\mathbb{A}}^n}{\times} 1_k$$

where $\Gamma_{\mathbb{A}}^n$ stands for $\bigcup_{i \in \mathbb{A}} \Gamma_i^n$. It should be noticed that this operation is probably overkill and some substantial improvement could be found.

All the operations we introduced in this chapter allow to take into account any comportment of a task execution, apart from random time preemption. So, most of the bases of the theory are set. Some further consideration can be made on operator, nonetheless.

A.7. Using operators

A.7.1. Definition: complete execution path. We call complete execution path any succession of instruction whose associated control flow is such as if the first instruction is executed then the last instruction of the path shall be executed.

A.7.2. General evolution operator. It has been seen in this chapter how to compute state vectors from one point in symbolic execution to the next one, taking into account all can possibly happens that may have some incidence on memory state. For any branch of execution, a state vector $|s_0\rangle$ can be computed before the branch execution. At each point t, one of these can happen:

- deterministic memory access: the associated operator shall be $A^*_{t+1,i}$ such as $|s_{t+1}\rangle = A^*_{t+1,i}|s_t\rangle$
- *for data accesses only:* access with data flow hazards: an operator $M^*_{t+1,t}$ can be found as has been seen in section A.6.2, so that $|s_{t+1}\rangle = M^*_{t+1,t}|s_t\rangle$. Using the former property, a straightforward recurrence shows that for any pure sequence of instructions an operator $O_{k0} = O_{k,k-1}\cdots O_{2,1}O_{1,0}$ can be found such as $|s_k\rangle = O_{k0}|s_0\rangle$

- *path recombination for data flow*: let $\left|s_f\right\rangle$ be the state vector before conditional treatment. If both path are pure sequence of instructions the previous result enables to write the vector space after recombination as a function of two state vectors $|s_e\rangle = O_{ef}\left|s_f\right\rangle$ and $|s'_e\rangle = O'_{ef}\left|s_f\right\rangle$, so there exist an operator O^*_{ef} such as vector state after recombination can be written $O^*_{ef}\left|s_f\right\rangle$. The actual mean to compute this operator has been discussed in section A.5. If paths are now pure sequence of instructions then a simple recurrence on this result allow to generalize it (since single instructions are a particular case of pure sequence of instruction) to any complete execution path.

Thus, the following theorem.

THEORÈME A.7.1. *For any complete execution path $\mathcal{P}$, one can find an operator $O_{\mathcal{P}} \in \mathcal{L}(\Upsilon)$ such as for any initial state vector $\left|s_i^{\mathcal{P}}\right\rangle \in \Upsilon$ before path execution, the final state $\left|s_f^{\mathcal{P}}\right\rangle \in \Upsilon$ after execution can be written:*

$$\left|s_f^{\mathcal{P}}\right\rangle = O_{\mathcal{P}}\left|s_i^{\mathcal{P}}\right\rangle$$

A similar result can be found in the probability space υ_p.

THEORÈME A.7.2. *For any complete execution path $\mathcal{P}$, one can find an operator $Q_{\mathcal{P}} \in \mathcal{L}(\upsilon_p)$ such as for any initial state vector $\left|p_i^{\mathcal{P}}\right\rangle \in \upsilon_p$ before path execution, the final state $\left|p_f^{\mathcal{P}}\right\rangle \in \upsilon_p$ after execution can be written:*

$$\left|p_f^{\mathcal{P}}\right\rangle = Q_{\mathcal{P}}\left|p_i^{\mathcal{P}}\right\rangle$$

This shows very clearly the Markov chain model underlying in the theory.

A.7.3. Combination properties. The previous theorems allows to determine general operators associated to any complete execution path. So, it is possible to combine these operators, once computed. For example, for two complete execution path $\mathcal{P}_1$ and $\mathcal{P}_2$ and their respective associated operators O_1 and O_2, the operator associated to the sequential execution of $\mathcal{P}_1$ and $\mathcal{P}_2$ is simply:

$$O_{12} = O_2 O_1$$

It should be noticed, nonetheless, that this kind of computation does not allow to find the less pessimistic operator since the computation of O_2 is generally done without any hypothesis on previously executed instructions. Thus, all history known in O_1 execution is lost and that should lead to over pessimistic operator. Anyway if the number of combination done like this is small before the number of accesses per reference, it won't be noticeable.

The same kind of things can be done for concurrent execution of paths (control flow hazard). Then the combined operator will be computed as has been discussed in section A.5. Result should not be over pessimistic here.

Both results are also valid for probability space and associated operators.

A.8. Stationary state

Let's consider the probability space υ_p and the probability operator Q_T associated with task T execution. Indeed, a real-time task is to be repeated over time and without limitation. For such a system, the repeated execution of task T is a classical time invariant Markov chain model. It is known that such a system shall find an equilibrium state and that the associated probability state vector is the unique eigen vector of Q_T associated with

eigen value 1 (exists and is unique [**23**]). Mathematically speaking:

$$\exists! \, |p_e\rangle \in \upsilon_p \, / \, |p_e\rangle = Q_T \, |p_e\rangle$$

Vector $|p_e\rangle$ is the equilibrium worst case state vector for everlooping task T.

It is easy to find the equilibrium number of cache misses as:

$$m_T = \langle m| \, O_T(|0\rangle \otimes |p_e\rangle)$$

where O_T is the global operator (in $\mathcal{L}(\Upsilon)$) associated to task T.

As can be seen, this method enables to do not any hypothesis on cache initial state. Thus, this result enables our theory to have better accuracy of computation than other abstract interpretation methods.

A.9. Summary of sequential case

The theorical scope having been explain in this chapter for the sequential case shows more original approach and results than other WCET methods:

- the abstract space having the interest it not directly bound to cache but to memory state regarding the cache
- model relies on a probability space very different than collection of sets generally met in other theories
- model still relies on abstract interpretation but uses a Markovian model to update abstract states.

This causes longer computation than other abstract interpretation model. One the other hand, no hypothesis is needed on initial states. Moreover, the use of a probability space enable to widen the scope of the theory in the multitask field.

CHAPTER B

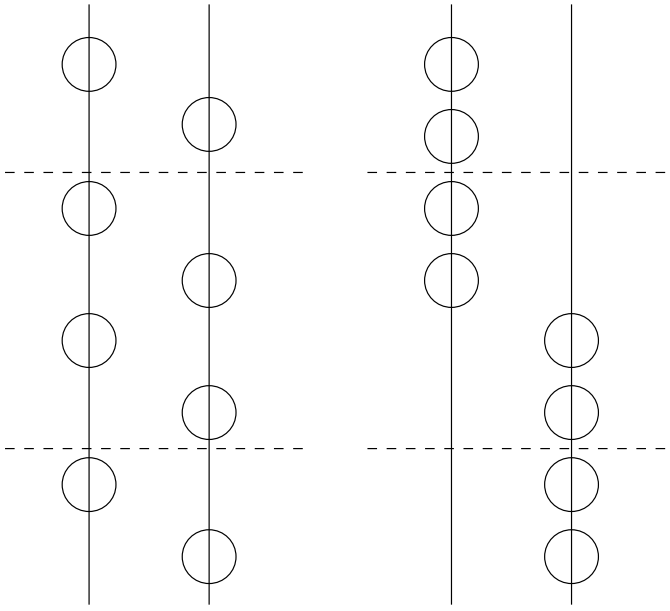
Multitask generalization

B.1. Issues

In section 5.2.2, the difficulty to find a model for multitasking WCET computation has been explain. Let's evaluate the difficulty for a straightforward generalization of a WCET computation method, with a quick computation of the associated entropy.

Let's consider the succession of hit access of two distinct references as the task is executed. in figure B.1.1 the four hit accesses has been represented in two opposite cases, one with strong correlation and one with weak correlation. The dotted line represent two worst case preemption points for the system. In the first case (strong correlation), $2 \times 2 = 4$ cache misses

FIGURE B.1.1. *Two cases of opposed correlation for two reference accesses*



shall be added (2 misses per preemption). In the second case (weak correlation) $1 \times 2 + 1 = 3$ cache misses shall be added (1 per preemption and 1 due to interface effect). As the number of accesses and preemptions grows if n_{max} is the maximum number of preemptions and N the number of references for the task, the number of added misses shall be $m_{sup} = N \times n_{max}$ for strong correlation and $m_{sup} = n_{max} + N - 1$ at worst for weakly correlated case. The main difficulty to find out the number of cache misses to add is then to compute an accurate correlation.

Let's suppose that the correlation m is for n references. Determining this need to test combinations of m among n so $\begin{pmatrix} n \\ m \end{pmatrix}$ possibilities. As m is not known, one has to make it vary between 1 and n to compute the actual correlation. So the number of combination to test is:

$$\sum_{m=1}^{n} \begin{pmatrix} n \\ m \end{pmatrix} = 2^n$$

So for N references and using $k = \frac{1}{\ln 2}$ for normalization it is straightforward to find the entropy:

$$S = N$$

Thus computation cost of correlation is exponential for such a system with the number of references (unless there is a great regularity of accesses). The conclusion is that is useless to use such a method to compute exact WCET.

Now, if an exact result is not needed neither as exact preemption points, the probabilistic aspect of the theory to compute upper bound of WCET in the multitask field.

B.2. General idea

So the general idea is to use the probabilistic properties of Markov models. In the monotask field all computations were deterministic, and now let's allow real probabilistic computation by an alteration of the propagator. What we want to do is to alter operators $a^\dagger$ and b so that they take into account the fact that a preemption may occur. Thus, replacing $a^\dagger$ by $a^\dagger\Pi$ and b by Πb, without changing anything to the previously exposed theory, we try to obtain a realistic estimation of the number of cache misses for the task as it is preempted by others. Then, to find a real worst case, a standard or worst case deviation has to be computed for that purpose.

Here are the notations that shall be used:

- n_{max}**:** maximum number of preemptions expected as the task is executed
- N**:** minimal number of instruction executed by the task (or minimum value of any advancement parameter for task execution measurement when the task is finished)
- $\mathcal{A}$**:** number of access to the studied reference as the task is executed
- p**:** probability of preemption.

n_{max}, N, and $\mathcal{A}$ shall be known parameters ; without them no computation can be done. p on the other hand shall be a result of the model.

What is being looked for here are preemption operators Π. When a Δs progression is made in task execution, we can write Π as $\pi^{\Delta s}$, for uniform preemption probability. If probability is not constant over time, it is still possible to write preemption operator as a product of basic preemption operators $\Pi = \prod_i \pi_i$ where the product is a left product. Estimation done and standard or maximum deviation computed should vary with these kind of hypotheses, but the problem is easy to solve.

B.3. Preemption operator

Let's choose s the coordinate for advancement in program execution such as preemption probability is a constant when s changes. This is not a necessary hypothesis but ease notation for calculation. One can write:

$$\pi = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph \end{pmatrix} \; with \; \aleph = \begin{pmatrix} 1-p & & & 0 \\ & \ddots & & \vdots \\ 0 & & 1-p & 0 \\ p & \cdots & p & 1 \end{pmatrix}$$

or write:

$$\aleph = (1-p) + p\perp \; with \; \perp = \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ 1 & \cdots & 1 \end{pmatrix}$$

Meaning of such an operator is simply that there is probability p that reference is drop out of the cache due to preemption and probability $1-p$ that nothing happens for elementary variation of s. A straightforward calculation brings the following result:

$$\pi^m = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph^m \end{pmatrix} \; and \; \aleph^m = (1-p)^m + [1-(1-p)^m]\perp$$

what enables to write the propagator for any Δs. It can easily be generalized to cases where preemption is non sufficient to reject the reference out of cache, but only makes it drop i places in LRU history (the general action on all references of the task is called *preemption footprint*). Then one has the

following results:

$$\pi_i = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i \end{pmatrix} \; with \; \aleph_i = (1-p) + p\lambda^i$$

where λ is the concurrency operator as defined in section A.4.2 and

$$\pi_i^m = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i^m \end{pmatrix} \; with \; \aleph_i^m = \left[(1-p) + p\lambda^i\right]^m$$

The expression of p as a function of known parameters is also needed. So, let's consider some imaginary program whose only action is to access a given reference. Once the initial stage done, this program won't generate any cache miss since all its needed references are in the cache, unless some preemption happens. The only source of cache misses being preemption, calculation of the number of misses using π operator and comparing it to n_{max} should be enough to express probability p. Let's start calculation:

Let's define operator ξ:

$$\xi = \pi a^\dagger = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ 0 & \tau_p \end{pmatrix} \; with \; \tau_p = (1-p)\top + p\bot$$

for simplicity we use $\bot$ rather than λ^i (limit case since $\forall \nu \geq A$, $\lambda^\nu = \bot$). Execution of program shall be modelized by powers of operator ξ. τ_p being idempotent, it is straightforward to find:

$$\xi^k = \begin{pmatrix} \mathbb{1} & \mathbb{1} + k\tau_p \\ 0 & \tau_p \end{pmatrix}$$

so for $k = N$ that stand for whole task execution, n_{max} cache misses should be reached. Thus:

$$Np = n_{max} \;\Leftrightarrow\; p = \frac{n_{max}}{N}$$

If accesses are less concentrated, and $\Delta s \neq 1$ so $\xi_m = \pi^m a^\dagger$ is the basic operator for task execution, calculations turn out to be similar by a simple replacement of $1-p$ by $(1-p)^m$ and p by $[1-(1-p)^m]$. And finally:

$$p = 1 - \left(1 - \frac{n_{max}}{N}\right)^{\frac{1}{m}}$$

for uniform accesses, one has $m = \frac{N}{A}$ and the associated preemption probability. Nonetheless, the first expression of probability is sufficient since it is a worst case.

THEORÈME B.3.1. *Operator π_i is the basic preemption operator, or propagator for progression in program execution and for a chosen reference when preemption is allowed. It can be expressed as*

$$\pi_i = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & \aleph_i \end{pmatrix} \; with \; \aleph_i = (1-p) + p\lambda^i$$

where i is the depth of memory footprint for preempting task for the given reference and where probability of preemption p has the following expression:

$$p = \frac{n_{max}}{N}$$

B.4. Standard deviation

Calculation shall be in two parts. At first, the case of a unique reference shall be discussed, then the case of multiple reference for the whole task. The coherence of accesses should be taken into account for the later case.

B.4.1. A unique reference. Let's proceed by analogy with the former p calculation. A Δs progression in program execution has been made. The

only source of cache rejection of reference is supposed to be due to preemptions (or else it is worthless to take this access into account for calculation). Miss probability is anoted $p_{\Delta s}$.

In fact $\Delta s \times p$ cache misses for the reference should be seen but only $p_{\Delta s}$ are counted. So the standard deviation can be deduced if $n_{max} \geq \mathcal{A}$:

$$\sigma^2 = < m^2 > - < m >^2 = n_{max}^2 - < \sum_{i \in (access to ref)} p >^2$$

If a straightforward generalization is done from that, the following result appears:

$$\sigma^2 = n^2 \times n_{max}^2 - < \sum_{j \in \mathcal{J}} p_{i,j} >^2$$

but this expression is an artificial serialization of accesses so is an strong over-estimation of standard deviation (it means that accesses are strongly correlated). The opposite hypothesis can be made also, and for weakly correlated accesses:

$$\sigma^2 = n \times n_{max}^2 - \sum_{j \in \mathcal{J}} < \sum_i p >^2$$

which under-estimate the exact number. So some other way must be found for the general case.

B.4.2. Multiple references. For a better evaluation it is a necessity to take coherence of accesses into account. Intuition tells that for uncorrelated accesses, standard deviation should be proportional to the square root of number of accesses, whereas it should be proportional for to the number of accesses for strongly correlated accesses.

For a given reference, each access is followed by an exponential decrease of hit probability, or equivalently an evanescent propagation of this along task execution. Let's use that for an evaluation of correlation.

Immediately after an access, the hit probability is 1 for acceded reference. If all hit probabilities at this time are summed up, a upper bound of the number of extra possible cache misses is obtained. So if t_a is the time where a reference is acceded:

$$\delta m(t_a) = \sum_{i \in \mathcal{J}} \left(1 - p_{\emptyset,i}(t_a)\right)$$

is this number. Choosing $a_1, a_2, \ldots a_{n_{max}}$ that maximize $\delta m(t_a)$ a maximum deviation can be computed:

$$\Delta m = \sum_{j \in \{1, \cdots n_{max}\}} \delta m(t_{a_j})$$

and also a n upper bound for standard deviation:

$$\sigma^{*2} = \sum_{j \in \{1, \cdots n_{max}\}} \delta m^2(t_{a_j})$$

It can be noted then that $\sigma^* \sqrt{n_{max}}$ is a correct evaluation of maximum deviation. So an upper bound of cache misses number in the worst case can be chosen as $\bar{m}(n_{max}) + \sigma^* \sqrt{n_{max}}$ before deducing WCET.

CHAPTER C

From number of cache misses for the worst case to WCET

Two cases can be discussed. For older processors, cache misses shall stall the pipeline but not for newer processor that keep running until dependencies on cache misses are no longer postponable.

C.1. Blocking caches

This case is very simple since time associated to any cache miss is roughly constant. Using the maximum delay for main memory access and multiplying by the number of misses is enough to compute the cache related WCET. A basic simulator of processor main core is still needed for pipeline related time computation and so having a complete WCET.

C.2. Non blocking caches

This case need a quite accurate simulator for processor main core. The exact delay that a cache miss can trigger off depends on the subtile game of data dependencies in the pipeline. So an accurate WCET computation also need an accurate processor main core simulator. Since it is also needed for pipeline related time computation, any WCET cannot live without such a tool. That is true for any WCET approach.

Anyway, this task itself is out of the scope of a thesis work. That's why for model evaluation, we had to test only simple programs since all modeling and time computation shall be done by hand and real world cases are out of the scope of a human work. Let's see that at present.

CHAPTER D

Results of the model

D.1. Benches used

For evaluation of the model, some simple programs have been tested. They had to remain simple since they were hand-modeled from compilation output of GCC compiler (version 2.95.3 with "-O3" optimization flag-we showed how to proceed on an example in apendix 1). The test programs are:

- matrix product of two 50×50 matrices
- sum of two 38×38 matrices
- Fast Fourier Transform (FFT) for 512 points sample
- vector normalization with 6000 coordinates
- bubble sort of 4200 numbers (so that control flow hazards can be tested).

As formerly said for a real case evaluation tools should be created.

D.2. Results

In table 1, a comparison of simulation and the model had been made.

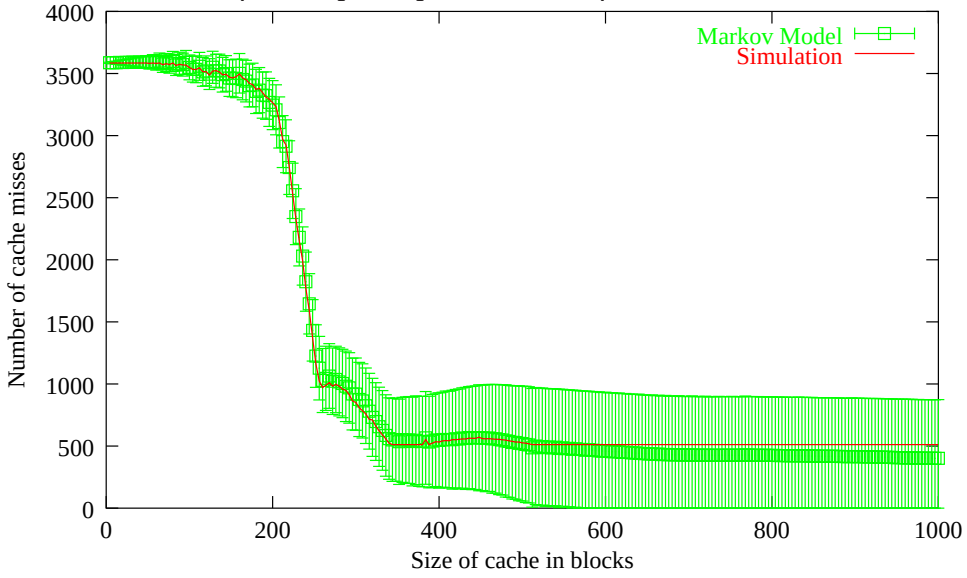
Results are good, knowing that present actual modeling makes very systematically pessimistic choices. On pure sequential code (without preemption) results are even very good as could be expected. It should be noted that in that case, σ^* being 0, there is no need to show it in the table.

For preemption case, although the table only shows up to 3 preemptions, it has been tested for higher numbers without noticeable differences

TABLE 1. *Comparison of cache misses for simulation m_{max} with results from the model ($\bar{m}$ and also σ^* when it applies).*

number of preemptions		0		1			2			3		
		m_{max}	$\bar{m}$	m_{max}	$\bar{m}$	σ^*	m_{max}	$\bar{m}$	σ^*	m_{max}	$\bar{m}$	σ^*
Matrix	4Ko	**32565**	32565	**32629**	32606	129	**32656**	32647	182	**32706**	32688	223
product 50×50	16Ko	**1250**	1250	**1889**	1875	970	**1941**	2490	1309	**2016**	3094	1538
Matrix	4Ko	**1083**	1083	**1086**	1106	125	**1089**	1129	171	**1092**	1151	203
sum 38×38	16Ko	**295**	295	**1084**	750	804	**1089**	1063	929	**1092**	1283	963
FFT	4Ko	**3520**	3520	**3523**	3550	129	**3526**	3579	182	**3529**	3608	223
512 points	16Ko	**0**	0	**512**	402	473	**768**	692	622	**1024**	927	715
Vector	4Ko	**2744**	2808	**2872**	2844	127	**3000**	2880	177	**3003**	2914	214
Normalization 6000pts	16Ko	**952**	1464	**1976**	1949	913	**3000**	2339	1160	**3003**	2662	1286
Bubble sort	4Ko	**2166140**	2166200	**2166272**	2166240	129	**2166400**	2166280	182	**2166528**	2166320	223
4200 numbers	16Ko	**7020**	7532	**8048**	8111	1025	**8968**	8679	1449	**9884**	9236	1775

FIGURE D.2.1. *Variation of number of cache misses with cache size for one preemption allowed for FFT*



from what can be already seen. Generally speaking observed derivations remains well under $\sigma^* \sqrt{n_{max}}$ but this evaluation still remains quite accurate in magnitude.

Variation with cache size can be seen in figure D.2.1. It can be seen that, for such a cache demanding program, results from simulation remains very close to the bare evaluation from the model and the maximum deviation is quite an overestimation. Nonetheless, for less cache demanding programs (like matrix summation), this result would have been less obvious, even though due to deviation expression itself, simulation would have remained within error bars of the maximum deviation.

D.3. WCET on the benches

D.3.1. Hypotheses for time computation. WCET have been simulated for PowerPC type superscalar processor. We supposed 2 integer units, one load/store unit, one float unit and one branch unit. Loading and ending of instructions are in order but all other intermediate operations are without

orders. To help simulation we supposed always enough renaming register for avoiding pipeline stalls due to that kind of reasons ; branch wrong prediction has also been taken as the mean when real control flow hazards exist.

Cache are non blocking, but some hypotheses have been simplified in order to make easier computation, since they are hand modeled. Main memory accesses are supposed to be 8 cycles delay. All these hypothesis are quite accurate for usually met real-time systems.

D.3.2. Results. WCET computation is made using $m_{maj} = \bar{m}(n_{max}) + \sqrt{n_{max}}\sigma^*$ evaluation for cache misses. Exact WCET is obtained by simulation and Δt is the overestimation made by the model. This overestimation is also given as a percentage in order to be able to compare the model results with paper results of other methods. This is shown is table 2.

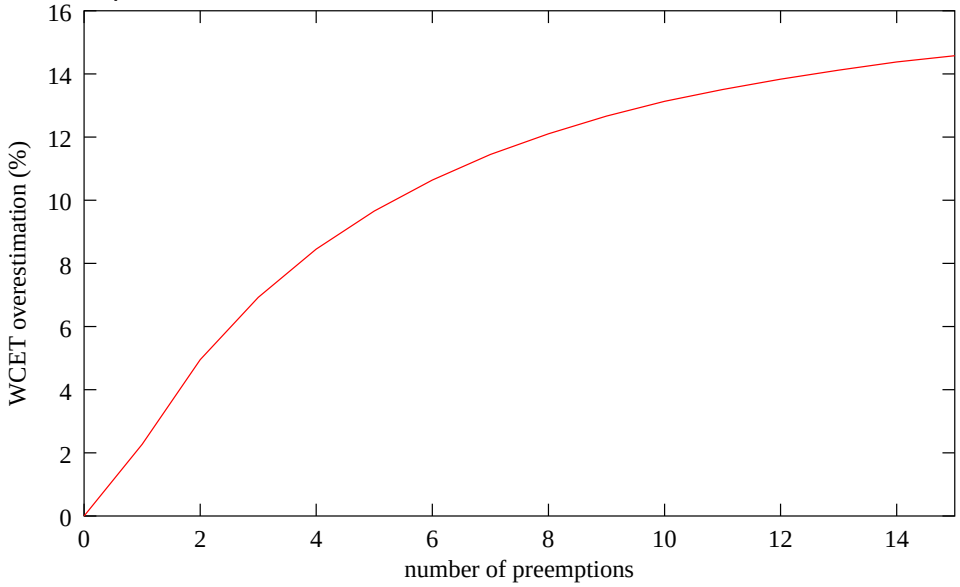
As can be seen, results are very good. They are the same order as other methods for the case without preemption and completely unmet for the case with preemption.

D.3.3. Variation with number of preemptions. This has been evaluated for the FFT case, for preemptions varying from 0 to 15 (beyond that only only the spacial coherence of cache accesses remains for cache hits). Results are shown in figure D.3.1 where we plotted overestimation as a function of number of preemptions, in percentage. As can be seen, overestimation remains below 15% even for a high number of preemptions, what is a very good result since this example is the worst we have that also make a real reuse of data. If invalid caches had been supposed, overestimation could have been as large as 75%(in particular for a few number of preemption). The benefice is high in computing power thanks to cache use.

TABLE 2. *WCET in number of cycles, and overestimation in cycle and percentage*

number of preemptions		0			1			2			3		
		WCET	Δt	%	WCET	Δt	%	WCET	Δt	%	WCET	Δt	%
Matrix	4Ko	**827316**	0	0,00	**825769**	625	0,08	**827852**	1465	0,18	**828147**	2173	0,26
product 50×50	16Ko	**641307**	0	0,00	**645572**	5258	0,81	**646634**	14161	2,19	**647076**	22077	3,41
Matrix	4Ko	**13486**	0	0,00	**13507**	280	2,07	**13528**	287	2,12	**13549**	476	3,51
sum 38×38	16Ko	**7085**	0	0,00	**13493**	812	6,02	**13528**	1918	14,2	**13549**	2296	16,9
FFT	4Ko	**142354**	0	0,00	**142377**	1170	0,82	**142400**	2328	1,63	**142422**	3489	2,45
512 points	16Ko	**115954**	0	0,00	**119794**	2723	2,27	**121714**	6027	4,95	**123634**	8561	6,92
Vector	4Ko	**92010**	186	0,20	**92381**	287	0,31	**92752**	378	0,41	**92761**	620	0,67
Normalization 6000pts	16Ko	**86812**	1485	1,71	**89782**	2569	2,86	**92752**	2840	3,06	**92761**	4285	4,62
Bubble sort	4Ko	**189434652**	360	0,00	**189435444**	582	0,00	**189436212**	824	0,00	**189436980**	1069	0,00
4200 numbers	16Ko	**176479932**	3072	0,00	**176486100**	6528	0,00	**176491620**	10561	0,00	**176497116**	14558	0,00

FIGURE D.3.1. *Variation* of WCET *overestimation as a function of number of preemptions for FFT, and data cache of 16KB.*



D.4. Computing cost of the method

This model is based on operator algebra. For modern processors, associativity A for the cache is typically 2, 4 or 8 for first level. Associated operators are then matrix of $\mathbb{R}^{6\times 6}$, $\mathbb{R}^{10\times 10}$ or $\mathbb{R}^{18\times 18}$. These matrix having lots of zeros, the actual complexity is $O\left((A+1)^2\right)$ for operators computation and $O(A+1)$ for state vectors. The choice enables to favor either efficiency or quickness of computation. If $\mathcal{S}$ is the total maximum number of accesses for the task and $l = Card\, \mathcal{J}$, the total number of operation is $\mathcal{S}\frac{l}{n}$ which can be quite high. With the current calculation library, it is rare to run over 10 minutes for a hundred of million of operation on rather old workstations. With vector units commonly found in todays processors and specially written and optimized library, several billions accesses seems reachable within hour computation. Anyway if one wants quick computation accent should be put on state vectors computation, much lighter, even

though is is less precise. An other way of doing such computation is to parallelize it, which should be rather efficient.

D.5. Comparison with other methods

For recent papers, and monotasking computation, we can cite most essentially [**11, 19, 26, 25, 20**]. Results obtained on similar bench as ours are very close. This was expected since this is also an abstract interpretation method and so the same level of precision can be achieved, mostly using the same techniques. Precision is only limited by intrinsic limitation of static analysis and so determination of incompatible execution path or some run time parameters. One noticeable difference, nonetheless is that the hypothesis of invalid cache at the beginning of the task can be avoided. So this is better suited for ever looping task as this is the rule in real-time systems.

The result are, however, unmet for multitasking systems. Hypotheses on preemptions are minimal and the fact that the same theory enables to take into account both the monotask part and the multitask part of the cache related WCET is totally new. As has been seen the results in the multitasking extension of the theory are typically a few percents, and the fact that no underestimation can occur is very promising. Nonetheless, the model should need a test on a real case for better evaluation. That is where future work will lead to.

D.6. Several cache levels

For a memory architecture with multiple levels of caches, there exist several policies for cache update. The two dominant methods are called "*write through*" and "*copy (or write) back*". For *write through* policy all levels in memory hierarchy are updated for each write access. This policy is

deterministic and allows the model to perfectly take into account that, with an independent model for each cache level. On the other hand, write back policy try to postpone writing to higher level in memory as long as possible. The main consequences is that write accesses are no longer deterministic. They depend on program history and is hard to predict statically. If ones want to take into account *write back* policy for multiple levels of cache an important work is needed on the present theory.

D.7. About SMP

Symmetric Multi Processing or SMP is a multiple processor system (all main processors being the same and equivalent) with a common memory and a common bus. Caches on such systems need complex synchronizations not taken into account by the present model. Since the shared variables should be as few as possible for security concerns in real-time systems, most of the time forbidding caching of such variables would not cause major performance problems. In this case, the present model is, then, good enough to do the job.

CHAPTER E

Conclusions and possible extensions

Summary

The goal of the thesis work was to open a new approach for treatments of cache related execution time hazards for hard real-time task. We exposed the context of the issues and particularly the importance of cache memory on execution time determinism, especially for multi-tasking systems. State of the art in recent paper brought the concept of abstract interpretation, but also limits for multi-tasking modeling for theses methods until now.

Despite an abstract interpretation based model, this work open a new way to compute cache related WCET. Indeed, at the opposite of other models, this work use a probabilistic base through the use of Markov chains in the theory. Still, in monotask problems, the model is perfectly deterministic and directly conduct to WCET computation. Thus results, in this scopes, are close to other abstract interpretation models, but enables to go a little further since no hypotheses on cache states have to be made. Computations are heavier, nonetheless.

For multitasking systems, a smart use of the probabilistic aspect of the model is used in order to extrapolate a result also for this case. A simple alteration of de default propagator is enough for that to happen. It could be compared to some wave propagation in a dispersive medium, monotask being then propagation without dispersion. The result need nonetheless to compute a maximum deviation from that.

Evaluation done on simple programs shows promising results. However, some work has to be done in order to create a complete tool for WCET computation and test it on real world systems.

Possible extensions

This model can be generalized without much difficulties to any Cartesian product of automatons, and especially if they are referenced by addresses. Thus, it would be possible, for example to use this formalism to compute branch prediction related delays, for simple branch predictors like one bit and two bits predictors like has been seen in section 3.3. So useful generalization of the formalism can be done with great interests. Indeed, this is a brand new field of abstract interpretation models that had been opened here.

Bibliographie

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS'96 Static Analysis Symposium*, pages 52–66, 1996.

[2] Christophe Aussaguès. *Placement Optimal de tâches pour les systèmes Parallèles Temps-Réel Critiques*. PhD thesis, 1998.

[3] O.I. Aven, E.G. Coffman, and Y.A. Kogan. *Stochastic Analysis of computer storage*. Reidel Amsterdam, 1987.

[4] J.V. Busquets-Mataix, J.J. Serrano-Martin, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceeding of the 2nd Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[5] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] Vincent David, Marc Aji, Jean Delcoigne, Christophe Aussaguès, and Chritophe Cordonnier. Le modèle de conception oasis/ψc pour les systèmes temps-réel complexes critiques. *Real-Time & Embedded Systems*, 1996.

[7] Vincent David, Christophe Aussaguès, and Jean Delcoigne. Seeking a deterministic multitask framework for safety critical systems : the oasis approach. In *ANS/ENS, International Topical Meeting on Nuclear Plant Instrumentation*, 2000.

[8] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustavson, and H. Hansson. Towards industry strenght worst-case execution time analysis. Technical report, ASTEC, February 1999.

[9] C. Ferdinand. Cache behavior prediction for real–time systems. Technical report, Universität des Saarlandes, september 1997.

[10] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN Worshop on Language, Compiler and Tool support for Real–Time Systems*, pages 37–46, 1997.

[11] Christian Ferdinand and Reinhard Whilhelm. Efficient and precise cache behaviour prediction for real–time systems. *Real–Time Systems*, 17 :131–181, nov 1999.

[12] Christine Fricker and Philippe Robert. On the representation of memory references generated by a program with application to the analysis of cache memories. Technical report, february 1990.

[13] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behaviour. Research Report CS-RR-349, October 1998.

[14] J.L.A. Hennessy and D.A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman, 2nd edition, 1996.

[15] H. Kopetz. The time-triggered approach to real-time system design in randel b, 1995.

[16] C.-G. Lee, J. Hahn, Y.-M. Seo, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. Technical Report Vol 47, IEEE Transaction on Computer, 1998.

[17] S. Lipschutz. *Probability*. McGraw-Hill, 1965.

[18] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real–time environment. *Journal of the ACM*, 20 :46–61, 1973.

[19] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle–level symbolic execution. *Real–Time Systems*, 17 :183–207, nov 1999.

[20] Franck Mueller. Timing analysis for instruction caches. *Real–Time Systems*, 18 :217–247, may 2000.

[21] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2) :193–204, 1999.

[22] G.S. Rao. Performance analysis of cache memories. *Journal Assoc. Comput*, pages 378–395, 1978.

[23] L.E. Reichl. *A modern course in Statistical Physics*. Wiley–Interscience Publication, 2nd edition, 1998.

[24] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1994.

[25] Heirik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real–Time Systems*, 18 :157–179, may 2000.

[26] Randall T. White, Frank Mueller, Chris Healy, David Whalley, and Marion Harmon. Timing analysis for data and wrap–around fill caches. *Real–Time Systems*, 17 :209–233, nov 1999.

[27] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. 1997.

Index