Master 2 de Recherche en Informatique
Université de Rennes 1

Transformation du code pour en améliorer la prévisibilité temporelle

Nicolas Curt
juin 2007

Institut de Recherche en Informatique de Toulouse

sous l'encadrement de
Mme Christine Rochange

Table des matières

1 Le temps d'exécution pire cas **7**
1.1 Définition et principes . . . 7
1.1.1 Définition . . . 7
1.1.2 Déterminer le *WCET* . . . 7
1.2 Approches dynamiques . . . 7
1.3 Approches statiques . . . 9
1.3.1 Analyse de haut niveau . . . 9
1.3.2 Analyse de bas niveau . . . 10
1.3.3 Calcul du pire temps . . . 10

2 *WCET* pour architectures modernes **13**
2.1 Les sources de complexité des microprocesseurs . . . 13
2.1.1 Le *pipeline* . . . 13
2.1.2 Exécution ordonnée et non-ordonnée . . . 14
2.1.3 Mémoire cache . . . 14
2.1.4 Prédiction de branchement . . . 15
2.1.5 Sujet de l'étude . . . 15
2.2 Anomalies temporelles . . . 15
2.2.1 Accès au cache . . . 15
2.2.2 Des délais difficiles à borner . . . 16

3 Modélisations pour le calcul du *WCET* **17**
3.1 Différentes approches pour le calcul du *WCET* . . . 17
3.1.1 Interprétation abstraite . . . 17
3.1.2 Exécution symbolique . . . 17
3.2 Modèle processeur . . . 18
3.3 Modèle de Li . . . 19
3.3.1 Graphes d'exécution . . . 20
3.3.2 Algorithme d'estimation des délais . . . 22
3.3.3 Calcul du *WCET* . . . 25
3.4 Les graphes paramétrés . . . 25
3.4.1 Modélisation des contextes d'exécution . . . 26
3.4.2 Calcul des délais . . . 27
3.4.3 Cacul du *WCET* . . . 29

4 Transformation du code pour déterminer le *WCET* **31**
4.1 Modifications de l'architecture et du code pour le calcul du *WCET* . . . 31
4.1.1 Adaptation de l'architecture . . . 31
4.1.2 Transformation du code . . . 32
4.2 Nouvelle approche pour la transformation de code . . . 35
4.2.1 Pourquoi transformer le code pour calculer le *WCET* . . . 35
4.2.2 Les effets temporels à éliminer . . . 35
4.2.3 Algorithme . . . 36
4.3 Evaluation . . . 38

4.3.1 Méthodologie 38
4.3.2 Résultats expérimentaux 38
4.3.3 Perspectives 43

Introduction

Un système temps-réel se différencie des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat.

On distingue, cependant, deux domaines dans l'informatique temps-réel : le temps réel dur ou strict (*hard real-time*) et le temps réel mou (*soft real-time*), correspondant à des degrés de criticité de l'application différents.

- Le temps réel dur ne tolère aucun dépassement des délais temporels spécifiés. Il concerne des applications hautement critiques comme un pilote automatique dans un avion ou un système de surveillance d'une centrale nucléaire.
- Le temps réel mou peut s'accomoder exceptionnellement de dépassements des contraintes temporelles au prix d'une altération de la qualité de l'application. On pense par exemple à des applications multimédias.

Pour valider un programme destiné à des applications temps-réel strict, il est nécessaire de garantir que les contraintes temporelles sont toujours respectées.

Ainsi, pour un programme donné, on désire calculer le pire temps d'exécution possible. A défaut de déterminer précisément ce pire temps d'exécution, il est indipensable d'en fournir un majorant, toute sous-estimation pouvant avoir des effets catastrophiques.

La connaissance de ce pire temps (*Worst Case Excecution Time* ou *WCET*) permettra de valider ou non le programme. Il permettra également d'ajuster les ressources matérielles disponibles. C'est pourquoi on cherche à obtenir le plus petit majorant possible de la durée d'exécution du programme, afin d'éviter un « gaspillage » de ressources.

Ce document reviendra dans un premier temps sur les principes de base du calcul du *WCET* et présentera quelques méthodes de calcul fondées sur une analyse statique. Les principaux défis posés par les architectures modernes des processeurs seront ensuite évoqués, puis l'on s'intéressera plus particulièrement aux problèmes liés au *pipeline*. Pour introduire les études menées, on s'attardera sur le modèle proposé dans [LRM06] et à l'approche apparentée faite sur des « graphes paramétrés ». Ce dernier modèle est à la base des travaux menés sur la transformation du code pour améliorer la prévisibilité temporelle, qui seront présentés dans la dernière partie.

Chapitre 1

Le temps d'exécution pire cas

Les applications temps réels strict nécessitent que l'on puisse garantir que les délais d'exécution seront toujours respectés. A cette fin a été introduite la notion de pire temps d'exécution ou *Worst Case Execution Time* (*WCET*). Cette partie se propose de donner quelques repères généraux sur le calcul du pire temps d'exécution d'un programme.

1.1 Définition et principes

1.1.1 Définition

L'ensemble des jeux d'entrées possibles pour un programme permet de définir l'ensemble des chemins d'exécution possibles pour le programme. Le pire temps d'exécution ou *Worst Case Execution Time* (*WCET*) est le plus grand des temps d'exécution des chemins possibles. Déterminer le pire temps d'exécution consiste à trouver, à partir d'un programme et d'une architecture cible donnés, un majorant aussi petit que possible du temps d'exécution du programme.

1.1.2 Déterminer le *WCET*

Différentes techniques ont été élaborées pour estimer le *WCET*. On distingue deux grandes approches :

- les méthodes **dynamiques** mesurent le pire temps d'exécution à partir de jeux d'entrées. Pour être fiables, ces méthodes doivent soit s'appuyer sur tous les jeux d'entrées possibles, soit utiliser les entrées dont on est sûr qu'elles conduiront au temps d'exécution le plus long. Ces deux conditions sont généralement très difficiles à garantir [CPRS03]. En effet, le nombre de jeux d'entrées possibles est souvent très grand : le temps nécessaire pour déterminer tous les jeux possibles est alors rédhibitoire. De même, il est souvent impossible de savoir a priori qu'un jeu d'entrée conduira au pire temps d'exécution.

- les méthodes **statiques** cherchent à évaluer le pire temps d'exécution par une analyse du programme. Elles se décomposent schématiquement en trois étapes : une analyse de haut niveau qui s'attache à déterminer les chemins d'exécution possibles ; une analyse de bas niveau qui calcule le temps d'exécution d'une séquence d'instructions ; et le calcul proprement dit du *WCET*. On peut noter que l'analyse bas niveau nécessite impérativement une modélisation fidèle de l'architecture ciblée.

1.2 Approches dynamiques

Les approches dynamiques pour déterminer le pire temps d'exécution d'un programme s'attachent à mesurer le temps d'exécution à partir de jeux d'entrées possibles.

Pour effectuer les mesures du temps d'exécution, on peut s'appuyer sur :

- le système matériel ciblé. Il permet de garantir la pertinence des mesures réalisées. Cependant, il n'est pas nécessairement disponible, par exemple si le système est en cours de conception.
- un simulateur logiciel. Il doit modéliser précisément le matériel et garantir une exécution fidèle « au cycle près ».

Déterminer le *WCET* par mesures nécessite de disposer du jeu de test qui conduit au temps d'exécution maximum. Différentes stratégies sont envisageables pour satisfaire cette exigence :

- utiliser tous les jeux d'entrées possibles.
  Pour un programme simple, l'utilisateur pourra peut-être fournir ces jeux de tests ; pour un programme plus complexe, on tentera de les générer automatiquement. [WMMR05] propose une méthode de génération de jeux de tests couvrant tous les chemins possibles d'un programme.
  Cette alternative passe cependant mal à l'échelle. En effet, pour un programme un peu complexe avec de multiples paramètres d'entrée, le nombre de jeux de tests à générer peut être rédhibitoire.
- connaître le jeu de test qui fournit de manière sûre le temps d'exécution le plus long.
  Cette solution est applicable pour un programme assez simple pour lequel l'utilisateur pourra identifier le jeu de test pire cas.
- générer un ensemble de jeux de test.
  A partir d'algorithmes évolutionnistes [WM01] ou d'après l'algorithme du recuit simulé [TCM98], on cherche à obtenir des jeux de test qui conduiront au pire temps d'exécution.
  [WM01] s'appuie par exemple sur un ensemble de jeux de test de base, sur lequel est pratiquée une sélection des jeux satisfaisant au mieux les propriétés désirées (dans notre cas, les temps d'exécution le plus long). Ceux-ci sont ensuite utilisés pour générés une nouvelle « population » par croisement et mutation. Ce procédé est ensuite appliqué itérativement.
  Dans les travaux de [TCM98], un algorithme du recuit simulé est utilisé pour trouver les jeux de test qui maximise le temps d'exécution. Le recuit simulé consiste à sélectionner de nouveaux candidats à la solution dans le « voisinage » d'un candidat courant. Les candidats répondant le mieux au problème sont toujours acceptés parmi les solutions. Quant aux candidats répondant moins bien au problème, ils peuvent être également acceptés selon une heuristique pré-définie. L'idée est de pouvoir « échapper » aux solutions locales. On peut accepter des candidats moins bons dans l'espoir qu'ils puissent nous conduire aux solutions optimales.
  Malheureusement, il n'y a pas de garantie que le jeu de test conduisant au pire temps d'exécution soit généré par de telles méthodes. Aussi, elles ne constituent pas une réponse satisfaisante pour déterminer le *WCET* d'un programme. Néanmoins, elles peuvent être mises à profit pour évaluer la correction et la pertinence d'un résultat obtenu, par exemple, par une méthode statique. En effet, la méthode statique tâche de fournir le plus petit des majorants du temps d'exécution et ces méthodes dynamiques par génération permettent d'obtenir des résultats s'approchant du temps d'exécution maximal.
  Si l'on note :
  - $WCET$ le pire temps d'exécution d'un programme P ;
  - T_{max} le temps maximum obtenu pour P à partir d'un ensemble de jeux de tests générés ;
  - $WCET_{statique}$ le pire temps d'exécution de P obtenu par méthode statique ;

  on doit avoir la relation suivante :

$$T_{max} \leq WCET \leq WCET_{statique} \tag{1.1}$$

Les méthodes dynamiques pour déterminer le pire temps d'exécution d'un programme ont donc un champ d'application limité. Hormis lorsque le jeu de test « pire cas » peut être déterminé aisément, c'est à dire pour des programmes simples, il est impossible de garantir la pertinence des résultats obtenus. Les méthodes de génération pourront néanmoins s'avérer utiles pour évaluer des méthodes « statiques » de calcul de *WCET*.

```
int impaire(int x){
    int result;
    if( x%2 ){
        result = 1;
    }
    else{
        result = 0;
    }
    return result;
}
```

Code 1.1: Fonction *impaire*

1.3 Approches statiques

Pour pouvoir déterminer un majorant du pire temps d'exécution d'un programme sans connaître a priori le jeu de test conduisant à ce résultat, ont été développés des méthodes basées sur une analyse statique du programme.

Ces méthodes se décomposent généralement en trois phases :

1. une analyse de flot ou « haut niveau » qui permet de trouver les chemins d'exécution possibles du programme ;
2. une analyse de « bas niveau » qui s'attache, par une modélisation fidèle de l'architecture sous-jacente, à capturer les effets des mécanismes matériels.
3. le calcul du *WCET* à proprement parler, qui utilise les résultats obtenus dans les étapes précédentes.

1.3.1 Analyse de haut niveau

L'analyse de haut niveau permet de déterminer les différents chemins d'exécution possibles.

Le programme est pour cela décomposé en un ensemble de blocs de base. Un bloc de base est défini comme « une suite d'instructions purement séquentielle ne contenant qu'un seul point d'entrée et un seul point de sortie ».

Deux représentations (figures 1.1) sont généralement utilisées pour étudier le flot de contrôle d'un programme :

- le graphe de flot de contrôle (figure 1.1a).
- l'arbre syntaxique (figure 1.1b).

Le graphe de flot de contrôle. Il décrit les transitions possibles entre les blocs de base du programme. La figure 1.1a représente le graphe de flot de contrôle associé au code 1.1.

Les arcs orientés du graphe représentent des relations de précédence entre les blocs. On distingue, dans ce cas, les arcs continus associés aux transitions purement séquentielles et les arcs discontinus utilisés pour les transitions avec « saut ».

L'arbre syntaxique. Il est composé de noeuds représentant les éléments structurels du langage de haut niveau et de feuilles qui sont les blocs de base du programme. La figure 1.1b montre l'arbre syntaxique associé au code 1.1. Cet arbre est composé des quatre blocs de base de la fonction *impaire* : $Bloc_1$, $Bloc_2$, $Bloc_3$, $Bloc_4$ et des noeuds : SEQ et IF.

Le noeud SEQ permet la mise en séquence de ses sous-arbres fils.

Le noeud IF permet de représenter une structure conditionnelle du type $if..then..else$. On trouve ainsi 3 sous-arbres fils qui sont chacun associés respectivement à la condition ($Bloc_1$), aux conséquents si la condition est vraie ($Bloc_2$) et si elle est fausse ($Bloc_3$).

Des informations viennent généralement compléter ces représentations et permettent de limiter le nombre de chemins possibles. Elles permettent notamment de borner le nombre d'itérations des boucles, d'exclure certains chemins infaisables sémantiquement [PK89]. Plusieurs moyens ont été

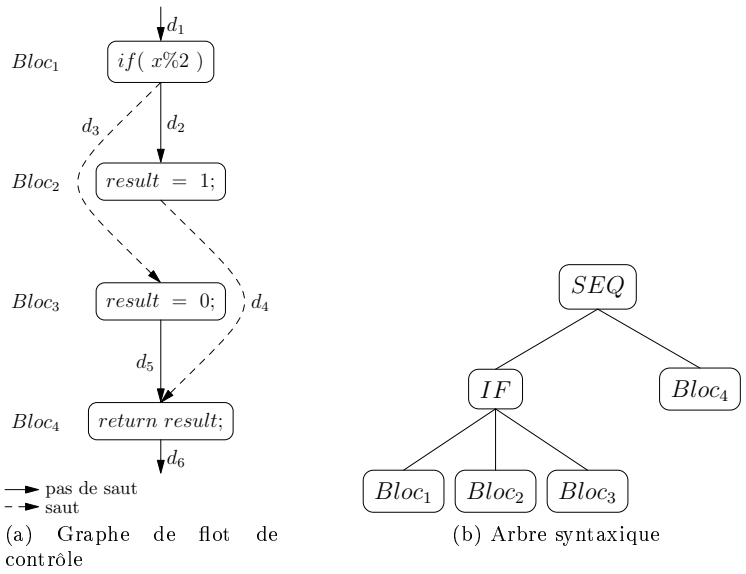


FIG. 1.1: Représentations de la fonction *impaire*

proposés pour obtenir ces informations, allant des annotations de l'utilisateur à des techniques d'analyse automatique basées par exemple sur l'interprétation abstraite [GE98].

1.3.2 Analyse de bas niveau

L'analyse de bas niveau travaille au niveau du code assembleur et utilise les caractéristiques de l'architecture cible pour calculer le pire temps d'exécution. Cette analyse est complémentaire de l'analyse de haut niveau. Pour une version naïve d'un estimateur de *WCET*, cette analyse consisterait à déterminer les temps d'exécution des blocs de base comme s'ils étaient exécutés seuls. Nous verrons par la suite que cette analyse est plus complexe que cela et que les mécanismes mis en jeu dans les processeurs modernes exigent une modélisation plus fine.

1.3.3 Calcul du pire temps

Après l'analyse de haut niveau et de bas niveau, on calcule le pire temps d'exécution. On note x_i le nombre de fois que le bloc de base B_i est exécuté (obtenu par analyse de haut niveau) et c_i le temps d'exécution de ce bloc dans le pire cas (obtenu par analyse de bas niveau). Dans ce cas, le calcul du *WCET* pour un programme ayant N blocs de base consiste à maximiser l'expression linéaire suivante :

$$\sum_{i=1}^{N} c_i x_i \tag{1.2}$$

La méthode d'énumération implicite des chemins (*Implicit Path Enumeration Technique* ou *IPET*) consiste à exprimer sous formes de contraintes linéaires la structure du graphe de flots de contrôle d'une part, et les résultats de l'analyse de haut niveau d'autre part [LM97]. L'expression (1.2) peut ensuite être maximisée, en tenant compte des contraintes exprimées, à l'aide d'un solveur de programmes linéaires en nombres entiers (*Integer Linear Programming*).

Un problème de programmation linéaire en nombres entiers (PLNE) est un programme linéaire, c'est-à-dire une fonction objectif linéaire à maximiser ou minimiser, sous des contraintes linéaires, dans lequel il y a la contrainte supplémentaire que les variables sont entières.

Si l'on considère le graphe présenté en figure 1.1a, l'équation 1.3 montre un exemple de contrainte qui peut être extraite.

$$x_1 = d_1 = d_2 + d_3 \tag{1.3}$$

Cette contrainte exprime que le nombre d'exécutions du bloc 1 (x_1) est égal au nombre d'exécutions de l'arc 1 (d_1) ainsi qu'à la somme des nombres d'exécutions des arcs 2 et 3 ($d_2 + d_3$).

Si, de plus, l'analyse de haut niveau a déterminé que la condition p (figure 1.1a) est vraie une fois sur dix, on peut également écrire :

$$10\ d_2 = x_1 \tag{1.4}$$

Outre la méthode *IPET*, on peut citer d'autres techniques permettant le calcul du *WCET* telles que l'utilisation de l'algorithmique des graphes ou l'exploitation de l'arbre syntaxique [PK89].

Cette partie vient de dégager les grandes étapes du calcul du pire temps d'exécution. Cependant, quelques mécanismes matériels, ignorés jusqu'à présent dans cette étude, viennent compliquer sensiblement le calcul.

Chapitre 2

WCET pour architectures modernes

Les architectures dont disposent les processeurs modernes ont introduit des mécanismes qui améliorent leur performance, mais qui complexifient le calcul du *WCET*. On présentera ici les mécanismes architecturaux et leurs effets qui exigent d'être pris en compte pour la détermination du pire temps d'exécution.

2.1 Les sources de complexité des microprocesseurs

Différentes techniques ont été développées pour améliorer les performances des microprocesseurs. Ainsi, depuis leur apparition dans les années 1970, leur performance n'a cessé de croître.

Ce gain en performance est dû en particulier à l'introduction de mécanismes architecturaux, qui complexifient significativement le calcul du pire temps d'exécution.

2.1.1 Le *pipeline*

Plutôt que d'exécuter les instructions de manière purement séquentielle, les processeurs utilisant un pipeline permettent aux exécutions de plusieurs instructions de se chevaucher (*overlap*). Ainsi, on peut atteindre théoriquement un débit d'une instruction exécutée par cycle processeur. Cette technique conduit à décomposer l'exécution d'une instruction en différentes étapes appelées étages du *pipeline*.

La figure 2.1(b) donne l'exemple d'un *pipeline* à 3 étages.

Dans une exécution non *pipelinée* (figure 2.1(a)), le temps d'exécution d'une séquence d'instructions est simple à déterminer : il suffit en effet de faire la somme des temps d'exécution de chacune des instructions. Dans le cas d'une exécution avec *pipeline* (figure 2.1(b)), il peut apparaître certaines interférences entre les instructions. On peut citer par exemple :

- le blocage (ou *stall*) d'une instruction à un étage à cause de l'indisponibilité de l'unité fonctionnelle suivante : on parle alors d'aléa structurel (*structural hazard*),
- la dépendance de données entre des instructions : on parle alors d'aléa de données (*data hazard*).

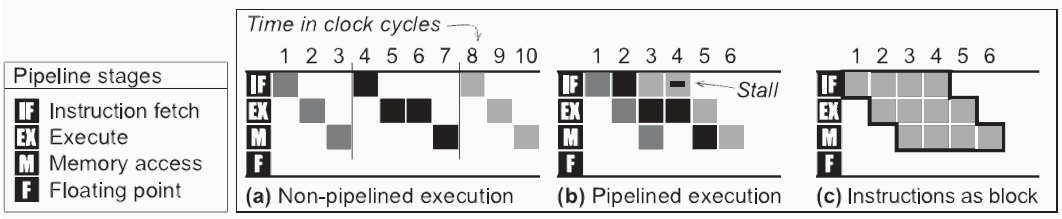


FIG. 2.1: Utilisation d'un pipeline [Eng02]

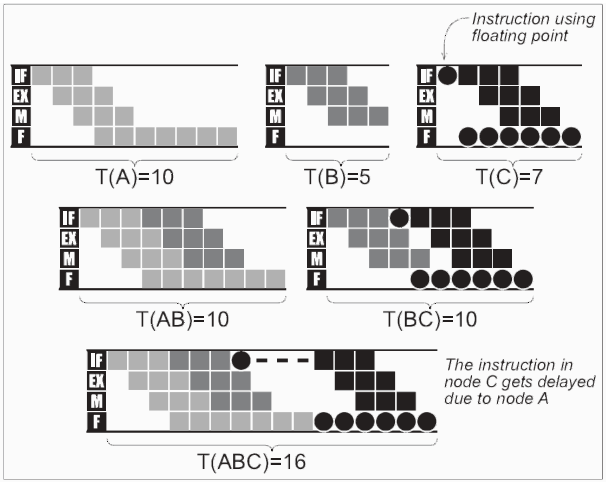


FIG. 2.2: Long effet temporel [Eng02]

On peut noter que ces aléas qui apparaissent dans un *pipeline* ne concernent pas uniquement les instructions voisines entre elles. En effet, de longs effets temporels [Eng02] peuvent impliquer des instructions non consécutives comme l'illustre la figure 2.2, et de la même manière avoir une portée supérieure au bloc.

Les processeurs superscalaires permettent en plus de lancer simultanément plusieurs instructions en multipliant les unités fonctionnelles [SS95]. Ils permettent de mettre en oeuvre du parallélisme d'instruction.

2.1.2 Exécution ordonnée et non-ordonnée

L'utilisation de processeurs superscalaires exige une bonne utilisation des différentes unités fonctionnelles. Or, souvent les instructions consécutives comportent des inter-dépendances qui peuvent ralentir l'exécution dans le *pipeline* (aléa de données). De manière générale, les blocages du *pipeline* sont autant que possible à éviter pour obtenir de bonnes performances.

C'est pourquoi les processeurs haute performance exécutent les instructions de manière non-ordonnée. Une instruction prête, c'est à dire dont les arguments et l'unité fonctionnelle sont disponibles, peut être lancée avant des instructions qui la précèdent dans la fenêtre d'exécution.

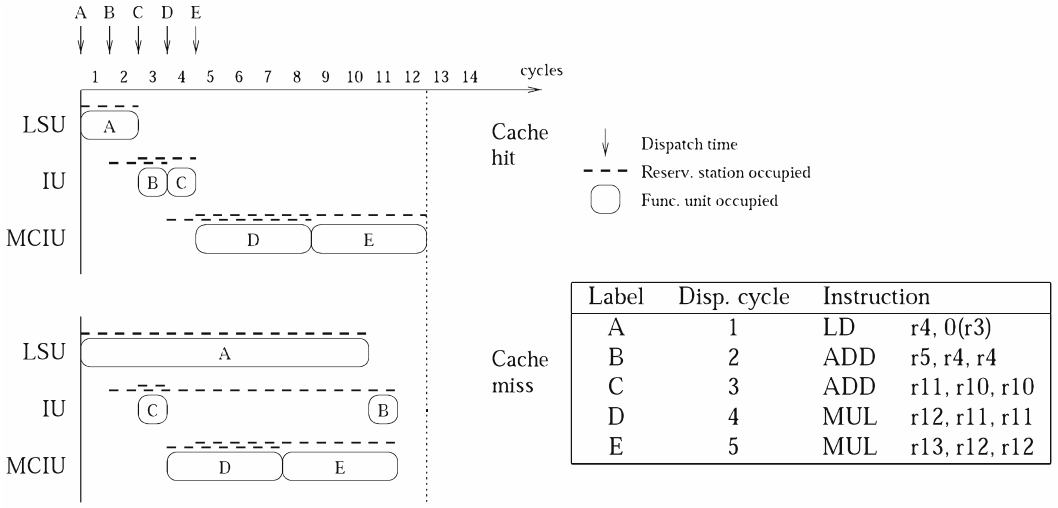
Ces processeurs à ordonnancement dynamique offrent donc une plus grande liberté dans l'exécution d'une séquence d'instructions. Néanmoins, pour le calcul du pire temps, l'exécution non-ordonnée multiplie de manière non négligeable les scénarii d'exécution possibles.

Les processeurs scalaires, ordonnés permettent de déterminer précisément le *WCET* au prix d'une performance plus réduite.

2.1.3 Mémoire cache

Les mémoires caches ont été introduites afin de réduire la latence des accès mémoire. L'idée générale est de stocker dans une mémoire intermédiaire d'accès plus rapide les données récemment ou fréquemment utilisées par le processeur pour éviter de faire des accès lents à la mémoire. Pour le calcul du *WCET*, l'accès à une donnée présente dans le cache (*cache hit*) ou non (*cache miss*) induit des latences sensiblement différentes. Ainsi, il est nécessaire de pouvoir prédire aussi précisément que possible les « défauts de cache ».

Dans un contexte de calcul du pire temps d'exécution, on pourrait vouloir considérer tous les accès mémoire comme des *cache miss*. Cependant, cette approche conduirait à une surestimation du *WCET*. De plus, l'hypothèse considérant que les pires temps d'exécution sont obtenus avec des *cache miss* peut s'avérer fausse à cause de possibles anomalies temporelles (cf section 2.2).



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	ADD	r5, r4, r4
C	3	ADD	r11, r10, r10
D	4	MUL	r12, r11, r11
E	5	MUL	r13, r12, r12

FIG. 2.3: Exemple d'anomalie temporelle dans l'accès au cache [LS99b]

2.1.4 Prédiction de branchement

Pour éviter une rupture dans le *pipeline*, les processeurs modernes mettent en oeuvre une exécution spéculative. Souvent l'exécution atteint une instruction de branchement dont la condition est une donnée non encore disponible (l'instruction de branchement dépend d'une instruction qui n'a pas encore terminée son exécution). Plutôt que d'attendre le résultat de l'instruction précédente, ce qui causerait la perte de quelques cycles, le résultat du branchement est prédit.

Différentes stratégies pour la prédiction des branchements peuvent être envisagées [SS95] :
- statiques : les prédictions sont effectuées au moment de la compilation ou par le matériel, de manière identique pour tous les branchements,
- dynamiques : les prédictions sont faites sur la base de l'historique des branchements.

Pour calculer le pire temps d'exécution, le mécanisme de prédiction de branchement doit être fidèlement pris en compte. Une mauvaise prédiction provoque, en effet, le vidage du pipeline et ralentit par conséquent l'exécution.

2.1.5 Sujet de l'étude

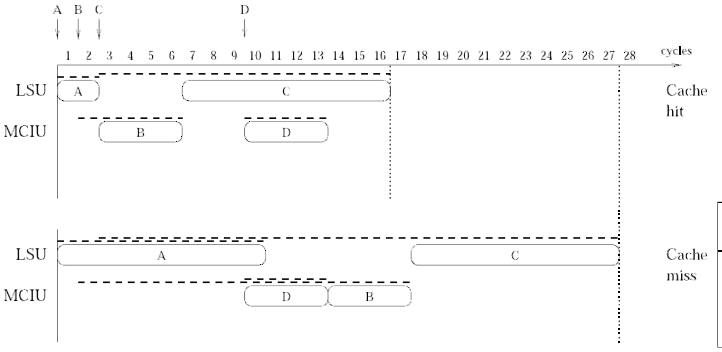
Pour ce qui concerne cette étude, on s'intéressera uniquement aux problèmes posés par le *pipeline* superscalaire à ordonnancement dynamique. On ne traitera pas directement les problèmes liés au cache et à la prédiction de branchement.

2.2 Anomalies temporelles

Pour calculer un majorant du pire temps d'exécution, certaines hypothèses « pessimistes » pourraient être envisagées afin de réduire la complexité du calcul. Cependant, certaines simplifications a priori naturelles s'avèrent conduire à des résultats erronés dans le calcul du *WCET* [LS99b].

2.2.1 Accès au cache

La latence d'une instruction d'accès à la mémoire varie fortement selon que la donnée accédée est présente ou non dans le cache. L'exemple 2.3 illustre une « anomalie temporelle » puisqu'il fournit un résultat contre-intuitif : un *cache hit* cause un temps d'exécution plus long qu'un *cache miss*.



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	MUL	r5, r4, r4
C	3	LD	r6, 0(r5)
D	10	MUL	r11, r10, r10

FIG. 2.4: Exemple d'anomalie temporelle avec un défaut de cache [LS99b]

Les instructions ($A - E$) sont traitées par trois unités fonctionnelles qui offrent des latences différentes. On distingue :

- l'unité de chargement et de stockage mémoire (*Load Store Unit : LSU*) dont la latence est variable selon la présence ou non des données dans le cache.
- l'unité de calcul entier simple (*Integer Unit : IU*), dont la latence est de 1 cycle.
- l'unité de calcul entier à latence de plusieurs cycles (*Multiple Cycle Integer Unit : MCIU*).

Dans le premier scénario, la donnée à l'adresse chargée par l'instruction A est présente dans le cache, la latence de l'instruction dans l'unité *LSU* est alors de 2 cycles.

L'instruction B, dont l'exécution dépend du résultat de A, va ainsi pouvoir démarrer au cycle 3.

L'instruction C, quant à elle, ne démarrera qu'au cycle 3.

Les autres instructions sont alors exécutées dans l'ordre du programme.

Dans le deuxième scénario, l'accès au cache pour l'instruction A est un échec, la latence de A dans l'unité *LSU* passe alors à 10 cycles.

Du coup, l'exécution de l'instruction B est retardée jusq'au cycle 11. L'étage étant ordonnancée dynamiquement, l'instruction C est exécutée au cycle 3. De la même manière, les instructions D et E s'exécutent dans le deuxième scénario un cycle plus tôt que dans le premier scénario.

Au final, si l'on compare, la durée d'exécution pour les deux scénarii, le deuxième scénario est plus court que le premier scénario de un cycle. Par conséquent, dans le cas d'un *cache miss*, le temps total d'exécution est 1 cycle plus court que dans le cas d'un *cache hit*.

2.2.2 Des délais difficiles à borner

L'ordonnancement dynamique des instructions rend difficile l'estimation des délais induits par un retard pris au sein d'une unité fonctionnelle.

L'exemple 2.4 montre comment un défaut de cache, qui implique une latence supplémentaire de 8 cycles dans l'unité LSU par rapport à un *cache hit*, provoque un temps d'exécution, pour la séquence étudiée, supérieur de 11 cycles au temps sans le défaut de cache.

L'ordonnancement dynamique des instructions peut donc provoquer des retards difficilement évaluables a priori car spécifiques à une séquence donnée et à ses différentes contraintes (dépendances de données, disponibilité des ressources).

Chapitre 3

Modélisations pour le calcul du *WCET*

Plusieurs modélisations ont été proposées pour capturer les propriétés temporelles des architectures modernes. Après un rapide aperçu, de quelques méthodes existantes, on s'attardera sur le modèle proposé par [LRM06]. On verra ensuite un modèle « héritier » qui est à la base des travaux réalisés.

3.1 Différentes approches pour le calcul du *WCET*

L'analyse de bas niveau d'un programme doit s'appuyer sur les caractéristiques du matériel cible. Pour évaluer le temps d'exécution d'un bloc de base, il est en effet indispensable de refléter fidèlement le comportement des différentes unités du processeur.

Plusieurs approches ont été développées pour fournir un cadre à l'analyse de bas niveau prenant en compte les spécificités des processeurs haute performance.

3.1.1 Interprétation abstraite

L'interprétation abstraite [CC77] consiste en l'exécution partielle d'un programme pour obtenir des informations sur sa sémantique (par exemple, sa structure de contrôle, son flot de données) sans avoir à en faire le traitement complet.

L'outil `aiT` [HF04] utilise l'interprétation abstraite pour effectuer l'analyse du *pipeline* et des caches. L'analyse du *pipeline* pour un bloc de base démarre avec un ensemble d'états déterminés grâce aux prédecesseurs du bloc considéré. Cet ensemble évolue instruction après instruction par une sorte de simulateur du processeur. A la différence d'une simulation réelle, l'exécution abstraite n'est en général pas déterministe car certaines informations déterminant l'évolution de l'exécution sont manquantes (par exemple le contenu du cache). Ainsi, l'interprétation abstraite d'une instruction peut conduire à une subdivision de l'état d'entrée en plusieurs états de sortie possibles, qui seront pris en compte pour l'instruction suivante.

A la fin de l'analyse d'un bloc de base, les états de sortie sont propagés à tous les blocs successeurs. Cette analyse est répétée pour chaque bloc jusqu'à stabilisation des ensembles des états possibles.

Ainsi, on peut obtenir par interprétation abstraite, pour chaque bloc de base d'un programme, la durée en cycles de son exécution pour tous les contextes possibles. Ces informations sont exploitées ensuite dans l'analyse des chemins par programmation linéaire en nombres entiers (section 1.3.3) pour obtenir le pire temps d'exécution du programme.

3.1.2 Exécution symbolique

L'approche proposée dans [LS99a] s'appuie une technique d'exécution symbolique des instructions du programme.

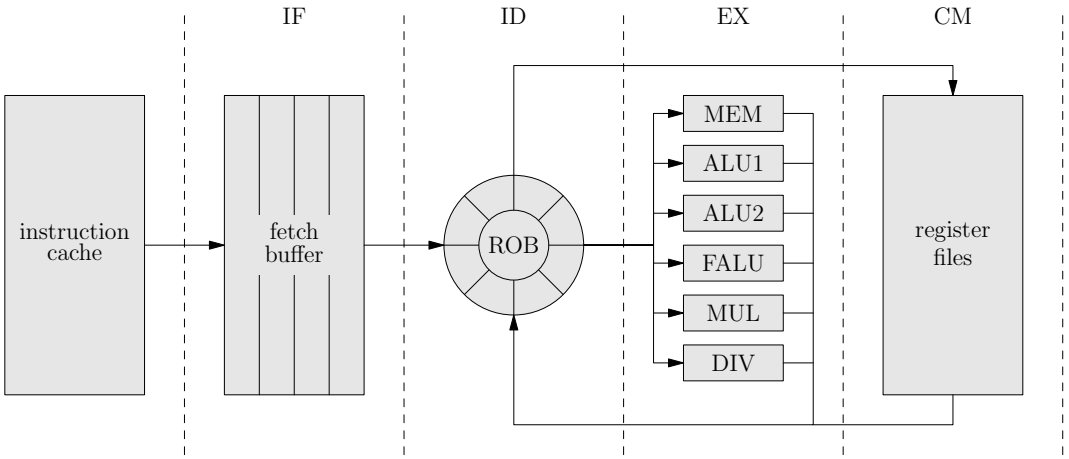


FIG. 3.1: Modèle d'un processeur superscalaire à ordonnancement dynamique

Là où un simulateur peut analyser à partir d'un jeu d'entrée un chemin possible du programme, l'exécution symbolique permet d'analyser tous les chemins possibles du programme sans énumération des différents jeux d'entrée possibles.

Pour cela, le simulateur traditionnel est étendu avec la possibilité de gérer des données d'entrée inconnues avec une sémantique particulière pour chaque instruction.

Une approche basée sur l'exécution symbolique apporte une réponse à l'identification des chemins possibles dans le programme. La prise en compte dans l'exécution symbolique des données statiques du programme permett d'exclure les chemins infaisables sémantiquement, de plus elle peut apporter quelques réponses sur le bornage des itérations des boucles.

Cependant, ce type d'approche conduit à une explosion du nombre de chemins à analyser, c'est pourquoi une stratégie pour fusionner des chemins est également proposée. Cette opération a néanmoins le désavantage d'ajouter de l'indéterminisme au chemin résultant de l'union de deux autres. La politique de « fusion » des chemins doit donc être choisie avec précaution : si la fusion est faite trop souvent, il y aura moins de chemins à analyser, par contre, les informations contenues dans les chemins fusionnés risquent d'être moins pertinentes.

3.2 Modèle processeur

Le modèle processeur proposé est « théorique » car il ne correspond pas directement à un processeur existant. Néanmoins, il est suffisamment générique pour que le discours puisse être adapté à un processeur réel moyennant quelques aménagements.

On s'appuiera donc sur un processeur superscalaire à ordonnancement dynamique. Le schéma 3.1 représente le modèle adopté.

Le *pipeline* est constitué de quatre étages.

1. Récupération de l'instruction ou *Instruction Fetch* (`IF`). Cet étage s'occupe d'amener les instructions du cache d'instruction vers la file d'instructions. Les instructions y sont placées dans l'ordre du programme. On considèrera une file de taille 4, avec au plus deux instructions chargées par cycle.

2. Décodage de l'instruction ou *Instruction Decode* (`ID`). Cet étage décode les instructions et les dispose dans le tampon de ré-ordonnancement (*re-order buffer* ou `ROB`). Les instructions sont décodées dans l'ordre du programme. La taille du `ROB` est fixée à 8 entrées. Les instructions sont présentes dans le `ROB` après le décodage (`ID`), jusqu'au retrait (étage *Commit* ou `CM`). Au plus deux instructions peuvent être décodées et placées dans le `ROB` par cycle.

3. Exécution (*Execute* ou `EX`). Lorsqu'une instruction dispose de tous ses opérandes et que l'unité fonctionnelle, dont elle a besoin, est libre, elle peut être lancée (*issued*) vers l'unité

fonctionnelle (*Functional Unit* ou `FU`) cible. Si plusieurs instructions correspondant à une `FU` peuvent être lancées, la première d'entre elles selon l'ordre du programme sera lancée. Au plus, deux instructions peuvent être lancées depuis le tampon de ré-ordonnancement vers les unités fonctionnelles à chaque cycle.
L'étage d'exécution (`EX`) est ordonnancé dynamiquement et ainsi, une instruction peut démarrer son exécution avant une autre qui la précède dans l'ordre du programme, dans la mesure où les dépendances sont respectées.

Le modèle 3.1 possède 5 unités fonctionnelles distinctes :

- `MEM`, l'unité de gestion de la mémoire, traite les opérations de `load-store` ou lecture/écriture. Cette unité est *pipelinée*, c'est à dire qu'elle est dotée de son propre *pipeline*. On distingue pour les unités *pipelinée* la latence (lat_{res}) fixée à 2 cycles pour l'unité `MEM` et le délai d'entrée dans cette ressource (ou *Initiation Interval* noté II_{res} fixé à 1 cycle.
- `ALU`, l'unité de calcul arithmétique et logique entier (*Arithmetic Logical Unit*), a une latence de 1 cycle. On peut noter que le modèle 3.1 possède deux unités `ALU`.
- `FALU`, l'unité de calcul arithmétique et logique flottant a une latence de 3 cycles. Cette unité est *pipelinée* et le délai d'entrée est de 1 cycle.
- `MUL`, l'unité de multiplication entière a une latence de 6 cycles et est *pipelinée*, le délai d'entrée y est de 1 cycle.
- `DIV`, l'unité de division entière a une latence de 15 cycles.

4. Retrait ou *Commit* (`CM`). C'est le dernier étage du *pipeline*. Lorsque le *pipeline* a été parcouru entièrement par une instruction, ses résultats sont écrits dans les registres et sa place dans le tampon de ré-ordonnancement (`ROB`) peut être libérée. L'opération de retrait est effectuée dans l'ordre du programme. Ainsi, une instruction qui a terminé son exécution (`EX`) doit attendre que les instructions qui la précèdent dans le programme aient commis leurs résultats. Pour le modèle utilisé 3.1, deux instructions peuvent être retirées par cycle processeur.

Le modèle ainsi présenté offre les sources de complexités (section 2.1.1 d'un *pipeline* d'un processeur moderne.
On peut néanmoins noter quelques simplifications faites dans ce modèle :

- Tous les étages et les unités fonctionnelles du modèle 3.1 ont des latences fixes. Cette simplification a pour but la clarté de l'exposé. Les stratégies et algorithmes proposés par la suite pourront être adaptés en considérant une latence minimale et une latence maximale, plutôt qu'une seule latence.
- Le cache de donnée est supposé parfait. Cette simplification est d'une certaine manière déjà présente dans le point précédent puisque l'unité de gestion de la mémoire `MEM` est supposée avoir une latence fixe. L'étude de la modélisation des caches est absente de ce document, qui se consacre aux problèmes posés par le *pipeline*.
- La problématique de l'exécution spéculative est également ignorée. Les branchements sont donc supposés toujours prédits correctement.

3.3 Modèle de Li

[LMR04] propose un modèle pour le calcul du pire temps d'exécution pour un processeur superscalaire à exécution non-ordonnée. Ce modèle s'attache tout d'abord à modéliser les *pipelines* superscalaires à ordonnancement dynamique, puis est étendu pour intégrer les effets dûs au cache et à la prédiction de branchement.
On ne présentera ici que la partie modélisant le *pipeline*.

A partir du graphe de flot de contrôle (figure 1.1a) d'un programme, on désire calculer le pire temps d'exécution de chaque bloc de base. En effet, à partir d'un solveur de programmes linéaires

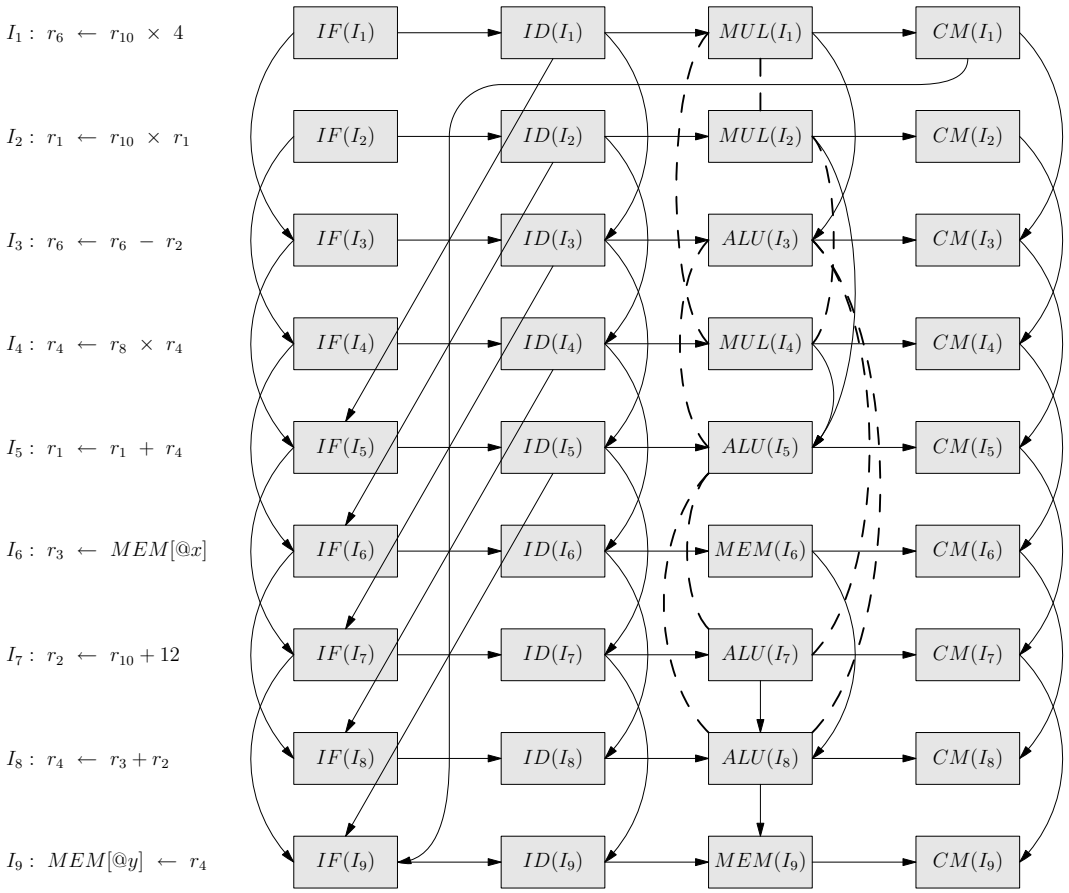


FIG. 3.2: Exemple de graphe d'exécution

en nombres entiers (*Integer Linear Programming* ou *ILP*), la question de déterminer le *WCET* du programme consistera à maximiser l'équation 3.1, où x_i est le nombre d'exécutions du bloc i est exécuté et c_i le pire temps d'exécution associé à ce bloc.

$$\sum_{i=1}^{N} c_i \cdot x_i \tag{3.1}$$

Le problème est donc de calculer le pire temps d'exécution des blocs de base du programme. La solution proposée par [LMR04] se formule en deux temps :

1. Une représentation par un graphe des différentes interactions possibles dans un *pipeline*.
2. Un algorithme itératif qui s'attache à estimer de plus en plus précisément les délais d'exécution.

3.3.1 Graphes d'exécution

Le modèle de graphe proposé dans [LMR04] s'attache à mettre en évidence les effets pouvant apparaître dans le *pipeline* et pouvant influer le délai d'exécution des instructions.

Les nœuds du graphe (figure 3.2) représentent des couples `(instruction,étage)`. En considérant une suite d'instruction $I_1 I_2 ... I_n$, le nœud `IF(`I_1`)` correspond à l'étage *Instruction Fetch* de l'instruction I_1. A chaque nœud est associée une latence, qui correspond à la latence de l'étage ou de l'unité fonctionnelle à laquelle il se rapporte.

Plusieurs types de relations peuvent exister et être représentées entre les nœuds du graphe (figure 3.2).

- dépendances entre les étages du *pipeline* d'une même instruction. Elles expriment simplement l'ordre séquentiel des étages du *pipeline*. Par exemple, l'étage `ID` suit l'étage `IF`, de même l'étage `EX` suit l'étage `DI` et précède l'étage `CM`.
- dépendances dûes aux tailles des files et tampons. Dans le modèle adopté (figure 3.1), la taille de la file d'instructions est de 4 entrées. Ainsi, il n'y aura pas de place disponible dans la file d'instructions pour I_4 tant que la fin du décodage (étage `ID`) de I_1 ne sera pas terminée, ce qui libèrera alors une place dans le *buffer* d'instruction. On observe donc (figure 3.2) un arc entre les nœuds `ID`(I_1) et `IF`(I_4).
  De même, la taille du *buffer* de ré-ordonnancement, fixée à 8, implique une dépendance entre `CM`(I_1) et `ID`(I_8).
- dépendances dûes à l'ordonnancement ordonné des instructions. Pour les étages `IF`, `ID` et `CM`, les instructions sont traitées dans l'ordre du programme. Si l'on considérait un processeur scalaire, on aurait une dépendance ou un arc entre les nœuds `IF`(I_1) et `IF`(I_2). Pour le modèle adopté (figure 3.1), le processeur est superscalaire de degré 2, on a ainsi des dépendances entre `IF`(I_1) et `IF`(I_3) et entre `CM`(I_2) et `CM`(I_4) (figure 3.2).
- dépendances de données entre les instructions. Si l'instruction I_i produit un résultat utilisé par l'instruction I_j, on aura une dépendance entre `EX`(I_i) et `EX`(I_j). Dans l'exemple 3.2, on observe un arc du nœud `EX`(I_1) vers le nœud `EX`(I_3) qui traduit que l'instruction I_3 a besoin du résultat contenu dans le registre r_6 produit par I_1.
- conflit de ressource. Ils sont représentés dans la figure 3.2 par des arcs non orientés en pointillés. Un conflit de ressource apparaît lorsque deux instructions entrent en concurrence pour accéder à une ressource ordonnancée dynamiquement. On peut observer dans l'exemple 3.2 un arc en pointillé entre les nœuds `EX`(I_1) et `EX`(I_4) qui représente le conflit possible entre les instructions I_1 et I_4 qui utilisent toutes les deux le multiplieur `MUL`.

Dans cette modélisation, il est important de distinguer la relation « conflit de ressource » des autres relations de dépendances entre nœuds.

Seuls les arcs pleins (figure 3.2) représentent une relation de précédence entre les nœuds. Aussi, lorsqu'on parle de « **chemin** » entre deux nœuds, il s'agit d'une suite d'arcs pleins, exprimant donc une dépendance.

Pour que deux instructions I_i et I_j puissent entrer en conflit, trois conditions sont nécessaires et suffisantes :

- I_i et I_j utilisent la même unité fonctionnelle ordonnancée dynamiquement.
- Il n'existe aucun chemin entre `EX`(I_i) et `EX`(I_j) ou inversement entre `EX`(I_j) et `EX`(I_i) dans le graphe. Autrement dit, il n'y aucune relation de précédence entre `EX`(I_i) et `EX`(I_j). En effet, s'il existait une relation de précédence entre ces deux nœuds, aucun conflit ne serait possible puisqu'il existerait au moins un chemin de dépendances qui assurerait que l'un soit finit avant l'autre.
- I_i et I_j doivent pouvoir être présentes simultanément dans le tampon de ré-ordonnancement. La relation 3.2 doit donc être vérifiée.

  $$|i - j| < TAILLE_ROB \tag{3.2}$$

  Cette condition signifie simplement que les instructions I_i et I_j ne peuvent pas coexister dans le `ROB` si $j > i + 8$.

Pour résoudre un conflit, deux règles s'appliquent :

1. La première des instructions (selon l'ordre du programme) en conflit accède à la ressource.
2. Il n'y a pas de pré-emption : lorsqu'une instruction accède à une ressource, son exécution va à son terme.

3.3.2 Algorithme d'estimation des délais

On désire calculer le pire temps d'exécution d'un bloc B, qui consiste en une séquence de n instructions : $I_1 I_2 ... I_n$.

Le *WCET* du bloc B est alors défini comme le temps maximum auquel le nœud `CM(`I_n`)` se termine, si l'on suppose que le nœud `IF(`I_1`)` démarre au temps zéro.

Le problème de déterminer le *WCET* de B n'est pas équivalent à celui de trouver le plus long chemin entre `IF(`I_1`)` et `CM(`I_n`)` à cause du problème posé par les conflits de ressources.

Le temps de début de réalisation d'un nœud est obtenu en prenant le temps maximum à partir duquel tous ses prédécesseurs dans le graphe auront terminé.

L'algorithme proposé par [LRM06] est basée sur une traversée du graphe d'exécution dans l'ordre topologique. En théorie des graphes, et plus spécialement en algorithmique des graphes, l'ordre topologique d'un graphe orienté acyclique est une extension linéaire de l'ordre partiel sur les nœuds déterminé par les arêtes. En d'autres termes, il s'agit d'un ordre de visite des nœuds tel qu'un nœud soit toujours visité avant ses successeurs.

Dans le graphe d'exécution considéré (figure 3.2), on sait donc que lorsqu'un nœud est visité, les temps de tous ses prédecesseurs ont été calculés. Pour chaque nœud du graphe, trois temps sont calculés :

- t^{ready}, correspond au temps où le nœud est prêt pour être réalisé.
- t^{start}, correspond au temps où le nœud démarre sa réalisation.
- t^{finish}, correspond au temps où le nœud a fini sa réalisation.

Initialement, les délais de chaque nœud sont surestimés (voir algorithme 1), de manière à maximiser les conflits possibles. Plusieurs itérations de l'algorithme 2 vont tâcher de les affiner en faisant apparaître des disjonctions entre les délais des nœuds, jusqu'à atteindre un point fixe qui marquera la fin du calcul.

Deux arguments permettent de s'assurer de la convergence de l'algorithme :

1. Au départ, chaque instruction peut potentiellement retarder les autres instructions à cause d'un conflit.
2. A chaque itération de l'algorithme 2, les instructions qui sont disjointes temporellement sont retirées de l'ensemble des instructions pouvant entrer en conflit.

L'idée est donc de calculer pour chaque nœud n les délais t_n^{ready}, t_n^{start} et t_n^{finish}.

Calcul du délai de démarrage d'un nœud. Le temps de démarrage d'un nœud i de l'instruction I_i est calculé grâce :

- au temps t_i^{ready} à partir duquel le nœud i est prêt pour être réalisé ;
- aux concurrents de i sur une ressource, c'est à dire les instructions pouvant entrer en conflit avec I_i ;
- aux **pairs** de i, c'est à dire les nœuds qui précèdent i dans le programme, qui utilisent le même étage du *pipeline* et qui peuvent démarrer leur exécution en parallèle de i.

On peut remarquer que le délai t_i^{start} est retardé au plus par un concurrent tardif. En effet, la politique d'ordonnancement utilisée (cf section 3.2) donne priorité aux instructions qui sont les premières selon l'ordre du programme. Ainsi, lorsque i est prêt à être réalisé, un concurrent qui suit i dans le programme peut déjà occuper l'unité fonctionnelle visée par i. Par contre, aucune autre instruction suivant i dans le programme ne pourra retarder la réalisation de i.

Dans un premier temps, on cherche donc à déterminer quels nœuds postérieurs à i peuvent le retarder (ligne 6) et avec quels délais.

On définit pour cela la relation booléenne *separated* entre deux nœuds. Etant donné deux nœuds i et j du graphe d'exécution, $separated(i, j)$ est `vrai` si le nœud i est prêt après la fin du nœud j ($t_i^{ready} \geq t_j^{finish}$) ou si le nœud j est prêt après la fin du nœud i ($t_j^{ready} \geq t_i^{finish}$).

Pour qu'un nœud j puisse être un concurrent tardif du nœud i, il faut que (ligne 6) :

Algorithme 1 : Initialisation des délais des nœuds du graphe d'exécution

foreach *node i in execution graph* **do**
  $t_i^{start} \leftarrow \infty$;
  $t_i^{ready} \leftarrow 0$;
  $t_i^{finish} \leftarrow \infty$;

Algorithme 2 : Calcul des délais des nœuds du graphe d'exécution

$t_{IF(I_1)}^{ready} \leftarrow 0$;
foreach *node i in execution graph topological order* **do**
  $t_v^{start} \leftarrow t_v^{ready}$;
  `/* Structural hazards */`
  Let res be the resource that i is contending for;
  $S_{late} \leftarrow \{\ j \mid j \in late_contenders(i)\ \wedge\ \neg separated(i,j)\ \wedge\ t_j^{start} < t_i^{ready}\ \}$;
  if $S_{late} \neq \varnothing$ **then**
    $\mathrm{tmp} \leftarrow \min\left(\ \max_{j \in S_{late}}(t_j^{start} + II(res))\ ,\ t_i^{ready} + II(res) - 1\ \right)$;
    $t_i^{start} \leftarrow \max(tmp, t_i^{start})$;

  $S_{early} \leftarrow \{\ j \mid j \in early_contenders(i)\ \wedge\ \neg separated(i,j)\ \}$;
  if $S_{early} \neq \varnothing$ **then**
    $\mathrm{tmp} \leftarrow \min\left(\max_{j \in S_{early}}(t_j^{start} + II(res))\ ,\ t_i^{start} + \lfloor \frac{|S_{early}| \times II(res)}{multiplicity(res)} \rfloor\right)$;
    $t_i^{start} \leftarrow \max(tmp, t_i^{start})$;

  `/* Superscalarity */`
  Let p be the degree of parallelism for the pipeline stage stg of node i;
  if *pipeline stage stg corresponds to in-order execution* **then**
    $S_{peer} \leftarrow \{\ j \mid j \in peers(i)\ \wedge\ t_j^{start} > t_i^{start}\ \}$;
    if $S_{peer} \neq \varnothing$ **then**
      $t_i^{start} \leftarrow \max_{j \in S_{peer}}(t_j^{start})$;

  else
    $S_{peer} \leftarrow \{\ j \mid j \in peers(i)\ \wedge\ \neg separated(i,j)\ \}$;
    if $|S_{peer}| \geq p$ **then**
      $t_i^{start} \leftarrow t_i^{start} + \lfloor \frac{S_{peer}}{p} \rfloor$;

  $t_i^{finish} \leftarrow t_i^{start} + lat_i$;
  foreach *immediate successor j of i* **do**
    $t_j^{ready} \leftarrow \max(\ t_j^{ready}\ ,\ t_i^{finish}\ )$;

- le nœud j soit un nœud postérieur à i et qu'ils accèdent à une même ressource ($j \in late_contenders(i)$),
- les deux nœuds ne soient pas séparés dans le sens défini par *separated* ci-dessus,
- le temps de démarrage du nœud j doit être inférieur au temps où i est prêt.

Le délai induit par un concurrent tardif j sur i est borné par le temps de début de réalisation de j (t_j^{start}) et par le délai d'entrée dans la ressource res (ou *Initiation Interval*) noté $II(res)$ (ligne 8).

De même, le délai maximum par lequel i peut être retardée par un concurrent tardif est donné par $t_i^{ready} + II(res) - 1$ (ligne 8). En effet, comme rappelé précédemment, un conflit n'est possible avec une instruction postérieure à i que si l'instruction concurrente occupe déjà la ressource au moment où i peut y accéder.

Dans un second temps, on cherche à évaluer les effets des conflits avec les concurrents précédant i dans le programme.

Un nœud j est un concurrent antérieur du nœud i si (ligne 11) :

- le nœud j est un nœud antérieur à i et accède avec le nœud i à une même ressource ($j \in early_contenders(i)$),
- les deux nœuds ne sont pas séparés dans le graphe, autrement dit $separated(i, j)$ doit être `faux`.

Le délai induit par les conflits précoces sur le nœud i ne peut pas dépasser le temps à partir duquel tous les concurrents de i ont démarré leur exécution et pour lequel le délai d'entrée dans la ressource est expiré, autrement dit :

$$t_i^{start} \leq \max_{j \in S_{early}} \left(t_j^{start} + II(res)\right) \tag{3.3}$$

Le délai induit par les concurrents antérieurs à i peut être borné grâce au nombre de concurrents détectés pour i ($|S_{early}|$), le délai d'accès à la ressource ($II(res)$) et le nombre de ressources disponibles dans le processeur ($multiplicity(res)$). Si chaque concurrent s'exécute avant le nœud i, le nœud i s'exécutera dans un délai égal à $\lfloor \frac{|S_{early}| \times II(res)}{multiplicity(res)} \rfloor$ (ligne 13).

Enfin, outre les conflits, le degré superscalaire de la ressource utilisée par le nœud i influence le temps auquel le nœud pourra démarrer.

Si le nœud i se situe à un étage qui n'est pas ordonnancé dynamiquement mais qui, au contraire, exécute les instructions dans l'ordre du programme (par exemple l'étage `DI`), les pairs du nœud i permettent de borner le temps où i va démarrer. Le nœud i ne pourra, en effet, démarrer qu'à partir du moment où tous ses pairs auront démarré (lignes 19-21). En effet, par définition, les pairs de i précèdent i dans l'ordre du programme et peuvent s'exécuter en parallèle de i.

Par contre si le nœud i utilise une ressource ordonnancée dynamiquement, il peut subir un retard dans la mesure où le nombre de ses pairs est supérieur au degré de parallélisme de l'étage du *pipeline* (lignes 25 et 26). Les calculs pour les conflits permettent d'assurer qu'au bout d'un certain délai, i pourra accéder à la ressource. Ce retard peut être borné par $\lfloor \frac{|S_{peer}|}{p} \rfloor$, où S_{peer} est l'ensemble des pairs du nœud i et p est le degré de parallélisme pour l'étage du *pipeline* où se situe i.

Calcul du délai pour qu'un nœud ait fini. Le délai pour lequel le nœud i a terminé son exécution est simplement obtenu grâce à la latence associée à i, c'est à dire la latence de l'étage ou de l'unité fonctionnelle de i. Le nœud se termine après que le nœud ait démarré et qu'il se soit exécuté (ligne 29).

Calcul du délai pour qu'un nœud soit prêt. Les successeurs immédiats du nœud i voient leur délai de disposition(t^{ready}) mis à jour si leur délai enregistré est inférieur au temps où i se termine (ligne 31).

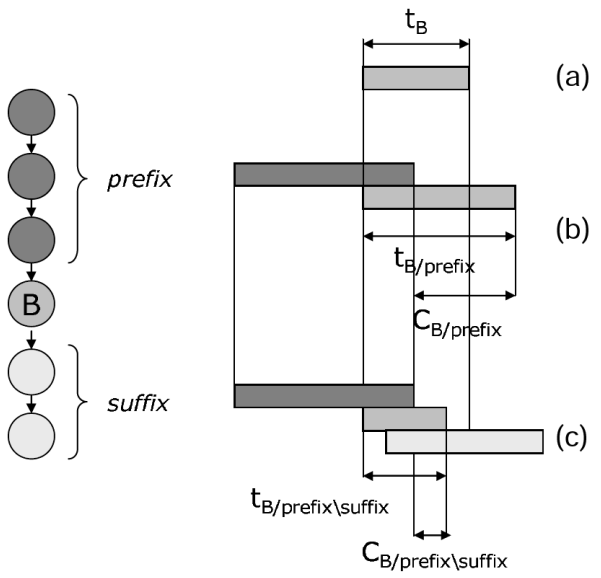


FIG. 3.3: Recouvrement du temps d'exécution des blocs

3.3.3 Calcul du *WCET*

L'algorithme 2 ignore le contexte dans lequel s'exécute le bloc dont on veut calculer le pire temps d'exécution. Aussi, [LRM06] étend cet algorithme pour prendre en compte les effets des instructions qui précèdent le bloc (le **prologue**) et des instructions qui lui succèdent (l'**épilogue**).

Cet algorithme suit les raisonnements effectués dans l'algorithme 2 mais travaille en plus sur les nœuds du prologue et de l'épilogue. Pour plus de détails sur les extensions de l'algorithme 2, on pourra se reporter à l'article [LRM06].

Pour déterminer le pire temps d'exécution d'un bloc B constitué d'une séquence de n instructions $(I_1 I_2 ... I_n)$, on souhaitait dans un premier temps le calculer comme le temps entre les nœuds `FI(`I_1`)` et `CM(`I_n`)`, autrement dit par :

$$t_B = t^{finish}_{CM(I_n)} - t^{ready}_{IF(I_1)} \tag{3.4}$$

Cependant, cette manière de calculer n'est pas très précise car l'exécution du bloc B et de son prédecesseur se recouvrent partiellement.

Le schéma 3.3 montre que le temps d'exécution d'un bloc dépend de son contexte : prologue et épilogue. De même, on voit que le temps d'exécution d'un bloc peut être vu de différentes manières : comme la différence entre la fin de l'exécution du bloc et son entrée dans le *pipeline* ou comme la différence entre la fin de l'exécution du bloc et la fin de l'exécution du bloc précédant.

C'est cette dernière définition que nous allons adopter pour éviter de compter le temps de recouvrement à la fois pour le prologue et à la fois pour le bloc.

Le temps d'exécution du bloc B sera donc défini par :

$$t_B = t^{finish}_{CM(I_n)} - t^{finish}_{CM(I_0)} \tag{3.5}$$

Finalement, pour déterminer le pire temps d'exécution d'un bloc B, on calcule le temps d'exécution de B avec toutes les combinaisons possibles de prologues et d'épilogues. Le maximum de ces temps d'exécution est utilisé comme le pire temps d'exécution du bloc B.

3.4 Les graphes paramétrés

[RS07] propose un modèle s'inspirant de la modélisation proposée par [LRM06] où l'exécution d'un programme est représentée sous la forme d'un graphe où chaque noeud correspond le

traitement d'une instruction dans un étage du *pipeline* ou d'une unité fonctionnelle.

Le contexte d'exécution d'un bloc de base y est défini grâce à un modèle paramétré qui permet d'exprimer tous les contextes d'exécution possibles du bloc. L'objectif de cette modélisation est de pouvoir calculer le moment de démarrage de chacun des nœuds du graphe d'exécution.

3.4.1 Modélisation des contextes d'exécution

Pour déterminer la date de démarrage σ_N d'un nœud N, il s'agit de déterminer la date de disponibilité du nœud N et les délais dûs aux conflits dans les étages du *pipeline* ordonnancés dynamiquement.

Les conflits internes peuvent retarder le moment de démarrage d'un nœud. Ainsi, la date de démarrage d'un nœud N (σ_N) peut être supérieur à sa date de disponibilité (ρ_N). On distingue pour les délais dûs aux conflits, les délais directs (notés ω_N) dûs aux conflits subis par le nœud N et les délais dûs aux conflits subis par d'autres nœuds et propagés au nœud N.

On peut définir la date de démarrage d'un nœud N par la relation suivante :

$$\sigma_N = \rho_N + \omega_N$$

Délais dûs aux ressources

Le contexte d'exécution d'un bloc de base est défini par les temps de libération des ressources qu'il utilise. Le terme « ressource » englobe les étages du *pipeline*, les places dans les files ou les tampons mais aussi les valeurs de registres.

Pour un processeur superscalaire de degré n, chaque étage du *pipeline* est considéré comme n ressources différentes.

On définit $\mathcal{R}$ comme l'ensemble des ressources disponibles, étendu avec une ressource virtuelle qui représente le temps à partir duquel le bloc entre dans le *pipeline*. Le contexte d'exécution est défini par un vecteur $\mathcal{A}$ de paramètres, fournissant pour chaque ressource de $\mathcal{R}$ son temps de libération :

$$\mathcal{A} = \{a^r | r \in \mathcal{R}\}$$

Les contraintes sur les temps de démarrage des nœuds N du graphe d'exécution sont exprimées avec les informations suivantes :

- $\bar{\mathcal{E}}_N$, un vecteur de booléens qui identifie la dépendance du temps de disponibilité du nœud N avec certaines ressources de $\mathcal{R}$.

$$\bar{\mathcal{E}}_N^r = \left\{\bar{e}_N^r \in \{0,1\} | r \in \mathcal{R}\right\}$$

- $\bar{\mathcal{D}}_N$, un vecteur de délais donnant la distance minimale entre la date de libération de la ressource et la date de disponibilité du nœud.

$$\bar{\mathcal{D}}_N = \left\{\bar{d}_N^r | r \in \mathcal{R}\right\}$$

Le temps de disponibilité d'un nœud N $\bar{\rho}_N$ peut être déterminé avec la formule suivante :

$$\bar{\rho}_N = \max_{r \in \mathcal{R}} \left(\bar{e}_N^r \cdot (\bar{d}_N^r + a^r)\right)$$

Le temps de disponibilité du nœud N y est défini comme le temps où toutes les ressources dont dépend N sont libres et que le délai minimal séparant N de chacune de ces ressources est respecté.

Délais directs dûs aux conflits

Un nœud qui utilise une unité fonctionnelle ordonnancée dynamiquement peut être retardé par d'autres instructions présentes dans le contexte d'exécution. Le délai subi par un nœud N est noté ω_N et peut être borné en connaissant le nombre maximum de nœuds qui peuvent entrer en concurrence avec N pour obtenir une ressource.

Délais indirects dûs aux conflits

Pour analyser les effets indirects des conflits internes, on définit :

- $\mathcal{S}$, l'ensemble des nœuds du graphe d'exécution qui utilisent une unité fonctionnelle ordonnancée dynamiquement et qui, de ce fait, peuvent être retardés à cause de conflits internes.
- $\breve{\mathcal{E}}_N$, un vecteur de booléens qui identifie la dépendance du temps de disponibilité du nœud N et de chaque nœud S de $\mathcal{S}$.

$$\breve{\mathcal{E}}_N = \{\breve{e}_N^S | S \in \mathcal{S}\}$$

- $\breve{\mathcal{D}}_N$, un vecteur de délais donnant la distance minimale entre le moment de démarrage du nœud S et le moment de disponibilité du nœud N.

$$\breve{\mathcal{D}}_N = \{\breve{d}_N^S | S \in \mathcal{S}\}$$

On note $\breve{\rho}_N$, le temps de disponibilité du nœud N relativement à ses dépendances avec les nœuds de $\mathcal{S}$. On défnit $\breve{\rho}_N$ de la façon suivante :

$$\breve{\rho}_N = \max_{S \in \mathcal{S}} \left(\breve{e}_N^S \cdot (\breve{d}_N^S + \sigma_S)\right)$$

$\breve{\rho}_N$ est donc le temps où toutes les ressources nœuds de $\mathcal{S}$ dont il est dépend ont démarré et que le délai minimal séparant N de chacune de ces ressources est respecté.

Finalement, [RS07] montre que le temps de démarrage d'un nœud N peut s'exprimer par :

$$\sigma_N = \max_{r \in \mathcal{R}} \left(e_N^r \cdot (d_N^r + a^r)\right) + \omega_N$$

où d_N^r est définit récursivement par :

$$d_N^r = \max\left(\bar{d}_N^r, max_{S \in \mathcal{S}}\left(\breve{e}_N^S \cdot \bar{e}_S^r \cdot (d_S^r + \breve{d}_N^S + \omega_S)\right)\right)$$

3.4.2 Calcul des délais

Pour déterminer les délais associés à chacun des nœuds du graphe d'exécution, un algorithme qui parcourt le graphe dans un ordre topologique est utilisé (algorithme 3).

Les vecteurs $\bar{\mathcal{E}}_N$, $\bar{\mathcal{D}}_N$, $\breve{\mathcal{E}}_N$ et $\breve{\mathcal{D}}_N$ pour chaque nœud du graphe sont initialisés de la manière suivante :

- si N est le premier nœud du graphe qui utilise la ressource r, $\bar{e}_N^r$ est initialisé à 1 et $\bar{d}_N^r$ à 0. Cela signifie que N dépend de r et que N ne pourra démarrer que lorsque la ressource r aura été libérée.
- si le nœud N appartient à l'ensemble $\mathcal{S}$ des nœuds ordonnancés dynamiquement, $\breve{e}_N^S$ est initialisé à 1 et $\breve{d}_N^S$ à 0. Cela signifie que la propagation des retards dûs aux conflits démarre à partir du nœud N.

Dans l'algorithme 3, les dépendances et les délais sont propagés d'un nœud à l'autre dans l'ordre topologique. Un nœud dépend des ressources dont au moins un des ses prédécesseurs dépend. Le délai pour chaque ressource est calculé comme le maximum des délais des nœuds précédents auxquels sont ajoutés la latence des prédécesseurs (lignes 9 et 21).

Pour chaque nœud S de l'ensemble $\mathcal{S}$, il est nécessaire de déterminer le délai ω_S induit par les conflits internes. La valeur maximale de ω_S peut être estimée de manière pessimiste en considérant que tous les nœuds dans le graphe d'exécution qui utilisent la même unité fonctionnelle peuvent entrer en conflit avec S.

Le nombre maximum de contentions subies par un noeud peut être borné grâce aux deux remarques suivantes :

- S ne peut subir une contention qu'avec au plus une instruction qui le succède dans la séquence du programme. En effet, la contention obéit à deux règles (cf section 3.2) :
  - si des instructions sont en concurrence pour accéder à une ressource au même cycle processeur, la première instruction suivant l'ordre séquentiel du programme est choisie.

Algorithme 3 : Calcul des délais des nœuds du graphe d'exécution paramétré

```
1  foreach predecessor P of node N do
2      foreach resource r in R do
3          if ē^r_P == 1 then
4              if ē^r_N == 0 then
5                  ē^r_N = 1;
6                  d̄^r_N = d̄^r_P + lat_P;
7              else
8                  if d̄^r_P + lat_P > d̄^r_N then
9                      d̄^r_N = d̄^r_P + lat_P ;
10
11
12
13
14     foreach node S in S do
15         if ĕ^S_P == 1 then
16             if ĕ^S_N == 0 then
17                 ĕ^S_N = 1;
18                 ď^S_N = ď^S_P + lat_P;
19             else
20                 if (ď^S_P + lat_P) > ď^S_N then
21                     ď^S_N = ď^S_P + lat_P ;
22
23
24
25
26
```

– une fois qu'une instruction accède à une ressource, elle ne peut être interrompue, il n'y a donc pas de préemption possible.

Ainsi, pour que S soit retardé par contention par une instruction qui lui succède dans le programme, il faut que l'instruction occupe la ressource avant l'entrée de S dans le tampon de ré-ordonnancement.

– S ne peut entrer en conflit qu'avec des instructions présentes en même temps dans le tampon de ré-ordonnancement. La taille du tampon permet donc de limiter le nombre de concurrents possibles.

De plus, le nombre de concurrents peut être réduit si on peut montrer qu'un concurrent potentiel est prêt après ou s'il termine avant le noeud S.

Le délai ω_S est calculé en divisant le nombre de concurrents trouvés avec le nombre d'unités fonctionnelles présentes.

3.4.3 Cacul du *WCET*

Lorsque les délais des nœuds ont été calculés, il est possible de déterminer en fonction du contexte, défini par un vecteur de dates de libération des ressources de $\mathcal{R}$.

Si l'on L_B le dernier nœud du bloc B, le temps d'exécution du bloc B est donné par :

$$t_B = \sigma_{L_B} + lat_{L_B}$$

Le graphe d'exécution est également étendu avec un prologue, c'est à dire une séquence d'instructions précédant le bloc B, pour pouvoir déterminer le temps d'exécution du bloc à proprement parlé. On note L_P le dernier nœud du prologue et C_B le coût de l'exécution du bloc B. On a ainsi :

$$C_B = t_B - t_P = \sigma_{L_B} + lat_{L_B} - \sigma_{L_P} - latL_P \tag{3.6}$$

Le coût du bloc B se définit comme la différence entre le temps d'exécution de B et de son prologue.

On peut observer que pour les derniers nœuds du bloc B et de son prologue, il n'y a pas de conflit possible. En effet, le dernier étage du *pipeline* s'exécute dans l'ordre du programme. On a donc : $\omega_{L_B} = \omega_{L_P} = 0$. De plus, dans le modèle proposé (section 3.2), les latences utilisées sont fixes, donc $lat_{L_B} = lat_{L_P}$. L'équation 3.6 devient :

$$C_B = \rho_{L_B} - \rho_{L_P} \tag{3.7}$$

[RS07] montre que le coût d'un bloc peut être borné en utilisant la ressource associée au dernier étage du *pipeline* noté λ. Cette ressource a deux propriétés spéciales :

– tous les nœuds situés au dernier étage du *pipeline* ont une dépendance avec cette ressource ($e^{\lambda}_{L_B} = e^{\lambda}_{L_P} = 1$), car les instructions y sont traitées dans l'ordre du programme.

– toutes les ressources de $\mathcal{R}$ sont libérées avec λ ($\forall r \in \mathcal{R}, a^r \leq a^{\lambda}$).

On peut donc borner le coût de l'exécution du bloc B comme indiqué dans l'équation ci-dessous :

$$C_B \leq \max_{r \in \mathcal{R}} \left(d^r_{L_B} - e^r_{L_P} \cdot d^r_{L_P} - (1 - e^r_{L_P}) \cdot d^{\lambda}_{L_P} \right) \tag{3.8}$$

Le pire temps d'exécution est obtenu en prenant le maximum des différences des délais calculés pour les derniers nœuds du bloc B et de son prologue P sur chaque ressource. Pour les ressources utilisées dans B et pas dans P, on utilise le délai calculé pour la ressource λ.

Chapitre 4

Transformation du code pour déterminer le *WCET*

4.1 Modifications de l'architecture et du code pour le calcul du *WCET*

Une approche originale pour calculer plus simplement le pire temps d'exécution d'un programme consiste à modifier le contexte d'exécution (architecture du processeur) et/ou le code du programme. L'idée est de rendre le programme conforme aux hypothèses permettant un calcul efficace et fiable du *WCET*.

4.1.1 Adaptation de l'architecture

Pour pouvoir calculer de manière très fiable le pire temps d'exécution, [DHPS03] propose une architecture de processeur, appelée SPEAR (*Scalable Processor for Embedded Applications in Real-Time Environments*). Cette architecture a été conçue pour répondre de manière spécifique aux contraintes des applications temps-réel, à savoir la prévisibilité temporelle. La solution proposée apparaît comme fortement contrainte :

- toutes les instructions ont un temps d'exécution constant (3 cycles),
- le *pipeline* est limité à trois étages,
- l'exécution se fait dans l'ordre.

De plus, le programme exécuté doit suivre un modèle de programmation spécifique, dit à « chemin unique ». Ce modèle s'appuie sur la possibilité de traduire un code « standard », c'est à dire avec des sauts conditionnels et des boucles, en une séquence unique d'instructions, comme illustré dans l'exemple 4.1, qui nécessite l'utilisation du `mov` conditionnel. Le jeu d'instruction utilisé doit disposer d'instructions qui permettent une exécution conditionnelle. A titre d'exemple, les processeurs ARM disposent de telles instructions.

L'approche proposée par [DHPS03] ignore les mécanismes complexes des processeurs haute-performance pour offrir un cadre contraint permettant un calcul fiable du *WCET*. Cependant, on

```
if cond                r1 = expr1;
then rr = expr1;       r2 = expr2;
else rr = expr2;       test cond;
                       movt rr, r1;
                       movf rr, r2;
```

(a) traditional (b) single-path

FIG. 4.1: Traduction d'un code traditionnel en un code à chemin unique [DHPS03]

peut regretter avec une telle solution les pertes en performance des programmes temps-réel.

Après avoir étudié les anomalies temporelles qui pouvaient apparaître dans un processeur à exécution non-ordonnée, [LS99b] propose une modification du programme intégrant des instructions de synchronisation avant et après chaque instruction de latence variable. Cette méthode, si elle a l'avantage d'éliminer les anomalies temporelles, ajoute un surcoût très important à l'exécution du programme. En effet, les instructions de synchronisation ont pour effet de vider le *pipeline*. De plus, cette méthode suppose l'existence d'une instruction de synchronisation et un contrôle des données du cache (invalidation de blocs).

[RS05] propose une modification architecturale afin de s'assurer que lorsqu'un bloc de base entre dans le *pipeline*, il ne sera pas retardé (*stall*) ou ne retardera pas d'autres instructions. Ce mécanisme est une sorte de « porte d'entrée » dans le *pipeline* pour s'assurer qu'aucun effet temporel n'est possible. Cependant, cette approche qui consiste à modifier l'architecture du processeur pour le calcul du *WCET* paraît difficile à imposer. Néanmoins, elle a le bénéfice d'exploiter les performances des processeurs modernes en les contraignant pour un contexte temps-réel strict.

Cette dernière proposition a été déclinée dans une approche travaillant au niveau du code du programme.

4.1.2 Transformation du code

La transformation de code est utilisée pour éviter les effets temporels inter-blocs en « retardant » l'entrée d'un bloc de base. Cette stratégie travaille au niveau du code du programme alors que [RS05] travaillait au niveau de l'architecture du processeur, c'est à dire le contexte d'exécution du programme.

D'autres solutions travaillant sur le code ont été proposées pour améliorer le calcul du *WCET*.

Le code du programme peut être généré pour optimiser le calcul du pire temps d'exécution. Dans l'article [ZWHM05], un algortihme pour positionner le code est présenté afin de minimiser le pire temps d'exécution d'un programme.

La solution proposée dans [RS06] consiste à ajouter du *padding* au code initial du programme afin d'espacer les différents blocs et ainsi d'éliminer les effets temporels.

Calcul du temps d'exécution d'une séquence de blocs

Le temps d'exécution d'une séquence de n blocs $N_1...N_n$ est exprimé, d'après [Eng02], par la relation suivante :

$$t_{N_1...N_n} = \sum_{i=1}^{n} t_{N_i} + \sum_{1 \leq j < n} \left(\sum_{j < k \leq n} \delta_{N_j...N_k} \right) \tag{4.1}$$

Le temps d'exécution est donc la somme des temps d'exécution de chaque bloc de base pris individuellement et des « longs effets temporels ». Les longs effets temporels représentent l'impact de l'exécution d'un bloc N_i sur l'exécution des blocs suivants $N_{i+1}...N_n$. Les longs effets temporels sont difficiles à évaluer au-delà des blocs consécutifs.

Si on élimine les longs effets temporels entre les blocs non consécutifs, l'équation 4.1 devient :

$$t_{N_1...N_n} = \sum_{j=1}^{n} t_{N_j} + \sum_{1 \leq j < n} \left(\sum_{s \in \mathcal{S}_j} \delta_{j,s} \right) \tag{4.2}$$

où $\mathcal{S}_j$ est l'ensemble des blocs successeurs du bloc N_j.

L'équation 4.2 est vérifiée si toutes les séquences de deux blocs consécutifs s'exécutent exactement comme si elles n'étaient pas précédées par d'autres blocs dans le *pipeline*.

Exemple de transformation

Pour illustrer cette approche, on s'appuie sur l'exemple donné dans [RS06]. On considère une séquence trois blocs A, B et C. Ils s'exécutent sur un *pipeline* à 3 étages avec deux unités fonctionnelles FU1 et FU2 non *pipelinées* qui ont une latence constante de 3 cycles. Les blocs A et C utilisent FU1 et le bloc B utilise FU2, comme illustrés par la figure 4.2.

Si on utilise l'équation 4.1, on a les relations suivantes :

$$
\begin{array}{lll}
t_A = t_B = t_C = 5 & & \\
t_{AB} = 6 & \Rightarrow & \delta_{AB} = t_{AB} - t_A - t_B = -4 \\
t_{BC} = 6 & \Rightarrow & \delta_{BC} = t_{BC} - t_B - t_C = -4 \\
t_{ABC} = 8 & \Rightarrow & \delta_{ABC} = t_{ABC} - t_A - t_B - t_C - \delta_{AB} - \delta_{BC} = +1
\end{array}
$$

On observe que le long effet temporel δ_{ABC} est positif, ce qui signifie que la séquence des blocs B et C est perturbé par le bloc précédent A. En effet, les blocs A et C ont recours à la même ressource FU1 et le bloc A « retarde » le passage du bloc C dans FU1.

On désire éliminer cette perturbation de manière à ce que les patrons d'exécution observés pour deux blocs consécutifs (figure 4.2) puissent être appliqués pour des séquences de blocs plus longues.

La figure 4.3 illustre le résultat souhaité. Ce comportement est obtenu en « occupant » la cellule noire par des instructions neutres.

Le code de remplissage a pour but d'espacer l'exécution de certains blocs afin d'éliminer les longs effets temporels entre ces blocs. Le code ainsi ajouté ne doit bien sûr pas affecter la sémantique du programme. Les instructions doivent être éliminées après l'étage de décodage dans le *pipeline* ; elles ne sont donc pas exécutées. La plupart des architectures dispose d'instructions `NOP` qui satisfont les caractéristiques énoncées ci-dessus.

La longueur du « code de remplissage » doit être calculée précisément par une analyse du programme initial. Le but est de minimiser la taille finale du code tout en éliminant les longs effets temporels inter-blocs.

Calcul du padding

Deux approches sont proposées dans [RS06] pour déterminer la taille du code de remplissage à insérer pour éliminer les longs effets temporels.

Une première approche consiste à analyser chaque séquence de deux blocs et à vérifier que le premier bloc n'a pas d'impact sur la disponibilité des ressources après la séquence. Ainsi, du code de *padding* est inséré de manière à ce que toutes les ressources soient disponibles après la séquence considérée de la même manière qu'après le second bloc exécuté seul, en tenant compte des éventuelles anomalies temporelles qui pourraient apparaître [LS99b]. Cette approche apparaît cependant trop contraignante car les ressources utilisées par A ne sont pas nécessairement utilisées par les blocs suivants.

Une deuxième approche proposée consiste à déterminer les ressources requises par les blocs successeurs de la séquence A-B pour savoir si l'exécution d'un bloc peut être retardée par l'exécution du bloc A. En considérant la séquence de n blocs $B_0 B_1 ... B_{n-1}$, deux cas exigent l'insertion de *padding* entre les blocs B_0 et B_1 :

1. B_i utilise une ressource qui est :
   (a) soit non disponible au moment où B_i en a besoin,
   (b) soit disponible plus tard après la séquence $B_0 ... B_{i-1}$ qu'après la séquence $B_1 ... B_{i-1}$.
2. Une ressource est disponible au bon moment pour tous les blocs de la séquence étudié mais plus tard après la séquence $B_0 ... B_{n-1}$ qu'après la séquence $B_1 ... B_{n-1}$.

Cette analyse doit prendre en considération les caractéristiques de l'architecture cible et de ses différentes unités fonctionnelles (latence, parallélisation).

	1	2	3	4	5
FETCH	A				
FU1		A	A	A	
FU2					
COMPLETE					A

	1	2	3	4	5
FETCH	B				
FU1					
FU2		B	B	B	
COMPLETE					B

	1	2	3	4	5
FETCH	C				
FU1		C	C	C	
FU2					
COMPLETE					C

	1	2	3	4	5	6
FETCH	A	B				
FU1		A	A	A		
FU2			B	B	B	
COMPLETE					A	B

	1	2	3	4	5	6
FETCH	B	C				
FU1			C	C	C	
FU2		B	B	B		
COMPLETE					B	C

	1	2	3	4	5	6	7	8
FETCH	A	B	C					
FU1		A	A	A	C	C	C	
FU2			B	B	B			
COMPLETE					A	B		C

FIG. 4.2: Patrons d'exécution des blocs A,B et C [RS06]

	1	2	3	4	5	6	7	8
FETCH	A		B	C				
FU1		A	A	A	C	C	C	
FU2				B	B	B		
COMPLETE					A		B	C

FIG. 4.3: Patron d'exécution des blocs A, B et C avec du padding [RS06]

4.2 Nouvelle approche pour la transformation de code

Une autre approche pour la transformation de code pour le calcul du pire temps d'exécution a été étudiée. Elle s'appuie sur la modélisation du contexte d'exécution d'un bloc par un graphe paramétré, modèle présenté à la section 3.4.

On essaiera de présenter dans un premier temps pourquoi il est intéressant de s'intéresser à la transformation de code pour calculer le pire temps d'exécution d'un programme.

4.2.1 Pourquoi transformer le code pour calculer le *WCET*

La transformation du code d'un programme pour le calcul du *WCET* doit répondre à deux enjeux pour le calcul du *WCET* :

1. limiter le temps de calcul du pire temps d'exécution d'un programme,
2. fournir une estimation du pire temps d'exécution du programme fiable, c'est à dire non sous-estimée et la moins sur-estimée possible.

Limiter le temps de calcul

L'analyse d'un programme complexe pour déterminer son pire temps d'exécution pose un problème combinatoire qui rend les méthodes de calcul très coûteuses en temps.

Pour que les méthodes citées précédemment en 3.3 et 3.4 puissent fournir un résultat pertinent pour le calcul du *WCET*, il est nécessaire pour chaque bloc de faire une analyse des blocs le précédant pour évaluer le contexte dans lequel il va s'exécuter dans le *pipeline*.

Cette profondeur d'analyse doit être suffisante pour garantir une couverture du tampon de ré-ordonnancement et déterminer les conflits possibles. Mais elle doit également pouvoir permettre de détecter les longs effets temporels et donc prendre en considération la latence maximale des ressources.

Dans la plupart des cas, une profondeur d'analyse de un bloc n'est pas suffisant pour garantir la pertinence du résultat. Dès lors, il faut envisager des profondeurs supérieures : deux, trois blocs ou plus ce qui posent un coût combinatoire pour le calcul qui n'est pas toujours négligeable.

L'approche proposée permet de travailler à une profondeur d'analyse fixée ce qui peut permettre de limiter le coût temporel du calcul du *WCET*.

Une estimation du *WCET*

La correction du résultat pour le domaine du calcul du pire temps d'exécution d'un programme est une exigence absolue. Parler de correction pour le *WCET* signifie que le résultat fourni ne doit pas être inférieur au pire temps d'exécution réel du programme.

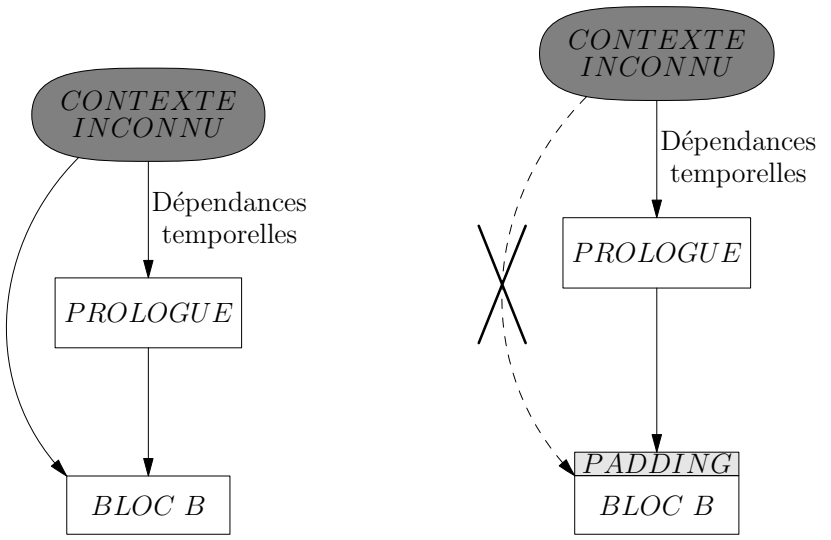
La limitation de la profondeur d'analyse pour le calcul conduit à faire apparaître un certain indéterminisme. Par exemple, si la profondeur choisie ne permet pas de limiter précisément le nombre de conflits possibles que peut subir une instruction. Cet indéterminisme doit être levé pour que le résultat fourni soit correct.

C'est pourquoi le code initial du programme peut être éventuellement modifié : il s'agit d'éliminer les indéterminismes posés par la limitation de la profondeur d'analyse. Ces transformations garantissent la correction de l'estimation fournie par cette approche.

La transformation du code du programme a un coût, il s'agit donc de trouver le compromis acceptable entre la profondeur d'analyse utilisée et l'impact que l'on peut tolérer sur le code initial.

4.2.2 Les effets temporels à éliminer

La transformation du code vise à éliminer différents effets temporels qui ne sont pas résolus faute d'une profondeur d'analyse suffisante. Les schémas 4.4 montrent l'intérêt de l'insertion de code d'espacement ou de *padding* dans le code du programme. Pour une profondeur d'analyse donnée, tous les effets temporels ne sont pas forcément pris en compte (figure 4.4a), aussi de longs effets temporels 2.1.1 peuvent retarder le bloc étudié et fausser le calcul. L'ajout de code d'espacement



(a) Longs effets temporels (b) Longs effets temporels éliminés

FIG. 4.4: Intérêt de l'insertion de code de *padding*

(figure 4.4b permet de résoudre ce problème. Le bloc étudié ne subit plus les effets du contexte inconnu.

On distingue dans cette partie les effets temporels dûs aux aléas de structure et de données dans le *pipeline* et ceux liés aux conflits sur une ressource ordonnancée dynamiquement.

Un nœud N du graphe d'exécution peut être retardé de différentes manières :

- lorsque l'étage du *pipeline* ou l'unité fonctionnelle que doit utiliser N est occupée par un autre nœud.
- lorsque N a besoin du résultat fournit par un nœud qui n'a pas encore terminé son exécution.
- lorsqu'il n'y a plus de place dans la file ou le tampon qu'utilise N.
- lorsque l'unité fonctionnelle que doit utiliser N est ordonnancée dynamiquement et qu'il y des conflits.

Si le nœud qui cause ce retard est présent dans le prologue, cette dépendance sera détectée et les délais calculés en conséquence. Par contre, si ce nœud n'est pas présent dans le prologue, il est nécessaire d'éliminer cet effet temporel.

4.2.3 Algorithme

L'algorithme 4 se charge de déterminer la taille du code d'espacement à insérer avant un bloc B de manière à éliminer les éventuels longs effets temporels qui ne pourraient pas être détectés à cause de la profondeur d'analyse choisie.

Pour chacun des prologues possibles du bloc B, on détermine l'espacement (*padding*) nécessaire. Le maximum de ces espacements calculés est utilisé pour effectivement « espacer » le bloc B.

Chaque nœud du bloc B est analysé (ligne 4). Ceux qui utilisent une ressource pour la première fois (ligne 8), c'est à dire les nœuds de B qui utilisent une ressource qui n'a pas déjà été utilisée dans le prologue, peuvent éventuellement être retardées par des instructions précédant le prologue.

L'espacement nécessaire pour « isoler » le nœud de tout effet extérieur à cause de cette ressource peut être estimé (ligne 9).

Le délai d'entrée dans une ressource res II_{res} donne l'attente maximale que peut subir un nœud à cause de cette ressource.

L'expression $\frac{size(P)+index_B(i)}{width(IF)}$ donne le nombre de cycles minimum séparant le début du prologue de l'instruction i. En effet, à partir du prologue, $size(P) + index_B(i)$ instructions doivent entrer

Algorithme 4 : Calcul du padding d'un bloc B

1 $\mathcal{P}_{depth} \leftarrow \{prologues\ of\ B\ built\ upto\ the\ specified\ depth\}$;
2 $padding \leftarrow 0$;
3 **foreach** *prologue* P *in* $\mathcal{P}$ **do**
4 **foreach** *node* n *in* B **do**
5 Let i be the instruction associated to node n;
6 `/* Compute padding for structural hazard */`
7 **foreach** *resource* res *in* $\mathcal{R}$ **do**
8 **if** $\bar{e}_n^{res} == 1 \wedge d_n^{res} == 0$ **then**
9 $padding \leftarrow \max\Big(\ padding\ ,\ II_{res} - \big(\frac{size(P)+index_B(i)}{width(IF)} + cost(IF(i) \rightarrow n)\big)\ \Big)$;
10
11
12 `/* Compute padding for structural hazard */`
13 **if** n *stage is dynamically scheduled* **then**
14 $padding \leftarrow \max\Big(\ padding\ ,\ size_{ROB} - \big(size(P) + index_B(i)\big)\ \Big)$;
15
16
17

dans le *pipeline* avant que i ne puisse y entrer. Sachant que $width(IF)$ instructions entrent dans le *pipeline* chaque cycle, il y a au moins $\frac{size(P)+index_B(i)}{width(IF)}$ cycles séparant le contexte précédant le prologue et l'entrée de l'instruction i.

Les instructions du prologue et du bloc B précédant i jouent donc aussi le rôle de code d'espacement car ils séparent l'instruction i des autres instructions du contexte inconnu qui peuvent éventuellement la retarder.

De plus, il y a un certain délai minimum entre l'entrée de l'instruction i dans le *pipeline* et l'arrivée à l'étage utilisant la ressource pouvant subir un effet temporel long. L'expression $cost(IF(i) \rightarrow n)$ représente ce délai entre les nœuds `IF(i)` et `n`. Entre l'entrée de l'instruction i dans le *pipeline* et le besoin de la ressource, il y a $cost(IF(i) \rightarrow n)$ cycles qui peuvent être considéré comme un « espace » temporel séparant l'instruction i des longs effets temporels.

Le délai d'entrée II_{res} dans la ressource res peut prendre différentes significations selon le type de ressource considéré :

- pour une unité fonctionnelle non *pipelinée*, il s'agit de la latence de l'étage ou de l'unité fonctionnelle.
- pour une unité fonctionnelle *pipelinée*, il s'agit du délai d'entrée dans l'unité fonctionnelle.
- pour une place dans une file ou un tampon, il s'agit du délai pour que la place se libère.
- pour la valeur d'un registre, il s'agit du délai maximum pour que les unités pouvant produire le résultat aient terminé.

De plus, l'espacement doit pouvoir éliminer d'éventuels concurrents pour l'accès à une ressource ordonnancée dynamiquement (ligne 13). L'espacement nécessaire (ligne 14) pour limiter les concurrents à ceux présents dans le prologue et dans le bloc B est calculé grâce à la taille du tampon de ré-ordonnancement. En effet, deux instructions i et $i + size_{ROB}$ ne peuvent pas coexister dans le tampon de ré-ordonnancement. Aussi, l'espace nécessaire pour éliminer les conflits externes est donné par la différence entre la taille du tampon de ré-ordonnancement et le nombre d'instruction entre le prologue et i (ligne 14).

4.3 Evaluation

4.3.1 Méthodologie

L'évaluation de l'algorithme proposé s'est appuyé sur un environnement dédié au calcul du *WCET* : OTAWA et sur des jeux de test dédiés au domaine du temps-réel.

Matériel

Les évaluations ont été menées sur un ordinateur disposant d'un processeur *Intel Pentium 4* à deux coeurs cadencés à 3 GHz, et disposant de 1 Go de mémoire DDR SDRAM.

OTAWA

OTAWA (*Open Tool for Adaptative WCET Analysis*) [CS06] est un *framework* développé pour permettre le calcul du pire temps d'exécution d'un programme.

OTAWA permet d'adopter une approche modulaire pour le calcul du pire temps d'exécution d'un programme en facilitant l'utilisation et la combinaison de différentes analyses. Il intègre, en outre, un support de base pour la méthode IPET et dispose d'un moteur ILP pour résoudre les systèmes d'équations linéaires entières.

Les tests ont été effectués en utilisant la méthode IPET au cours de laquelle le code à évaluer a été transformé selon l'algorithme 4.

Jeux de test

Les résultats présentés ont été obtenus à partir de plusieurs jeux de test, extraits d'un ensemble de benchmarks, utilisés dans le cadre des recherches sur le pire temps d'éxécution.

Ils font partie de la suite de jeux de tests temps-réel SNU (`http ://archi.snu.ac.kr/realtime/benchmark/`). Ces tests, écrits en C, sont issus de programmes de calcul numérique ou d'algorithmes utilisés dans les processeurs de signal numérique (DSP).

Le tableau ci-dessous les présente rapidement.

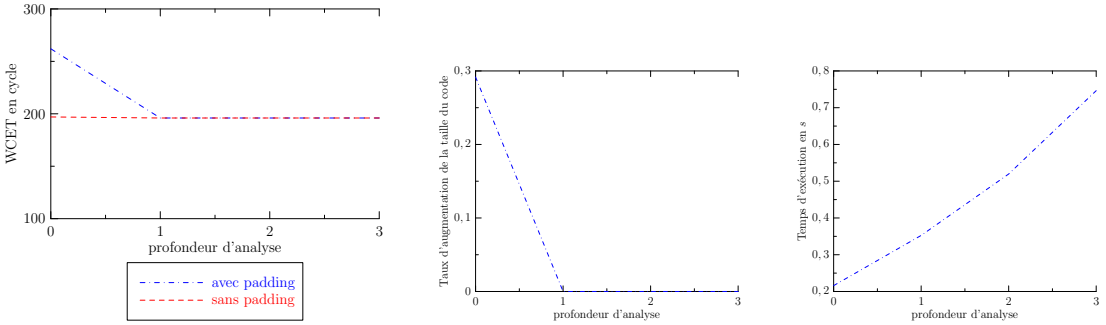
`fibcall`	Calcul d'une série de Fibonacci.
`insertsort`	Algorithme de tri par insertion.
`minver`	Inversion d'une matrice de flottants.
`bs`	Recherche binaire dans un tableau.
`fft1`	Calcul de la transformée de Fourier rapide.
`jfdctint`	Implémentation entière de la transformation en cosinus discrète.

4.3.2 Résultats expérimentaux

Les graphes 4.5 à 4.10 présentent, pour chacun des tests utilisés :

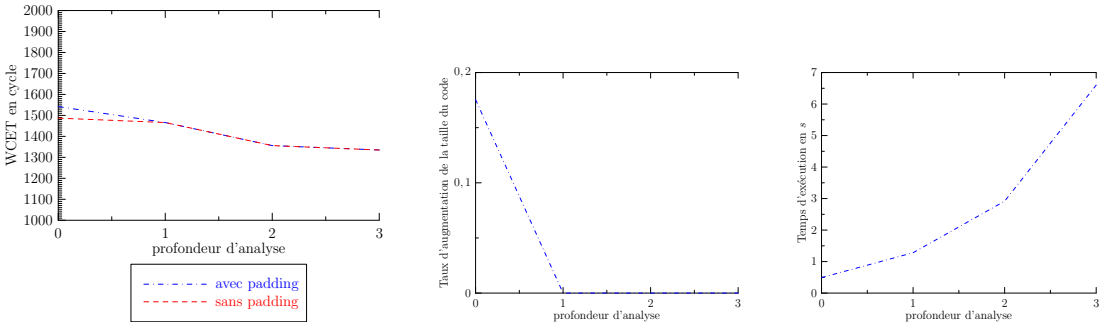
- l'évolution du calcul du pire temps d'exécution suivant la profondeur d'analyse adoptée. Figurent sur les graphes (a), la courbe représentant le *WCET* déterminé après la transformation du code et la courbe représentant le *WCET* du code d'origine (calculé avec la méthode des graphes paramétrés).
- l'évolution de l'impact du code d'espacement dans sur le code initial. Les graphes (b) représentent le taux d'augmentation de taille du code après application de l'algorithme ajoutant du code de *padding*.
- l'évolution du temps pour calculer le *padding* et évaluer le *WCET* suivant la profondeur de l'analyse dans les graphes (c).

Les allures des différentes courbes représentées sont similaires d'un jeu de test à l'autre. Elles permettent d'observer les caractéristiques de la transformation ajoutant du code d'espacement.



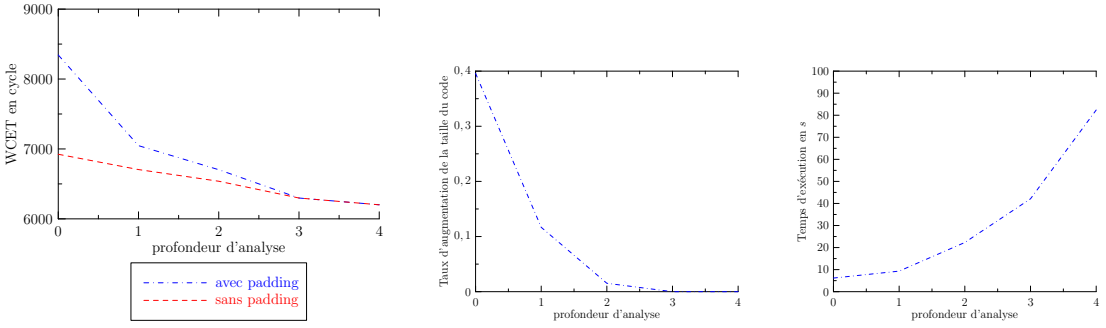
(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.5: Jeu de Test *fibcall*



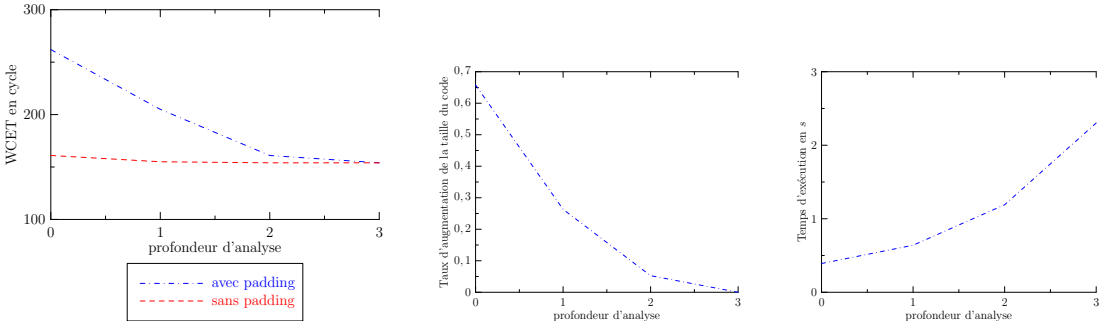
(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.6: Jeu de test *insertsort*



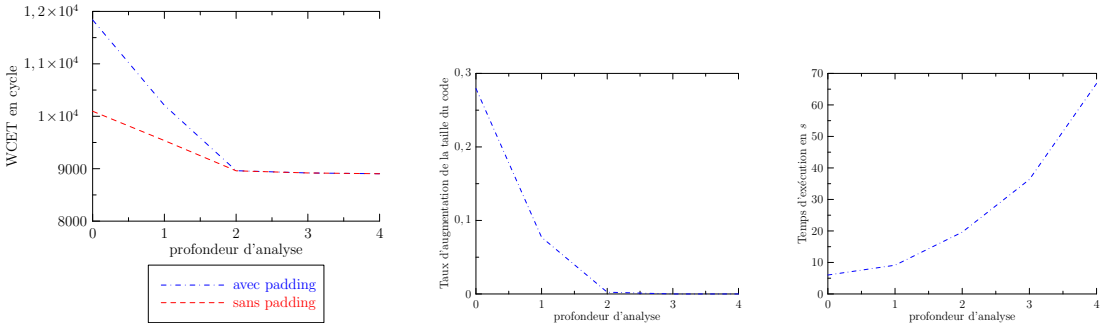
(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.7: Jeu de test *minver*



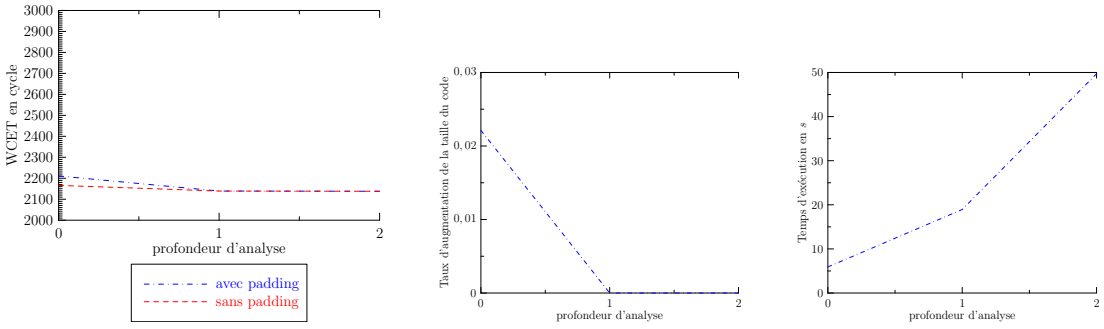
(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.8: Jeu de test *bs*



(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.9: Jeu de test $fft1$



(a) Pire temps d'exécution (b) Impact du code de padding (c) Temps de calcul

FIG. 4.10: Jeu de test $jfdctint$

Le pire temps d'exécution

Le pire temps d'exécution du code modifié par l'algorithme 4 converge vers celui du code d'origine calculé à partir de l'algorithme basé sur les graphes d'exécution paramétrés (figures (a)).

En effet, plus la profondeur utilisée pour le calcul du code d'espacement est grande, moins il est nécessaire d'ajouter du code de *padding* (figure 4.7b).

Ce résultat était attendu. L'un des objectifs de l'ajout du code d'espacement est de justement pouvoir limiter la profondeur de l'analyse effectuée. Aussi, si l'on ne limite pas la profondeur d'analyse, le code de *padding* est inutile.

Ceci est confirmé par les graphes (b) : à partir d'une certaine profondeur (différente selon les jeux de test), la taille du code d'espacement est nulle, autrement dit il n'y a plus de code d'espacement.

Impact du code d'espacement

L'impact du code de *padding* pour une profondeur donnée est très variable. Par exemple à la profondeur d'analyse 0, la figure 4.8b montre le code initial du test `bs` a subit une augmentation de près de 66% à cause du code de *padding*, alors que d'après la figure 4.10b, le test `jfdctint` augmente d'à peine 2%.

Ces résultats disparates sont naturellement dûs aux jeux de tests. Le test `jfdctint` a une structure très séquentielle avec peu de sauts et de grands blocs de base (figure 4.11). Par contre, le test `bs` est composé de petits blocs avec des sauts fréquents (figure 4.12). Ainsi pour le test `jfdctint` une faible profondeur d'analyse permet d'éliminer la potentialité d'avoir de longs effets temporels. alors que pour le test `bs`, le même niveau d'analyse conduira à de nombreux indéterminismes qu'il faudra lever en ajoutant du code d'espacement.

L'impact du code de *padding* sur ces quelques tests est relativement faible en moyenne : moins de 30% d'augmentation du code. De même, on peut observer que plus aucun code d'espacement n'est ajouté à partir d'un niveau de profondeur que l'on peut qualifier de relativement faibles pour ces jeux de test.

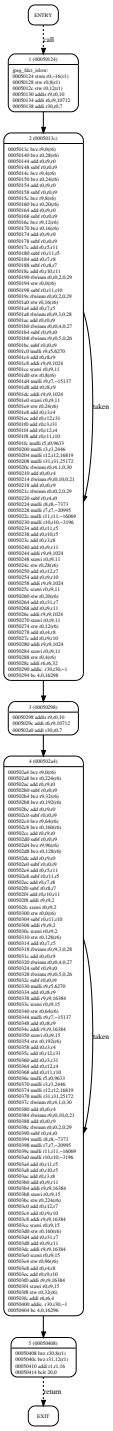


FIG. 4.11: Graphe de flot de contrôle du test *jfdctint*

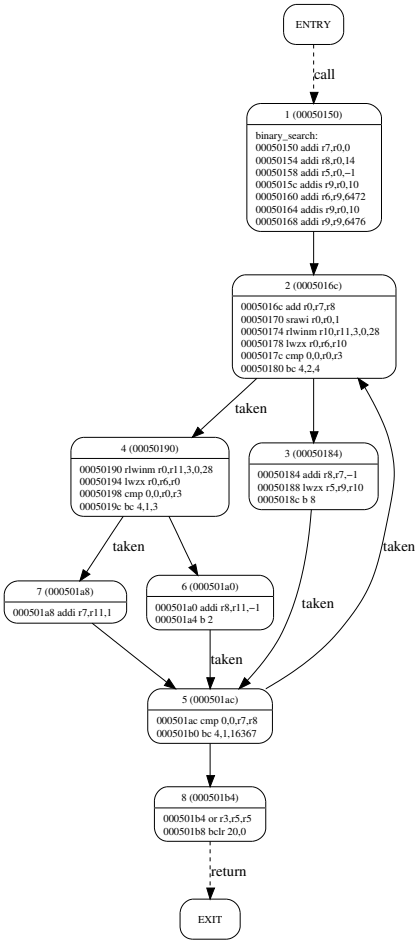


FIG. 4.12: Graphe de flot de contrôle du test *bs*

Le modèle adopté (cf 3.2) pour faire les mesures, permet en partie d'expliquer cette observation. On peut facilement se rendre compte que si la taille du tampon de ré-ordonnancement était plus grande, le taille du code d'espacement le serait aussi pour pouvoir prévenir tout risque de conflits.

4.3.3 Perspectives

Pour évaluer la pertinence des transformations effectuées, il serait pertinent d'utiliser une autre méthode de calcul du *WCET* par exemple la méthode de Li [LRM06].

Pour se donner une idée plus précise du coût de la transformation du code, d'autres architectures avec des tailles de tampon de ré-ordonnancement et des degrés superscalaires différents, pourront être utilisées pour l'évaluation.

Pour limiter la taille du code d'espacement à ajouter au début d'un bloc, un algorithme pour réordonner les instructions au sein du bloc peut être envisagé. Il s'agit, tout en respectant les dépendances logiques du programme, d'étudier les arrangements possibles des instructions de manière à minimiser le nombre d'instructions de *padding* à ajouter.

Conclusion

Modifier le code du programme dont on veut calculer le temps d'exécution pire cas est une démarche singulière. Tout en limitant le coût de calcul du *WCET*, cette approche a le mérite de garantir un résultat pertinent. En effet, la transformation du code vient compenser la limitation des calculs.

Dans ce document, nous nous sommes intéressés aux problèmes posés par les *pipelines* superscalaires à ordonnancement dynamiques et avons proposé une méthode de transformation de code pour déterminer le temps d'exécution pire cas d'un programme.

Cette méthode, basée sur la modélisation du programme en graphe d'exécution paramétré, permet de limiter le coût de calcul du *WCET* en permettant de fixer la profondeur de l'analyse. Pour garantir la correction du résultat fournit, des instructions d'espacements (par exemple des `NOPs`) sont insérés dans le code initial. Ces instructions ont pour but d'éliminer les éventuels effets temporels non pris en compte dans le calcul.

L'insertion de code de *padding* a, néanmoins quelques effets gênants, sur le programme initial. Il conduit naturellement à une augmentation de la taille du code et à ralentir son exécution. C'est pourquoi, il est essentiel de le limiter au maximum. Aussi, une stratégie de ré-ordonnement des instructions dans un bloc de base est à l'étude.

Bibliographie

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[CPRS03] Antoine Colin, Isabelle Puaut, Christine Rochange, and Pascal Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *RSTI-TSI - Temps réel*, pages 651–677, 2003.

[CS06] Hugues Cassé and Pascal Sainrat. OTAWA, a Framework for Experimenting WCET Computations. *ERTS'06*, janvier 2006.

[DHPS03] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor Support for temporal Predictability - The SPEAR Design Example. *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, 2003.

[Eng02] Jakob Engblom. *Processor Pipelines and Static WCET Analysis*. PhD thesis, Uppsala University, 2002.

[GE98] Jan Gustafsson and Andreas Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2) :61 − 74, juin 1998.

[HF04] Reinhold Heckmann and Christian Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. *Parallel and Distributed Computer Symposium*, 2004.

[LM97] Yau-Tsun Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 16(12), décembre 1997.

[LMR04] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling Control Speculation for Timing Analysis. *IEEE Real-Time Systems Symposium*, 2004.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling Out-of-Order Processors for WCET Analysis. *Real Time Systems*, 34(3), 2006.

[LS99a] Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 1999.

[LS99b] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. *Proceedings of the IEEE*, décembre 1999.

[PK89] Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2) :159–176, septembre 1989.

[RS05] Christine Rochange and Pascal Sainrat. A Time-Predictable Execution Mode for High-Performance Processors. *ACM Conference on Computing Frontiers*, 2005.

[RS06] Christine Rochange and Pascal Sainrat. Code padding to improve the WCET calculability. *14th International Conference on Real-Time and Network Systems , Poitiers*, 2006.

[RS07] Christine Rochange and Pascal Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Publication interne*, mars 2007.

[SS95] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83 :1609–1624, décembre 1995.

[TCM98] Nigel Tracey, John Clark, and Keith Mander. The Way Forward for Unifying Dynamic Test-Case Generation : The Optimisation-Based Approach. In *Internation Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, janvier 1998.

[WM01] Joachim Wegener and Frank Mueller. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. *Real-Time Systems*, 21(3) :241–268, 2001.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *EDCC*, pages 281–292, 2005.

[ZWHM05] Wankang Zhao, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by applying a WC code-positioning optimization. *ACM Trans. Archit. Code Optim.*, 2(4) :335–365, 2005.