Vers une prédictibilité temporelle des processeurs haute-performance

Christine Rochange et Pascal Sainrat

Institut de Recherche en Informatique de Toulouse
118 route de Narbonne, 31062 Toulouse cedex 4
rochange@irit.fr

Résumé

De nombreuses méthodes basées sur une analyse statique du code exécutable ont été proposées dans la littérature pour évaluer le temps d'exécution maximum (WCET) de programmes temps-réel. Le principal avantage de ces méthodes est de limiter les mesures à de petites portions de code (par exemple des blocs de base). Cependant, elles ont été développées pour des architectures de processeurs très simples et des travaux récents ont montré que ces méthodes ne sont pas fiables pour des architectures plus modernes, à cause de possibles interactions entre blocs de base distants. Dans cet article, nous suggérons que les architectures haute-performance pourraient intégrer des mécanismes qui préviennent ces interactions et ainsi être analysables par les méthodes habituelles. Des résultats de simulation indiquent que le prix de la prédictibilité, en termes de performance, est limité (moins de 10% de perte).

Plan

Mots clés : *temps réel, WCET, processeur superscalaire*

1. Introduction

La connaissance du **temps d'exécution maximum** d'un programme (ou **WCET** : Worst-Case Execution Time) est utile et nécessaire dès lors que l'on étudie des systèmes temps-réel stricts. Dans un contexte mono-tâche, elle permet de vérifier que le programme respecte les contraintes temporelles. Elle peut éventuellement conduire à redimensionner le système (en particulier par le choix d'un processeur plus adapté) si le temps de calcul maximum est très inférieur ou au contraire très supérieur aux contraintes (ce qui peut signifier que le processeur est trop ou pas assez puissant). Par ailleurs, dans une démarche d'optimisation de code, elle peut permettre d'identifier les points critiques du programme. Dans un contexte multi-tâches, le WCET estimé est utilisé par l'algorithme d'ordonnancement hors-ligne.

Le temps d'exécution d'un programme dépend souvent des données en entrée (par exemple de mesures issues de capteurs). Le WCET est la valeur maximale que l'on peut observer pour ce temps d'exécution, quel que soit le jeu de données en entrée. On parle de "temps d'exécution pire-cas". Il peut éventuellement être très supérieur au temps d'exécution habituellement mesuré (si, par exemple, il correspond à un profil d'exécution rare, comme un traitement d'erreur). Lorsque l'on ne sait pas en déterminer la valeur exacte, ce qui est le plus souvent le cas, on cherche à en calculer un **majorant**. Pour être utile, ce majorant doit avant tout être fiable, c'est-à-dire qu'il doit être impossible d'observer un temps d'exécution supérieur. Toutefois, il est aussi souhaitable qu'il ne soit pas trop surestimé. On peut argumenter que, si le temps d'exécution réel est finalement très inférieur au WCET estimé, le temps pendant lequel la tâche n'occupe pas le processeur peut être ré-alloué dynamiquement à une tâche sans contrainte temporelle. Cependant, il ne faut pas oublier que si l'estimation du WCET est trop éloignée de la réalité, il se peut que le système soit finalement déclaré à tort "non ordonnançable".

La recherche de méthodes d'estimation du temps d'exécution maximum de programmes a fait l'objet de nombreux travaux ces quinze dernières années. Pourtant, les résultats de ces travaux sont encore très éloignés des besoins puisqu'ils ne permettent pas de prendre en compte les matériels utilisés aujourd'hui. Dans cet article, nous nous intéressons aux difficultés qui se posent lorsque l'on souhaite évaluer le WCET pour un processeur cible pipeliné et superscalaire (comme de nombreux processeurs actuels).

Nous proposons une modification de l'architecture qui permettrait de rendre le processeur prédictible, c'est-à-dire compatible avec une estimation assez précise du WCET. Le défi d'une telle modification est de conserver un haut niveau de performance. Nous montrerons que c'est le cas pour la solution que nous suggérons.

L'organisation de l'article est la suivante. Tout d'abord, nous dressons un panorama rapide des méthodes de calcul du WCET et nous expliquons en quoi ces méthodes ne permettent pas de prendre en compte correctement les architectures modernes. Nous examinons ensuite plus en détail le problème posé par l'exécution dans un pipeline superscalaire. Nous proposons alors une modification du pipeline qui éviterait les interactions entre blocs de base, et nous montrons comment cette modification pourrait être intégrée dans le matériel. Enfin, nous évaluons les performances de l'architecture proposée et nous montrons que la dégradation nécessaire pour rendre le processeur prédictible reste très limitée.

2. Calcul du temps d'exécution maximum : état de l'art

2.1. Panorama des méthodes de calcul

Les méthodes d'estimation du WCET se divisent en deux catégories : les méthodes dynamiques, uniquement basées sur des mesures (ou simulations), et les méthodes statiques, qui reposent sur une analyse statique du code.

2.1.1 Recherche dynamique du WCET. Il s'agit de mesurer le temps d'exécution du programme, sur un processeur réel ou sur un simulateur. Obtenir un temps d'exécution pire-cas nécessite, soit de déterminer le jeu d'entrée qui conduit au chemin le plus long (ce qui est a priori complexe), soit de mesurer le temps d'exécution de tous les chemins possibles, c'est-à-dire de définir des jeux d'entrées qui contrôlent chacun de ces chemins (ce qui n'est pas plus facile). Si l'on utilise un simulateur, on peut mettre en œuvre une stratégie d'exécution symbolique [LuSt99] qui consiste à attribuer la valeur *INCONNUE* à chaque donnée en entrée, à propager cette valeur au fil des opérations réalisées par le programme, et, lorsqu'un branchement conditionnel dont la condition est *INCONNUE* est rencontré, à explorer les deux chemins (un algorithme de fusion de chemins permet

d'éviter que le nombre de chemins n'explose, mais c'est au prix d'approximations qui font que le résultat perd en précision). Dans tous les cas, si le nombre de chemins est très important, le temps de mesure (ou de simulation) est prohibitif.

2.1.2. Estimation du WCET par analyse statique. L'objectif des méthodes statiques est de limiter les mesures à des portions de code (par opposition à des chemins complets). Souvent, le grain est celui d'un bloc de base[(1)]. Découper ainsi chaque chemin en blocs de base permet de limiter le temps de mesure puisque chaque bloc est susceptible d'être commun à plusieurs chemins.
De manière schématique, l'analyse statique se déroule en trois phases. L'**analyse de flot** consiste à déterminer les chemins d'exécution possibles du programme. L'ensemble de ces chemins est représenté par un graphe de flot de contrôle et/ou par un arbre syntaxique annotés avec des indications de bornes de boucles, de chemins impossibles, etc … Ces annotations peuvent être fournies par l'utilisateur ou, dans certains cas, calculées de manière automatique [ChBW96] [ErGu98] [Park93] [PuKo89]. L'**analyse temporelle** a pour objectif de déterminer le temps d'exécution des blocs de base. Elle se fait généralement en deux étapes. Dans un premier temps, on analyse les mécanismes dont le comportement est lié à l'historique (mémoires caches, prédicteur de branchement, …) [AFMW96][CoPu00] [FeMW97]. Par exemple, il peut s'agir de déterminer si un accès au cache sera un succès ou un échec. Cette étape, dite **globale**, nécessite la prise en compte de chemins complets. Elle met en œuvre des techniques dérivées de l'interprétation abstraite [CoCo77]. Dans un second temps, on recherche (par simulation) le temps d'exécution des blocs de base, en exploitant les résultats de l'étape globale pour estimer le temps d'exécution des instructions. Cette étape est dite **locale** car le temps d'exécution d'un bloc de base ne dépend, en principe, que des blocs exécutés juste avant et juste après. Le **calcul du WCET** proprement dit consiste à combiner les temps d'exécution des blocs de base en parcourant l'arbre syntaxique [PuKo89] [LMLP94] (de bas en haut, avec des règles de combinaison des temps d'exécution de blocs liées aux structures algorithmiques) ou le graphe de flot de contrôle (comme le fait la méthode IPET décrite dans le paragraphe suivant).

---

[1] Il s'agit d'un bloc de base au sens "compilation" : une séquence d'instructions dont le seul point d'entrée (respectivement de sortie) est la première (respectivement la dernière) instruction.

2.1.3. La méthode IPET : Implicit Path Enumeration Technique. Le choix de la méthode IPET, basée sur le graphe de flot de contrôle, est lié au fait que nous nous intéressons au calcul de WCET pour des processeurs haute-performance. Dans ce contexte, il semble peu réaliste de se limiter à des codes non optimisés. Or, il n'est pas toujours possible, pour des programmes optimisés, d'établir un lien entre le code objet et l'arbre syntaxique (dérivé du code source).
La méthode IPET [LiMa95], associe un temps d'exécution et un nombre d'exécutions à chaque nœud (bloc de base) et à chaque arc (mise en séquence de deux blocs de base) du graphe. Sur un chemin d'exécution donné, on définit B l'ensemble des blocs de base qui constituent le chemin, b_i le nombre d'exécutions du bloc i sur ce chemin et t_i son temps d'exécution. On définit également A l'ensemble des arcs appartenant au chemin, a_i le nombre d'exécutions de l'arc i sur le chemin considéré et δ_i le gain apporté par l'exécution en séquence des deux blocs reliés par cet arc (ce gain est nul ou négatif dans un pipeline où le temps d'exécution total est inférieur ou égal à la somme des temps d'exécution individuels). Le temps d'exécution du chemin considéré est alors égal à :

$$T = \sum_{i \in B} b_i t_i + \sum_{i \in A} a_i \delta_i$$

Le WCET du programme est alors calculé en cherchant les valeurs qui maximisent T tout en respectant des contraintes structurelles (qui traduisent la structure du graphe de flot de contrôle, en liant entre eux les nombres d'exécutions des blocs et des arcs) et des contraintes de flot (qui expriment les résultats de l'analyse de flot, comme des bornes de boucles par exemple).

2.2. Analyse statique des architectures modernes

Les diverses méthodes d'analyse statique proposées dans la littérature ne permettent pas d'étudier correctement les architectures modernes. Nous allons énumérer les principales difficultés qui se posent.

2.2.1. Difficulté à identifier les chemins possibles à cause de l'exécution spéculative. L'exécution spéculative consiste à poursuivre l'exécution sur une des branches après qu'un branchement ait été prédit et en attendant qu'il soit effectivement résolu, quitte à revenir en arrière (mécanisme de récupération) si le chemin suivi n'était pas le bon. L'exécution du code sur un mauvais chemin ne doit pas être ignorée car elle peut avoir des conséquences sur l'exécution ultérieure du chemin correct : impact sur

l'ordonnancement des instructions dans le pipeline, pollution des mémoires caches [PiMu96] et des tables de prédiction de branchement [JHSP96], … Or, il ne nous semble pas possible de prendre en compte cette exécution d'un mauvais chemin dans les méthodes statiques que nous avons présentées. En effet, l'analyse des chemins est, en général, totalement découplée de l'analyse temporelle, alors que ce n'est que de manière dynamique que l'on peut savoir jusqu'où le processeur s'engage dans un chemin erroné. Notons que l'outil AbsInt (`www.absint.com`) contourne la difficulé en envisageant tous les cas possibles, mais le système devient alors très complexe (il nous a été rapporté que le nombre d'états possibles du processeur lors du traitement d'une instruction pouvait être de l'ordre du millier).

2.2.2. Problèmes d'indéterminisme dans l'analyse temporelle globale. L'analyse temporelle globale considère tous les chemins en parallèle pour modéliser les mécanismes qui reposent sur l'historique d'exécution (mémoires caches, prédicteur de branchement) et déterminer le comportement de certaines instructions. Son rôle est de déterminer, pour chaque accès au cache, si ce sera un succès ou un échec, et pour chaque accès au prédicteur de branchement, si le branchement sera correctement prédit ou non. Dans un certain nombre de cas, l'analyse est capable de donner une réponse précise mais c'est loin d'être toujours possible.
Quand elle ne sait pas donner une réponse fiable, l'analyse doit donner une réponse qui corresponde au pire cas. C'est là que réside l'une des difficultés. En effet, alors que pendant longtemps l'échec dans le cache a été considéré comme le pire cas, il a été montré que ce n'était pas forcément vrai si l'on considère des processeurs récents, à ordonnancement dynamique des instructions [LuSt99]. Ce que nous appelons *indéterminisme* est le fait que l'analyse temporelle ne soit pas toujours capable de définir un comportement pire-cas pour les instructions.

2.2.3. Difficultés de prise en compte des interactions entre blocs de base. Nous avons vu que les méthodes statiques reposent sur des mesures du temps d'exécution de blocs de base considérés de manière indépendante. Or, dans les architectures évoluées organisées autour d'un pipeline, il y a des interactions temporelles entre blocs de base. Comme nous l'avons déjà indiqué, l'exécution en séquence de deux blocs de base dans un pipeline dure moins longtemps que la somme des temps d'exécution individuels. La prise en compte des interactions entre deux

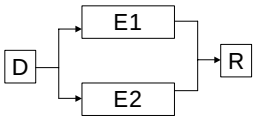
blocs adjacents dans une méthode telle que la méthode IPET nécessite que l'analyse temporelle locale calcule aussi le temps d'exécution de la séquence pour en déduire le gain de l'arc reliant les deux blocs.
Nous allons voir dans la section suivante que ce n'est pas suffisant et que l'exécution d'un bloc de base peut avoir un impact non seulement sur l'exécution du bloc suivant mais aussi sur celle d'un bloc beaucoup plus éloigné. Les méthodes statiques actuelles, qu'elles soient basées sur le graphe de flot de contrôle ou sur l'arbre syntaxique, ne permettent pas de modéliser ce type d'interaction.

3. Pipelines et effets à long terme

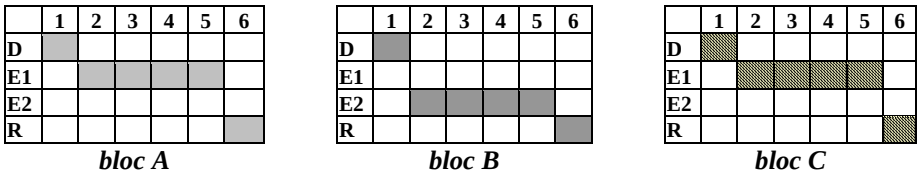
3.1. Exposé du problème

Le fait que l'exécution de deux blocs de base en séquence dans un pipeline produise un gain sur le temps d'exécution total semble naturel. Dans le cadre du calcul de WCET par la méthode IPET, ce gain (exprimé par une valeur négative) peut être considéré comme le temps d'exécution de l'arc qui représente la mise en séquence des deux blocs. Or Engblom a mis en évidence dans sa thèse [Engb02] qu'il pouvait aussi y avoir des interactions entre blocs distants. Ce phénomène est illustré par l'exemple suivant.

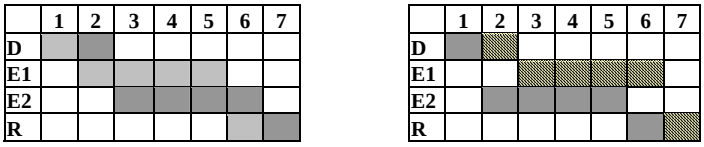
Considérons un pipeline à 3 étages : *décodage, exécution* et *retrait*. L'étage d'exécution comporte deux unités fonctionnelles, chacune étant dédiée à certains types d'instructions.



Les tables de réservation ci-dessous représentent l'exécution de 3 blocs de base, *A*, *B* et *C*, ainsi que celle de séquences de 2 de ces blocs (chaque bloc ne contient ici qu'une seule instruction, mais le phénomène se produit aussi avec des blocs plus longs).



bloc A ***bloc B*** ***bloc C***



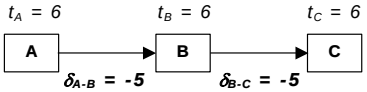
séquence A-B ***séquence B-C***

On peut en déduire le temps d'exécution des blocs et des séquences, et par suite les gains associés aux séquences. On trouve :

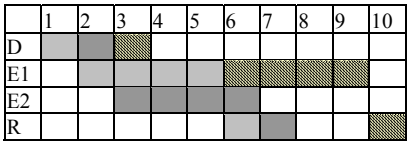
$t_A = 6$, $t_B = 6$, $t_C = 6$

$t_{A\text{-}B} = 7$ donc $\delta_{A\text{-}B} = t_{A\text{-}B} - t_A - t_B = -5$

$t_{B\text{-}C} = 7$ donc $\delta_{B\text{-}C} = t_{B\text{-}C} - t_B - t_C = -5$



Si l'on appliquait le calcul de la méthode IPET à ce mini-graphe de flot, on trouverait que le temps d'exécution de la séquence *A-B-C* est $t_{A\text{-}B\text{-}C} = t_A + t_B + t_C + \delta_{A\text{-}B} + \delta_{B\text{-}C} = 8$. Or, si l'on examine la table de réservation correspondante, on trouve $t_{A\text{-}B\text{-}C} = 10$.



séquence A-B-C

La différence entre le temps calculé et le temps observé est appelée *effet à long terme*. Elle peut être exprimée par un gain associé à la séquence : $\delta_{A\text{-}B\text{-}C} = t_{A\text{-}B\text{-}C} - t_A - t_B - t_C - \delta_{A\text{-}B} - \delta_{B\text{-}C} = +2$. Notons que le gain est ici positif (il augmente le temps d'exécution total), mais qu'il pourrait aussi bien être négatif ou nul.

L'origine du gain associé à la séquence de 3 blocs est l'impact de l'exécution de *A* sur celle de la séquence *B-C*. En effet, on observe que le séquencement du bloc *C* par rapport à *B* est différent selon que *A* a été exécuté avant *B* ou non : si *A* n'a pas été exécuté avant *B*, l'entrée de

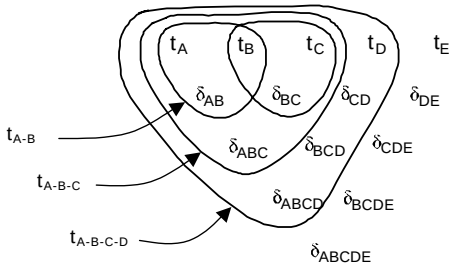
l'instruction de *C* dans l'étage E1 se produit au 3ème cycle après le début de *B* ; si *A* a été exécuté avant, cette entrée est retardée au 5ème cycle à cause de l'occupation de E1 par une instruction de *A* non terminée.

3.2. Le modèle de Engblom

Jakob Engblom [Engb02] a proposé un modèle temporel général pour représenter les temps d'exécution des blocs et des séquences de blocs. Ce modèle est exprimé par les équations et le schéma suivants :

$$t_{N_1 \ldots N_n} = \sum_{j=1}^{n} t_{N_j} + \sum_{1 \le i < k \le n} \delta_{N_i \ldots N_k}$$

$$\delta_{N_1 \ldots N_n, n \ge 2} = t_{N_1 \ldots N_n} - t_{N_2 \ldots N_n} - t_{N_1 \ldots N_{n-1}} + t_{N_2 \ldots N_{n-1}}$$



3.3. Origine des effets à long terme.

Engblom a recensé un certain nombre d'éléments susceptibles de générer des effets à long terme :

– présence de pipelines parallèles, et en particulier d'un pipeline superscalaire : si le nombre d'instructions d'un bloc de base n'est pas un multiple du degré[2] du pipeline, il peut engendrer des effets à long terme. Nous avons montré, dans une étude précédente [RoSa03], que pour un pipeline superscalaire parfait (sans bulles), les effets à long terme pouvaient prendre les valeurs 0, -1 et +1 selon les tailles de blocs constituant les séquences. Engblom a par ailleurs montré sur un exemple que des séquences de longueur quelconque (éventuellement jusqu'à la fin du programme) pouvaient présenter un gain non nul.

---

[2] le degré est la largeur du pipeline, c'est-à-dire le nombre d'instructions qui peuvent être traitées par un étage (décodage, émission, retrait, …) en un cycle.

- possibilité de dépendances entre instructions non adjacentes : un blocage du pipeline (en attendant que l'instruction productrice ait été exécutée) peut engendrer un effet à long terme.
- présence d'unités fonctionnelles de latence supérieure à un cycle (comme dans l'exemple que nous avons présenté pour introduire le problème) : elles peuvent elles-aussi être source de blocages du pipeline et donc d'effets à long terme.

Cette liste n'est sans doute pas exhaustive. De plus, ces différents éléments étant généralement présents conjointement, il est fréquent que les effets à long terme aient plusieurs sources pas toujours faciles à identifier précisément.
De manière plus générale, Engblom a montré [Engb02] qu'il ne peut y avoir d'effet à long terme pour une séquence d'instructions $I_1 \ldots I_m$, avec $m \geq 3$, que si I_1 bloque l'exécution d'une instruction dans la séquence $I_2 \ldots I_m$ ou si I_1 est *finalement* parallèle à $I_2 \ldots I_m$, c'est-à-dire si I_1 se termine après la séquence $I_2 \ldots I_m$. Comme nous supposons que le retrait des instructions se fait dans l'ordre du programme (ce qui est vrai dans tous les processeurs actuels), nous ne retiendrons pas le second cas.

3.4. Calcul de WCET et effets à long terme

3.4.1. Peut-on prendre en compte les effets à long terme dans le calcul de WCET ? Nous avons vu que, dans le cadre de la méthode IPET, le recouvrement de deux blocs exécutés en séquence dans le pipeline peut être modélisé par un gain (négatif ou nul) associé à l'arc qui relie les deux nœuds correspondants dans le graphe de flot de contrôle.
Il semble difficile de prendre en compte les effets à long terme de la même manière, et ce pour les raisons suivantes. Tout d'abord, comme nous l'avons mentionné, un effet à long terme peut se produire pour une séquence de longueur quelconque. Evaluer cet effet nécessiterait donc de mesurer des séquences de longueur quelconque, jusqu'à des chemins complets. Cela va à l'encontre du principe de base des méthodes statiques qui limite les mesures à des blocs de base. Ensuite, l'introduction d'un effet à long terme dans le jeu de contraintes n'est sans doute pas évidente : il faudrait ajouter dans le modèle des nombres d'exécution associés à toutes les séquences possibles de blocs de base, ce qui augmenterait sans doute la complexité du problème de manière insupportable.

3.4.2. Solutions. Puisque les effets à long terme ne peuvent pas, a priori, être pris en compte dans le calcul du WCET par des méthodes statiques, il est nécessaire de limiter le choix du processeur à une architecture qui ne peut pas donner lieu à de tels effets. Cet objectif peut être recherché selon plusieurs approches différentes.

Tout d'abord, comme Engblom le suggère [Engb02], un certain nombre de processeurs simples ne peuvent pas générer d'effets à long terme. Il s'agit de processeurs scalaires, à exécution ordonnée et non spéculative, sans dépendance entre instructions non adjacentes. C'est le cas des architectures ARM7 et ARM9, par exemple. Engblom montre aussi comment on peut résoudre le problème pour des processeurs un peu plus compliqués, susceptibles de générer des effets à long terme mais des effets qui ne peuvent se produire qu'immédiatement après leur cause (c'est le cas du NEC V850E) : un algorithme permet de calculer les séquences à mesurer (à simuler) pour être sûr d'observer tous les effets à long terme. Toutefois, nous pensons que ce type d'architecture simple ne peut pas satisfaire des besoins croissants de performance pour certaines applications temps-réel.

Une autre approche consiste à utiliser un modèle de processeur très simple (sans effets à long terme) pour calculer le WCET tout en retenant comme architecture cible un processeur haute-performance [ASPR03]. Pour garantir le respect de contraintes temps-réel, le processeur haute-performance est modifié de manière à pouvoir basculer dans un mode d'exécution sûr (celui du processeur simple pour lequel le WCET a été calculé) en cas de dépassement du temps prévu. Cette approche est basée sur l'idée que, le plus souvent, le temps d'exécution sur le processeur rapide sera inférieur au temps d'exécution pire-cas calculé pour le processeur lent. Comme nous l'avons évoqué en introduction, quand la tâche se termine plus tôt que prévu, le processeur peut être alloué à d'autres tâches non critiques. Toutefois, avec un WCET trop largement surestimé, on court le risque de ne pas réussir à ordonnancer l'ensemble des tâches critiques.

C'est pourquoi nous proposons une autre voie qui consiste à intégrer dans les architectures modernes la logique nécessaire pour supprimer les risques d'effets à long terme tout en maintenant un niveau de performance élevé. Afin de ne pas limiter le processeur au marché du temps-réel, ce qui ne serait pas viable d'un point de vue économique, cette logique peut être désactivée et le processeur retrouve alors ses performances initiales. C'est cette approche que nous allons exposer dans la suite de cet article.

4. Vers des processeurs pour le temps-réel

L'objectif est de faire en sorte qu'il ne puisse pas se produire d'effets à long terme entre blocs de base. Une solution triviale consisterait à interdire l'entrée d'un nouveau bloc de base dans le pipeline tant que le bloc précédent n'en est pas sorti. La figure 1 montre la perte de performance engendrée par cette solution pour un processeur superscalaire à exécution ordonnée (il s'agit de résultats de simulations dont les paramètres sont décrits en annexe).

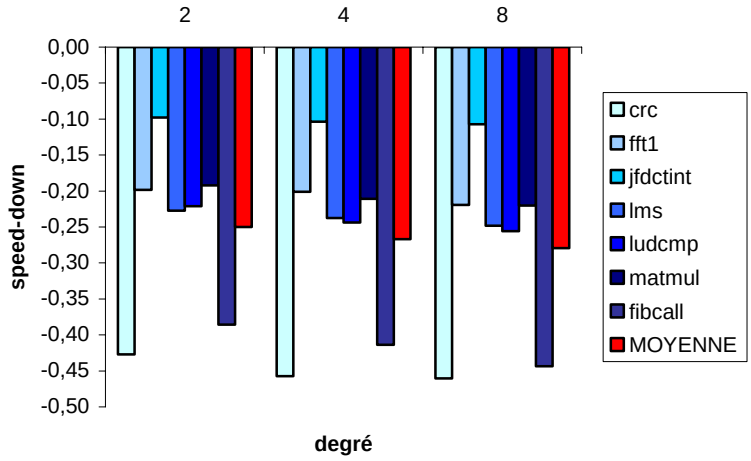


Figure 1. *Perte de performance liée à une purge du pipeline entre deux blocs de base*

On observe que la perte (calculée sur le nombre d'instructions exécutées par seconde) est de l'ordre de 20 à 28% en moyenne, selon le degré du processeur, avec des variations entre les différents benchmarks. Les deux programmes qui subissent le plus de perte sont *crc* et *fibcall*. Or ils se caractérisent par le fait qu'ils comportent des blocs de base très courts : respectivement 5,6 et 8 instructions en moyenne. Aussi, le coût de la purge du pipeline entre deux blocs est très lourd et la perte peut atteindre 46%. Au contraire, *jfdctint*, qui a les blocs les plus longs (34,3 instructions en moyenne) enregistre le moins de perte. Les quatre autres programmes ont des résultats très proches avec des tailles moyennes de blocs allant de 10,6 à 16,9 instructions. D'autre part, plus le

degré est grand, plus le "manque à gagner" est important. C'est ce qui explique que la perte croît avec le degré.
Au vu de l'importance de la perte de performance atteinte lorsque l'on attend que le pipeline soit vide pour charger un nouveau bloc, on peut se demander si, dans ces conditions, il ne vaudrait pas mieux se restreindre à un pipeline de degré 1, dont il a été montré qu'il ne pouvait pas générer d'effet à long terme. La figure 2 montre que c'est le cas : même avec un pipeline de degré 8, le niveau de performance est inférieur, pour tous les benchmarks sauf *jfdctint*.

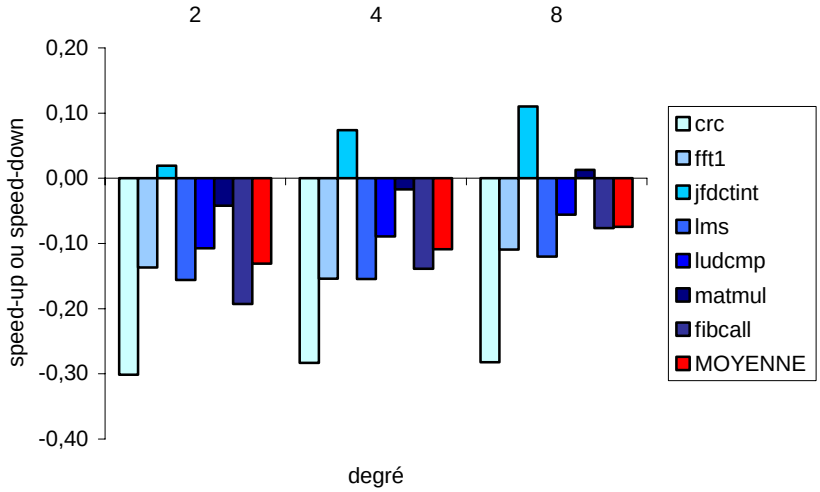


Figure 2. *Perte de performance pour un processeur avec purge du pipeline entre blocs de base par rapport à un processeur scalaire*

Ainsi, il semble souhaitable d'adopter une approche moins brutale : c'est ce que nous proposons dans la section suivante.

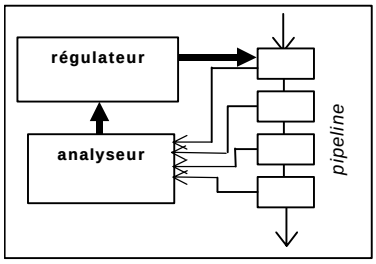
5. Solution proposée : régulation du flot d'instructions

5.1. Principe

L'idée est d'interdire l'entrée d'un nouveau bloc dans le pipeline non pas tant que le bloc précédent n'est pas sorti mais *tant que le bloc précédent est susceptible de générer des effets à long terme*.

Comme expliqué au paragraphe 3.3, un effet à long terme est lié au fait qu'une instruction bloque l'exécution d'une autre instruction appartenant à un bloc de base ultérieur, à cause d'une dépendance de donnée ou de ressource. La clé du problème est alors d'être capable de déterminer quand une instruction du bloc présent dans le pipeline est susceptible de bloquer une instruction à venir. Pour ce faire, nous proposons d'anticiper dynamiquement les tables de réservation des différentes unités du processeur et de les confronter aux dates au plus tôt de requêtes aux unités d'une instruction qui entrerait dans le pipeline. Cette confrontation permet de déterminer si cette nouvelle instruction risque d'être retardée par une des instructions du pipeline.
En pratique, cette solution peut être implémentée dans le matériel sous la forme de deux modules :

- un module dit *analyseur* examine, à chaque cycle, si le pipeline contient des instructions susceptibles de provoquer un blocage.
- un module dit *régulateur*, intégré à l'unité de chargement des instructions, empêche le chargement d'un nouveau bloc tant que le pipeline n'est pas garanti "sans risque de blocage".



5.2. Algorithmes d'analyse et de régulation

5.2.1. Chronologie des instructions. La solution proposée est basée sur le calcul de la chronologie des instructions qui traversent le pipeline et sur l'occupation des ressources qui en découle.
Les points-clés de la chronologie d'une instruction sont les dates d'émission de l'instruction vers une unité fonctionnelle, d'obtention du résultat et de retrait de l'instruction du pipeline. Différents éléments liés à l'architecture et aux autres instructions présentes dans le pipeline interviennent dans le calcul de ces dates : la durée de traitement de l'instruction considérée (la latence de l'unité fonctionnelle peut être fixe ou variable, auquel cas on doit en connaître la valeur maximum), les dépendances de données vis-à-vis des instructions antérieures et les

dépendances de ressources (unités fonctionnelles mais aussi étages du pipeline).
Le calcul de la chronologie d'une instruction nécessite alors de connaître les occupations (tables de réservation) des ressources, ainsi que les dates de disponibilités des opérandes :

- la date d'émission d'une instruction est le minimum des dates de disponibilité de l'étage d'émission, des opérandes en entrée et de l'unité fonctionnelle. Comme les instructions sont émises dans l'ordre, les dates de disponibilité de tous les crénaux de l'étage d'émission sont alignées sur la date d'émission de l'instruction considérée.
- la date de fin d'exécution de l'instruction est la date d'émission à laquelle on ajoute la latence de l'unité fonctionnelle (si cette latence n'est pas fixe, on considère la latence pire-cas). L'unité fonctionnelle est occupée jusqu'à cette date si elle n'est pas pipelinée. Sinon, elle est occupée pendant un cycle à compter de la date d'émission. Les opérandes résultat sont notés disponibles au dernier cycle d'exécution de l'instruction (prise en compte du mécanisme de *bypass*).
- la date de complétion d'une instruction est le minimum des dates de fin d'écriture du résultat (cycle suivant la fin de l'exécution) et de disponibilité de l'étage de complétion. Comme les instructions sont retirées dans l'ordre, les dates de disponibilité de tous les crénaux de l'étage de complétion sont alignées sur la date de complétion de l'instruction considérée.

5.2.2. Politique de régulation. On veut pouvoir déterminer, lorsque l'on va charger la première instruction d'un nouveau bloc de base, si les instructions déjà présentes dans le pipeline peuvent avoir une influence sur sa chronologie (c'est-à-dire si elles peuvent la retarder). Si c'est le cas, on diffère le chargement du nouveau bloc (jusqu'à ce qu'il n'y ait plus de risques de blocage).
Pour cela, on définit la chronologie *meilleur-cas* d'une instruction qui entrerait dans le pipeline. Il s'agit des dates d'émission et de retrait du pipeline en supposant que ce dernier est vide et que l'instruction sera exécutée par l'unité fonctionnelle de latence minimale. On vérifie alors que ces dates sont compatibles avec les tables de réservation des différentes ressources et avec les dates de disponibilités des registres : l'étage d'émission, toutes les unités fonctionnelles et tous les registres doivent être disponibles à la date d'émission meilleur-cas ; l'étage de retrait doit être disponible à la date de retrait meilleur-cas. Tant que ces

conditions ne sont pas vérifiées, le chargement du nouveau bloc de base est retardé.

5.2.3. Implémentation matérielle. Dans la pratique, on a besoin de détecter l'arrivée d'un nouveau bloc de base dans le flot d'instructions. Rappelons qu'on parle de blocs de base au sens "compilation", et qu'ils ne se terminent donc pas forcément par une instruction de branchement. Aussi, un support du compilateur est nécessaire. Si le jeu d'instructions le permet, on peut envisager qu'un bit non utilisé du code des instructions soit exploité pour indiquer le début des blocs de base. Si cela n'est pas possible, on peut demander au compilateur de terminer tous les blocs de base par des instructions de branchement (en introduisant si nécessaire un branchement vers l'instruction suivante). Ainsi, le matériel repère la fin d'un bloc de base dès qu'il trouve une instruction de branchement.
Notons que la détection d'une instruction de branchement ne peut se faire qu'à l'étage de décodage. Cela n'est pas un problème : il suffit de décider que c'est au niveau de cet étage que se fait la régulation (qu'un bloc soit bloqué au chargement ou au décodage ne change rien à son temps de traitement).
Enfin, pour décider si un nouveau bloc de base peut entrer dans le pipeline, on doit connaître la chronologie de toutes les instructions qui le précèdent et la disponibilité des ressources qui en découle. Aussi, on doit empêcher un nouveau bloc de progresser dans le pipeline au-delà de l'étage de décodage tant que cet étage n'est pas vide.

Nous donnons sur la figure 3 un schéma de principe pour l'implémentation du calcul des disponibilités des ressources et des registres. A chaque ressource et à chaque registre, on associe un vecteur de bits V. Le $i^{ème}$ bit de ce vecteur est à 1 si le composant associé doit être occupé dans i cycles : autrement dit, si le composant doit être disponible dans n cycles (et occupé pendant les n-1 prochains cycles) son vecteur de disponibilité est égal à 2^n-1. A chaque cycle, tous les vecteurs de disponibilité sont décalés d'un cran vers la droite. Le schéma représente la logique de mise à jour de l'ensemble des vecteurs lors du calcul de la chronologie d'une instruction.
Dans le cas d'un processeur superscalaire, on doit calculer ainsi les chronologies de plusieurs instructions par cycle, dont chacune dépend de la précédente. S'il n'est pas envisageable de répéter plusieurs fois en série la logique ci-dessus, il est possible de calculer toutes les chronologies en

parallèle en calculant de manière anticipée la disponibilité des ressources et des registres après traitement de l'instruction précédente.

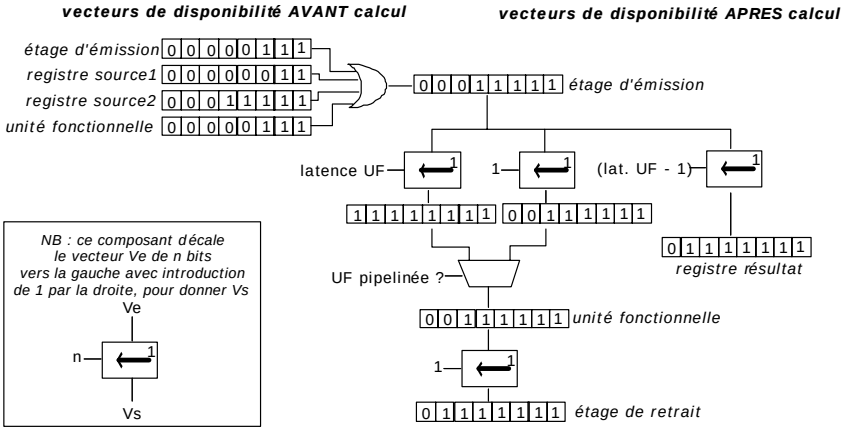


Figure 3. *Implémentation matérielle du mécanisme de régulation.*

5.3. Evaluation de performances

L'histogramme de la figure 4 présente la perte de performance d'un processeur lorsque la régulation du flot d'instructions pour empêcher les effets à long terme est activée, par rapport au même processeur sans régulation. Il apparaît que la perte est très modérée : de 8 à 10% en moyenne, selon le degré du processeur, et 22% dans le pire des cas. Certains benchmarks enregistrent une dégradation plus importante que les autres, pour les mêmes raisons que celles évoquées au paragraphe 4. D'autres conservent quasiment leur débit d'instructions. Une comparaison avec un processeur sans effets à long terme (processeur scalaire) confirme que notre approche est intéressante puisque l'accélération moyenne est de 7%, 11% et 18% pour des processeurs de degré 2, 4 ou 8. Nous avons également mesuré que la durée moyenne de retardement de l'entrée d'un bloc dans le pipeline est de 11,9 cycles pour un processeur de degré 2 (respectivement 14 cycles pour un degré 4 et 16,2 cycles pour un degré 8). Le fait que ces retardements qui paraissent importants n'induisent que peu de perte de performance est lié au fait que, sans régulation, une partie des instructions sont bloquées un peu plus loin dans le pipeline (rappelons que les instructions sont exécutées dans l'ordre du

programme, ce qui limite le parallélisme) et qu'il en résulte que le recouvrement de blocs de base dans le pipeline est loin d'être complet.

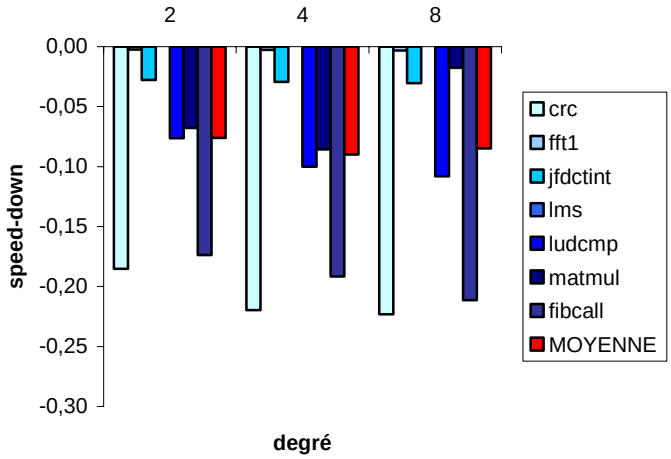


Figure 4. *Perte de performance engendrée par la régulation du flot d'instruction.*

6. Conclusion

Cet article fait suite à la thèse de Engblom [Engb02] qui a mis en évidence que l'exécution d'un bloc de base pouvait avoir un impact sur l'exécution d'un bloc ultérieur. Cet impact peut aussi bien augmenter que diminuer le temps d'exécution global et doit être pris en compte si on veut calculer un WCET fiable. Or, dans le cadre des méthodes statiques d'évaluation du WCET, on souhaite limiter les mesures à des blocs de base (et non pas à des séquences de blocs). Ceci ne permet pas d'évaluer les effets entre blocs distants, et donc il faudrait se restreindre à des architectures qui ne peuvent pas générer de tels effets. Engblom a montré que des processeurs très simples avaient cette propriété.
Dans l'hypothèse où de tels processeurs ne suffiraient pas à satisfaire les besoins en performance d'applications temps-réel, nous avons proposé une autre approche qui consiste à inclure dans un processeur haute-performance un dispositif qui détecte les situations dans lesquelles un effet à long terme est susceptible de se produire et prend les mesures appropriées pour l'éviter. Ces mesures consistent à réguler le flot d'instructions en retardant l'introduction d'un nouveau bloc de base dans

le pipeline tant que ce dernier contient des instructions susceptibles d'avoir un impact sur la progression du nouveau bloc. Nous avons appliqué cette approche à un processeur superscalaire à exécution ordonnée. Les résultats de simulation montrent que la dégradation des performances induite par l'activation de ce dispositif est très limitée, ce qui confirme l'intérêt de l'approche proposée.

Références

[AFMW96] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm. *Cache behavior prediction by abstract interpretation*. Static Analysis Symposium (LNCS vol. 1145). 1996.

[ASPR03] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, F. Mueller. *Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems.* 30th International Symposium on Computer Architecture. 2003.

[ChBW96] R. Chapman, A. Burns, A. Wellings. *Combining static worst-case analysis and program proof.* Real-Time Systems, vol. 11, n°2. 1996.

[CoCo77] P. Cousot, R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints*. 4th ACM Symposium on Principles of Programming Languages. 1977.

[CoPu00] A. Colin, I. Puaut. *Worst-case execution time analysis for a processor with branch prediction.* Real-Time Systems, vol. 18, n°2-3. 2000.

[Engb02] J. Engblom. *Processor pipelines and static worst-case execution time analysis.* PhD thesis, Univ. of Uppsala. 2002.

[ErGu98] A. Ermedahl, J. Gustafsson. *Automatic derivation of path and loop annotations in object-oriented real-time programs*. Journal of Parallel and Distributed Computing, vol.1, n°2. 1998.

[FeMW97] C. Ferdinand, F. Martin, R. Wilhelm. *Applying compiler technique to cache behavior prediction*. Workshop on Languages, Compilers and Tools for Embedded Systems. 1997.

[JHSP96] S. Jourdan, T.-H. Hsing, J. Stark, Y. Patt. *The effects of mispredicted-path execution on branch prediction structures*. International Conference on Parallel Architecture and Compilation Techniques. 1996.

[LiMa95] Y.-T. Li, S. Malik. *Performance analysis of embedded software using implicit path enumeration*. ACM SIGPLAN Notices, vol. 30, n°11. 1995

[LMLP94] S.-S. Lim, S. Min, M. Lee, C. Park, H. Shin, C.-S. Kim. *An accurate instruction cache analysis technique for real-time systems*. IEEE Workshop on Architectures for Real-Time Applications. 1994.

[Park93] C. Park. *Predicting program execution times by analyzing static and dynamic program paths*. Real-Time Systems, vol. 5, n°1. 1993.

[PiMu96] J. Pierce, T. Mudge. *Wrong-path instruction prefetching*. IEEE Symposium on Microarchitecture. 1996.

[PuKo89] P. Puschner, C. Koza. *Calculating the maximum execution time of real-time programs*. Real-Time Systems, vol. 1, n°2. 1989.

[RoSa03] C. Rochange, P. Sainrat. *Towards designing WCET-predictable processors.* 3rd Workshop on WCET analysis. 2003.

Annexe : conditions de simulation

Les mesures présentées dans cet article ont été obtenues par simulation. Le processeur modélisé est un processeur superscalaire, de degré variable, à exécution des instructions dans l'ordre, avec une prédiction de branchement parfaite et une hiérarchie mémoire également parfaite (accès en un cycle).

Les benchmarks utilisés sont issus de la suite SNU de l'université de Singapour (`http://archi.snu.ac.kr/realtime/benchmark/`). Il s'agit de petits programmes qui réalisent des fonctions courantes dans les applications temps-réel. Le fait que les exécutables soient de petite taille n'est pas gênant dans notre étude puisque nous étudions des phénomènes locaux qui ne nécessitent pas la construction d'historiques.