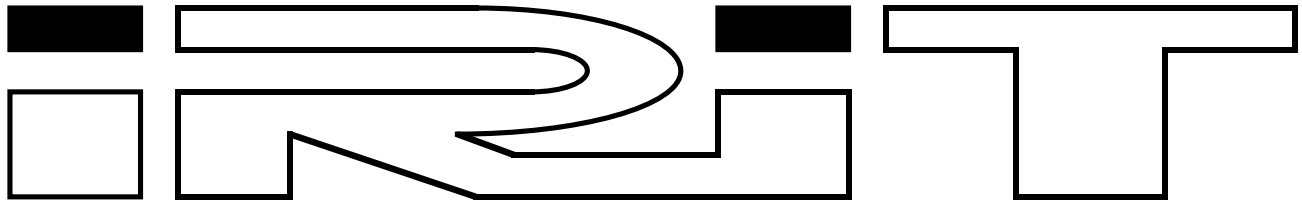


INSTITUT DE RECHERCHE EN INFORMATIQUE DE TOULOUSE

RAPPORT
INTERNE
IRIT-2002-15-R

PUBLICATION
INTERNE
1461

Calcul de majorants de pire temps d'exécution : état de l'art

Antoine Colin – Isabelle Puaut
Christine Rochange – Pascal Sainrat

CNRS (UMR 5505) - INP - UPS

Rapport interne IRIT 2002-15-R et publication interne IRISA 1461

Calcul de majorants de pire temps d'exécution : état de l'art

Antoine Colin[*] – **Isabelle Puaut**[**]
Christine Rochange[***] – **Pascal Sainrat**[***]

[*] Department of Computer Science, University of York,
Heslington - York YO10 5DD United Kingdom
acolin@cs.york.ac.uk

[**] Institut de Recherche en Informatique et des Systèmes Aléatoires,
Campus universitaire de Beaulieu, 35042 Rennes
puaut@irisa.fr

[***] Institut de Recherche en Informatique de Toulouse,
Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse cedex, France
{rochange, sainrat}@irit.fr

Résumé

La particularité des systèmes temps-réel strict est de devoir respecter de manière impérative des contraintes temporelles, qui sont le plus souvent des échéances de terminaison au plus tard. Dans de tels systèmes, il est courant d'utiliser des méthodes d'analyse d'ordonnancement, qui à partir de l'ensemble des tâches du système, déterminent si les échéances seront ou non respectées. La plupart de ces méthodes reposent sur la connaissance d'une borne supérieure du temps d'exécution de chaque tâche du système, nommée WCET pour `Worst-Case Execution Time`*. Cet article propose une synthèse des travaux effectués dans le domaine du calcul du WCET.*

Mots-clés : WCET, analyse statique, simulation, architecture des processeurs

Abstract

The main characteristic of hard real-time systems is that they must guarantee a correct timing behaviour. Each hard real-time task has a deadline to meet, otherwise the real-time system fails and the failure can have catastrophic consequences. Schedulability analysis methods are commonly used in hard real-time systems to check whether or not all tasks deadlines will be met. Most of them rely on the knowledge of an upper bound on the computation time of every task, named WCET, for Worst-Case Execution Time. This paper gives an overview of the methods used to compute WCETs.

Keywords: WCET, static analysis, simulation, processor architecture

Table des matières

1 Introduction **1**

2 Méthodes dynamiques de détermination du WCET **3**
2.1 Méthodes de mesure 4
2.2 Jeux de test explicites 4
2.3 Jeux de test symboliques 4

3 Obtention du WCET par analyse statique **5**
3.1 Étapes de l'analyse 5
3.2 Analyse de flot 5
3.2.1 Représentations du flot de contrôle 5
3.2.2 Blocs de base 5
3.2.3 Graphe de flot de contrôle. 5
3.2.4 Arbre syntaxique 6
3.2.5 Informations supplémentaires sur le flot de contrôle 7
3.2.6 Obtention des informations de flot de contrôle 7
3.2.7 Correspondances entre représentations : compilateurs optimisants 8
3.3 Calcul du WCET 8
3.3.1 Techniques utilisant l'algorithmique des graphes (*Path-based*) 8
3.3.2 Techniques *IPET* 8
3.3.3 Techniques basées sur l'arbre syntaxique (*Tree-based*) 9
3.3.4 Analyse symbolique de WCET 10

4 Analyse temporelle **11**
4.1 Analyse temporelle globale 11
4.1.1 Prise en compte des caches d'instructions 11
4.1.2 Prise en compte du mécanisme de prédiction de branchement 13
4.2 Analyse temporelle locale 13
4.2.1 Prise en compte de l'exécution pipelinée 14
4.2.2 Influence sur le WCET des blocs de base 14
4.2.3 Effet inter-bloc de base 14

5 Calcul du WCET et processeurs haute performance **15**

6 Conclusion **17**

Chapitre 1

Introduction

La particularité des tâches temps-réel est d'avoir à respecter des contraintes temporelles. On distingue les systèmes temps-réel *stricts*, pour lesquels le non respect de ces contraintes peut avoir des conséquences catastrophiques, des systèmes temps-réel *souples* pour lesquels les contraintes sont définies pour assurer une qualité de service mais peuvent exceptionnellement être violées sans que cela ne soit dramatique. Cet article traite plus particulièrement des systèmes temps-réel stricts.

Le temps d'exécution d'un programme dépend, en général, des valeurs des données en entrée du programme : ces valeurs déterminent un certain chemin d'exécution par le biais d'instructions de contrôle de flot. On appelle **temps d'exécution au pire cas** ou **WCET** (*Worst-Case Execution Time*) la valeur maximale de ce temps d'exécution pour l'ensemble des jeux de données en entrée possibles.

La connaissance du WCET d'une tâche peut être utile lors de la conception d'un système temps-réel, que ce soit au niveau du matériel, du système opératoire ou des applications :

- elle peut aider à dimensionner le matériel, qu'il s'agisse de déterminer si tel ou tel système est suffisamment puissant pour que les contraintes temps-réel d'une application puissent être garanties, ou bien d'estimer le nombre de tâches qu'un système donné peut supporter en assurant un respect de leurs échéances.
- dans le cadre du choix d'une politique d'ordonnancement pour un système temps-réel, l'analyse d'ordonnancement [9], réalisé hors-ligne, a pour but de vérifier que les échéances de toutes les tâches peuvent être satisfaites. Cette analyse est basée sur une estimation du WCET des différentes tâches.
- des informations sur le temps d'exécution de différentes parties d'un programme (boucles ou chemins critiques) peuvent être exploitées dans une démarche d'optimisation du code.

Le WCET calculé doit être supérieur (ou égal) au WCET réel, sinon il y a risque de violation de contraintes temps-réel, ce qui peut être fatal dans un contexte de temps-réel strict. Toutefois, pour être utile, il ne doit pas être trop surestimé, ce qui conduirait à un surdimensionnement inutile du système.

Les méthodes habituellement mises en œuvre pour calculer le WCET se divisent en deux grandes catégories :

- les méthodes **dynamiques** consistent à mesurer le temps d'exécution du programme considéré sur un système réel ou sur un simulateur. Ces mesures doivent être réalisées pour tous les jeux d'entrées possibles, ou alors il faut être capable de définir un jeu d'entrées dont on est certain qu'il conduit au temps d'exécution le plus long.
- les méthodes **statiques** sont fondées sur une analyse statique du programme dans le but de s'affranchir des jeux d'entrée. Elles comportent en général deux phases ; l'analyse de *haut niveau* détermine tous les chemins possibles, et l'analyse de *bas niveau* évalue le temps d'exécution de chacun de ces chemins.

Notons que les méthodes dynamiques permettent d'obtenir une valeur précise du WCET (si les mesures sont réalisées avec le jeu de test de pire cas) tandis que les méthodes statiques conduisent à poser des hypothèses conservatrices pour des informations qui ne peuvent pas être connues précisément lors de l'analyse et calculent alors une borne supérieure du WCET (c'est-à-dire que le temps d'exécution maximum réel ne peut pas être supérieur au WCET calculé).

Le paragraphe 2 sera consacré aux méthodes d'évaluation dynamiques. Nous présenterons l'analyse statique dans sa globalité au paragraphe 3, et examinerons plus en détail l'analyse de bas niveau dans le paragraphe 4. Nous discuterons dans le paragraphe 5 des (im-)possibilités des méthodes présentées à prendre en compte de manière correcte les mécanismes avancés présents dans les processeurs les plus récents. Nous conclurons par quelques réflexions sur les perspectives d'évolution du calcul de WCET.

Chapitre 2

Méthodes dynamiques de détermination du WCET

2.1 Méthodes de mesure

La méthode la plus directe consiste à exécuter le programme dont on souhaite déterminer le WCET sur le système matériel ciblé et de mesurer la durée de cette exécution. La mesure peut être réalisée par des équipements externes spécifiques (comme un analyseur logique) ou en exploitant des compteurs internes disponibles dans certains processeurs (le Pentium, par exemple).

Parfois, on ne dispose pas du système matériel et/ou des moyens de mesure adéquats. C'est le cas, par exemple, pendant la phase de conception du système si le matériel prévu n'est pas encore disponible ou si l'on souhaite comparer à moindre frais plusieurs matériels possibles. On peut alors avoir recours à un simulateur logiciel qui modélise un système matériel et calcule son évolution cycle par cycle. La principale difficulté réside dans la validation du simulateur qui doit être suffisamment précis et fiable, ce qui induit un coût de développement non négligeable. Actuellement, un certain nombre de démarches sont menées dans le milieu de l'architecture des machines en vue de définir des plate-formes efficaces pour un développement rapide et fiable de simulateurs.

2.2 Jeux de test explicites

Quel que soit le milieu de mesure (système matériel ou simulateur), il doit être alimenté par un code exécutable et par un jeu de données en entrée de ce programme. Pour mesurer le temps d'exécution au pire cas, il faut fournir le jeu de test qui conduit au temps d'exécution maximum. Se pose alors le problème de la définition de ce jeu de test de pire cas. Plusieurs solutions sont envisageables :

- mesurer le temps d'exécution du programme pour *tous* les jeux d'entrées possibles : la durée du processus (temps de génération de tous les jeux de test, puis temps de mesure pour chacun d'entre eux) est, en général, rédhibitoire. Cependant, cette solution peut être retenue dans le cas de programmes simples, admettant peu de données en entrée, et pour lesquels le domaine de variation des entrées est limité.
- laisser le soin à l'utilisateur de définir le jeu de test pire cas : sa parfaite connaissance du programme peut lui permettre d'identifier ce jeu de test sans erreur. Là encore, cette solution est sans doute limitée à des applications très simples (qui ne sont pas forcément rares).
- générer automatiquement un jeu de test (ou un ensemble de jeux de test) qui doit conduire au temps d'exécution maximum. Quelques travaux basés sur des algorithmes évolutionnistes [46] ou sur l'algorithme du recuit simulé [44] ont été menés dans ce sens, mais les résultats ne sont pas complètement sûrs dans la mesure où le jeu de test généré n'est pas garanti être celui du pire cas L'intérêt de ces méthodes n'est toutefois pas négligeable puisqu'elles permettent de donner une limite inférieure au WCET et de compléter ainsi les calculs statiques qui, eux, en donnent une limite supérieure.

2.3 Jeux de test symboliques

Si l'on ne sait pas générer de jeu de test assurant un temps d'exécution de pire cas, une autre stratégie consiste à utiliser un jeu de test symbolique. L'idée sous-jacente est que, en réalité, ce qui nous intéresse est de mesurer les temps d'exécution de tous les chemins d'exécution possibles du programme (l'énumération de jeux d'entrée ne servant qu'à alimenter le programme de manière à parcourir tous ces chemins).

Le principe de cette méthode est de poser l'hypothèse que les données en entrée sont inconnues et d'étendre le jeu d'instruction du processeur cible de sorte qu'il puisse réaliser des calculs à partir d'un ou plusieurs opérandes de valeur *inconnue* (par exemple, la somme d'un opérande de valeur connue et d'un opérande de valeur inconnue donne un résultat de valeur inconnue). Lorsqu'un branchement conditionnel est exécuté avec une condition de valeur inconnue, les deux chemins doivent être explorés.

Une telle extension ne peut être réalisée que dans le cadre d'un simulateur logiciel. Une solution permettant de réaliser une exécution symbolique par simulation est présentée dans [32].

Chapitre 3

Obtention du WCET par analyse statique

3.1 Étapes de l'analyse

Les méthodes d'obtention du WCET par analyse statique opèrent par analyse de la structure des programmes, au niveau de leur code source et/ou objet. Elles peuvent être séparées en plusieurs composants logiques :

- l'*analyse de flot* qui, à partir du code source et/ou objet des programmes, détermine les chemins d'exécution possibles dans le programme ;
- l'analyse de *bas niveau* qui évalue l'impact de l'architecture matérielle sur le pire temps d'exécution du programme ;
- le *calcul* qui détermine le WCET à partir des résultats des autres analyses.

Nous nous concentrons dans ce paragraphe sur l'analyse de flot et le calcul de WCET. Les paragraphes 4 et 5 détailleront l'analyse de bas niveau.

3.2 Analyse de flot

Nous développons ici les représentations du flot de contrôle les plus souvent utilisées par les analyseurs statiques de WCET (§ 3.2.1), puis la manière dont elles sont obtenues (§ 3.2.6).

3.2.1 Représentations du flot de contrôle

3.2.2 Blocs de base

L'analyse du code objet des programmes est nécessaire pour l'analyse de bas niveau, car c'est à ce niveau que sont accessibles les informations sur la durée d'exécution des instructions. 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.

L'exemple de la figure 3.1 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 comptent onze blocs de base notés BB_1 à BB_{11}. Ils nous serviront à illustrer nos propos dans la suite de l'article.

Les blocs de base BB_1 à BB_{11} ainsi obtenus sont les éléments de base des deux représentations décrites dans les paragraphes suivants.

3.2.3 Graphe de flot de contrôle.

Un *graphe de flot de contrôle* décrit tous les enchaînements possibles entre blocs de base. La forme la plus utilisée de graphes de flot de contrôle (figure 3.2) est celle des graphes dont les nœuds sont des blocs de base et où les arcs représentent les relations de précédence entre blocs. On peut alors distinguer deux types d'arcs : 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" qui

```
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. 3.1 – *Code source d'un programme analysé (langage C)*

représentent les branchements pris (*e.g.* arc de BB_{10} à BB_6). Cette catégorie de graphe de flot de contrôle est utilisée dans de nombreux travaux d'analyse statique basés sur les graphes [27, 37, 43].

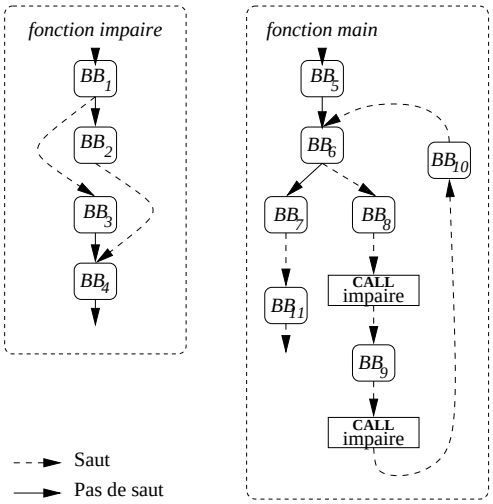


FIG. 3.2 – *Graphes de flot de contrôle du programme analysé*

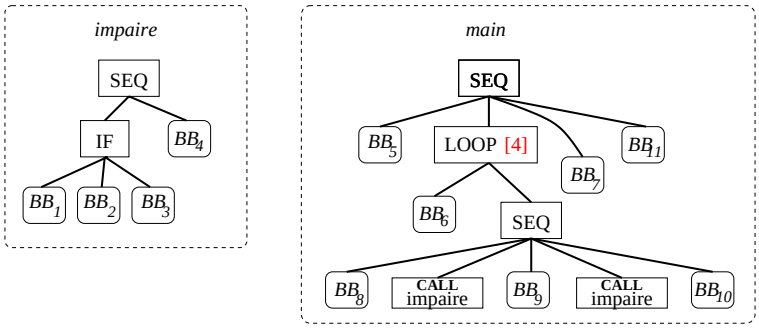


FIG. 3.3 – *Arbres syntaxiques des fonctions impaire et main*

3.2.4 Arbre syntaxique

Un *arbre syntaxique* (*cf.* figure 3.3) 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 ci-dessus 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.
– 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.

L'arbre syntaxique est plus riche en informations que le graphe de flot de contrôle. En pratique, les deux représentations sont le plus souvent utilisées conjointement, ce qui impose quelques restrictions sur le langage source (pas de `GOTO` et assimilés) afin d'avoir une correspondance directe entre les deux représentations.

3.2.5 Informations supplémentaires sur le flot de contrôle

Les représentations en blocs de base, graphes de flot de contrôle et arbre syntaxique ne sont pas suffisantes pour le calcul sûr et précis du WCET. En particulier, rien n'indique dans ces représentations que les chemins d'exécution sont de taille finie. Ces représentations doivent être complétées par des informations sur le comportement dynamique du code à analyser, de manière à 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 d'itérations des boucles. En effet, 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, et elle l'est le plus souvent sous forme d'une constante [40, 17]. Ainsi, la constante [4] dans la figure 3.3 représente le pire nombre d'itérations de la boucle présente dans la fonction *main*.
– Contraindre le choix d'une branche dans une structure conditionnelle. Ceci est utile 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 (chemins apparaissant dans le graphe de flot de contrôle du programme mais qui ne feront jamais partie d'une exécution réelle) ; par exemple, deux branches qui s'excluent mutuellement.

3.2.6 Obtention des informations de flot de contrôle

Le graphe de flot de contrôle est le plus souvent construit par analyse statique de code de bas niveau (code assembleur ou bytecode [4]), et en utilisant soit un compilateur modifié [33], soit un outil dédié à la manipulation de code de bas niveau [6]. P. Puschner envisage aussi dans [38] l'obtention du graphe de flot de contrôle à partir du code de haut niveau du programme. La représentation en arbre syntaxique, pour sa part, est obtenue par analyse statique d'un langage de haut niveau.

En ce qui concerne les informations supplémentaires sur le flot de contrôle, que nous avons énumérées dans le paragraphe 3.2.5, on peut distinguer deux possibilités pour les obtenir. 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 [40, 10, 36], ou interactivement [24]. La plupart des travaux concernant l'analyse statique de WCET utilise ce type de méthodes.

La deuxième possibilité est de dériver ces informations automatiquement. Cette deuxième classe de méthodes ne permet pas d'obtenir ces informations de manière systématique. En effet, certains programmes ne se terminent pas, par exemple lorsqu'ils comportent des boucles infinies ; pour d'autres, le nombre maximum d'itérations des boucles peut aussi dépendre d'éléments extérieurs au programme, impossibles à identifier statiquement. Toutefois, dans le cas où ces méthodes sont utilisables, elles permettent d'éliminer les erreurs humaines dans le processus de mise en place des annotations. Les travaux décrits dans [21] utilisent l'analyse de flot de données pour identifier les nombres d'itérations des boucles ainsi que les chemins infaisables à l'intérieur de celles-ci. Ferdinand et al. [19] utilisent une méthode d'interprétation abstraite sur le contenu des registres du processeur pour identifier les chemins d'exécution infaisables. Ermedahl et Gustafsson [17] utilisent une méthode d'interprétation abstraite sur le code source pour obtenir l'ensemble des informations de flot de contrôle.

3.2.7 Correspondances entre représentations : compilateurs optimisants

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 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 [45] mais effectue également des optimisations.

J. Engblom et al. ont proposé dans [16] 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 spécification par un langage dédié nommé ODL (pour *Optimization Description Language*) des optimisations pouvant être réalisées par le compilateur. Les informations de haut niveau (annotations de boucle par exemple) sont alors transformées en parallèle avec l'optimisation du code, et ce avec seulement de très faibles modifications du compilateur (obtention de la trace des transformations). Une autre méthode, proposée par R. Kirner et P. Puschner dans [23] consiste à transformer les informations de flot de contrôle à l'intérieur même du compilateur, par introduction d'instructions dans le code intermédiaire (RTL - *Register Transfer Language*) généré par le compilateur. Ces instructions sont transformées en même temps que sont effectuées les optimisations de compilation.

3.3 Calcul du WCET

Une fois les informations de flot de contrôle obtenues, il faut chercher le pire chemin d'exécution à partir de leur représentation. La méthode utilisée pour la recherche du plus long chemin permet de distinguer les différentes méthodes d'analyse statique de WCET. Pour simplifier, on suppose ici que le temps d'exécution de chaque instruction (et par conséquent de chaque bloc de base) est connu et constant, quel que soit son contexte d'exécution. Cette hypothèse simplificatrice, qui est à la base des premiers travaux du domaine, nous permet de ne pas nous préoccuper pour l'instant du niveau bas de l'analyse statique de WCET. Cette hypothèse sera levée au paragraphe 4.

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

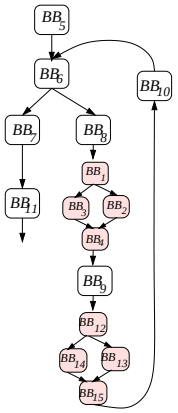
La connaissance des WCET individuels des blocs de base permet d'associer des WCET aux nœuds du graphe de flot de contrôle. On obtient alors un graphe valué avec un seul point d'entrée et un seul point de sortie. On peut chercher le pire chemin d'exécution dans le graphe de flot de contrôle [20, 43] en utilisant les algorithmes traditionnels de l'algorithmique des graphes 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.

3.3.2 Techniques *IPET*

Cette classe de méthodes, dite *d'énumération implicite des chemins* (ou *IPET - Implicit Path Enumeration Technique*) est utilisée dans de nombreux travaux d'analyse statique de WCET [18, 26, 35, 41]. 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 supplémentaires sur le flot de contrôle, vues au paragraphe 3.2.5.

La figure 3.4 montre le système de contraintes généré pour l'exemple de la figure 3.2. Chaque nœud du graphe de flot de contrôle 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 nœuds prédécesseurs doit être égale à celle des nombres d'occurrences des nœuds successeurs. On obtient ainsi un premier système de contraintes qui décrit la structure du graphe (contraintes (a) sur la

figure). Le deuxième système de contraintes (contraintes (b) sur la figure) regroupe par exemple les contraintes sur les nombres maximum d'itérations des boucles.



(a) Description du graphe

$$
\begin{aligned}
n_5 + n_{10} &= n_6 \\
n_6 &= n_7 + n_8 \\
n_7 &= n_{11} \\
n_8 &= n_1 \\
n_1 &= n_2 + n_3 \\
n_2 + n_3 &= n_4 \\
n_4 &= n_9 \\
n_9 &= n_{12} \\
n_{12} &= n_{13} + n_{14} \\
n_{13} + n_{14} &= n_{15} \\
n_{15} &= n_{10}
\end{aligned}
$$

(b) Contraintes du graphe

$$
\begin{aligned}
n_5 &= 1 \\
n_6 &\leq 4 \times n_5
\end{aligned}
$$

FIG. 3.4 – *Traduction d'un graphe de flot de contrôle en un système de contraintes*

Étant donnés 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 BB_i. Les valeurs des w_i sont fournies par l'analyse de bas niveau décrite au paragraphe 4.

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 (§ 3.2.4). Les méthodes IPET permettent donc d'analyser plus de programmes 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 nœuds a_α et a_β). Cette possibilité est exploitée par exemple dans [26].

La maximisation de l'expression du WCET ci-dessus fournit les valeurs des n_i (et évidemment 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 temps d'analyse est fonction de la complexité du système et peut donc être important [29].

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

Cette classe de méthodes, proposée initialement par P. Puschner et C. Koza dans [40], s'appuie sur la représentation du programme en arbre syntaxique pour calculer récursivement son WCET. Un ensemble de formules 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) 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.

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 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 3.2.5, sont prises en compte par des 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.

On obtient ainsi un arbre temporel (*timing tree*) [38] qui contient les WCET calculés aux différents nœuds de l'arbre.

3.3.4 Analyse symbolique de WCET

De nombreuses méthodes d'obtention de WCET par analyse statique calculent les WCET sous la forme d'une constante, indépendante des entrées du programme analysé. Cette approche, bien que sûre, peut se révéler très pessimiste, dans les cas où le WCET d'un code est dépendant d'un certain *contexte*. Par exemple, une fonction peut avoir des temps d'exécution différents selon ses paramètres d'entrées [4], la valeur de certaines variables locales (indices de boucles englobantes dans le cas de boucles non rectangulaires), la configuration globale du système (par exemple dans [12], il a été observé lors de l'analyse statique de WCET du noyau de système RTEMS que le WCET de certains appels dépendent du nombre de tâches), ou encore le type de matériel ciblé. Cette variation au contexte a entraîné un certain nombre d'études sur des représentations *symboliques* des WCET [4]. Un WCET, au lieu d'être exprimé par une constante, est une expression, qui ne sera évaluée que lorsque toutes les variables la composant seront connues. Une telle approche est utilisée notamment dans [39, 13] au sein de méthodes d'analyse statique de WCET à base d'arbre.

Chapitre 4

Analyse temporelle

Une technique d'analyse statique de WCET "naïve" repose sur deux hypothèses: (i) elle 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, (ii) elle 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 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 (*e.g.* les caches et pipelines). Ainsi, le pipeline, en introduisant du parallélisme dans le traitement d'une suite d'instructions, remet en cause la premiere hypothèse. De même, le cache d'instructions introduit une variation du temps de traitement d'une instruction en fonction du contexte, ce qui contrarie la deuxième hypothèse.

La modélisation du comportement de ces éléments d'architecture n'est pas triviale, et leur prise en compte introduit une dépendance au contexte. Cependant, 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 [1, 8, 28, 30, 2] et les pipelines [5, 47].

On peut classer les effets de l'architecture matérielle sur le temps d'exécution en deux catégories: les effets *locaux* et les effets *globaux*. Un élément d'architecture a un effet local lorsque, par l'entremise de cet élément, le comportement temporel d'une instruction ne peut affecter le comportement d'une autre instruction que si celle-ci est "proche" dans le flot d'instruction. C'est typiquement le cas du pipeline pour lequel le comportement temporel d'une instruction n'affecte que les quelques instructions suivantes. On dit d'un élément d'architecture qu'il a un effet global s'il permet à une instruction d'affecter le comportement d'autres instructions quelles que soient leurs distances dans le flot d'exécution. C'est par exemple le cas du cache d'instructions pour lequel le chargement d'une instruction peut causer le remplacement d'une autre instruction et ainsi affecter le temps d'exécution de cette dernière, même si celle-ci n'est exécutée que bien plus tard.

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 classées selon les deux catégories précédentes et pour les différentes méthodes de calculs de WCET présentées au paragraphe 3.3.

4.1 Analyse temporelle globale

Parmi les éléments d'architecture les plus souvent pris en compte pour l'analyse de WCET, les caches (d'instructions principalement) et le mécanisme de prédiction des branchements ont un effet global.

4.1.1 Prise en compte des caches d'instructions

L'usage d'une mémoire cache introduit de l'indéterminisme dans l'architecture. Il faut pouvoir estimer de façon *sûre* (et pas seulement de manière probabiliste) le résultat (succès/échec) 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, seul les accès mémoire pour lesquels on est certain que l'accès au cache sera un succès sont classés comme tel (s'il y a un doute, on considére l'estimation la plus pessimiste : échec).

Plusieurs méthodes ont pour but d'intégrer le comportement du cache d'instructions dans le calcul du WCET. 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.

Analyse du comportement 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 [2, 34] 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.

Adaptation de la méthode de calcul *tree-based*

Pour permettre la prise en compte du cache d'instructions décrite dans le paragraphe précédent, l'ensemble des formules de calcul récursif du WCET (*cf.* § 3.3.3). doit être modifié. Cette modification, exposée dans [22], permet de ne plus manipuler directement le WCET des structures de contrôle 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. Ils comportent entre autre deux ensembles. Le premier 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.

Analyse du comportement du cache par *IPET*

Une autre 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 [27, 29, 35]. 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 possibles (cas d'un échec ou d'un succès) d'exécution de chaque instruction.

Adaptation de la méthode de calcul *IPET*

Pour prendre en compte l'effet du cache d'instructions, l'expression du WCET à maximiser (originalement $\sum_i n_i \times w_i$, *cf.* § 3.3.2) est adaptée pour refléter (i) le fractionnement des blocs de base dans le cache et (ii) les deux temps d'exécution possibles de ces fragments de blocs de base. Dans [27], un bloc de base i donne N_i fragments correspondants aux lignes de cache occupées par ce fragment. à chaque fragment j d'un bloc de base i sont associés deux WCET, $w_{i,j}^{succes}$ et $w_{i,j}^{echec}$, correspondant respectivement aux cas de succès et d'échec. Et le nombre d'occurrences du bloc de base (originalement n_i) est partagé : $n_i = n_{i,j}^{succes} + n_{i,j}^{echec}$. La nouvelle expression du WCET à maximiser est alors :

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

Un nouveau jeu de contraintes est généré et ajouté au système existant pour le calcul des valeurs des $n_{i,j}^{succes}$ et $n_{i,j}^{echec}$. 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 *échec* lors de sa première exécution puis qui reste toujours dans le cache est représenté par : $n_{i,j}^{echec} \leq 1$.

Analyse du comportement du cache par interprétation abstraite

Enfin, la méthode présentée dans [1, 18], basée sur l'analyse de programme par interprétation abstraite [14], distingue les mêmes catégories que la méthode de simulation statique de cache (*cf.* 4.1.1). Elle a la particularité de différencier 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 mémoire considérés (et donc des instructions).

4.1.2 Prise en compte du mécanisme de prédiction de branchement

Cet élément d'architecture permet d'éviter une part importante des pénalités temporelles liées aux instructions de branchement, et plus particulièrement à leur effet sur l'exécution pipelinée (*cf.* § 4.2.1). Le nombre d'étages des pipelines des processeurs actuels ne cesse d'augmenter (*e.gpar exemple* 5 pour l'Intel Pentium et 20 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 charger 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. Pour éviter ces passages à vide dans le pipeline, coûteux en nombre de cycles, la plupart des 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. 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. Lorsque le processeur exécute un branchement, le résultat de son exécution est enregistré. Puis, lorsqu'il est de nouveau rencontré dans le flot d'instructions chargées, et si les informations sur sa ou ses exécutions passées sont connues, 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.

Simulation statique de prédiction de branchement. Cette méthode présentée dans [11] distingue quatre catégories de branchements en fonction de la méthode utilisée pour établir la prédiction. On distingue deux méthodes de prédiction. Un branchement peut être **H-prédit** si l'historique de ses précédentes exécutions est connu lors de sa prédiction. La prédiction est alors calculée en fonction de celui-ci. Ou bien il peut être **D-prédit** si c'est la première fois qu'il est rencontré ou que son historique n'est pas connu car il a été remplacé par celui d'un autre branchement. Dans ce deuxième cas, la prédiction est une prédiction par défaut. Les branchements sont classés en quatre catégories : (i) *toujours D-prédit* si l'historique du branchement n'est jamais connu lors de la prédiction, (ii) *d'abord D-prédit* si l'historique du branchement est inconnu la première fois, et connu pour les prédictions suivantes, (iii) *d'abord inconnu* si on ne sait pas si l'historique est connu ou non la première fois, mais qu'on est sûr de sa présence pour les prédictions suivantes, et (iv) *toujours inconnu* sinon (c'est le cas le plus pessimiste). Une fois chaque branchement classé dans une de ces catégories, il est possible de déterminer si, dans le pire cas d'exécution, le mécanisme de prédiction se trompera. Il faut pour cela connaître le rôle des branchements ainsi que le schéma de compilation utilisé. Par exemple, un branchement qui a pour rôle d'être un test de sortie de boucle et dont la catégorie est *toujours H-predicted* provoquera une erreur de prédiction à chaque fin de boucle.

4.2 Analyse temporelle locale

Le pipeline est un élément d'architecture dont les effets sont locaux. Si on considère une séquence d'instructions exécutées dans un pipeline, l'effet d'une instruction porte sur les quelques instructions suivantes et devient nul au bout d'un certain temps. Nous examinons ici les méthodes de prise en compte de l'exécution pipelinée dans le cas mono-pipeline et pour les processeurs qui exécutent les instructions dans l'ordre de leur apparition dans le programme.

4.2.1 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'introduire 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. 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 le traitement d'une instruction peut débuter avant la fin de celle de l'instruction précédente.

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 pour la prise en compte des aléas de contrôle.

4.2.2 Influence sur le WCET des blocs de base

Aucun aléa de contrôle ne peut avoir lieu pendant l'exécution d'un bloc de base 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 [42] repose sur la représentation de l'occupation du pipeline par des tables de réservation [25] pendant l'exécution de ces blocs.

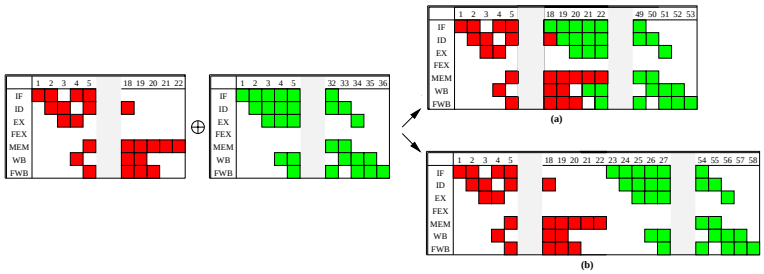


FIG. 4.1 – *Composition de descripteurs de pipeline*

4.2.3 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 4.1). Un descripteur peut donc se limiter aux extrémités du descripteur d'occupation du pipeline. 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 d'une architecture dotée d'un 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 sont exécutés en séquence sans que le branchement les reliant ne soit mal prédit, 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 4.1.a) et se recouvrent tout en respectant les contraintes d'occupation des différents étages du pipeline. Dans le cas contraire, 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 4.1.b).

Chapitre 5

Calcul du WCET et processeurs haute performance

L'architecture des processeurs embarqués a tendance, avec quelques années de retard, à suivre celle des processeurs haute performance. De fait, la prédiction de branchement, qui existe depuis longtemps dans les processeurs haute performance a été mise en œuvre il y a quelques années seulement dans les processeurs embarqués et sa prise en compte dans des mécanismes de détermination de WCET est récente. Ainsi, il semble important de chercher à prendre en compte dans le calcul du WCET les mécanismes avancés présents dans les processeurs haute performance de manière à disposer des modèles correspondants dès leur intégration dans les processeurs embarqués.

Bien évidemment, les mécanismes les plus récemment mis en œuvre dans les processeurs haute performance n'ont pas été intégrés dans les analyses de WCET. Ainsi, on peut citer le cache de traces présent dès les premières versions du Pentium 4 ou encore l'exécution multiflot introduite dans sa dernière version. Si le cache de traces ressemble, dans son principe, à un cache d'instructions, l'exécution multiflot ajoute un degré d'indéterminisme du fait qu'elle effectue, dynamiquement et à chaque cycle, de multiples choix pour déterminer le flot qui va être servi. Plus surprenant, le mécanisme de mémoire virtuelle intégré dans tous les processeurs modernes et dans certains processeurs embarqués n'a, à notre connaissance, pas été étudié. Il est vrai que son comportement est proche de celui d'un cache et ne présente donc sans doute pas, d'un point de vue théorique, de problèmes particuliers. Par ailleurs, ce mécanisme est souvent désactivé dans les systèmes temps-réels stricts. En ce qui concerne les caches qui ont été abondamment étudiés, la politique de remplacement est peu abordée. Une politique LRU (moins récemment utilisé) est prise comme hyptohèse. En pratique, certains processeurs ont un politique pseudo-LRU qui ne devrait pas poser de problème particulier de modélisation mais aussi quelquefois une politique de remplacement aléatoire peu compatible avec une analyse statique. Enfin peu d'articles ont traité le cas de l'exécution non ordonnée, mécanisme dont la nature hautement dynamique est certainement difficile à prendre en compte de manière purement statique. Pour ce faire, [7] utilise une approche de simulation à l'aide de réseaux de Petri colorés.

Certains mécanismes, en particulier la prédiction de branchement (voir §4.1.2) commencent à être étudiés mais sur des cas particuliers simples. Or, dans la plupart des processeurs haute performance, la prédiction de la direction d'un branchement peut être influencée par le comportement d'un ou plusieurs autres branchements. Par ailleurs, certains prédicteurs de branchement sont mis à jour de façon spéculative. Ainsi, l'état du prédicteur est modifié par le mauvais chemin qui est chargé dans le processeur après une mauvaise prédiction de branchement. Le plus souvent, ces modifications ne sont pas corrigées (ou pas totalement) lorsque le processeur détecte l'erreur de prédiction car leur influence est faible sur la performance. Toutefois, l'impact de ces modifications intempestives devrait être pris en compte pour une estimation sûre du WCET.

La plupart des études se focalisent sur un mécanisme matériel et très peu portent sur les interactions entre mécanismes. Or, Lundqvist a montré qu'un échec dans le cache pouvait amener à un temps d'exécution plus court pour peu que le processeur ait une exécution non ordonnée [31]. Ainsi, ce qui semblait acquis comme un cas pire jusqu'à présent se révèle ne pas l'être forcément du fait des répercussions d'un mécanisme sur l'autre. On peut multiplier les exemples. Dans le cas de la prédiction de branchement, on considère que la mauvaise prédiction est forcément le pire cas puisqu'elle entraîne une purge du pipeline et elle est prise en compte en ajoutant simplement une

pénalité de durée fixe. Cette approche est exacte si l'on considère la prédiction de branchement en elle-même mais c'est oublier l'exécution spéculative. Lorsqu'une mauvaise prédiction est effectuée, les instructions situées sur le mauvais chemin sont chargées et, si l'exécution est non ordonnée, peuvent être exécutées. Elles peuvent donc avoir une influence, par exemple sur le contenu des caches d'instructions et de données en éjectant un bloc du cache utilisé par la suite et compté comme présent par une analyse de WCET ignorant l'exécution spéculative. De même, les instructions du mauvais chemin ne sont pas éliminées en un cycle du processeur. Souvent, elles poursuivent leur exécution dans le pipeline sans toutefois mettre pas à jour les registres. Ainsi, une instruction du mauvais chemin et à latence longue peut avoir retarder les instructions du bon chemin. Ignorer l'exécution spéculative du mauvais chemin peut donc mener à un WCET sous-estimé. Pour la prendre en compte lors d'une analyse statique, il faut certainement envisager une interaction plus forte entre analyse de flot et analyse temporelle.

Chapitre 6

Conclusion

Les processeurs intègrent des mécanismes de plus en plus complexes. Ceci pose un certain nombre de problèmes si l'on souhaite évaluer le WCET de manière purement statique :

- les constructeurs ne diffusent pas toujours des informations suffisamment détaillées ;
- même si l'on dispose d'une documentation suffisante, la modélisation de ces mécanismes requiert la mise en œuvre d'outils théoriques de plus en plus évolués (on est passé d'une simple addition de temps d'exécution élémentaires à la construction de modèles beaucoup plus complexes basés, par exemple, sur des réseaux de Petri colorés) ;
- il est alors de plus en plus difficile de valider la conformité du modèle avec la réalité [3][15].

Aussi, la conjonction de méthodes statiques (pour tout ce qui relève de l'analyse de chemins) et de mesures dynamiques (capables de déterminer de manière sûre les temps d'exécution de portions de code) semble inéluctable.

Bibliographie

[1] 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, septembre 1996.

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *15th IEEE Real-Time Systems Symposium (RTSS94)*, pages 172–181, December 1994.

[3] P. Atanassov, R. Kirner, and P. Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *IEEE/IEE Real-Time Embedded Systems Workshop*, London, UK, décembre 2001.

[4] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable WCET analysis framework. In *7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, December 2000.

[5] S. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical Report TR94-072, University of North Carolina, April 1995.

[6] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *6th Workshop on Compilers for Parallel Computers*, December 1996.

[7] F. Burns, A. Koelmans, and A. Yakovlev. Wcet analysis of superscalar processors using simulation with coloured petri nets. *Real-Time Systems*, 18(2-3):275–288, mai 2000.

[8] 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 *Real-Time technology and Applications Symposium*, pages 204–212, June 1996.

[9] G. C. Buttazzo, editor. *Hard real-time computing systems - predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[10] 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.

[11] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, mai 2000.

[12] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, June 2001.

[13] Antoine Colin and Guillem Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[14] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, 1977.

[15] J. Engblom. On hardware and hardware models for embedded real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, London, UK, décembre 2001.

[16] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution time analysis for optimized code. In *10th Euromicro Conference on Real-Time Systems*, Berlin, Germany, June 1998.

[17] 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.

[18] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler technique to cache behavior prediction. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 37–46, June 1997.

[19] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for real-life processor. In *1st international workshop on embedded software*, volume 2211 of *Lecture Notes in Computer Sciences*, pages 469–485. Springer-Verlag, octobre 2001.

[20] 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.

[21] 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, mai 2000.

[22] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Real-Time technology and Applications Symposium*, pages 230–240, June 1996.

[23] R. Kirner and P. Puschner. Transformation of path information for wcet analysis during compilation. In *13th Euromicro Conference on Real-Time Systems*, pages 29–36, Delft, The Netherlands, June 2001.

[24] 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.

[25] P. Kogge. *The Architecture of Pipelined Computers*. McGraw Hill Book Company, New York, NY, 1981.

[26] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, November 1995.

[27] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *16th IEEE Real-Time Systems Symposium (RTSS95)*, pages 298–307, Pisa, Italy, December 1995.

[28] 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.

[29] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, December 1996.

[30] 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. In *IEEE Workshop on Architectures for Real-Time Applications*, avril 1994.

[31] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[32] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, novembre 1999.

[33] D. Macos and F. Mueller. Integrating gnat/gcc into a timing analysis environment. In *10th Euromicro Conference on Real-Time Systems*, pages 15–18, June 1998.

[34] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, mai 2000.

[35] 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.

[36] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[37] 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 *6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[38] P. Puschner. A tool for high-level language analysis of worst-case execution times. In *10th Euromicro Conference on Real-Time Systems*, pages 130–137, Berlin, June 1998.

[39] P. Puschner and G. Bernat. Wcet analysis of reusable portable code. In *13th Euromicro Conference on Real-Time Systems*, pages 45–52, Delft, The Netherlands, June 2001.

[40] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[41] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat Wien, avril 1995.

[42] 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 *11th Workshop on Real-Time Operanting Systems and Software*, pages 59–62, mai 1994.

[43] 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.

[44] N. Tracey, J. Clark, and K. Mander. The way forward for unifying test case generation: The optimisation-based approach. In *IFIP Workshop on Dependable Computing and Its Applications*, pages 73–81, janvier 1998.

[45] A. Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Institut für Technische Informatik, Technische Universität Wien, Austria, avril 1994.

[46] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):239–264, novembre 2001.

[47] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, octobre 1993.