N° d'ordre: 2584

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Antoine Colin

Équipe d'accueil : Solidor
École Doctorale : Sciences Pour l'Ingénieur
Composante universitaire : IRISA

Titre de la thèse :

Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel

soutenue le 29 Octobre 2001 devant la commission d'examen

M. :	François	Bodin	Président
MM. :	Pascal	Sainrat	Rapporteurs
	Yvon	Trinquet	
MM. :	Michel	Banâtre	Examinateurs
	Guillem	Bernat	
	Isabelle	Puaut	

Au temps heureux des cadrans solaires,
il n'y avait pas l'ombre d'une exactitude.

Albert Willemetz.

Remerciements

Ce travail de thèse a été réalisé à l'Institut de Recherche en Informatique et Système Aléatoires (IRISA), dans l'équipe Solidor.

Je tiens à remercier mon directeur de thèse, Michel Banâtre, responsable de l'équipe Solidor. Je remercie vivement Isabelle Puaut pour son encadrement, son interêt et son aide précieuse pendant les trois années de cette thèse.

Enfin, je remercie tous les membres de l'équipe Solidor pour la bonne ambiance de travail au sein de l'équipe.

Sommaire

I Obtention de temps d´exécution au pire cas : état de l'art **5**

1 Estimation de temps d´exécution au pire cas de programmes **7**
1.1 La correction des systèmes temps-réel strict 7
1.2 L'estimation du WCET : introduction 9
1.2.1 La place de l'estimation de WCET dans l'analyse temporelle de système 9
1.2.2 Estimation par test et mesure - analyse dynamique 10
1.2.3 Estimation analytique – analyse statique 12
1.2.4 Qualités relatives des estimations de WCET obtenues 13

2 L'analyse statique de WCET **15**
2.1 Les données de l'analyse statique de WCET 15
2.1.1 Langages sources 15
2.1.2 Représentations logiques 19
2.2 L'analyse statique de WCET de haut niveau 24
2.2.1 Techniques utilisant l'algorithmique des graphes (*Path-based*) 25
2.2.2 Techniques *IPET* 26
2.2.3 Techniques basées sur l'arbre syntaxique (*Tree-based*) 28
2.2.4 Comparaison des techniques d'analyses de haut niveau 29
2.3 L'analyse statique de WCET de "bas niveau" 29
2.3.1 Prise en compte des caches d'instructions 30
2.3.2 Prise en compte de l'exécution pipelinée 32
2.3.3 Intégration des techniques de prise en compte de l'architecture 35
2.3.4 Prise en compte de quelques autres éléments d'architecture 37
2.4 Récapitulatif 39

II Analyse statique de WCET *tree-based* **43**

3 Réduction du pessimisme de l'analyse par annotation symbolique des boucles **47**
3.1 Introduction 47

3.2 Les boucles non rectangulaires 48
3.2.1 Définition 48
3.2.2 Importance de la prise en compte du caractère non rectangulaire des boucles 49
3.3 Une nouvelle technique d'annotation 50
3.3.1 Annotation symbolique 50
3.3.2 Nommage relatif des expressions symboliques et pile des annotations . 52
3.3.3 Le calcul symbolique du nombre d'itérations 53
3.4 Intégration dans un analyseur à base d'arbre syntaxique 54
3.4.1 Schéma temporel intégrant la présence d'annotations symboliques . . 55
3.4.2 Exemple d'utilisation des annotations symboliques pour l'analyse statique de WCET 57
3.4.3 Évaluation du schéma temporel par un outil d'évaluation symbolique . 59
3.5 Récapitulatif 61

4 Réduction du pessimisme de l'analyse par la prise en compte d'éléments d'architecture 63
4.1 Introduction 63
4.2 Niveaux d'emboitement de blocs 64
4.2.1 Intérêt à considérer le niveau d'emboitement des boucles 64
4.2.2 Système de nommage des boucles emboitées 65
4.2.3 Transposition aux blocs conditionnels 66
4.3 Prise en compte du cache d'instructions 68
4.3.1 Types de cache d'instructions pris en compte 68
4.3.2 Les états abstraits de cache d'instructions 72
4.3.3 Estimation de la présence des instructions 78
4.3.4 Résultats de la simulation statique 82
4.4 Prise en compte d'un mécanisme de prédiction de branchement 83
4.4.1 La prédiction de branchement : introduction 83
4.4.2 Les états abstraits de cache de branchements (ABS) 85
4.4.3 Estimation du mode de prédiction des branchements 87
4.4.4 Hypothèses nécessaires à l'interprétation des catégories de branchements 89
4.4.5 Estimation de la correction des prédictions 92
4.4.6 Résultats de la simulation statique 93
4.5 La simulation statique de pipeline 94
4.5.1 Simulation de l'exécution pipelinée d'un bloc de base 94
4.5.2 Simulation de l'effet inter bloc de base du pipeline 98
4.6 Adaptation du schéma temporel aux représentations de WCET incrémentales 100
4.7 Récapitulatif 103

5 HEPTANE : un outil pour l'analyse statique de WCET 105
5.1 L'analyseur HEPTANE 105
5.1.1 Vue d'ensemble de l'analyseur 105
5.1.2 Génération des représentations logiques du programme - bloc ① 106
5.1.3 Gestion du code assembleur - bloc ② 108
5.1.4 Cœur de l'analyseur - bloc ③ 110
5.1.5 Calcul du WCET - bloc ④ 111
5.2 Estimation du WCET d'un programme 112
5.2.1 Adaptation du schéma temporel du chapitre 3 aux annotations symboliques 112
5.2.2 Exemple récapitulatif 113
5.3 Adaptabilité de l'analyseur 118
5.3.1 Modularité de l'analyseur 118
5.3.2 Modification du langage source 121
5.3.3 Modification de la syntaxe langage assembleur 122
5.3.4 Reciblage de l'analyseur 122

6 Résultats d'analyse statique de WCET 125
6.1 Méthode d'évaluation du prototype 125
6.1.1 L'exécution réelle, base de comparaison 125
6.1.2 La plate-forme de test et mesure PENTANE 126
6.1.3 Conditions expérimentales 129
6.2 Analyse statique du WCET de code simple (benchmark) 131
6.3 Analyse statique de WCET de code système 133
6.3.1 Le noyau temps réel RTEMS 134
6.3.2 Modifications du code source de RTEMS 135
6.3.3 Étude du comportement pire-cas de RTEMS 137
6.3.4 Résultats quantitatifs de l'analyse de RTEMS 140
6.3.5 Enseignements pour l'analyse statique de WCET de code système . . 141
6.4 Conclusion 143

A 149
A.1 DTD spécifiant le format XML des données d'entrées d'HEPTANE 149
A.2 Exemple de représentation XML des données d'entrées d'HEPTANE 150
A.3 Extrait de la description XML du programme du paragraphe 5.2.2, page 113 153

B 155
B.1 Code Maple de gestion de la pile des annotations 155
B.2 Code Maple de gestion des *ln-levels* 156
B.3 Code Maple de gestion des représentations de WCET 156

C **161**
C.1 Le langage de commande de Pentane 161
C.2 Les événements comptables par les MSR 163

Table des figures

1.1 Ordonnancements produits par RM et EDF pour le même ensemble de tâches périodiques . . . 8
1.2 Les différents niveaux de l'analyse temporelle . . . 10
1.3 Comparaison de deux classes de méthodes d'estimations de WCET . . . 13

2.1 Exemple de code source d'un programme analysé (langage de haut niveau) . 20
2.2 Exemple de décomposition d'un code assembleur en blocs de base . . . 20
2.3 Graphes de flot de contrôle du programmes analysé . . . 21
2.4 *T-graphs* équivalents aux graphes de flot de contrôle . . . 21
2.5 Arbres syntaxiques des fonctions *impaire* et *main* . . . 22
2.6 Relations de précédence induites de l'arbre syntaxique . . . 23
2.7 Arbre syntaxique et graphe de flot de contrôle après inclusion des fonctions appelées . . . 24
2.8 Deux schémas de compilation possibles pour une même structure de boucle . 25
2.9 Traduction du *T-graph* en un système de contraintes . . . 26
2.10 Occupation du pipeline du processeur MicroSPARC I . . . 33
2.11 Composition de descripteurs de pipeline . . . 35

3.1 Exemples de boucles dont l'obtention automatique des bornes est impossible . 48
3.2 Exemple de boucles emboitées . . . 49
3.3 Surestimation du nombre d'itérations des boucles non rectangulaires . . . 50
3.4 Trois boucles emboitées et les annotations correspondantes . . . 51
3.5 *maxiter* dépendant du chemin d'exécution . . . 52
3.6 Exemple d'utilisation de la pile des annotations . . . 53
3.7 Graphe de dépendance entre les annotations symboliques de l'exemple de la figure 3.4 . . . 56
3.8 Code source d'une fft et l'arbre syntaxique correspondant . . . 58
3.9 Graphe de dépendance entre les annotations symboliques du programme *fft* . 59
3.10 Traduction des équations du paragraphe 3.3.3 pour Maple . . . 60

4.1 Niveau d'emboitement des boucles et conflit d'accès au cache . . . 64
4.2 Arbre syntaxique et niveaux d'emboitement de boucles (*ln-levels*). . . . 65

4.3 Arbre syntaxique et niveaux d'emboitement de blocs conditionnels (*cn-levels*) 67
4.4 Niveau d'emboitement de blocs conditionnels et conflit d'accès au cache . . . 67
4.5 Trois types d'organisations du cache 70
4.6 Occupation d'une ligne de cache par deux blocs de base 71
4.7 Décomposition d'un bloc de base en iblocs 71
4.8 Structure des états abstraits de caches (ACS) 72
4.9 Calcul des ACS^{pré} 74
4.10 Calcul des ACS^{post} 75
4.11 Restriction de l'ensemble $Pred$ pour le calcul des LCS 75
4.12 Algorithme de calcul des ACS de F. Mueller [AMWH94] 76
4.13 Calcul des ACS des blocs de base d'une boucle while 77
4.14 Ordre de parcours pour le calcul des ACS dans une boucle 78
4.15 Exemple d'un ibloc classé «d'abord *miss*» 79
4.16 Calcul des sous-ensembles $\bigtriangledown$, $\triangle$ et $\blacktriangle$ de $\text{ACS}_{\gamma}^{pré}[1,0]$ 81
4.17 Impact d'une erreur de prédiction sur le contenu du pipeline 84
4.18 Historique à 4 états codé par un compteur 2 bits 85
4.19 Conflit dans le BTB 88
4.20 Schémas de compilation des structures boucle et conditionnelle 90
4.21 Détail de l'exécution d'un bloc de base dans le pipeline 95
4.22 Illustration de l'effet inter bloc de base du pipeline 99
4.23 Exécution séquentielle de deux blocs de base de *ln-levels* différents 99
4.24 Assemblage des tables de réservation de deux blocs de base 100
4.25 Détails du sous-arbre *test* d'une structure conditionnelle 103

5.1 Vue d'ensemble de l'analyseur HEPTANE 106
5.2 Génération du fichier XML représentant le programme analysé 107
5.3 Utilisation des étiquettes pour établir la correspondance entre le code source et le code assembleur 109
5.4 Le cœur de l'analyseur 110
5.5 Arbre syntaxique du programme d'exemple 114
5.6 Traduction des blocs de base en iblocs 114
5.7 Reciblage de l'analyseur par remplacement de modules et changement de description d'architecture 123

6.1 Organisation de la plate-forme de test et mesure PENTANE 127
6.2 Exemple d'utilisation interactive de PENTANE 129
6.3 Plan mémoire de PENTANE 130
6.4 Pessimisme des estimations 133
6.5 Liste des directives RTEMS analysées 135
6.6 Exemple d'exécution de _Thread_Dispatch 138

Liste des tableaux

2.1 Formules de calcul du WCET [PK89] 28
2.2 Schéma temporel étendu pour prendre en compte les caches et le pipeline . . 36
2.3 Panorama des principaux travaux de recherche concernant l'analyse de WCET 40

3.1 Formules de calcul du WCET [PK89] (avec annotations constantes) 55
3.2 Formules de calcul du WCET avec annotations symboliques 56
3.3 Annotations constantes et symboliques de *fft* ($NbSamples = 2048$) 57

4.1 Formules de calcul du WCET [PK89] (sans prise en compte de l'architecture) 101
4.2 Schéma temporel adapté aux représentations incrémentales du WCET 102

5.1 Volume de code et langages utilisés pour l'implantation d'Heptane 107
5.2 Schéma temporel adapté aux annotations symboliques 113
5.3 Système d'équations de l'exemple de la figure 5.5 117

6.1 Liste des commandes de Pentane 128
6.2 SNU Real-Time Benchmark Suite 131
6.3 Performances de l'analyse statique de WCET 132
6.4 Performances des mécanismes de prise en compte de l'architecture 132
6.5 Confrontation entre résultats de l'analyse et résultats expérimentaux 140

C.1 Liste des événements comptables (extrait du *Pentium Processor Family Developer's Manual* d'Intel) 164

Introduction

La part du logiciel ne cesse de croître dans les systèmes de transports terrestres et aériens, de production d'énergie, de télécommunications, de paiement électronique, etc. La complexité du logiciel augmente en proportion des nouvelles fonctions qui lui sont dévolues. La sécurité des hommes et des biens d'une part et la maîtrise des coûts d'autre part conduisent à accorder une attention plus grande à la vérification de ces logiciels avant leur mise en service. Ce problème devient primordial lorsqu'on considère des systèmes qualifiés de *critiques*, c'est-à-dire dont les défaillances peuvent avoir des conséquences dramatiques au niveau humain, écologique ou financier. À titres d'exemples, citons les systèmes de contrôle des centrales nucléaires, les systèmes de commande de vol ou encore les systèmes de sécurité automobile. L'importance de la vérification des logiciels se rencontre aussi dans le cas d'applications embarquées non critiques, mais dupliquées à des millions d'exemplaires, et pour lesquelles les coûts de mise à jour sont prohibitifs.

Pour toutes ces raisons, la vérification des logiciels est aujourd'hui considérée comme un enjeu crucial sur le plan économique. On distingue plusieurs types de vérification de logiciels pour plusieurs types d'erreurs. L'implémentation d'une spécification peut être incorrecte pour trois raisons principales : (i) la présence d'erreurs d'exécution, (ii) la non satisfaction des spécifications fonctionnelles et (iii) le non respect des spécifications temporelles.

Nous nous concentrons dans notre étude à la vérification du respect des spécifications temporelles.

Étant donné l'aspect critique de certains systèmes temps-réel (on parle alors de *temps-réel strict*), il est important de s'assurer du respect des contraintes de temps imposées par la spécification temporelle avant le lancement effectif du système critique. Cette preuve des systèmes temps-réel strict est apportée notamment par l'*analyse d'ordonnançabilité* [But97], qui vérifie que toutes les tâches d'un système respecteront leurs contraintes de temps.

Cette analyse requiert une connaissance du *pire* comportement temporel du système. En particulier, il est nécessaire de connaître en particulier les pires instants d'arrivées des tâches (leur *loi d'arrivée*), et les *temps d'exécution dans le pire des cas* des tâches. C'est sur l'obtention de cette dernière information que porte cette étude.

Le temps d'exécution pire cas d'un programme peut être estimé en mesurant son temps d'exécution pour un cas de test, ou bien en utilisant des méthodes d'analyse statique de

son code source. Notre approche est basée sur l'utilisation d'une méthode d'analyse statique. Cette approche a l'avantage d'être sûre, cette sûreté des estimations a une contrepartie, les estimations obtenues sont pessimistes. Ce pessimisme induit une sur-estimation des moyens matériels nécessaires au fonctionnement du système.

Les propositions de cette thèse se situent dans un cadre temps-réel strict. Nos deux premières propositions portent sur la réduction du pessimisme des méthodes d'analyse statique d'estimation de temps d'exécution pire cas. Nous proposons d'une part une amélioration de la précision de l'identification du pire chemin d'exécution par l'utilisation d'une nouvelle technique d'annotations des boucles : les annotations symboliques. D'autre part nous proposons l'amélioration de techniques de prise en compte de l'architecture matérielle existante, ainsi qu'une technique originale de prise en compte du mécanisme de prédiction des branchements. Enfin, notre dernière proposition est l'implantation des propositions précédentes dans un prototype d'analyseur statique de WCET ayant une structure modulaire. Le but d'une telle structure étant l'amélioration des possibilités d'adaptation de l'analyseur à de nouvelles architectures matérielles.

L'organisation du document est la suivante. Le chapitre 1 présente le cadre et la problématique de l'estimation de temps d'exécution au pire cas des programmes. Le chapitre 2 passe en revue les principales classes de méthodes d'estimation analytique de temps d'exécution au pire cas. Nous présentons au chapitre 3 notre proposition pour réduire le pessimisme des estimations en décrivant plus précisément le comportement dynamique du code analysé. Le chapitre 4 introduit les méthodes de prise en compte de l'architecture matérielle, qui nous permettent d'estimer précisément le comportement pire cas du matériel exécutant les programmes, et ainsi de réduire le pessimisme des estimations de temps d'exécution au pire cas de ces programmes. Nous présentons au chapitre 5 l'implantation des propositions précédentes dans un prototype d'analyseur statique de temps d'exécution au pire cas. Nous présentons également l'aspect modulaire et les possibilités d'adaptation de cet analyseur. L'utilisation de l'analyseur pour l'estimation du temps d'exécution au pire cas de différents types de code est présentée au chapitre 6. Nous y présentons la méthode utilisée pour évaluer la précision de l'analyseur, ainsi que les résultats d'analyse concernant du code simple d'une part, et le code d'un système d'exploitation temps-réel d'autre part. Enfin, nous concluons cette étude en mettant en avant les apports de notre travail et les perspectives ouvertes par ce dernier.

première partie

Obtention de temps d´exécution au pire cas : état de l'art

Chapitre 1

Estimation de temps d´exécution au pire cas de programmes

1.1 La correction des systèmes temps-réel strict

Les systèmes temps-réel se distinguent par un critère de correction plus fort que celui des autres systèmes. En effet, pour être correcte, une application temps-réel doit non seulement délivrer un résultat correct, comme dans tout autre système, mais elle doit le délivrer en respectant une certaine contrainte sur sa date de transmission (par exemple une échéance). La correction du résultat dépend donc non seulement de sa valeur mais également du moment auquel il est délivré. On parle d'erreur temporelle lorsqu'une application temps-réel, ou encore une tâche du système temps-réel dépasse l'échéance qui lui était attribuée. De façon générale, un système temps-réel se distingue d'un système traditionnel par sa capacité à garantir que l'exécution de tâches respecte certaines échéances [Sta96]. On distingue deux catégories de systèmes en fonction de la gravité de l'impact d'une telle erreur sur le système.

La première catégorie est celle des systèmes temps-réel dit “souple” ou non-critique. Dans leur contexte, une erreur temporelle a un impact sur la qualité de service du système mais ne remet pas en cause la sécurité des matériels ou des personnes impliqués. Par exemple, dans un système multimédia, une tâche de traitement vidéo peut omettre quelques images sans que la qualité visuelle n'en soit affectée.

La deuxième catégorie est celle des systèmes dits temps-réel strict (ou “dur”). Dans cette catégorie, le système doit impérativement garantir le respect des échéances fixées pour l'exécution des tâches [But97]. Cette contrainte est inhérente aux applications à sûreté critique, pour lesquelles une erreur temporelle peut avoir des conséquences catastrophiques (humaines, matérielles, financiaires, écologiques, etc.). C'est le cas par exemple des applications de contrôle de centrales nucléaires ou de pilotage de fusées. Une telle contrainte signifie par exemple réagir dans un délai maximum à la variation de la température au sein d'un réacteur.

Dans un tel système, une hypothèse forte est la correction temporelle des tâches du système, et le dépassement d'échéance d'une seule des tâches temps-réel strict peut conduire à une défaillance générale du système. Pour cette catégorie de système on cherche à avoir la certitude absolue, avant exécution, que toutes les échéances seront toujours respectées et dans tous les cas, lors de l'exécution.

Dans la suite de ce document nous nous intéressons à cette deuxième catégorie de systèmes temps-réel. Les méthodes et propositions qui y sont exposées sont du domaine des systèmes temps-réel *strict*.

Dans cette catégorie de systèmes, les tâches sont ordonnancées de manière à respecter, et ce dans tous les cas de figures, leurs échéances. Ces questions d'ordonnancement, maintenant étudiées depuis plus d'une trentaine d'années, ont donné lieu à de nombreux travaux, et à une littérature fournie dans ce domaine [LL73, Xu93, SR88, But97]. Une politique d'ordonnancement est constituée de l'algorithme d'ordonnancement d'une part et du test d'ordonnançabilité d'autre part.

L'algorithme d'ordonnancement est chargé de décider de l'ordre d'exécution des tâches. Voici deux exemples d'algorithme d'ordonnancement :

- l'algorithme EDF [LL73] - Earliest-Deadline First - accorde la priorité maximale à la tâche dont l'échéance est la plus proche,
- et pour l'algorithme RM [LL73] - Rate Monotonic - la tâche ayant la plus petite période est la plus prioritaire.

Deux tâches périodiques : échéances $= T_i$, périodes $= P_i$, WCET $= C_i$

- $T_1 = 6$, $P_1 = 6$, $C_1 = 1$
- $T_2 = 8$, $P_1 = 8$, $C_1 = 6$

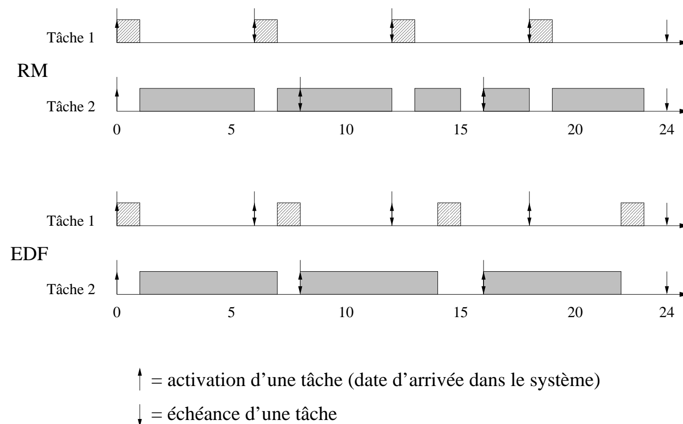


FIG. 1.1 – *Ordonnancements produits par RM et EDF pour le même ensemble de tâches périodiques*

Le test d'ordonnançabilité (appelé aussi analyse d'ordonnançabilité, ou analyse de faisabilité) permet de vérifier que l'algorithme d'ordonnancement choisi permettra à toutes les tâches de respecter leurs échéances dans tous les cas. Par exemple, le test d'ordonnançabilité associé à EDF (sous certaines hypothèses) pour un ensemble de tâches périodiques de période P_i à échéance sur requête (*i.e.* $T_i = P_i$) est le suivant [LL73] :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

où C_i est le temps maximum nécessaire au processeur pour exécuter la tâche i sans interruption. Ce test, pratiqué hors-ligne (*i.e.* avant toute exécution du système), est le garant du bon fonctionnement du système temps-réel strict.

Le test d'ordonnançabilité nécessite, entre autres, la connaissance sûre des temps maximum que nécessiteront les tâches pour s'exécuter (les C_i dans l'exemple ci-dessus). Cette caractéristique des tâches est appelée le *temps d'exécution au pire cas de la tâche*, elle est notée WCET (pour Worst Case Execution Time) dans la suite de ce document.

Malgré le fait qu'une connaissance sûre du WCET des tâches est nécessaire au test d'ordonnançabilité, la recherche d'une méthode d'obtention d'une estimation sûre de ce WCET n'a débuté qu'il y a une quinzaine d'années [KS86, PK89, Sha89]. Notre travail de thèse se situe dans le cadre de la recherche d'une méthode d'estimation du WCET des tâches, qui soit à la fois sûre (voir § 1.2.3) et précise, pour l'application des tests d'ordonnançabilité des systèmes temps-réel strict.

Dans la suite de ce document, on désignera la connaissance qu'a le système du WCET d'une tâche par "estimation du WCET" ou plus simplement "WCET" de cette tâche, et on utilisera le terme de "WCET réel" pour décrire le pire temps d'exécution effectivement nécessaire au processeur pour exécuter une tâche sans interruption.

1.2 L'estimation du WCET : introduction

Nous présentons ici les deux principales classes de techniques utilisées pour évaluer le WCET. Pour déterminer le WCET d'un programme, nous pouvons soit *mesurer* son temps d'exécution dans un environnement de test (§ 1.2.2), soit *analyser* son code (§ 1.2.3). Ces deux classes de techniques diffèrent en terme de qualité des estimations du WCET obtenues (§ 1.2.4) et en terme d'effort à fournir pour obtenir ces estimations.

1.2.1 La place de l'estimation de WCET dans l'analyse temporelle de système

La figure 1.2 présente les 2 niveaux classiques nécessaires à l'analyse temporelle d'un système temps-réel strict.

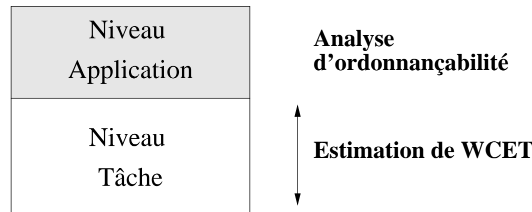


FIG. 1.2 – *Les différents niveaux de l'analyse temporelle*

- le niveau noté *estimation de WCET* sur la figure 1.2 nous donne une estimation du temps nécessaire à l'exécution d'une tâche considérée de manière isolée. C'est-à-dire que l'influence des autres tâches du système sur son temps d'exécution n'est pas pris en compte.
- le niveau supérieur est l'analyse d'ordonnançabilité. À ce niveau, plusieurs tâches s'exécutent sur le processeur. Elles sont interruptibles et sont en compétition pour l'utilisation des ressources. C'est à ce niveau que le test d'ordonnançabilité utilise les estimations de WCET du niveau inférieur.

Nous ne nous intéressons ici qu'à la partie de l'analyse temporelle qui concerne l'estimation du temps d'exécution au pire cas, et non à son utilisation par les tests d'ordonnançabilité.

Les techniques d'estimation de WCET peuvent permettre d'estimer le surcoût temporel dû aux autres activités du système pour les prendre en compte lors de l'analyse d'ordonnançabilité.

1.2.2 Estimation par test et mesure - analyse dynamique

Les techniques d'analyse dite dynamique sont fondées sur l'observation du comportement dynamique du programme dans sa globalité par exécution ou simulation pour en estimer le WCET.

1.2.2.1 Principe général

Ces techniques se basent sur les données d'entrée du programme (et non sur sa structure) pour prédire son temps d'exécution au pire cas. À partir d'un test, c'est-à-dire d'un jeu de données d'entrée provoquant un comportement particulier du programme, on mesure son temps d'exécution. Cette mesure peut être effectuée de différentes manières :

(i) L'utilisation de matériel spécialisé tel qu'un analyseur logique permet de compter le nombre de cycles nécessaires à l'exécution d'un code en environnement réel. Cette technique a l'avantage de ne pas être intrusive.

(ii) Certaines architectures (l'Intel Pentium par exemple, *cf.* 6.1.2) comportent des registres de débogage et sont programmables pour pouvoir compter différents types d'événements, et en particulier les cycles processeur.

(iii) Si on ne dispose pas d'un moyen de mesurer précisément le temps, on peut effectuer des mesures moins précises sur un grand nombre de répétitions du même test, puis en établir une moyenne. La technique dite de "*dual loop benchmark*" [AW89] consiste à mesurer le temps d'exécution d'une boucle contenant le code sous test ainsi que le temps d'exécution de la même boucle à vide pour corriger la première mesure du coût de la gestion de boucle et du code d'instrumentation.

(iv) Enfin, il est possible d'exécuter le code dans un environnement simulé. Certains simulateurs d'architectures matérielles (*cycle accurate simulator*) permettent d'obtenir de nombreuses informations pendant l'exécution du code, et, entre autres, le nombre de cycles. Cette technique nécessite un simulateur de processeur très précis non seulement au niveau fonctionnel, mais aussi au niveau temporel.

En théorie, le test dynamique est approprié pour l'analyse de WCET de code. Le temps d'exécution du programme est mesuré pour une exécution avec des données d'entrée particulières. Un outil de test idéal exécuterait le programme pour chaque ensemble possible de valeurs de ces données d'entrée et mesurerait le temps d'exécution. Mais ceci n'est pas possible en pratique car la complexité des problèmes à traiter conduit à un nombre de tests possibles extrêmement grand (problème d'explosion combinatoire). Il faut donc réduire l'espace de recherche en ne considérant qu'un sous-ensemble des tests possibles. Plusieurs approches pour réduire l'espace de recherche sont présentées ci-dessous.

1.2.2.2 Réduction de l'explosion combinatoire

Approche manuelle

L'approche la plus simple et la plus couramment utilisée est l'approche manuelle pour laquelle les tests sont élaborés manuellement en même temps que le développement du code à tester. Ces tests sont définis par des personnes ayant une connaissance en profondeur du code et qui sont donc bien placées pour choisir le sous-ensemble de tests provoquant le pire cas d'exécution. Les principaux inconvénients de cette approche sont son coût élevé de développement des cas de test (on peut estimer le coût de cette méthode à celui du test de logiciel qui dépasse souvent les 40% du coût total de développement [Pre94]) et la possibilité d'erreurs humaines dans le choix et la génération des tests.

Génération automatique de tests

Une méthode originale, présentée dans [WM01], propose la génération automatique de tests par un algorithme évolutionniste [Bac96]. Cette famille d'algorithmes, dont le représentant le plus connu est l'algorithme génétique, représente une classe de techniques de recherche basée sur le processus de sélection naturelle et la théorie de l'évolution de Darwin [Dar59]. Ces techniques de recherche sont caractérisées par un fonctionnement itératif et travaillent sur un grand nombre de solutions potentielles (une population constituée d'individus) en parallèle. Les algorithmes évolutionnistes sont particulièrement bien adaptés aux problèmes comprenant

un grand nombre de variables aux domaines de valeurs complexes.

La recherche d'une solution est basée sur trois opérations : la sélection, le croisement et la mutation. Le concept des algorithmes évolutionnistes est de faire évoluer des générations successives en sélectionnant les "meilleurs individus" et en les croisant deux à deux pour obtenir la génération suivante. Des exemples d'algorithmes de sélection et croisement sont présentés dans [WSP99].

Dans le cadre de l'analyse dynamique de WCET, les variables du problème sont les données d'entrée du programme à tester. Chaque individu représente un test, c'est-à-dire un vecteur de valeurs correspondant aux données d'entrée du programme à tester. Chaque individu est évalué en exécutant le test qu'il représente et en mesurant le temps d'exécution, puis on effectue la sélection et le croisement, puis la mutation de certains des individus de la nouvelle génération ainsi obtenue.

Le critère d'arrêt de cette évolution peut être un nombre prédéfini de générations ou encore une diminution des variations, voire une stabilisation, des résultats d'évaluation des meilleurs individus. Dans tous les cas, l'arrêt de l'algorithme ne garantit pas que le résultat final est le WCET réel. En effet, la construction d'une génération à partir de la génération précédente étant basée sur le hasard, un individu représentant le pire cas d'exécution (et donc le WCET réel) peut ne jamais apparaître.

1.2.2.3 Problème de sûreté

Le choix d'une méthode d'analyse dynamique pour estimer le WCET pose la question de la sûreté des résultats obtenus. En effet, on a vu que la seule manière sûre d'obtenir le WCET réel par test et mesure est le test exhaustif, ce qui est pratiquement impossible car l'ensemble des tests possibles est trop vaste pour que la campagne de tests ne les essaie tous. C'est pourquoi seule une petite partie de ces tests est effectivement exécutée et donne lieu à une mesure de temps. Il est alors très difficile d'être certain que le test qui permet d'exhiber le pire comportement temporel du programme ait été retenu dans ce sous-ensemble.

Un indice de cette incertitude dans l'obtention des estimations de WCET par tests et mesures est l'habitude qu'a prise le monde industriel de multiplier les résultats de ces estimations empiriques par un facteur dit de "sécurité" choisi arbitrairement.

1.2.3 Estimation analytique – analyse statique

Cette deuxième catégorie regroupe les techniques basées sur l'analyse statique de programmes. Dans le contexte de l'estimation de WCET, on parle de techniques d'*analyse statique de WCET*. Le but de ces techniques d'estimation analytique du WCET est de s'affranchir du test global pour l'estimation du WCET. La connaissance du code du programme est nécessaire pour pouvoir l'analyser statiquement, c'est-à-dire sans l'exécuter entièrement ni simuler une exécution particulière avec un jeu de données.

Contrairement aux méthodes par tests et mesures pour lesquelles le résultat de l'estimation

est au plus égal au WCET réel mais peut être inférieur, le résultat d'une analyse statique de WCET est *au moins* égal au WCET réel et peut être supérieur. En effet, l'analyse statique de WCET cherche à calculer un majorant du WCET réel, elle est donc sûre, mais pessimiste. La qualité des techniques d'analyse statique de WCET est d'autant plus grande que le facteur de surestimation (ou pessimisme) qui affecte les résultats est faible.

Un autre atout des techniques d'analyse statique de WCET est la possibilité de les inclure plus tôt dans le processus de développement du logiciel. En effet, ces techniques permettent (si les temps d'analyse ne sont pas prohibitifs) d'analyser du code pendant sa phase de développement et d'utiliser les résultats d'analyse pour l'améliorer.

Dans la suite de ce document, nous ne nous intéresserons qu'à cette classe de technique d'estimation du WCET.

1.2.4 Qualités relatives des estimations de WCET obtenues

La figure 1.3 représente les relations entre les temps d'exécution estimés par les deux classes de techniques présentées dans les paragraphes précédents. Les différents résultats obtenus par tests et mesures sont classés par ordre croissant des résultats de mesure et sont notés $t_1, t_2, \dots t_{n-1}, t_n$. Les résultats obtenus par analyse statique de WCET sont notés t_{s0}, t_{s1}, t_{s2}.

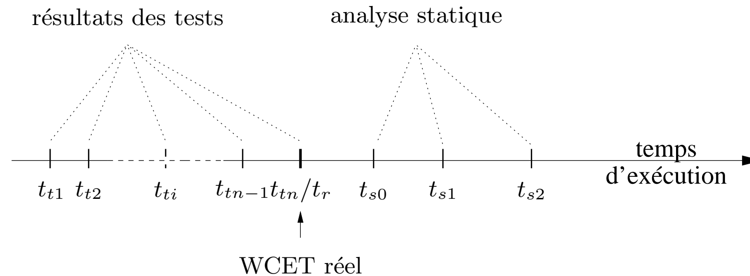


FIG. 1.3 – *Comparaison de deux classes de méthodes d'estimations de WCET*

Les temps d'exécution mesurés sont inférieurs (ou égaux dans le cas idéal) au temps d'exécution réel, et les temps d'exécution estimés par analyse statique sont supérieurs (ou égaux dans le cas idéal).

$$t_{ti} \leq t_r \leq t_{s0} \leq t_{s1} \leq t_{s2}$$

Un résultat d'analyse statique correpond à l'estimation du temps d'exécution le long d'un chemin d'exécution particulier, le pire chemin d'exécution. Le pire chemin d'exécution identifié par l'analyse statique peut être soit un chemin faisable, c'est-à-dire un chemin qui peut effectivement apparaître pendant une exécution réelle, soit un chemin infaisable, c'est-à-dire un chemin d'exécution présent dans le graphe de flot de contrôle du programme mais qui ne peut pas apparaître pendant une exécution réelle (code mort, exclusion mutuelle entre deux

portions du programme, etc.). Une technique d'analyse statique qui considère tous les chemins d'exécutions (faisables et infaisables) présent dans le graphe de flot de contrôle donnera des estimations de WCET supérieures à celles obtenues en ne considérant que les chemins faisables. Sur la figure 1.3, le résultat t_{s1} correspond à une analyse statique ne prenant en compte que les chemins faisables tandis que t_{s2} considère tous les chemins faisables mais aussi des chemins infaisables (exception faite de ceux qui sont statiquement exclus de l'analyse). Si le pire chemin d'exécution est un chemin faisable alors les deux résultats précédents sont égaux ($t_{s1} = t_{s2}$). L'écart entre ces deux résultats peut être réduit si on est capable d'identifier statiquement une partie des chemins infaisables pour les exclure de l'analyse.

Le résultat t_{s0} correspond à une analyse statique avec prise en compte de certains effets de l'architecture matérielle sur le WCET. Certains éléments d'architecture comme les caches et les pipelines permettent de réduire les temps d'exécution réels des programmes. Si on ne les prend pas en compte, les estimations sont très pessimistes ($t_r < t_{s0} << t_{s1}$), et si on sait les modéliser parfaitement (*i.e.* connaître statiquement et parfaitement le comportement dynamique de l'architecture matérielle) alors $t_r = t_{s0}$.

Le choix d'une méthode d'estimation implique un compromis entre la précision et la sûreté. L'analyse statique de WCET offre la sûreté au dépend de la précision.

Chapitre 2

L'analyse statique de WCET

2.1 Les données de l'analyse statique de WCET

Comme on l'a vu précédemment, l'analyse statique de WCET analyse le code des programmes. Nous allons maintenant présenter les différents types de code analysés et quelles sont les autres informations nécessaires à l'analyse (§ 2.1.1). Puis nous décrivons la mise en forme de ces informations (§ 2.1.2).

2.1.1 Langages sources

Pour pouvoir analyser statiquement le WCET d'un programme, celui-ci doit être écrit dans un langage analysable statiquement. Nous présentons ici les caractéristiques de langages analysables et les modifications à apporter à un langage pour le rendre analysable. Nous discutons aussi de l'influence des compilateurs optimisants sur l'analyse statique de WCET.

2.1.1.1 Type de langage analysé

Le code analysé peut être de plus ou moins haut niveau ; il peut imposer plus ou moins de structuration, ou encore introduire des abstractions. Nous allons étudier plusieurs possibilités de langages sources, utilisés dans diverses techniques d'analyse de WCET, en commençant par le langage source de plus bas niveau.

Le langage source de plus bas niveau analysable statiquement est le code assembleur lié à l'architecture dans lequel est codé le programme à analyser, soit directement (encore souvent le cas pour les systèmes embarqués), soit après compilation d'un langage de plus haut niveau. L'analyse du code assembleur ne permet d'obtenir (directement) que la représentation par graphe de flot de contrôle (*cf* § 2.1.2.2) qui est utilisée par les techniques d'analyse statique exposées aux paragraphes 2.2.1 et 2.2.2. Le code assembleur est également utile pour l'analyse de bas niveau (voir § 2.3).

Pour obtenir plus d'informations, il est également possible d'analyser le code source d'un programme écrit dans un langage de plus haut niveau (*e.g.* C, ADA, etc.). Un tel langage introduit la notion de structure syntaxique et permet ainsi la construction d'une représentation du programme par arbre syntaxique (*cf.* § 2.1.2.3) qui est nécessaire pour l'analyse à base d'arbre (*cf.* § 2.2.3).

Le langage analysé peut aussi être spécialement conçu pour le codage de tâches temps-réel comme RT-Euclid [KS86], FLEX [LN88] ou encore MPL [NTA90]. Le langage MPL (*Maruti Programing Language*) est basé sur le langage C++ et a été développé pour le système d'exploitation distribué *Maruti*. MPL offre plusieurs constructions permettant de spécifier des contraintes temporelles. Ces contraintes s'appliquent aux envois de messages et aux blocs de code. Le langage FLEX est lui aussi basé sur le langage C++, et associe des contraintes temporelles à des blocs de code et permet d'exprimer des relations de précédence. Ces informations incluses dans la syntaxe du langage sont principalement dédiées au test d'ordonnançabilité.

Dans le cas de Real-Time Euclid, la définition même du langage fait que chaque construction du langage est bornée dans le temps et l'espace. Ainsi, un programme écrit en RT-Euclid peut toujours être analysé statiquement pour obtenir une estimation de son WCET.

Une troisième possibilité est l'analyse d'un langage de programmation à objets qui introduisent d'autres abstractions telles que l'héritage, le polymorphisme, etc. L'analyse statique de WCET de ce type de langage, étudiée dans [EG98], soulève plusieurs difficultés. Tout d'abord, le code résultant de la compilation de ce type de langage est trop dynamique. Un exemple est la possibilité de redéfinir des méthodes définies dans les classes parentes. Cela revient à dire que lorsqu'on exécute une méthode d'un objet on ne sait pas quel code sera vraiment exécuté. Un des derniers langages étudiés dans le cadre de l'analyse statique de WCET est le langage Java, ou plus précisément le bytecode Java (noté JBC, pour JavaByteCode). Java est un langage objet, on retrouve donc les problèmes liés au dynamisme de ces langages présenté précédemment. Le JBC est un langage de programmation de machine à pile, et peut être obtenu par compilation d'un langage de plus haut niveau, dont Java (mais aussi ADA, C, etc.). Le JBC est une représentation intermédiaire entre un source de haut niveau et sa traduction en assembleur proche de l'architecture. Le but de l'analyse de WCET de JBC [BBMP00, PB01] est d'être portable tout en restant précise grâce à une prise en compte de l'architecture matérielle sur laquelle il est porté. La portabilité est assurée par l'utilisation d'un *modèle temporel* dépendant de l'architecture cible. On dispose d'un modèle temporel pour chacune des architectures sur lesquelles peut s'exécuter le programme en JBC. Le WCET d'un programme est décrit de manière unique et indépendante de l'architecture. La combinaison de ce WCET portable et d'un modèle temporel d'architecture donne le WCET du programme sur cette architecture.

2.1.1.2 Restriction du langage source

Afin d'être analysable, le langage source choisi doit posséder certaines propriétés permettant son analyse statique. Si ces propriétés ne sont pas intrinsèques au langage, celui-ci doit être restreint (et/ou augmenté, *cf.* § 2.1.1.3) pour respecter ces propriétés et ainsi permettre son analyse statique dans le but d'estimer son WCET.

La première propriété que doit respecter le code analysé est de permettre à l'analyseur de connaître *statiquement* tous les chemins d'exécution possibles. Pour ce faire, les branchements dynamiques (*e.g.* appel de fonction par pointeur) sont interdits, car ils introduisent de nouveaux chemins d'exécution dynamiquement. Dans le même but, on interdit souvent la récursivité (même indirecte) car elle introduit des chemins d'exécution de longueurs potentiellement infinies. Il est cependant possible de la prendre en compte pour l'analyse statique si on est capable de borner sa profondeur statiquement.

Ces premières restrictions s'appliquent quelque soit le langage analysé. On s'intéresse maintenant à un langage de haut niveau dont on peut extraire la structure syntaxique. Un tel langage nous permet de représenter le programme par un arbre syntaxique (voir § 2.1.2.3). D'autre part, le code assembleur obtenu par compilation du programme source haut niveau est représenté par un graphe de flot de contrôle (voir § 2.1.2.2).

Si on souhaite pouvoir analyser statiquement le programme en utilisant conjointement les deux niveaux du langage source (langage de haut niveau et code objet), il faut que leurs représentations (l'arbre syntaxique et le graphe de flot de contrôle) soient en correspondance directe. Pour assurer cette correspondance entre les deux représentations, il est nécessaire de restreindre l'expressivité du langage source. On cherche à avoir un langage de haut niveau ne permettant d'écrire que des programmes "bien structurés", c'est-à-dire dont tous les chemins d'exécution apparaissent dans l'arbre syntaxique. Pour ce faire, on doit restreindre le langage afin de supprimer les possibilités de faire diverger le graphe et l'arbre. C'est pourquoi on peut interdire l'utilisation des commandes permettant des branchements non structurés tel que les GOTO, EXIT et CONTINUE du langage C, et parfois la sortie multiple des fonctions ou procédures.

2.1.1.3 Informations complémentaires

Le but de ces informations est de fournir à l'analyseur des informations sur le comportement dynamique du code à analyser. Ces informations permettent de restreindre le nombre de chemins d'exécution possibles, et donc le nombre de chemins à prendre en compte pour l'analyse.

Ces informations sont le plus souvent utilisées pour :

– Borner le nombre maximum d'itérations des boucles. Le WCET d'une boucle dépend non seulement de son code mais aussi de son pire nombre d'itérations. Cette information doit donc être associée aux boucles, le plus souvent sous forme d'une constante [PK89, EG98].

- Contraindre le choix d'une branche dans une structure conditionnelle. Par exemple quand le choix d'une branche conditionnelle dépend d'un paramètre que l'on sait être constant.
- Restreindre le nombre de chemins d'exécution possibles en indiquant les chemins infaisables. Par exemple, les informations complémentaires peuvent indiquer qu'une partie du code ne peut faire partie du pire chemin d'exécution (code mort ou cas d'erreur), ou encore, pour indiquer que deux branches s'excluent mutuellement. Un exemple simple est :

```
if(x=0) then A else B ; if (x>0) then C;
```

La valeur de la variable `x` implique que les branches `A` et `C` sont mutuellement exclusives.

On distingue deux possibilités pour obtenir ces informations. La première fait appel à l'utilisateur (*i.e.* le programmeur) qui doit fournir ces informations en les ajoutant au code source sous forme d'annotations [PK89, CBW96, Par93], ou interactivement [KWH95]. La plupart des travaux concernant l'analyse statique de WCET utilise cette méthode.

La deuxième possibilité [HSR+00, EG98] permet dans certains cas d'obtenir automatiquement les bornes sur les nombres d'itérations des boucles en analysant le code et les données manipulées. Bien que ces techniques soient efficaces pour des boucles relativement simples, l'annotation des boucles les plus complexes reste à la charge du programmeur.

Enfin, l'utilisation de techniques de preuves formelles a été envisagée dans [CBW96] pour vérifier les annotations fournies par l'utilisateur.

2.1.1.4 Optimisations de compilation

Même si l'analyse statique du code source de haut niveau permet aisément d'accéder à la structure du programme, les temps d'exécution de fragments de code ne peuvent être estimés précisément qu'à partir du code assembleur. Une approche courante d'analyse statique du WCET consiste à utiliser conjointement le code source haut niveau et l'assembleur. Pour ce faire, il est nécessaire d'établir une correspondance étroite entre la structure syntaxique de langage haut niveau et les enchaînements possibles des fragments de code identifiés dans le code assembleur. Ceci reste simple si les schémas de compilation mis en jeu pour passer du langage haut niveau à l'assembleur sont connus et constants, mais l'établissement d'une telle correspondance peut être rendue très difficile si le compilateur ne se contente pas de traduire le code source en assembleur [Vrc94]. En effet, les compilateurs optimisants, qui ont pour but d'améliorer les performances moyennes du code compilé, réorganisent le code assembleur en déroulant par exemple certaines boucles et appels de fonctions (fonction *inline*), ou encore en les dupliquant et les spécialisant.

J. Engblom et al. ont proposé dans [EEA98] une nouvelle approche (appelée *co-transformation*) pour l'établissement d'une correspondance entre les informations provenant du code source de haut niveau et le code objet optimisé, afin de permettre l'analyse conjointe de ces deux niveaux de langage même en présence d'optimisations de compilation. Son principe

de base est la transformation des informations de haut niveau (annotations de boucles par exemple) par le compilateur en parallèle avec l'optimisation du code. Le *co-transformateur* transforme les informations complémentaires fournies avec le code source de haut niveau de façon à ce qu'elles correspondent à la structure du code objet obtenu après compilation optimisante. Pour ce faire, on fournit au *co-transformateur* la description des différentes transformations que peut effectuer le compilateur (les définitions des transformations) ainsi que la trace des transformations ayant effectivement eu lieu lors de la compilation. Cette trace des transformations, générée par le compilateur, indique quelles transformations ont été appliquées et où elles ont été appliquées. Le *co-transformateur* applique les transformations correspondantes aux informations complémentaires dans le même ordre.

2.1.2 Représentations logiques

Une fois le programme écrit dans un langage analysable, on cherche à construire sa représentation logique. Le code objet est tout d'abord scindé en plus petites unités, les blocs de base (§ 2.1.2.1). Puis, suivant le niveau du langage et les informations souhaitées, on va représenter les fonctions du programme par des *graphes de flot de contrôle* (§ 2.1.2.2) ou des arbres syntaxiques (§ 2.1.2.3). Enfin, les représentations des fonctions du programme sont assemblées pour ne former qu'une unique représentation logique du programme (§ 2.1.2.4).

2.1.2.1 Décomposition du programme en blocs de base

Presque toutes les méthodes d'analyse statique de WCET utilisent le découpage du code objet en *blocs de base*. Un bloc de base est une suite d'instructions purement séquentielle ne contenant qu'un seul point d'entrée et un seul point de sortie.

- Un seul point d'entrée signifie que si une instruction de branchement hors du bloc de base effectue un saut vers une instruction dans le bloc de base, c'est obligatoirement la première instruction qui est visée.
- Un seul point de sortie signifie que les branchements d'un bloc de base sont soit des branchements inconditionnels vers l'instruction suivante dans le bloc de base, soit la dernière instruction du bloc.

L'exemple de la figure 2.1 présente le code C d'un programme constitué de deux fonctions. La fonction principale `main` comporte une boucle, et pour chaque itération de la boucle la fonction `impaire` est appelée deux fois. Cette deuxième fonction comporte une structure conditionnelle.

Les deux fonctions constituant ce programme nous serviront à illustrer nos propos dans tous les exemples de ce chapitre.

Ce programme, une fois compilé sans optimisations, donne un code assembleur présenté à la figure 2.2 (code Intel *80x86*). Les blocs de base sont décrits par les zones de couleurs différentes sur la figure. La construction des blocs de base par regroupement d'instructions

```
int impaire(int x) {
  int result;

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

```
void main() {
  int tab[4] = {34,45,12,5};
  int nb_impaires = 0;
  int nb_paires = 4;
  int i;

  for(i=0;i<4;i++) {
     nb_impaires += impaire(tab[i]);
     nb_paires -= impaire(tab[i]);
  }
}
```

Fig. 2.1 – *Exemple de code source d'un programme analysé (langage de haut niveau)*

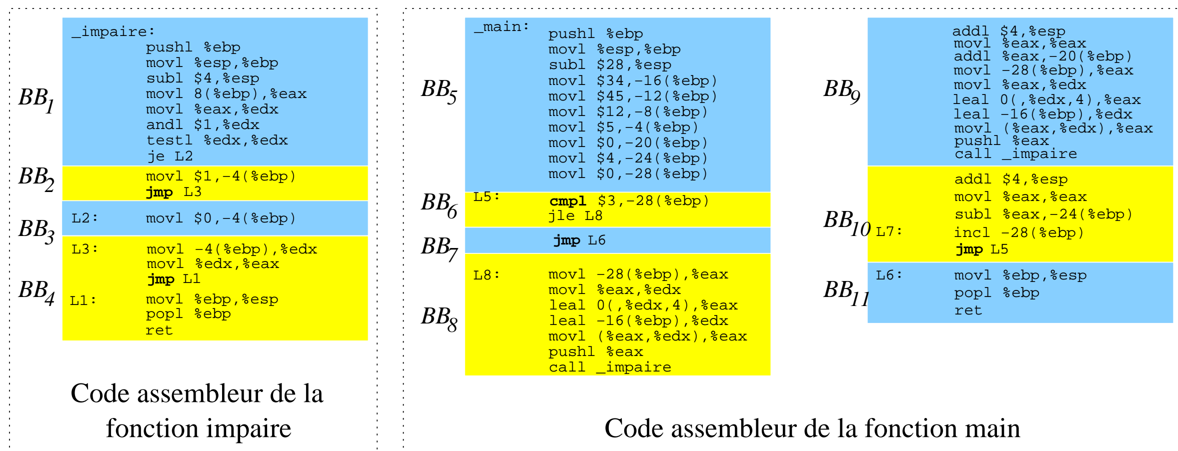


Fig. 2.2 – *Exemple de décomposition d'un code assembleur en blocs de base*

obéit aux règles précitées : instructions qui se suivent, un seul point d'entrée, et un seul point de sortie. Notons que le bloc de base BB_4 contient deux branchements, ce qui est possible car le premier branchement saute vers l'instruction suivante de façon inconditionnelle. Les blocs de base BB_1 à BB_{11} ainsi obtenus sont les éléments de base des deux représentations décrites dans les paragraphes suivants.

2.1.2.2 Le graphe de flot de contrôle

Un *graphe de flot de contrôle* décrit tous les enchaînements de blocs de base possibles pour chaque fonction du programme. Il y a un graphe de flot de contrôle par fonction. Il est obtenu soit par analyse statique du code bas niveau en utilisant un compilateur modifié [MM98], soit par un outil dédié à la manipulation de code bas niveau comme par exemple SALTO [BRS96]. On distingue deux types de graphes équivalents et couramment utilisés.

La première catégorie (*cf.* figure 2.3) est celle des graphes dont les nœuds sont des blocs de base et où les arcs représentent les relations de précédences entre blocs. On peut alors distinguer deux types d'arc : les arcs "pas de saut" qui représentent l'exécution de deux blocs qui se suivent sans saut (absence de branchement, ou branchement non-pris) et les arcs "saut"

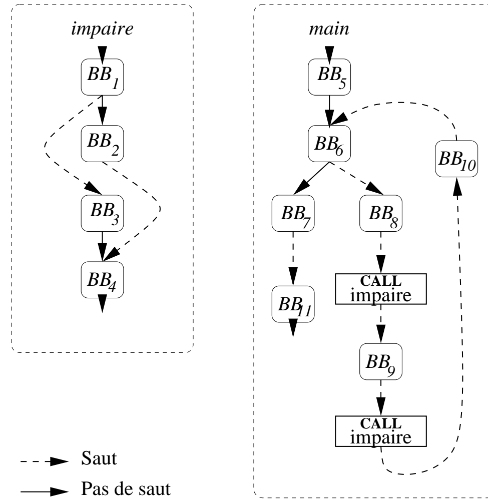


FIG. 2.3 – *Graphes de flot de contrôle du programmes analysé*

qui représentent les branchements pris (*e.g.* arc de BB_{10} à BB_6). Cette première catégorie de graphe de flot de contrôle est utilisée dans de nombreux travaux d'analyse statique basés sur les graphes [LMW95a, PF99, SA00].

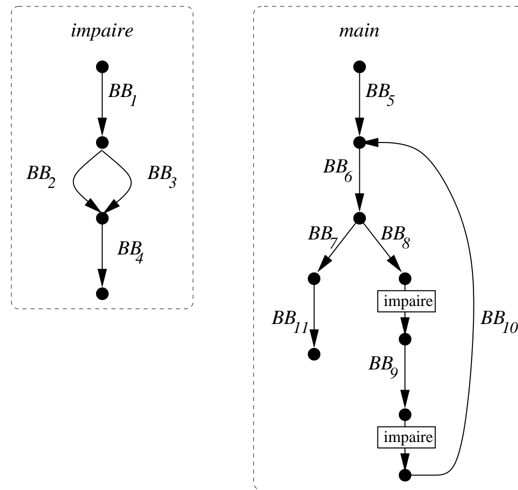


FIG. 2.4 – *T-graphs équivalents aux graphes de flot de contrôle*

La deuxième catégorie de graphe (*cf.* figure 2.4), baptisée *T-graph* par P. Puschner [PS97], est une représentation duale de la première. Les arcs y représentent les blocs de base et sont valués par le WCET de ces blocs, et les nœuds du graphe représentent les points dans le code où le flot de contrôle du programme converge et/ou diverge. Cette représentation est utilisée dans les travaux [PS97] et [OS97].

Le graphe de flot de contrôle est le plus souvent construit par analyse statique de code de

bas niveau : du code assembleur le plus souvent, mais aussi du bytecode [BBW00, BBMP00]. P. Puschner envisage aussi dans [Pus98] l'obtention de *T-graphs* à partir du code de haut niveau du programme.

2.1.2.3 L'arbre syntaxique

Un *arbre syntaxique* (*cf.* figure 2.5) est un arbre dont les nœuds représentent les structures du langage de haut niveau et dont les feuilles sont les blocs de base.

Une représentation simple du programme par un arbre syntaxique peut être basée sur trois types de nœuds et deux types de feuilles.

- Les nœuds de type SEQ possèdent au moins un fils. Ils représentent la mise en séquence de leurs sous-arbres fils.
- Les nœuds de type LOOP ont deux fils : le sous-arbre test et le corps de la boucle. Le nombre d'itérations autorisées est borné (sur la figure 2.5, la borne est *[4]*).
- Les nœuds de type IF sont constitués d'un sous-arbre test et de deux sous-arbres "then" et "else".
- Les feuilles de type CALL représentent des appels de fonctions.
- Enfin, les autres feuilles de l'arbre syntaxique sont les blocs de base du programme.

On peut imaginer d'autre types de nœuds pour représenter par exemple différents types de boucles.

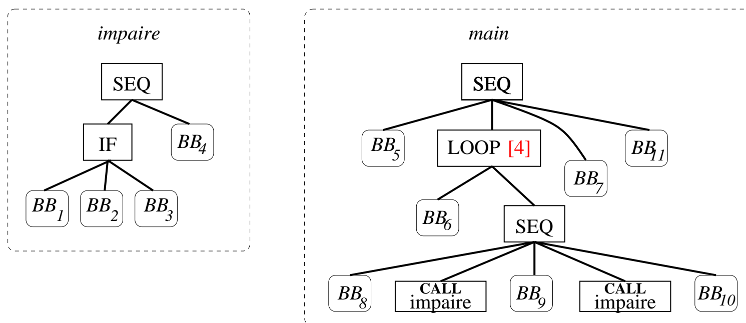


Fig. 2.5 – *Arbres syntaxiques des fonctions impaire et main*

L'arbre syntaxique est obtenu par analyse statique d'un langage de haut niveau. Les feuilles de l'arbre syntaxique coïncident avec les nœuds du graphe de flot de contrôle décrit au paragraphe précédent. Les structures du langage (séquence, conditionnelle, boucles) représentées dans l'arbre syntaxique induisent des relations de précédence entre leurs fils comme le montre la figure 2.6.

Par exemple, si on suppose que A_1 à A_n sont les sous-arbres d'un nœud séquence. Alors, dans un chemin d'exécution, la feuille la plus à droite du sous-arbre A_m sera suivi de la feuille la plus à gauche de A_{m+1}. L'arbre syntaxique contient les même informations sur les relations

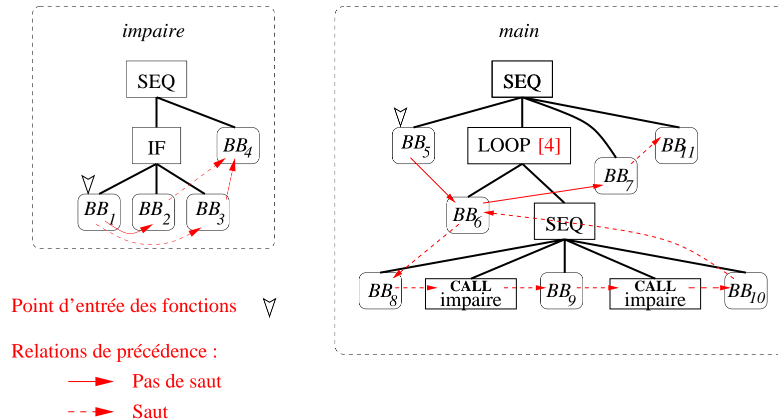


FIG. 2.6 – *Relations de précédence induites de l'arbre syntaxique*

de précédences entre les blocs de bases que celles représentées par les arcs du graphe de flot de contrôle. L'arbre syntaxique apporte donc une information nouvelle par rapport au graphe de flot de contrôle (les structures du langage) tout en comportant les mêmes informations concernant le flot de contrôle entre les blocs de base (de façon moins directe toutefois).

2.1.2.4 Dépliage des appels de fonctions

Dans les deux types de représentation précédents, chaque fonction est représentée séparément et en un seul exemplaire.

Si on prend en compte l'effet de certains éléments de l'architecture matérielle comme par exemple les caches, le WCET d'un bloc de base va dépendre de l'état interne du matériel au moment de son exécution (le contenu des caches par exemple). Si une fonction est appelée de plusieurs "lieux" différents, les blocs de base qui la composent seront exécutés dans plusieurs contextes différents. Leur WCET peut donc varier en fonction du lieu d'appel.

Cette observation a mené F. Mueller à déplier les appels de fonctions [MWH94] afin d'obtenir : (i) une seule représentation (arbre ou graphe) contenant tout le code à analyser, et (ii) autant d'instances de fonctions que de "lieux" d'appel différents.

Pour transformer les appels de fonctions en instances de fonctions, les fonctions sont dupliquées et incluses dans l'arbre principal en lieu et place des feuilles de type CALL (*cf.* exemple figure 2.7). On fait de même en ce qui concerne le graphe de flot de contrôle et le *T-graph* (*cf.* figure 2.7). Dans ces deux exemples, la fonction *impaire* est dupliquée une fois, les blocs de base qui la composent (BB_1 à BB_4) ont donc été dupliqués (les duplicatas sont BB_{12} à BB_{15}). Ainsi, BB_1 et BB_{12} sont en tout point identiques (même contenu, adresse, etc.) mais pourront se voir attribuer des résultats d'analyse différents.

L'absence de récursivité, ou la limitation statique de la récursivité permet d'assurer un nombre fini de duplications de fonctions.

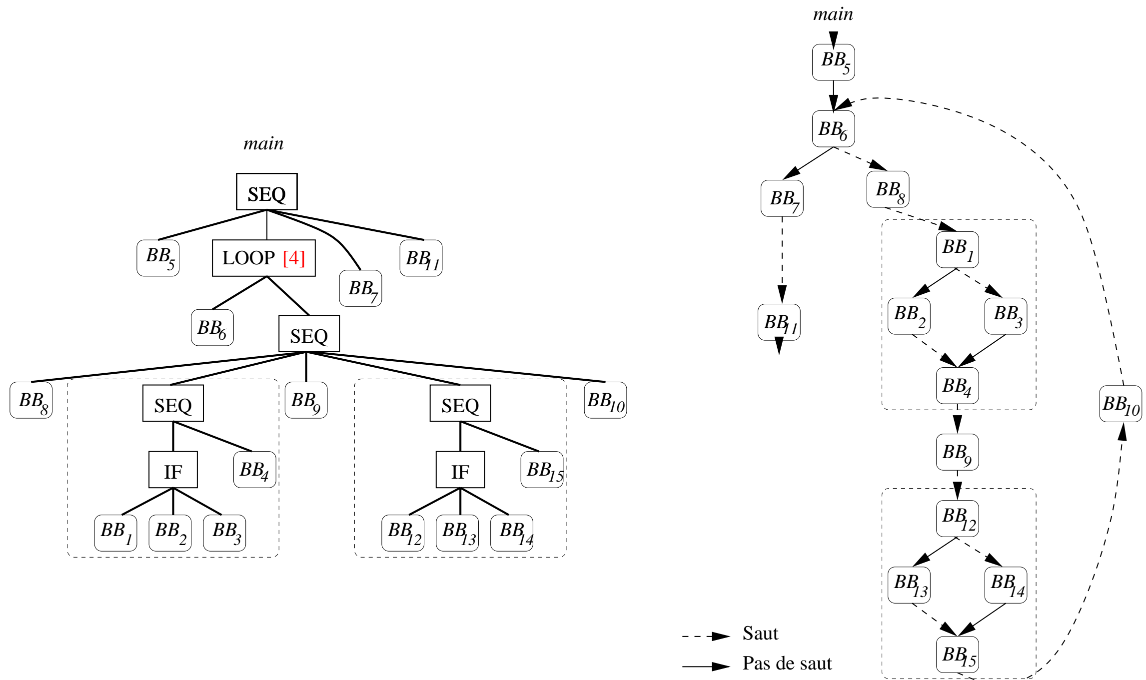


FIG. 2.7 – *Arbre syntaxique et graphe de flot de contrôle après inclusion des fonctions appelées*

2.1.2.5 Influence du schéma de compilation

La manière dont le code source est compilé en assembleur a une grande influence sur la construction des représentations de programme précédentes.

Le schéma de compilation détermine le type de mise en séquence des blocs de base. La figure 2.8 présente un exemple des résultats possibles de la compilation d'une structure de boucle simple.

Ainsi, la correspondance entre l'arbre et le graphe de la figure 2.7 n'est valable que pour un schéma de compilation particulier. De même, les relations de précédence implicites entre les feuilles de l'arbre syntaxique ainsi que le type de ces relations de précédence dans les deux représentations (branchement "saut" ou "pas de saut") dépendent du schéma de compilation mis en œuvre par le compilateur choisi.

2.2 L'analyse statique de WCET de haut niveau

Une fois la représentation du programme à analyser obtenue, on va chercher le pire chemin d'exécution dans cette représentation, c'est-à-dire le plus long chemin dans le graphe de flot de contrôle, ou encore le plus long parcours de l'arbre syntaxique correspondant à une exécution possible.

La méthode utilisée pour la recherche du plus long chemin permet d'identifier les différentes méthodes d'analyse statique de WCET. On suppose ici que les WCET élémentaires des blocs de base sont connus, on s'intéressera à leur obtention au paragraphe 2.3. De plus, dans un

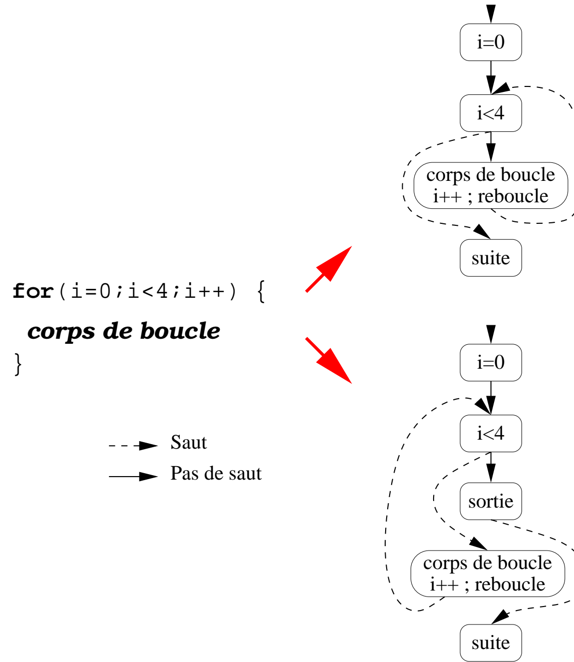


FIG. 2.8 – *Deux schémas de compilation possibles pour une même structure de boucle*

premier temps, on pose les deux hypothèses simplificatrices suivantes :

- $\mathcal{H}1$: On suppose que le WCET d'une séquence de deux blocs de base est égal à la somme des WCET des blocs de base pris séparément.
- $\mathcal{H}2$: On suppose que le WCET d'un bloc de base est constant quelque soit son contexte d'exécution (*i.e* son WCET ne dépend pas de l'enchaînement de blocs de base le précédant dans un chemin d'exécution).

Ces hypothèses simplificatrices, qui sont à la base des premiers travaux du domaine, nous permettent de ne pas nous préoccuper pour l'instant du niveau bas de l'analyse statique de WCET. Elles seront levées au paragraphe 2.3.

2.2.1 Techniques utilisant l'algorithmique des graphes (*Path-based*)

La connaissance des WCET individuels des blocs de base permet d'associer des WCET aux nœuds du graphe de flot de contrôle, ou aux arcs du *T-graph*. On obtient alors un graphe valué avec un seul point d'entrée et un seul point de sortie. On peut chercher le pire chemin d'exécution dans le graphe de flot de contrôle [HAM+99, SA00] en utilisant les algorithmes traditionnels de l'algorithmique des graphes (*e.g.* algorithme de Dijkstra [Dij59]) recherchant le plus long chemin dans un graphe. Puis on vérifie que le chemin trouvé est un chemin d'exécution possible, et si ce n'est pas le cas, on l'exclut du graphe et on recommence la recherche. Les informations de nombre maximum d'itérations limitent le nombre d'occurrences d'un bloc de base ou d'un arc dans un chemin d'exécution.

2.2.2 Techniques *IPET*

Cette méthode, dite *d'énumération implicite des chemins* (noté *IPET - Implicit Path Enumeration Technique*) est utilisée dans de nombreux travaux d'analyse statique de WCET [FMW97, LM95a, OS97, PS95]. Cette approche ne s'appuie que sur la représentation du programme sous forme de graphe de flot de contrôle, qu'elle transforme en un ensemble de contraintes devant être respectées. Un premier jeu de contraintes décrit la structure du graphe, et un deuxième permet de prendre en compte les informations de bornes des boucles vues au paragraphe 2.1.1.3. L'ensemble des contraintes a pour rôle d'éliminer les chemins infaisables, et de restreindre l'ensemble des chemins possibles. On cherche ensuite à maximiser l'expression du WCET en respectant les deux jeux de contraintes. Les techniques *IPET* consistent donc en une énumération implicite des chemins d'exécution susceptibles de maximiser le WCET.

En utilisant l'exemple introduit à la figure 2.4, la figure 2.9 illustre la traduction d'un *T-graph* (où les appels de fonctions dépliés sont en grisé) en un système de contraintes. Cette méthode de traduction de *T-graph* est utilisée par P. Puschner dans [PS97].

Chaque arc a_i du *T-graph* est valué par n_i, son nombre d'occurrences dans le pire chemin d'exécution. À chaque nœud du graphe, la somme des nombres d'occurrences des arcs entrants doit être égale à celle des nombres d'occurrences d'arcs sortants. On obtient ainsi un premier système de contraintes qui décrit la structure du graphe (contraintes (a) sur la figure).

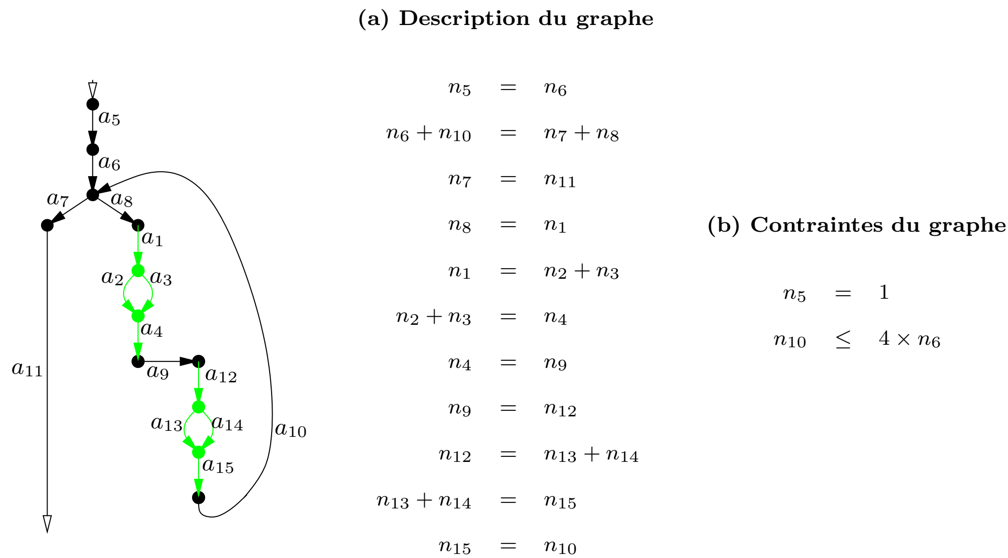


FIG. 2.9 – *Traduction du T-graph en un système de contraintes*

Les annotations de boucle fournissent les contraintes de boucle (contraintes (b) sur la figure), par exemple le nombre d'occurrences de l'arc de rebouclage a_{10} est au plus quatre fois

plus grand que le nombre d'occurrences de l'arc entrant dans la boucle a_6. Ce qui contraint la boucle $a_8 \rightarrow \ldots \rightarrow a_{10}$ à itérer au maximum quatre fois pour chaque entrée (a_6) dans la boucle. Le premier arc du *T-graph* est valué par $n_5 = 1$ pour que le calcul estime le WCET pour *une* exécution du programme.

De la même façon qu'avec un *T-graph*, un système de contraintes peut être obtenu à partir de la représentation équivalente par graphe de flot de contrôle.

Étant donné ces deux systèmes de contraintes, on cherche à maximiser l'expression du WCET :

$$WCET = \sum_i n_i \times w_i$$

où w_i est le WCET du bloc de base représenté par l'arc a_i dans le *T-graph*. Les valeurs des w_i sont fournies par l'analyse de bas niveau (voir § 2.3).

Les techniques de calcul IPET ne font appel qu'au graphe de flot de contrôle du programme. Comme l'arbre syntaxique n'est pas utilisé conjointement au graphe de flot de contrôle, ce dernier n'a pas besoin d´être bien structuré (*cf.* § 2.1.1.2). En levant cette restriction, les méthodes IPET permettent d'analyser plus de programme que les méthodes à base d'arbre présentées ci-après. De plus, elles permettent d'ajouter d'autres contraintes que celles liées aux boucles pour, par exemple, prendre en compte l'exclusion mutuelle de deux blocs de base (par exemple, $n_\alpha + n_\beta \leq 1$ implique l'exclusion mutuelle entre les arcs a_α et a_β). Cette possibilité est exploitée par exemple dans [LM95b].

La maximisation de l'expression du WCET ci-dessus fournit les valeurs des n_i (et évidement le WCET). Les méthodes de résolution utilisées sont similaires à celles utilisées pour résoudre les problèmes de programmation linéaire (Integer Linear Programming) ou de satisfaction de contraintes. Le problème standard de la programmation linéaire entière (ILP) est la recherche de l'extremum d'une fonction linéaire à plusieurs variables, ces variables étant assujetties à un ensemble de contraintes fonctionnelles (*i.e.* elles doivent vérifier un système d'équations et/ou d'inéquations). Un tel problème peut être résolu en utilisant un outil tel que *lp_solve* (Université de Technologie d'Eindhoven), ILOG CPLEX (société ILOG) ou encore Maple [CGG91].

Ces méthodes comportent toutefois un inconvénient majeur : la complexité de la résolution du système impose parfois des temps d'analyse importants (on parle de quelques secondes à plusieurs heures dans [LMW96]).

À noter les récents travaux de P. Puschner [Pus98] sur l'énumération implicite des chemins dans un arbre syntaxique. Il s'agit alors d'appliquer les mêmes techniques (traduction en système de contraintes, puis résolution) à un arbre syntaxique au lieu d'un graphe de flot de contrôle.

2.2.3 Techniques basées sur l'arbre syntaxique (*Tree-based*)

Cette classe de méthodes, proposée initialement par P. Puschner et C. Koza dans [PK89], s'appuie sur la représentation du programme en arbre syntaxique pour calculer récursivement son WCET. Un ensemble de formules (*cf.* table 2.1), nommé schéma temporel (*timing schema*), permet d'associer à chaque structure syntaxique du langage source (un nœud de l'arbre syntaxique) un WCET, et ce en fonction des sous-arbres qui la composent (les fils du nœud). Et ainsi de suite, les WCET des sous-arbres étant eux-mêmes calculés à partir des WCET de leurs fils jusqu'à arriver aux feuilles de l'arbre que sont les blocs de base dont on suppose les WCET connus. On effectue donc un parcours de bas en haut (*bottom-up*) de l'arbre en partant des feuilles porteuses de l'information de WCET, pour obtenir le WCET de la racine.

	Formules pour le calcul du WCET : $W(S)$
$S = S_1; \ldots ; S_n$	$WCET(S) = WCET(S_i) + \ldots + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
$S =$ loop(tst) S_1	$WCET(S) = maxiter \times (WCET(Test) + WCET(S_1)) + WCET(Test)$ où $maxiter$ est le nombre maximum d'itérations.
$S = f(exp_1, \cdots , exp_n)$	$W(S) = W(exp_1) + \cdots + W(exp_n) + W(f())$
$S =$ bloc de base BB_x	$WCET(S) =$ WCET du bloc de base BB_x

TAB. 2.1 – *Formules de calcul du WCET [PK89]*

À chaque nœud de l'arbre où un choix est possible, on choisit le chemin qui maximise le temps d'exécution. Par exemple, le WCET d'une séquence est simplement la somme des WCET des structures qui la composent (*cf.* première ligne du tableau 2.1), et le WCET d'une conditionnelle implique l'utilisation de l'opérateur *max* pour choisir la branche conditionnelle dont le temps d'exécution est le plus important.

Les informations concernant les boucles définies dans le paragraphe 2.1.1.3 sont prises en compte par les formules associées aux structures de boucle. L'énumération de tous les chemins d'exécution possibles est réalisée par l'application récursive des équations le long de toute la structure arborescente du programme. Par exemple, pour l'arbre syntaxique représenté dans la figure 2.7, où le nombre maximum d'itérations de la boucle est de 4, on obtient le système d'équation suivant :

$$WCET(main) = WCET(BB_5) + WCET(LOOP) + WCET(BB_7) + WCET(BB_{11})$$
$$WCET(LOOP) = 4 \times (WCET(BB_6) + WCET(SEQ_1)) + WCET(BB_6)$$

$WCET(SEQ_1) = WCET(BB_8) + WCET(SEQ_2) + WCET(BB_9) + WCET(SEQ_3) + WCET(BB_{10})$
$WCET(SEQ_2) = WCET(IF_1) + WCET(BB_4)$
$WCET(SEQ_3) = WCET(IF_2) + WCET(BB_{15})$
$WCET(IF_1) = WCET(BB_1) + max(WCET(BB_2), WCET(BB_3))$
$WCET(IF_2) = WCET(BB_{12}) + max(WCET(BB_{13}), WCET(BB_{14}))$

On obtient ainsi un arbre temporel (*timing tree*) [Pus98] qui contient les WCET calculés aux différents nœuds de l'arbre.

2.2.4 Comparaison des techniques d'analyses de haut niveau

Les méthodes *tree-based* se distinguent des autres méthodes (*IPET* et *graph-based*) par l'utilisation de l'arbre syntaxique au lieu du graphe de flot de contrôle. Les principales différences qui en découlent sont :

- l'obligation d'analyser des programmes bien structurés,
- la difficulté à ajouter des contraintes sur les chemins d'exécution,
- les avantages liés à l'utilisation d'une représentation hiérarchique (emboîtement de boucles, blocs conditionnels, etc.),
- la possibilité d'utiliser un système d'annotations des boucles basé sur leur emboitement (voir chapitre 3),
- la connaissance du rôle de chaque branchement (test de boucle, test de conditionnelle, etc.),
- la possibilité d'utiliser le calcul symbolique pour l'estimation du WCET (voir chapitre 3).

Les méthodes *IPET* et *graph-based* utilisent le graphe de flot de contrôle. La principale différence entre ces deux classes de méthodes est que les méthodes *IPET* ne cherchent pas le pire chemin d'exécution mais donnent le pire nombre d'exécutions de chacun des nœuds (ou arcs suivant les méthodes) du graphe de flot de contrôle. Elles ne donnent pas d'information précise sur l'ordre d'exécution. Les approches *path-based*, quant à elles, calculent explicitement le plus long chemin d'exécution dans le programme, mais elles requièrent de connaître le nombre d'exécutions maximum des nœuds avant de commencer la recherche.

2.3 L'analyse statique de WCET de “bas niveau”

Les hypothèses simplificatrices $\mathcal{H}1$ et $\mathcal{H}2$ du paragraphe 2.2 ne sont vérifiées que si on ignore l'effet de certains éléments de l'architecture matérielle qui permettent d'améliorer les performances moyennes du système (par exemple les caches et pipelines). Par exemple, le cache d'instruction introduit une variation du temps de traitement d'une instruction en fonction du contexte, ce qui contrarie l'hypothèse $\mathcal{H}2$ de temps de traitement constant. De même, le pipeline, en introduisant du parallélisme dans le traitement d'une suite d'instructions, remet

en cause l'hypothèse $\mathcal{H}1$ de composition par simple somme du WCET des séquences de blocs de base.

Si on considère que l'effet de ces éléments d'architecture est nul (hypothèses $\mathcal{H}1$ et $\mathcal{H}2$), le WCET obtenu est une approximation très pessimiste du WCET réel. Par exemple, F. Mueller fait état dans [AMWH94] d'un facteur de surestimation des WCET allant de 4 à 9 pour les programmes de son jeu de test sans prise en compte du cache d'instructions, facteur qui est ramené entre 1 et 2 par sa méthode de prise en compte. Ce pessimisme de l'estimation entraîne une sous-utilisation importante du matériel. La prise en compte de ces éléments permet de réduire cette sous-utilisation et donc d'améliorer le taux d'utilisation d'un système temps-réel *strict* sans modification du matériel.

La modélisation du comportement de ces éléments d'architecture n'est pas triviale, et leur prise en compte introduit une dépendance au contexte. Par exemple, la durée d'exécution d'une instruction dans le pipeline dépend des instructions précédentes, les durées des accès mémoire en lecture et écriture dépendent de l'état des caches et donc de l'historique de l'exécution (instructions et données accédées). Si on veut lever les hypothèses trop pessimistes $\mathcal{H}1$ et $\mathcal{H}2$, il faut adapter les méthodes de calcul du WCET des instructions et donc des blocs de base.

Malgré ces difficultés, l'intégration de ces éléments dans l'estimation du WCET permet d'améliorer considérablement sa précision tout en garantissant la sûreté des estimations. C'est pourquoi de nombreux travaux concernent la prise en compte de ces éléments, principalement les caches [AFMW96, BMSO+96, LMW95b, LML+94, AMWH94] et les pipelines [BJ95, ZBN93], et ce le plus souvent pour des architectures de type RISC.

Dans les paragraphes qui suivent, nous donnons une vue d'ensemble des méthodes de prise en compte de l'effet de l'architecture matérielle sur le WCET. Les deux éléments les plus couramment considérés pour l'analyse statique de WCET sont le cache d'instructions (§ 2.3.1) et le pipeline (§ 2.3.2). L'intégration des techniques de prise en compte de ces deux éléments est présentée au paragraphe 2.3.3. La prise en compte de quelques autres éléments architecturaux intéressants est présentée au paragraphe 2.3.4.

2.3.1 Prise en compte des caches d'instructions

Les évolutions techniques récentes ont eu pour effet une augmentation considérable de la différence entre les vitesses des processeurs et celles des mémoires. La mémoire est devenue un élément limitant les performances de l'architecture. Or, les mémoires rapides sont d'un coût trop élevé pour que l'on puisse en disposer d'une quantité importante. Une solution à ce problème est l'installation d'une hiérarchie de caches organisée en plusieurs niveaux. Le niveau le plus bas, le niveau 1, est rapide mais de taille réduite. Quand un bloc de données auquel on veut accéder en est absent (défaut de cache), il y est recopié depuis le niveau supérieur. Grâce à ce mécanisme on dispose d'une quantité importante de mémoire presque aussi rapide que le cache, à un coût presque aussi faible que celui de la mémoire centrale.

L'usage d'une mémoire cache introduit de l'indéterminisme dans l'architecture, car en ajoutant cet intermédiaire entre la mémoire centrale et le processeur, la durée des accès mémoire n'est plus constante. En effet, deux durées correspondant à deux cas d'accès au cache sont à considérer. Une première durée correspond à l'accès à un bloc mémoire qui est dans le cache (succès, ou *hit*). Une deuxième durée correspond au cas où le bloc mémoire référencé est absent du cache (échec, ou *miss*), il est alors recopié depuis le niveau supérieur. La durée d'un accès *miss* est évidemment bien plus importante que celle d'un accès *hit*.

Il faut donc pouvoir estimer de façon *sûre* (et pas seulement de manière probabiliste) le résultat (*hit/miss*) des accès au cache et pour cela connaître statiquement le pire comportement des accès mémoire vis à vis du cache lors de l'exécution d'un programme. Ainsi, si on peut assurer qu'un accès mémoire sera un succès, on estime que le résultat est *hit*, et s'il y a un doute on garde l'estimation la plus pessimiste : *miss*.

Le cache d'instructions est un cas particulier pour deux raisons. D'une part, les seuls accès qui le concernent sont des accès en lecture car il ne contient que des instructions. D'autre part, toutes les références mémoire (*i.e.* les adresses des instructions) peuvent être connues statiquement, il suffit pour cela de connaître l'adresse d'implantation du programme en mémoire.

Plusieurs méthodes ont pour but d'intégrer le comportement du cache d'instructions dans le calcul du WCET [BN94]. Leur objectif commun est d'effectuer une classification de toutes les instructions d'un programme en fonction de leur comportement pire-cas par rapport au cache.

2.3.1.1 Prise en compte du cache par simulation statique de cache

Une première méthode est la *simulation statique de cache*. Ce terme, introduit par F. Mueller, désigne le fait d'envisager statiquement et en même temps tous les chemins d'exécution possibles du graphe de flot de contrôle, de manière à calculer une représentation de tous les contenus possibles du cache à différents moments de l'exécution. Cette méthode présentée dans [AMWH94, Mue00] distingue quatre catégories d'accès aux instructions selon que l'on peut ou non garantir statiquement leur présence dans le cache d'instructions au moment de leur exécution. Les catégories d'instructions sont : *always hit*, *always miss*, *first miss* et *conflict*. Les accès en mémoire aux instructions classées *always hit* sont toujours des succès, et les accès aux instructions classées *miss* sont considérés comme des échecs. La catégorie *first miss* indique une instruction qui cause un échec uniquement la première fois qu'elle est exécutée (dans une boucle par exemple). Enfin, la dernière catégorie, *conflict*, indique qu'un accès à une instruction est considéré comme un échec car on ne peut obtenir d'estimation sûre de son comportement. Le classement des instructions du programme dans ces catégories est réalisé en deux temps : (i) le graphe de flot de contrôle est utilisé pour calculer les états

abstraits du cache (les représentations des contenus possibles du cache avant l'exécution des blocs de base), puis (ii) les états abstraits sont analysés pour classer les instructions.

Ces catégories, ainsi que leur méthode d'obtention, sont largement détaillées dans le paragraphe 4.3 consacré à l'adaptation de la simulation statique de cache à un nouveau cadre d'analyse statique, c'est pourquoi nous ne les détaillons pas plus ici. Le classement des instructions est réalisé en tenant compte des différentes instances des fonctions et procédures (voir § 2.1.2.4), c'est-à-dire que pour deux appels d'une même fonction à deux endroits différents du programme, on distingue deux catégories pour chacune des instructions de la fonction. En effet, le classement des instructions d'une fonction dépend du point d'appel de cette fonction, la réplication des fonctions dans la représentation logique permet cette distinction.

2.3.1.2 Prise en compte du cache par interprétation abstraite

La deuxième méthode, présentée dans [AFMW96, FMW97] et basée sur l'analyse de programme par interprétation abstraite [CC77], distingue les mêmes catégories que la précédente. Elle ne considère pas les instances de fonctions. Par contre elle sépare la première itération d'une boucle des itérations suivantes. Deux analyses sont en fait réalisées : l'analyse *Must* pour déterminer si un bloc mémoire est *définitivement* présent dans le cache, et l'analyse *May* pour vérifier qu'un bloc mémoire n'est *jamais* dans le cache. Les résultats de ces deux analyses permettent la classification des blocs mémoires considérés (et donc des instructions).

2.3.1.3 Prise en compte du cache par *IPET*

Enfin, une technique de prise en compte du cache basée sur une analyse statique à énumération implicite des chemins d'exécution a été proposée par Li et al. dans [LMW95a, LMW96]. Cette technique consiste à ajouter un nouveau jeu de contraintes pour modéliser le comportement du cache d'instructions, et à modifier les contraintes représentant la structure du graphe de flot de contrôle pour prendre en compte les deux temps d'exécution de chaque instruction correspondant aux deux cas possibles de lecture (*hit* et *miss*). Cette technique étant principalement une modification du système de contraintes généré par le niveau haut de l'analyse, les détails la concernant sont présentés au paragraphe 2.3.3.2, qui traite de l'intégration des techniques de prise en compte de l'architecture matérielle dans les méthodes d'analyse de haut niveau.

2.3.2 Prise en compte de l'exécution pipelinée

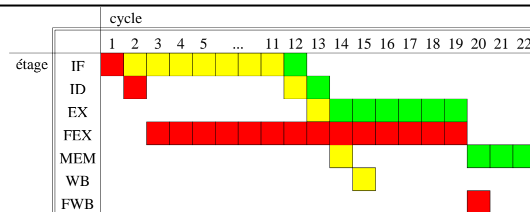
Le pipeline est une technique dans laquelle plusieurs instructions se recouvrent au cours de leur exécution. C'est la technique fondamentale utilisée pour réaliser des processeurs rapides.

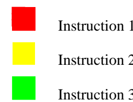
Afin d'expliciter brièvement le principe du pipeline, on rappelle que le cycle d'exécution d'une instruction se décompose en plusieurs étapes, par exemple : lecture d'instruction, décodage d'instruction, exécution, accès mémoire, écriture du résultat. Le but du pipeline est d'intro-

duire du parallélisme entre les traitements de différentes instructions. Pour ce faire, il comporte plusieurs étages qui lui permettent de traiter en parallèle les différentes étapes des instructions. La figure 2.10 présente une table représentant l'occupation d'un pipeline de 7 étages par une séquence de 3 instructions sur un processeur *MicroSPARC*.

Étages du pipeline
IF, Instruction Fetch
ID, Instruction Decode
EX, EXecution
FEX, Floating EXecution
MEM, MEMory access
WB, Write Back
FWB, Floating Write Back

Occupation du pipeline par les instructions 1,2 & 3





Instructions SPARC			
I 1,1	:	faddd	%f2,%f0,%f2
I 1,2	:	sub	%o4,%g1,%i2
I 1,3	:	std	f2,[%o0+8]

FIG. 2.10 – *Occupation du pipeline du processeur MicroSPARC I*

Les exécutions des instructions se chevauchent et le temps d'exécution de deux instructions dans le pipeline n'est pas la somme de leurs temps d'exécution unitaires, car l'exécution d'une instruction peut débuter avant la fin de celle de l'instruction précédente.

Le parallélisme d'instruction n'est pas maximum car l'occupation du pipeline par une séquence d'instructions peut être perturbée par des aléas (*hazard*) :

- dépendance de donnée (*data hazards*) quand une instruction utilise le résultat d'une instruction précédente dont l'exécution n'est pas encore terminée,
- aléa de contrôle (*control hazards*) lorsque le résultat d'une instruction est un branchement pris (rupture de séquence par un saut).
- aléa de structure (*structural hazards*) lorsqu'il y a un conflit d'accès à une ressource entre plusieurs instructions dans différents étages du pipeline.

Les effets du pipeline sur le calcul du WCET se retrouvent à deux niveaux : (i) intra blocs de base pour la prise en compte du chevauchement entre instructions et des aléas de données

et (ii) inter blocs de base où les aléas de contrôle sont pris en compte.

2.3.2.1 Influence sur le WCET des blocs de base

De par la définition d’un bloc de base, on sait qu’aucun aléa de contrôle ne peut avoir lieu (pas de rupture de séquence), mais la présence de dépendances de donnée et d’aléas de structure est possible. Pour prendre en considération ces deux aspects, le calcul du WCET des blocs de base présenté dans [RLM+94] repose sur la représentation de l’occupation du pipeline pendant l’exécution de ce bloc. Connaissant les occupations des étages du pipeline par les instructions, un ajustement est effectué afin de prendre en compte les aléas ; la table de réservation ainsi obtenue fournit le WCET du bloc de base.

La figure 2.10 présente une illustration de cette technique. La première instruction réalise une addition flottante qui nécessite 20 cycles. Le résultat de cette opération est utilisé par la troisième instruction, il y a donc une dépendance de donnée entre la première et la troisième instruction. C’est pourquoi le traitement de cette dernière dans l’étage MEM du pipeline est retardée jusqu’à la fin de l’opération flottante. Le temps d’exécution est, sur cet exemple, imposé par la dépendance de donnée entre les instructions 1 et 3.

Après avoir analysé l’effet du pipeline sur chaque bloc de base, on dispose de descripteurs d’occupation (des tables de réservation [Kog81]) du pipeline pendant la durée nécessaire à leur exécution.

2.3.2.2 Effet inter-bloc de base

On s’intéresse ici à la concaténation avec recouvrement de ces descripteurs, et pour cela le début et la fin des descripteurs suffisent (*cf.* figure 2.11). Un descripteur peut donc se limiter aux extrémités du descripteur d’occupation du pipeline de la figure 2.10.

À cause des dépendances de donnée et des aléas de contrôle, la durée d’exécution d’un bloc de base dépend du bloc exécuté avant. Dans l’hypothèse de l’absence de tout mécanisme de prédiction de branchement, le calcul du WCET global par composition des WCET des blocs de base est réalisé de la manière suivante. Si le long du chemin d’exécution considéré, deux blocs de base en séquence se suivent réellement dans le programme (instructions contigües en mémoire), alors une composition des descripteurs d’occupation du pipeline par recouvrement permet un calcul plus précis du WCET. Cette composition a pour but de réduire la durée d’exécution estimée de la séquence constituée des deux blocs de base. Les descripteurs sont donc ajustés (*cf.* figure 2.11.a) et se recouvrent tout en respectant les contraintes d’occupation des différents étages du pipeline. Sinon, on se trouve dans le cas où la mise en séquence des deux blocs provient d’un saut (*i.e.* les adresses des instructions des blocs ne se suivent pas). Dans ce cas, le pipeline est inefficace, il est vidé pour reprendre l’exécution à partir de l’instruction cible du saut. Il suffit alors de concaténer les descripteurs d’occupation du pipeline (*cf.* figure 2.11.b).

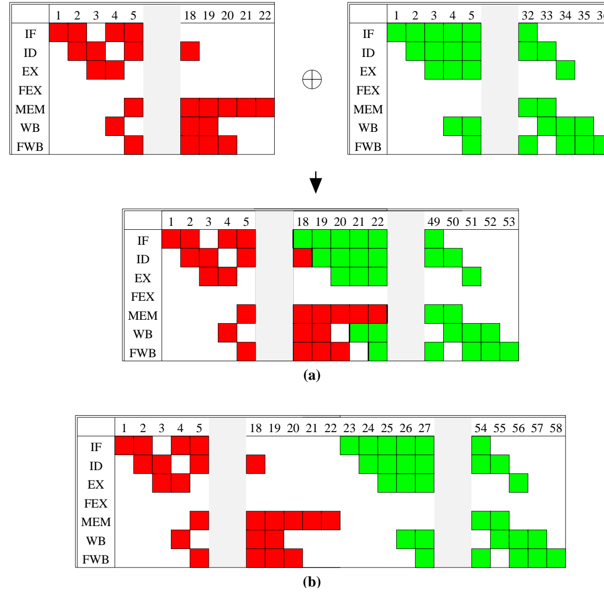


FIG. 2.11 – *Composition de descripteurs de pipeline*

2.3.3 Intégration des techniques de prise en compte de l'architecture

Nous présentons ici un aperçu de l'intégration des techniques de prise en compte de l'architecture dans chacune des classes de méthodes d'analyse présentées au paragraphe 2.2.

2.3.3.1 Adaptation de la méthode *tree-based*

Pour permettre les prises en compte du cache d'instruction et du pipeline décrites dans les paragraphes précédents, l'ensemble des formules de calcul récursif du WCET constituant le schéma temporel vu au paragraphe 2.2.3 (*cf.* table 2.1) doit être modifié. Cette modification, exposée dans [KMH96], permet de ne plus manipuler directement le WCET des structures de contrôle (comme dans [PK89]) mais des ensembles de WCTA (*Worst Case Timing Abstraction*). Les WCTA, plus complexes que les temps bruts représentés par les WCET, combinent les représentations de l'occupation du pipeline et de l'état du cache.

Le tableau 2.2 présente l'adaptation du schéma temporel du tableau 2.1 aux WCTA. La définition d'opérateurs ensemblistes ($\bigoplus$ et $\bigcup$) sur les WCTA permet ainsi le calcul des WCTA des structures de contrôle dans un contexte d'exécution, et à partir des WCTA, le calcul du WCET à la racine de l'arbre.

Pour illustrer la fonction des opérateurs $\bigoplus$ et $\bigcup$, prenons le cas du pipeline. Supposons deux sous-arbres, A et B, dont les WCTA sont W_A et W_B. Le WCTA de la séquence S composée de A et B est : $WCTA(S) = W_A \bigoplus W_B$. Chaque WCTA peut comporter plusieurs représentations différentes d'occupation du pipeline. L'opérateur $\bigoplus$ sur les ensembles

	Formules pour le calcul du WCTA : W(S)
$S : S_1; S_2$	$W(S) = W(S_1) \bigoplus W(S_2)$
S : if (exp) then S_1 else S_2	$W(S) = (f(exp) \bigoplus W(S_1)) \bigcup (W(exp) \bigoplus W(S_2))$
S : while (exp) S_1	$W(S) = (\bigoplus_{i=1}^{N}(W(exp) \bigoplus W(S_1))) \bigoplus W(exp)$
$S : f(exp_1, \cdots, exp_n)$	$f(S) = W(exp_1) \bigoplus \cdots \bigoplus W(exp_n) \bigoplus W(f())$

TAB. 2.2 – *Schéma temporel étendu pour prendre en compte les caches et le pipeline*

de représentation de pipeline est défini en utilisant l'opérateur $\oplus$ de concaténation de deux représentations de pipeline :

$$W_A \bigoplus WB = \{w_A \oplus w_B | w_A \in W_A, w_B \in W_B\}$$

Une sélection des résultats est ensuite effectuée pour ne garder que les résultats significatifs pour le calcul du WCET.

Les WCTA comportent aussi des informations concernant le comportement des instructions vis à vis du cache d'instruction. Un premier ensemble contient les adresses des instructions dont les références causeront un *hit* ou un *miss* dans le cache d'instructions en fonction du contenu du cache avant l'exécution du code auquel est associé le WCTA. Le deuxième ensemble contient les adresses des instructions qui resteront dans le cache après l'exécution du code associé au WCTA.

2.3.3.2 Adaptation de la méthode *IPET*

L'adaptation d'une technique d'analyse statique dite *IPET*, c'est-à-dire basée sur la résolution d'un système de contraintes pour la prise en compte du cache d'instructions et du pipeline a été proposée dans [LMW95a, OS97].

La prise en compte du pipeline se fait par l'utilisation de tables de réservation, et l'effet inter-bloc de base (*cf.* § 2.3.2.2) peut être pris en compte [OS97], ou non [LMW95a]. Les valeurs de w_i représentant les WCET des blocs de base dans le système de contraintes sont modifiées pour prendre en compte l'effet du pipeline.

Pour prendre en compte l'effet du cache d'instructions, l'expression du WCET à maximiser (originalement $\sum_i n_i \times w_i$) est adaptée pour refléter (i) le fractionnement des blocs de base dans le cache et (ii) les deux temps d'exécutions possibles de ces fragments de blocs de base. Dans [LMW95a], un bloc de base i donne N_i fragments correspondants aux lignes de cache occupées. À chaque fragment j d'un bloc de base i sont associés deux WCET, $w_{i,j}^{hit}$ et $w_{i,j}^{miss}$, pour les cas *hit* et *miss*. Et le nombre d'occurrences du bloc de base (originalement n_i) est partagé : $n_i = n_{i,j}^{hit} + n_{i,j}^{miss}$. La nouvelle expression du WCET à maximiser est alors :

$$\sum_i \sum_j^{N_i} n_{i,j}^{hit} \times w_{i,j}^{hit} + n_{i,j}^{miss} \times w_{i,j}^{miss} \leq \sum_i n_i \times w_i$$

Un nouveau jeu de contraintes est généré et ajouté au système existant pour le calcul des valeurs des $n_{i,j}^{hit}$ et $n_{i,j}^{miss}$. Ces nouvelles contraintes représentent le comportement des fragments de bloc de base vis à vis du cache d'instructions. Par exemple, un fragment de bloc de base qui cause un *miss* lors de sa première exécution puis qui reste toujours dans le cache est représenté par : $n_{i,j}^{miss} \leq 1$.

2.3.4 Prise en compte de quelques autres éléments d'architecture

Les travaux de recherche visant à prendre en compte l'effet de l'architecture matérielle sur l'analyse de WCET se sont principalement intéressés au cache d'instructions et au pipeline. Mais d'autres aspects architecturaux ont été étudiés au nombre desquels : le mécanisme de DMA [HL95], le cache de données [LMW96, WMH+99, KMH96, FW98], le mécanisme de prédiction des branchements [CP00], les processeurs super-scalaires [SF99, LHM97, CLK94] et l'exécution dans le désordre [LS99]. Nous allons maintenant décrire brièvement ces travaux.

2.3.4.1 Cache de données

La prise en compte du cache de données est plus complexe que celle du cache d'instructions, car d'une part il est difficile ou même parfois impossible de connaître statiquement la plupart des adresses des variables d'un programme, et d'autre part ce cache peut-être accédé aussi bien en lecture qu'en écriture. Plusieurs problèmes se posent donc : (i) déterminer pour quelles variables les adresses peuvent être connues statiquement, (ii) calculer leurs adresses lorsque c'est possible, et (ii) gérer les accès en écriture.

Identification statique des variables utilisées
L'analyse de programme utilisée dans [LMW96] permet de différencier les variables statiques (qui ont une adresse fixe) des autres variables, dites dynamiques (dont l'adresse n'est connue qu'à l'exécution).

Dans [KMH96], une référence dynamique est toujours considérée comme un accès *miss* et induit deux pénalités de cache : une pour l'absence de cette référence dans le cache et une deuxième pour la prise en compte du préjudice éventuel causé par le remplacement d'un bloc mémoire présent dans le cache. Dans ce cas, la distinction entre variables statiques et dynamiques est très importante.

Une autre possibilité est de n'autoriser à être mises en cache, que les variables statiques.

La solution proposée dans [WMH+99] permet de connaître statiquement les adresses d'un sous-ensemble des variables dynamiques : les variables locales (allouées dynamiquement dans la pile d'exécution). Cette méthode se base sur la simulation statique du niveau de la pile d'exécution pour estimer à tout moment l'adresse du sommet de pile et donc l'adresse des variables locales allouées en début de fonction.

Gestion des accès en écriture

Dans le cas du cache d'instructions, seules les lectures dans le cache ont été considérées. Dans le cas du cache de données les écritures dans le cache doivent être prises en compte. Les différentes techniques de gestion du cache de données lors des écritures (*write-back*, *write-through*) induisent différents temps pour les accès en écriture (*cf.* [HP94]). Ces différents temps doivent être pris en compte par l'analyse statique de WCET.

2.3.4.2 Processeurs super-scalaires

Une autre technique d'accélération consiste à disposer plusieurs pipelines en parallèle de manière à augmenter le débit de traitement des instructions. La modélisation de l'exécution en parallèle de plusieurs instructions a été étudiée dans [SF99, LHM97, CLK94]. Les nouveaux problèmes posés par l'aspect super-scalaire des processeurs sont : (i) l'appariement des instructions pour savoir quelles instructions peuvent être exécutées en parallèle, (ii) les dépendances structurelles entre instructions (conflits de ressource) et (iii) les dépendances de données.

2.3.4.3 Exécution dans le désordre

Le modèle d'exécution super-scalaire permet d'extraire et d'exécuter des instructions indépendantes d'une fenêtre de largeur limitée. Ce modèle préserve l'ordre des modifications de l'état de la machine. Mais lorsque les longueurs des pipelines dépassent certaines limites, seule une exécution dans le désordre permet d'obtenir des performances satisfaisantes. Cette technique consiste à exécuter les instructions dès que possible sans attendre la terminaison des précédentes.

Avec l'exécution dans le désordre, l'hypothèse selon laquelle le pire temps d'exécution d'une instruction correspond à son pire comportement vis à vis des éléments architecturaux est remise en cause. Il a été montré dans [LS99] que cette hypothèse est fausse si l'on prend en compte l'exécution dans le désordre. Un *miss* dans le cache peut dans certains cas donner un temps d'exécution de l'instruction plus court qu'un *hit*. La méthode de prise en compte des processeurs super-scalaires de C. Ferdinand et J. Schneider [SF99], prend en compte l'exécution dans le désordre.

2.3.4.4 Prédiction de branchement

Un des derniers éléments d'architecture a avoir été pris en compte est le mécanisme de prédiction des branchements [CP00, CP01b] qui permet d'augmenter encore le taux d'occupation des pipelines en réduisant le nombre de passages à vide dans le pipeline dûs aux aléas de contrôle.

La méthode que nous avons mise au point pour prendre en compte cet élément des architectures des processeurs modernes, pour l'analyse statique du WCET, fait partie des idées originales proposées dans cette thèse. Une description détaillée de cette méthode se trouve au paragraphe 4.4.

2.4 Récapitulatif

Nous avons présenté dans ce chapitre un large panorama des techniques d'estimation du WCET. Les techniques d'analyse statique de WCET peuvent être différenciées selon cinq caractéristiques principales :

- le niveau du langage source analysé (langage de haut niveau, langage objet),
- la méthode d'obtention des informations sur les chemins d'exécutions (annotations manuelles ou obtention automatique),
- la ou les représentations de programme utilisées (graphe de flot de contrôle, *T-graph* ou arbre syntaxique),
- la méthode d'énumération des chemins d'exécution (*path-based*, *IPET* ou *tree-based*),
- les éléments d'architecture pris en compte.

Les différentes techniques de prise en compte du matériel par l'analyse bas niveau peuvent être classées en deux catégories. La première est l'analyse bas niveau *globale*, qui considère l'impact des éléments d'architecture ayant un effet non local. C'est le cas par exemple des caches et la prédiction de branchement. La deuxième catégorie est l'analyse bas niveau *locale*, qui s'intéresse aux éléments n'ayant qu'un effet local (*i.e.* sur une instruction et son voisinage immédiat). C'est le cas par exemple du pipeline.

Le tableau 2.3 dresse un panorama des principaux travaux de recherche concernant l'analyse de WCET dans le monde.

La colonne II indique la méthode d'obtention des informations complémentaires sur les chemins d'exécution (a = automatique, m = manuelle). La colonne III liste les éléments architecturaux pris en compte par l'analyse bas niveau globale (i = cache d'instructions, d = cache de données, b = prédiction de branchement). La colonne IV liste les éléments architecturaux pris en compte par l'analyse bas niveau locale (p = pipeline simple, s = superscalaire). Enfin, la colonne V indique la méthode de recherche du plus long chemin d'exécution utilisée (P = *path-based*, T = *tree-based*, I = *IPET*).

Toutes ces méthodes d'estimation du WCET basées sur l'analyse statique n'étaient, il y a quelques années encore, étudiées et envisagées avec intérêt que dans un milieu académique. Le monde industriel n'utilisait que des techniques à base de tests et mesures. On voit cependant apparaître depuis peu quelques outils, commerciaux ou non, d'analyse statique de WCET. Ces outils représentent les premières applications industrielles d'une quinzaine d'années de recherche sur le domaine de l'analyse statique de WCET.

Les analyseurs statiques de WCET développés pour une utilisation industrielle à notre connaissance sont :

- L'analyseur de temps d'exécution **Bound-T** de la société *Space Systems Finland Ltd*[1] Cet analyseur n'utilise que le graphe de flot de contrôle du programme (pas l'arbre syntaxique), ce qui lui permet d'être indépendant du langage de haut niveau (il analyse le

1. *http://www.ssf.fi*

	I	II	III	IV	V	
C-Lab (Allemagne)	Altenbernd et al.	a	i	p	P	[AFMW96, SA00]
University of York (Angleterre)	Bernat et al.	m	-	p	T	[BBW00, BBMP00]
University of York (Angleterre)	Chapman et al.	a,m	-	-	P	[CBW96]
IRISA (France)	Colin et al.	m	i,b	p	T	[CP00, CP01b]
Artes (Suéde)	Engblom et al.	a,m	i	p	I	[OS97, HSRW98, Eng99, EE00]
Universität des Saarlandes (Allemagne)	Ferdinand et al.	m	i	p,s	I	[FMW97, SF99]
Princeton University (USA)	Li et al.	m	i	p	I	[LMW95a, LMW96]
Seoul National University (Corée)	Lim et al.	m	i,d	p,s	T	[LML^{+}94, KMH96, LHM97]
Indiana University (USA)	Liu et al.	a	-	-	T	[LG98]
Artes (Suède)	Lundqvist et al.	a	i	p	P	[LS98]
University of washington (USA)	Park et al.	m	-	-	T	[PS91, Par93]
Lund University (Suède)	Persson et al.	m	-	-	T	[PH99]
Technische Universität Wien (autriche)	Puschner et al.	m	-	-	T,I	[PK89, PS95]
Florida University (USA)	Whalley et al.	a,m	i,d	p	P	[HSRW98, HAM^{+}99, WMH^{+}97, Mue97]

TAB. 2.3 – *Panorama des principaux travaux de recherche concernant l'analyse de WCET*

code objet). L'obtention des informations complémentaires sur les chemins d'exécution est semi-automatique (l'obtention des bornes de boucles est automatique pour les boucles dites à compteurs). La recherche du pire chemin d'exécution est basée sur une méthode *IPET*. Les différents processeurs cibles de cet outil sont : l'Intel-8051 (8 bits), deux DSPs 32 bits et le SPARC V7. Un point faible de cet analyseur est l'absence à ce jour de prise en compte de l'architecture matérielle.

– L'environnement d'analyse statique de la société *AbsInt*[2] (fondée en 1998 et issue de l'*Universität des Saarlandes*) offre des outils d'analyse statique du cache d'instruction, de simulation de pile d'exécution, et un générateur d'analyseur statique de programme. *AbsInt* ne propose donc pas d'analyseur statique de WCET mais propose les outils pour en construire un.
– L'analyseur statique de WCET *Cinderella 3.0* de l'université de princeton[3] est l'outil académique le plus avancé. Il utilise une méthode *IPET* et prend en compte les effets du cache d'instructions et du pipeline des processeurs Intel I960KB et Motorola 68000. Les informations complémentaires sur les chemins d'exécution (*e.g.* les bornes sur le nombre d'itérations des boucles) sont demandées interactivement à l'utilisateur.

2. *http://www.absint.com*
3. *http://www.ee.princeton.edu/ yauli/cinderella-3.0*

deuxième partie

Analyse statique de WCET

tree-based

Les propositions exposées dans les chapitres suivants ont été élaborées dans le cadre des méthodes d'analyse statique dites *tree-based*. Les représentations de programmes utilisées sont celles présentées au chapitre précédent. Le code assembleur est réparti en blocs de base, et on utilise conjointement les représentations par graphe de flot de contrôle et arbre syntaxique (dont la correspondance est requise).

Nos propositions portent sur l'amélioration de la précision des estimations de WCET obtenues par analyse statique *tree-based* d'une part (chapitres 3 et 4), et la structure interne des outils d'analyse statique *tree-based* d'autre part (chapitre 5). Le dernier chapitre de cette deuxième partie (chapitre 6) décrit les résultats des expérimentations conduites sur du code simple d'une part, et le code d'un système d'exploitation temps-réel d'autre part.

Chapitre 3

Réduction du pessimisme de l'analyse par annotation symbolique des boucles

3.1 Introduction

La précision de l'estimation du WCET d'un programme dépend, entre autres, de la précision avec laquelle on est capable d'identifier le pire chemin d'exécution faisable. Pour pouvoir identifier le plus précisément possible ce pire chemin dans le programme analysé, on doit entre autres connaître le nombre maximum d'itérations de toutes les boucles du code source. On distingue alors deux possibilités d'obtention de ces informations : (*i*) l'obtention automatique de bornes sur le nombre d'itérations des boucles par analyse statique, et (*ii*) l'annotation manuelle des boucles soit dans le code par l'utilisateur, de façon séparée ou encore interactivement. Nous avons déjà présenté la première méthode au paragraphe 2.1.1.3 et même si ses avantages sont nombreux (sûreté, automatisme), elle reste difficile à mettre en œuvre, et coûteuse en temps de traitement (requiert une analyse statique flot de donnée). De plus, comme le montrent les exemples de la figure 3.1, elle n'est pas toujours applicable. C'est pourquoi nous nous concentrons ici sur la deuxième possibilité d'obtention du nombre maximum d'itérations des boucles.

Les informations concernant les boucles, fournies par l'utilisateur par annotation du code source, sont le plus souvent de simples constantes qui indiquent directement le nombre maximum d'itérations des boucles auxquelles elles sont associées. Comme on le verra par la suite, ce type d'annotation a ses limites, notamment en ce qui concerne la prise en compte d'un type particulier de boucles, les *boucles non rectangulaires*, définies au paragraphe 3.2. Les techniques à base d'annotations constantes entraînent sur ce type de boucle une surestima-

```
char c;
FILE * f = fopen("file","r");
while(! feof(f)) {
      fread(&c,1,1,f);
      putchar(c);
}
```

```
unsigned char string[256];
unsigned char c;
int n=0;

// get dos filename (12 chars)
while ( (c=getchar()) != 10 )
   string[n++]=c;

string[n]=0;
```

FIG. 3.1 – *Exemples de boucles dont l'obtention automatique des bornes est impossible*

tion de la longueur du pire chemin d'exécution. Notre proposition d'annotations adaptées aux boucles non rectangulaires est décrite au paragraphe 3.3. Enfin, l'intégration de notre proposition d'annotation dans une analyse de WCET à base d'arbre est présentée au paragraphe 3.4.

3.2 Les boucles non rectangulaires

3.2.1 Définition

Le nombre maximum d'itérations des boucles n'est pas leur seule caractéristique. Dans le code source du programme à analyser, les boucles sont caractérisées par :

- le *nombre maximum d'itérations* (noté *maxiter*),
- l'*index de boucle*, qui évolue de 0 à $maxiter-1$,
- les *compteurs de boucle.*

Le rôle de l'index de boucle est de compter les itérations de la boucle. Les compteurs de boucle permettent de représenter l'évolution de certaines variables de boucle particulières (les compteurs).

Dans l'exemple ci-dessous, le *nombre maximum d'itérations* de la boucle est 10, l'*index de boucle* prend successivement toutes les valeurs entre 0 et 9 pour compter les itérations, enfin i et j sont les *compteurs de boucle.* Le compteur i évolue de 0 à 9 au pas de 1, et le compteur j évolue de 5 à 23 de 2 en 2.

☞ *Exemple*:

```
for (i=0,j=5;i<10;i++,j+=2)
```

Soient deux boucles, E (pour englobante) et I (pour interne), I emboitée dans E. Considérons que la boucle E est exécutée $maxiter_E$ fois. Les différentes valeurs du couple $< compteur_E, compteur_I >$ peuvent être représentées par le polyèdre à deux dimensions qui les contient toutes.

Si le nombre maximum d'itérations de la boucle interne ($maxiter_I$) est indépendant du numéro de l'itération en cours de la boucle englobante (*i.e.* ne dépend pas de $compteur_E$) alors le polyèdre représentant ces deux boucles est un rectangle de taille $maxiter_E \times maxiter_I$. On parle alors de boucle rectangulaire (*cf.* exemple figure 3.2.a).

Dans le cas où le paramètre $maxiter_I$ de la boucle interne I est une fonction de la valeur du compteur de la boucle englobante E, on parle de boucle non rectangulaire (*cf.* exemple figure 3.2.b). La forme du polyèdre correspondant aux boucles non rectangulaires dépend de la fonction définissant le nombre maximum d'itérations de la boucle interne. Par exemple, les boucles triangulaires sont un cas simple où le paramètre $maxiter_I$ de la boucle interne est une fonction linéaire du compteur de la boucle englobante ($maxiter_I = a \times compteur_E + b$).

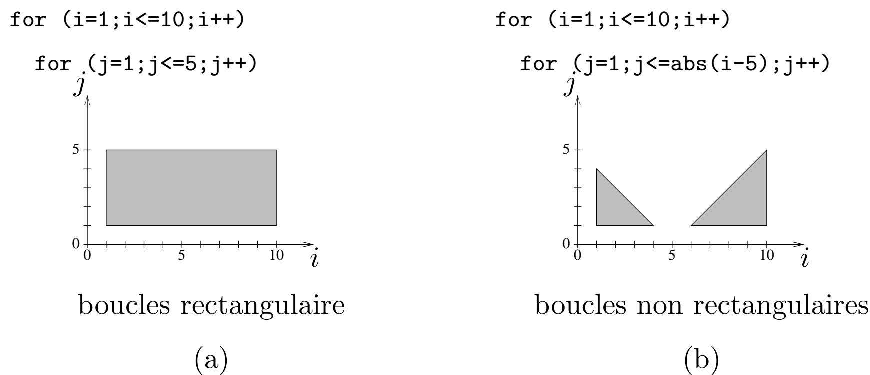


FIG. 3.2 – *Exemple de boucles emboitées*

Une boucle B peut être emboitée dans plusieurs boucles englobantes (*cf.* exemple ci-dessous), dans ce cas il suffit que $maxiter_B$ dépende d'au moins un compteur de boucle d'une des boucles englobantes pour que la boucle B soit non rectangulaire. Cet exemple présente une boucle non rectangulaire dont le *maxiter* dépend des compteurs des deux boucles l'englobant.

☞ *Exemple*:

```
for(a=0;a<10;a++)
  for(b=0;b<10;b++)
    for (c=0;c<a+b;c++)
```

Une boucle est dite *n*on rectangulaire si son nombre maximum d'itérations est fonction d'au moins un compteur d'une boucle l'englobant, et si cette fonction n'est pas une fonction constante.

3.2.2 Importance de la prise en compte du caractère non rectangulaire des boucles

Si on considère une boucle comme rectangulaire alors qu'elle ne l'est pas, on surestime son nombre d'itérations. La figure 3.3 illustre ce cas. Sur cet exemple, seuls les points appartenant au polyèdre gris sont effectivement obtenus à l'exécution, et la surestimation est de 25

itérations sur 50.

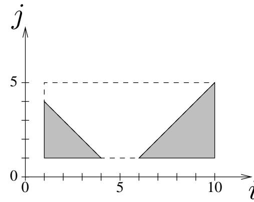


FIG. 3.3 – *Surestimation du nombre d'itérations des boucles non rectangulaires*

De plus, pour des boucles non rectangulaires fortement emboitées, la surestimation aura un impact très important sur le WCET car elle est multipliée par le *maxiter* de chacune des boucles englobantes. Ainsi, sur l'exemple suivant, la surestimation du nombre d'itérations de la boucle la plus emboitée (considérée de manière isolée) est d'un facteur 1% (on estime 1 itération de trop par rapport aux 100 itérations estimées), ce qui peut sembler faible mais qui donne une surestimation de 10000 itérations de la boucle c $(1 \times 100 \times 100)$ pour tout le programme.

☞ *Exemple* :

```
for(a=0;a<100;a++)
    for(b=0;b<100;b++)
        for (c=0;c<100;(c+1==b)?c+=2:c++)
```

3.3 Une nouvelle technique d'annotation

Nous avons montré que les annotations constantes représentent le comportement des boucles non rectangulaires de façon pessimiste. Notre proposition [CP00] consiste à remplacer ces annotations constantes par des expressions symboliques qui permettent de représenter précisément le comportement des boucles non rectangulaires.

3.3.1 Annotation symbolique

Nous proposons d'annoter chaque boucle par une liste d'expressions symboliques :

$$[M, C^1, \ldots, C^n]$$

- M est l'expression représentant le paramètre *maxiter* de la boucle. Elle peut être exprimée en utilisant les expressions M et C^i associées aux boucles englobantes. Si la boucle à annoter est une boucle rectangulaire, M est une constante.
- C^i représente la valeur de la $i^{\text{éme}}$ variable compteur de la boucle. C'est une fonction de la variable λ qui prend successivement toutes les valeurs entre 0 et $M-1$. Tout comme

l'expression M, elle peut être exprimée en utilisant les expressions M et C^i des boucles englobantes.

À une boucle B on associe les expressions M_B et $C_B^1 \ldots C_B^n$. Lorsque la boucle ne comporte qu'un seul compteur de boucle (cas le plus courant) l'annotation est de la forme $[M_B, C_B]$.

L'exemple de la figure 3.4 illustre l'utilisation des annotations symboliques.

Boucles	Annotations
`for (a=0 ; a<10 ; a++)`	$[\ M_a = 10\ ,\ C_a = \lambda\ ]$
`  for (b1=0,b2=0;b1<a;b1++,b2+=a)`	$[\ M_b = C_a\ ,\ C_b^1 = \lambda\ ,\ C_b^2 = C_a \times \lambda\ ]$
`    for (c=0;c<b1+b2;c++)`	$[\ M_c = C_b^1 + C_b^2\ ,\ C_c = \lambda\ ]$

FIG. 3.4 – *Trois boucles emboitées et les annotations correspondantes*

La boucle externe (boucle a) itère au plus 10 fois, donc $M_a = 10$ et la variable compteur C_a prend successivement toutes les valeurs entre 0 et 9 ce qui est décrit par $C_a = \lambda$. La boucle du milieu (boucle b) a la particularité d'avoir deux variables compteurs de boucle (`b1` et `b2`) dont les évolutions respectives sont décrites par les expressions symboliques C_b^1 et C_b^2. De plus, le *maxiter* de cette boucle est égal à C_a qui varie entre 0 et $M_a - 1$, M_b est donc une fonction non constante et la boucle b est non rectangulaire. La boucle la plus emboitée (boucle c) est elle aussi non rectangulaire, et son maxiter a la particularité d'être une fonction de deux compteurs (ceux de la boucle b).

Les expressions constituant une annotation symbolique seront évaluées par un outil de calcul symbolique (*e.g.* Maple [CGG91]). L'utilisateur peut donc inclure dans ses annotations toutes les fonctions connues de l'outil (racine, exponentielle, etc.), ou bien définir ses propres fonctions.

De plus, l'évaluation du WCET étant symbolique, il est possible d'utiliser des symboles arbitrairement choisis par l'utilisateur dans les annotations de boucle. Ces symboles apparaissent alors dans les annotations de boucle et les formules de calcul du WCET qui en découlent. Après évaluation, les symboles de l'utilisateur restent non-instanciés, et le WCET résultant est un WCET symbolique.

Cette possibilité permet d'obtenir différentes valeurs de WCET selon certains paramètres non instanciée lors de la phase d'analyse, sans avoir à recommencer l'analyse statique de WCET. Par exemple, si un programme comporte la boucle

`for (a=0;a<tabsize;a++)` $[M_a = TABSIZE, C_a = \lambda]$

son WCET sera par exemple :

$$WCET(prog) = 37523 + (TABSIZE \times 327)$$

3.3.2 Nommage relatif des expressions symboliques et pile des annotations

Le fait de pouvoir utiliser une ou plusieurs expressions symboliques des boucles englobantes pour exprimer les paramètres d'une boucle emboitée nécessite un système de nommage unique des expressions, permettant de les identifier sans ambiguïté. Comme on l'a vu, à chaque boucle est associée une liste d'expressions symboliques (au minimum deux) : M et C^i. Pour identifier toutes ces expressions, il suffit de nommer les boucles. Il y a alors deux possibilités :

– un nom absolu qui permet d'identifier chaque boucle du code analysé de manière unique,

– un nom relatif pour identifier uniquement les boucles englobantes d'une boucle.

Nous avons préféré cette deuxième solution qui consiste à numéroter toutes les boucles englobant la boucle dont on veut exprimer les paramètres M et C^i en prenant celle-ci comme référence.

Le nommage absolu implique la vérification de l'unicité des noms donnés aux boucles, ce que nous évitons ici. De plus, le nommage relatif permet de décrire le cas d'une boucle dont le nombre maximum d'itérations dépend des compteurs de différentes boucles en fonction du chemin d'exécution. Ce cas est illustré sur la figure 3.5. La boucle de la fonction f est exécutée à chaque itération des deux boucles de la fonction g, et son nombre maximum d'itérations dépend du compteur de la boucle où a lieu l'appel de f. L'utilisation des noms relatifs permet de prendre en compte ce cas de figure en spécifiant que le *maxiter* de la boucle de f dépend du compteur de la boucle qui la contient immédiatement.

```
void f(int M) {
  int i;
  for(i=0;i<M;i++) {
    ...
  }
}
```

```
void g()
  int j;
  for(j=0;j<10;j++) {
    f(j);
  }
  for(j=0;j<50;j+=2) {
    f(j);
  }
}
```

FIG. 3.5 – *maxiter dépendant du chemin d'exécution*

Considérons la boucle c de la figure 3.4. Pour annoter cette boucle, on a accès aux annotations des boucles englobantes par l'intermédiaire d'une *pile d'annotations* P. Pour la boucle c, l'élément au sommet de cette pile ($P[0]$) est l'annotation de la boucle qui l'englobe immédiatement (la boucle b dans l'exemple). De même, l'élément au fond de la pile ($P[\perp]$) est l'annotation de la boucle englobante la plus externe (la boucle a dans l'exemple). La pile d'annotations permet d'annoter une boucle en utilisant les expressions $P[x].M$ et $P[x].C^i$ (avec $0 \leq x \leq \perp$) des annotations d'une ou plusieurs boucles englobantes.

La figure 3.6 reprend le code présenté à la figure 3.4 pour comparer les annotations utilisant des noms absolus et celles utilisant des noms relatifs et la pile d'annotations.

Boucles	Noms absolus	Noms relatifs	Pile P
`for (a=0 ; a<10 ; a++)`	$[10, \lambda]$	$[10, \lambda]$	$[]$
`for (b1=0,b2=0;b1<a;b1++,b2+=a)`	$[C_a, \lambda, C_a \times \lambda]$	$[P[0].C, \lambda, P[0].C \times \lambda]$	$\begin{bmatrix} M_a, C_a \end{bmatrix}$
`for (c=0;c<b1+b2;c++)`	$[C_b^1 + C_b^2, \lambda]$	$[P[0].C^1 + P[0].C^2, \lambda_c]$	$\begin{bmatrix} M_b, C_b^1, C_b^2 \\ M_a, C_a \end{bmatrix}$

FIG. 3.6 – *Exemple d'utilisation de la pile des annotations*

Cet accès aux expressions symbolique par la pile d'annotations nécessite d'effectuer statiquement deux vérifications :

- Il faut vérifier que les éléments utilisées pour exprimer d'autres annotations existent dans la pile d'annotations. Par exemple, si le code analysé ne comporte que les trois boucles de notre exemple, l'élément $P[3]$ ne peut être utilisé pour annoter la boucle c car il n'existe pas.
- Il faut aussi vérifier que les expressions compteurs utilisées existent. Dans notre exemple, on ne peut pas utiliser $P[0].C^2$ dans l'annotation de la boucle b car l'annotation $P[0]$ (annotation de la boucle a) ne définit qu'une seule expression compteur.

3.3.3 Le calcul symbolique du nombre d'itérations

Les annotations symboliques décrivent statiquement le comportement des boucles du code analysé. L'annotation symbolique associée à une boucle et la pile des annotations des boucles qui l'englobent permettent de calculer son nombre maximum d'itérations à l'aide de sommes finies et des fonctions utilisées dans les expressions symboliques de l'annotation.

Tout d'abord, et pour simplifier l'exposé, nous utilisons le nommage absolu des boucles. Si on considère la boucle c de l'exemple 3.6 seule, c'est-à-dire en faisant abstraction des boucles l'englobant, son nombre d'itérations est calculé par :

$$\sum_{\lambda=0}^{M_c-1} 1$$

M_c est exprimé en fonction des expressions M et C^i d'une ou plusieurs boucles englobantes. Pour exprimer le nombre d'itérations total de la boucle c par une seule expression, il faut identifier de manière unique les variables λ.

$$\sum_{\lambda_a=0}^{(M_a-1)} \sum_{\lambda_b=0}^{(M_b-1)} \sum_{\lambda_c=0}^{(M_c-1)} 1$$

Dans les systèmes d'équations suivants, on utilise la pile des annotations P au lieu du nommage absolu des expressions. Les expressions M et C^i associées à une boucle sont donc accessibles aux boucles internes par l'intermédiaire de P. Les annotations sont donc empilées (opérateur $\curvearrowleft$) et P est la seule inconnue des équations.

Le nombre d'itérations de la boucle interne de l'exemple de la figure 3.6 est exprimé par les équations suivantes :

$$\begin{aligned}
N_a(P) &= \sum_{\lambda=0}^{(10-1)} N_b(P^\curvearrowleft[10, \lambda]) \\
N_b(P) &= \sum_{\lambda=0}^{((P[0].C)-1)} N_c(P^\curvearrowleft[(P[0].C), \lambda, (P[0].C) \times \lambda]) \\
N_c(P) &= \sum_{\lambda=0}^{((P[0].C^1+P[0].C^2)-1)} 1
\end{aligned}$$

Examinons la fonction $N_b(P)$. La borne supérieure de la sommation est exprimée par l'intermédiaire de la pile d'annotations P, et sa valeur est $(P[0].C) - 1$ soit $\lambda_a - 1$ puisque la variable λ de la boucle a est empilée par l'équation $N_a(P)$. Les expressions symboliques associées à la boucle courante sont empilées sur P et la pile résultante est passée comme paramètre à la fonction $N_c(P)$.

Pour chaque itération de la boucle b, la boucle c est exécutée N_c fois, et pour chaque exécution de ce code (composé des trois boucles) la boucle c est exécutée N_a fois. Pour formuler N_a en une seule équation, il est nécessaire d'identifier de manière unique les variables λ des trois sommes finies.

$$N_a = \sum_{\lambda_a=0}^{(10-1)} \sum_{\lambda_b=0}^{(\lambda_a-1)} \sum_{\lambda_c=0}^{((\lambda_b+(\lambda_b\times\lambda_a))-1)} 1$$

3.4 Intégration dans un analyseur à base d'arbre syntaxique

Nous avons défini une nouvelle technique d'annotations des boucles qui permet la prise en compte précise des boucles non rectangulaires. Il reste maintenant à intégrer cette technique pour que l'analyse statique de WCET bénéficie de cette précision. La technique que nous avons retenue est une technique "tree-based" (*cf.* § 2.2.3) car l'utilisation de l'arbre syntaxique permet, entre autre, une identification aisée des emboitements de boucles. La méthode de calcul du WCET retenue pour estimer le WCET du code à partir de l'arbre syntaxique est celle basée sur les schémas temporels (*timing schema*, *cf.* § 2.2.3).

3.4.1 Schéma temporel intégrant la présence d'annotations symboliques

Les équations de WCET de P. Puschner et C. Koza [PK89] permettent de calculer une estimation du WCET de chacune des structures du langage à analyser. Ce jeu d'équations (rappelé dans le tableau 3.1) peut aisément être adapté pour prendre en compte notre proposition d'annotations symboliques.

	Formules pour le calcul du WCET : $W(S)$
$S = S_1; \ldots; S_n$	$WCET(S) = WCET(S_i) + \ldots + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
$S =$ loop(tst) S_1	$WCET(S) = maxiter \times (WCET(Test) + WCET(S_1)) + WCET(Test)$ où *maxiter* est le nombre maximum d'itérations.
$S =$ bloc de base BB_x	$WCET(S) =$ WCET du bloc de base BB_x

TAB. 3.1 – *Formules de calcul du WCET [PK89] (avec annotations constantes)*

Tout d'abord, la pile des annotations (P) doit être ajoutée comme paramètre à chaque formule. Ensuite, la formule de calcul du WCET nécessitant le plus de modifications est celle exprimant le WCET des structures de boucle du langage. En effet, la multiplication utilisée dans le cas d'annotations constantes doit être ici remplacée par une somme finie, telle que nous l'avons introduite dans le paragraphe 3.3.3.

Pour calculer le WCET d'une boucle,
- on commence par calculer le WCET d'une itération qui dépend de P' (la pile des annotations sur laquelle on a empilé l'annotation de la boucle courante),

$$WCET_iteration(P') = WCET(Test, P') + WCET(Body, P')$$

- puis on calcule $WCET_iteration(P')$, pour $\lambda = 0$ à $M - 1$,
- enfin, on ajoute le WCET de la sortie de boucle, $WCET(Test, P'')$ avec P'' la pile des annotations sur laquelle on a empilé une annotation constituée de l'expression symbolique M de la boucle courante et de ses expressions de compteurs de boucle C^i dans lesquelles on a remplacé la variable λ par l'expression M. Ainsi, $WCET(Test, P'')$ est le WCET du test de la boucle pour l'itération $\lambda = M$.

Par exemple, le WCET d'une boucle annotée par $[10, \lambda + 2]$ est :

$$WCET(boucle, []) = \sum_{\lambda=0}^{(M-1)} WCET_iteration([10, \lambda + 2]) + WCET(Test, [10, 10 + 2])$$

Structure S	
$S_1; \ldots; S_n$	$WCET(S,P) = WCET(S_1,P) + \ldots + WCET(S_n,P)$
if (tst) then S_1 else S_2	$WCET(S,P) = WCET(Test,P) + max(WCET(S_1,P), WCET(S_2,P))$
for(;tst;inc) S_1	$WCET(S,P) = \sum_{\lambda=0}^{M-1} \big(WCET(Test,P') + WCET(S_1,P')\big) + WCET(Test,P'')$ avec $P' = (P^\frown M, C^1, \ldots, C^n)$ avec $P'' = (P^\frown M, C^1[M/\lambda], \ldots, C^n[M/\lambda])$
$S =$ bloc de base BB_x	$WCET(S,P) = WCET_BB_x$

TAB. 3.2 – *Formules de calcul du WCET avec annotations symboliques*

Il existe des relations de dépendances entre les expressions des annotations reflétant les dépendances entre compteurs de boucles pour les boucles emboitées (voir figure 3.7). Une expression symbolique peut dépendre d'une autre expression *directement* ou *indirectement*. Dans notre exemple constitué des boucles a, b et c, l'expression M_c dépend *directement* de C_b^1 et C_b^2 car M_c est exprimé en fonction de $P[0].C^1$ et $P[0].C^2$. De même, M_c dépend *directement* de C_a par l'intermédiaire de C_b^2.

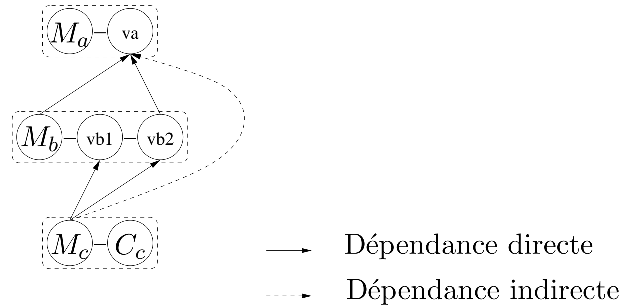


FIG. 3.7 – *Graphe de dépendance entre les annotations symboliques de l'exemple de la figure 3.4*

Si dans l'annotation de la boucle B, aucune des expressions de compteur C_B^* n'est utilisée par une autre annotation, alors il est inutile d'énumérer les valeurs des variables λ_B et C_B^* par une somme finie. Et il n'est donc pas nécessaire d'exprimer le WCET de la boucle B en utilisant une somme finie.

Ainsi, le graphe de dépendance des annotations présenté en figure 3.7 indique que le WCET de la boucle la plus emboitée peut être exprimé sans utiliser de somme finie. L'équation $N_c(P)$

du paragraphe précédent peut être alors simplifiée car l'expression C_c n'est pas utilisée.

$$N_c(P) = \sum_{\lambda_c=0}^{((P[0].C^1+P[0].C^2)-1)} 1$$

est simplifiée en :

$$N_c(P) = (P[0].C^1 + P[0].C^2) \times 1$$

On définit pour ce cas une version simplifiée de la formule du schéma temporel qui concerne les boucles.

$$WCET(S, P) = M \times \left(WCET(Test, P') + WCET(S_1, P')\right) + WCET(Test, P')$$
avec $P' = (P^\frown M)$

Cette nouvelle formulation est plus simple à évaluer. Elle permet d'éviter les énumérations superflues et donc de réduire le temps d'évaluation du système d'équations donnant le WCET.

3.4.2 Exemple d'utilisation des annotations symboliques pour l'analyse statique de WCET

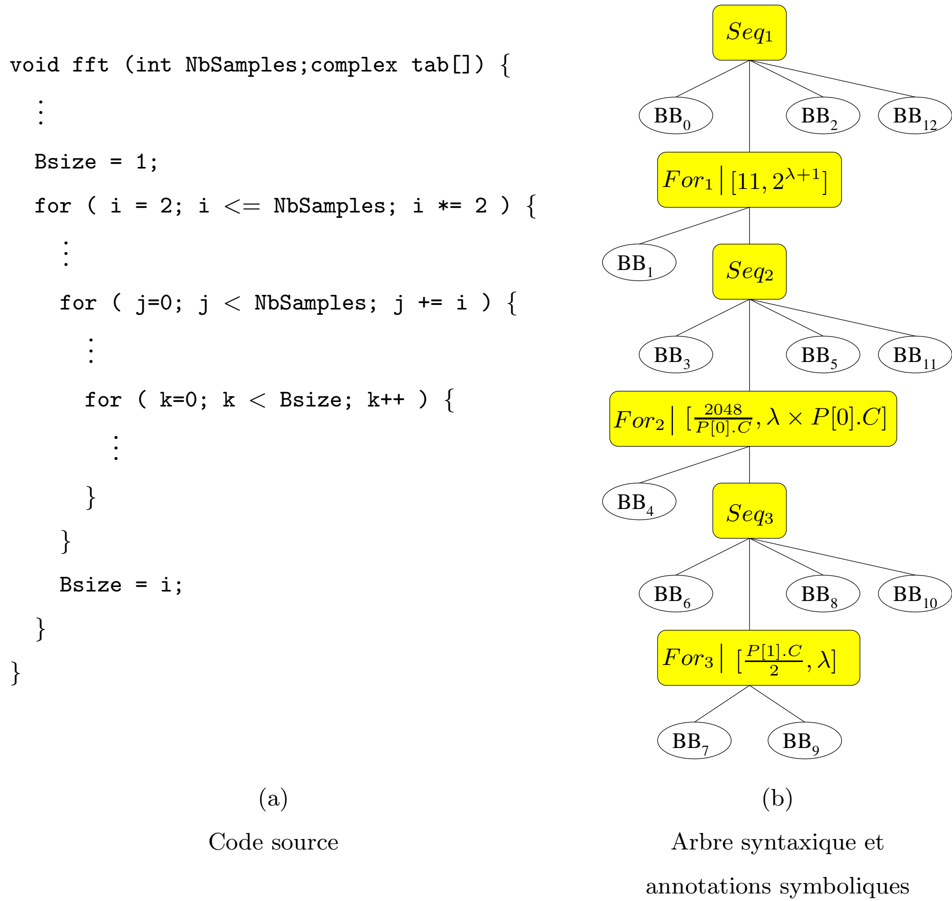
L'exemple suivant illustre l'utilisation de ce nouveau jeu d'équations pour estimer le WCET d'une transformée de fourrier rapide (notée *fft* pour *Fast Fourier Transform*) dont le code est présenté par la figure 3.8.a. Le code de cette *fft* comporte trois boucles emboitées, dont deux boucles non-rectangulaires (les boucles j et k). La figure 3.8.b représente l'arbre syntaxique correspondant. Les nœuds correspondant aux boucles (*i.e* nœuds For_1, For_2 et For_3) sont annotés par des expressions symboliques.

Dans cet exemple, on suppose que l'algorithme de *fft* est appliqué à un tableau de 2048 éléments ($NbSamples = 2048$). Les annotations symboliques utilisées pour décrire le comportement des trois boucles emboitées de la *fft* ainsi que les annotations constantes équivalentes sont résumées dans la table 3.3.

Annotations constantes	Annotations symboliques
11	$[\ 11\ ,\ 2^{\lambda+1}\ ]$
1024	$[\ \frac{2048}{P[0].C}\ ,\ \lambda \times P[0].C\ ]$
1024	$[\ \frac{P[1].C}{2}\ ,\ \lambda\ ]$

TAB. 3.3 – *Annotations constantes et symboliques de fft ($NbSamples = 2048$)*

Un moyen simple d'évaluer (sur cet exemple) le gain en terme de précision de l'estimation du comportement des boucles est de comparer le nombre d'itérations de la boucle la plus interne (For_3) calculé en utilisant les annotations constantes et les annotations symboliques du tableau 3.3.



(a)

Code source

(b)

Arbre syntaxique et

annotations symboliques

FIG. 3.8 – *Code source d'une fft et l'arbre syntaxique correspondant*

$$\begin{array}{lcl} \text{Annotations constantes :} & 11 \times 1024 \times 1024 & = & 11534336 \\ \text{Annotations symboliques :} & \sum_{\lambda=0}^{(11-1)} \left(\left(\frac{2048}{2^{(\lambda+1)}}\right) \times \left(\frac{2^{(\lambda+1)}}{2}\right) \times 1 \right) & = & 11264 \end{array}$$

Dans cet exemple, l'estimation du nombre d'itérations de la boucle la plus interne est divisée par 1000 par l'utilisation d'annotations symboliques.

À partir de l'arbre syntaxique de la figure 3.8.b et des équations de WCET de la table 3.3, on construit un système d'équations qui permet de calculer les WCET de chacun des nœuds de l'arbre, des feuilles (*i.e.* les blocs de base) jusqu'à la racine (*i.e.* le WCET global). Pour cela, on suppose connus les WCET des blocs de base. On note $WCET_BB_x$ le WCET du bloc de base BB_x. Chacune des équations du système suivant correspond à un nœud de l'arbre syntaxique.

$$\begin{aligned}
WCET(Seq_1, P) &= WCET_BB_0 + WCET(For_1, P) + WCET_BB_2 + WCET_BB_{12} && (3.1)\\
WCET(For_1, P) &= \sum_{\lambda=0}^{11-1} \left(WCET_BB_1 + WCET(Seq_2, P^\frown(2^{(\lambda+1)}))\right) + WCET_BB_1 && (3.2)\\
WCET(Seq_2, P) &= WCET_BB_3 + WCET(For_2, P) + WCET_BB_5 + WCET_BB_{11} && (3.3)\\
WCET(For_2, P) &= \sum_{\lambda=0}^{\frac{2048}{P[0].C}-1} \left(WCET_BB_4 + WCET(Seq_3, P^\frown(\lambda \times P[0].C))\right) + WCET_BB_4 && (3.4)\\
WCET(Seq_3, P) &= WCET_BB_6 + WCET(For_3, P) + WCET_BB_8 + WCET_BB_{10} && (3.5)\\
WCET(For_3, P) &= \sum_{\lambda=0}^{\frac{P[1].C}{2}-1} \left(WCET_BB_7 + WCET_BB_9\right) + WCET_BB_7 && (3.6)
\end{aligned}$$

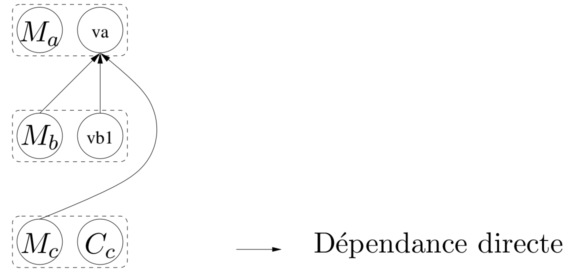


FIG. 3.9 – *Graphe de dépendance entre les annotations symboliques du programme fft*

Le graphe de dépendance des annotations présenté en figure 3.9 indique que seule l'annotation de la boucle For_1 est nécessaire pour exprimer le nombre d'itérations des boucles For_2 et For_3. Donc, seule l'expression du WCET de la boucle For_1 requiert une somme finie, et les équations des WCET des deux autres boucles peuvent être simplifiées comme suit :

$$\begin{aligned}
WCET(For_2, P) &= \frac{2048}{P[0].C} \times \left(WCET_BB_4 + WCET(Seq_3, P^\frown 0)\right) + WCET_BB_4 && (3.4)\\
WCET(For_3, P) &= \frac{P[1].C}{2} \times \left(WCET_BB_7 + WCET_BB_9\right) + WCET(BB_7) && (3.6)
\end{aligned}$$

Le WCET du code de la *fft* est le résultat de l'équation de WCET de la racine de l'arbre syntaxique (*i.e.* Seq_1) avec une pile d'annotation initialement vide comme paramètre.

$$WCET(fft) = WCET(Seq_1, Pile_Vide)$$

3.4.3 Évaluation du schéma temporel par un outil d'évaluation symbolique

Comme on l'a dit précédemment, le système d'équations de WCET résultant est évalué par un outil de calcul symbolique. On peut par exemple utiliser le système de calcul formel Maple [CGG91] (*i.e.* un outil de calcul symbolique).

Les équations sont traduites en code Maple. Pour ce faire :

- à chaque équation du système correspond une fonction Maple.
- les WCET manipulés sont des scalaires,
- les opérateurs + et $\sum$ sont disponibles directement,
- et les opérations de gestion de la pile d'annotations sont réalisées par des fonctions Maple *ad hoc*.

La figure 3.10 présente un exemple de traduction du jeu d'équations présenté au paragraphe 3.3.3. Les cinq premières lignes du "programme" Maple définissent les fonctions de gestion de la pile d'annotations. L'opérateur $P \frown x$ est traduit par l'appel de la fonction `push(P,x)`, et les accès aux éléments de la pile $P[n].M$, $P[n].C^1$, $P[n].C^2$ et $P[n].C$ sont réalisés respectivement par les fonctions `getM(P,n)` , `getC1(P,n)`, `getC2(P,n)`, et `getC(P,n)` (voir code en annexe B).

Code Maple

```
Na := proc(P) RETURN( sum(
  Nb(push(P,[10,ia])) ,
  ia = 0 .. (10-1)
)) end:

Nb := proc(P) RETURN( sum(
  Nc(push(P,[getC(P,0),ib,getC(P,0)*ib])) ,
  ib = 0 .. (getC(P,0)-1)
)) end:

Nc := proc(P) RETURN( sum(
  1 ,
  ic = 0 .. (getC1(P,0)+getC2(P,0)-1)
)) end:
```

(a)

Évaluation

```
#MAPLE> Nc( [ [Mb,Cb1,Cb2] ,[Ma,Ca] ] );
```

$\Rightarrow Cb1 + Cb2$

```
#MAPLE> Nb( [ [Ma,Ca] ] );
```

$\Rightarrow -\frac{1}{2}Ca + \frac{1}{2}(Ca)^3$

```
#MAPLE> Na(pile_vide);
```

$\Rightarrow 990$

(b)

FIG. 3.10 – *Traduction des équations du paragraphe 3.3.3 pour Maple*

Les résultats des évaluations du code Maple de la figure 3.10 nous indiquent que :

- Si l'on calcule la fonction `Nc` avec comme paramètre une pile fictive contenant les annotations des boucles englobantes (*cf.* première commande, figure 3.10.b), le résultat obtenu est le nombre d'itérations de la boucle c pour une itération de la boucle b, soit $C_b^1 + C_b^2$ itérations.

– De même, la deuxième commande de la figure 3.10.b indique que pour chaque itération de la boucle a, la boucle c itère au plus $\frac{1}{2} \times C_a + \frac{1}{2} \times (C_a)^3$ fois.
– Enfin, la dernière commande évalue la fonction `Na` avec une pile vide en paramètre. Le résultat indique que la boucle c itère 990 fois quand le programme est exécuté.

3.5 Récapitulatif

Nous avons présenté dans ce chapitre une nouvelle technique d'annotation des boucles qui permet de représenter précisément et statiquement le comportement dynamique des boucles, même en présence de boucles non-rectangulaires. Cette technique d'annotation est basée sur l'utilisation d'annotations symboliques et sur le calcul du WCET par un outil d'évaluation symbolique. Les annotations symboliques sont constituées de plusieurs expressions symboliques qui décrivent le nombre maximum d'itérations des boucles et l'évolution des compteurs de boucles. L'annotation d'une boucle peut être exprimée en fonction des annotations des boucles englobantes. Le schéma temporel utilisé pour la génération du système d'équations de WCET à été adapté pour utiliser les annotations symboliques. Un des avantages du calcul symbolique du WCET est la possibilité d'exprimer le WCET en utilisant des variables qui restent non-instanciées dans le résultat final.

Chapitre 4

Réduction du pessimisme de l'analyse par la prise en compte d'éléments d'architecture

4.1 Introduction

Comme on l'a vu au paragraphe 2.3, les architectures actuelles comportent certains éléments qui améliorent grandement leurs performances. L'ignorance de ces éléments d'architecture dans le cadre de l'analyse statique de WCET est une cause importante du pessimisme des estimations de WCET. Un moyen de réduire le pessimisme de l'analyse statique de WCET est donc de prendre en compte l'effet de ces éléments d'architecture. Cette prise en compte ne doit pas remettre en cause la sûreté des estimations, on va donc chercher à savoir quand, dans le pire des cas, ces éléments d'architecture sont efficaces (*i.e.* quand on est sûr qu'ils réduisent effectivement le temps d'exécution).

Nous avons choisi d'étudier trois éléments couramment rencontrés dans les architectures actuelles : le cache d'instructions, le mécanisme de prédiction de branchement et le pipeline. Le paragraphe 4.2 présente le moyen que nous utilisons pour différencier les contextes d'analyse. Un contexte d'analyse représente l'état de l'architecture matérielle à un moment particulier de l'exécution, et les résultats d'analyse peuvent varier en fonction de l'état de l'architecture considéré. Nous détaillons ensuite les méthodes de prise en compte *bas niveau* du cache d'instructions (§ 4.3), de la prédiction de branchement (§ 4.4) et du pipeline (§ 4.5).

4.2 Niveaux d'emboitement de blocs

Nous justifions ici l'intérêt de la prise en compte de deux types de niveaux d'emboitement de blocs pour l'analyse statique de WCET : les niveaux d'emboitement de boucles, et les niveaux d'emboitement des blocs conditionnels. Comme on le verra plus en détails dans les paragraphes 4.3, 4.4 et 4.5, cet intérêt est fortement lié à la prise en compte de l'effet de certains éléments architecturaux sur le temps d'exécution des programmes.

4.2.1 Intérêt à considérer le niveau d'emboitement des boucles

La définition des niveaux d'emboitement des boucles est d'une importance capitale pour l'obtention d'estimations de WCET précises. En effet, le comportement d'une instruction d'un bloc de base peut dépendre de son contexte d'exécution (*i.e.* des instructions qui la précèdent dans un chemin d'exécution). Ce contexte est principalement défini par la boucle englobante considérée.

Prenons l'exemple du cache d'instructions. Le temps d'exécution d'une instruction dépend, entre autres, de sa présence dans le cache (voir § 4.3 pour plus de détails). L'exemple de la figure 4.1 représente le cas où deux instructions, a et b, sont en compétition pour l'occupation de la même ligne dans le cache d'instructions. Chaque fois que l'instruction a est exécutée, elle remplace l'instruction b dans le cache et vice versa.

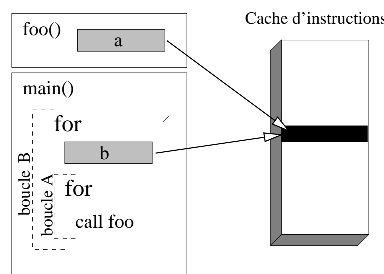


FIG. 4.1 – *Niveau d'emboitement des boucles et conflit d'accès au cache*

Si on ne prend pas en compte l'emboitement des boucles, et pour être certain de ne pas être optimiste, on considérera qu'aucune des deux instructions n'est dans le cache au moment de son exécution, à cause du conflit, ce qui induit un pessimisme excessif.

Or, la boucle A (qui contient a) est emboitée dans la boucle B. Ainsi, à chaque itération de la boucle B (et donc à chaque exécution de b) l'instruction a peut être exécutée M_A fois (M_a est le nombre maximum d'itérations). Son absence dans le cache d'instructions n'est donc pas aussi fréquente que supposé précédemment, car après chaque exécution de b, la première exécution de a (*i.e.* première itération de A) a pour effet le chargement de a dans le cache, et l'instruction a sera dans le cache pour les $M_A - 1$ exécutions suivantes. Dans cet exemple, si M_A et M_B sont les nombres d'itérations des boucles A et B, et s'il n'y pas

d'autres instructions en conflit avec a et b, on peut estimer de façon certaine que a sera absent du cache M_B fois et présent dans le cache $M_B \times (M_A - 1)$ fois.

Cet exemple montre l'utilité des niveaux d'emboitement de boucles. En effet, l'analyse statique de WCET doit prendre en compte le fait que l'instruction a n'est absente du cache que pour les itérations de la boucle B.

Les niveaux d'emboitement des boucles (appelés *ln-levels* pour *loop nesting levels*) permettent de connaître toutes les boucles englobant un bloc de base et de discerner autant de comportements potentiellement différents des instructions le composant.

Les techniques d'analyse de WCET bas niveau peuvent ainsi être affinées pour prendre en compte les différents niveaux d'emboitement de boucles, et fournir des résultats potentiellement différents selon le contexte considéré.

4.2.2 Système de nommage des boucles emboitées

Nous définissons ici un système de nommage hiérarchique des boucles emboitées, les *ln-levels*. Le *ln-level* d'une boucle indique sa position par rapport au niveau zéro d'emboitement et par rapport aux autres boucles. Ce système de nommage est basé sur la notion d'arbre syntaxique introduite au paragraphe 2.1.2.3.

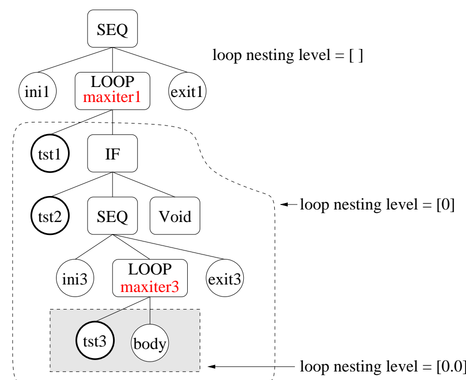


FIG. 4.2 – *Arbre syntaxique et niveaux d'emboitement de boucles (ln-levels).*

Aux boucles les plus externes (*i.e.* qui ne sont englobées par aucune autre boucle) on associe les *ln-levels* [0], [1], etc. Les boucles immédiatement englobées par la boucle [0] sont les boucles [0.0], [0.1], et ainsi de suite. Le niveau zéro d'emboitement est représenté par une boucle fictive, dont le *ln-level* est [ ], qui englobe entièrement l'arbre syntaxique. Les niveaux d'emboitements sont donc bornés par [ ] à la racine de l'arbre syntaxique. Par exemple, la boucle dont le *ln-level* est [2.3.1] (appelée boucle [2.3.1]) est emboitée dans les boucles [2.3], [2] et [ ], et elle englobe la boucle [2.3.1.0].

De plus, on définit un niveau d'emboitement fictif : [*never*]. Ce niveau d'emboitement est celui d'une boucle qui n'est jamais exécutée et permet de représenter, par exemple, un comportement d'instruction qui ne se produit à *aucun* des *ln-levels* de l'arbre.

Sur l'exemple de la figure 4.2, à chaque boucle est associé un *ln-level*, et tous les sous-arbres fils du nœud de type boucle sont contenus dans ce *ln-level*. Dans la suite de ce document, une boucle dont le *ln-level* est $[x]$ est appelée boucle $[x]$.

La boucle la plus emboitée qui contient le bloc de base BB_α est appelée boucle $[\perp_\alpha]$ de ce bloc de base (son *ln-level* est $[\perp_\alpha]$). Ainsi, pour le bloc de base BB_β contenu par les boucles [2.3], [2] et [ ]) mais par aucune boucle $[2.3.x]$ $(\forall x)$, on a $\perp_\beta = 2.3$. Sur l'exemple de la figure 4.2, le *ln-level* $[\perp]$ du bloc de base noté $ini3$ est [0].

Sur la base de ce système de nommage des boucles, on définit une relation d'ordre partiel et une fonction de recherche du plus proche "parent" de deux *ln-levels*.

- L'ordre partiel $x \succeq y$ sur les *ln-levels* indique si la boucle x est emboitée dans la boucle y. Il est défini par :

$$\begin{aligned} x &\succeq y \text{ si } x \text{ est un préfixe gauche de } y \\ [never] &\succ [\,] \\ [\,] &\succ x \end{aligned}$$

  La relation $\succ$ est définie par : $x \succ y \Longleftrightarrow ((x \succeq y) \wedge (x \neq y))$. Ainsi, $x \succ y$ implique que la boucle x englobe la boucle y.

- La fonction Ψ indique la boucle la plus emboitée englobant deux autres boucles (*e.g.* $\Psi([2.3.1],[2.3.4]) = [2.3]$)).

$$q = \Psi(x,y) \Longleftrightarrow (q \succeq x \wedge q \succeq y) \wedge \big(\nexists\, p \mid (q \succ p) \wedge (p \succeq x) \wedge (p \succeq y)\big)$$

4.2.3 Transposition aux blocs conditionnels

Le concept de niveau d'emboitement de bloc conditionnel (ou *cn-level* pour *conditionnal nesting level*) est similaire à celui des *ln-levels* du paragraphe précédent. Ils sont associés non pas aux boucles, mais aux branches conditionnelles (*cf.* figure 4.3). Chacune des deux branches d'une structure conditionnelle (*i.e.* *then* et *else*) se voit attribuer un *cn-level*. Le système de nommage est le même que pour les *ln-levels*, et l'ordre partiel $\succeq$ ainsi que la fonction Ψ définis pour les *ln-levels* s'appliquent aux *cn-levels*.

On définit l'opérateur $\overset{\neg\text{if}}{\rightsquigarrow}$ opérant sur deux blocs de base comme suit : par définition $BB_\alpha \overset{\neg\text{if}}{\rightsquigarrow} BB_\beta$ est vrai quand pour tous les chemins d'exécution possibles T tel que $BB_\alpha \in T$ on a $BB_\beta \in T$. Soit $[\perp_\alpha]$ le niveau d'emboitement conditionnel minimum (au sens de $\succeq$) de BB_α et $[\perp_\beta]$ celui de BB_β, la relation $\overset{\neg\text{if}}{\rightsquigarrow}$ est alors exprimée en utilisant Ψ :

$$BB_\alpha \overset{\neg\text{if}}{\rightsquigarrow} BB_\beta \;\Leftrightarrow\; \Psi([\perp_\alpha],[\perp_\beta]) = [\perp_\beta]$$

L'utilité des *cn-levels* est illustrée par le cas des conflits dans le cache d'instructions (*cf.* figure 4.4). Dans cet exemple, les blocs de base BB_α, BB_β et BB_γ sont en compétition

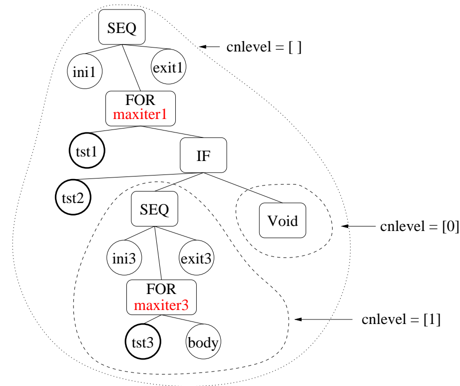
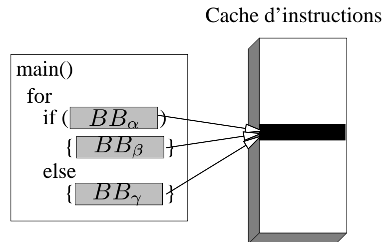


FIG. 4.3 – *Arbre syntaxique et niveaux d'emboitement de blocs conditionnels (cn-levels)*



$[\perp_\alpha] = [0]$, $[\perp_\beta] = [0.0]$ et $[\perp_\gamma] = [0.1]$

$\Psi([\perp_\gamma], [\perp_\beta]) \neq [\perp_\beta]$

$\Psi([\perp_\gamma], [\perp_\alpha]) = [\perp_\alpha]$

FIG. 4.4 – *Niveau d'emboitement de blocs conditionnels et conflit d'accès au cache*

pour la même ligne du cache d'instructions. La relation $BB_\gamma \overset{\neg\text{if}}{\rightsquigarrow} BB_\beta$ n'étant pas vérifiée, le remplacement du bloc BB_γ par le bloc BB_β n'est que *potentiel*. En revanche, la relation $BB_\gamma \overset{\neg\text{if}}{\rightsquigarrow} BB_\alpha$ est vérifiée, on est donc certain que le bloc BB_α remplacera le bloc BB_γ dans le cache d'instructions à chaque itération de la boucle.

4.3 Prise en compte du cache d'instructions

La prise en compte de l'effet du cache d'instructions dans une analyse statique de WCET a été introduite par F. Mueller en 1994 [AMWH94]. La technique proposée nous permet de savoir si, dans le pire des cas, une instruction sera dans le cache d'instructions au moment de son exécution. Avec cette technique, une instruction supposée présente *sera* dans le cache lors de son exécution, une instruction supposée absente sera *peut-être* absente du cache lors de son exécution. Cette estimation de la présence des instructions dans le cache d'instructions est donc sûre, dans la mesure où si l'efficacité de l'élément d'architecture ne peut être assurée, on retourne le résultat pessimiste (*i.e.* sans prise en compte de l'élément améliorant les performances).

Cette technique est appelée *simulation statique de cache*, terme introduit par F. Mueller (*static cache simulation*) et qui désigne le fait d'envisager statiquement tous les chemins d'exécution possibles du graphe de flot de contrôle pour calculer une représentation de tous les contenus possibles du cache à différents moments de l'exécution. Cette technique ne se base pas sur une simulation d'exécution le long d'un chemin d'exécution particulier mais les considère tous en même temps. Elle se décompose en deux étapes.

La première consiste à calculer des représentations du contenu du cache : les états abstraits de cache. Ces représentations permettent de connaître à tout moment quels sont les contenus *possibles* du cache d'instructions. Le calcul de ces représentations est possible car l'adresse et la taille de chacune des instructions du programme analysé sont connues statiquement, et on peut savoir *a priori* quelles lignes du cache d'instructions elles occuperont, et donc calculer statiquement l'utilisation du cache d'instructions.

La deuxième étape utilise les représentations des contenus possibles du cache pour effectuer un classement des instructions en fonction de l'estimation de leur comportement au pire cas vis-à-vis du cache d'instruction. Ainsi, en fonction de la catégorie dans laquelle se trouve une instruction on est capable d'estimer qu'elle causera un *hit*, ou bien qu'elle causera *peut-être* un *miss* dans le cache lors de son exécution.

La proposition que nous présentons dans les paragraphes suivants est une adaptation de la *simulation statique de cache* de F. Mueller. Les principales améliorations apportées à la technique originale sont l'utilisation des contextes d'analyse présentés au paragraphe 4.2 pour calculer et exprimer les estimations de comportement des instructions vis-à-vis du cache, et la décomposition des blocs de base en bloc d'instructions pour l'estimation du comportement pire cas des instructions.

4.3.1 Types de cache d'instructions pris en compte

Pour répondre au problème de l'écart toujours croissant entre les temps de cycle des processeurs et le temps d'accès de la mémoire, la majorité des processeurs actuels comporte des caches d'instructions et de données. Ces caches offrent des temps d'accès bien plus faibles que la mémoire centrale, mais ont une capacité limitée. Nous présentons ici les différents types

de cache d'instructions que cette méthode peut prendre en compte, ainsi que quelques notions et notations nécessaires à l'exposé de la méthode.

4.3.1.1 Principe et types de caches d'instructions

On suppose ici que les caches d'instructions et de données sont deux entités séparées. Il n'y a donc aucune interférence des accès aux données sur le contenu du cache d'instructions (les caches séparés sont courants dans les architectures actuelles).

Quand une instruction est exécutée sur une architecture comprenant un cache d'instructions, son temps d'exécution dépend de l'état du cache. On distingue deux cas :

- cas *hit*, si l'instruction est dans le cache avant son exécution,
- cas *miss*, si elle est absente du cache lors de son exécution. Elle doit alors être chargée depuis le niveau supérieur de la hiérarchie mémoire, et son préchargement nécessite donc plus de temps.

Le cache d'instructions est constitué de L lignes de cache, chacune de ces lignes peut contenir des instructions en provenance du (ou des) niveau(x) supérieur(s) de la hiérarchie mémoire. Le bloc d'instructions pouvant être contenu dans une *ligne de cache* est appelé *ligne de programme.*

La correspondance entre une ligne de programme et une ligne de cache dépend de l'organisation du cache : à correspondance directe, totalement associatif, ou associatif par ensembles.

Prenons l'exemple d'une ligne de programme débutant à l'adresse *addr*. Dans un cache à *correspondance directe* (*cf.* figure 4.5.a), cette ligne de programme ne peut être contenue que par une ligne de cache dont le numéro est (*addr* modulo L) (sur l'exemple $13 \bmod 8 = 5$). Dans un cache *associatif*, toutes les lignes du cache peuvent contenir la ligne de programme (figure 4.5.b). Enfin, le cache *associatif par ensembles à V voies* (figure 4.5.c) est un compromis entre les deux organisations de caches précédentes. À une ligne de programme correspond un ensemble de V lignes de caches. Le cache est donc découpé en L/V ensembles de V lignes, et une ligne de programme peut être contenue par les V lignes de l'ensemble numéro (*addr* modulo (L/V)). Sur l'exemple, la ligne de programme 13 correspond à l'ensemble de lignes de cache numéro 1 ($13 \bmod (8/2) = 1$).

En fixant l'associativité V du cache à 1 (respectivement L), on se ramène à un cache à correspondance directe (respectivement totalement associatif). Ainsi, pour être la plus générale possible, notre proposition de simulation statique de cache portera sur un cache associatif par ensembles de V voies.

Le plus souvent, lorsqu'une ligne de programme est chargée dans le cache, elle va y remplacer une autre ligne de programme précédemment chargée. Le choix de cette ligne parmi les V lignes de l'ensemble est dictée par la *politique de remplacement* du cache. La plus courante, est celle que nous considérons dans la suite de ce document, est le remplacement de la ligne la plus anciennement référencée (LRU - least-recently-used).

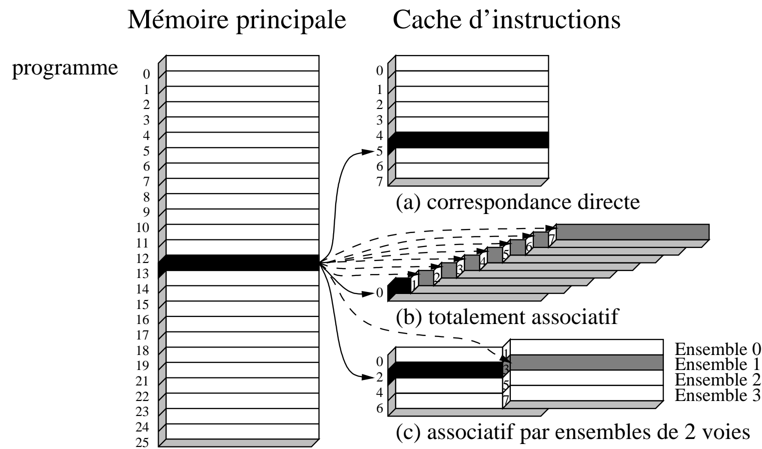


FIG. 4.5 – *Trois types d'organisations du cache*

4.3.1.2 Décomposition des blocs de base en iblocs

Selon le jeu d'instructions de l'architecture considérée, les instructions peuvent être de taille fixe comme sur les architectures RISC (e.g. ARM, UltraSPARC) ou variable comme les instructions des architectures CISC (e.g. Intel x86, 68K). Dans le cas d'instructions de taille fixe, la taille des lignes du cache d'instructions peut différer de celle des instructions. Et cette différence de taille est systématique dans le cas des instructions de taille variable. Nous nous plaçons donc dans le cadre le plus général, dans lequel une ligne de cache peut contenir une ou plusieurs instructions et/ou fragments d'instructions.

Nous définissons donc la notion d'*ibloc* (ou bloc d'instructions) comme une représentation intermédiaire des éléments manipulés par le cache d'instructions. Un ibloc correspond à un contenu d'une ligne de cache en ne considérant qu'un seul bloc de base. Si un ligne de cache contient des instructions (ou fragments d'instruction) appartenant à deux blocs de base différents, on a alors deux iblocs.

La différence entre la notion de ligne de programme définie au paragraphe précédent et la notion d'ibloc est que toutes les instructions et fragments d'instructions d'un ibloc proviennent d'un seul et même bloc de base. La figure 4.6 présente le cas d'une ligne de cache partagée par deux blocs de base contigus. Les instructions pouvant être contenues dans cette ligne de cache sont distribuées entre deux iblocs. Ces iblocs sont identifiés par le nom de leur bloc de base d'origine (α et β dans l'exemple) et un numéro qui nous permet de connaître l'adresse des instructions de l'ibloc (8 dans l'exemple). L'ibloc $IB_{\alpha,8}$ ne contient que les instructions correspondant à la ligne 8 du cache et appartenant à BB_{α}, c'est-à-dire les instructions allant de l'adresse 64 ($8 \times 8 = 64 = 40h$) à 71. De même, $IB_{\beta,8}$ ne contient que les instructions correspondant à la ligne 8 du cache et appartenant à BB_{β}. Ces deux iblocs ne sont que partiellement occupés, en effet les instructions de BB_{α} ne vont que jusqu'à l'adresse 67 ($43h$)

et celle de BB_β commence à l'adresse 68.

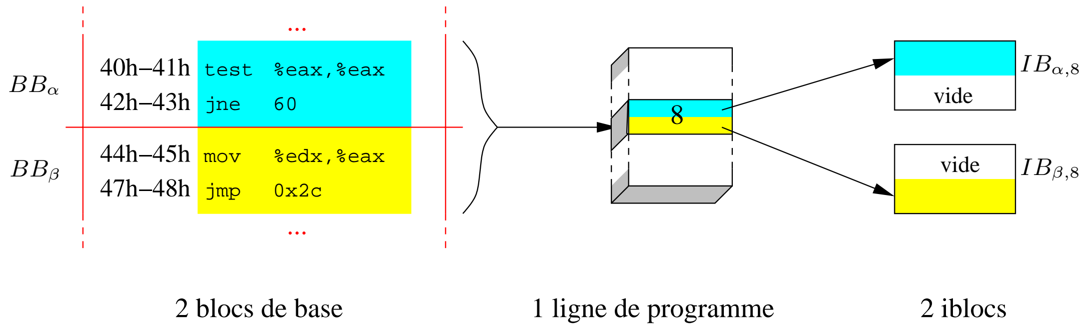


FIG. 4.6 – *Occupation d'une ligne de cache par deux blocs de base*

La figure 4.7 illustre la décomposition d'un bloc de base en plusieurs iblocs, pour un cache d'instructions dont la taille des lignes est 8 octets.

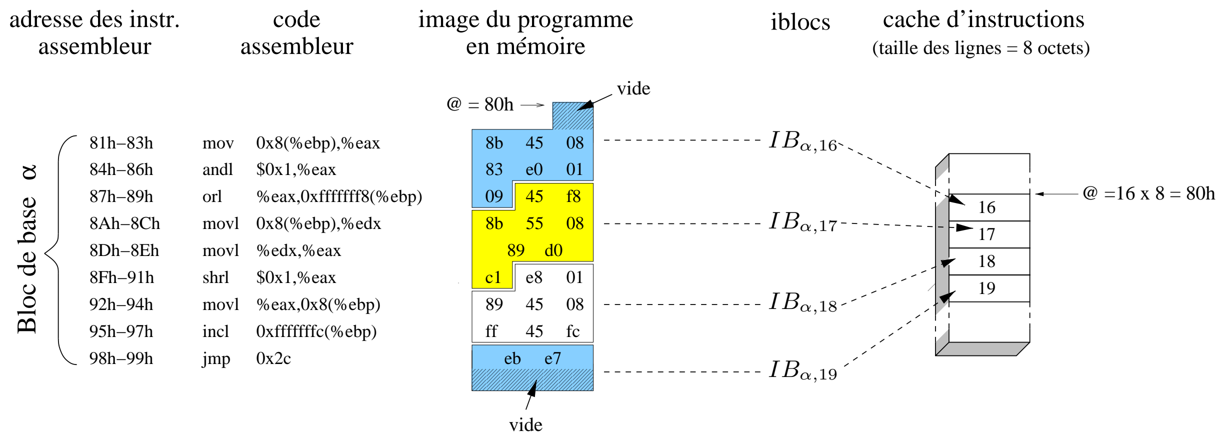


FIG. 4.7 – *Décomposition d'un bloc de base en iblocs*

Pour traiter les problèmes de correspondance entre tailles d'instruction et tailles de ligne de cache, nous avons choisi d'associer les informations de présence/absence, habituellement associées aux instructions, aux iblocs. On peut *in fine* se rapporter aux instructions par application de la règle suivante :

L'exécution d'une instruction cause un *miss* dans le cache si l'un des iblocs la contenant n'est pas dans le cache et qu'elle est en première position dans au moins un de ces iblocs absents du cache. Sinon c'est un *hit*.

Sur l'exemple de la figure 4.7, la première instruction (81h-83h) sera un *miss* si l'ibloc $IB_{\alpha,16}$ est absent du cache d'instructions au moment de son exécution. En effet, bien que cette instruction n'occupe pas le début de l'ibloc, elle est bien en première position dans celui-ci.

Autre exemple, l'instruction 87h-89h, qui est à cheval entre les iblocs $IB_{\alpha,16}$ et $IB_{\alpha,17}$, causera un *miss* uniquement si l'ibloc $IB_{\alpha,17}$ est absent du cache. Ceci est dû au fait qu'au moment où l'instruction 87h-89h est exécutée, la présence de l'ibloc $IB_{\alpha,16}$ dans le cache est assurée car il est chargé lors de l'exécution de l'instruction précédente du bloc de base BB_{α}.

4.3.2 Les états abstraits de cache d'instructions

Nous allons maintenant présenter les états abstraits de cache d'instructions et leur méthode de calcul. Nous présentons d'abord la notion d'états abstraits comme représentations de tous les contenus possibles du cache à différents instants. Puis, la méthode de calcul d'un état abstrait en fonction des autres états abstraits du graphe de flot de contrôle est présentée. Enfin, nous décrivons la méthode itérative de calcul (et recalcul) de l'ensemble des états abstraits jusqu'à stabilisation, ainsi qu'une optimisation du nombre d'itérations nécessaires.

4.3.2.1 Description des états abstraits

Les états abstraits de cache (notés ACS pour *Abstract Cache State*) représentent, pour chaque bloc de base, tous les contenus possibles du cache avant et après leur exécution. Ces contenus possibles sont les ensembles d'iblocs pouvant éventuellement être dans le cache. À chaque bloc de base on associe deux ACS :

- l'$\text{ACS}_{\alpha}^{pré}$ qui indique quels iblocs sont potentiellement dans le cache *avant* l'exécution du bloc de base BB_{α},
- l'$\text{ACS}_{\alpha}^{post}$ qui représente les états possibles du cache *après* exécution de tous les iblocs composant le bloc de base BB_{α}.

Les ACS ont la même structure que le cache d'instructions à simuler, c'est-à-dire $(L/V)\times V$ ensembles d'iblocs, et on les représente par des tableaux d'ensembles d'iblocs à deux dimensions.

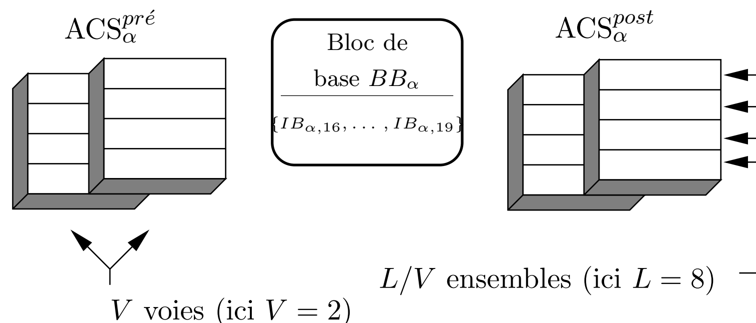


FIG. 4.8 – *Structure des états abstraits de caches (ACS)*

4.3.2.2 Calcul des états abstraits

On s'intéresse ici à la simulation d'un cache associatif par ensemble à V voies. Ses états abstraits sont représentés par des tableaux à deux dimensions ("ensembles" et "voies") (voir figure 4.8). Les ensembles d'iblocs contenus dans ces tableaux sont les ensembles ACS$[e, v]$ avec $0 \leq v < V$ et $0 \leq e < L/V$. Comme présenté au paragraphe précédent, les ACS sont divisés en deux catégories, *pré* et *post*. Nous détaillons maintenant le calcul des valeurs de ces deux types d'ACS.

Calcul des ACS^{pré}

L'ACS$_\alpha^{pré}$ doit représenter tous les contenus possibles du cache d'instructions avant l'exécution de BB_α. Puisque le pire chemin d'exécution aboutissant à BB_α est inconnu, on doit prendre en compte toutes les possibilités et donc toutes les séquences de blocs de base pouvant conduire à BB_α. Pour ce faire, ACS$_\alpha^{pré}$ est calculé en réalisant *l'union* des ACS^{post} de tous les blocs de base précédant BB_α.

Notons que les ACS^{pré} ne sont pas obtenus par une *simulation* du contenu du cache selon le chemin d'exécution emprunté, mais sont calculés en considérant l'ensemble des chemins différents arrivant à un bloc de base.

- Soit $Pred(BB_\alpha)$ l'ensemble des blocs de base précédant immédiatement le bloc de base BB_α dans le graphe de flot de contrôle.
- Soit $\uplus$ l'opérateur d'union des ACS défini par :

$$\text{ACS}_\gamma = \text{ACS}_\alpha \uplus \text{ACS}_\beta \iff \forall e \forall v \ \text{ACS}_\gamma[e, v] = \text{ACS}_\alpha[e, v] \cup \text{ACS}_\beta[e, v]$$

(avec $\cup$ l'union ensembliste).

L'ACS$_\alpha^{pré}$ est calculé selon l'équation suivante :

$$\text{ACS}_\alpha^{pré} = \biguplus_{BB_\beta \in Pred(BB_\alpha)} \text{ACS}_\beta^{post} \tag{4.1}$$

On ajoute un bloc de base fictif au début du graphe de flot de contrôle, dont le rôle est de représenter l´état initial du cache. Ce bloc de base va occuper toutes les lignes de l'ACS avec des iblocs notés *vide*. Ces iblocs permettent de savoir si la présence d'un autre ibloc est certaine, ou bien n'est que potentielle. Par exemple, l'ensemble $\{IB_{\alpha,x}, vide\}$ représentant les contenus possibles d'une ligne de cache indique que la ligne contient soit l'ibloc $IB_{\alpha,x}$, soit rien.

La figure 4.9 illustre le calcul d'un ACS^{pré} à partir des ACS^{post} des deux blocs de base le précédant. Pour cet exemple, les caractéristiques du cache d'instructions simulé sont : $L = 8$, $V = 2$.

Calcul des ACS^{post}

Pour un bloc de base BB_α donné, l'ACS$_\alpha^{post}$ correspondant représente l'état du cache après le chargement des iblocs composant le bloc de base, et donc une modification de l'ACS$_\alpha^{pré}$.

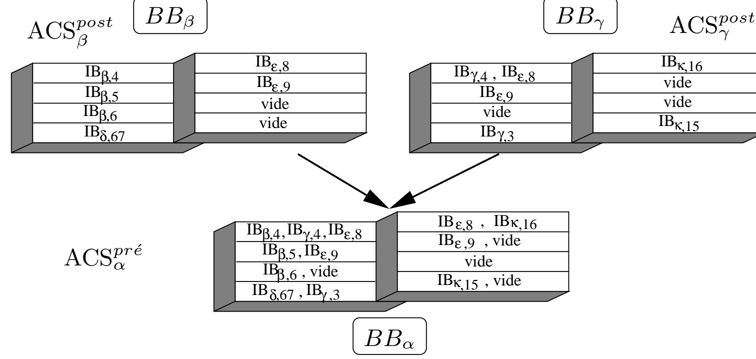


Fig. 4.9 – *Calcul des $ACS^{pré}$*

Les équations (4.2) gèrent l'ajout d'un ibloc de $IB_{\alpha,x}$ dans un ACS^{post}. C'est un codage sous forme d'équations du mécanisme de remplacement LRU d'un cache associatif par ensemble de V voies.

Le calcul de l'ACS$_\alpha^{post}$ s'effectue donc en deux temps : (i) copie de l'ACS$_\alpha^{pré}$ dans l'ACS$_\alpha^{post}$, puis (ii) ajout un à un et par ordre croissant de tous les iblocs $IB_{\alpha,i}$ composant le bloc de base BB_α par application des équations (4.2).

$$
\begin{aligned}
\mathrm{ACS}_\alpha^{post}[e,v]' &= \begin{cases} \text{si } \Delta_{e,v} = \emptyset \text{ alors } \mathrm{ACS}_\alpha^{post}[e,v] \\ \Delta_{e,v} \text{ sinon} \end{cases} \\
\Delta_{e,v} &= \begin{cases} \text{si } \big(\ (v = 0) \wedge (e = i\ modulo\ (L/V))\ \big) alors IB_{\alpha,i} \\ \text{sinon } \mathit{diff}_{e,v} \end{cases} \\
\mathit{diff}_{e,v} &= \{IB_{x,y} \mid IB_{x,y} \in \mathrm{ACS}_\alpha^{post}[e, v-1] \wedge IB_{*,y} \notin \Delta_{e,v-1}\}
\end{aligned}
\tag{4.2}
$$

L'ensemble ACS$_\alpha^{post}$[e,v]' est le résultat de l'introduction des iblocs de l'ensemble $\Delta_{e,v}$ dans ACS$_\alpha^{post}$[e,v]. L'ensemble $\Delta_{e,v}$ est l'ensemble des iblocs à introduire dans la voie v de l'ensemble e de l'ACS^{post}. Enfin, la fonction $\mathit{diff}_{e,v}$ calcule l'ensemble des iblocs en conflit avec les iblocs introduits dans la voie précédente $(v-1)$.

Si certains iblocs entrent en conflit avec des iblocs déjà présents dans ACS^{post}[e,v], ces derniers sont supprimés de la voie v et forment l'ensemble $\Delta_{e,v+1}$ des iblocs à introduire dans la voie suivante $(v+1)$. Quand $v+1$ est supérieur à V, cela revient à supprimer les iblocs du cache.

La figure 4.10 illustre ce calcul avec un bloc de base composé de deux iblocs ($IB_{\alpha,16}$ et $IB_{\alpha,17}$). Le détail du calcul y est également proposé.

Sur cet exemple, on voit que le chargement de l'ibloc $IB_{\alpha,16}$ dans le premier ensemble de l'ACS^{pré} a pour effet la disparition de $IB_{\iota,16}$ qui était potentiellement dans le cache. En effet, cet ibloc est en conflit avec $\Delta_{0,1}$, on le retrouve donc dans $\Delta_{0,2}$ qui n'est jamais introduit car

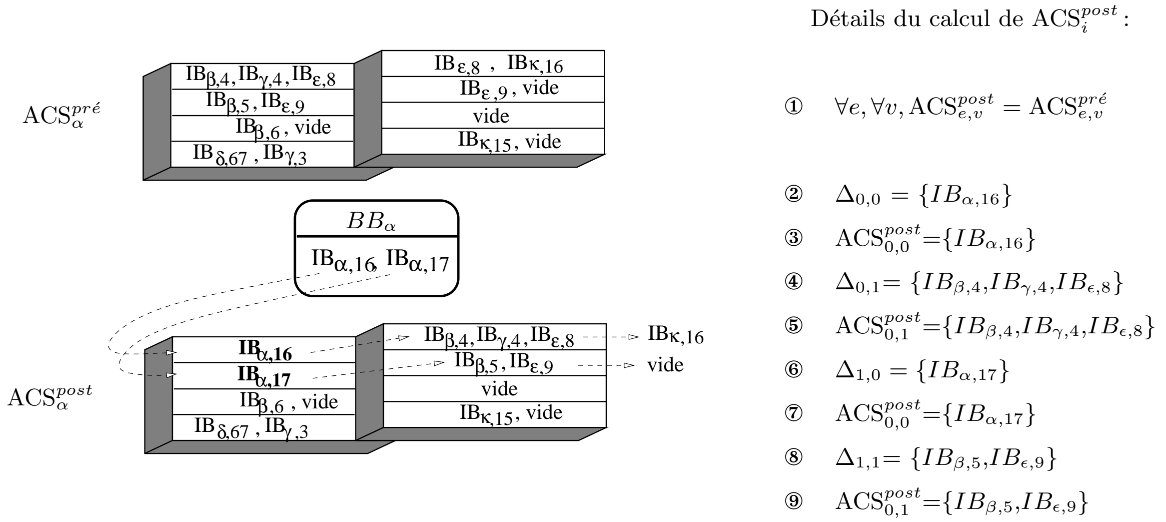


FIG. 4.10 – *Calcul des ACS^{post}*

l'ACS n'a que deux voies.

Le calcul des ACS^{post} présenté ici repose sur l'hypothèse d'une politique de remplacement LRU (remplacement de la ligne la plus anciennement utilisée). La technique de simulation statique de cache d'instructions peut être adaptée à d'autres politiques de remplacement en modifiant le calcul des ACS^{post}.

Un autre type d'état abstrait de cache

En plus des ACS, on calcule un autre type d'état abstrait : les LCS (pour *linear cache states*). Les LCS sont presque identiques aux ACS aussi bien en terme de structure que de méthode de calcul. La seule différence réside dans l'ensemble des blocs de base utilisé pour le calcul de $\mathrm{LCS}_\alpha^{pré}$ (*i.e.* $Pred'(BB_\alpha)$). La différence entre $Pred(BB_\alpha)$ et $Pred'(BB_\alpha)$ est que ce dernier ne contient pas de dépendance avant (voir figure 4.11).

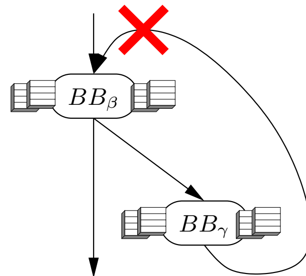


FIG. 4.11 – *Restriction de l'ensemble Pred pour le calcul des LCS*

Considérons le cas de deux blocs de bases contigüs, BB_α et BB_β, qui occupent chacun

la moitié de la ligne de cache x (*i.e.* la frontière entre ces deux blocs de base est au milieu de la ligne de cache). En terme d'iblocs, le partage de la ligne de cache x par deux blocs de base se traduit par l'existence des iblocs $IB_{\alpha,x}$ et $IB_{\beta,x}$. Si BB_{α} est exécuté avant BB_{β}, alors l'ibloc $IB_{\beta,x}$ est chargé dans le cache d'instructions avant son exécution sans jamais avoir été exécuté.

Le but de LCS est de permettre de savoir si un ibloc peut être chargé avant son exécution même si on ne considère pas les boucles (voir page 80 pour les détails de l'utilisation des LCS). Exception faite de cette différence, la gestion des LCS est identique à celle des ACS et ne sera pas détaillée plus avant dans la suite de ce document.

4.3.2.3 Résolution du système d'équation par itération de point fixe

La méthode de prise en compte du cache d'instructions calcule les ACS associés aux blocs de base à partir des équations (4.1) et (4.2).

Ces équations calculent les ACS d'un nœud en fonction des ACS des nœuds précédents dans le graphe de flot de contrôle. Les relations de précédence entre nœuds du graphe de flot de contrôle expriment donc aussi des relations de dépendance entre les ACS des nœuds.

Les boucles introduisent des dépendances «arrière» dans le graphe. Le problème du calcul des ACS est donc celui de la résolution d'un système récursif d'équations. Un tel système peut être résolu par itération de point fixe. De plus, nous avons montré que les conditions nécessaires à l'existence d'une solution (point fixe) sont vérifiées (voir application du théorème de Tarski dans [Col98]).

```
1    input_state(top) := all invalid lines
2    WHILE any change do
3         FOR each basic block instance B DO
4              input_state(B) := NULL
5              FOR each immed pred P of B DO
6                   input_state(B) += output_state(P)
7              output_state(B) := (input_state(B) + prog_lines(B))
8                                   - conf_lines(B)
```

FIG. 4.12 – *Algorithme de calcul des ACS de F. Mueller [AMWH94]*

Un parcours du graphe où chaque nœud n'est examiné qu'une seule fois n'est pas suffisant en raison de la présence possible de cycles dans le graphe et donc dans les dépendances entre

ACS. Sur l'exemple de la figure 4.13, le calcul des ACS pour un graphe de flot de contrôle représentant une boucle *while* fait apparaître le cycle de dépendances suivant : $ACS_\beta^{pré} \rightarrow ACS_\gamma^{post} \rightarrow ACS_\gamma^{pré} \rightarrow ACS_\beta^{post} \rightarrow ACS_\beta^{pré}$.

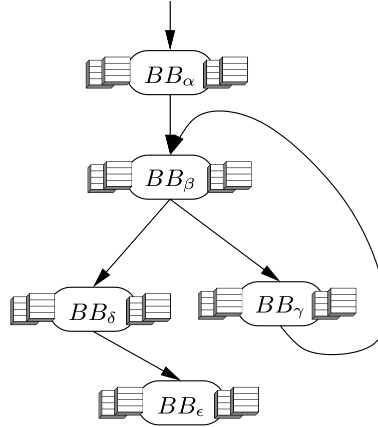


FIG. 4.13 – *Calcul des ACS des blocs de base d'une boucle while*

L'algorithme de recherche de point fixe tel que décrit par F. Mueller dans [MWH94] ne précise pas l'ordre de parcours des nœuds du graphe (*cf.* figure 4.12, ligne 3). Les ACS sont calculés selon un ordre ne faisant apparaître les nœuds du graphe qu'une fois par itération. La question de l'ordre d´évaluation des ACS, laissée ouverte par cet algorithme générique, est abordée au paragraphe suivant.

4.3.2.4 Optimisation du calcul de point fixe

Comme nous l'avons montré au paragraphe précédent, à chaque itération de l'algorithme de recherche du point fixe, toutes les variables du système (les ACS) sont recalculées. Le nombre d'itérations nécessaires dépend de l'ordre dans lequel les ACS sont calculés.

Nous proposons ici un ordre de parcours des nœuds du graphe de flot de contrôle permettant de minimiser le nombre de calculs d'ACS. Cet ordre s'appuie sur la syntaxe du langage, et est défini pour les trois constructions syntaxiques suivantes : la séquence, la conditionnelle et la boucle.

- Les éléments fils d'un nœud séquence (blocs de base ou sous-arbres) doivent être évalués dans l'ordre de la séquence.
- En ce qui concerne les structures conditionnelles, le test doit être évalué avant les fils *then* et *else*.
- Les structures de boucle impliquent plusieurs évaluations des mêmes blocs de base à l'intérieur de la boucle. La figure 4.14 illustre l'ordre d'évaluation des éléments de la boucle. Le test est évalué une première fois, puis le corps de la boucle. Une deuxième

passe est nécessaire pour propager l'information calculée dans le corps de la boucle vers le test, et vers le corps de boucle lui même.

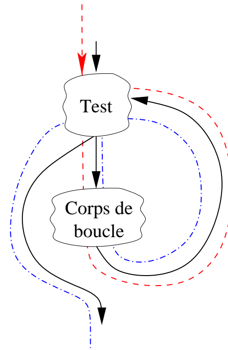


FIG. 4.14 – *Ordre de parcours pour le calcul des ACS dans une boucle*

Le principe général est ici de choisir, de façon statique, de ne recalculer les ACS que pour les nœuds qui peuvent le nécessiter (*i.e* les nœuds des boucles). Les bénéfices, en terme de réduction du nombre de calculs d'ACS, obtenus par cet ordre de parcours sont présentés sur un exemple dans [Col98].

4.3.3 Estimation de la présence des instructions

Le but de la simulation statique de cache d'instructions est de déterminer si une référence à une instruction sera un succès (*hit*) ou un échec (*miss*) lors de l'exécution du programme. Pour cela, on va observer les résultats obtenus lors de la phase de calcul des ACS et en tirer certaines conclusions sur le comportement des iblocs par rapport au cache d'instructions. Ces informations sur le comportement des iblocs nous permettront de déduire le comportement des instructions.

F. Mueller distingue quatre catégories d'instructions que l'on applique ici aux iblocs :

- «toujours *hit*» (*always hit*) : la référence à l'ibloc est toujours un *hit* dans le cache d'instructions.
- «toujours *miss*» (*always miss*) : la référence à l'ibloc est toujours un *miss* (*i.e.* on ne peut pas garantir que c'est un *hit*).
- «d'abord *miss*» (*first miss*) : la première référence à l'ibloc est un *miss* et les suivantes sont des *hit*.
- «conflit» (*conflict*) : la référence à l'ibloc peut être un *hit* ou un *miss*.

Cette classification des iblocs n'est pas indispensable à notre adaptation de la simulation statique de cache, mais elle permet d'illustrer plus intuitivement le comportement des

iblocs. Puisqu'un programme peut être composé de plusieurs boucles emboitées, la classification présentée par F. Mueller dans [AMWH94] associe à chaque instruction une catégorie potentiellement différente pour chacun de ses niveaux d'emboitement.

Notre approche est quelque peu différente. Nous remplaçons les multiples catégories associées à une instruction dans un emboitement de boucles par un unique *niveau d'absence* associé à un ibloc. Le *niveau d'absence* de l'ibloc $IB_{\alpha,x}$ (que l'on appelle *miss-level*$_{\alpha,x,v}$), est le *ln-level* en dessous duquel l'ibloc est garanti d'être dans le cache d'instructions au moment de son exécution et à partir duquel il est considéré comme provoquant un *miss* dans le cache.

La correspondance entre les *miss-levels* et les catégories de F. Mueller est la suivante :
- catégorie de $IB_{\alpha,x}$ = «toujours *hit*» ⇔ *miss-level*$_{\alpha,x}$ = [*never*].
- catégorie de $IB_{\alpha,x}$ = «toujours *miss*» ou «conflit» ⇔ *miss-level*$_{\alpha,x}$ = $[\perp_\alpha]$.
- catégorie de $IB_{\alpha,x}$ = «d'abord *miss*» ⇔ $[\perp_\alpha] \succ$ *miss-level*$_{\alpha,x} \succ$ [*never*].

Pour calculer ce *miss-level*$_{\alpha,x,v}$, on s'intéresse à l'ibloc $IB_{\alpha,x}$ du bloc de base BB_α. On examine l'ensemble des lignes d'ACS désignées par $\mathrm{ACS}_\alpha^{pré}[e,*]$ où $e = x$ modulo (L/V)) (*nb.* l'ensemble $\mathrm{ACS}_\alpha^{pré}[e,*]$ est l'ensemble des lignes de l'ACS pouvant contenir $IB_{\alpha,x}$).

L'absence de $IB_{\alpha,x}$ de tous les ensembles d'iblocs $\mathrm{ACS}_i^{pré}[e,*]$ assure son absence du cache d'instructions lors de son exécution (*miss-level*$_{\alpha,x}$ = $[\perp_\alpha]$).

S'il est présent dans $\mathrm{ACS}_\alpha^{pré}[e,*]$, on examine chacune des voies $\mathrm{ACS}_i^{pré}$[e,v] qui le contient (les autres voies ne sont pas examinées). Pour chaque voie v à considérer, on calcule un *miss-level*$_{\alpha,x,v}$. On obtient alors un *miss-level*$_{\alpha,x,v}$ pour chacune des voies contenant $IB_{\alpha,x}$. Enfin, le *miss-level*$_{\alpha,x}$ associé à $IB_{\alpha,x}$ est le plus petit des *miss-level*$_{\alpha,x,v}$.

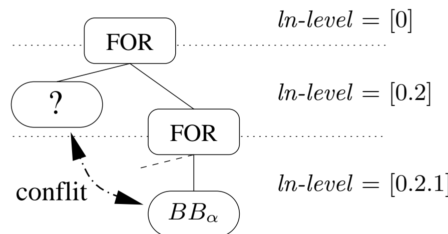


FIG. 4.15 – *Exemple d'un ibloc classé «d'abord miss»*

Étudions le cas d'un ibloc du bloc de base BB_α de la figure 4.15. Supposons que cet ibloc est contenu par les boucles [0], [0.2] et [0.2.1] dont les nombres maximum d'itérations sont respectivement M_1, M_2 et M_3. Supposons que le chargement de cet ibloc cause un *miss* à chaque première itération de la boucle [0.2] (c'est le cas s'il y a conflit avec un ibloc de la boucle [0]), le *miss-level* est alors [0] et on considère donc que l'ibloc est dans le cache pour

chaque itération de la boucle [0.2] et [0.2.1], et absent pour chaque itération de la boucle [0] et [ ]. Le nombre total de *miss* pour l'exécution du programme est alors M_1, et le nombre de *hit* est $M_1 \times ((M_2 \times M_3) - 1)$.

Le calcul du *miss-level* d'un ibloc est basé sur l'observation de l'$\text{ACS}^{pré}$ de son bloc de base. Par exemple, pour calculer le *miss-level* de l'ibloc $IB_{\alpha,x}$ on vérifie qu'il est présent dans l'$\text{ACS}^{pré}$ de BB_α. Mais la présence de l'ibloc $IB_{\alpha,x}$ est équivalente à la présence de l'ibloc $IB_{\beta,x}$, ou $IB_{\delta,x}$, car le fait que $IB_{\alpha,x}$ ou bien que $IB_{\beta,x}$ soit chargé dans le cache correspond à la même réalité, c'est-à-dire à la présence de toutes les instructions de l'ibloc numéro x quel que soit leur bloc de base d'origine. Ainsi, les présences d'$IB_{\alpha,x}$ et d'$IB_{\beta,x}$ sont équivalentes, et on va vérifier la présence de l'ibloc générique $IB_{*,x}$ pour le calcul du $\textit{miss-level}_{\alpha,x}$.

Le cas le plus simple est celui où l'ibloc considéré est absent de l'$\text{ACS}^{pré}$ de son bloc de base. L'absence de $IB_{*,x}$ dans $\text{ACS}^{pré}_\alpha$ signifie qu'aucun des chemins d'exécution menant à BB_α ne permet à $IB_{*,x}$ d'être chargé dans le cache avant l'exécution de BB_α. Dans ce cas, on est certain de son absence au moment de l'exécution du bloc de base. On lui attribue donc le *miss-level* $[\bot_\alpha]$, comme noté dans la formule ci-dessous.

$$\forall v, IB_{*,x} \notin \text{ACS}^{pré}_\alpha[e,v] \Leftrightarrow \textit{miss-level}_{\alpha,x} = [\bot_\alpha]$$

Sinon, à partir de chaque ensemble d'iblocs $\text{ACS}^{pré}_\alpha[e,v]$ contenant $IB_{*,x}$, on va calculer trois sous-ensembles notés $\bigtriangledown$, $\bigtriangleup$ et $\blacktriangle$. Ces sous-ensembles calculés à partir de la voie v pour la recherche du *miss-level* de $IB_{*,x}$ sont indicés par α, x, v.

- $\bigtriangledown_{\alpha,x,v}$ est le sous-ensemble de $\text{ACS}^{pré}_\alpha[e,v]$ constitué de tous les iblocs $IB_{\beta,y}$, excepté $IB_{*,x}$, tels que *ln-level* $[\bot_\beta]$ est plus petit ou égal (au sens de $\succeq$) à $[\bot_\alpha]$ (*i.e.* tous les iblocs en conflit pour la ligne de cache et englobés dans $[\bot_\alpha]$).
- $\bigtriangleup_{\alpha,x,v}$ est le sous-ensemble de $\text{ACS}^{pré}_\alpha[e,v]$ contenant tous les iblocs excepté $IB_{*,x}$ et les iblocs contenus dans $\bigtriangledown_{\alpha,x,v}$.
- $\blacktriangle_i[v]$ est le sous-ensemble de $\bigtriangleup_{\alpha,x,v}$ ne contenant que les iblocs en conflit avec $IB_{*,x}$ à un *ln-level* minimum.

Plus formellement :

$$\begin{aligned}
\bigtriangledown_{\alpha,x,v} &= \left\{IB_{\beta,y} \in \text{ACS}^{pré}_\alpha[e,v] \;\middle|\; (y \neq x) \wedge ([\bot_\alpha] \succeq [\bot_\beta])\right\} \\
\bigtriangleup_{\alpha,x,v} &= \left(\text{ACS}^{pré}_\alpha[e,v] \setminus \{IB_{*,x}\}\right) \setminus \bigtriangledown_{\alpha,x,v} \\
\blacktriangle_{\alpha,x,v} &= \{IB_{\beta,y} \in \bigtriangleup_{\alpha,x,v} \;|\; [\bot_\beta] = \textit{ln-level}_{\blacktriangle_{\alpha,x,v}}\} \\
&\quad \text{avec } \textit{ln-level}_{\blacktriangle_{\alpha,x,v}} = \underset{\forall IB_{\gamma,z} \in \bigtriangleup_{\alpha,x,v}}{Min} \Psi([\bot_\alpha],[\bot_\gamma])
\end{aligned}$$

La figure 4.16 présente le calcul des sous-ensembles $\bigtriangledown_{\gamma,15,0}$, $\bigtriangleup_{\gamma,15,0}$ et $\blacktriangle_{\gamma,15,0}$ à partir de l'état abstrait du cache avant exécution du bloc de base BB_γ ($\text{ACS}^{pré}_\gamma$).

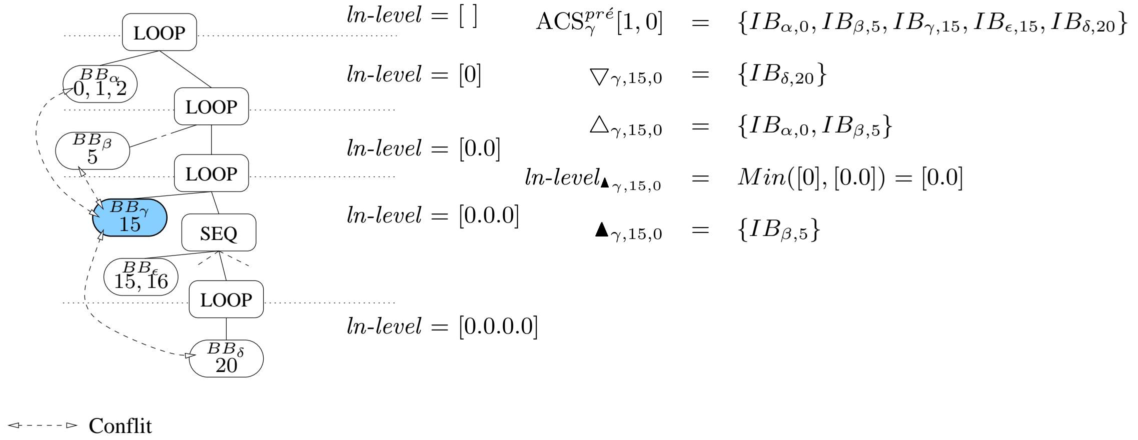


FIG. 4.16 – *Calcul des sous-ensembles* $\triangledown$, $\triangle$ *et* $\blacktriangle$ *de* $ACS_\gamma^{pré}[1,0]$

Ces sous-ensembles nous permettent d'obtenir un *miss-level* pour chaque voie v de l'$\text{ACS}_\alpha^{pré}[e,*]$ à considérer. Pour ce faire, on cherche à appliquer les règles ① à ④ à l'ibloc considéré.

① Si $\triangledown_{\alpha,x,v}$ n'est pas vide, $IB_{\alpha,x}$ est potentiellement en conflit avec un ibloc de *ln-level* inférieur pour l'occupation de cette ligne de cache. On considère donc qu'il est toujours absent du cache d'instructions au moment de son exécution.

② Si $\triangledown_{\alpha,x,v}$ et $\triangle_{\alpha,x,v}$ sont vides et que $IB_{*,x}$ est présent dans $\text{LCS}_\alpha^{pré}$[e,v], on considère qu'il est toujours présent dans le cache d'instructions au moment de son exécution.

③ Si $\triangledown_{\alpha,x,v}$ et $\triangle_{\alpha,x,v}$ sont vides et que $IB_{*,x}$ est absent de $\text{LCS}_\alpha^{pré}$[e,v], on considère qu'il est toujours présent dans le cache d'instructions au moment de son exécution **sauf** lors de sa première exécution.

④ Si $\triangledown_{\alpha,x,v}$ est vide mais pas $\triangle_{\alpha,x,v}$ on retrouve le cas de la figure 4.15, le *ln-level* où peut se produire le conflit est $\textit{ln-level}_{\blacktriangle_{\alpha,x,v}}$.

Ce qui se traduit plus formellement par :

①	$\textit{miss-level}_{\alpha,x,v} = [\bot_\alpha]$	$\Leftrightarrow$	$\triangledown_{\alpha,x,v} \neq \emptyset$
②	$\textit{miss-level}_{\alpha,x,v} = [never]$	$\Leftrightarrow$	$\triangledown_{\alpha,x,v} = \emptyset \wedge \triangle_{\alpha,x,v} = \emptyset \wedge IB_{*,x} \in \text{LCS}_\alpha^{pré}[e,v]$
③	$\textit{miss-level}_{\alpha,x,v} = [\,]$	$\Leftrightarrow$	$\triangledown_{\alpha,x,v} = \emptyset \wedge \triangle_{\alpha,x,v} = \emptyset \wedge IB_{*,x} \notin \text{LCS}_\alpha^{pré}[e,v]$
④	$\textit{miss-level}_{\alpha,x,v} = \textit{ln-level}_{\blacktriangle_{\alpha,x,v}}$	$\Leftrightarrow$	$\triangledown_{\alpha,x,v} = \emptyset \wedge \triangle_{\alpha,x,v} \neq \emptyset$

Enfin, on garde le plus petit des $\textit{miss-level}_{\alpha,x,v}$ calculés (pour les $\text{ACS}_\alpha^{pré}$[e,*] qui contenait $IB_{\alpha,x}$) et on l'associe à l'ibloc $IB_{\alpha,x}$.

$$\textit{miss-level}_{\alpha,x} = Min_v \ \textit{miss-level}_{\alpha,x,v}$$

4.3.4 Résultats de la simulation statique

Une fois la simulation statique de cache d'instructions effectuée, chaque ibloc est associé à un *miss-level* indiquant le niveau d'emboitement de la boucle où peut se produire un conflit d'accès au cache d'instructions. Les résultats de la simulation statique sont représentés par un ensemble, noté *IB-est* (estimation du comportement des iblocs), qui contient des couples $<$ ibloc, *miss-level* $>$. On regroupe les résultats concernant tous les iblocs d'un même bloc de base BB_α dans $IB\text{-}est_\alpha$. Par exemple, si on considère le bloc de base BB_α de la figure 4.7 (voir page 70), $IB\text{-}est_\alpha$ pourrait être (aux valeurs des *miss-levels* près) :

$$IB\text{-}est_\alpha = \{< 16, [0.2.0] >, < 17, [\ ] >, < 18, [never] >, < 19, [0.2] >\}$$

Cette structure de données est utilisée comme donnée d'entrée pour la prise en compte de l'effet du pipeline sur l'analyse statique de WCET décrite au paragraphe 4.5.

4.4 Prise en compte d'un mécanisme de prédiction de branchement

Le mécanisme de prédiction de branchement joue un rôle de plus en plus important pour l'efficacité des processeurs. Cet élément d'architecture permet d'éviter une part importante des pénalités temporelles liées aux instructions de transfert du flot de contrôle, et plus particulièrement à leur effet sur l'exécution pipelinée (*cf.* § 4.5).

Nous présentons ici, après une bréve introduction du mcanisme de prédiction de branchement (§ 4.4.1), une méthode appelée *simulation statique de prédiction de branchement* (voir § 4.4.2) par analogie avec la technique de prise en compte du cache d'instructions présentée précédemment, et qui en reprend plusieurs aspects.

La prise en compte du mécanisme de prédiction de branchement pour affiner les estimations de WCET n'avait, à notre connaissance, pas été étudiée avant notre proposition [CP00].

4.4.1 La prédiction de branchement : introduction

Les instructions de transfert du flot de contrôle sont des instructions particulières du jeu d'instructions dédiées à la gestion du flot de contrôle. Ces instructions, que nous désignons ici par le terme générique de *branchement*, sont les sauts (conditionnels et inconditionnels) et les appels et retours de fonctions.

4.4.1.1 But de la prédiction de branchement et impact sur le calcul de WCET

Le nombre d'étages des pipelines des processeurs actuels ne cesse d'augmenter (*e.g* 5 pour l'Intel Pentium et 20 pour sa dernière version, le Pentium 4). Quand un branchement est rencontré, il peut causer une discontinuité dans le flot d'instructions car le résultat d'un branchement conditionnel ainsi que sa cible ne sont connus qu'à la fin des étages d'exécution. Le processeur ne peut donc connaître la prochaine instruction à précharger après un branchement qu'après exécution de celui-ci. Ainsi, les branchements, qui représentent entre 15 et 30 % des instructions, peuvent causer une rupture du flot d'instructions dans le pipeline. Si le résultat des exécutions des branchements ne sont pas prédits, de nombreux cycles sont perdus à attendre l'exécution d'un branchement avant de pouvoir charger les instructions suivantes.

Pour éviter ces passages à vide dans le pipeline, coûteux en nombre de cycles, certains processeurs comportent un mécanisme de prédiction de branchement. Le but de ce mécanisme est de permettre de précharger et décoder le flot d'instructions au-delà des branchements. Pour ce faire, il prédit si un branchement a une forte probabilité d'être «pris» ou «non pris», ainsi que la cible du branchement.

Si la prédiction est vérifiée lorsque le branchement est traité par l'étage d'exécution du pipeline, alors les instructions qui ont été chargées par anticipation et qui suivent dans le pipeline sont bien celles qui doivent être exécutées. Dans ce cas, il n'y a aucun surcoût à ajouter au temps du branchement. En revanche, si la prédiction ne se vérifie pas, il faut vider

les instructions chargées par anticipation dans le pipeline et recommencer le chargement avec les bonnes instructions, d'où un temps d'exécution plus long du branchement (voir figure 4.17).

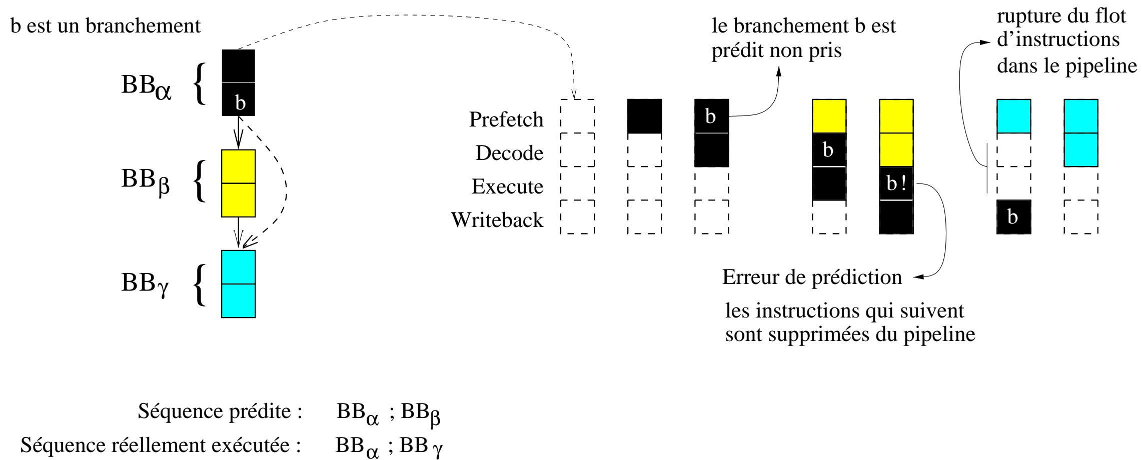


FIG. 4.17 – *Impact d'une erreur de prédiction sur le contenu du pipeline*

Pour incorporer le coût de la prédiction de branchement dans l'analyse de WCET, l'approche la plus pessimiste consiste à supposer que toutes les exécutions des branchements sont mal prédites, et donc à ajouter systématiquement le surcoût d'une mauvaise prédiction au temps d'exécution des branchements. Cette approche est sûre mais trop pessimiste, et la prise en compte de la prédiction de branchement doit permettre de réduire ce pessimisme tout en garantissant la sûreté de l'estimation.

Les prédictions que nous considérons ici sont effectuées à partir des informations recueillies lors des précédentes exécutions des branchements selon une technique détaillée au prochain paragraphe.

4.4.1.2 BTB à compteurs 2 bits à saturation

L'enregistrement de l'historique d'exécution des branchements peut être réalisé de diverses manières comme par exemple l'utilisation d'un cache dédié (*e.g.* le cache de branchements de l'Intel Pentium), ou encore un champ réservé dans les lignes du cache d'instructions (*e.g.* architecture UltraSPARC). Pour simplifier l'exposé, nous considérons le cas d'une architecture comportant un cache dédié aux informations de branchement : le BTB (pour *Branch Target Buffer*).

Lorsque le processeur exécute un branchement, le résultat de son exécution est enregistré dans le BTB. Puis, lorsqu'il est de nouveau rencontré dans le flot d'instructions chargées, et si les informations sur sa ou ses exécutions passées sont présentes dans le BTB, le mécanisme de prédiction de branchement les utilise pour prédire l'adresse de la prochaine instruction à charger en se basant sur l'historique des exécutions du branchement. De plus, un branchement

dont l'historique n'est pas dans le BTB, soit car c'est la première fois qu'il est rencontré, soit car il a été remplacé par celui d'un autre branchement dans le BTB, se verra attribuer une prédiction par défaut (*e.g.* la prédiction «non pris» sur l'Intel Pentium).

Pour rester le plus général possible, nous considérons un BTB qui est en fait un cache de L lignes dont le degré d'associativité est V. Chacune des lignes du BTB se décompose en :

[identificateur, cible, historique]

L'identificateur et la cible sont des adresses dans le programme. On considère ici que l'historique d'exécution d'un branchement est codé sur 2 bits (prédicteur de Smith [Smi81]). Ceci se traduit par un historique à quatre états (voir figure 4.18). Les transitions entre les états «fortement pris», «pris», «non pris» et «fortement non pris» ont lieu lorsque le branchement est pris (flèches +) ou non pris (flèches −). À chaque exécution d'un branchement, son historique (*i.e.* le compteur de la ligne de BTB qui lui est associé) est soit mis à jour, soit initialisé s'il était absent du BTB (*e.g.* initialisé à «fortement non pris» sur l'Intel Pentium).

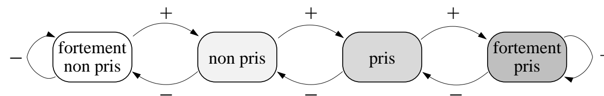


FIG. 4.18 – *Historique à 4 états codé par un compteur 2 bits*

Cette technique utilisant des compteurs à saturation comme éléments d'inertie pour mesurer le comportement de chaque instruction de branchement est utilisée dans de nombreux processeurs : Intel Pentium, AMD Nx586, Cyrix M1, PowerPC 604-620, Ultra Sparc, ALPHA.

4.4.2 Les états abstraits de cache de branchements (ABS)

Nous allons maintenant présenter les états abstraits de cache de branchements qui sont des représentations des contenus possibles du BTB à différents instants de l'exécution. Nous décrivons leur méthode de calcul, ainsi que la méthode itérative qui permet de calculer tous les états abstraits de cache de branchements d'un graphe de flot de contrôle.

4.4.2.1 Qu'est ce qu'un état abstrait de cache de branchements ?

Le rôle, la structure et la méthode de calcul des états abstraits de cache de branchements (notés ABS pour *Abstract BTB State*) sont très similaires à ceux des ACS définis au paragraphe 4.3.2.

Les ABS représentent, pour chaque bloc de base, les contenus possibles du BTB avant et après leur exécution. Ces contenus possibles sont des ensembles de branchements (des identificateurs de branchements pour être plus précis). Comme pour les ACS, à chaque bloc de base BB_α on associe un $\text{ABS}_\alpha^{pré}$ et un ABS_α^{post}.

Les ABS contiennent des ensembles de branchement. On conserve, pour les ABS, la même structure que celle définie pour les ACS, à savoir un tableau de L/V lignes et V colonnes. Par exemple, les caractéristiques du BTB de l'Intel Pentium sont : $L = 256$, $V = 4$.

4.4.2.2 Calcul des ABS

Calcul des ABS^{pré}

L'ABS$_\alpha^{pré}$ représente tous les contenus possibles du BTB avant l'exécution de BB_α. Pour le calculer, on doit prendre en compte toutes les possibilités et donc toutes les séquences de blocs de base pouvant conduire à BB_α. Pour ce faire, ABS$_\alpha^{pré}$ est calculé en réalisant l'union des ABS^{post} de tous les blocs de base précédant BB_α (*i.e.* l'ensemble $Pred(BB_\alpha)$). L'ACS$_\alpha^{pré}$ est calculé selon l'équation suivante :

$$\text{ABS}_\alpha^{pré} = \biguplus_{BB_\beta \in Pred(BB_\alpha)} \text{ABS}_\beta^{post} \tag{4.3}$$

Comme pour le calcul des ACS^{pré}, on ajoute un bloc de base fictif au début du graphe de flot de contrôle pour représenter l´état initial du BTB : *vide*.

Calcul des ABS^{post}

Pour un bloc de base BB_α donné, la valeur de l'ABS$_\alpha^{post}$ correspondant représente l'état du BTB après l'exécution du ou des branchements d'un bloc de base[1], et donc une modification de l'ABS$_\alpha^{pré}$. Les équations (4.4) gèrent l'ajout d'un branchement de $IB_{\alpha,x}$ dans un ABS^{post}. Le calcul de l'ABS$_\alpha^{post}$ s'effectue en deux temps : (i) copie de l'ABS$_\alpha^{pré}$ dans l'ABS$_\alpha^{post}$, puis (ii) ajout un à un et par ordre croissant de tous les branchements présents dans le bloc de base BB_α par application des équations (4.4). On note $Branche_{\alpha,i}$ le i$^{\text{ème}}$ branchement du bloc de base BB_α.

$$\begin{aligned} \text{ABS}_\alpha^{post}[e,v]' &= \begin{cases} \text{si } \Delta_{e,v} = \emptyset \text{ alors ABS}_\alpha^{post}[e,v] \\ \text{sinon } \Delta_{e,v} \end{cases} \\ \Delta_{e,v} &= \begin{cases} \text{si } v = 0 \text{ et } e = i \text{ } modulo \text{ } (L/V) \text{ alors } IB_{\alpha,i} \\ \text{sinon } \mathit{diff}_{e,v} \end{cases} \\ \mathit{diff}_{e,v} &= \{Branche_{x,y} \mid Branche_{x,y} \in \text{ABS}_\alpha^{post}[e, v-1] \wedge Branche_{*,y} \notin \Delta_{e,v-1}\} \end{aligned} \tag{4.4}$$

La fonction $\mathit{diff}_{e,v}$ calcule l'ensemble des branchements en conflit avec les branchements introduits dans la voie précédente $(v - 1)$. L'ensemble $\Delta_{e,v}$ est l'ensemble des branchements à introduire dans la voie v de l'ensemble e de l'ABS^{post}.

1. Un bloc de base peut contenir plusieurs branchements tant qu'il respecte la définition donnée au paragraphe 2.1.2.1

4.4.2.3 Calcul des états abstraits par itération de point fixe

En raison de la présence possible de cycles dans le graphe de flot de contrôle, le calcul des ABS est réalisé par itération de point fixe. Les ABS sont donc calculés par application des formules (4.3) et (4.4), jusqu'à stabilisation. Les optimisations de l'ordre de parcours de graphe de flot de contrôle et la preuve de terminaison de l'itération de point fixe présentées aux paragraphes 4.3.2.3 et 4.3.2.4 s'appliquent directement au calcul des ABS.

4.4.3 Estimation du mode de prédiction des branchements

Les ABS^{pré} associés aux blocs de base permettent de savoir si un branchement sera présent ou non dans le BTB lors de sa prédiction. En revanche, il ne permet pas de prévoir dans tous les cas le *résultat* de la prédiction. Dans ce paragraphe, on cherche à savoir quelle *mode de prédiction* sera appliquée pour chaque branchement. L'estimation de la correction de la prédiction en utilisant le mode de prédiction sera vu ultérieurement (§ 4.4.5). Pour ce faire, nous définissons les deux sources possibles de prédiction des branchements.

- Un branchement est dit **H-prédit** si l'historique de ses précédentes exécutions est présent dans le BTB lors de sa prédiction. La prédiction est alors calculée en fonction de l'état du compteur qui code l'historique.
- Un branchement est dit **D-prédit** si au moment de sa prédiction, il est absent du BTB soit car c'est la première fois qu'il est rencontré, soit car il a été remplacé dans le BTB par un autre branchement. Dans ce cas la prédiction est une prédiction par défaut. On considère ici que la prédiction par défaut est «non pris» (c'est le cas pour l'Intel Pentium).

Les ABS nous permettent de savoir si un branchement sera H-prédit, D-prédit ou bien si la source de prédiction est inconnue. Pour cela, on classe les branchements en quatre catégories :

- «toujours D-prédit» : si l'historique du branchement n'est jamais dans le BTB lors de la prédiction.
- «d'abord D-prédit» : si l'historique du branchement est absent du BTB la première fois, et présent pour les prédictions suivantes.
- «d'abord inconnu» : si on ne sait pas si l'historique est absent ou présent dans le BTB la première fois, mais qu'on est sûr de sa présence pour les prédictions suivantes.
- «toujours inconnu» : sinon (c'est le cas le plus pessimiste).

Remarquons que contrairement aux iblocs, un branchement ne peut être présent dans le BTB dès sa première exécution, il n'y a donc pas de catégorie «toujours H-prédit».

La catégorie d'un branchement dépend du contexte d'exécution et donc du *ln-level* que l'on observe. Il faudrait donc affecter des catégories potentiellement différentes selon le *ln-level* pour un même branchement. Dans l'exemple de la figure 4.19, un branchement de BB_α ($Branche_{\alpha,1}$) est exclu du BTB à chaque itération de la boucle [0.2]. Quand la boucle [0.2.1]

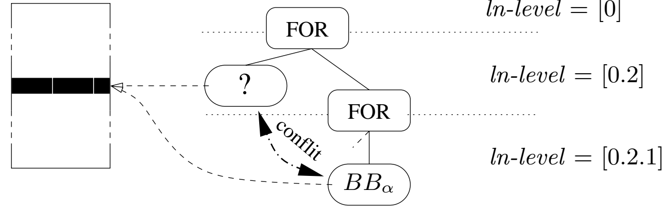


FIG. 4.19 – *Conflit dans le BTB*

est exécutée, $Branche_{\alpha,1}$ est absent du BTB la première fois (et donc prédit par défaut) mais sera présent pour les itérations suivantes (tant que le flot de contrôle ne sort pas de la boucle).

De même que pour le cache d'instructions, nous avons choisi de définir un *miss-level* pour caractériser ce type de situation sans avoir à associer plusieurs catégories différentes à un branchement. Le *miss-level* indique à partir de quel niveau d'emboitement le conflit a lieu et donc quand le branchement apparaît ou disparaît du BTB. Dans l'exemple précédent, le *miss-level* du branchement $Branche_{\alpha,1}$ est [0.2]. Donc, dans la mesure où il n'y a pas d'autres conflits, $Branche_{\alpha,1}$ est H-prédit pour toutes les boucles de *ln-level* inférieur à [0.2].

L'obtention de la catégorie et donc du *miss-level* d'un branchement passe par l'analyse de l'ABS^{pré} du bloc de base le contenant. On considère maintenant le branchement x du bloc de base BB_α, qui correspond à l'entrée e du BTB. La présence du branchement $Branche_{\alpha,x}$ dans un ABS^{pré} est équivalente à celle de $Branche_{\beta,x}$ ou $Branche_{\delta,x}$. Ces trois situations correspondent à la même réalité : la présence du branchement x. C'est pourquoi on ne différencie pas les branchements des différentes instances d'un même bloc de base. Et on désigne par $Branche_{*,x}$ le branchement x quelle que soit son bloc de base d'origine.

Si aucune des voies de l'entrée e de l'ABS$_\alpha^{pré}$ (*i.e.* ABS$_\alpha^{pré}[e,*]$) ne contient ce branchement, celui-ci est considéré «toujours D-prédit» et le $miss\text{-}level_{\alpha,x}$ est alors $[\perp_\alpha]$.

$$\forall v, Branche_{*,x} \notin \text{ABS}_i^{pré}[e,v] \Leftrightarrow miss\text{-}level_{\alpha,x} = [\perp], \text{catégorie} = \text{¡¡toujours D-prédit¿¿}$$

Sinon, les sous-ensembles $\bigtriangledown$, $\triangle$ et $\blacktriangle$ sont calculés à partir de l'ensemble de branchements ABS$_\alpha^{pré}[e,v]$ comme décrit au paragraphe 4.3.3 pour le cache d'instructions. On calcule alors un *miss-level* et une catégorie pour le branchement selon les règles ① à ④ pour chacune des voies v de l'ABS$_\alpha^{pré}[e,*]$ qui contient le branchement. Puis on garde le résultat le plus pessimiste, *i.e.* le *miss-level* minimum (au sens de $\succeq$) et la pire catégorie.

① $Branche_{\alpha,x}$ est estimé *toujours D-prédit* s'il est en conflit avec un branchement $Branche_{\beta,y}$ de *ln-level* inférieur ou égal tel que la relation $Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}$ soit vérifiée. Dans ce cas, le *miss-level* est $[\perp_\alpha]$.

② $Branche_{\alpha,x}$ est estimé *d'abord D-prédit* s'il n'est en conflit avec aucun branchement ou s'il est en conflit avec un branchement $Branche_{\beta,y}$ appartenant à $\blacktriangle_{\alpha,x,v}$ tel que la relation $Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}$ soit vérifiée. Dans ce cas, le *miss-level* est $ln\text{-}level_{\blacktriangle_{\alpha,x,v}}$.

③ $Branche_{\alpha,x}$ est estimé *d'abord inconnu* s'il n'est en conflit avec aucun branchement de *ln-level* inférieur ou égal, et qu'il n'y a aucun branchement $Branche_{\beta,y}$ dans l'ensemble $\blacktriangle_{\alpha,x,v}$ tel que la relation $Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}$ soit vérifiée. Dans ce cas, le *miss-level* est $\textit{ln-level}_{\blacktriangle_{\alpha,x,v}}$.

④ Si aucune des règles précédentes ne s'applique, $Branche_{\alpha,x}$ est estimé *toujours inconnu*, et le *miss-level* est $[\bot_\alpha]$.

Ce qui se traduit plus formellement par :

$$cat_{\alpha,x,v}, \textit{miss-level}_{\alpha,x,v} = \begin{cases} \text{toujours D-prédit}\ ,\ \textit{miss-level}_{\alpha,x,v} = [\bot_\alpha] \\ \quad \text{si} \quad (Branche_{\alpha,x} \in \text{ABS}_\alpha^{pré}[e,v]) \\ \qquad\quad \wedge(\exists Branche_{\beta,y} \in \triangledown_{\alpha,x,v} \mid Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}) \\ \\ \text{d'abord D-prédit}\ ,\ \textit{miss-level}_{\alpha,x,v} = \textit{ln-level}_{\blacktriangle_{\alpha,x,v}} \\ \quad \text{si} \quad (Branche_{\alpha,x} \in \text{ABS}_\alpha^{pré}[e,v]) \wedge (\triangledown_{\alpha,x,v} = \emptyset) \wedge (\triangle_{\alpha,x,v} = \emptyset) \\ \quad \text{ou} \quad (Branche_{\alpha,x} \in \text{ABS}_\alpha^{pré}[e,v]) \wedge (\triangledown_{\alpha,x,v} = \emptyset) \\ \qquad\quad \wedge(\exists\ Branche_{\beta,y} \in \blacktriangle_{\alpha,x,v} \mid Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}) \\ \\ \text{d'abord inconnu}\ ,\ \textit{miss-level}_{\alpha,x,v} = \textit{ln-level}_{\blacktriangle_{\alpha,x,v}} \\ \quad \text{if} \quad (Branche_{\alpha,x} \in \text{ABS}_\alpha^{pré}[e,v]) \wedge (\triangledown_{\alpha,x,v}] = \emptyset) \wedge (\triangle_{\alpha,x,v} \neq \emptyset) \wedge \\ \qquad (\nexists\ Branche_{\beta,y} \in \blacktriangle_{\alpha,x,v} \mid Branche_{\alpha,x} \overset{\neg\text{if}}{\rightsquigarrow} Branche_{\beta,y}) \\ \\ \text{toujours inconnu}\ ,\ \textit{miss-level}_{\alpha,x,v} = [\bot_\alpha] \\ \quad \text{otherwise} \end{cases}$$

Enfin, on garde le plus petit des $\textit{miss-level}[\alpha, x, v]$ calculés et on l'associe au branchement.

4.4.4 Hypothèses nécessaires à l'interprétation des catégories de branchements

Comme montré plus haut, l'analyse des ABS nous permet de connaître le *mode* des prédictions, mais ne permet pas de connaître statiquement les *résultats* des prédictions qui seront effectuées. Pour estimer la correction des prédictions en fonction de leur source, nous proposons d'utiliser le *type* des branchements. Les restrictions que nous avons établies sur le langage source analysable nous permettent de limiter le nombre de types de branchements à trois. Un branchement peut donc être :

- un *saut inconditionnel*, par exemple l'instruction *jump* sans condition ou bien les appels et retours de fonction,
- un *test de structure conditionnelle* qui réalise le branchement vers la branche *then* ou *else*,

– un *test de fin de boucle*.

Comme on l'a vu au paragraphe 2.1.2.5, les représentations logiques du programme sont fortement dépendantes du schéma de compilation mis en œuvre par le compilateur. Les hypothèses que nous formulons au paragraphe 4.4.4.1 fixent le schéma de compilation pour les deux structures principales du langage. Puis, nous posons et discutons deux hypothèses sur le comportement au pire cas des branchements (§ 4.4.4.2), qui nous permettent, dans le paragraphe 4.4.5, de dériver le résultat des prédictions à partir de leur mode de prédiction.

4.4.4.1 Hypothèses sur le schéma de compilation

De la manière de compiler le langage source en assembleur dépend la structure du graphe de flot de contrôle. Ce graphe étant la base de la technique de simulation statique de la prédiction de branchement, les schémas de compilation mis en œuvre par le compilateur ont un impact sur cette technique. Nous fixons ici le schéma de compilation des structures "If" et "For" du langage source.

- "**If**" : quand le branchement *test* est exécuté, il est soit **non pris** (*i.e.* "pas de saut") pour passer au *then*, soit **pris** (*i.e.* "saut") pour exécuter la branche *else* (voir figure 4.20.a).
- "**For**" : quand le branchement *test* est exécuté, il est soit **pris** pour exécuter le corps de la boucle (saut), soit **non pris** pour en sortir (pas de saut). Notons aussi que les branchements inconditionnels de rebouclage et de sortie sont toujours **pris** (voir figure 4.20.b).

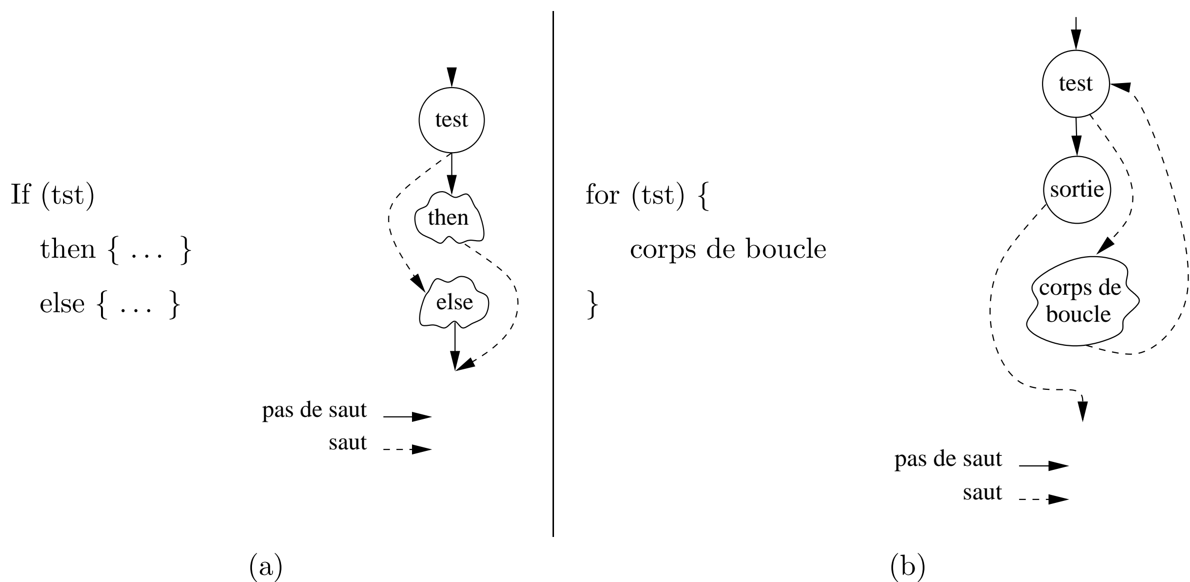


Fig. 4.20 – *Schémas de compilation des structures boucle et conditionnelle*

Les résultats donnés dans les paragraphes suivants peuvent s'adapter facilement à un autre schéma de compilation.

4.4.4.2 Hypothèses sur le comportement des branchements au pire cas

En supposant les schémas de compilation ci-dessus, on peut faire deux hypothèses sur le comportement des branchements, issus d'informations sur le type des branchements.

$\mathcal{H}1$: les branchements H-prédit sont toujours prédit correctement exception faite des branchements qui testent le rebouclage (ou la fin) d'une boucle pour lesquels la prédiction de la sortie de la boucle est toujours incorrecte.

$\mathcal{H}2$: dans le chemin d'exécution correspondant au pire temps d'exécution, et pour une structure conditionnelle donnée, on ne trouve qu'une seule des deux branches ("*then*" ou "*else*") de cette structure. Ce qui signifie qu'un branchement test d'une structure conditionnelle a un comportement constant dans le pire chemin d'exécution. Par conséquent, un tel branchement est prédit correctement lorsqu'il est H-prédit.

L'hypothèse $\mathcal{H}1$ est basée sur le fait que dans le pire des cas, toutes les boucles seront exécutées autant de fois que le permet la contrainte sur leur nombre maximum d'itérations. Considérons maintenant les trois types de branchement définis précédemment (inconditionnel, test de conditionnelle, test de boucle).

Le comportement d'un branchement inconditionnel est toujours le même, et sa prédiction sera correcte s'il est H-prédit. Dans le cas d'un branchement de type test de structure conditionnelle, on se ramène au cas précédent grâce à l'hypothèse $\mathcal{H}2$. Enfin, si le branchement est un test de fin de boucle, son historique passe dans l'état «fortement pris» lors de son premier enregistrement (*cf.* § 4.4.1.2). Puisqu'il est impossible de sortir deux fois d'une boucle sans y ré-entrer entre temps, l'historique d'un tel branchement oscille entre les états «fortement pris» et «pris». Cette hypothèse n'est valide que si les nombres maximum d'itérations des boucles sont non nuls (dans le cas contraire, une telle boucle est évidemment inutile).

Selon l'hypothèse $\mathcal{H}2$, pour obtenir le pire cas d'exécution, il suffit de considérer une seule des deux branches d'une structure conditionnelle. En effet, pour cette dernière, le cas le plus défavorable est celui où la branche dont le WCET est le plus important est toujours choisie. On peut alors considérer ce branchement comme un branchement inconditionnel. En revanche, en ce qui concerne le branchement, le cas le plus défavorable est celui où on alterne entre les exécutions des deux branches. Les prédictions peuvent alors se révéler être systématiquement fausses si son historique oscille entre les deux états centraux («pris» et «non pris»). L'hypothèse $\mathcal{H}2$ n'est donc valable qu'à la condition de vérifier que la différence de WCET entre les deux branches est au moins égale au coût de deux mauvaises prédictions. Si ce n'est pas le cas, et pour ne pas remettre en question l'hypothèse $\mathcal{H}1$, on suppose que la source de prédiction de ce branchement est inconnue.

4.4.5 Estimation de la correction des prédictions

Pour chaque branchement conditionnel, on a deux possibilités de branchement. Les erreurs de prédiction pour ces deux possibilités de branchement n'ont pas obligatoirement lieu en même temps et donc pas pour les mêmes *ln-levels*.

Par exemple, un branchement x du bloc de base BB_α dont le *miss-level* est $[\perp_\alpha]$ sera toujours D-prédit «non pris». Sa prédiction sera donc toujours incorrecte lorsqu'il est pris et toujours correcte lorsqu'il n'est pas pris. Cette situation peut être représentée par de nouveaux attributs du branchement, les *error-levels*. On associe à chaque branchement deux *error-levels*, un par possibilité de branchement. Ils sont marqués p pour «pris» et $\neg p$ pour «non pris». Dans notre exemple du branchement x du bloc de base BB_α, on a $\textit{error-level}_{\alpha,x,p}=[\perp_\alpha]$ car sa prédiction est toujours (*i.e.* pour chaque itération de la boucle $[\perp_\alpha]$) incorrecte lorsqu'il est pris. Et on a $\textit{error-level}_{\alpha,x,\neg p}=[never]$ car sa prédiction n'est jamais incorrecte lorsqu'il n'est pas pris.

Les *error-levels* sont calculés à partir du *miss-level*, de la catégorie, et du type des branchements comme suit :

- **Toujours D-prédit** : $\textit{error-level}_{\alpha,x,\neg p} = [never]$, $\textit{error-level}_{\alpha,x,p} = [\perp_\alpha]$.
  Le branchement est toujours prédit "non pris". Les prédictions sont fausses pour la branche *saut* quelque soit le *ln-level* considéré. Rappellons que le *ln-level* $[\perp_\alpha]$ est le *ln-level* de la boucle la plus emboitée qui contient $Branche_{\alpha,x}$.

- **Toujours inconnu** : $\textit{error-level}_{\alpha,x,\neg p} = [\perp_\alpha]$, $\textit{error-level}_{\alpha,x,p} = [\perp_\alpha]$.
  On ne sait rien de la source de prédiction de ce branchement, on suppose donc que les prédictions sont toujours erronées.

- **D'abord D-prédit** : $\textit{error-level}_{\alpha,x,\neg p} = [never]$, $\textit{error-level}_{\alpha,x,p} = \textit{miss-level}_{\alpha,x}$.
  La valeur du *ln-level* associée à la branche *seq* (*i.e.* *ln-level* L) dépend du rôle de $Branche_{\alpha,x,seq}$.
  Si c'est un test de boucle on doit prendre en compte le cas particulier de la sortie de la boucle. Le branchement sera D-prédit à chaque itération de la boucle $\textit{miss-level}_{\alpha,x}$, puis correctement H-prédit pour toutes les itérations de la boucle dont il teste la sortie (selon l'hypothèse $\mathcal{H}1$). Le *ln-level* associé à la branche *saut* qui correspond à une itération de la boucle est donc $\textit{miss-level}_{\alpha,x}$. La sortie de la boucle est toujours incorrectement H-prédite, c'est pourquoi on lui associe le *ln-level* $[\perp_\alpha]+1$.
  Pour un branchement qui n'est pas un test de boucle, on sait qu'il sera D-prédit "non pris" à chaque itération de la boucle $\textit{miss-level}_{\alpha,x}$, on aura donc une erreur sur la prédiction de la branche *saut* à chaque itération de la boucle $\textit{miss-level}_{\alpha,x}$. Les prédictions suivantes sont des H-prédictions, et d'après l'hypothèse $\mathcal{H}2$ elles sont correctes. Ainsi, la branche *seq* n'est jamais incorrectement prédite, et on associe le *ln-level*$[never]$ à la branche notée *seq*.

– **D'abord inconnu** : $error\text{-}level_{\alpha,x,\neg p} = miss\text{-}level$, $error\text{-}level_{\alpha,x,p} miss\text{-}level$.
  Cette catégorie est assez similaire à la précédente à la différence que le branchement peut être d'abord D-prédit ou H-prédit. C'est pourquoi la pénalité de temps (le *miss-level*) est associée aux deux possibilités de branchement.

4.4.6 Résultats de la simulation statique

Un bloc de base peut contenir plusieurs branchements, mais contient au plus un branchement conditionnel (en fin de bloc). À chaque branchement on a associé deux *error-levels*. Un seul est utile pour les branchements inconditionnels : $error\text{-}level_p$. Les résultats de la simulation statique de prédiction de branchement sont présentés sous forme d'un ensemble de couples et de triplets appelé ensemble *Préd-est* (pour estimation des prédictions) du bloc de base. Les couples concernant les branchements inconditionnels sont de la forme $< Branche_{\alpha,x}, error\text{-}level_{\alpha,x,?} >$, tandis que les triplets $< Branche_{\alpha,x}, error\text{-}level_{\alpha,x,\neg p}, error\text{-}level_{\alpha,x,p} >$ représentent les résultats de la simulation statique de prédiction des branchements concernant les branchements conditionnels. Ces résultats sont utilisés comme données d'entrée de la simulation statique de pipeline au paragraphe 4.5.

4.5 La simulation statique de pipeline

Le but de l'exécution pipelinée est de réduire le nombre de cycles nécessaires à l'exécution de chaque instruction. Le pipeline permet d'introduire du parallélisme dans l'exécution des instructions. Pour ce faire, l'exécution d'une instruction est découpée en étapes, et une étape de l'exécution d'une instruction est effectuée par un étage du pipeline. Ainsi, les étages du pipeline peuvent effectuer simultanément chacun une étape d'une instruction différente.

La méthode de prise en compte de l'effet du pipeline sur le temps d'exécution au pire cas consiste d'une part à simuler statiquement le remplissage des différents étages du pipeline sous forme de tables d'occupation du pipeline (en utilisant les informations récoltées pendant les simulations statiques de caches d'instructions et de prédictions de branchement, voir § 4.5.1), et d'autre part, à simuler l'effet du pipeline sur les séquences de blocs de base, c'est-à-dire l'effet inter blocs de base (*cf.* § 4.5.2).

4.5.1 Simulation de l'exécution pipelinée d'un bloc de base

La simulation de l'exécution d'un bloc de base dans le pipeline permet d'estimer le pire temps nécessaire à son exécution. Ce pire temps d'exécution dépend du *contexte* qu'on considère et qui permet de savoir si les instructions du bloc de base seront présentes ou absentes du cache d'instructions lors de leur exécution, et si les branchements du bloc de base seront correctement prédits. La simulation de pipeline donne donc une estimation du WCET d'un bloc de base pour un contexte particulier, et ce résultat peut varier selon le contexte considéré. Comme on l'a vu précédemment, le contexte d'exécution est principalement caractérisé par les *ln-levels*. Le résultat de l'exécution du bloc de base BB_α par le pipeline est donc potentiellement différent pour *tous* les *ln-levels* compris entre $[\bot_\alpha]$ et $[\ ]$.

4.5.1.1 La simulation par table de réservation de ressource

Pour estimer le temps d'exécution au pire cas d'une séquence d'instructions nous simulons statiquement l'occupation des étages du pipeline. On représente les étages du pipeline (les 5 étages du pipeline Pentium dans notre exemple) par un ensemble de ressources, les instructions étant quant à elles représentées par des utilisations de ces ressources. On représente alors l'utilisation des ressources dans le temps par une table (table de réservation décrite dans [Kog81]) comme l'illustre la figure 4.21. Dans cette table de réservation des étages du pipeline le temps est mesuré sur l'axe horizontal.

Cette table de réservation est une représentation intermédiaire utilisée pour le calcul du WCET d'un bloc de base et le WCET est obtenu en calculant la longueur de table occupée (21 cycles dans l'exemple de la figure 4.21).

Les résultats d'analyse concernant le cache d'instructions et le mécanisme de prédiction de branchement sont utilisés pour remplir la table de réservation. Les résultats de la simulation statique du cache d'instructions sont utilisés pour déterminer le temps d'occupation du

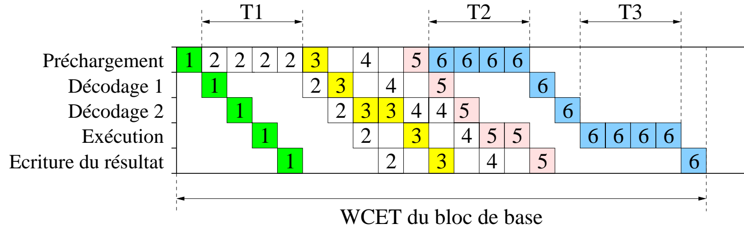


FIG. 4.21 – *Détail de l'exécution d'un bloc de base dans le pipeline*

premier étage (*i.e.* l'étage de préchargement) pour chaque instruction. Dans notre exemple, les temps de préchargement des instructions 2 et 6, qui causent des *miss* dans le cache d'instructions, sont représentés par des occupations plus longues de cet étage : $T1$ et $T2$.

De même, si on suppose que l'instruction 6 est un branchement, l'utilisation anormalement importante pour ce type d'instruction de l'étage d'exécution ($T3$) est due au fait qu'on sait qu'elle sera mal prédite par le mécanisme de prédiction de branchement. Cette information est fournie par la simulation statique de prédiction de branchement.

4.5.1.2 Utilisation des résultats de simulation précédents

La simulation statique de pipeline requiert plusieurs informations : (i) une description du bloc de base, c'est-à-dire la liste de ses instructions et leur décomposition en iblocs, (ii) des informations sur l'utilisation (au pire cas) de chacun des étages du pipeline par les instructions (utilisation de ressources, temps de traitement), (iii) les résultats des simulations statiques de prédiction de branchement (*cf.* § 4.4) et de cache d'instructions (*cf.* § 4.3) pour le bloc de base considéré.

Résultats de la simulation statique de cache d'instructions.
Ces résultats portent sur les iblocs, et doivent être transformés pour estimer le temps d'exécution d'une d'instruction.

Considérons léstimation de WCET du bloc de base BB_α dans le contexte décrit par le *ln-level* L. Pour ce faire, on cherche à savoir quelles instructions de BB_α causeront un *miss* dans le cache lors de leur exécution au *ln-level* L. Le résultat de la simulation statique de cache d'instructions concernant BB_α nous permet de savoir si un ibloc de BB_α est présent dans le cache au moment de son exécution pour un *ln-level* donné, et ce en utilisant la règle suivante, qui découle de la définition des *miss-levels* (*cf.* page 71).

L'exécution d'une instruction du bloc de base BB_α cause un *miss* dans le cache si la simulation statique de cache d'instructions a associé à l'un des iblocs $IB_{\alpha,x}$, dans lequel elle est en première position, un *miss-level* tel que $L \succeq$ *miss-level*$_{\alpha,x}$, sinon c'est un *hit*.

Cette règle permet de passer des informations sur la présence/absence des iblocs à la

présence/absence des instructions. On l'utilise pour calculer le sous-ensemble des instructions du bloc de base BB_α qui causent un *miss* au *ln-level* L, que nous nommons $INSTmiss_{\alpha,L}$.

Résultats de la simulation statique de prédiction de branchement.
L'ensemble *Préd-est*$_\alpha$ est le résultat de la simulation statique de prédiction de branchement concernant les branchements du bloc de base BB_α.

Pour un branchement $Branche_{\alpha,x}$, un *ln-level* donné (L) et une possibilité de branchement b, on sait que le branchement est estimé mal prédit si la relation $L \succeq$ *error-level*$_{\alpha,x,b}$ est vérifiée, sinon le branchement est estimé correctement prédit.

On utilise cette règle pour calculer l'ensemble des branchements du bloc de base BB_α qui sont supposés mal prédit au *ln-level* L, que nous nommons $BRANCHerr_{\alpha,L}$.

4.5.1.3 La fonction de simulation d'occupation du pipeline

La mise en œuvre de la simulation statique de pipeline pour un *ln-level* L donné est effectuée par une fonction de simulation d'occupation du pipeline. Les paramètres de cette fonction sont :

- le bloc de base dont on veut estimer le WCET (BB_α),
- le résultat du dernier branchement du bloc de base (pris ou non pris) s'il y a lieu,
- l'ensemble $INSTmiss_{\alpha,L}$,
- et l'ensemble $BRANCHerr_{\alpha,L}$,

Avec ces paramètres, la fonction *PipeSim* calcule le temps d'exécution au pire cas du bloc de base dans le contexte décrit par $INSTmiss_{\alpha,L}$ et $BRANCHerr_{\alpha,L}$: c'est la longueur de la table d'occupation (*cf.* figure 4.21).

$$PipeSim(BB_\alpha, branch, INSTmiss_{\alpha,L}, BRANCHerr_{\alpha,L})$$

L'occupation des étages de décodage et d'écriture des résultats dépend uniquement du type des instructions. On doit avoir des informations sur l'occupation des étages du pipeline par les différentes instructions du jeu d'instructions. L'occupation de l'étage de préchargement dépend du comportement des instructions vis à vis du cache d'instructions (présent dans $INSTmiss_{\alpha,L}$ ou non). Enfin, le temps d'occupation de l'étage d'exécution par les instructions de branchement dépend de la correction de leur prédiction (présent dans $BRANCHerr_{\alpha,L}$ ou non).

Le résultat de *PipeSim* est associé à un *ln-level* particulier, car suivant le *ln-level* L choisi, les paramètres passés à la fonction (les 2 derniers) et donc son résultat vont varier.

4.5.1.4 Résultats de la simulation d'exécution pipelinée d'un bloc de base

Puisque l'estimation du WCET d'un bloc de base BB_α par la fonction *PipeSim* dépend du *ln-level* considéré, l'évaluation de cette fonction pour les *ln-levels* entre $[\perp_\alpha]$ et $[\,]$ donnera plusieurs WCET potentiellement différents pour un même bloc de base.

À partir de maintenant, les WCET manipulés ne sont plus des WCET scalaires tels que définis dans le schéma temporel de base (tableau 2.1), mais ce sont des structures plus complexes dans lesquelles la dépendance au *ln-level* peut être representée.

De plus, certains blocs de base ont deux possibilités de branchement en fin de bloc : «pris» ou «non pris». Quand c'est le cas, la durée d'exécution de ces deux possibilités de branchement varie selon le résultat de la prédiction du branchement. Les résultats de la simulation statique associée dépendent eux aussi du *ln-level* considéré.

Ainsi, chaque bloc de base peut avoir au plus deux WCET potentiellement différents pour un même *ln-level* (correspondant aux deux possibilités de branchements) et ces couples de WCET peuvent être différents en fonction des *ln-levels*.

On peut cependant remarquer qu'il est inutile de calculer la valeur de *PipeSim* pour tous les *ln-levels* entre $[\perp]$ et $[\,]$. En effet, l'estimation du WCET de BB_α ne change que si un des paramètres de la fonction *PipeSim* change, ce qui ne peut être le cas que pour les *ln-levels* présents soit dans *Préd-est*$_\alpha$ soit dans *IB-est*$_\alpha$. Ainsi, pour calculer une représentation du WCET du bloc de base BB_i, on évalue *PipeSim* pour tous les *ln-levels* de *Préd-est*$_\alpha$ et *IB-est*$_\alpha$. Ces *ln-levels* sont regroupés dans une liste de *ln-level* notée Λ_α, ordonnée selon l'ordre $\succ$ ($\Lambda_\alpha = L_1 \dots L_n$ avec $L_{k+1} \succ L_k$).

Le calcul de la représentation du WCET du bloc de base BB_α s'effectue en deux étapes.

Premièrement, le WCET de BB_α est calculé pour chaque possibilité de branchement (p pour pris, et $\neg p$ pour non pris) et pour chaque *ln-level* de Λ_α. Pour k allant de 1 à n, ces WCET sont exprimés par :

$$\begin{aligned} T_{\alpha,p,k} &= PipeSim(BB_\alpha, p, INSTmiss_{\alpha,L_k}, BRANCHerr_{\alpha,L_k}) \\ T_{\alpha,\neg p,k} &= PipeSim(BB_\alpha, \neg p, INSTmiss_{\alpha,L_k}, BRANCHerr_{\alpha,L_k}) \\ L_k &= k^{\text{ème}} \text{ élément de } \Lambda_\alpha \end{aligned}$$

Deuxièmement, un WCET fictif est calculé pour chaque possibilité de branchement. Il servira de base à une représentation incrémentale du WCET.

$$\begin{aligned} T_{\alpha,p,0} &= PipeSim(BB_\alpha, p, \emptyset, \emptyset) \\ T_{\alpha,\neg p,0} &= PipeSim(BB_\alpha, \neg p, \emptyset, \emptyset) \end{aligned}$$

Il est associé au *ln-level* du bloc de base ($[\bot_\alpha]$) et représentent l'exécution "optimiste" de BB_α, c'est-à-dire sans aucun *miss* ni aucune erreur de prédiction.

Ces valeurs de WCET vont nous servir de base pour la construction d'une *représentation incrémentale* du WCET. Les WCET $T_{\alpha,p,0}$ et $T_{\alpha,\neg p,0}$ sont les valeurs de base et sont associés au *ln-level* $[\bot_\alpha]$. On va maintenant ajouter les coûts des *miss* et des erreurs de prédiction pour chacun des *ln-levels* de Λ_α. La différence entre un WCET $T_{\alpha,p,k}$ et le résultat précédent $T_{\alpha,p,k-1}$ représente le surcoût en temps dû au *ln-level* L_k. Les deux WCET associés à un bloc de base ayant deux possibilités de branchement en fin de bloc sont :

$$\begin{aligned}
WCET(BB_\alpha, p) &= \{< T_{\alpha,p,0}, [\bot_\alpha] >, < t_{\alpha,p,1}, L_1 >, \ldots, < t_{\alpha,p,n}, L_n >\} \\
WCET(BB_\alpha, \neg p) &= \{< T_{\alpha,\neg p,0}, [\bot_\alpha] >, < t_{\alpha,\neg p,1}, L_1 >, \ldots, < t_{\alpha,\neg p,n}, L_n >\} \\
\text{avec } t_{\alpha,p,k} &= T_{\alpha,p,k} - T_{\alpha,p,k-1} \\
\text{avec } t_{\alpha,\neg p,k} &= T_{\alpha,\neg p,k} - T_{\alpha,\neg p,k-1}
\end{aligned}$$

Cette représentation incrémentale des WCET sera introduite dans un nouveau schéma temporel présenté au paragraphe 4.6.

4.5.2 Simulation de l'effet inter bloc de base du pipeline

Le pipeline permet l'exécution simultanée de plusieurs instructions, permettant ainsi l'exécution d'un bloc de base avant la fin de l'exécution du précédent. Le temps d'exécution d'une séquence de deux blocs de base peut donc être inférieur à la somme des temps d'exécution de ces deux blocs de base. Nous présentons ici un moyen de prendre en compte l'effet du pipeline entre les blocs de base.

4.5.2.1 Simulation de séquences de deux blocs

Le principe de cette technique est la simulation statique de l'occupation du pipeline par deux blocs de base à la suite. Elle est réalisée par la fonction *PipeSim*2, définie ci-dessous, qui estime la réduction de WCET due au parallélisme d'exécution entre blocs qui se suivent le long d'un chemin d'exécution. Les paramètres de *PipeSim*2 sont de même nature que ceux de *PipeSim* (voir § 4.5.1.3), mais concernent deux blocs de base consécutifs dans le graphe de flot de contrôle.

$$PipeSim2(BB_\alpha, branch_\alpha, BB_\beta, INSTmiss_{\alpha,L}, INSTmiss_{\beta,L}, BRANCHerr_{\alpha,L}, BRANCHerr_{\beta,L})$$

Le résultat de la fonction représente un gain de temps, il est donc négatif.

Dans l'exemple suivant (*cf.* figure 4.22), on considère deux séquences de deux blocs de base : $BB_\alpha \rightarrow BB_\beta$ et $BB_\alpha \rightarrow BB_\gamma$. Les réductions des WCET dues au parallélisme d'exécution au *ln-level* L entre BB_α et BB_β d'une part, et BB_α et BB_γ d'autre part sont respectivement exprimées par :

- $PipeSim2(BB_\alpha, \neg p, BB_\beta, \ldots, L)$

– $PipeSim2(BB_\alpha, p, BB_\gamma, \dots, L)$

Les choix de branchement de fin d'exécution des blocs BB_β et BB_γ n'ont aucune influence sur le résultat.

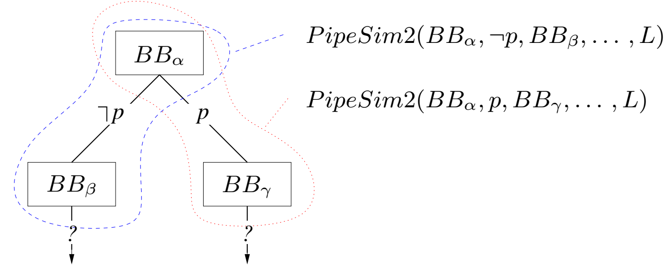


FIG. 4.22 – *Illustration de l'effet inter bloc de base du pipeline*

4.5.2.2 Les résultats de la simulation de l'effet inter bloc de base du pipeline

Comme c'était le cas pour $PipeSim$, le résultat de $PipeSim2$ dépend du *ln-level* considéré. On cherche à savoir quels *ln-levels* doivent être pris en compte pour le calcul de l'effet inter bloc de base du pipeline. L'extrait d'arbre syntaxique de la figure 4.23 présente le cas de l'exécution en séquence de deux blocs de base appartenant à des *ln-levels* différents (*i.e.* dans deux boucles différentes).

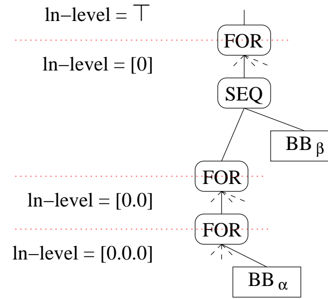


FIG. 4.23 – *Exécution séquentielle de deux blocs de base de ln-levels différents*

Dans cette exemple, la séquence $BB_\alpha \rightarrow BB_\beta$ est exécutée à chaque itération de la boucle [0]. Donc, le *ln-level* [0] et tous ses supérieurs ([ ] dans l'exemple) doivent être pris en compte.

On définit $\Lambda_{\alpha\beta}$ comme étant la liste ordonnée selon $\succ$ des *ln-levels* utiles au calcul du gain de temps de la séquence $BB_\alpha \rightarrow BB_\beta$. Cette liste est définie par :

$$L' \in \Lambda_{\alpha\beta} \Leftrightarrow \big(L' = \Psi([\bot_\alpha], [\bot_\beta])\big) \vee \big((L' \in (\Lambda_\alpha \cup \Lambda_\beta)) \wedge (L' \succeq \Psi([\bot_\alpha], [\bot_\beta]))\big)$$

La fonction $PipeSim2$ est évaluée pour chaque *ln-level* de $\Lambda_{\alpha\beta}$. Il s'agit de fusionner au plus juste les tables de réservation (*cf.* figure 4.24) représentant les deux blocs (les tables de réservation créées lors du calcul de $PipeSim$ peuvent être réutilisées).

FIG. 4.24 – *Assemblage des tables de réservation de deux blocs de base*

La possibilité de branchement (notée p ou $\neg p$ précédemment) est fixée par l'enchaînement des blocs de base. On note $next$ le branchement réalisé pour passer de BB_α à BB_β.

Comme pour la simulation intra blocs de base, les résultats sont représentés de manière incrémentale, une valeur de base du gain temporel, G_0, est calculée pour le *ln-level* L'_0 (premier élément de $\Lambda_{\alpha\beta}$), puis les gains temporels supplémentaires pour les autres *ln-levels* de $\Lambda_{\alpha\beta}$ sont calculés en faisant la différence entre G_k et G_{k-1}.

$$\begin{aligned}\Lambda_{\alpha\beta} &= L'_0 \ldots L'_m \text{ avec } L'_{k+1} \succ L'_k \\ G_k &= PipeSim2(BB_\alpha, next, BB_\beta, ?, \ldots, L'_k) \\ g_k &= G_k - G_{k-1}\end{aligned}$$

On ajoute alors à la représentation incrémentale du WCET d'un bloc de base, telle qu'elle a été définie au paragraphe 4.5.1.4, l'ensemble $INTER_BB$ pour prendre en compte l'effet inter blocs de base du pipeline.

$$INTER_BB^{next}_{\alpha\beta} = \{< G_0, L'_0 >, < g_1, L'_1 >, \ldots, < g_m, L'_m >\}$$

La nouvelle représentation incrémentale du WCET d'un bloc de base est :

$$WCET(BB_\alpha, next) = \{< T_0, [\perp] >, < t_1, L_1 >, \ldots, < t_n, L_n >, < G_0, L'_0 >, < g_1, L'_1 >, \ldots, < g_m, L'_m >\}$$

avec $next \in \{ p \, , \, \neg p \}$.

4.6 Adaptation du schéma temporel aux représentations de WCET incrémentales

L'analyse statique de WCET à base d'arbre syntaxique calcule l'estimation du WCET du programme à partir des WCET locaux (les WCET des blocs des base) par un parcours de bas en haut de l'arbre syntaxique. Le WCET d'un construction syntaxique est calculé en utilisant les WCET de ses constituants (blocs de base ou sous-arbres).

	Formules pour le calcul du WCET : $W(S)$
$S = S_1; \ldots; S_n$	$WCET(S) = WCET(S_i) + \ldots + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
$S =$ loop(tst) S_1	$WCET(S) = maxiter \times (WCET(Test) + WCET(S_1)) + WCET(Test)$ où *maxiter* est le nombre maximum d'itérations.
$S =$ bloc de base BB_x	$WCET(S) =$ WCET du bloc de base BB_x

TAB. 4.1 – *Formules de calcul du WCET [PK89] (sans prise en compte de l'architecture)*

Les formules originales, proposées par Puschner et Koza [PK89] pour une architecture sans cache ni pipeline, permettent de calculer le temps d'exécution maximum de chaque construction syntaxique du langage (voir table 4.1).

Ce schéma temporel suppose une représentation du WCET des blocs de base par des scalaires.

Pour utiliser une telle méthode de calcul basée à la fois sur l'arbre syntaxique et nos résultats locaux non scalaires (les représentations incrémentales des WCET des blocs de base), il nous faut définir un nouveau schéma temporel. On cherche à adapter les formules originales pour travailler sur des ensembles de couples $< WCET, ln\text{-}level >$. Le schéma temporel de Puschner et Koza est modifié pour :

- prendre en compte le *ln-level* de la construction syntaxique dont on veut calculer le WCET,
- adaptater les opérateurs aux WCET non scalaires.

Prise en compte du *ln-level*

Pour prendre en compte les *ln-levels*, la première modification est l'ajout d'un paramètre aux formules de calcul du WCET des nœuds de l'arbre : le niveau d'emboitement de boucle courant. Ce *ln-level* qui est associé à chaque nœud du graphe indique dans quelle boucle (la plus emboitée) se trouve un bloc de base ou une construction du langage.

Le but de ce *ln-level* est de savoir à quel niveau d'emboitement de boucle on s'intéresse et ainsi de n'appliquer certains traitements (la multiplication par exemple) qu'aux éléments de la représentation incrémentale du WCET dont le *ln-level* est inférieur ou égal au *ln-level* courant.

Adaptation des opérateurs

Les représentations incrémentales des WCET nécessitent la définition d'opérateurs adaptés à leur manipulation. Les opérateurs $\overset{L}{\oplus}$ et $\overset{L}{\otimes}$ remplacent respectivement les opérateurs $+$ et $\times$ du jeu de formules 4.1. Les opérandes de ces opérateurs sont des ensembles de couples $< WCET, \textit{ln-level} >$.

L'opérateur $\overset{L}{\oplus}$ est l'opérateur d'addition de WCET. Deux WCET sont additionnés par $\overset{L}{\oplus}$ en réalisant l'union des deux ensembles les représentant puis en regroupant les éléments de l'ensemble dont les *ln-levels* sont égaux, enfin en regroupant les éléments dont le *ln-level* est plus petit ou égal (au sens de $\succeq$) à L.

L'opérateur $\overset{L}{\otimes}$ est l'opérateur de multiplication d'un WCET par un scalaire. Il applique un coefficient multiplicateur à tous les éléments de l'ensemble représentant un WCET dont le *ln-level* est plus petit ou égal à L et les regroupe.

☞ Exemple :

$$\{<\alpha,[\,]>,<\beta,[0]>\} \overset{[0]}{\oplus} \{<\delta,[\,]>,<\gamma,[0,0,2]>\} = \{<(\alpha+\delta),[\,]>,<(\beta+\gamma),[0]>\}$$
$$M \overset{[0]}{\otimes} \{<\alpha,[\,]>,<\beta,[0]>,<\gamma,[0,0,2]>\} = \{<\alpha,[\,]>,<M\times(\beta+\gamma),[0]>\}$$

Ces opérateurs nous permettent de définir le schéma temporel présenté en table 4.2.

Structure S	
$S_1;\ldots;S_n$	$WCET(S,L) = WCET(S_1,L) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_n,L)$ $WCET^{\neg p}(S,L) = WCET(S_1,L) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L) \overset{L}{\oplus} WCET^{\neg p}(S_n,L)$ $WCET^{p}(S,L) = WCET(S_1,L) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L) \overset{L}{\oplus} WCET^{p}(S_n,L)$
if (tst) then S_1 else S_2	$WCET(S,L) = max(\ WCET^{\neg p}(Test,L) \overset{L}{\oplus} WCET(S_1,L)\ ,$ $WCET^{p}(Test,L) \overset{L}{\oplus} WCET(S_2,L)\ )$
for (;tst;inc) S_1	$WCET(S,L) = maxiter \overset{L}{\otimes} \left(WCET^{p}(Test,L') \overset{L}{\oplus} WCET(S_1,L')\right)$ $\overset{L}{\oplus} WCET^{\neg p}(Test,L')$ L' est le *ln-level* de la boucle
basic block BB_x	$WCET(BB_\alpha,L) = WCET_BB_\alpha^{?}$ $WCET^{\neg p}(BB,L) = WCET_BB_\alpha^{\neg p}$ $WCET^{p}(BB,L) = WCET_BB_\alpha^{p}$

$WCET_BB_\alpha^{?}$ est le WCET de BB_α, un bloc de base qui n'a qu'une possibilité de branchement en fin de bloc.

M est la borne supérieure sur le nombre d'itérations de la boucle (supposé constant ici).

TAB. 4.2 – *Schéma temporel adapté aux représentations incrémentales du WCET*

Lors du calcul du WCET des boucles et des structures conditionnelles deux types d'exécution (p et $\neg p$) du bloc de base *test* doivent être considérés. Le sous-arbre *test* des structures

boucle et conditionnelle peut être soit un simple bloc de base, soit une séquence de sous-arbres. Dans ce deuxième cas, c'est le bloc de base le plus à droite qui réalise le test (*cf.* figure 4.25). C'est pourquoi seules les formules de calcul de WCET des blocs de base et des séquences sont différenciées en WCET^p et $\mathrm{WCET}^{\neg p}$.

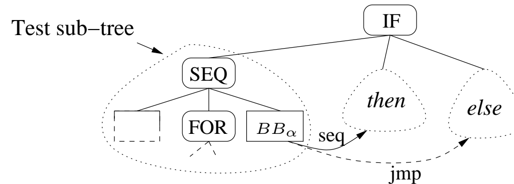


FIG. 4.25 – *Détails du sous-arbre test d'une structure conditionnelle*

Les formules de calcul du WCET des structures de boucles et conditionnelles de la table 4.2 dépendent des hypothèses faites sur leurs schémas de compilation.

4.7 Récapitulatif

Nous avons présenté dans ce chapitre plusieurs propositions concernant le niveau bas de l'analyse statique. La première est l'utilisation du niveau d'emboitement des boucles comme information représentant le contexte d'exécution des instructions. Le comportement des instructions vis-à-vis du cache d'instruction et du mécanisme de prédiction de branchement peut être estimé pour un niveau d'emboitement de boucle donné.

Nous avons adapté deux techniques de prise en compte d'éléments d'architecture (le cache d'instruction, et le pipeline) pour qu'elles utilisent les niveaux d'emboitement des boucles. Nous avons proposé une technique permettant d'estimer le comportement des instructions de branchement vis-à-vis du mécanisme de prédiction des branchements, qui elle aussi utilise les niveaux d'emboitement des boucles.

Les estimations du WCET des blocs de base obtenus sont maintenant des représentations incrémentales, et nous avons adapté le schéma temporel pour prendre en compte ces représentations incrémentales.

Un exemple complet ainsi que l'adaptation de ce nouveau schéma temporel pour prendre en compte les annotations symboliques présentées au chapitre 3, sera présenté dans le prochain chapitre.

L'évaluation quantitative des performances de ces techniques est présenté au chapitre 6.

Chapitre 5

HEPTANE : un outil pour l'analyse statique de WCET

Les travaux théoriques présentés aux chapitres précédents ont fait l'objet d'une implémentation ayant pour but la validation et l'estimation des performances des méthodes proposées. Nous avons ainsi développé un analyseur statique de WCET complet dans lequel s'insèrent les mécanismes de réduction de pessimisme des chapitres 3 et 4.

5.1 L'analyseur HEPTANE

Notre prototype d'analyseur, nommé HEPTANE pour "Hades Embedded Processor Timing ANalyzEr" [ACC+99], a pour fonction d'estimer le temps d'exécution au pire cas, sur une architecture Intel Pentium, des programmes écrits en C et respectant les contraintes sur le langage introduites au paragraphe 2.1.1.2. Ces contraintes interdisent l'usage : (i) des appels de fonction par pointeur, (ii) de la récursivité, (iii) des commandes permettant des branchements non structurés (par exemple les commandes GOTO, EXIT et CONTINUE du C), et (iv) la sortie multiple des fonctions. De plus, ces contraintes imposent que toutes les boucles soient annotées par l'utilisateur pour indiquer leur nombre maximum d'itérations en utilisant le système d'annotations présenté au chapitre 3.

5.1.1 Vue d'ensemble de l'analyseur

Une vue générale de l'analyseur est proposée à la figure 5.1. Pour plus de clarté, nous décomposons ici l'analyseur en quatre sous-parties dont nous détaillons ci-après les fonctions et l'organisation.

Le bloc ① est chargé de la préparation du code à analyser, il reçoit en entrée le code source et le nom de la fonction qui sera la racine de l'arbre (celle dont on veut calculer le WCET), et

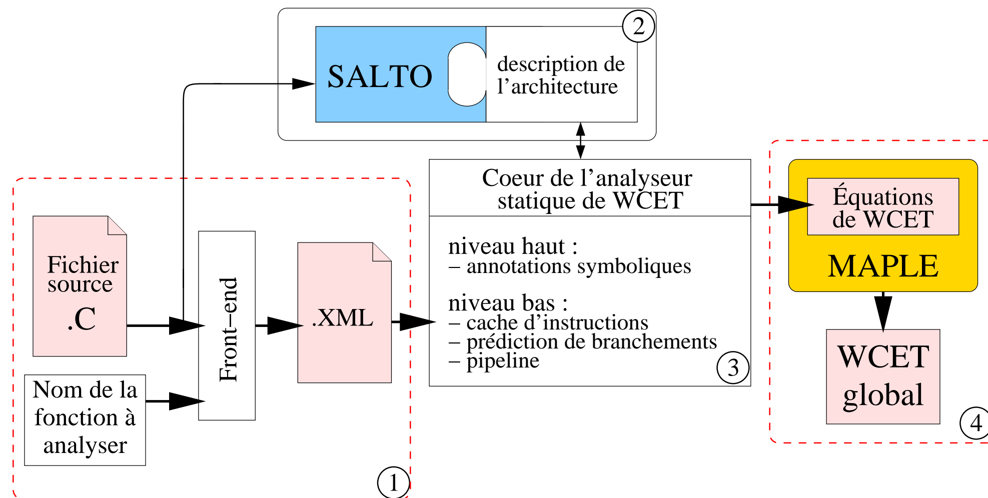


FIG. 5.1 – *Vue d'ensemble de l'analyseur* HEPTANE

fournit en sortie une représentation du code source pour l'analyse de WCET. Le bloc ②, basé sur l'outil de manipulation de code assembleur SALTO, réalise la gestion du code assembleur. Le bloc ③ est le cœur de notre analyseur, il est constitué de trois modules de prise en compte de l'architecture (niveau bas), et génére le système d'équation de WCET qui sera évalué par le bloc ④. Ce dernier bloc, qui inclut un outil d'évaluation symbolique, calcule le WCET du programme à partir des équations de WCET.

Dans sa configuration actuelle, HEPTANE estime le WCET de programme s'exécutant sur une architecture Intel Pentium. Les modules du bloc ③ sont conçus pour prendre en compte :

- le cache d'instructions constitué de 128 ensembles de 2 voies de 32 octets chacune (soit 8Ko), et dont la politique de remplacement est LRU.
- le mécanisme de prédiction de branchement utilisant un BTB de 64 ensembles de 4 voies avec des historiques à 2 bits.
- un seul des deux pipelines entiers du processeur.

HEPTANE ne prend pas en compte l'impact du cache de données, ni celui du caractère superscalaire du processeur, sur le WCET des programmes.

Le tableau 5.1 présente le volume de code (hors commentaires) ainsi que les langages utilisés pour l'implantation d'HEPTANE. Seul le code spécialement développé pour cette thèse y apparait (*e.g.* le volume de code associé au bloc ② n'est pas le code de SALTO mais le code d'utilisation de Salto).

5.1.2 Génération des représentations logiques du programme - bloc ①

Cette première partie de l'analyseur a pour fonction de préparer les structures de données nécessaires à l'analyse. Une analyse syntaxique du fichier source, du C ansi dans notre cas,

Bloc	Langage de programmation	Nombre de lignes
bloc ①	Objective CAML	2364
	C	1986
	C++	847
bloc ②	Objective CAML	294
	C++	1100
bloc ③	Objective CAML	5289
	C++	30
module btb	Objective CAML	569
module icache	Objective CAML	527
module pipeline	C++	697
bloc ④	Maple	95
	Objective CAML	552

TAB. 5.1 – *Volume de code et langages utilisés pour l'implantation d'*HEPTANE

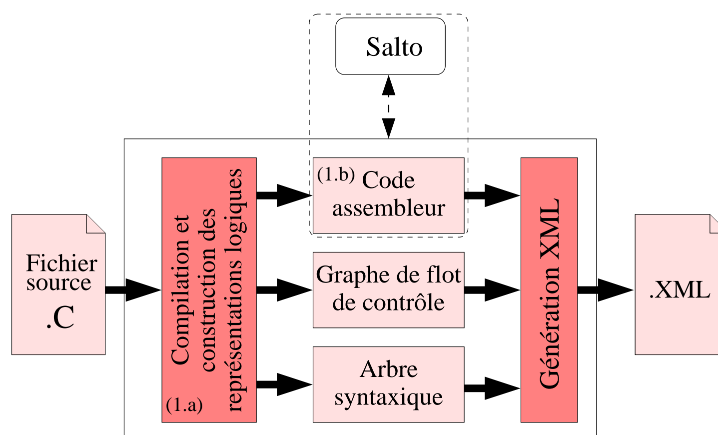


FIG. 5.2 – *Génération du fichier XML représentant le programme analysé*

permet de vérifier si les contraintes d'analysabilité du langage sont respectées (repère 1.a dans la figure 5.2). Si un des critères d'analysabilité n'est pas respecté (présence de constructions syntaxiques interdites, absence d'annotations, etc.), le programme est refusé et la cause du refus est localisée.

Le code source doit être modifié pour être analysé. Ces modifications ont pour but de permettre l'établissement d'une correspondance entre la structure syntaxique du programme et le code assembleur issu de sa compilation. La figure 5.3 illustre les modifications apportées avant la compilation. Le fichier source (a) est modifié pour y ajouter des *étiquettes* (b) permettant d'identifier les différents éléments des constructions syntaxiques. Parallèlement à cette modification du code source, l'arbre syntaxique (c) de chacune des fonctions est construit. La correspondance entre les arbres syntaxiques (plus particulièrement leurs feuilles) et le code assembleur (d) est assurée par les étiquettes.

Le code est compilé (gcc 2.8.1) et le code assembleur résultant est découpé en blocs de base par l'outil externe SALTO présenté au paragraphe suivant (repère 1.b dans la figure 5.2). Le code assembleur généré comporte certaines informations supplémentaires : les adresses et les tailles de chacune des instructions, mais aussi les étiquettes ajoutées précédemment.

Une vérification structurelle est effectuée entre le graphe de flot de contrôle et l'arbre syntaxique, elle consiste à vérifier que tous les chemins d'exécution représentés par le graphe de flot de contrôle sont aussi présents dans l'arbre syntaxique, et réciproquement.

On dispose aussi du graphe d'appel des fonctions ce qui nous permet pour une fonction racine donnée (un paramètre d'entrée du bloc) d'élaguer la forêt d'arbre fournie par l'outil de compilation et de ne conserver ainsi que les arbres utiles. Les feuilles des arbres syntaxiques restants (*i.e.* les blocs de base) leur sont ajoutées. Puis l'arbre syntaxique de la fonction racine est développé par inclusion des arbres syntaxiques des fonctions appelées (*cf.* § 2.1.2.4).

On dispose alors des représentations du programme en arbre syntaxique et en graphe de flot de contrôle telles que décrites au paragraphe 2.1.2. Les deux représentations du programme sont alors stockées selon un format XML (voir annexe A.1) dans un fichier de description de programme qui rassemble les données d'entrées pour l'analyse statique à proprement parler (voir annexe A.2 pour un exemple).

5.1.3 Gestion du code assembleur - bloc ②

La gestion du code assembleur est réalisée par un outil spécialisé : SALTO (voir bloc ② sur la figure 5.1). SALTO[1] [BRS96] propose un environnement de manipulation de programmes en langage assembleur. Il offre de plus une représentation abstraite des ressources de l'architecture matérielle ce qui présente deux avantages :

- le même algorithme peut être appliqué à des programmes écrits pour différentes architectures avec peu de modifications,

1. *http://www.irisa.fr/caps/projects/Salto*

(a) Code source

```
unsigned ReverseBits ( unsigned index ,
                       unsigned NumBits ) {
  unsigned i , rev ;
  i = rev = 0 ;
  for(  ; i < NumBits ; i ++ ) {
    rev *= 2 ;
    rev |= ( index & 1 ) ;
    index /= 2 ;
  }
  return rev ;
}
```

(b) Code source avec étiquettes

```
unsigned ReverseBits ( unsigned index ,
                       unsigned NumBits ) {
  unsigned i , rev ;

  i = rev = 0 ; asm("__halt28:");
  for(  ; i < NumBits ; i ++ ) {
    asm("__halt29:");
    rev *= 2 ;
    rev |= ( index & 1 ) ;
    index /= 2 ;
    asm("__halt30:");
  } asm("__halt31:");
  asm("__halt32:");
  return rev ; }
asm("__halt27:");
```

(c) Représentation textuelle de l'arbre syntaxique généré lors de la compilation

```
_ReverseBits =
  Seq [
    Code( _ReverseBits: , __halt28: );
    For( [10,i],
      Vide,
      Code( __halt28: , __halt29: ),
      Code( __halt30: , __halt31: ),
      Code( __halt29: , __halt30: )
    );
    Code( __halt32: , __halt27: )
  ]
```

(d) Code assembleur avec étiquettes

```
.globl _ReverseBits
_ReverseBits:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $0,-8(%ebp)
        movl $0,-4(%ebp)
__halt28:
L24:
        movl -4(%ebp),%eax
        cmpl %eax,12(%ebp)
        ja L27
        jmp L25
        .align 2,0x90
L27:
__halt29:
        movl -8(%ebp),%eax
        movl %eax,%edx
        movl %edx,%eax
        addl %edx,%eax
        movl %eax,-8(%ebp)
        movl 8(%ebp),%eax
        andl $1,%eax
        orl %eax,-8(%ebp)
        movl 8(%ebp),%edx
        movl %edx,%eax
        shrl $1,%eax
        movl %eax,8(%ebp)
__halt30:
L26:
        incl -4(%ebp)
        jmp L24
        .align 2,0x90
L25:
__halt31:
__halt32:
        movl -8(%ebp),%edx
        movl %edx,%eax
        jmp L23
        .align 2,0x90
L23:
        movl %ebp,%esp
        popl %ebp
        ret
__halt27:
```


FIG. 5.3 – *Utilisation des étiquettes pour établir la correspondance entre le code source et le*

– la manipulation du code assembleur et la gestion des ressources matérielles sont grandement simplifiées.

Le code assembleur obtenu par compilation du code source à analyser (voir paragraphe précédent) est traiter par SALTO qui va effectuer :

– l'analyse lexicale et syntaxique du code,
– l'éclatement du code en blocs de base,
– la génération de la représentation par graphe de flot de contrôle,
– le calcul des dépendances entre instructions (dues aux aléas dans le pipeline).

Pour cela, SALTO utilise une description de la machine cible, c'est-à-dire un fichier qui décrit le jeu d'instructions et l'ensemble des ressources matérielles de l'architecture cible, ainsi que la syntaxe de l'assembleur analysé. Le remplacement de ce fichier permet le reciblage de SALTO vers un microprocesseur différent. Ici, nous utilisons le fichier de description de l'architecture Intel Pentium et de l'assembleur Intel fourni avec SALTO.

5.1.4 Cœur de l'analyseur - bloc ③

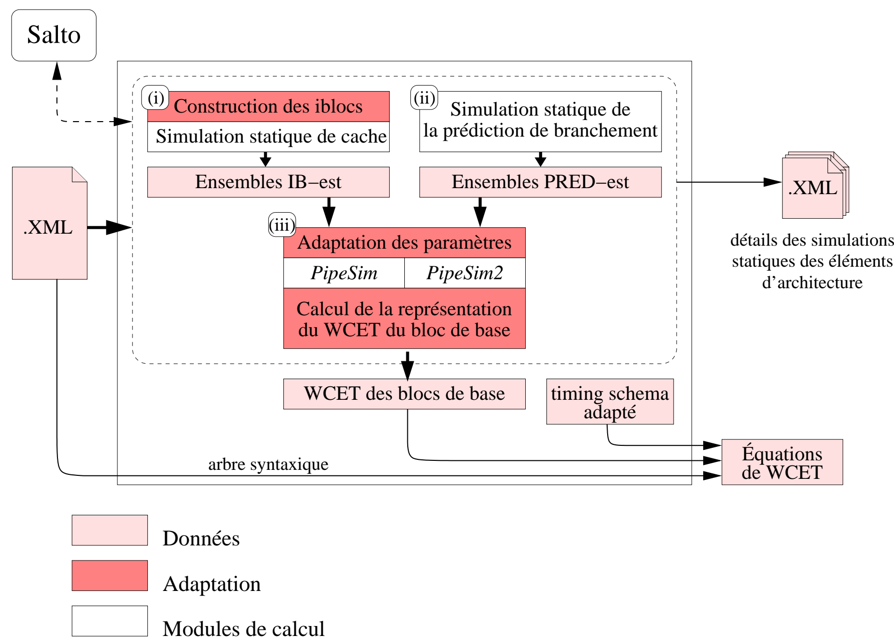


FIG. 5.4 – *Le cœur de l'analyseur*

Les données mises en forme en ① sont passées en entrée du bloc ③ qui constitue le cœur de notre analyseur. De haut en bas sur la figure, on distingue :

– Les trois modules de prise en compte de l'architecture : (i) module de simulation statique de cache d'instruction, (ii) module de simulation statique de prédiction de branchement et (iii) module de simulation d'occupation du pipeline.

– Les résultats du module de simulation d'occupation du pipeline (les WCET des blocs de base).
– Le jeu de formules pour la transformation de l'arbre syntaxique et des WCET des blocs de base en un système d'équations (*i.e.* les données en sortie du bloc).

Le fonctionnement interne des trois modules présentés sera décrit plus en détail au paragraphe 5.3.1. La méthode de génération du système d'équations pour le calcul du WCET du programme à partir des WCET des blocs de base fait l'objet du paragraphe 5.2.

Sans entrer plus avant dans les détails de ces modules, on peut déjà remarquer sur la figure 5.4 plusieurs "couches" d'adaptation des données en entrée et en sortie des modules.

La première, libellée "construction des iblocs" sur la figure, a pour but de former des iblocs (comme définis au 4.3.1.2) à partir de la description d'un bloc de base. Les informations de taille et d'adresse liées aux instructions sont utilisées pour les regrouper et/ou les fragmenter en un ou plusieurs iblocs. Ces iblocs sont ensuite traités par le module de simulation statique de cache d'instruction (*cf.* § 4.3).

La deuxième adaptation des données est réalisée en entrée du module de prise en compte du pipeline. Les données d'entrées de ce module sont les informations fournies en sortie des modules du cache d'instruction et de la prédiction de branchement. Les résultats concernant les *iblocs* (les ensembles de couples *IB-est*) sont transformés en informations sur les *instructions* selon la règle suivante (déjà énoncée au paragraphe 4.5.1.2).

L'exécution d'une instruction du bloc de base BB_α cause un *miss* dans le cache si la simulation statique de cache d'instructions a associé à l'un des iblocs $IB_{\alpha,x}$, dans lequel elle est en première position, un *miss-level* tel que $L \succeq$ *miss-level*$_{\alpha,x}$, sinon c'est un *hit*.

Enfin, les résultats des fonctions $PipeSim$ et $PipeSim2$ pour chacun des *ln-levels* concernés sont traités pour former la représentation incrémentale du WCET de blocs de base qui intègre les résultats des trois types d'analyse (*cf.* § 4.5.1.4).

Une fois calculés les WCET de tous les blocs de base (les feuilles de l'arbre syntaxique), les formules du schéma temporel (*timing schema*) sont appliquées à l'arbre syntaxique pour générer le système d'équations permettant le calcul du WCET global ci dessous.

5.1.5 Calcul du WCET - bloc ④

Le système d'équations résultant du bloc ③ de l'analyseur est un "programme" Maple. Il est constitué d'une bibliothèque de fonctions dédiées à la manipulation des structures de données (pile d'annotation et représentation incrémentale des WCET) d'une part (*cf.* annexe B), et des équations de calcul des WCET pour chaque nœud et chaque feuille de l'arbre syntaxique d'autre part.

Le résultat de la résolution du système d´équations symboliques par Maple peut être soit

un résultat numérique (le WCET exprimé en nombre de cycles processeur), soit une expression symbolique si un ou plusieurs symboles restent non évalués. Ce deuxième cas de figure permet d'obtenir des valeurs du WCET paramétrées, par exemple, pour prendre en compte l'impact du changement de la taille d'une donnée d'entrée sur le WCET sans avoir à tout recalculer.

5.2 Estimation du WCET d'un programme

Nous présentons, au paragraphe 5.2.1, l'adaptation du schéma temporel défini à la fin du chapitre précédent. Ce nouveau schéma temporel nous permet d'utiliser à la fois les représentations incrémentales de WCET (*cf.* § 4.5.1.4) et les annotations symboliques des boucles (*cf.* § 3) pour le calcul du WCET global. Le paragraphe 5.2.2 propose un exemple récapitulatif des différents mécanismes mis en œuvre pour l'analyse statique de WCET.

5.2.1 Adaptation du schéma temporel du chapitre 3 aux annotations symboliques

La prise en compte des annotations symboliques nécessite une nouvelle adaptation du schéma temporel. Cette modification, décrite au paragraphe 3.4.1, consiste à :

- ajouter la pile d'annotation comme paramètre des formules de calcul du WCET,
- exprimer le WCET des boucles par des sommes finies.

Dans le paragraphe précédent, l'opérateur $\overset{L}{\otimes}$ n'applique le coefficient multiplicateur qu'à un sous-ensemble des éléments de la représentation de WCET.

Dans certains cas (voir § 3.4.1), cette multiplication doit être remplacée par une sommation. Nous définissons donc l'opérateur de somme finie sélectif :

$$L \bigodot_{\lambda = 0}^{bornesup}$$

Cet opérateur regroupe les éléments dont les *ln-levels* sont inférieurs ou égaux à L, et applique l'opérateur de somme finie au WCET résultant. Les éléments de l'ensemble dont le *ln-level* n'est pas inférieur ou égal ne sont pas affectés et on les retrouve inchangés dans la représentation incrémentale résultante. Le calcul d'une somme finie étant plus complexe qu'une simple multiplication, l'opérateur $\overset{L}{\otimes}$ ne doit être remplacé par ce nouvel opérateur que pour les équations qui le requièrent. C'est pourquoi le schéma temporel de la table 5.2 comporte deux formules pour la traduction des boucles.

☞ Exemple :

$$[0] \bigodot_{\lambda = 0}^{M-1} \{< \alpha, [\,] >, < \beta, [0] >, < \gamma, [0, 0, 2] >\} = \{< \alpha, [\,] >, < \sum_{\lambda=0}^{M-1} (\beta + \gamma), [0] >\}$$

Le nouveau schéma temporel pour un calcul de WCET utilisant à la fois la représentation incrémentale du WCET et les annotations symboliques est présenté au tableau 5.2.

Structure S	
$S_1; \ldots ; S_n$	$WCET(S,L,P) = WCET(S_1,L,P) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_n,L,P)$ $WCET^{\neg p}(S,L,P) = WCET(S_1,L,P) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L,P) \overset{L}{\oplus} WCET^{\neg p}(S_n,L,P)$ $WCET^{p}(S,L,P) = WCET(S_1,L,P) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L,P) \overset{L}{\oplus} WCET^{p}(S_n,L,P)$
if (tst) then S_1 else S_2	$WCET(S,L,P) = max(\ WCET^{\neg p}(Test,L,P) \overset{L}{\oplus} WCET(S_1,L,P)\ ,$ $WCET^{p}(Test,L,P) \overset{L}{\oplus} WCET(S_2,L,P)\ )$
for (;tst;inc) S_1	$WCET(S,L,P) = \ L \bigodot_{\lambda=0}^{maxiter-1} \big(WCET^{p}(Test,L',P') \overset{L}{\oplus} WCET(S_1,L',P')\big) \overset{L}{\oplus} WCET^{\neg p}(Test,L',P'')$ avec $P' = (P \frown M, C^1, \ldots, C^n)$ avec $P'' = (P \frown M, C^1[M/\lambda], \ldots, C^n[M/\lambda])$ et avec L' le *ln-level* de la boucle
for (;tst;inc) S_1	$WCET(S,L,P) = maxiter \overset{L}{\otimes} \big(WCET^{p}(Test,L',P') \overset{L}{\oplus} WCET(S_1,L',P')$ $\overset{L}{\oplus} WCET^{\neg p}(Test,L',P')$ avec $P' = (P \frown ?)$ (on empile un élément nul qui ne sera jamais accédé) et avec L' le *ln-level* des sous-arbres du nœud boucle $(L \succ L' \wedge (\nexists K \mid L \succ K \succ L'))$
basic block BB_x	$WCET(BB,L,P) = WCET_BB^{?}$ $WCET^{\neg p}(BB,L,P) = WCET_BB^{\neg p}$ $WCET^{p}(BB,L,P) = WCET_BB^{p}$

TAB. 5.2 – *Schéma temporel adapté aux annotations symboliques*

5.2.2 Exemple récapitulatif

La figure 5.5 présente l'arbre syntaxique d'un programme qui va nous servir d'exemple pour illustrer la coopération des modules de prise en compte de l'architecture et le calcul du WCET global.

Détailler les traitements effectués par les trois modules sur chaque bloc de base serait trop fastidieux, on se limite donc à observer en détail les blocs de base BB_5 et BB_7. L'extrait de la description XML du programme qui contient les données concernant ces deux blocs de base est présenté en annexe A.3.

Le module cache d'instruction

Les données d'entrée de ce module sont celles présentées en annexe A.3. D'autres informations sont internes au module, comme par exemple le nombre de lignes du cache, leur tailles, etc.

La première étape de notre simulation statique de cache est le calcul des iblocs constituant chaque bloc de base à partir des informations sur les instructions. La figure 5.6 présente ce calcul en supposant que le cache est constitué de lignes de 4 octets. Les instructions du bloc de base BB_5 sont réparties en 4 iblocs, et celles de BB_7 en 3 iblocs.

La deuxième étape est le calcul des *miss-levels* associés à ces iblocs. Supposons que les

```
void fct (int* tab) {

  int i , j , s = 0 ;

  for ( i = 0 ; i < 10 ; i ++ ) [ ArrSize , 2 * i ] {
    for ( j = 0 ; j < i * i ; j ++ ) [ nlast ( P , 1 ) , i ] {
      s += tab[i] ;
    }
  }
}
```

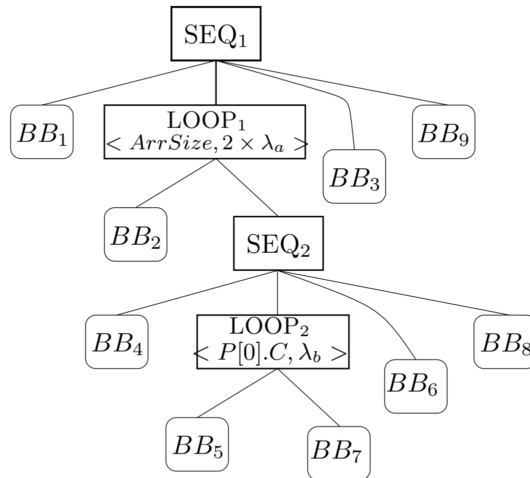


FIG. 5.5 – *Arbre syntaxique du programme d'exemple*

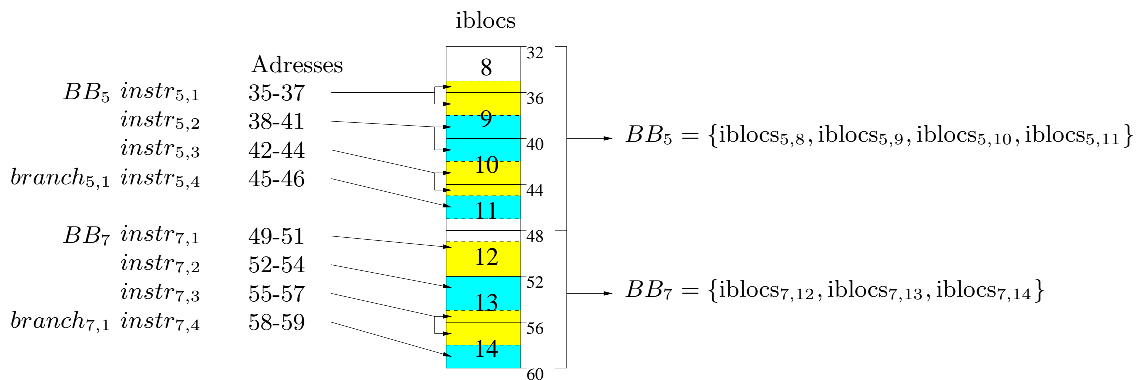


FIG. 5.6 – *Traduction des blocs de base en iblocs*

résultats de ces calculs concernant les deux blocs de base qui nous intéressent soit représentés par les ensembles *IB-est* suivants :

$$\begin{aligned} IB\text{-}est_5 &= \{<8, never>, <9, [\,]>, <10, [0]>, <11, [0]>\} \\ IB\text{-}est_7 &= \{<12, [0]>, <13, [0]>, <14, [0]>\} \end{aligned}$$

On y remarque que l'ibloc$_{5,8}$ est toujours présent dans le cache au moment de son exécution, ce qui s'explique par le fait que les fragments d'instructions de BB_5 qu'il contient sont chargés lors de l'exécution de l'ibloc$_{4,8}$ (BB_5 est le test de la boucle). L'ibloc$_{5,9}$, quant à lui, est chargé une première fois dans le cache et ne cause plus de *miss* ensuite.

Les *IB-est* sont les données d'entrée pour le module d'exécution pipelinée.

Le module prédiction de branchement

Les données d'entrée de ce module sont celles présentées en annexe A.3. Dans ce module, les seules instructions considérées sont les instructions de branchement. Les blocs de base BB_5 et BB_7 en comportent une chacun. Le branchement de BB_7 (noté $Branche_{7,1}$) est inconditionnel, et celui de BB_5 (noté $Branche_{5,1}$) est un branchement conditionnel (c'est le test de la boucle). Les résultats de la simulation statique de prédiction de branchement sont les ensembles *Préd-est* des blocs de base. Les ensembles *Préd-est*$_7$ et *Préd-est*$_5$ comportent chacun un élément, car ces deux blocs de base n'ont qu'un branchement chacun. L'élément de *Préd-est*$_7$ est un doublet, alors que celui de *Préd-est*$_5$ est un triplet. Ce dernier représente pour le branchement $Branche_{5,1}$ les *error-levels* des deux possibilités du branchement conditionnel (p et $\neg p$).

$$\begin{aligned} Préd\text{-}est_5 &= \{<1, never, [0.0]>\} \\ Préd\text{-}est_7 &= \{<1, [0.0]>\} \end{aligned}$$

C'est le cas si on suppose que ces deux branchements sont en conflit pour l'utilisation de la même entrée du BTB. Aucun des deux n'est alors présent dans le BTB au moment de sa prédiction, ils sont donc tous les deux prédits par défaut (*i.e.* D-prédits). La prédiction par défaut étant «non-pris», le branchement $Branche_{7,1}$, qui est inconditionnellement pris, est toujours mal prédit (d'où le *error-level* = [0.0]). En ce qui concerne $Branche_{5,1}$, sa possibilité de branchement $\neg p$ («non-pris») est toujours bien prédite (*error-level* = *never*), alors que l'autre, p («pris»), ne l'est jamais (*error-level* = [0.0]).

Les ensembles *Préd-est* sont des données d'entrée pour le module d'exécution pipelinée.

Le module d'exécution pipelinée

Ce module est chargé de calculer le pire temps d'exécution de chacun des blocs de base. Le résultat de la fonction *PipeSim* dépend du *ln-level* considéré. C'est pourquoi, dans un

premier temps, on calcule la liste des *ln-levels* "intéressants" pour chaque bloc de base (liste Λ définie au paragraphe 4.5.1.4). Pour les deux blocs de base de notre exemple, on a :

$$\Lambda_5 = \{[\,], [0], [0.0]\}$$
$$\Lambda_7 = \{[0], [0.0]\}$$

Puis, pour chaque bloc de base, on calcule les paramètres de la fonction *Pipesim* pour les *ln-levels* pour lesquels c'est nécessaire (*i.e.* ceux contenus dans les listes Λ associées aux blocs de base).

BB_5 (p)	BB_5 $(\neg p)$	BB_7
$INSTmiss_{5,[\,]} = \{1, 2, 3\}$	$INSTmiss_{5,[\,]} = \{1, 2, 3\}$	
$INSTmiss_{5,[0]} = \{2, 3\}$	$INSTmiss_{5,[0]} = \{2, 3\}$	$INSTmiss_{7,[0]} = \{1, 2, 3\}$
$INSTmiss_{5,[0.0]} = \{\ \}$	$INSTmiss_{5,[0.0]} = \{\ \}$	$INSTmiss_{7,[0.0]} = \{\ \}$
$BRANCHerr_{5,[\,]} = \{1\}$	$BRANCHerr_{5,[\,]} = \{\ \}$	
$BRANCHerr_{5,[0]} = \{1\}$	$BRANCHerr_{5,[0]} = \{\ \}$	$BRANCHerr_{7,[0]} = \{1\}$
$BRANCHerr_{5,[0.0]} = \{1\}$	$BRANCHerr_{5,[0.0]} = \{\ \}$	$BRANCHerr_{7,[0.0]} = \{1\}$

Ces paramètres sont ensuite utilisés pour remplir les tables de réservation (*cf.* § 4.5.1.1) permettant d'estimer le pire temps d'exécution de chaque bloc de base BB_α pour chaque *ln-level* de la liste Λ_α. Les différentes estimations de WCET d'un même bloc de base sont ensuite écrites sous forme de WCET incrémental tel que décrit au paragraphe 4.5.1.4.

Pour exemple, les WCET incrémentaux de tous les blocs de base de l'exemple de la figure 5.5 sont présentés dans le tableau suivant. Notons que les blocs de base BB_2 et BB_4 ont deux représentations de WCET différentes chacun. Ceci est dû à leur rôle de test de fin de boucle, leur dernière instruction est un branchement conditionnel qui peut être «pris» (p) ou «non pris» $(\neg p)$.

Bloc de base	branchement	WCET
BB_1		$[< 104, [] >]$
BB_2	p	$[< 34, [0] >, < 4, [] >]$
BB_2	$\neg p$	$[< 30, [0] >, < 4, [] >]$
BB_3		$[< 15, [] >]$
BB_4		$[< 40, [0] >, < 8, [] >]$
BB_5	p	$[< 21, [0.0] >, < 16, [0] >, < 8, [] >]$
BB_5	$\neg p$	$[< 16, [0.0] >, < 16, [0] >, < 8, [] >]$
BB_6		$[< 11, [0] >, < 4, [] >]$
BB_7		$[< 17, [0.0] >, < 24, [0] >]$
BB_8		$[< 58, [0] >]$
BB_9		$[< 79, [] >]$

À partir de ces données, de l'arbre syntaxique de la figure 5.5 et des formules de la table 5.2, le système d'équations de la table 5.3 est construit. Enfin, ce système d'équations est traduit en code compréhensible par l'outil Maple qui est chargé de sa résolution.

$$
\begin{aligned}
WCET(SEQ_1,[\,],P) &= WCET(BB_1,[\,],P) \overset{[\,]}{\oplus} WCET(LOOP_1,[\,],P) \overset{[\,]}{\oplus} WCET(BB_3,[\,],P) \overset{[\,]}{\oplus} WCET(BB_9,[\,],P) \\
WCET(LOOP_1,[\,],P) &= [\,]\sum_{\lambda_a=0}^{ArrSize-1} \big(WCET^{p}(BB_2,[0],Push(P,2\times\lambda_a)) \overset{[\,]}{\oplus} WCET(SEQ_2,[0],Push(P,2\times\lambda_a))\big) \overset{[\,]}{\oplus} WCET^{\neg p}(BB_2,[0],Push(P,2\times ArrSize)) \\
WCET(SEQ_2,[0],P) &= WCET(BB_4,[0],P) \overset{[0]}{\oplus} WCET(LOOP_2,[0],P) \overset{[0]}{\oplus} WCET(BB_6,[0],P) \overset{[0]}{\oplus} WCET(BB_8,[0],P) \\
WCET(LOOP_2,[0],P) &= (P[0].C) \overset{[0]}{\otimes} \big(WCET^{p}(BB_5,[0.0],Push(P,x)) \overset{[0]}{\oplus} WCET(BB_7,[0.0],Push(P,x))\big) \overset{[0]}{\oplus} WCET^{\neg p}(BB_5,[0.0],Push(P,x)) \\
WCET(BB_1,*,*) &= \{<104,[]>\} \\
WCET^{p}(BB_2,*,*) &= \{<35,[0]>,<4,[]>\} \\
WCET^{\neg p}(BB_2,*,*) &= \{<30,[0]>,<4,[]>\} \\
WCET(BB_3,*,*) &= \{<15,[]>\} \\
WCET(BB_4,*,*) &= \{<40,[0]>,<8,[]>\} \\
WCET^{p}(BB_5,*,*) &= \{<21,[0.0]>,<16,[0]>,<8,[]>\} \\
WCET^{\neg p}(BB_5,*,*) &= \{<16,[0.0]>,<16,[0]>,<8,[]>\} \\
WCET(BB_6,*,*) &= \{<11,[0]>,<4,[]>\} \\
WCET(BB_7,*,*) &= \{<17,[0.0]>,<24,[0]>\} \\
WCET(BB_8,*,*) &= \{<58,[0]>\} \\
WCET(BB_9,*,*) &= \{<79,[]>\}
\end{aligned}
$$

TAB. 5.3 – *Système d'équations de l'exemple de la figure 5.5*

5.3 Adaptabilité de l'analyseur

Le prototype d'analyseur statique de WCET *tree-based* HEPTANE estime le temps d'exécution au pire cas de programmes écrits en C, sur une architecture Intel Pentium P54C. Nous présentons maintenant la possibilité de recibler cet analyseur, c'est-à-dire son adaptabilité à de nouvelles architectures.

Nous allons dans un premier temps décrire la structure modulaire d'HEPTANE (voir § 5.3.1), puis nous envisagerons son adaptation à un nouveau langage source (§ 5.3.2), à un nouveau langage assembleur (§ 5.3.3), et à une nouvelle architecture (§ 5.3.4).

5.3.1 Modularité de l'analyseur

Comme on l'a vu précédemment, le cœur de l'analyseur comporte trois modules dédiés à la prise en compte de l'effet de trois éléments d'architecture.

Chacun des modules effectue un traitement des informations disponibles en entrée (graphe de flot de contrôle, arbre syntaxique, blocs de base, instructions, etc.) et fournit en sortie un résultat concernant le comportement des instructions vis à vis de l'élément d'architecture qu'il simule.

La coopération entre le cadre de l'analyseur et les modules de modélisation d'architecture d'une part, et entre les modules eux-mêmes d'autre part, est assurée par la spécification des formats de données d'entrée/sortie des modules que nous détaillons dans les paragraphes suivants.

L'analyse de haut niveau utilise les WCET des blocs de base fournis par les modules de prise en compte de l'architecture, l'arbre syntaxique et le schéma temporel, pour calculer l'estimation du WCET global. Le niveau haut de l'analyse n'a pas à connaître le fonctionnement interne et les mécanismes mis en œuvre pour calculer les WCET des blocs de base. La seule contrainte imposée par le niveau haut de l'analyse sur les modules constituant le niveau bas est le respect des interfaces entre le cadre de l'analyse et les modules.

Nous présentons maintenant chacun des trois modules en détaillant ces formats de données en entrée et en sortie.

5.3.1.1 Le module de simulation statique de cache d'instruction

Les données en entrée de ce module sont :

- les informations sur les instructions (taille, adresse),
- le regroupement des instructions en blocs de base,
- les enchaînements possibles entre blocs de base (graphe de flot de contrôle)
- le *ln-level* de chaque bloc de base.

Toutes ces informations sont contenues dans le fichier XML de description du programme à analyser (résultat du bloc ①) dont le format est présenté en annexe A.1. Un exemple de description de programme en XML est présenté en annexe A.2. On y remarque les champs

`size` et `addr` pour les instructions, le champ `asm` qui regroupe les instructions d'un même bloc de base, enfin le champ `lnlev` indique le *ln-level* [⊥] du bloc de base.

Chaque bloc de base est ensuite converti en un ou plusieurs iblocs. Puis la technique de simulation statique décrite au paragraphe 4.3 est appliquée aux iblocs pour déterminer quand ils seront potentiellement absents du cache d'instructions.

Enfin, les résultats sont mis en forme (en XML). À chaque bloc de base est associé un ensemble de couples ¡ibloc,*miss-level*¿, c'est l'ensemble *IB-est* défini au paragraphe 4.3.4. Dans l'ensemble *IB-est*, le *miss-level* associé à un ibloc indique à quel *ln-level* il peut causer un *miss* dans le cache.

Voici un exemple du format des résultats :

```
1  <basic-block><CFG>2</CFG><BB>18</BB>
2    <ibloclst>
3      <ibloc>186</ibloc><misslev>0,1</misslev>
4      <ibloc>187</ibloc><misslev>0</misslev>
       ...
5    </ibloclst>
6  </basic-block>
   ...
```

Dans cet exemple, les informations associées à un bloc de base sont délimitées par les marqueurs <`basic-block`> (ligne 1) et <`/basic-block`> (ligne 6). Ces informations sont : l'identification du bloc de base (marqueurs `BB` et `CFG`), et la liste des iblocs associés à leur *miss-level* (marqueurs `ibloclst`).

5.3.1.2 Le module de simulation statique de cache de prédiction de branchement

Les données en entrée de ce module sont :

- les informations sur les instructions de branchements (adresse, rôle),
- le regroupement des instructions de branchements en blocs de base,
- les enchaînements possibles entre blocs de base (graphe de flot de contrôle)
- le *ln-level* de chaque bloc de base.

Comme le module précédent, le module de simulation statique de cache de prédiction de branchement prend en entrée la description en XML du programme à analyser. Cette description contient les données nécessaires aux traitements effectués par le module à l'exception du rôle des instructions. Cette information manquante est obtenue en interrogeant SALTO (bloc ②) pour savoir quelles instructions sont des instructions de branchement, et quels sont les branchements conditionnels. Le rôle des branchements conditionnels est ensuite déduit de leurs positions dans l'arbre syntaxique.

La méthode de simulation statique du BTB décrite au paragraphe 4.4 est utilisée pour calculer, pour chaque branchement de chaque bloc de base, le *ln-level* à partir duquel il sera potentiellement mal prédit. Le résultat de cette technique est l'ensemble *Préd-est* associé au bloc de base (*cf.* § 4.4.6).

Voici un exemple illustrant le format des résultats :

```
1 <basic-block><CFG>2</CFG><BB>18</BB>
2   <ct-instlst>
3     <ct-inst>
4       <addr>252</addr>
5       <pris>0</pris>
6       <nonpris>0,1</nonpris>
7     </ct-inst>
8   </ct-instlst>
9 </basic-block>
  ...
```

Cet extrait de résultat XML indique que le seul branchement du bloc de base identifié par <CFG>2</CFG><BB>18</BB> doit être considéré mal prédit quand il sera pris à partir du *ln-level* [0], et mal prédit quand il sera non pris à partir du *ln-level* [0.1].

5.3.1.3 Le module de simulation d'occupation du pipeline

Ce module prend en entrée les résultats des deux modules précédents. Il transforme les informations sur le comportement des instructions vis à vis du cache d'instruction et de la prédiction de branchement en estimation de temps d'exécution. Une autre source d'information est fortement sollicitée par ce module, il s'agit de la description d'architecture liée à l'outil Salto. Cette description d'architecture nous permet de connaître l'utilisation des différents étages et ressources du pipeline par chaque instruction, et ainsi de construire les tables de réservation.

La première partie du module réalise l'adaptation des données en entrée. Elle transforme les informations sur les iblocs en informations sur les instructions en utilisant la règle de conversion énoncée au paragraphe 4.5.1.1, page 94.

Puis, pour chaque bloc de base BB_α, les ensembles *IB-est*$_\alpha$ et *Préd-est*$_\alpha$ sont utilisés pour construire les ensembles $INSTmiss_{\alpha,L}$ et $BRANCHerr_{\alpha,L}$. L'ensemble d'instructions $INSTmiss_{\alpha,L}$ est le sous-ensemble des instructions du bloc de base BB_α qui causent un *miss* au *ln-level* L. Il est calculé pour tous les *ln-levels* L qui apparaissent dans *IB-est*$_\alpha$. De même, $BRANCHerr_{\alpha,L}$ est calculé pour tous les *ln-levels* L qui apparaissent dans *Préd-est*, et contient les instructions de branchement qui sont supposées mal prédites au *ln-level* L.

Les fonctions $PipeSim$ et $PipeSim2$, décrites aux paragraphes 4.5.1.3 et 4.5.2.1, four-

nissent respectivement les estimations de WCET, et les estimations de réduction de WCET dues aux chevauchements de l'exécution des blocs de base, et ce pour tous les *ln-levels* pour lesquels c'est nécessaire. Enfin, les résultats de ces deux fonctions sont modifiés et combinés pour former la représentation incrémentale du WCET du bloc de base considéré (voir § 4.5.1.4 et § 4.5.2.2 pour les détails du calcul de la représentation incrémentale du WCET). Cette représentation est un ensemble de couples $< WCET, \textit{ln-level} >$.

Le résultat fourni par ce module est donc un ensemble de représentations incrémentales de WCET. Les blocs de base ayant deux possibilités de branchement en fin de bloc (p et $\neg p$) se voient associer deux représentations de WCET.

Voici un exemple illustrant le format des résultats :

```
...
<basic-block><CFG>2</CFG><BB>23</BB>
  <pris>
    <wcetincrlst>
      <wcetincr><wcet>76</wcet><lnlev></lnlev>0,1,0</wcetincr>
      ...
      <wcetincr><wcet>4</wcet><lnlev></lnlev></wcetincr>
    </wcetincrlst>
  </pris>
  <nonpris>
    <wcetincrlst>
      <wcetincr><wcet>76</wcet><lnlev>0,1,0</lnlev></wcetincr>
      ...
      <wcetincr><wcet>0</wcet><lnlev></lnlev></wcetincr>
    </wcetincrlst>
  </nonpris>
</basic-block>
...
```

5.3.2 Modification du langage source

Comme on le voit sur la figure 5.4, les informations concernant le programme à analyser, qui sont passées au cœur de l'analyseur (bloc ④), sont regroupées et stockées sous forme d'un fichier au format XML. Cette représentation du programme contient à la fois les représentations logiques de haut niveau (graphe de flot de contrôle, arbre syntaxique, annotations, etc.) et des informations concernant les instructions (adresses, taille, blocs de base, etc.). La partie de l'analyseur qui génère cette représentation (bloc ②) peut être remplacée pour analyser un autre langage source (ADA par exemple) si l'analyse de ce nouveau langage permet de générer un fichier de représentation du programme qui respecte la spécification XML décrite

en annexe A.1. Ainsi, les langages compilés, "à blocs", avec du code bien structuré (C, Pascal, ADA) peuvent, en leur ajoutant quelque contraintes, convenir.

5.3.3 Modification de la syntaxe langage assembleur

À architecture matérielle identique, il se peut qu'en changeant le bloc ①, le compilateur du nouveau langage source génère une autre syntaxe de code assembleur. On peut par exemple retrouver de l'assembleur en syntaxe Intel dans les champs `code` de la représentation de programme alors qu'on avait une syntaxe AT&T auparavant. Ce code assembleur étant totalement géré par SALTO, la seule adaptation nécessaire est le remplacement du fichier de description d'architecture, chargé par SALTO à son initialisation, pour qu'il puisse de nouveau effectuer l'analyse syntaxique du code assembleur. La partie concernant uniquement l'utilisation des ressources matérielles n'a pas à être modifiée.

5.3.4 Reciblage de l'analyseur

La séparation entre les niveaux haut et bas de l'analyse statique d'une part, et entre les techniques de simulation statique des différents éléments d'architecture d'autre part, facilite grandement le reciblage de l'analyseur vers d'autres architectures.

L'adaptabilité de l'outil d'analyse bénéficie de son caractère modulaire. En effet, les éléments de l'analyseur qui dépendent fortement de l'architecture sont isolés du reste de l'analyseur et isolés les uns des autres. Il s'agit en particulier de la description d'architecture utilisée par l'outil de manipulation de code assembleur SALTO et des modules de simulation statique des éléments d'architecture.

Le reciblage de l'analyseur statique de WCET est illustré sur la figure 5.7.

Pour adapter HEPTANE à une nouvelle architecture, il faut :

- Fournir à SALTO la description de la nouvelle architecture cible et la syntaxe de l'assembleur associé. Il existe déjà plusieurs descriptions d'architecture, au format SALTO, pour des processeurs Alpha, MIPS, Sparc, Intel Pentium, etc. Il est bien sûr aussi possible d'écrire ses propres descriptions d'architecture dans la mesure où on dispose des informations nécessaires.
- Remplacer, si nécessaire (*i.e.* si le module déjà présent ne remplit pas la même fonction), les modules de simulation statique d'architecture par les modules correspondant à la nouvelle architecture. Rappelons que le module de simulation statique de cache d'instructions peut prendre en compte un cache d'instructions de n'importe quelle taille, à correspondance directe, associatifs par ensembles, ou totalement associatifs, du moment que le remplacement des lignes de cache est géré par LRU. Aucune contrainte particulière n'est imposée en ce qui concerne le fonctionnement interne et les couches d'adaptation des modules. Mais ils doivent respecter les interfaces définies précédemment.

Il est aussi possible de désactiver la prise en compte d'un des éléments d'architecture. Il suffit pour cela de remplacer son module par un module "pessimiste". Par exemple la

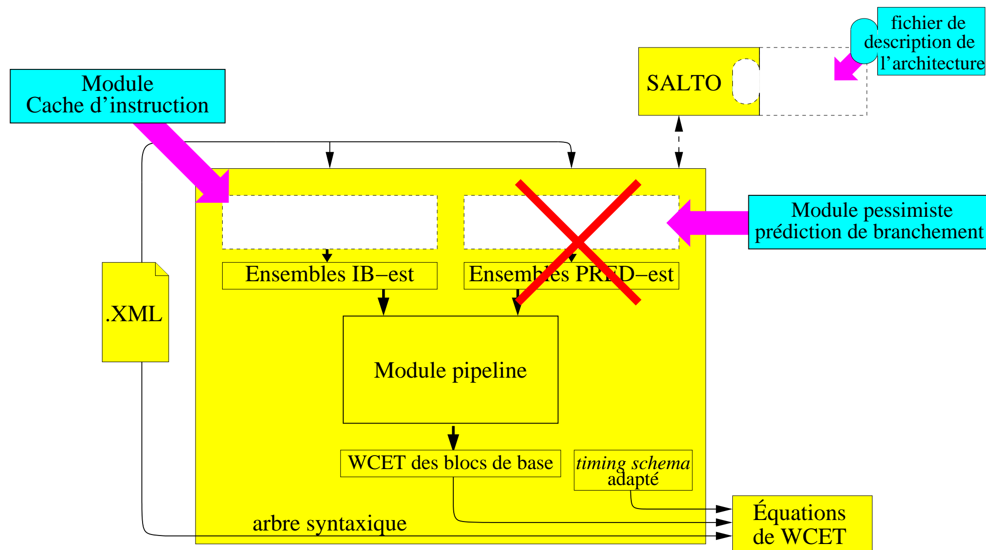


FIG. 5.7 – *Reciblage de l'analyseur par remplacement de modules et changement de description d'architecture*

mise en place du module de simulation de prédiction de branchement qui associe à tous les branchements le *ln-level* le plus bas possible (et estime donc qu'ils sont tous toujours mal prédit) permet d'adapter l'analyseur à une architecture sans prédiction de branchement.

Une autre possibilité pour adapter l'analyseur est l'ajout de nouveaux modules pour prendre en compte d'autres éléments d'architecture comme par exemple le cache de données. Pour pouvoir s'insérer dans le cadre d'analyseur existant, ces modules additionnels doivent prendre comme donnée d'entrée la description du programme analysé telle que définie en annexe A.1. De plus, ces modules se placent au dessus du module du pipeline et doivent fournir des informations sur le comportement des instructions et les exprimer en utilisant les *ln-levels*. Enfin, l'utilisation de ces résultats supplémentaires nécessite le remplacement du module de pipeline actuel par un module adapté pour recevoir des données supplémentaires en entrée et les utiliser.

Chapitre 6

Résultats d'analyse statique de WCET

Ce chapitre présente une partie des expérimentations réalisées avec l'analyseur statique de WCET Heptane. Ces expérimentations consistent en l'analyse statique de code source de programmes de diverses natures allant du plus simple (petits codes numériques, *cf.* § 6.2) au plus compliqué (code de système d'exploitation, *cf.* § 6.3). Ces résultats d'analyse ont permis d'évaluer les performances de l'analyseur modulaire à base d'arbre syntaxique d'une part, et celles des techniques de prise en compte de l'architecture matérielle d'autre part. Les expérimentations conduites nous ont également permis d'évaluer le coût de mise en place des annotations ainsi que le réalisme des contraintes imposées sur le code source analysé.

6.1 Méthode d'évaluation du prototype

L'évaluation des performances de l'analyseur statique de WCET est basée sur la comparaison des résultats d'analyse aux résultats obtenus par test et mesure du temps d'exécution des mêmes codes source sur l'architecture matérielle considérée. Nous présentons ici la technique de test et mesure employée pour obtenir les données de référence servant de base à l'estimation de performance.

6.1.1 L'exécution réelle, base de comparaison

Les WCET réels des programmes analysés sont obtenus par mesure de leurs temps d'exécution pour un cas de test qui doit provoquer le pire cas d'exécution. L'obtention de ces cas de tests n'est triviale que si on considère des programmes très simples.

Ces mesures du WCET vont nous servir de base pour estimer la précision de WCET obtenu analytiquement. Il faut cependant garder à l'esprit que la seule garantie que nous

ayons sur la correction de ces mesures est le fait que nous ayons une connaissance parfaite du code sous test, et donc que nous sommes en mesure de développer le test adéquat pour provoquer le pire cas d'exécution.

Dans le cas du code des benchmarks du paragraphe 6.2, l'écriture du test pire cas n'a pas posé de problème particulier. Les algorithmes sont relativement simples et bien connus, et les données manipulées peu conséquentes.

En revanche, en ce qui concerne les tests et mesures des temps d'exécution du code des appels système du paragraphe 6.3, la recherche du test pire cas a été bien plus difficile. D'une part les algorithmes mis en œuvre sont plus complexes, et d'autre part les données qui influencent le temps d'exécution sont non seulement les données représentées par le cas de test, mais aussi l'état du système avant l'exécution du test. Une grande attention a donc été apportée au développement des cas de test pour la mesure des WCET de code système. Nous avons donc l'intime conviction que nos cas de test provoquent le pire comportement des programmes analysés, mais sans en avoir la garantie absolue.

6.1.2 La plate-forme de test et mesure Pentane

Une fois le cas de test écrit, il reste à exécuter et à mesurer son temps d'exécution précisément. Pour ce faire, le programme à tester est exécuté sur une plate-forme de test. Cette plate-forme de test, appelée Pentane, a les fonctionnalités suivantes :

- Le chargement dynamique de code à tester.
- La ré-initialisation du contexte d'exécution.
- L'activation et la désactivation de certains éléments d'architecture.
- La mise en place et l'exécution du code à tester en isolation (il n'y a aucune autre activité pendant l'exécution des tests).
- La mesure du nombre d'occurrences de certains événements et du temps pendant l'exécution à une granularité très fine (*i.e.* le cycle pour les mesures de temps).
- Et l'envoi des résultats vers l'extérieur.

6.1.2.1 Architecture de la plate-forme de test

La plate-forme de test Pentane est constituée de deux machines (voir figure 6.1). La première est un PC équipé du processeur cible de notre analyse, un Intel Pentium P54C. C'est la machine cible, sur laquelle s'exécute le système minimaliste Pentane. Celle-ci est reliée par ligne série et par le réseau à une machine hôte dédiée au contrôle interactif de Pentane. Cette deuxième machine a pour rôle la préparation du code objet et des tests à exécuter ainsi que leur transfert vers la machine de test. Un fois le code et les données implantés en mémoire sur la machine cible, l'exécution est commandée depuis la machine hôte qui reçoit les résultats en retour.

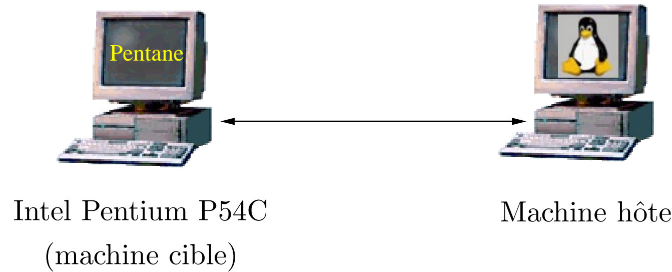


FIG. 6.1 – *Organisation de la plate-forme de test et mesure* PENTANE

6.1.2.2 Exécution d'un test et mesure du temps d'exécution

Le test et la mesure du temps d'exécution d'un programme s'effectue en 4 étapes.

- Premièrement, le code binaire du test est préparé et envoyé par la machine hôte. Le code est reçu par la machine cible et implanté en mémoire, où il est alors près à être exécuté.
- Avant chaque exécution d'un test, les conditions expérimentales décrites au paragraphe 6.1.3 (en particulier l'état de l'architecture matérielle) sont initialisées, et la mesure du temps d'exécution préparée.
- Le code à tester est alors exécuté en utilisant les données du test.
- Enfin, les différents compteurs de mesures (et éventuellement l'état de l'architecture) sont relevés, et les résultats sont envoyés à la machine hôte.

Lors de la préparation du code pour son analyse statique (*cf.* § 5.1), le programme est compilé et le code assembleur est généré (sans aucune optimisation à la compilation). C'est ce code qui va servir à l'obtention du WCET par test et mesure. Le code assembleur du programme est tout d'abord traduit en code binaire prévu pour être implanté en mémoire comme décrit en figure 6.3. Le code binaire obtenu est scindé en deux parties, le "code" d'une part et les "données" d'autre part [1]. Ces deux parties sont ensuite envoyées sur la plate-forme de test où ils sont implantés en mémoire.

En ce qui concerne les mesures pendant l'exécution, on utilise un outil offert par les processeurs de la famille Intel Pentium : le jeu de registres appelé MSR (pour *Machine Specific Registers*). Le nombre et les fonctions de ces registres dépendent de la génération du processeur considéré. Les MSR ont pour rôle de permettre l'accès à des fonctionnalités qui dépendent de l'implémentation d'un processeur particulier. Ils permettent l'accès (en lecture et écriture) à deux compteurs d'événements et à un compteur de cycles dont la lecture permet une mesure du temps très précise. Les événements qui peuvent être comptés sont par exemple le nombre d'instructions exécutées, le nombre de *miss* dans le cache d'instructions, etc. (voir annexe C pour la liste des événements). Les MSR permettent aussi d'accéder (en lecture et écriture) aux contenus des différents caches de l'architecture : cache d'instructions, cache de données,

1. Cette séparation permet de contrôler indépendamment l'effet des caches sur les données et sur le code

BTB, TLB. Enfin, ils permettent d'activer et de désactiver certains éléments d'architecture tels que la prédiction de branchement et le deuxième pipeline du processeur. Les MSR sont donc utilisés pour initialiser les conditions expérimentales de test (*cf.* § 6.1.3), configurer et réinitialiser les compteurs d'événements et de cycles, et enfin lire les résultats du test (valeurs des compteurs et état des caches).

6.1.2.3 Commande de la plate-forme

Le système de test PENTANE, qui s'exécute sur la machine test, est contrôlé interactivement depuis la station de travail par la ligne série. Le langage de commande offert par PENTANE permet de configurer en partie les conditions expérimentales, d'exécuter les tests, et d'obtenir les résultats. Le tableau 6.1 résume les différentes commandes du langage de commande de PENTANE qui sont détaillées en annexe C.

Commande	Description
pipe	permet d'activer et de désactiver le deuxième pipeline du processeur.
tlb	permet de lire le contenu du buffer de traduction d'adresses (TLB).
btb	pour activer, désactiver, vider et lire le contenu du buffer de prédiction de branchements (BTB).
icache, dcache	pour activer, désactiver, vider et lire les caches d'instructions et de données.
count	permet le comptage d'un ou deux types d'événement.
timer	donne le temps d'exécution du dernier test exécuté.
info	affiche certaines informations sur l'état de la plate-forme de test.
test	lance l'exécution du code à tester.
?	fournit l'aide en ligne des autres commandes.

TAB. 6.1 – *Liste des commandes de* PENTANE

La figure 6.2 présente un exemple d'utilisation interactive du langage de commande. Les compteurs sont tout d'abord configurés pour compter les événements `iexec` et `icachemiss` (commande 0). Puis, le cache est désactivé et vidé, et la configuration de la plate-forme est affichée. Le test numéro 3 est exécuté (commande 4), et les valeurs des compteurs après exécution sont affichées (commande 5). Les commandes suivantes répètent les mêmes opérations, mais en activant le cache d'instructions (on remarque la différence du compteur d'événements 1 qui compte le nombre d'échecs dans le cache d'instructions).

```
    - PENTANE (Intel Pentium Benchmark) -

0 > count iexec icachemiss
1 > icache off
2 > icache raz
3 > info msr
--------------------------------------
TR12 = 0x0000425f = 16991
CESR = 0x00ce00d6 = 13500630
   [ ]   Branch prediction mechanism
   [ ]   Double pipeline
   [ ]   Instruction cache L1
  Evt0 : iexec
  Evt1 : icachemiss
--------------------------------------


4 > test 3
Looking for test 3
Test 3 :
   - Code from 0x00800000 to 0x00800960,
     starting at 0x00800064
   - Data from 0x00c00000 to 0x00c002405

5 > count show
Evt0 = 0x00000000 0x00013908
Evt1 = 0x00000000 0x00008a22

6 > icache on
7 > icache raz
8 > info msr
--------------------------------------
TR12 = 0x00004257 = 16983
CESR = 0x00ce00d6 = 13500630
   [ ]   Branch prediction mechanism
   [ ]   Double pipeline
   [*]   Instruction cache L1
  Evt0 : iexec
  Evt1 : icachemiss
--------------------------------------


9 > test 3
Looking for test 3
Test 3 :
   - Code from 0x00800000 to 0x00800960,
     starting at 0x00800064
   - Data from 0x00c00000 to 0x00c00240

10 > count show
Evt0 = 0x00000000 0x00013908
Evt1 = 0x00000000 0x0000000d
```

Fig. 6.2 – *Exemple d'utilisation interactive de* Pentane

6.1.3 Conditions expérimentales

Pour pouvoir comparer les résultats des mesures avec les résultats d'analyse, les conditions d'exécution des tests doivent être aussi proches que possible des hypothèses posées pour les conditions d'exécution. Il faut de plus s'assurer que ces conditions réelles ne soit jamais plus défavorables que les hypothèses d'analyse, sans quoi les WCET mesurés pourraient être supérieurs au WCET estimé analytiquement. Pour l'exécution des tests, les conditions expérimentales sont les suivantes :

- (i) le cache d'instruction et le BTB sont vides au début de l'exécution du test,
- (ii) le coût de la traduction des adresses virtuelles en adresses physiques est constant,
- (iii) le code est considéré en isolation des autres activités du système,
- (iv) le cache de données est désactivé,
- (v) le pipeline considéré est un pipeline entier simple (le second pipeline de l'Intel Pentium est désactivé).

Ces conditions expérimentales sont initialisées par la plate-forme de test PENTANE avant l'exécution des tests. Pour ce faire :

- (i) PENTANE invalide toutes les entrées du cache d'instructions et du BTB avant chaque exécution d'un test. Le cache et le BTB sont donc vides au début de chaque exécution.
- (ii) On utilise une pagination par pages de 4Mo, et le nombre de pages utilisées (*cf.* figure 6.3) est limité pour ne pas dépassé le nombre d'entrées du TLB (32 entrées). Le TLB est rempli artificiellement pour la traduction de l'espace adressable. On évite ainsi les *miss* dans le TLB, et le coût de la traduction d'adresses est ainsi constant.
- (iii) Les interruptions sont désactivées pendant l'exécution du test pour assurer une exécution du code isolée.
- (iv) L'architecture Intel Pentium ne permet pas de désactiver séparément les caches d'instructions et de données. Pour assurer que le cache de donnée n'est pas utilisé lors du test, les données manipulées par le programme de test (et par la plate-forme) sont placées dans une page non-cachable. Tous les accès aux données causent donc des *miss*.
- (v) Enfin, l'Intel Pentium offre la possibilité de désactiver son deuxième pipeline entier, ce que nous faisons.

Les différents éléments de la plate-forme sont organisés en mémoire comme présenté dans la figure 6.3.

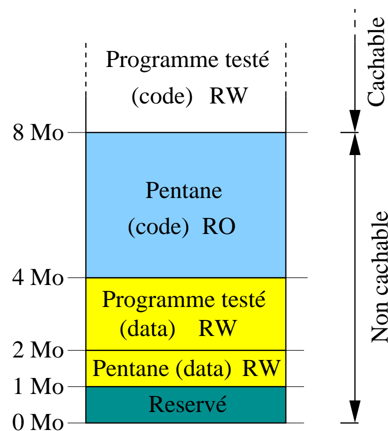


FIG. 6.3 – *Plan mémoire de* PENTANE

La correspondance entre l'espace d'adressage virtuel et la mémoire physique est directe. Une première partie, de 0 à 1Mo, est un espace réservé (mémoire vidéo, tables des pages, ...). De 1 à 2Mo on trouve les données de la plate-forme PENTANE, puis, de 2 à 4Mo, les données du programme à tester. Cette première page de 4Mo est définie non-cachable. La page suivante, de 4 à 8Mo, contient le code de la plate-forme PENTANE. Cette page est elle aussi marquée non-cachable pour ne pas interférer avec le code testé dans le cache d'instructions. Enfin, tout l'espace disponible au dessus de 8Mo est prévu pour recevoir le code à tester et est donc cachable (ce sont les seules pages cachables).

6.2 Analyse statique du WCET de code simple (benchmark)

Nous avons évalué la précision de la méthode d'analyse statique de WCET sur un ensemble d'applications de test. Cet ensemble de programmes est fourni par le groupe *RTOS Lab* de l'université de Séoul (*SNU real-time benchmarks suite*). Les programmes de cet ensemble sont souvent utilisés dans le domaine et décrivent un large spectre d'algorithmes et de comportements. Les caractéristiques de ces benchmarks (nom, taille, nombre de fonctions et description) sont présentées dans le tableau 6.2.

Appli.	Taille code	Nbr. de fonctions	Description
bs	934	2	Recherche binaire
crc	2223	3	Exemple de vérification CRC (Cyclic Redundancy Check)
fft1	4693	6	FFT (Fast Fourier Transform) utilisant l'algorithme de Cooly-Turkey
fibcall	599	2	Calcul de la suite de Fibonacci
inssort	753	1	Tri par insertion
jfdctint	2918	2	JPEG - transformation *forward DCT* en nombres entiers
matmul	1663	2	Multiplication de matrices
minver	6325	4	Inversion de matrice
qurt	2445	4	Calcul de la racine d'équations quadratiques

Tab. 6.2 – *SNU Real-Time Benchmark Suite*

L'évaluation de la précision de la méthode d'analyse statique consiste à comparer les estimations de WCET fournies par l'outil Heptane aux mesures effectuer pendant l'exécution du même code sur la plate-forme Pentane.

Les performances de la méthode d'analyse statique de WCET décrite au chapitre précédent sont résumées dans le tableau 6.3. Ce tableau confronte les résultats obtenus par analyse statique (colonnes intitulées *"est."*) avec ceux obtenus par exécution des programmes de test dans leur pire scénario d'exécution (colonnes *"exec."*).

La partie II du tableau permet d'évaluer l'analyse de haut niveau et en particulier la précision de la détermination du pire chemin d'exécution. On remarque que le ratio entre nombre d'instructions exécutées et obtenues par analyse est toujours supérieur à 1, ce qui montre (sur l'exemple) que l'analyse est sûre. Le ratio moyen est de 1.4. Après examen du code source des applications analysées, il s'est avéré que la sur-estimation ne provient pas d'annotations de boucles trop pessimistes mais plutôt de chemins d'exécutions infaisables qui ne sont pas détectés par notre outil d'analyse. Les parties III et IV permettent d'évaluer les méthodes de prise en compte du cache d'instructions et de la prédiction de branchement. Le ration $est/exec$ moyen est dans les deux cas de 1.5.

La partie V synthétise les résultats précédents en donnant directement le WCET, exprimé en nombre de cycles. Le ratio moyen est de 1.7, ce qui est une valeur tout à fait raisonnable.

I	II			III			IV			V		
	Instr. exécutées			Échecs cache			Échecs préd. br.			WCET		
Appli.	*est.*	*exec.*	$\frac{est}{exec}$	*est.*	*exec.*	$\frac{est}{exec}$	*est.*	*exec.*	$\frac{est}{exec}$	*est.*	*exec.*	$\frac{est}{exec}$
bs	136	134	1.01	7	6	1.17	20	16	1.25	1925	1840	1.05
crc	2906	2601	1.12	29	19	1.53	22	19	1.16	35440	31732	1.12
fft1	19759	14529	1.36	266	163	1.63	166	87	1.9	449852	254441	1.77
fibcall	349	348	1	5	4	1.25	10	9	1.11	8106	8008	1.01
inssort	1761	1760	1	11	8	1.38	53	41	1.29	21609	19307	1.12
jfdctint	17514	8340	2.1	174	69	2.52	44	15	2.94	237510	87000	2.73
matmul	7761	7761	1	19	13	1.46	155	154	1.01	110367	100017	1.1
minver	5681	4814	1.18	170	120	1.42	256	155	1.65	117214	76411	1.53
qurt	3645	1184	3.08	60	37	1.62	58	47	1.23	71892	19691	3.65

TAB. 6.3 – *Performances de l'analyse statique de WCET*

Le tableau 6.4 présente des résultats d'analyse statique de WCET pour lesquels un ou plusieurs modules de prise en compte de l'architecture ont été désactivés.

I	II	III	IV	V	VI	VII	VIII
Appli.	*exec.*	*est.*	*sans btb*	*sans icache*	*sans pipeline*	*sans archi*	$\frac{sans\ archi}{exec}$
bs	1840	1925	2083	4323	10245	12832	6,97
crc	31732	35440	38513	100152	86056	224740	7,08
fft1	254441	449852	487850	894153	1983670	3372442	13,25
fibcall	8008	8106	8720	16606	28903	54536	6,81
inssort	19307	21609	23416	51219	85329	161713	8,38
jfdctint	87000	237510	256505	460104	805540	1528447	17,57
matmul	100017	110367	119153	242744	378213	748129	7,48
minver	76411	117214	126484	269177	449088	854947	11,19
qurt	19691	71892	78039	170479	268908	519140	26,36

TAB. 6.4 – *Performances des mécanismes de prise en compte de l'architecture*

Ce tableau rappelle les WCET mesurés (colonne II) et estimés (colonne III) du tableau 6.3. Les colonnes IV, V et VI présentent les résultats d'analyse statique de WCET sans prise en compte, respectivement, du mécanisme de prédiction des branchements, du cache d'instructions, et du pipeline. On remarque que le pessimisme dû à la non prise en compte du mécanisme de prédiction des branchements (IV) est relativement faible par rapport à ceux exposés aux colonnes V et VI. Ce qui se comprend car ce mécanisme n'affecte qu'une partie relativement faible des instructions : les branchements alors que les deux autres mécanismes

(cache d'instruction et pipeline) ont un impact sur toutes les instructions. La colonne VII présente les résultats d'analyse de WCET quand aucun élément d'architecture n'est pris en compte. Ces résultats sont comparés aux WCET mesurés dans la dernière colonne. Le ratio obtenu si on effectue l'analyse statique sans aucune prise en compte de l'architecture matérielle est de 11.7 en moyenne. Ce qui montre que l'analyse au niveau micro-architecture est incontournable pour obtenir des performances d'analyse raisonnables.

Le graphique de la figure 6.4 présente les résultats du tableau 6.4 sous une forme différente. La première portion de chaque histogramme (celle du bas, notée II) est le WCET mesuré. La portion suivante (notée III-II) est le pessimisme de l'analyse par rapport au WCET mesuré, quand les trois éléments d'architecture sont pris en compte. Les trois portions suivantes représentent respectivement les pessimismes des analyses statiques sans prise en compte de la prédiction de branchement (IV-III), du cache d'instruction (V-III), et du pipeline (VI-III), par rapport aux résultats d'une analyse qui prend en compte ces trois éléments. On remarque que la plus grande source de pessimisme est la non prise en compte du pipeline.

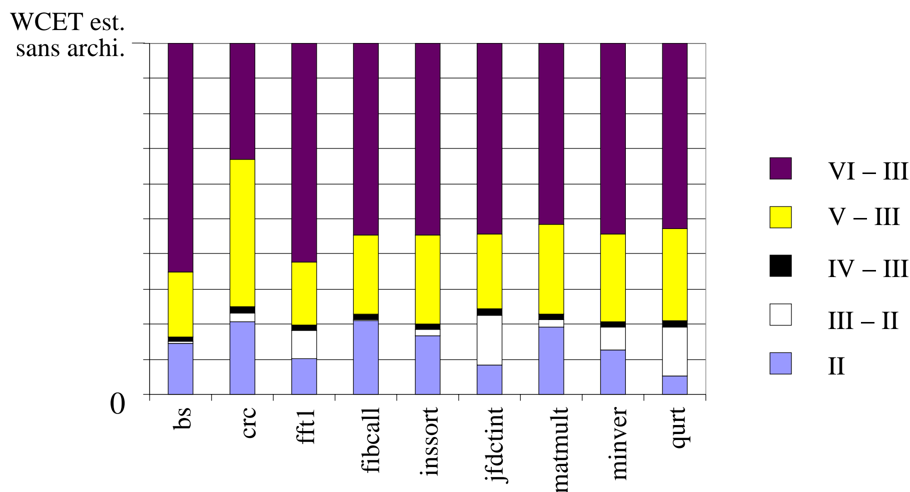


FIG. 6.4 – *Pessimisme des estimations*

6.3 Analyse statique de WCET de code système

Nous avons utilisé la méthode d'analyse statique de WCET présentée au chapitre précédent pour analyser le code source d'un noyau de systèmes temps-réel, le noyau RTEMS (Real-Time Executive for Multiprocessor Systems) [OAR98]. La contribution de ce travail [CP01c, CP01a] est double.

D'une part, nous avons expérimenté l'utilisation d'une méthode d'analyse statique de WCET sur du code de taille conséquente. À l'heure actuelle, de nombreuses méthodes et de nombreux outils d'analyse statique de WCET sont évalués sur des petits codes applicatifs,

exception faite du travail décrit dans [Eng99], qui étudie les propriétés de code applicatif embarqué complexe (dans les domaines des télécommunications, du contrôle de véhicule, et de l'électronique grand public).

D'autre part nous avons appliqué notre méthode d'analyse sur du code *système* et non pas utilisateur comme cela est pratiqué usuellement. Ceci permet d'obtenir une évaluation sûre du WCET d'un système d'exploitation. L'utilisation de ce type de méthode est inhabituelle dans le domaine des systèmes d'exploitation temps-réel, pour lesquels les performances sont évaluées grâce à des jeux de test (*benchmarks*), et sont souvent peu représentatives du pire cas d'utilisation du système d'exploitation.

6.3.1 Le noyau temps réel RTEMS

Cette étude porte sur le code source de RTEMS 4.0.0 pour une architecture i386. RTEMS est un micro-noyau temps-réel pour applications embarquées, dont les sources sont publiques. Les caractéristiques du noyau temps-réel RTEMS sont les suivantes :

- des capacités multi-tâches,
- un ordonnancement préemptif, à base de priorités,
- la communication et la synchronisation entre tâches,
- l'héritage de priorité pour éviter les problèmes d'inversions de priorités,
- l'allocation dynamique de la mémoire,
- des capacités de configuration.

L'interface de programmation de RTEMS comporte 85 appels système (appelés *directives* dans la terminologie RTEMS) regroupés en 17 modules (initialization, task, interrupt, clock, timer, semaphore, message, event, signal, partition, region, dual ported memory, I/O, fatal error, rate monotonic, user extensions, et multiprocessing). Tous les éléments gérés par RTEMS, tels que les tâches, les sémaphores et les messages (i.e. les *objets* dans la terminologie RTEMS) sont nommés et alloués de la même façon.

Nous présentons maintenant les résultats de l'étude de 12 directives clés de gestion des tâches et de synchronisation du noyau RTEMS (voir tableau 6.5).

Le code source des 12 directives analysées est réparti entre 91 fichiers sources dont : 28 fichiers C, un fichier assembleur, 38 fichiers de définition de constantes, de macro-commandes et de types, et 24 fichiers de fonctions "en-ligne". Le code source (hors commentaires) comporte 14532 lignes, et définit 355 fonctions.

Le code source a été adapté pour respecter les restrictions du langage analysable décrites au paragraphe 5.1. De plus, on suppose une architecture mono-processeur. Le code source a donc été spécialisé, quand cela était nécessaire, pour supprimer le code relatif à la gestion des multiprocesseurs.

Nom	Description
rtems_clock_get	Informations de date et heure du système
rtems_semaphore_create	Création de sémaphore
rtems_semaphore_obtain	Prise de sémaphore
rtems_semaphore_release	Libération de sémaphore
rtems_task_create	Création de tâche
rtems_task_delete	Destruction de tâche
rtems_task_ident	Renvoi de l'identifiant d'une tâche
rtems_task_start	Démarrage d'une tâche
rtems_task_suspend	Suspension d'une tâche
rtems_task_resume	Reprise d'une tâche
rtems_task_set_priority	Affectation d'une priorité à une tâche
rtems_task_wake_after	Réveil d'une tâche après un certain délai

FIG. 6.5 – *Liste des directives RTEMS analysées*

6.3.2 Modifications du code source de RTEMS

En ce qui concerne le code source analysé, nous avons constaté que les restrictions imposées par HEPTANE sont raisonnables et que l'annotation des boucles a été possible (même si elle n'a pas toujours été triviale). Nous présentons maintenant plus en détail les différents aspects du code source qui ne respectait pas les contraintes sur le langage.

6.3.2.1 Les boucles

Comme on l'a déjà remarqué, l'analyse des boucles est d'une grande importance pour l'analyse statique de WCET. Le code analysé contient relativement peu de boucles (seulement 21 boucles pour 14532 lignes de code analysées).

Boucles sans fin

Une seule boucle sans fin a été rencontrée dans le code source de RTEMS. C'est le corps de la tâche *idle*, qui n'est exécutée que lorsqu'aucune autre tâche n'est prête à être exécutée. La fonction dans laquelle cette boucle apparaît n'appartient pas au graphe d'appel des directives analysées, et n'a donc aucune influence sur leur WCET.

Détermination du nombre maximum d'itérations des boucles

Notre analyseur nécessite que le code source soit annoté par l'utilisateur. Ces annotations indiquent le nombre maximum d'itérations de chaque boucle. Bien que les boucles soient peu nombreuses, l'obtention du nombre maximum d'itérations des boucles n'a été aisée que pour quatre d'entre elles (boucles *for* dont les nombres maximum d'itérations étaient directement

fournis par le test de boucle). Les 17 autres boucles ont exigé un examen plus fin du code source.

- Dans RTEMS, il existe deux types de nommage des objets : chaînes de caractères de type C ou valeur 32 bits. Les fonctions de gestion des noms (comparaison, copie et suppression de noms) comportent des boucles dont le nombre d'itérations dépend de la longueur du nom. Dans notre cas, seul le deuxième type de noms est utilisé.
- La conversion des noms des tâches en identificateurs est réalisée par une boucle qui balaye la table des tâches du système à la recherche du nom indiqué. Le nombre maximum d'itérations de cette boucle est *MAX_TASK*, où *MAX_TASK* est le nombre maximum de tâches présentes dans
  le système à un moment donné. RTEMS impose que cette valeur soit connue statiquement.
- Le concepteur fournit la liste de ses extensions. Le nombre d'extensions d'utilisateur est donc connu et l'annotation des boucles de gestion de ces extensions est possible.
- RTEMS utilise l'allocation dynamique de mémoire *first-fit* lors de la création des tâches et de leur suppression afin d'allouer et de libérer leur pile. La taille minimum des blocs allouables est *MIN_BLOCK_SIZE.* Dans le pire cas, la mémoire est constituée d'une succession de blocs libres et de blocs occupés de taille minimum. Le pire nombre d'itérations de la boucle de recherche d'un bloc libre est alors *HEAP_SIZE* / (2 * *MIN_BLOCK_SIZE*), où *HEAP_SIZE* est la taille de la zone mémoire consacrée à l'allocation dynamique.
- Le code de l'ordonnanceur comporte des boucles pour l'ajout des descripteurs des tâches dans la file d'exécution, selon leurs niveaux de priorité. Il parcourt la file d'exécution jusqu'à ce qu'un descripteur de tâche avec un niveau de priorité donné soit trouvé. Ces boucles itèrent au pire *MAX_READY_TASK* fois, où *MAX_READY_TASK* est le nombre maximum de tâches prêtes (au pire, *MAX_READY_TASK* est égal au nombre maximum des tâches *MAX_TASK*).

6.3.2.2 Les appels dynamiques de fonction

Les appels de fonctions via l'utilisation de pointeurs (appels *dynamiques* de fonctions) posent problème pour l'analyse statique de WCET, car la fonction effectivement appelée n'est connue qu'à l'exécution. Parmi les 368 appels de fonctions présents dans les 12 directives étudiées, nous n'avons trouvé que 15 appels dynamiques de fonctions, répartis en deux catégories :

- *Appels des extensions utilisateurs.* RTEMS permet aux concepteurs d'applications d'ajouter des routines d'extension au noyau. Ces routines seront appelées lors de l'occurrence d'événements spécifiques du système, comme par exemple la création de tâches ou le changement de contexte. Pour permettre l'analyse, nous avons choisi de demander au

concepteur d'identifier avant exécution toutes les routines d'extension à prendre en compte.
– *Appels des extensions d'API.* Les extensions d'API permettent à une API donnée de fournir des routines d'extension qui seront invoquées à certains points spécifiques : après une commutation de contexte, avant/après un appel à un gestionnaire de périphérique. Heureusement, les fonctions appelées sont connues statiquement.

Nous avons retrouvé la même situation concernant les appels dynamiques de certaines fonctions dépendant de l'architecture. Puisque l'architecture cible est connue statiquement, ces appels dynamiques ont pu être aisément remplacés par des appels statiques.

6.3.2.3 Les appels récursifs

La présence de récursivité dans le code est problématique pour l'identification du chemin d'exécution au pire cas. Aucune fonction récursive n'a été rencontrée parmi les fonctions analysées durant notre étude. Cette conclusion diffère de celle donnée dans [Eng99], qui se base sur l'étude de code applicatif (dans cette étude, 0.3% des fonctions sont récursives).

6.3.2.4 Les graphes de flot de contrôle non structurés

Un seul fichier source assembleur a été nécessaire pour l'analyse des douze directives choisies. Il contient le code de sauvegarde et de restauration du contexte d'une tâche (*i.e.* sauvegarder et restaurer les registres). Ce code n'a posé aucun problème d'analyse car il ne contient pas d'instructions de branchement.

Quatre boucles contenant des *goto* ont été rencontrées pendant l'étude des directives de RTEMS. Dans tous les cas, il a été possible de restructurer ces boucles.

6.3.3 Étude du comportement pire-cas de RTEMS

Nous présentons ici les observations effectuées lors de l'analyse de plusieurs algorithmes de RTEMS dont la détermination du comportement pire-cas (et donc du WCET) n'est pas triviale. Ces observations concernent l'allocation dynamique de mémoire, le mécanisme de changement de contexte, les possibilités de blocages, et l'influence des interruptions sur le WCET de RTEMS.

6.3.3.1 L'allocation dynamique de mémoire

Nous avons présenté, au paragraphe 6.3.2.1, une borne supérieure sur le nombre d'itérations de la boucle d'allocation dynamique de mémoire. La valeur de cette borne nous amène à constater que la stratégie d'allocation *first-fit* n'est pas adaptée à l'analyse statique de WCET.

Comme nous le verrons au paragraphe 6.3.4, la surestimation du temps d'exécution de la fonction d'allocation dynamique a un impact important sur l'estimation du WCET des directives qui l'utilisent.

6.3.3.2 La boucle principale de l'ordonnanceur

La partie la plus difficile à analyser dans le code source de RTEMS a été le code de commutation de contexte : la fonction *_Thread_Dispatch*.

Cette fonction est appelée par toutes les directives et gestionnaires d'interruption de RTEMS qui peuvent modifier l'attribution de processeur. Chaque fois que les files d'exécution sont modifiées, la variable partagée *_Context_Switch_necessary* devient *VRAI* mais l'attribution du processeur n'est pas changée immédiatement. L'attribution du processeur est modifiée exclusivement par la fonction *_Thread_Dispatch*.

Si un gestionnaire d'interruption demande une commutation de contexte, il ne fait appel à la fonction *_Thread_Dispatch* que si aucun changement de contexte n'est en cours. Si une exécution de *_Thread_Dispatch* est déjà en cours, le changement de contexte demandé par le gestionnaire d'interruption est effectué par la fonction *_Thread_Dispatch* interrompue ce qui se traduit par une itération supplémentaire de sa boucle principale.

La figure 6.6 présente une exécution possible de la fonction *_Thread_Dispatch*. Lors de cette exécution, un processus $P1$ demande un sémaphore déjà pris par un autre processus $P2$.

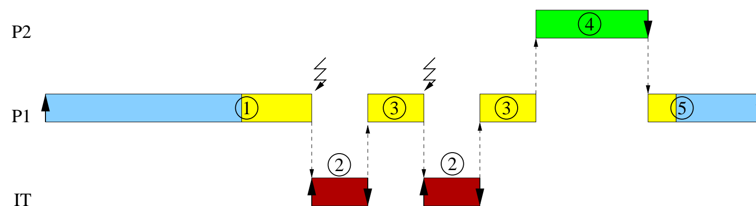


FIG. 6.6 – *Exemple d'exécution de _Thread_Dispatch*

Quand *rtems_semaphore_obtain* (processus $P1$) découvre que le sémaphore n'est pas libre, la file d'exécution de l'ordonnanceur ainsi que la file d'attente du sémaphore sont modifiées, et un changement de contexte est demandé. Puis, la fonction *_Thread_Dispatch* est appelée pour effectuer le changement de contexte (①). Supposons qu'une interruption se produise pendant la première itération de la boucle et induise une modification de l'attribution du processeur (②). Le changement de contexte demandé par le gestionnaire de l'interruption est retardé et sera effectué lors d'une deuxième itération de la fonction interrompue *_Thread_Dispatch* (③). Le même scénario peut se répéter et causer une troisième itération. La tâche dont la priorité est la plus élevée (ici *P2*) est alors terminée (④). Et la fin de la fonction *_Thread_Dispatch* et du processus $P1$ est exécutée (⑤). Pour borner le nombre d'itérations de la boucle principale de *_Thread_Dispatch* on doit connaître le nombre maximum d'interruptions pouvant arriver

pendant son exécution.

6.3.3.3 Gestion des appels bloquants et des changements de contexte

L'analyse d'ordonnançabilité nécessite l'identification exhaustive des blocages potentiels des tâches ainsi qu'une estimation de la durée de ces blocages.

La détermination des durées de blocage des tâches requiert une connaissance statique de l'ensemble des tâches à exécuter et de leurs synchronisations. La détermination des durées de blocage est du ressort de l'analyse d'ordonnançabilité et non de l'analyse statique de WCET (qui analyse du code ne comportant pas de synchronisation interne). Nous avons dû simplement vérifier que le noyau ne fait pas appel à des primitives bloquantes, c'est-à-dire qu'il n'utilise ni sémaphore ni événement pour synchroniser les accès concurrents à ses structures de données internes. Cette vérification a été réalisée par une analyse manuelle du code source du noyau. Le contrôle des accès concurrents aux structures de données de RTEMS ne fait pas appel à des primitives bloquantes, mais utilise la désactivation des interruptions.

En ce qui concerne les directives de RTEMS qui peuvent provoquer un changement de contexte (*par exemple rtems_semaphore_obtain*), l'outil d'analyse statique de WCET prend en compte le temps nécessaire à la sauvegarde et à la restauration du contexte de la tâche dans le code de *_Thread_Dispatch*).

Le WCET d'une directive comprend donc le temps requis pour passer à une autre tâche dans le cas où la tâche appelante se bloque.

Cependant, lors de l'analyse statique de WCET, l'analyseur ne sait pas si le changement de contexte est réellement effectué car il ne connaît pas la sémantique du code analysé. Il considère donc que la fonction de changement de contexte est une fonction ordinaire et ne prend pas en compte le surcoût dû au cache, à la prédiction de branchement et au pipeline lors du changement de contexte. Au pire, il faut supposer que le cache d'instructions et le tampon de cibles de prédiction de branchement sont complètement vidés à chaque changement de contexte ; le surcoût correspondant doit être ajouté au WCET de la fonction de changement de contexte calculé par l'analyseur de WCET.

6.3.3.4 Influence des interruptions

Une interruption peut se produire à presque n'importe quel moment de l'exécution d'une tâche. La présence d'interruptions a une influence directe sur le WCET d'une tâche, car elle conditionne le nombre d'itérations de la boucle principale de la fonction *_Thread_Dispatch* (voir § 6.3.2.1). C'est l'unique impact des interruptions sur l'analyse statique de WCET, et toute autre influence concerne l'analyse d'ordonnançabilité.

Pour pouvoir prendre en compte les interruptions lors de l'analyse d'ordonnançabilité, le WCET de tous les gestionnaires d'interruptions doit être connu, ce que permet d'obtenir l'analyse statique de WCET. Cependant, celle-ci ne considère que du code isolé, et ne prend donc pas en compte l'impact des interruptions sur la tâche interrompue (vidage du cache d'instruc-

tions, rupture du flot d'instructions dans le pipeline). Ce surcoût doit être ajouté au WCET du gestionnaire d'interruption avant l'analyse d'ordonnançabilité. [JS93] et [SEF98] présentent des exemples de prise en compte des interruptions dans l'analyse d'ordonnançabilité.

6.3.4 Résultats quantitatifs de l'analyse de RTEMS

Ce paragraphe présente les résultats quantitatifs de l'analyse. Nous comparons les résultats obtenus par l'analyse statique des 12 directives de RTEMS étudiées aux résultats de l'exécution réelle.

Pour mesurer les temps d'exécution de chacune des 12 directives, nous avons reproduit son pire scénario d'exécution. Par exemple, concernant les directives qui utilisent l'allocation de mémoire dynamique, comme *rtems_task_create*, nous avons fragmenté la mémoire avant d'appeler la directive concernée. Les mesure de temps d'exécution ont été réalisées en utilisant les mécanismes utilisées dans PENTANE, mais la plate-forme PENTANE n'a pu être employée pour l'exécution des tests en raison du caractère multi-tâche préemptif de RTEMS.

Le tableau 6.5 présente nos résultats. Les colonnes sont étiquetées *"est."* pour les résultats obtenus par analyse statique de TEPC et *"exec."* pour les résultats obtenus en exécutant les directives.

I	II			III			IV		
	Échecs cache			Échecs préd. br.			WCET		
Directive RTEMS	*est.*	*exec.*	$\frac{est}{exec}$	*est.*	*exec.*	$\frac{est}{exec}$	*est.*	*exec.*	$\frac{est}{exec}$
rtems_clock_get	10	10	1.00	8	7	1.14	1420	1302	1.09
rtems_semaphore_create	106	88	1.20	204	112	1.82	27050	11878	2.28
rtems_semaphore_obtain	380	180	2.11	983	363	2.71	288484	97417	2.96
rtems_semaphore_release	227	137	1.66	271	131	2.07	40414	17025	2.37
rtems_task_create	393	231	1.70	852	283	3.01	435550	309618	1.41
rtems_task_delete	315	213	1.48	630	262	2.40	72679	47275	1.54
rtems_task_ident	23	35	1.21	229	31	7.39	150093	98439	1.52
rtems_task_resume	93	90	1.03	138	83	1.66	19045	11521	1.65
rtems_task_set_priority	164	140	1.17	182	139	1.31	24728	20179	1.23
rtems_task_start	473	214	2.21	730	149	4.90	138940	36532	3.80
rtems_task_suspend	111	100	1.11	149	110	1.35	20106	16237	1.24
rtems_task_wake_after	156	116	1.34	250	122	2.05	19823	16923	1.17
Moyenne			1.44			2.65			1.86

TAB. 6.5 – *Confrontation entre résultats de l'analyse et résultats expérimentaux*

La colonne II donne le nombre de lectures de blocs d'instructions ayant provoqué un défaut dans le cache ; la colonne III présente le nombre de branchements mal prédits par le mécanisme de prédiction de branchement. La colonne IV confronte les TEPCs mesurés et

estimés, exprimés en nombres de cycles. Ces résultats montrent (sur l'exemple) que l'analyseur Heptane fournit des résultats sûrs.

Les deux sources principales du pessimisme de l'analyse ont été identifiées. La première est la multiplicité des appels à la même fonction dans une même directive, alors que cette fonction est exécutée au plus une fois par directive. Cette fonction peut être par exemple *Thread_Dispatch*, qui effectue le changement de contexte. Son WCET est important et elle peut apparaître jusqu'à 3 fois dans l'arbre d'appel de certaines directives. La deuxième source de pessimisme provient de quelques fonctions qui peuvent être appelées aussi bien par l'utilisateur que par les directives de RTEMS. Dans le cas des appels depuis les directives de RTEMS, il s'avère que la plupart des paramètres de ces fonctions sont constants. Et donc qu'une partie du code de ces fonctions n'est pas utilisée. D'où une surestimation du WCET des fonctions due à la prise en compte de chemins d'exécution impossibles lors du calcul du WCET. Nous étudions actuellement la possibilité d'éliminer ces deux sources de pessimisme en appliquant une des techniques d'évaluation partielle sur le code source de RTEMS. L'évaluation partielle est une transformation du code source qui devrait nous permettre de supprimer ce type de chemin d'exécution impossible.

Cependant, certains chemins d'exécution impossibles restent difficilement détectables statiquement, comme par exemple des branchements conditionnels mutuellement exclusifs.

Sur l'exemple suivant, le pire chemin d'exécution est A;B;C;D. Mais les branches B et D s'excluent mutuellement. La détection de ce type de situation requiert une analyse flot de données.

☞ Exemple :

```
if(i==0)                      | A
      {some code; j=1;}       | B
if(j==0)                      | C
      {some code}             | D
```

6.3.5 Enseignements pour l'analyse statique de WCET de code système

Notre expérience d'estimation du WCET des directives de RTEMS par analyse statique nous a permis de tirer quelques enseignements en ce qui concerne l'adaptation de l'outil d'analyse de WCET à du code système, et plus généralement à du code complexe.

En ce qui concerne le code source analysé, nous avons constaté que les restrictions imposées par Heptane sont raisonnables : la plupart des appels dynamiques ont pu être remplacés par des appels statiques, et les "goto" ont été facilement remplacés par des boucles. Nous avons aussi observé que RTEMS est constitué d'un grand nombre de fichiers répartis dans de nombreux répertoires et que le degré de réutilisation des fonctions est élevé. Dans une large majorité des cas (89%), les fonctions appelées et appelantes sont déclarées dans des fichiers différents. Pour être facilement utilisable, un outil d'analyse statique de WCET devrait

pouvoir traiter des fonctions dispersées dans différents fichiers et répertoires. En outre, nous avons observé que la plupart des fichiers sont utilisés pour l'analyse de plusieurs directives de RTEMS. En moyenne, un fichier de code (*c.-à-d.* un fichier contenant du code de C ou des fonctions "en-ligne") est utilisé pour l'analyse de 6 directives. Par conséquent, puisque l'analyse statique de WCET peut être longue (jusqu'à 10 minutes au pire pour une directive de RTEMS), nous pensons qu'il serait avantageux de pouvoir garder des résultats partiels de WCET afin de les réutiliser plus tard. Une question importante est alors celle du choix des informations qui doivent constituer un WCET partiel permettant l'obtention de WCET qui ne soient pas trop pessimistes.

Concernant l'identification du pire chemin d'exécution, l'obtention du nombre maximum d'itérations des boucles n'a été évidente que pour 25% d'entre elles. L'obtention du nombre maximum d'itérations des boucles restantes a nécessité une étude approfondie du code source de RTEMS.

De plus, nous avons remarqué que le nombre maximum d'itérations de la boucle de la fonction *_Thread_Dispatch* dépend de l'occurrence d'interruptions pendant son exécution. Si la seule source d'interruption dans le système est l'interruption d'horloge, alors, grâce à l'intervalle de temps entre deux interruptions d'horloge, il ne peut y avoir au plus qu'une interruption d'horloge pendant la durée d'exécution de la fonction *_Thread_Dispatch*. Cette boucle peut alors être annotée avec la valeur 2. S'il existe d'autres sources d'interruption, l'estimation du nombre maximum d'itérations de cette boucle n'est possible que si on dispose de suffisamment d'informations pour majorer le nombre de demandes de changement de contexte effectuées par les gestionnaires d'interruption pendant l'exécution de la fonction *_Thread_Dispatch*. Pour cela, la connaissance du temps d'exécution du gestionnaire d'interruption, ainsi que l'intervalle minimum entre deux interruptions successives suffisent.

En ce qui concerne les routines d'allocation dynamique de mémoire, et bien que l'on ait pu exhiber une borne supérieure sur leurs WCET, nous ne pouvons que constater que la stratégie d'allocation *first-fit* n'est pas adaptée à l'analyse statique de WCET, et plus généralement, aux systèmes temps-réel strict. La surestimation du temps d'exécution de la fonction d'allocation dynamique a un impact important sur l'estimation du WCET de la directive de création de tâche. Cette surestimation pourrait être réduite de plusieurs manières :

- en utilisant des stratégies d'allocation dynamique mieux adaptées à l'analyse de WCET que *first-fit*,
- en restreignant les possibilités d'allocation dynamique (par exemple, en réduisant le choix des tailles des blocs allouables dynamiquement).

Enfin, nous avons remarqué que les bornes sur le nombre d'itérations de certaines boucles dépendent du nombre de tâches dans les files de l'ordonnanceur. En effet, l'attribution du processeur aux tâches par l'ordonnanceur de RTEMS est réalisée par un algorithme préemptif à base de priorité, les tâches de même priorité étant gérées par la méthode du tourniquet (*round-robin*). On peut réduire le pessimisme si l'utilisateur connaît statiquement la répartition des

tâches dans la file d'exécution.

6.4 Conclusion

Dans ce chapitre nous avons évalué les performances de notre méthode d'analyse statique de WCET. Les résultats des analyses, portant aussi bien sur du code simple que sur du code de système d'exploitation, ont montré l'efficacité des méthodes de prise en compte de l'architecture matérielle. De plus, on a pu constater la nécessité d'utiliser de telles méthodes de prise de prise en compte de l'architecture, sans quoi le pessimisme des estimations est bien trop important pour qu'elles soient utilisables (ratio $est/exec > 10$).

En ce qui concerne le niveau haut de l'analyse, on a pu constater que la majeure partie du pessimisme à ce niveau de l'analyse est due à l'incapacité de la méthode d'analyse à prendre en compte les chemins infaisables tels que les portions de code mutuellement exclusives.

Cette expérimentation nous a apporté des résultats autres que l'estimation de la précision des estimations. Comme par exemple le fait que les restrictions posées sur le langage source des programmes analysés n'ait pas posé de problème majeur ni pour le code des "petits" *benchmarks*, ni pour du code complexe, à savoir le code d'un système d'exploitation.

En ce qui concerne l'analyse du code système, nous avons noté le manque de certaines informations pour pouvoir soit l'analyser, soit réduire le pessimisme des estimations. Ces informations peuvent être par exemple un délai minimum d'inter-arrivée des interruptions si on suppose qu'elles influencent le comportement de certains algorithmes (*cf.* paragraphe 6.3.3.2 pour un exemple)

Un autre problème posé par le code système est l'algorithme d'allocation dynamique de mémoire dont le temps d'exécution est très variable et le WCET très important. Il nous semble important, si on veut pouvoir utiliser un tel mécanisme dans un système temps-réel, de se pencher sur ce problème.

Nous avons tirer plusieurs enseignements de ces expérimentations, en particulier sur les fonctionnalités que devrait offrir un analyseur statique de WCET. Une première amélioration possible de l'outil est liée à l'organisation du code à analyser. Il arrive souvent que les fonctions appelées et appelantes soient déclarées dans des fichiers différents. Ceci favorise la construction d'outils d'analyse de WCET permettant l'analyse multi-fichier, voir même capable d'effectuer des estimations de WCET *partielles* (*c.-à-d.* l'analyse de fragments de programmes plutôt que des programmes entiers).

La précision des résultats de l'analyseur statique peut être améliorée, par exemple en prenant en compte le deuxième pipeline de l'Intel Pentium. Une telle adaptation est grandement facilitée par la structure modulaire de l'analyseur. Une autre piste pour améliorer la précision de l'analyse serait la possibilité de spécifier les chemins infaisables.

Conclusion

Le but de ce travail de thèse est l'étude de l'estimation de temps d'exécution au pire cas par analyse statique de programme. Pour conclure cette étude, nous proposons un bilan de notre travail puis examinons les perspectives ouvertes par ce dernier.

Bilan

Nous avons étudié dans ce document l'utilisation de méthodes d'analyse statique pour estimer le temps d'exécution au pire cas de programme.

Après avoir présenté la problématique de l'estimation du temps d'exécution au pire cas, nous avons comparé deux approches : l'estimation par test et mesure, et l'analyse statique de WCET. Nous avons ensuite étudié les différentes méthodes d'analyse statique de WCET existantes sous plusieurs aspects : les différents types de données nécessaires à ces méthodes d'analyse statique de WCET, et les techniques mises en œuvre par ces méthodes. À cet effet, nous avons distingué deux niveaux d'analyse, le niveau haut qui regroupe les techniques de recherche du pire chemin d'exécution, et le niveau bas qui est chargé de la modélisation du comportement de l'architecture matérielle. Enfin, nous avons présenté les différentes techniques existantes permettant de réaliser ces deux niveaux de l'analyse statique.

Dans un deuxième temps, nous avons fait plusieurs propositions pour améliorer la précision de l'analyse statique à base d'arbre syntaxique. La première proposition est une nouvelle technique d'annotation des boucles qui permet de représenter précisément et statiquement le comportement dynamique des boucles, et ce même en présence de boucles non-rectangulaires. Cette technique d'annotation est basée sur l'utilisation d'annotations symboliques et sur le calcul du WCET par un outil d'évaluation symbolique. Un avantage du calcul symbolique du WCET est alors la possibilité d'exprimer le WCET en utilisant des variables qui restent non-instanciées dans le résultat final.

Une autre proposition est l'utilisation du niveau d'emboitement des boucles comme information représentant le contexte d'exécution des instructions. Le comportement des instructions vis-à-vis de l'architecture peut être estimé pour un niveau d'emboitement de boucle par-

ticulier et être plus précis. Ainsi, estimer le comportement d'une instruction revient à connaître les estimations de son comportement pour tous les niveaux d'emboitement de boucles auxquels elle appartient. Nous avons adapté deux techniques d'analyse bas niveau existantes pour qu'elles utilisent les niveaux d'emboitement de boucles. Il s'agit de la simulation statique du cache d'instructions de F. Mueller [AMWH94] et de la simulation d'exécution pipelinée par table de réservation [Kog81].

Nous avons aussi étudié l'impact du mécanisme de prédiction des branchements sur l'analyse de WCET. La prise en compte de cet élément d'architecture par l'analyse de WCET bas niveau n'avait, à notre connaissance, pas été envisagée. La technique proposée permet d'estimer le comportement des instructions de branchement vis-à-vis du mécanisme de prédiction des branchements. Les branchements ne sont donc plus systématiquement estimés mal-prédits et la précision de l'analyse s'en trouve améliorée. Comme les deux autres techniques de prise en compte de l'architecture, la simulation statique de prédiction de branchement utilise les niveaux d'emboitement de boucles.

Nous avons présenté dans le chapitre suivant l'implémentation d'un analyseur statique de WCET à base d'arbre syntaxique mettant en œuvre nos propositions. Le prototype d'analyseur a été réalisé de manière à être modulaire. En effet, il est constitué d'un cadre générique pour l'analyse de WCET *tree-based* dans lequel viennent prendre place des modules de prise en compte de l'architecture matérielle. Nous avons discuté des avantages de cette modularité en ce qui concerne l'adaptabilité d'un tel outil, en particulier en ce qui concerne le langage source et l'architecture matérielle.

Enfin, dans le dernier chapitre du document ont été présentées les expérimentations réalisées avec notre prototype d'analyseur statique de WCET. Nous avons tout d'abord présenté la plate-forme de test et mesure utilisée pour obtenir les résultats de référence, puis nous avons comparé les résultats d'analyse à ces résultats de référence.

Les expérimentations ont porté sur deux types de code. Nous avons tout d'abord analysé de petits codes numériques couramment utilisés dans le domaine. Les résultats obtenus ont montré l'efficacité de la méthode d'annotation proposée ainsi que celle de la prise en compte de l'architecture, et nous ont permis d'identifier d'autres sources de pessimisme. Puis, nous avons analysé une partie du code source d'un système d'exploitation temps-réel (RTEMS). De cette deuxième expérimentation, nous avons tirer plusieurs enseignements en ce qui concerne l'applicabilité de l'analyse statique de WCET à ce genre de code, les améliorations à apporter à l'outil d'analyse, et les difficultés rencontrées qui sont spécifiques au code système.

Perspectives

De nombreuses voies restent à explorer suite à notre travail. Certaines de ces voies ont d'ailleurs été mentionnées au cours de l'exposé.

En premier lieu, toutes les propositions qui ont été faites dans ce document l'ont été dans le cadre de l'analyse statique de WCET à base d'arbre syntaxique. Nous n'avons pas considéré la possibilité de les appliquer à une méthode *IPET* (l'autre classe importante de méthodes d'analyse statique). Ainsi, la possibilité d'utiliser des annotations symboliques dans le cadre d'une analyse statique par *IPET* mérite d'être étudiée de manière approfondie. Il en va de même en ce qui concerne l'utilisation des niveaux d'emboitement de boucles. L'intérêt d'une telle méthode serait d'associer les avantages des méthodes *IPET*, c'est-à-dire la possibilité de spécifier des contraintes sur les chemins d'exécutions (chemins infaisables, exclusions mutuelles de blocs), et l'utilisation des contextes d'analyses et les annotations symboliques de notre méthode à base d'arbre syntaxique.

L'évaluation comparée des performances de ces deux classes de méthodes d'analyse statique (*tree-based* et *IPET*) selon différents critères est une comparaison qui nous intéresse et qui mérite d'être conduite.

Comme nous l'avons montré au chapitre 1, il existe plusieurs techniques permettant la génération automatique des bornes sur le nombre d'itérations des boucles par analyse statique. L'adaptation d'une de ces méthodes pour obtenir automatiquement les annotations symboliques pour les boucles non-rectangulaires est un problème qui mérite d'être étudié.

Nous travaillons à l'utilisation de *l'évaluation partielle* avant analyse de WCET (*c.-à-d.* la spécialisation d'un programme en exploitant la connaissance partielle de ses entrées) pour améliorer la précision des résultats de l'analyse de WCET. Cette technique devrait permettre d'écarter quelques chemins d'exécution impossibles lors de l'analyse.

Une autre limitation de l'analyse statique de WCET est la nécessité d'avoir accès à l'intégralité du code source constituant le programme analysé. On se heurte alors à la question de la divulgation du code source, qui dans le milieu industriel peut poser problème. D'autre part certaines portions de code, les librairies par exemple, seront analysées très souvent car elles apparaissent souvent dans le code des programmes. C'est pourquoi nous pensons qu'il serait avantageux de pouvoir conserver des résultats *partiels* d'estimation de WCET afin de les réutiliser plus tard, et éventuellement se dispenser de l'accès à une partie du code source. Par exemple, les WCET partiels des fonctions d'une librairie propriétaire pourraient être joints au code binaire de la librairie. Ainsi, les WCET partiels seraient utilisés pour calculer le WCET global des programmes utilisant cette librairie. Cet aspect mérite aussi d'être étudié plus avant.

Dans le même ordre d'idée d'économie d'analyse, l'obtention d'estimations de WCET portables d'une architecture à une autre est une perspective intéressante. Ainsi, le WCET de programme portable (en bytecode Java par exemple) pourrait être estimé en faisant abstraction de l'architecture cible. Un tel WCET pourrait être spécialisé pour une architecture particulière sans qu'on ait besoin de recommencer l'analyse du programme.

Concernant le prototype d'analyseur statique Heptane, nous envisageons sa diffusion. Le reciblage d'Heptane sur une nouvelle architecture, ainsi que l'ajout de nouveaux modules de prise en compte d'éléments d'architecture restent à expérimenter.

Enfin, une thèse actuellement en cours s'intéresse à la possibilité d'utiliser les résultats obtenus dans le domaine de l'analyse statique de WCET pour la génération dynamique de tests. Le but étant alors de pouvoir estimer le WCET de programmes pour lesquels le code source n'est pas disponible.

Annexe A

XML [Con] est aujourd'hui présenté comme un nouveau standard dont la vocation est de standardiser le formatage des données indépendamment d'un quelconque format propriétaire, quelle que soit leur type. XML permet, entre autre, le stockage de données structurées de manière arborescente. XML est un langage ouvert dans le sens où il est possible de créer ses propres marqueurs de manière à définir la structure d'un document.

Les documents XML ont parmi leurs nombreuses propriétés la possibilité d'être définis de manière plus ou moins formelle. Les DTD (Document Type Definition) ont été la première technologie permettant de spécifier les éléments contenus dans un document XML et leurs enchaînements logiques.

A.1 DTD spécifiant le format XML des données d'entrées d'HEPTANE

Le rôle d'une DTD est de définir le vocabulaire et la structure d'un document XML. Une DTD est caractérisée par un ensemble de règles. Elles permettent de spécifier les éléments et leurs attributs, ainsi que leurs relations, leurs ordres et leurs fréquences d'apparition dans le document XML. La DTD ci-dessous spécifie le format XML de stockage de la représentation des programmes analysés.

```
1  <!ENTITY % node-leaf "(sequence|if|loop|basic-block|extcall|void)">
2  <!ENTITY % sequence-child "(if|loop|basic-block|extcall)">
3
4  <!ELEMENT program (%node-leaf;)>
5  <!ELEMENT sequence (%sequence-child;,(%sequence-child;)+)>
6  <!ELEMENT if (%node-leaf;,%node-leaf;,%node-leaf;)>
7  <!ELEMENT loop   ((simple-annotatio|variable-annotation),%node-leaf;,%node-leaf;)>
8
9  <!ELEMENT basic-block (CFGname,CFG,BB,lnlev,cnlev,asm,lstpred,lstsuc)>
10 <!ELEMENT extcall (#PCDATA)>
```

```
11  <!ELEMENT void EMPTY>
12
13  <!ELEMENT CFGname (#PCDATA)>
14  <!ELEMENT CFG (#PCDATA)>
15  <!ELEMENT BB (#PCDATA)>
16  <!ELEMENT lnlev (#PCDATA)>
17  <!ELEMENT cnlev (#PCDATA)>
18  <!ELEMENT asm (label|inst)+>
19  <!ELEMENT label (#PCDATA)>
20  <!ELEMENT inst (addr,size,code)>
21  <!ELEMENT addr (#PCDATA)>
22  <!ELEMENT size (#PCDATA)>
23  <!ELEMENT code (#PCDATA)>
24  <!ELEMENT lstpred (pred)*>
25  <!ELEMENT lstsuc (suc)*>
26  <!ELEMENT pred (CFG,BB)>
27  <!ELEMENT suc (CFG,BB)>
28
29  <!ELEMENT simple-annotation (maxiter)>
30  <!ELEMENT variable-annotation (maxiter,counter)>
31  <!ELEMENT maxiter (#PCDATA)>
32  <!ELEMENT counter (#PCDATA)>
```

La première ligne donne la liste des éléments constituant un arbre syntaxique. La ligne 2 définit les éléments qui peuvent être des fils d'un nœud `sequence`. Cette définition interdit aux nœuds de type `vide` et `sequence` d'être mis en séquence. Les lignes 4 à 11 donne la structure de l'arbre syntaxique en définissant ces nœuds (ligne 4 à 7) et ces feuilles (ligne 9 à 11). Les lignes 13 à 27 décrivent les informations liées aux blocs de base. Ces informations sont organisées sous forme d'arbre et les feuilles sont repérées par le type `#PCDATA`. Enfin, les lignes 29 à 32 définissent les deux types d'annotations utilisables pour annoter les nœuds `loop`.

A.2 Exemple de représentation XML des données d'entrées d'HEPTANE

Un document XML se compose de deux parties :

- une en-tête indiquant la version du langage XML employé (la première ligne de l'exemple ci-dessous),
- le corps du document.

Le corps du document se compose d'éléments délimités par les marqueurs définis dans la DTD (par exemple `BB`, `lnlev`, *cf.* A.1). Chaque élément doit comporter un marqueur d'ouverture (par exemple `<BB>`) et un marqueur de fermeture (par exemple `<\BB>`). Les éléments doivent être organisés de manière arborescente, à partir d'un unique élément racine.

Le code XML ci-dessous est un exemple de représentation de programme respectant la DTD du paragraphe précédent.

```
 1 <?xml version="1.0" encoding="ISO-8859-1"?>
 2 <!DOCTYPE program SYSTEM "hexane.dtd">
 3 <program>
 4 <sequence>
 5   <basic-block>
 6     <CFGname>_main</CFGname>
 7     <CFG>0</CFG>
 8     <BB>0</BB>
 9     <lnlev></lnlev>
10     <cnlev></cnlev>
11     <asm>
12       <label>_main: </label>
13       <inst><addr>0</addr><size>1</size><code>  push ebp</code></inst>
14       <inst><addr>1</addr><size>2</size><code>  mov ebp,esp</code></inst>
15       .
16       .
17       .
18       <inst><addr>251</addr><size>1</size><code>        push eax</code></inst>
19       <inst><addr>252</addr><size>5</size><code>        call _NumberOfBitsNeeded</code></inst>
20     </asm>
21     <lstpred>
22     </lstpred>
23     <lstsuc>
24       <suc><CFG>1</CFG><BB>0</BB></suc>
25     </lstsuc>
26   </basic-block>
27       .
28       .
29       .
30 </sequence>
31 </program>
```

Les lignes 1 et 2 constituent l'entête XML qui indique la version d'XML, le nœud racine et la DTD à utiliser. Les lignes 3 et 31 délimitent le corps du fichier XML (`program` est la racine de l'arbre XML). Un exemple de bloc de base est ensuite présenté. On remarque en particulier les instructions qui le composent (11-19) et ces prédécesseurs et successeurs dans le graphe de flot de contrôle (21-25).

A.3 Extrait de la description XML du programme du paragraphe 5.2.2, page 113

```
...
<for-loop>
  <symbolic-annotation>
    <maxiter>getC(P,0)</maxiter>
    <counter>i</counter>
  </symbolic-annotation>
  <basic-block>
    <CFGname>_main</CFGname>
    <CFG>0</CFG>
    <BB>5</BB>
    <lnlev>0,0</lnlev>
    <cnlev></cnlev>
    <asm>
      <label>__halt5: </label>
      <label>L6: </label>
      <inst><addr>35</addr><size>3</size><code>   mov eax,dword ptr [ebp+0xfffffffc]</code></inst>
      <inst><addr>38</addr><size>4</size><code>   imul eax,dword ptr [ebp+0xfffffffc]</code></inst>
      <inst><addr>42</addr><size>3</size><code>   cmp dword ptr [ebp+0xfffffff8],eax</code></inst>
      <inst><addr>45</addr><size>2</size><code>   jl L9</code></inst>
    </asm>
    <lstpred>
      <pred><CFG>0</CFG><BB>4</BB></pred>
    </lstpred>
    <lstsuc>
      <suc><CFG>0</CFG><BB>7</BB></suc>
      <suc><CFG>0</CFG><BB>6</BB></suc>
    </lstsuc>
  </basic-block>
  <basic-block>
    <CFGname>_main</CFGname>
    <CFG>0</CFG>
    <BB>7</BB>
    <lnlev>0,0</lnlev>
    <cnlev></cnlev>
    <asm>
      <label>L9: </label>
      <label>__halt6: </label>
      <inst><addr>49</addr><size>3</size><code>   mov eax,dword ptr [ebp+0xfffffff8]</code></inst>
      <inst><addr>52</addr><size>3</size><code>   add dword ptr [ebp+0xfffffff4],eax</code></inst>
      <label>__halt7: </label>
      <label>L8: </label>
      <inst><addr>55</addr><size>3</size><code>   inc dword ptr [ebp+0xfffffff8]</code></inst>
      <inst><addr>58</addr><size>2</size><code>   jmp L6</code></inst>
    </asm>
    <lstpred>
      <pred><CFG>0</CFG><BB>5</BB></pred>
    </lstpred>
    <lstsuc>
```

Annexe B

Un certain nombre de fonctions doit être implémenté en Maple pour pouvoir manipuler les représentations incrémentales de WCET, les *ln-levels*, les annotations, etc. Nous donnons ici le code des fonctions Maple dédiées à la gestion de la pile des annotations (§ B.1), aux opérations sur les *ln-levels* (§ B.2), et aux opérations sur les représentations de WCET incrémentales (§ B.3).

B.1 Code Maple de gestion de la pile des annotations

Les fonctions suivantes permettent d'empiler une annotation sur la pile d'annotations et d'accéder au contenu de la pile. Une pile vide est aussi définie.

```
# Gestion de pile


push := (P,x) -> [x,op(P)]:    # Empile x sur la pile P
stack_get := (P,n) -> P[n+1]:  # Renvoi un element de la pile (0=sommet de pile)
getM  := (P,n) -> (P[n+1])[1]: # Acc{\'e}s aux {\'e}l{\'e}ments maxiter et compteur
getC1 := (P,n) -> (P[n+1])[2]:
getC2 := (P,n) -> (P[n+1])[3]:
getC := getC1;
pile_vide = []:                # Definition de la pile vide
```

Exemples d'utilisation des fonctions Maple de gestion de pile :

```
> P := push (P,[[Ma,Ca1,Ca2]]):
> P := push (P,[[Mb,Cb]]):
> P;
                                          [[[Mb, Cb]], [[Ma, Ca1, Ca2]]]


> getC2(P,1);
                                                      Ca2
```

```
> stack_get(P,0);
                                                  [[Mb, Cb]]
```

B.2 Code Maple de gestion des *ln-levels*

La fonction `SuccEq` correspond à l'ordre partiel $\succeq$ défini au paragraphe 4.2.2.

```
# Ordre partiel sur les ln-levels


SuccEq := proc(L1,L2)
        if (L1 = []) then RETURN(true); fi;
        if (L2 = []) then RETURN(false); fi;
        if (L1[1] = L2[1])
          then RETURN(SuccEq(L1[2..-1],L2[2..-1]));
          else RETURN(false);
        fi;
end:
```

Exemples d'utilisation de la fonction `SuccEq` :

```
> SuccEq([0],[]);
                                          false
> SuccEq([1],[1,0,2]);
                                          true
> SuccEq([2],[1,0,2]);
                                          true
```

B.3 Code Maple de gestion des représentations de WCET

L'opérateur `&U` réalise l'union ensembliste de deux représentations incrémentales de WCET.

```
# Union des ensembles de couples <WCET,ln-level>


`&U`:= proc()
        local i,L;
        L:=[(args[1])[i] $ i = 1 .. nops(args[1]) , (args[2])[i] $ i = 1.. nops(args[2])];
        RETURN(L)
end:
```

Exemples d'utilisation de l'opérateur `&U` :

```
> [ [12,[0]] , [18,[0,0]] ] &U [[13,[1]]];
```

```
[[12, [0]], [18, [0, 0]], [13, [1]]]
```

La fonction suivante et une fonction de factorisation des représentations incrémentales de WCET. Les éléments <WCET,*ln-level*> sont regroupés par *ln-levels*, puis tous les éléments dont le *ln-level* est inférieur au deuxième paramètre de la fonction (le *ln-level* L) sont regroupés, et associés à L.

```
# Factorisation


Facto := proc(Lst,L)
  local R,S,Lev,LevSet,i,j;
  R:= [];  LevSet:={};
  for i from 1 to nops(Lst) do
    if [op(Lst[i])][2] <> [-1] then
      if SuccEq( L , [op(Lst[i])][2])
        then LevSet := LevSet union {L}
        else LevSet := LevSet union {op(Lst[i])[2]}
      fi;
    fi;
  od;
  for i from 1 to nops(LevSet) do
    S := 0; Lev := LevSet[i];
    for j from 1 to nops(Lst) do
      if Lev = L then
        if SuccEq( Lev , [op(Lst[j])][2]) then S:=S+(op(Lst[j])[1]); fi;
      else
        if (op(Lst[j])[2]) = Lev then S:=S+(op(Lst[j])[1]); fi;
      fi;
    od;
    R := R &U [[ factor(S) , Lev ]];
  od;
  RETURN(R);
end:
```

Exemples d'utilisation de la fonction `Facto` :

```
> Facto( [[12, [0]], [18, [0, 0]], [13, [1]]]  , [0] );
                                              [[30, [0]], [13, [1]]]


> Facto( [[12, [0]], [18, [0, 0]], [13, [1]]] , [] );
                                                    [[43, []]]
```

La fonction `Mult` correspond à l'opérateur $L\bigcirc\!\!\!\!\!\sum$ défini au paragraphe 5.2.1. Ses paramètres sont : une représentation de WCET, l'expression *maxiter*, le *ln-level* de la boucle dont on calcule le WCET, une variable d'indice (qui correspond à λ).

```
# Multiplication


Mult1 := proc(Wcet,M,Lev,indice)
        if SuccEq( Lev , [op(Wcet)][2] )
        then RETURN([sum([op(Wcet)][1],indice=0..M-1),[op(Wcet)][2]]);
        else RETURN(Wcet);
        fi;
end:


Mult := proc(Lst,M,Lev,indice)
        local i,R;
        R:= [];
        for i from 1 to nops(Lst) do
                R := R &U [Mult1(Lst[i],M,Lev,indice)]
        od;
        RETURN(R)
end:


> Mult( [[12, [0]], [18, [0, 0]], [13, [1]]] , 100 , [0] , i );
                            [[1200, [0]], [1800, [0, 0]], [13, [1]]]
```

Annexe C

C.1 Le langage de commande de Pentane

Le langage de commande de Pentane est constitué des commandes suivantes.

`?`	`[commande]`	: affiche l'aide concernant une commande.
`pipe`	`on`	: active le deuxième pipeline (pipeline V) du processeur.
	`off`	: désactive le pipeline V du processeur.
`tlb`	`show`	: affiche le contenu du TLB.
	`enrg`	: enregistre l'état du TLB.
	`diff`	: affiche les différences entre le TLB et son dernier enregistrement.
`btb`	`on`	: active le BTB.
	`off`	: désactive le BTB.
	`flush`	: invalide le BTB.
	`raz`	: vide le BTB.
	`show`	: affiche le contenu du BTB.
	`showvalid`	: affiche les lignes valides du BTB.
	`enrg`	: enregistre l'état du BTB.
	`diff`	: affiche les différences entre le BTB et son dernier enregistrement.

Commande	Argument	Description
`icache`	`on`	: active le cache d'instructions.
	`off`	: désactive le cache.
	`flush`	: invalide le cache.
	`raz`	: vide le cache.
	`show`	: affiche le contenu du cache.
	`showvalid`	: affiche les lignes valides du cache.
	`showasm`	: affiche les instructions assembleur dans le cache.
	`showvalidasm`	: affiche les instructions assembleur dans les lignes valides du cache.
	`enrg`	: enregistre l'état du cache.
	`diff`	: affiche les différences entre le cache et son dernier enregistrement.
`dcache`	`flush`	: invalide le cache de données.
	`raz`	: vide le cache.
	`show`	: affiche le contenu du cache.
	`enrg`	: enregistre l'état du cache.
	`diff`	: affiche les différences entre le cache et son dernier enregistrement.
`count`	`list`	: donne la liste de événements comptables (voir table C.1).
	`evt1` `[evt2]`	: configure le(s) événement(s) à compter.
	`show`	: affiche les valeurs des compteurs.
`timer`		: affiche la durée d'exécution du test en cycles.
`info`	`msr`	: affiche l'état des registres MSR.
	`cr`	: affiche l'état des registres CR.
	`paging`	: affiche la configuration de la pagination.
`test`	`n`	: exécute le test numéro n.
`reboot`		: redémarre la machine de test.
`h`		: affiche l'historique des commandes.
	`n1`	: rappel de la commande $n1$.
	`n1 n2`	: rappel de toutes les commandes entre $n1$ et $n2$.

C.2 Les événements comptables par les MSR

Le tableau C.1 présente la liste des événements dont les occurences ou les durées peuvent être comptabilisées par les deux compteurs de l'Intel Pentium. Ces événements peuvent être les paramètres de la commande `counter`.

	Performance Monitoring Event	Occurrence or Duration?
0	Data Read	O
1	Data Write	O
2	Data TLB Miss	O
3	Data Read Miss	O
4	Data Write Miss	O
5	Write (hit) to M- or E-state lines	O
6	Data Cache Lines Written Back	O
7	External Snoops	O
8	External Data Cache Snoop Hits	O
9	Memory Accesses in Both Pipes	O
10	Bank Conflicts	O
11	Misaligned Data Memory or I/OReferences	O
12	Code Read	O
13	Code TLB Miss	O
14	Code Cache Miss	O
15	Any Segment Register Loaded	O
16	Reserved	
17	Reserved	
18	Branches	O
19	BTB Hits	O
20	Taken Branch or BTB hit	O
21	Pipeline Flushes	O
22	Instructions Executed	O
23	Instructions Executed in the v pipe e.g.parallelism/pairing	O
24	Clocks while a bus cycle is in progress(bus utilization)	D
25	Number of clocks stalled due to full writebuffers	D
26	Pipeline stalled waiting for data memoryread	D
27	Stall on write to an E- or M-state line	D
28	Locked Bus Cycle	O
29	I/O Read or Write Cycle	O
30	Non-cacheable memory reads	O
31	Pipeline stalled because of an addressgeneration interlock	D
32	Reserved	
33	Reserved	
34	FLOPs	O
35	Breakpoint match on DR0 Register	O
36	Breakpoint match on DR1 Register	O
37	Breakpoint match on DR2 Register	O
38	Breakpoint match on DR3 Register	O
39	Hardware Interrupts	O
40	Data Read or Data Write	O
41	Data Read Miss or Data Write Miss	O

Tab. C.1 – *Liste des événements comptables (extrait du Pentium Processor Family Developer's Manual d'Intel)*

Bibliographie

[ACC+99] E. Anceaume, G. Cabillic, P. Chevochot, A. Colin, D. Decotigny, and I. Puaut. Un support d'exécution flexible pour applications distribuées temps-réel dur. In *Première Conférence française sur les systèmes d'exploitation*, pages 109–120, Rennes, France, June 1999.

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[AMWH94] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS94)*, pages 172–181, December 1994.

[AW89] N. Altman and N. Weiderman. Timing variations in dual loop benchmarks. *Ada Letters*, 8(3):98–106, 1989.

[Bac96] T. Back. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 198 Madison Avenue, New York, New York 10016, 1996.

[BBMP00] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable WCET analysis framework. In *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, December 2000.

[BBW00] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using Java byte code. In *Proc. of the 12th Euromicro Workshop of Real-Time Systems*, pages 81–88, 2000.

[BJ95] S. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical Report TR94-072, University of North Carolina, Chapel Hill, April 1995.

[BMSO+96] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 204–212. IEEE Computer Society Press, June 1996.

[BN94] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1994.

[BRS96] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Proc. of the Sixth Workshop on Compilers for Parallel Computers*, December 1996.

[But97] G. C. Buttazzo, editor. *Hard real-time computing systems - predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[CBW96] R. Chapman, A. Burns, and A.J. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[CGG91] B. W. Char, K. O. Geddes, and G. H. Gonnet. *MAPLE V language reference manual*. Springer-Verlag, 1991.

[CLK94] J. Choi, I. Lee, and I. Kang. Timing analysis of superscalar processor programs using ACSR. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 63–67, May 1994.

[Col98] A. Colin. Estimation du temps d'exécution au pire cas d'applications temps-réel. Rapport de stage de DEA, Université de Rennes I, 1998.

[Con] World Wide Web Consortium. Extensible markup language (xml) 1.0. available at: http://www.w3.org/tr/rec-xml.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[CP01a] A. Colin and I. Puaut. Analyse de temps d'exécution au pire cas du système d'exploitation temps-réel RTEMS. In *Seconde Conférence française sur les systèmes d'exploitation*, pages 73–84, Paris, France, April 2001.

[CP01b] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[CP01c] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, June 2001.

[Dar59] C. Darwin. *The Origin of the Species by Means of Natural Selection*. Murray, London, 1859.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[EE00] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21th IEEE Real-Time Systems Symposium (RTSS00)*, Orlando, Florida, December 2000.

[EEA98] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution time analysis for optimized code. In *Proc. of the 10th Euromicro Conference on Real-Time Systems*, Berlin, Germany, June 1998.

[EG98] A. Ermedahl and J. Gustafsson. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2):61–74, 1998.

[Eng99] J. Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proceedings of the 1999 Real-Time technology and Applications Symposium*, pages 46–55, Vancouver, CA, June 1999.

[FMW97] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler technique to cache behavior prediction. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 37–46, June 1997.

[FW98] C. Ferdinand and R. Wilhelm. On predicting data cache behaviour for real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474, pages 16–30, Montreal, Canada, June 1998.

[HAM+99] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[HL95] T.-Y. Huang and W.-S. Liu. Predicting the worst-case execution time of the concurrent execution of instructions and cycle-stealing DMA I/O operations. In Richard Gerber and Thomas Marlowe, editors, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 30 of *ACM SIGPLAN Notices*, pages 1–6, New York, NY, USA, November 1995. ACM Press.

[HP94] J. Hennessy and D. Patterson. *Computer Organization and Design. The Hardware/Software Interface.* Morgan Kaufmann, Inc., 1994.

[HSR+00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3):129–156, May 2000.

[HSRW98] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Fourth IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.

[JS93] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS93)*, pages 212–221, Raleigh-Durham, North Carolina, December 1993.

[KMH96] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 230–240. IEEE Computer Society Press, June 1996.

[Kog81] P. Kogge. *The Architecture of Pipelined Computers*. McGraw Hill Book Company, New York, NY, 1981.

[KS86] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[KWH95] L. Ko, D. B. Whalley, and M. G. Harmon. Supporting user-friendly analysis of timing constraints. *ACM SIGPLAN Notices*, 30(11):99–107, November 1995.

[LG98] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1998.

[LHM97] S.-S Lim, J. H. Han, and S. L. Min. A worst case timing analysis technique for superscalar processors. Technical report, Departement of Computer Engineering, Seoul National University, 1997.

[LL73] C.L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LM95a] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Richard Gerber and Thomas Marlowe, editors, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 30 of *ACM SIGPLAN Notices*, pages 88–98, New York, NY, USA, November 1995. ACM Press.

[LM95b] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 30(11):88–98, November 1995.

[LML+94] S.-S Lim, S. L. Min, M. Lee, C. Y. Park, H. Shin, and C.-S. Kim. An accurate instruction cache analysis technique for real-time systems. *IEEE Workshop on Architectures for Real-Time Applications*, April 1994.

[LMW95a] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS95)*, pages 298–307, Pisa, Italy, December 1995.

[LMW95b] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *International Conference on Computer Aided Design*, pages 380–387, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, IEEE Computer Society Press, December 1996.

[LN88] K. Lin and S. Natarajan. Expressing and maintaining timing constraints in Flex. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS88)*, pages 96–105, December 1988.

[LS98] T. Lundqvist and P. Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–15, June 1998.

[LS99] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[MM98] D. Macos and F. Mueller. Integrating gnat/gcc into a timing analysis environment. In *Work-in-Progress of the 10th Euromicro Conference on Real-Time Systems*, pages 15–18, June 1998.

[Mue97] F. Mueller. Generalizing timing predictions to set-associative caches. In *Proc. of EuroMicro Workshop on Real-Time Systems*, pages 64–71, June 1997.

[Mue00] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[MWH94] F. Mueller, D. Whalley, and M. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1994.

[NTA90] V. Nirkhe, S. K. Tripathi, and A. K. Agrawala. Language support for the Maruti real-time system. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS90)*, pages 257–266, December 1990.

[OAR98] On-Line Applications Research Corporation, Huntsville, AL, USA. *RTEMS Applications C User's Guide*, 4.0 edition, October 1998. (http://www.oarcorp.com/RTEMS/rtems.html).

[OS97] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems (LCTRTS'97)*, June 1997.

[Par93] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[PB01] P. Puschner and G. Bernat. Wcet analysis of reusable portable code. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 45–52, Delft, The Netherlands, June 2001.

[PF99] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[PH99] P. Persson and G. Hedin. Interactive execution time predictions using reference attributed grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 173–184, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[Pre94] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1994.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[PS95] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat, Institut fur Technische Informatik, Wien, April 1995.

[PS97] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *Proc. of IEEE Real-Time Systems Symposium*, volume 13, pages 67–91. Kluwer Academic Publishers, 1997.

[Pus98] P. Puschner. A tool for high-level language analysis of worst-case execution times. In *Proc. of the 10th Euromicro Conference on Real-Time Systems*, pages 130–137, Berlin, Germany, June 1998.

[RLM+94] B.-D. Rhee, S.-S. Lim, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. Issues of advanced architectural features in the design of a timing tool. In *Proc. of the 11th Workshop on Real-Time Operanting Systems and Software*, pages 59–62, May 1994.

[SA00] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[SEF98] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of the 1998 Real-Time technology and Applications Symposium*, Hiroshima, Japan, October 1998.

[SF99] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 35–44, June 1999.

[Sha89] A. C. Shaw. Reasoning about time in higher-level langage software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[SR88] J.A. Stankovic and K. Ramamritham, editors. *Tutorial on Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[Sta96] J.A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, December 1996.

[Vrc94] A. Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Institut für Technische Informatik, Technische Universität Wien, Austria, april 1994.

[WM01] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 2001.

[WMH+97] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 1997 Real-Time technology and Applications Symposium*, pages 192–202, June 1997.

[WMH+99] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, 1999.

[WSP99] Joachim Wegener, Harmen Sthamer, and Hartmut Pohlheim. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *IEEE Real-Time Systems Symposium*, pages 270–271, 1999.

[Xu93] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

[ZBN93] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.